Renesas RA Family

Developing with RA8 Dual Core MCU

Introduction

This application project highlights the performance advantages of the RA8 dual-core MCUs, which includes the Cortex-M85 (CM85) core operating at 1 GHz and the Cortex-M33(CM33) core at 250 MHz. It discusses the dual-core architecture and provides use cases that illustrate how to effectively partition tasks between the two cores. Additionally, it addresses development and debugging processes specific to dual-core systems. This application note outlines the creation of applications that enhance performance with Renesas RA8 dual-core MCUs, utilizing both the CM85 core with Helium™ and the CM33 core at the same time.

It provides guidance on the necessary steps to create an application for the RA8 dual-core MCU, including.

- Application highlights
- Use cases block diagram
- Tool configuration
- Example projects confirmation
- Steps to migrate application projects to a new RA8 dual-core MCU

Required Resources

The following resources are referenced throughout this application note.

Development Tools and Software

- e² studio version: 2025-10 (25.10.0)
- LLVM Embedded Toolchain for Arm v18.1.3
- Renesas Flexible Software Package (FSP) v6.2.0 or later.

Target Devices

Below are the Renesas MCU products to which the information within this document is applicable:

- RA8P1.
- RA8D2.
- RA8M2.
- RA8T2.

Hardware

- EK-RA8P1, Evaluation Kit for RA8P1 MCU Group (http://www.renesas.com/ek-ra8p1).
- EK-RA8D2 (Optional), Evaluation Kit for RA8D2 MCU Group (http://www.renesas.com/ek-ra8d2).
- The description in the application note uses PC running Windows® 10 OS as an example. Refer to the corresponding user manual for the Development Tools and Software for a complete list of Operating Systems supported.
- One USB device cable (type-C) is used to connect the Evaluation Kit and the PC.

Contents

1. Example Application Overview

The example application projects included in this document outline the fundamental procedures for developing an application on the RA8 dual-core MCU using Renesas FSP, specifically focusing on the RA8P1 MCU. The procedures are also applicable to other RA8 dual-core MCUs. They demonstrate methods for partitioning tasks between CPU cores by utilizing the Inter-Processor Communication (IPC) module, sharing memory and resources to enhance performance on dual-core MCUs. They also illustrate the uses of tightly coupled memory (TCM) and cache in a dual-core MCU.

2. RA8 Dual Core MCU

The RA8 dual-core is an asymmetric architecture with a High-performance 1 GHz Arm® Cortex®-M85 core and 250 MHz Arm® Cortex®-M33 core, enabling the concurrent execution of multiple tasks by leveraging both cores with the following key features.

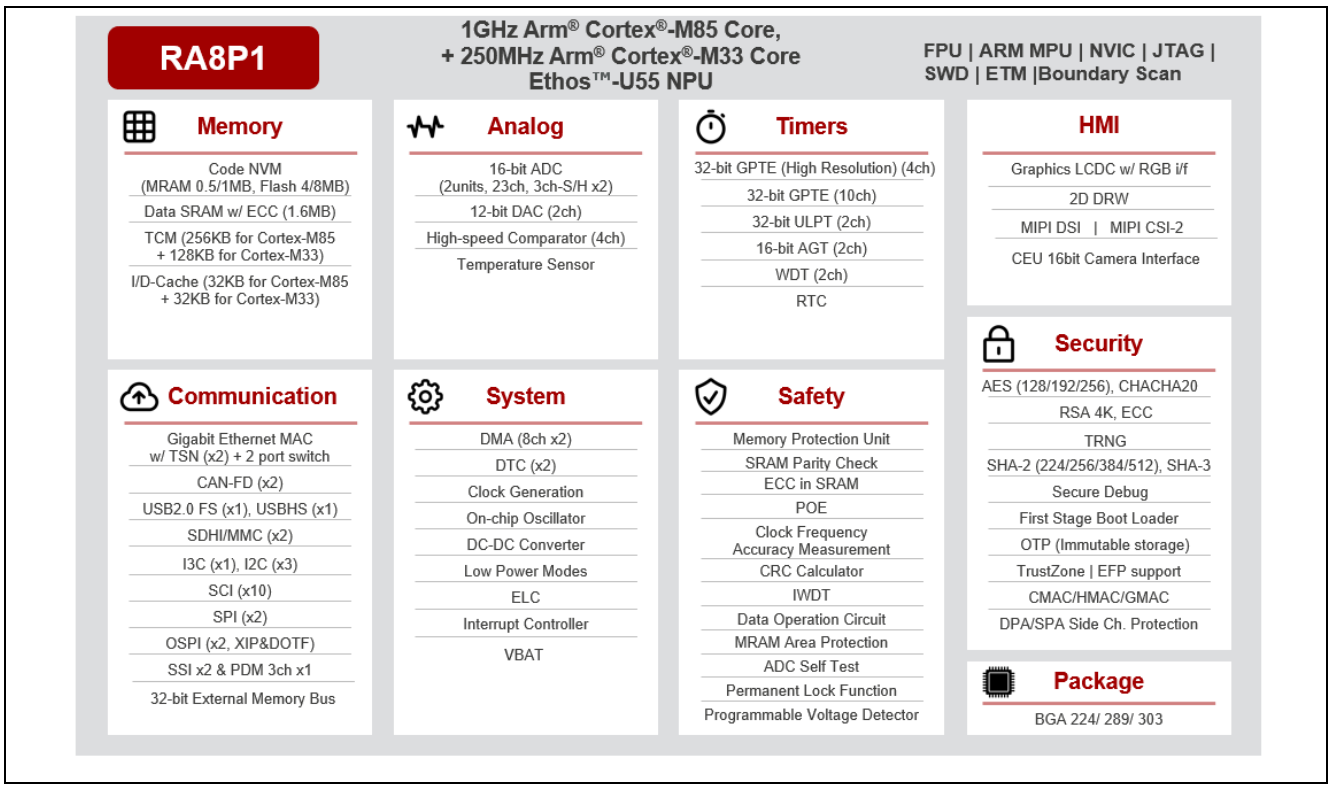

Figure 1. Example of Key Features in RA8P1 MCU

3. Create RA8 Dual Core Application with Renesas e²studio

When creating a project, you need to select the project type, provide a name and location, and configure the project settings. The primary settings include the FSP version, target board, toolchain version, and debugger, as shown in Figure 3. This section provides step-by-step instructions for creating a dual-core project based on the **Bare Metal-Blinky** project. An FSP solution project includes a solution project along with separate projects for CPU0 and CPU1. The solution project establishes the overall memory and clock configuration for both CPUs, allowing users to define and configure memory partitions for all projects within the solution.

3.1 Create A Solution Project for RA8P1 Dual Core MCU

For RA8P1 MCU applications, follow these steps to create a multicore project on e² studio. This section is based on the **Bare Metal-Blinky** project template. Another option available when creating a project is the Bare Metal-Minimal template.

Launch e² studio, click **File** > **New** > **Renesas C/C++ Project** > **Renesas RA,** and select **Renesas RA FSP Solution** > **Next**.

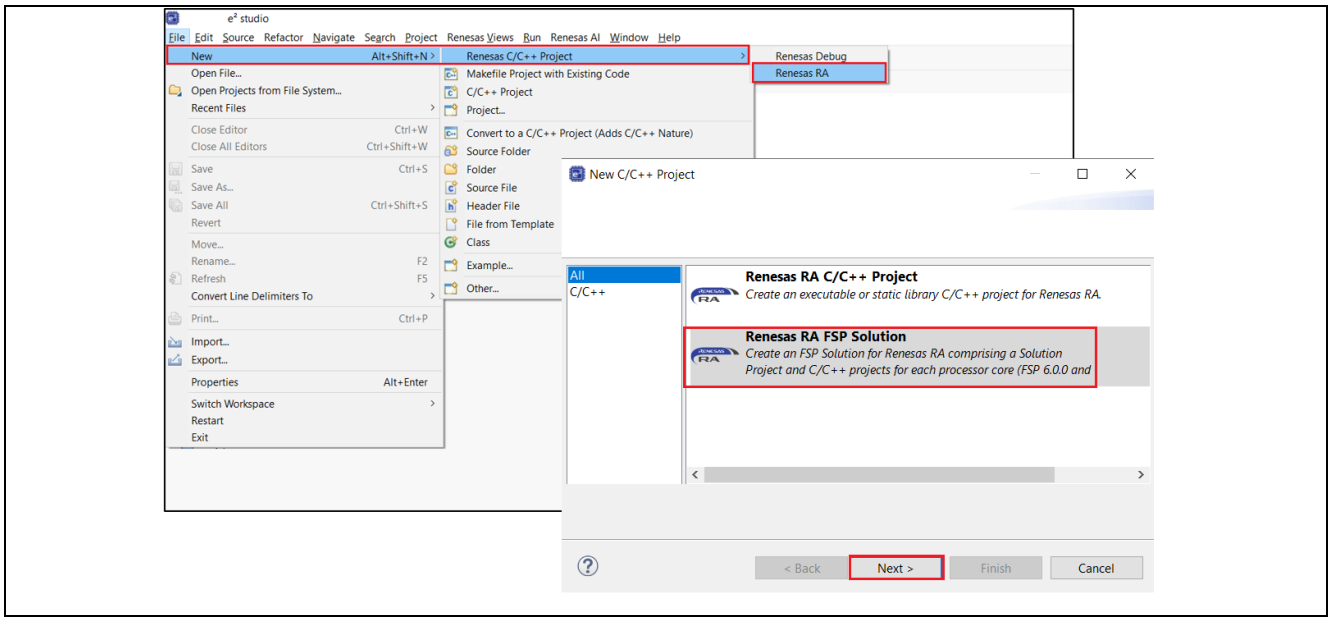

Figure 2. Example of Creating a Dual Core Project with RA FSP Solution

Assign a name for this new project such as **ek_ra8p1_blinky,** then > **Next**.

From the **Device Selection,** select the **Board** type of any supported dual-core RA kit (for example, EK-RA8P1 or EK-RA8D2).

Select the **Blinky** project template from **Multicore > Flat > Bare Metal** under the **Solution Template Selection** box, set the **Toolchains** to **LLVM Embedded Toolchain for Arm,** and J-Link ARM for **Debugger,** then > **Finish**.

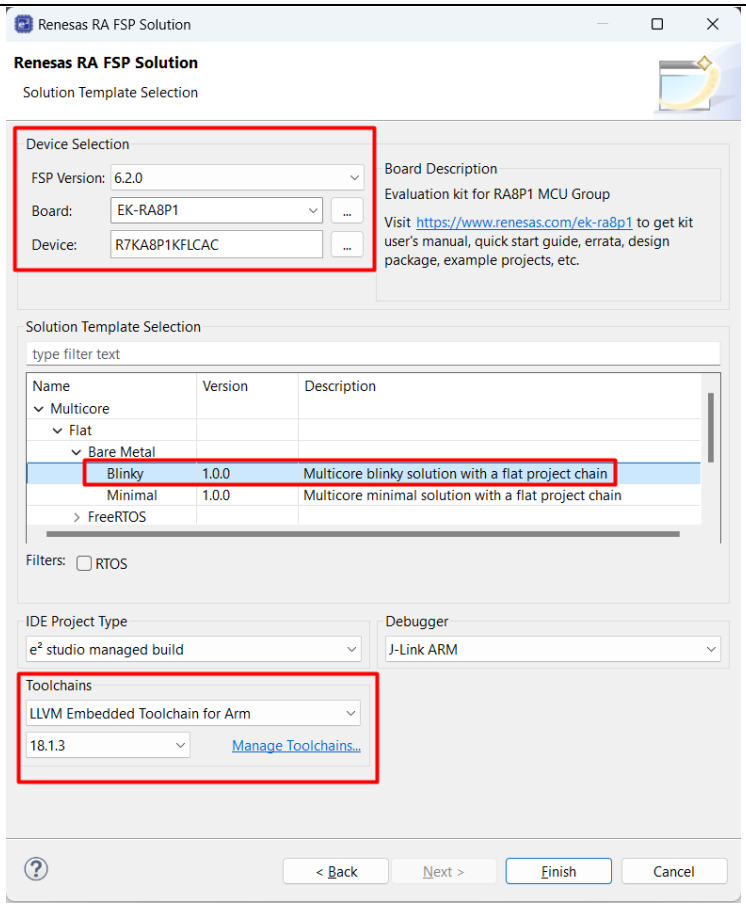

Figure 3. Example of Project Selections

Note that a dual-core project should be successfully created and built automatically after this step. It consists of three project folders, as shown in Figure 4.

- The solution project named **ek_ra8p1_blinky** provides a chain of projects for both CPU0 and CPU1. It manages the **Clock** configuration and **Memories** settings for both cores.
- The ek_ra8p1_blinky_CPU0 project corresponds to the CPU0 application.
- The ek_ra8p1_blinky_CPU1 project corresponds to the CPU1 application.

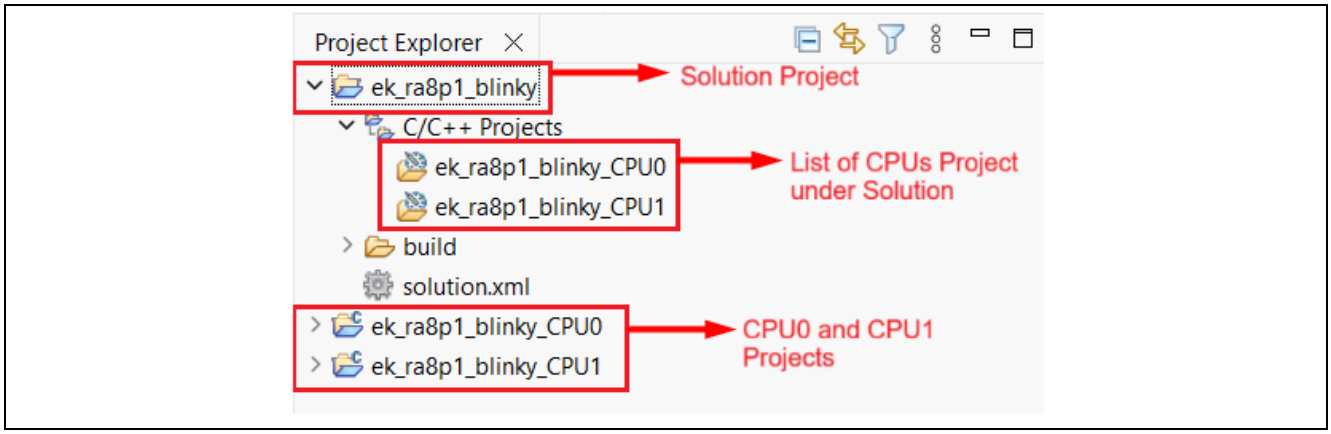

Figure 4. Example of Dual-Core Project Creation Success

Calling the FSP inline function `R_BSP_SecondaryCoreStart()` within the `hal_entry()` on CPU0 activates CPU1, as shown in Figure 5.

```
 * @brief  Blinky example application
void hal_entry (void)
{
#if BSP_TZ_SECURE_BUILD

    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif

    /* Define the units to be used with the software delay function */
    const bsp_delay_units_t bsp_delay_units = BSP_DELAY_UNITS_MILLISECONDS;

    /* Set the blink frequency (must be <= bsp_delay_units / 2) */
    const uint32_t freq_in_hz = 1;

    /* Calculate the delay in terms of bsp_delay_units */
    const uint32_t delay = bsp_delay_units / (freq_in_hz * 2);

    /* LED type structure */
    bsp_leds_t leds = g_bsp_leds;

    /* Wake up 2nd core if this is first core and we are inside a multicore project. */
#if (0 == BSP_CFG_RA_CORE) && (1 == BSP_MULTICORE_PROJECT) && !BSP_TZ_NONSECURE_BUILD
    R_BSP_SecondaryCoreStart();
#endif

    /* If this board has no LEDs then trap here */
    if (0 == leds.led_count)
    {
        while (1)
        {
            ;                           // There are no LEDs on this board
        }
    }
```

Start CPU1

Figure 5. Example of Starting CPU1 in hal_entry.c within the CPU0 Project.

In this blinky example, CPU0 blinks the LED1, and CPU1 blinks the LED2 on the board. This feature allows users to easily identify which core is currently toggling the LEDs, as shown in Figure 6.

```
    while (1)
    {
        /* Enable access to the PFS registers. If using r_ioport module then regist
         * handled. This code uses BSP IO functions to show how it is used.
         */
        R_BSP_PinAccessEnable();

#if BSP_NUMBER_OF_CORES == 1

        /* Update all board LEDs */
        for (uint32_t i = 0; i < leds.led_count; i++)
        {
            /* Get pin to toggle */
            uint32_t pin = leds.p_leds[i];

            /* Write to this pin */
            R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
        }
#else

        /* Update LED that is at the index of this core. */
        R_BSP_PinWrite((bsp_io_port_pin_t) leds.p_leds[_RA_CORE], pin_level);
#endif

        /* Protect PFS registers */
        R_BSP_PinAccessDisable();
```

Toggle a single LED based on the CPU number

Figure 6. Corresponding Blinky LED Implementation in hal_entry.c.

When rebuilding the project, follow this order:

- Build the CPU0 project first.
- Then build the CPU1 project.

If any settings are modified at the solution project, the solution project must also be rebuilt to propagate those changes to both CPU0 and CPU1 projects.
Alternatively, you can simplify it by right-clicking on the dual-core solution project and selecting **Build Project**. The auto-build process will take place in this order: first, the CPU0 project will be built, and then the CPU1 project will follow.
Building from the solution project is recommended for solution-based or multicore projects to ensure consistent configuration and dependency handling across both CPUs.

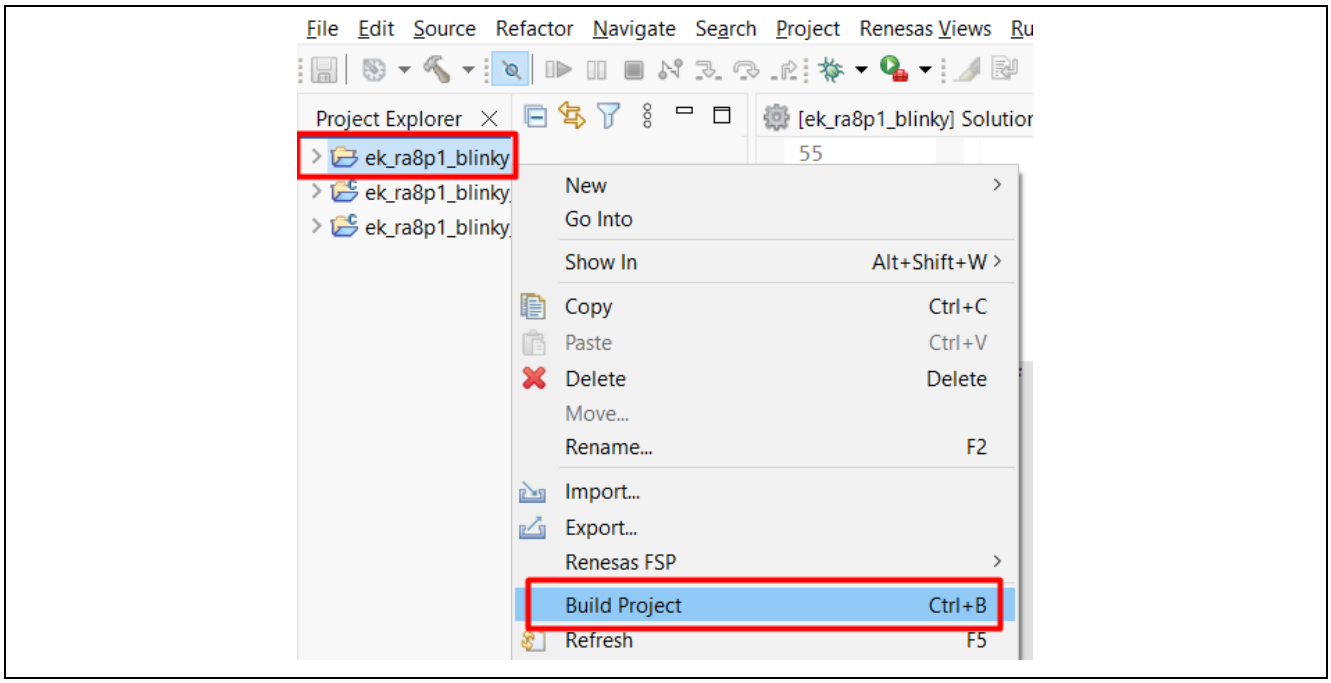

Figure 7. Example of Building the Dual-Core solution project.

Ensure that the project builds successfully for both CPUs and that the corresponding application images (*elf files) are generated in the Debug/ directory `(e.g., Debug/*.elf)`.

3.2 Debug and Run Multicore Project

To debug dual cores simultaneously, first use either Renesas Flash Programmer or Renesas Device Partition Manager to perform an initialization operation on your MCU to ensure that it is set to protection level 2 and that the TrustZone boundary has not been previously set. If the TrustZone boundary has been set, you will need to reset it to establish a proper environment for debugging. Once the initialization is complete, you can proceed to configure the dual-core settings and start the debugging process effectively. If you fail to complete this step, you might face difficulties in downloading your project images and initiating the debug process.

Follow steps below to initialize the device using Renesas Device Partition Manager:

1. Open **Renesas Device Partition Manager** from **Run > Renesas Debug Tools.**
2. Enable **Initialize device** in Action box.
3. Select the **Target MCU Connection** and **Connection type.**
4. Click **Run** to Initialize the MCU.

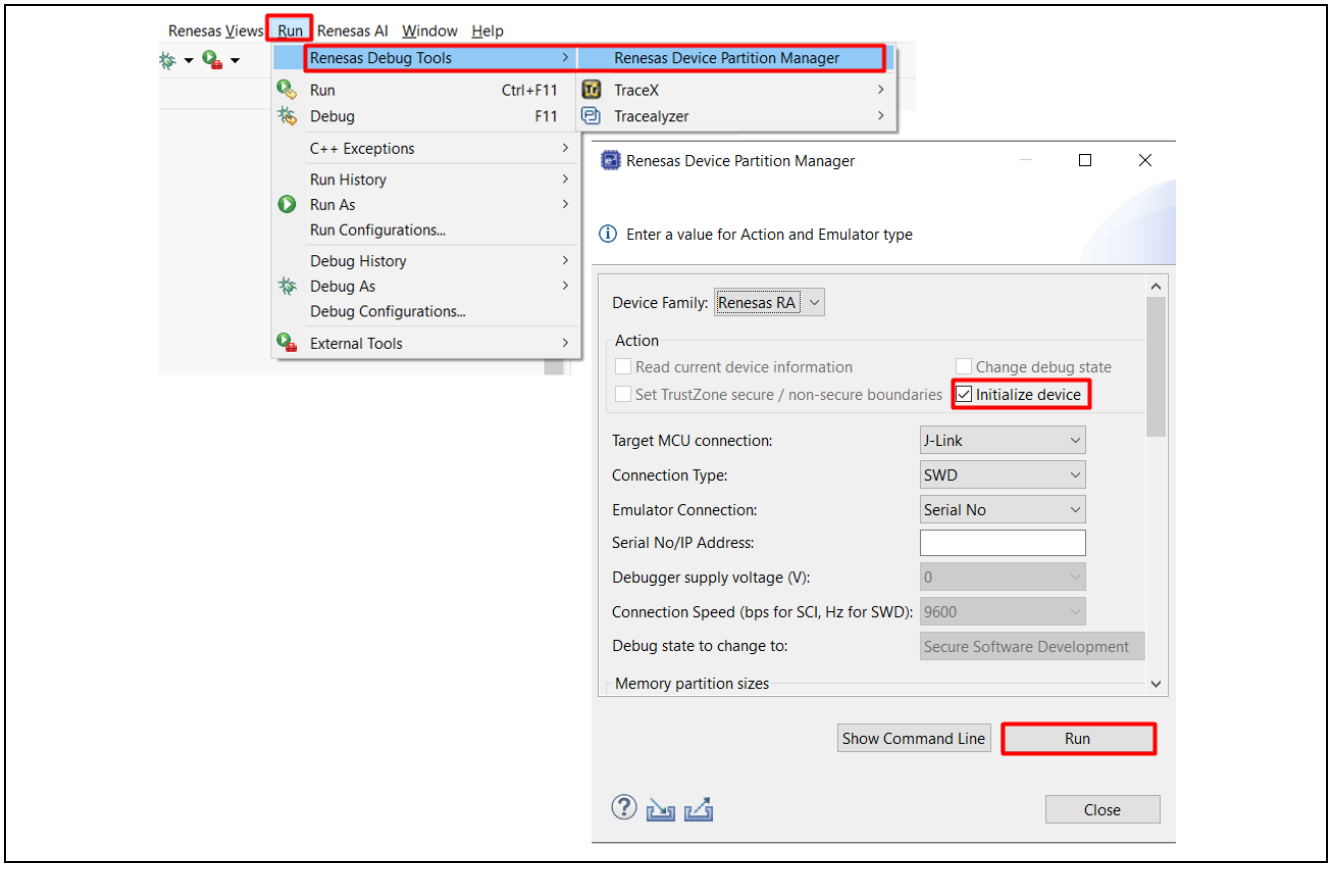

Figure 8. Initialize MCU with RDPM

A successful initialization results in the message shown in Figure 9.

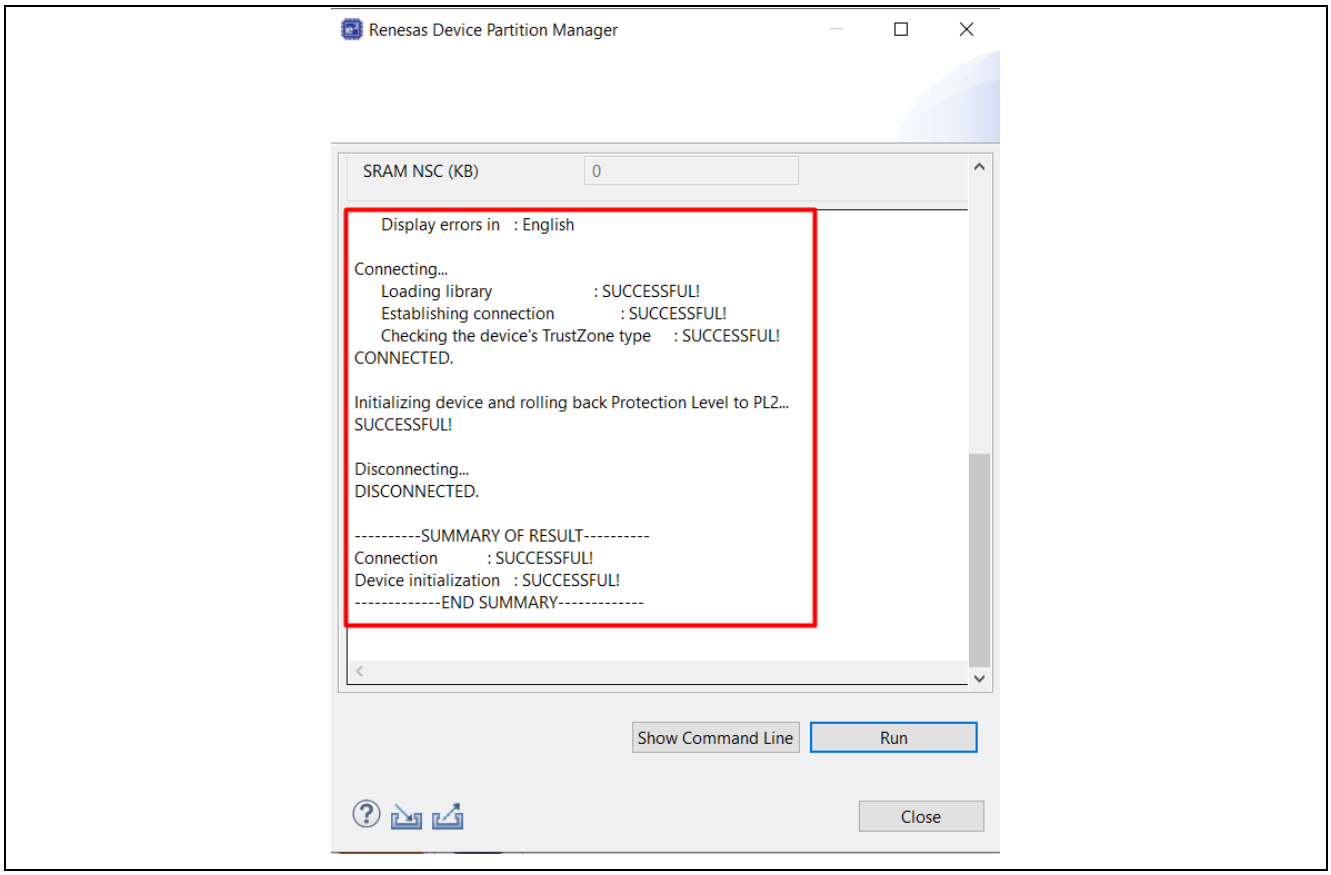

Figure 9. Successful Device Initialization Message on RDPM

To initialize the device using Renesas Flash Programmer:

1. Open the Renesas Flash Programmer software.
2. Create a new project and establish a connection to the target MCU.
3. Navigate to the **Target Device** tab.
4. Click **Initialize Device** to perform the initialization operation.

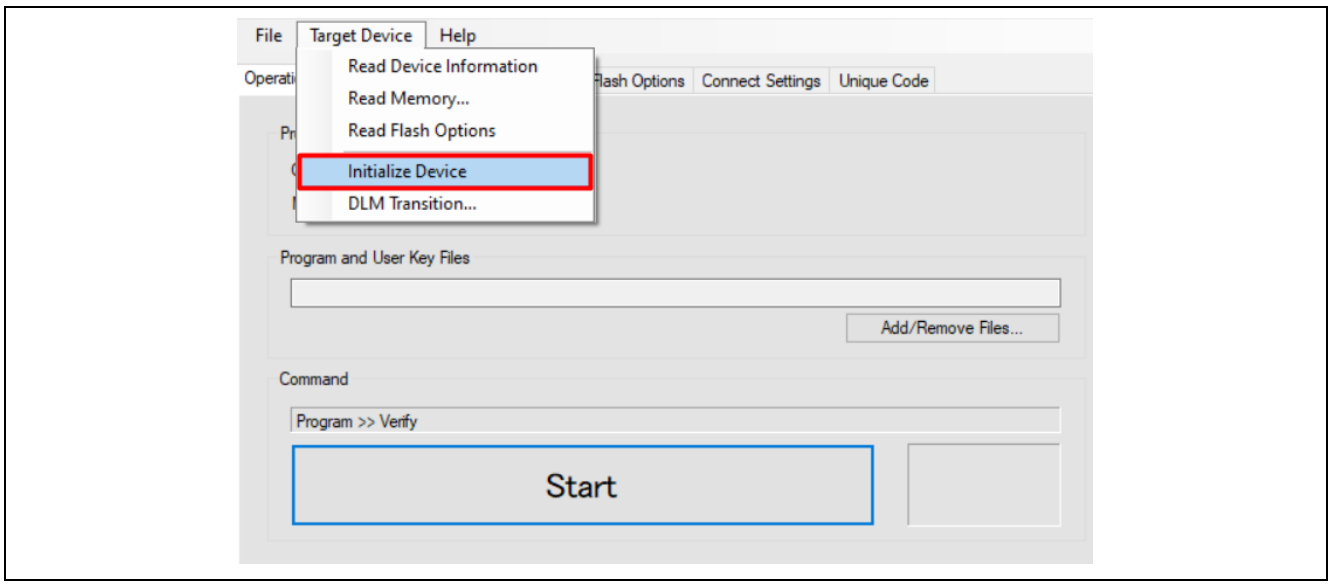

Figure 10. Initialize MCU with RFP

The Status Message will be displayed on the console as Figure 11.

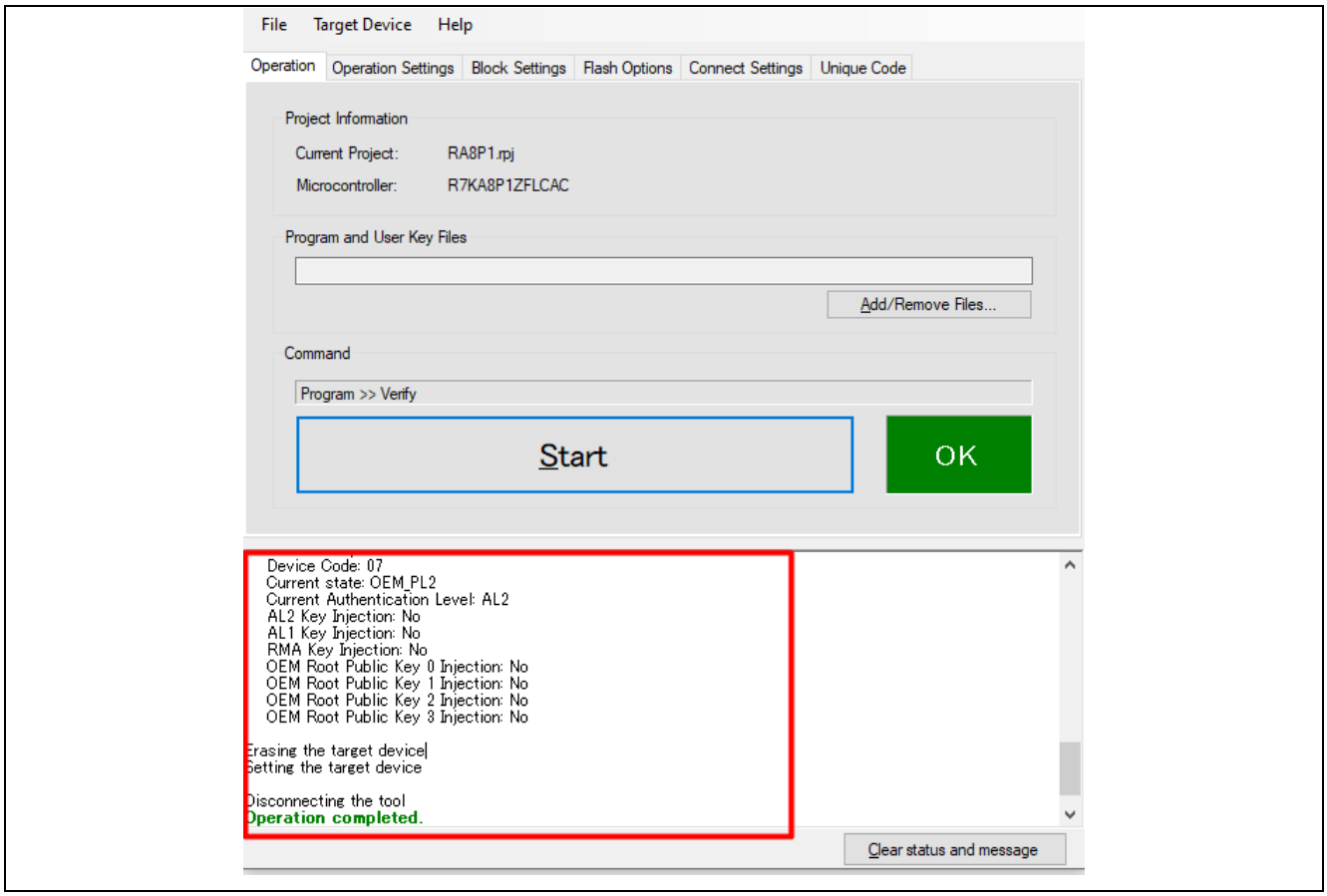

Figure 11. Successful Device Initialization Message on RFP.

After initializing the device with RDPM or RFP, access the Debug Configuration. Select ek_ra8p1_blinky_CPU1 Debug_Multicore, then navigate to the Debugger tab and click on Connection Settings. Ensure that the TrustZone boundary settings are disabled.

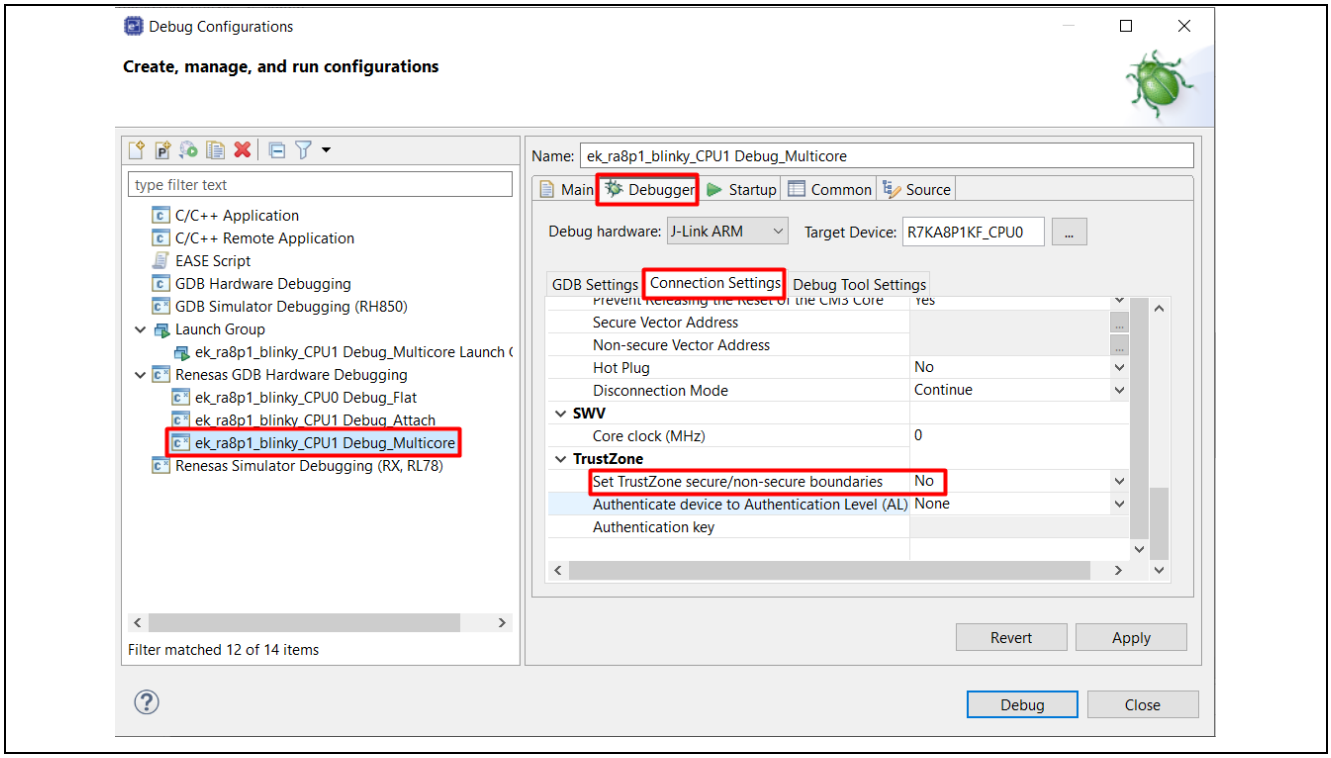

Figure 12. Example of Disabling TrustZone Boundary Setting

The e²studio provides efficient debug capability that allows debugging both core projects simultaneously. You can achieve this by using a "Launch Group" that combines both individual launch configurations. When e²studio generates the CPU1 project, it automatically creates this launch group.

Open the Debug Configurations dialog, select the Debug Multicore Launch Group that was created, and click Debug to begin the debug session.

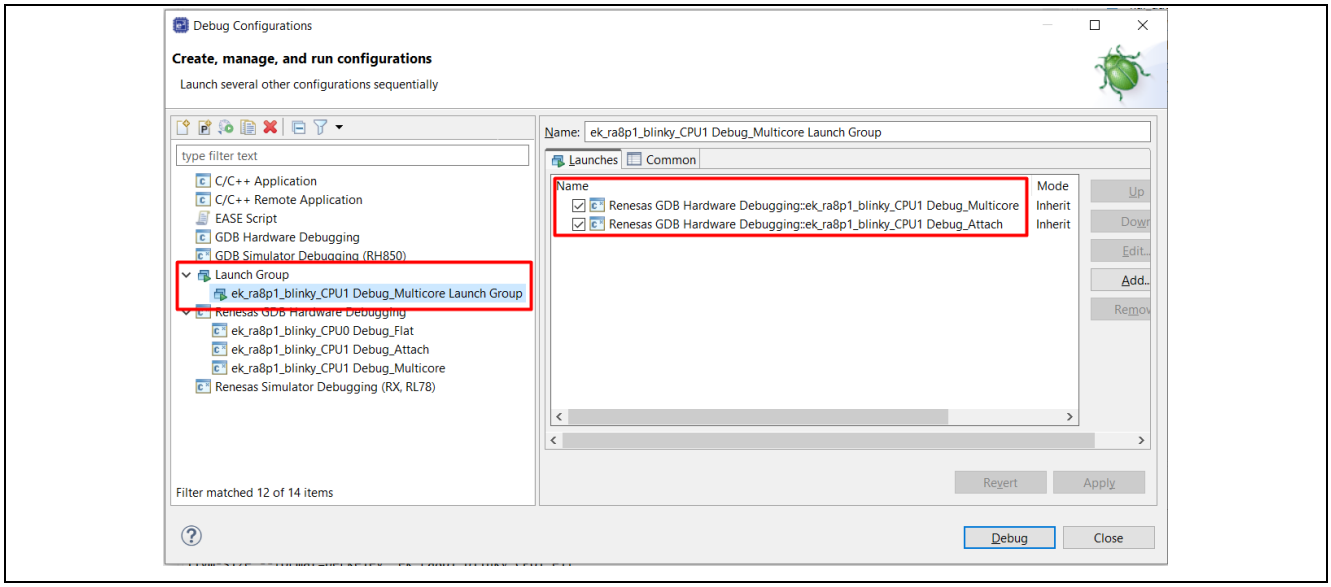

Figure 13. Example of Debug Multicore Launch Group

In the Debug information tab, the debug session for the CPU0 project is launched and connected first, followed by the CPU1 project. The debugger will halt at the `Reset_Handler()` of the CPU0 project. When you click the Resume button twice, CPU0 will start executing. After passing the `main()` function, the LED1 on the target board should start blinking, indicating successful operation.

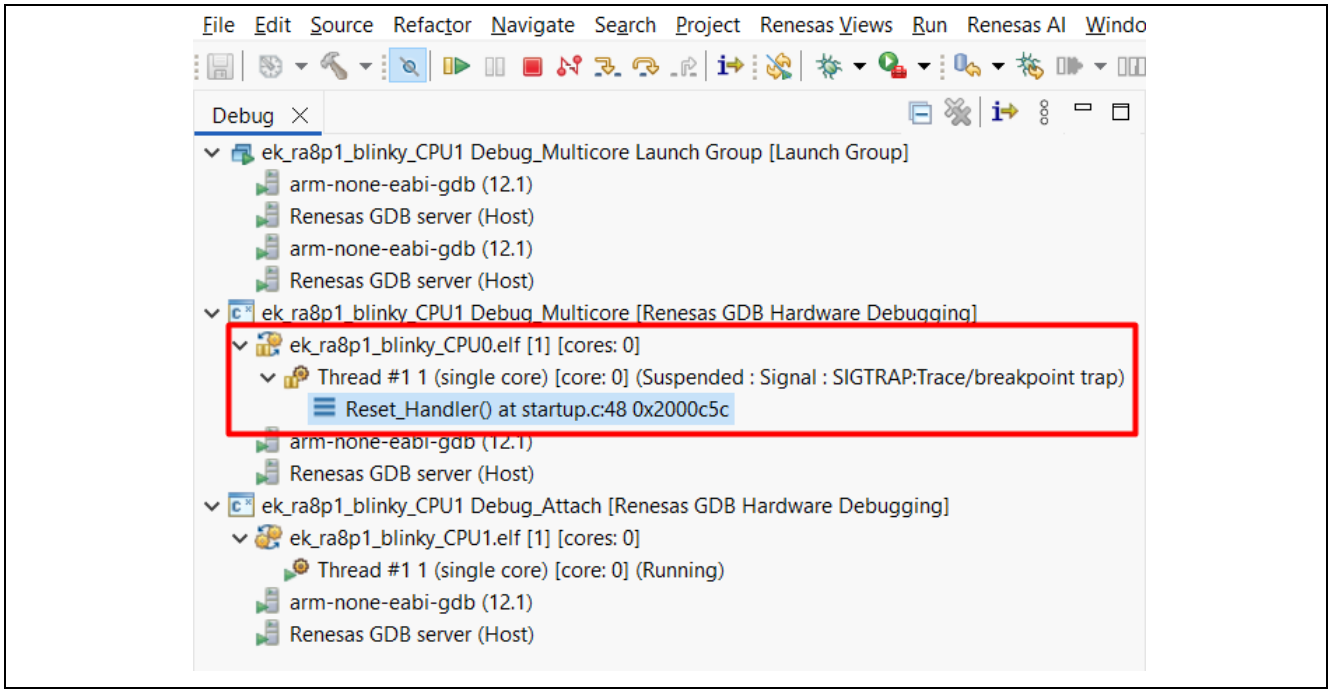

Figure 14. Example of Startup Debug State for Dual-Core Execution.

As shown in Figure 15, once CPU0 invokes `R_BSP_SecondaryCoreStart()`, the CPU1 debug session fully activates, enabling standard debugging control.

Press the **Resume** button to continue execution; LED2 will start blinking, indicating that CPU1 has successfully started up.

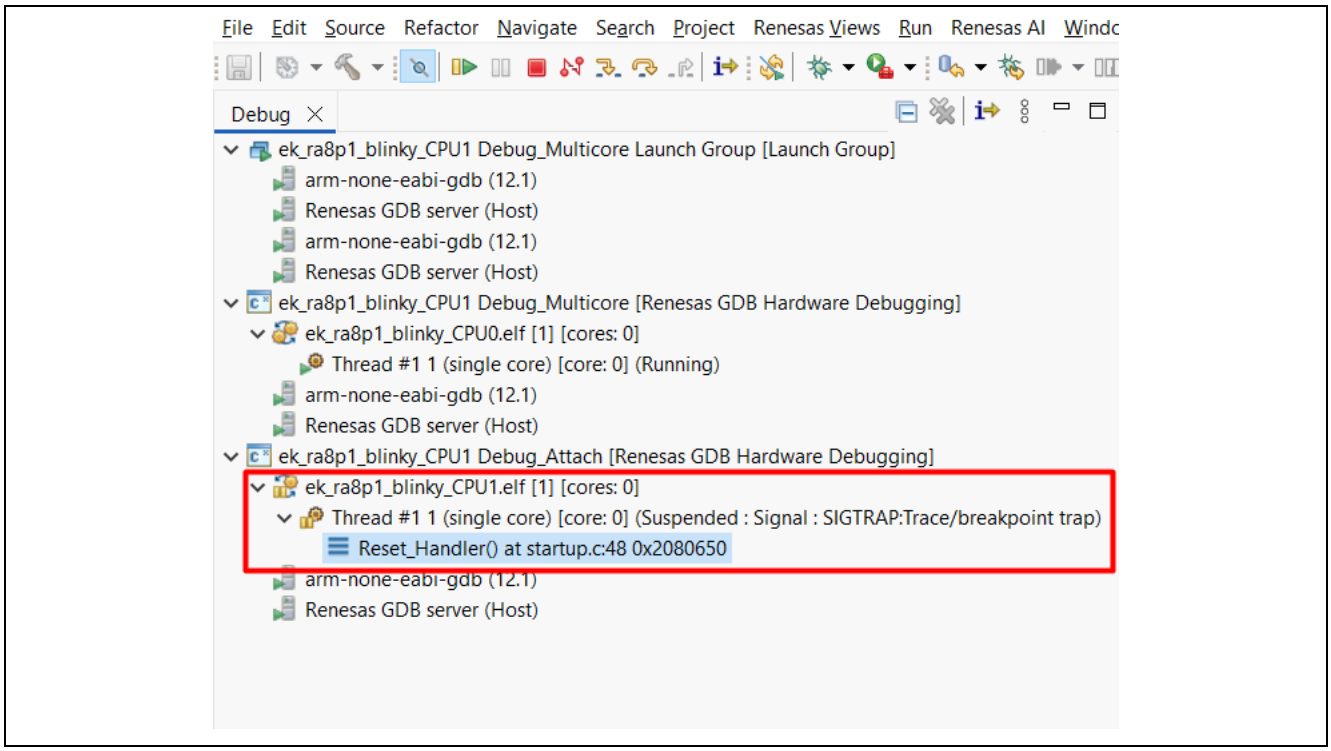

Figure 15. Example of Debug State after R_BSP_SecondaryCoreStart()

After completing your first multicore debug session for the project pair, you can quickly start debug sessions by selecting the appropriate Launch Group from the drop-down menu.

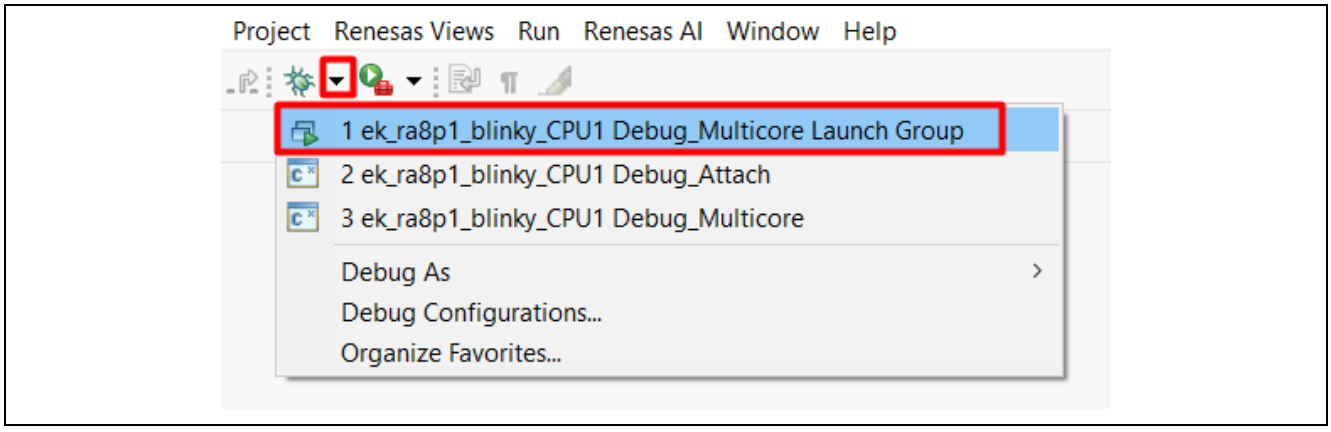

Figure 16. Example of Quick Access Using the Debug Drop-Down Menu

4. Developing Application Using RA8 Dual Core MCU

When planning and designing an application that utilizes RA8 dual-core MCUs, it is crucial to consider several factors to maximize performance.

- Divide the application into multiple tasks that each CPU can process independently.
- Use the IPC (Inter-Process Communication) module to facilitate communication between tasks running on different cores. Additionally, manage shared resources carefully to prevent conflicts, maintain data integrity, and optimize the overall efficiency of the application.
- Utilize ITCM, DTCM, CTCM, and STCM to enhance performance.
- Leverage instruction and data cache to further improve performance.
- Assign heavier tasks, such as graphics-related operations, digital signal processing (DSP), and artificial intelligence/machine learning (AI/ML), to the CM85 core. This core can take advantage of its higher clock frequency, the M-Profile Vector Extension (MVE). In contrast, lighter tasks, including data acquisition (sensor inputs) and user messages (UART input/output), should be managed by the CM33 core.

4.1 Partition the system and maximize performance

We distribute real-time control, AI/ML, and graphics tasks across the CPU cores to leverage the dual-core architecture. For example, we can divide a typical graphics application between the two CPUs as follows.

- CPU0 manages the graphics module, JPEG decoder, and SDRAM access.
- CPU1 handles inputs/outputs and the user interface, including data acquisition, touch controllers, and output controls.

This division of responsibilities reduces resource contention particularly on system buses, shared memory and enables each CPU to operate with minimal interference. It also creates a scalable foundation for additional workloads, such as real-time control loops, sensor fusion, or AI/ML inference, to be assigned to either core based on performance requirements.

Well-defined partitioning, combined with inter-processor communication (e.g., IPC interrupts, shared memory, and message queues), allows the dual-core system to operate efficiently while ensuring that high-priority tasks execute predictably.

4.2 Using Inter-Processor Communication in Application

Inter-processor communication (IPC) allows for the sharing of hardware resources and the exchange of data between the two processors within the MCU. IPC also supports communication by generating interrupt events that help synchronize and coordinate actions among the processors.

This section highlights the key components of IPC, including:

- Data exchange
- Task synchronization
- Efficient resource sharing

4.2.1 Using Inter-Processor Interrupts

Inter-processor interrupts provide a low-latency mechanism for notifying the other CPU when shared data or system events require immediate attention. The RA dual-core architecture supports both maskable and non-maskable inter-processor interrupts, enabling flexible priority handling depending on the criticality of the event.

In a typical workflow shown in Figure 17, one core performs event monitoring for example, peripheral activity, buffer availability, or completion of a processing task, and then issues an inter-processor interrupt to the peer core. The interrupt acts as a lightweight notification mechanism, allowing the receiving CPU to respond immediately when an event occurs.

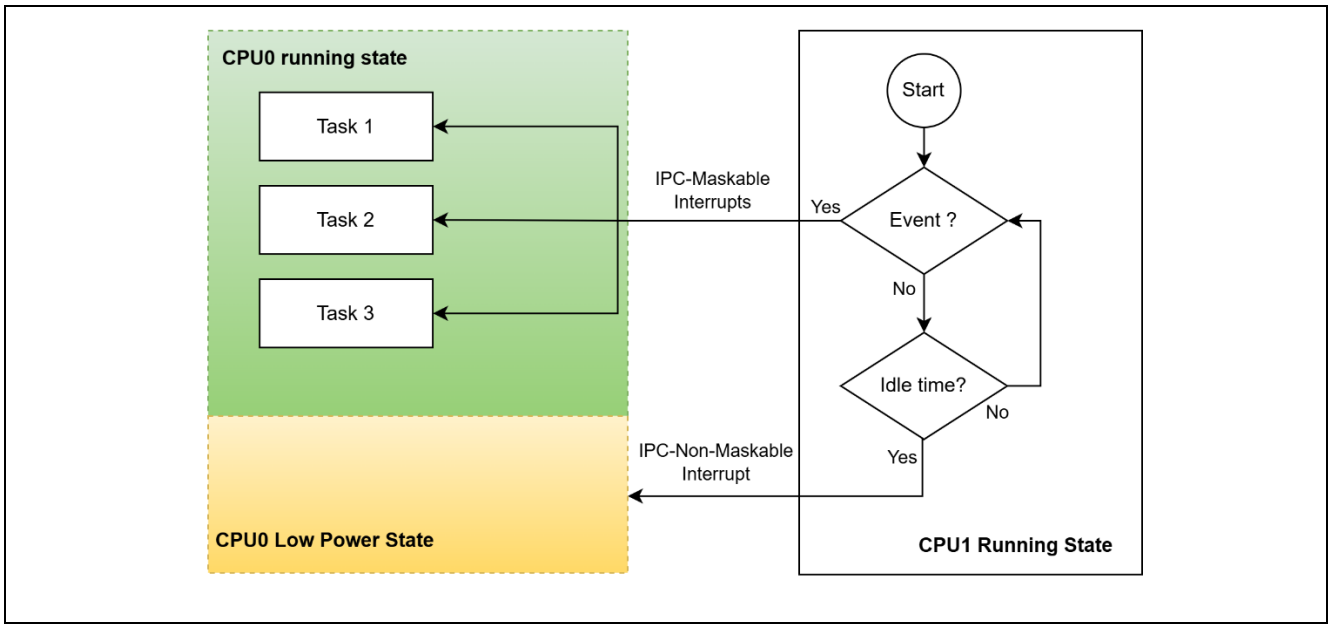

Figure 17. Example of CPU0 Notifications Using IPC Interrupts

To avoid race conditions when sharing memory or status flags, each interrupt notification is typically paired with a well-defined communication protocol. This may include writing to a reserved shared-memory structure, updating state flags, or posting data through an IPC message queue. The receiving CPU can then process the updated information, perform the required task, and optionally generate a return interrupt if bidirectional synchronization is needed. These concepts are further detailed in Section 4.3, and the exclusive-control flow involved in shared-memory access is illustrated in Figure 20.

When combined with semaphores, message queues, and shared-memory buffers, inter-processor interrupts form the backbone of reliable inter-processor coordination in real-time or graphics-intensive dual-core systems. They ensure deterministic wake-ups, minimize latency, and reduce system-bus contention compared to software polling or timer-based notification loops.

For implementation details regarding inter-processor interrupts in the application, refer to section 5.1.1.

4.2.2 Using Inter-Processor Communication FIFO Messages.

The IPC module provides four hardware FIFOs that enable efficient and deterministic message passing between the two CPU cores. Each FIFO is unidirectional, allowing the software to establish separate communication channels for different message types or priority levels.

- **IPC00 and IPC01**: FIFOs for messages sent from **CPU1 → CPU0.**
- **IPC10 and IPC11**: FIFOs for messages sent from **CPU0 → CPU1.**

Each FIFO contains four stages and supports a 32-bit data width, enabling fast transfer of small command structures, flags, or message tokens without involving shared memory or complex synchronization. Because the FIFOs are implemented in hardware, data transfers occur with minimal latency and without requiring software-based arbitration.

This mechanism is well suited for:

- Command and event notifications.
- Lightweight synchronization.
- Posting small data tokens.
- Triggering state changes or dispatching tasks on the peer CPU.

Figure 18 illustrates the data exchange mechanism using IPC Message FIFOs.

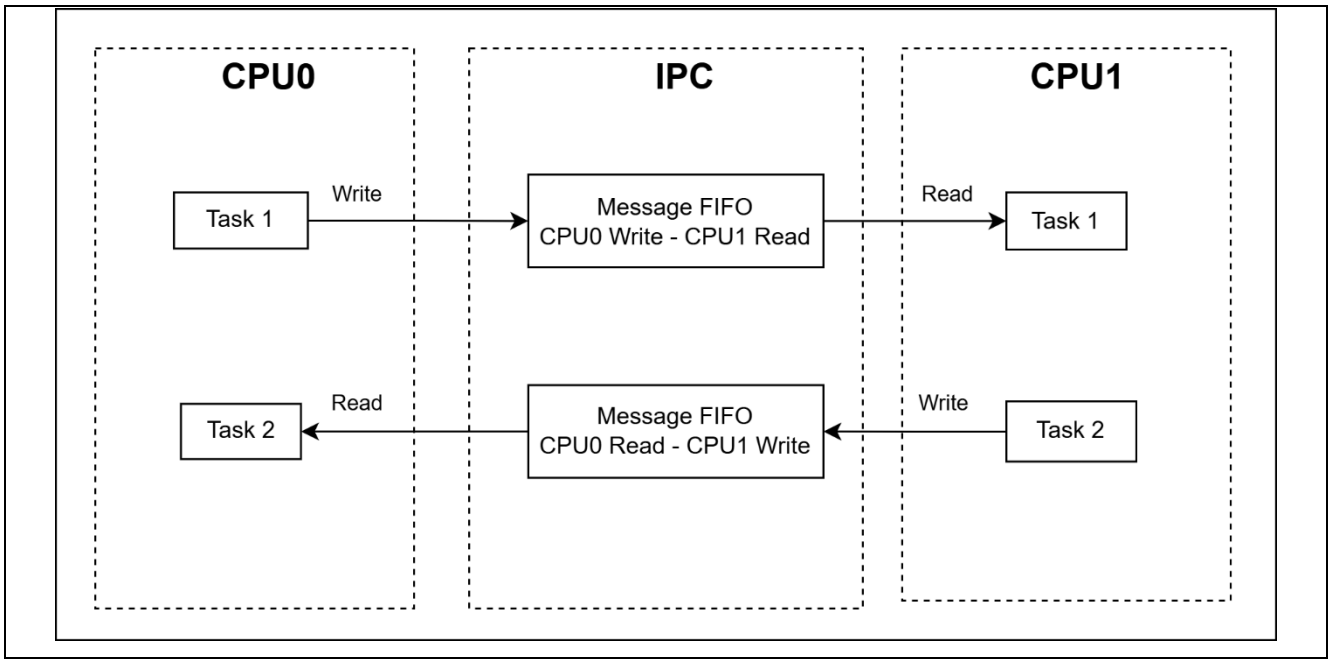

Figure 18. Example of Data Exchange with IPC-Messages FIFO

For more advanced shared-memory communication or larger data exchange patterns, refer to Section 5.1, where IPC FIFOs are combined with shared-memory structures and an RTOS queue for inter-processor data exchange.

4.3 Using Shared Memory and Resources in RA8 Dual Core MCU

Shared memory is one of the most efficient methods of exchanging large amounts of data between two cores in a dual-core system. This mechanism allows two cores to directly access a common memory area to read/write data, but concurrency control is required to avoid race conditions and data corruption.

4.3.1 Using Share Memory and Resources in FSP Flat Projects

When two cores need to exchange high-speed, large amounts of data (e.g., video streaming), shared memory is the ideal choice because of its higher speed compared to other IPC methods. However, appropriate synchronization mechanisms are needed to avoid race conditions. Figure 19 demonstrates a method for data exchange and notifications to ensure synchronized access to a shared memory region.

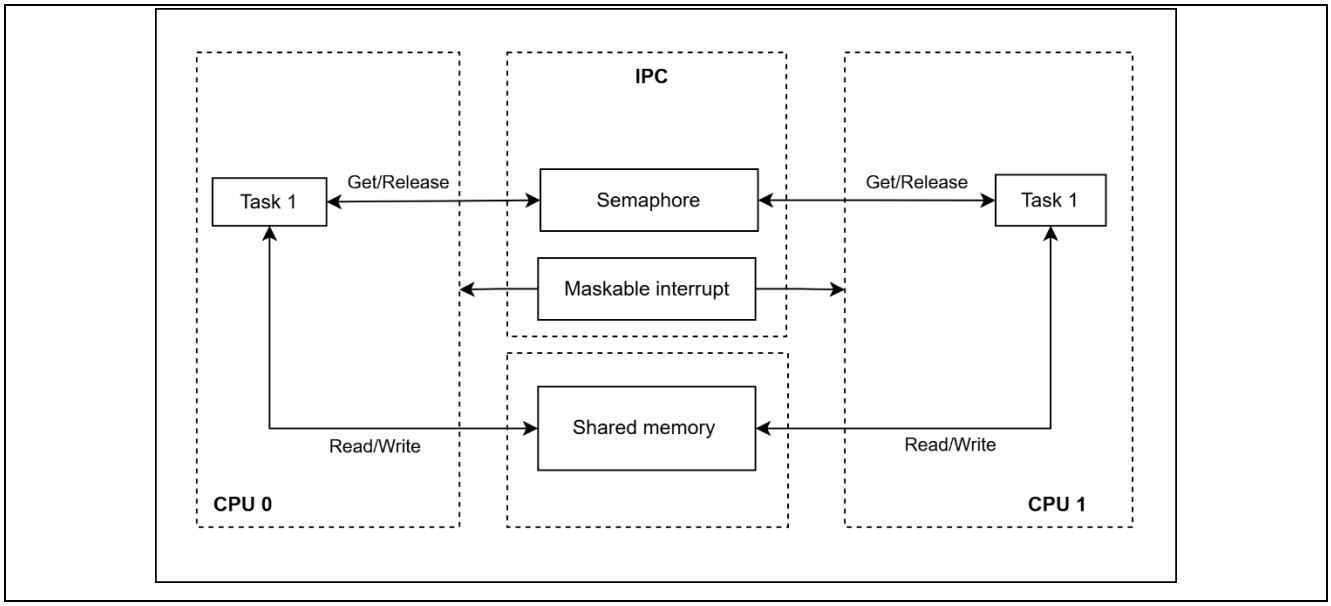

Figure 19. Example of Shared Memory Data Exchange and Synchronization

When implementing shared memory in a dual-core system, several key considerations must be addressed.

- Memory Allocation: A dedicated memory region should be reserved and configured as shared memory that is accessible by both cores.
- Synchronization Mechanism: Appropriate synchronization methods, such as mutexes, semaphores, or hardware flags, are necessary to prevent data corruption and ensure correct access sequencing.
- Data Exchange: One core writes data to the shared memory region while the other core reads it as needed, facilitating coordinated communication between the cores.

Figure 20 illustrates the exclusive control flow that occurs when both CPUs execute a semaphore and send core notifications through an inter-processor maskable interrupt.

Refer to 5.1 for instructions on implementing shared memory in the application.

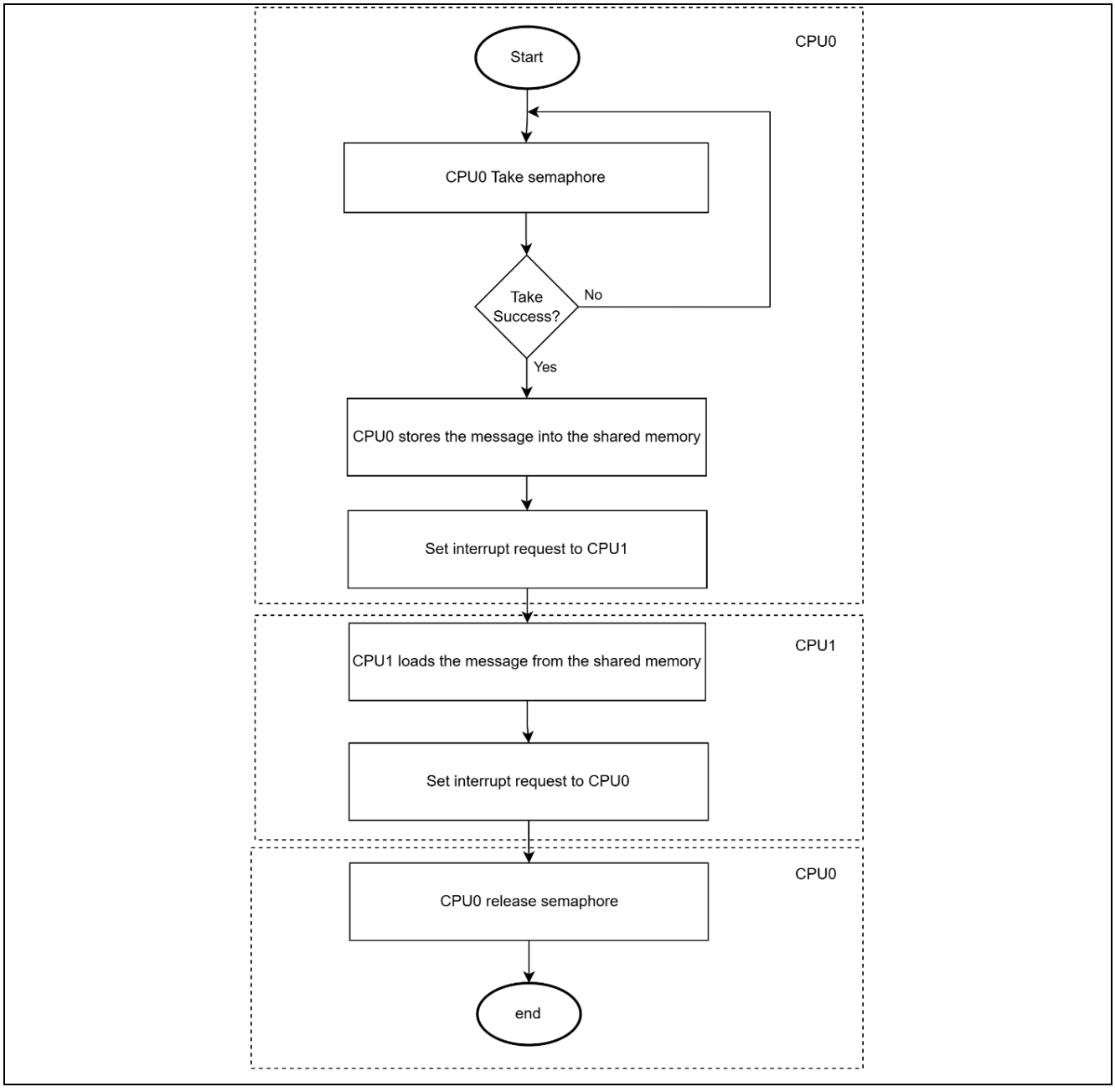

Figure 20. Example of Application Exclusive Control Flow

When using a dual-core system, dedicating certain services to a single core can optimize performance and avoid resource duplication. This strategy ensures that each core can concentrate on its designated tasks without interference, resulting in enhanced efficiency. By effectively managing workloads, users can enjoy smoother performance and faster response times in applications. This method also assists in optimizing

code size and addressing resource-sharing challenges. When a task needs these services, it can send a request through the inter-processor communication channel.

Some types of services are shared between the two cores and handled on one core, such as peripheral device management (UART/SPI/I2C), file system management, and network stack management. Figure **21** illustrates the shared UART peripheral, which is managed by CPU1, while CPU0 accesses the UART service via IPC mechanisms.
This architecture is also demonstrated in the accompanying sample application.

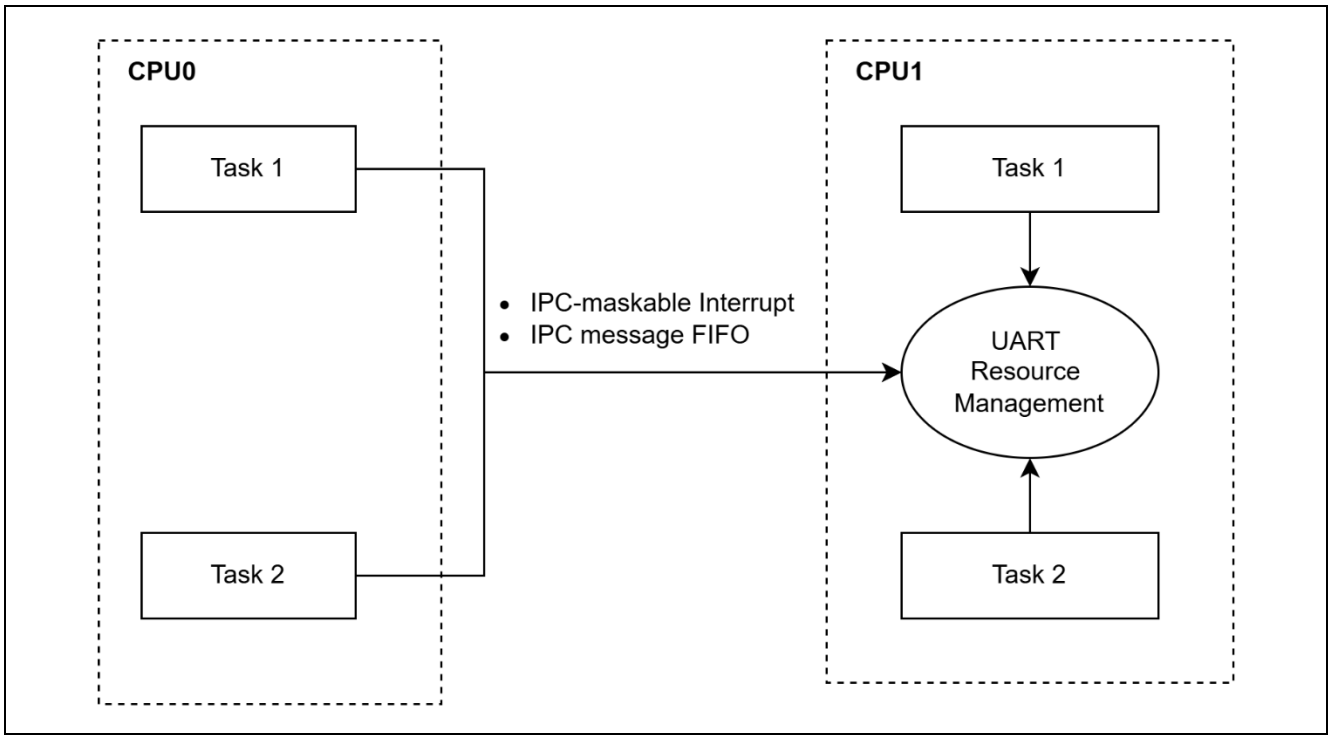

Figure 21. **Example of Share Resource - UART diagram**

4.3.2 Using Share Memory and Resources in RTOS Based Projects

In FreeRTOS-based multicore applications, the use of shared memory and shared resources is conceptually same to bare-metal implementations. Users must still apply appropriate techniques to avoid race conditions and bus contention when both CPUs access common memory regions or peripherals.

However, FreeRTOS provides additional synchronization mechanisms such as semaphores, mutexes, queues, stream buffers, and message buffers that simplify the development of complex multitask, multicore applications. In this project, a lightweight message queue wrapper built on top of the Renesas FSP IPC module is used to exchange data between the two cores. Users may alternatively employ other FreeRTOS primitives, such as stream buffers or message buffers, depending on the requirements of their application.

For further examples related to shared memory usage and IPC mechanisms with FreeRTOS, refer to 5.2.

4.4 Utilize Caches and TCM In RA8 Dual Core Applications

For optimal performance, Tightly Coupled Memory (TCM) and cache can be leveraged in conjunction with Helium™ technology.

TCM typically offers deterministic, single-cycle access, minimizing latency in time-critical operations. Placing performance-critical code and frequently accessed data in TCM ensures faster and more predictable execution.

4.4.1 Tightly Coupled Memory (TCM)s

With the RA8 dual core MCU, each CPU has its own dedicated TCM resource.

- CPU0: Instruction Tightly Coupled Memory (ITCM); Data Tightly Coupled Memory (DTCM).
- CPU1: C-AHB Tightly Coupled Memory (CTCM); S-AHB Tightly Coupled Memory (STCM).

The 256 KB TCM memory in CPU0 consists of 128 KB ITCM and 128 KB DTCM.

The 128 KB TCM memory in CPU1 consists of 64 KB CTCM and 64 KB STCM.

Note: Accessing TCMs is not available in CPU Deep Sleep mode, Software Standby mode, and Deep Software Standby mode.

Figure 22 shows TCM in the local MCU subsystem.

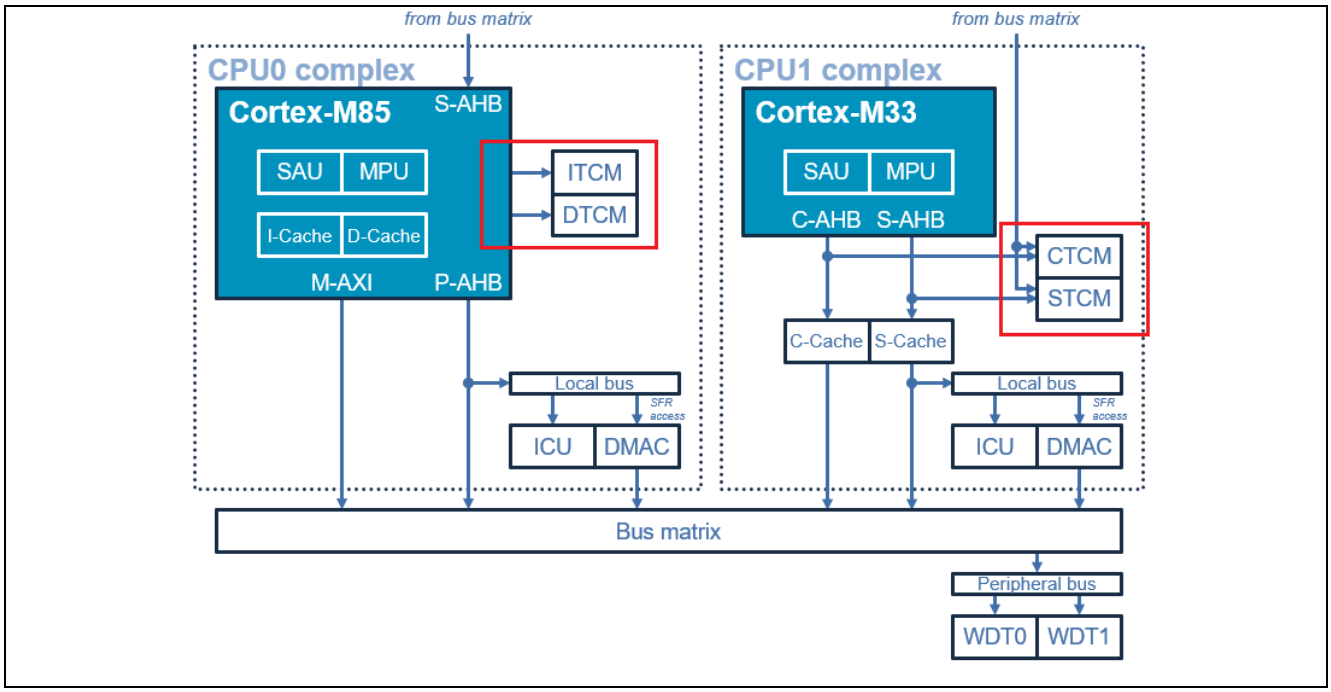

Figure 22. Example of TCM Memory in Local MCU Subsystem

4.4.1 Improve Performance Using ITCM

To achieve optimal performance for time-critical functions, specific program instructions can be placed in Instruction Tightly Coupled Memory (ITCM). This configuration can be adjusted using the Linker Section settings under C/C++ project FSP configurator in e²studio.

The following example illustrates how to allocate the `arm_cmplx_mag_f32` function to ITCM memory within the RA8P1_DSP_example project using the Linker Sections Configuration interface.

1. Open the project in **e² studio**.
2. In the **Project Explorer**, double-click on configuration.xml.
3. Navigate to the **Linker Sections** tab (refer to Figure 23).
4. Assign the function arm_cmplx_mag_f32 to the designated ITCM section (Figure 24 to Figure 26).

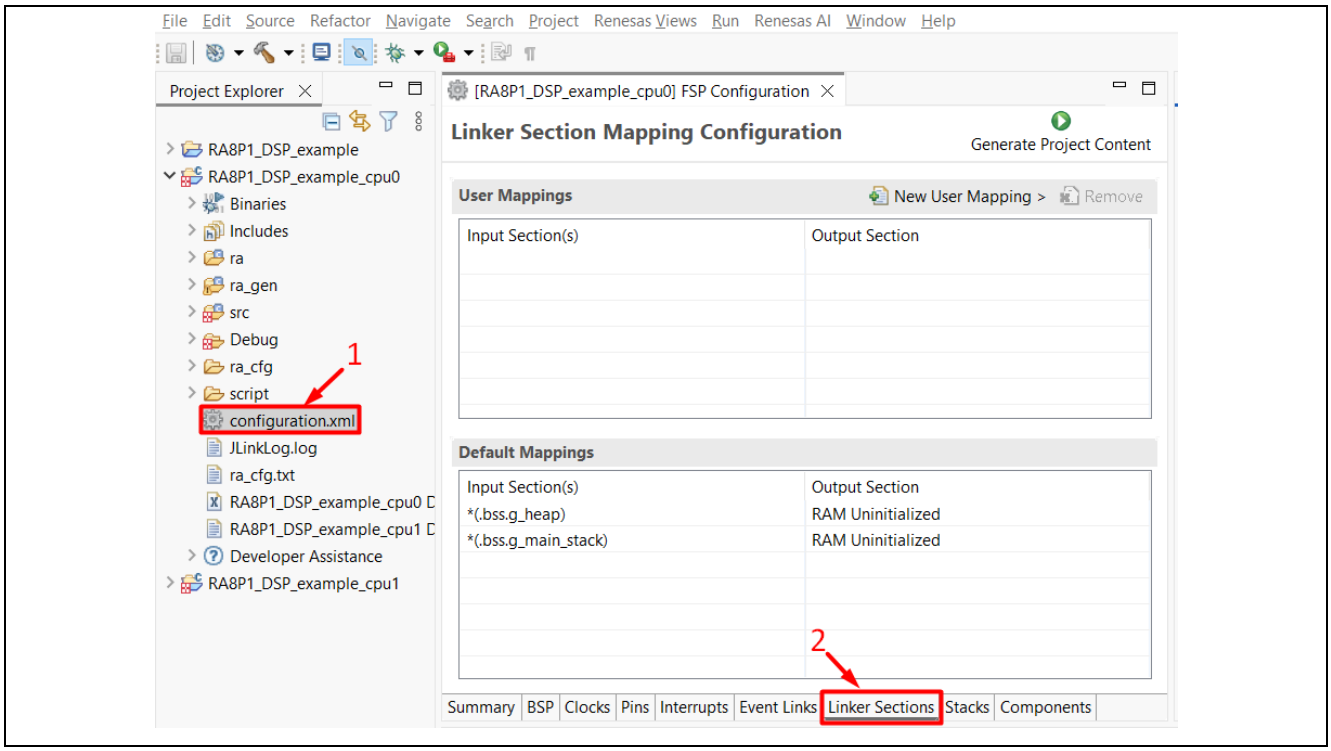

Figure 23. Open Linker Sections Tab

Click **New User Mapping** > **ITCM** > **Code initialized from** > **ITCM code from FLASH**.

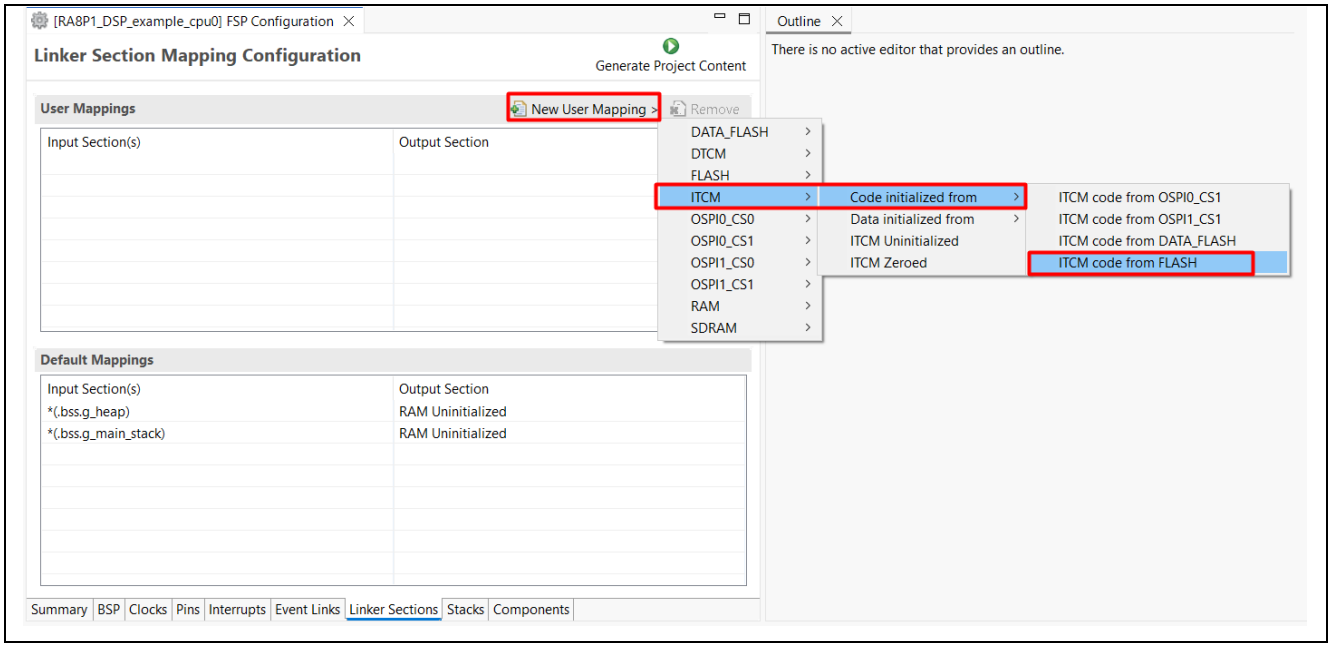

Figure 24. Example of Defining a New Section Mapping for ITCM

A new user mapping window appears. The input section name must adhere to a specific format. For sections that include instruction code, ensure the input section name follows the required naming convention, **.text.<Function Name>**. Figure 25 illustrates an example of placing the `arm_cmplx_mag_f32` function in ITCM memory.

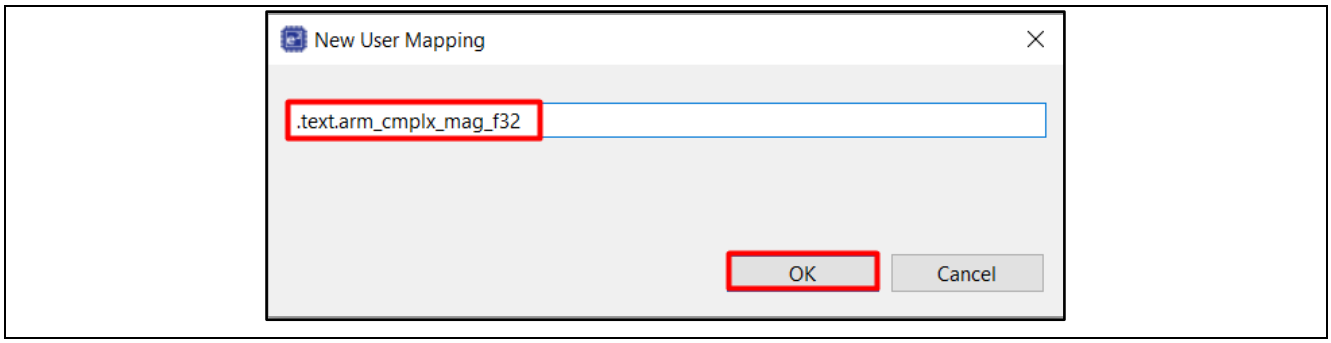

Figure 25. Example of Defining Input Section Name for Instruction Code

After completing the configuration shown in Figure 26, follow these steps to apply the changes: Click on "**Generate Project Content**" and then select "**Build Project**" to generate the code and compile the updated project configuration.

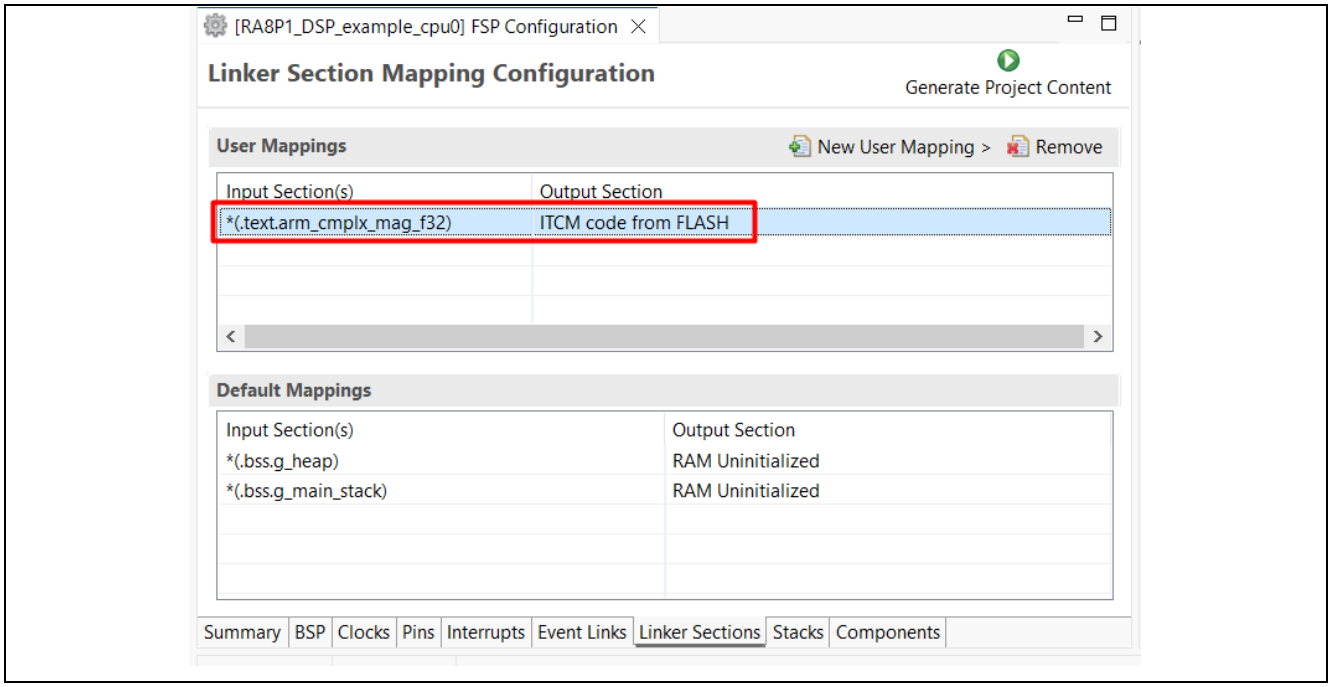

Figure 26. Example of Successful Setup for Function in ITCM section.

Upon a successful build, the placement of code or data in the specified memory section can be verified by inspecting the map file located at `Debug/*.map`. This file provides a detailed memory layout, including section assignments. Refer to Figure 27 for example illustrating the correct mapping of configured sections.

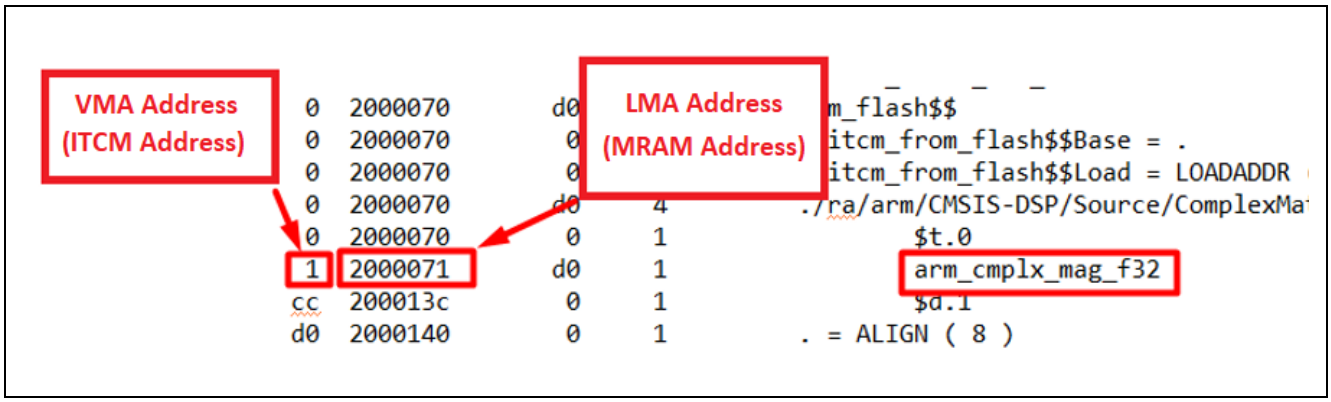

Figure 27. Example of Verifying Code Placement in ITCM

4.4.2 Improve Performance Using DTCM

To improve runtime performance, data buffers may be allocated in the Data Tightly Coupled Memory (DTCM) region. Management of these buffers can be performed through the Linker Sections tab in the e²studio configuration for the project.

The following procedure demonstrates how to locate the testInput_f32_10khz buffer used in the RA8P1_DSP_example project (as shown in Figure 28) in DTCM memory by utilizing the Linker Sections Configuration interface:

Click the **Linker Sections** tab > **New User Mapping** > **DTCM** > **Data initialized from** … > **DTCM data from FLASH**, as shown in Figure 29.

This configuration ensures that the buffer is copied from MRAM to DTCM at startup and enables faster data access during execution.

```
/* ------------------------------------------------------------------
Test Input signal contains 10KHz signal + Uniformly distributed white noise
** ------------------------------------------------------------------ */

float32_t testInput_f32_10khz[2048] =
{
-0.865129623056441,     0.000000000000000,  -2.655020678073846,     0.0000000000
-2.899160484012034,     0.000000000000000,  2.563004262857762,  0.00000000000000
0.048366940168201,  0.000000000000000,  -0.145696461188734,     0.00000000000000
-1.176633086028377,     0.000000000000000,  3.690223557991855,  0.00000000000000
2.739754205367484,  0.000000000000000,  -0.062610410524552,     0.00000000000000
1.195039415434387,  0.000000000000000,  -2.177388969045026,     0.00000000000000
```

Figure 28. Example of Placing testInput_f32_10khz Buffer to DTCM

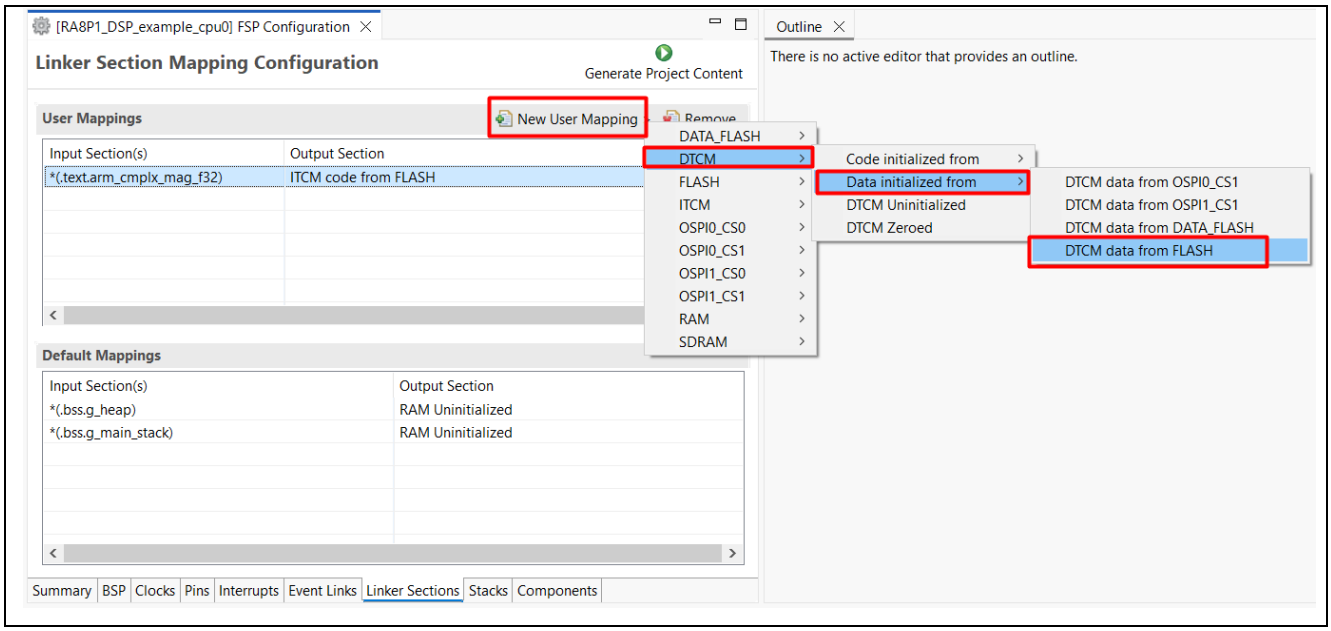

Figure 29. Example of Defining a New Section Mapping for DTCM

Next, assign “Input section name” corresponding to the initialized data. Use the following format for section naming: `.data.<buffer_name>`

For example, to map the initialized buffer testInput_f32_10khz, specify the input section as: `.data.testInput_f32_10khz.`

Refer to Figure 30 for a visual example of this configuration.

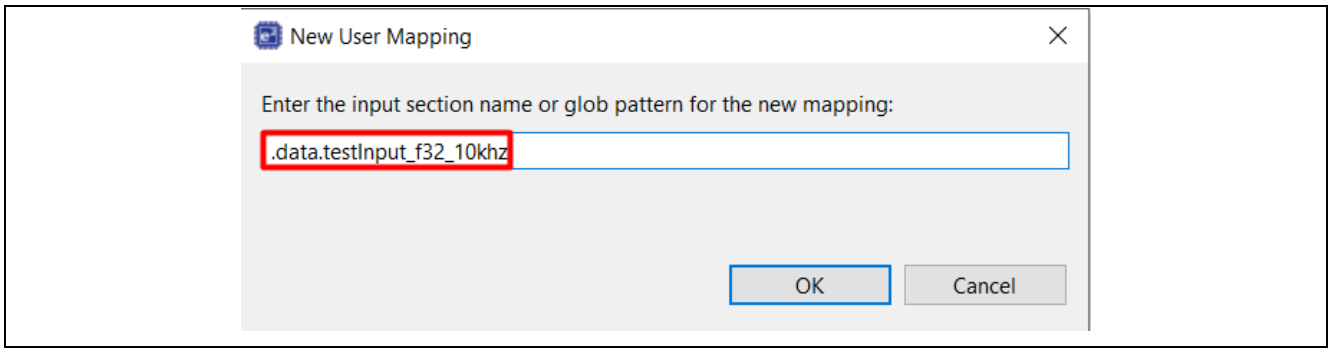

Figure 30. Example of Defining Input Section Name for Initialized Data

After finishing the configuration illustrated in Figure 31, click on **Generate Project Content**, and then choose **Build Project** to apply the changes and compile the updated project.

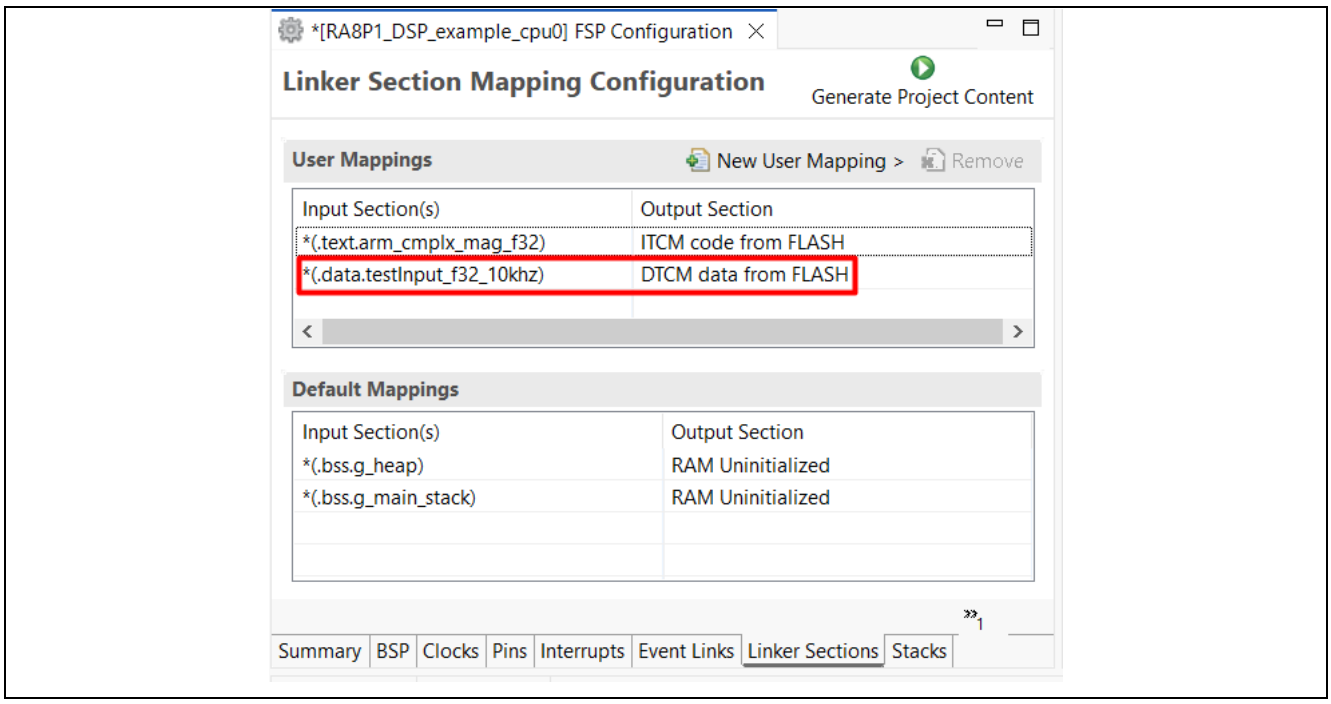

Figure 31. Successful Setup for Data in DTCM Section

Upon a successful build, the placement of code or data in the specified memory section can be verified by inspecting the map file located at `Debug/*.map`. This file provides a detailed memory layout, including section assignments. Refer to Figure 32 for examples illustrating the correct mapping of configured sections.

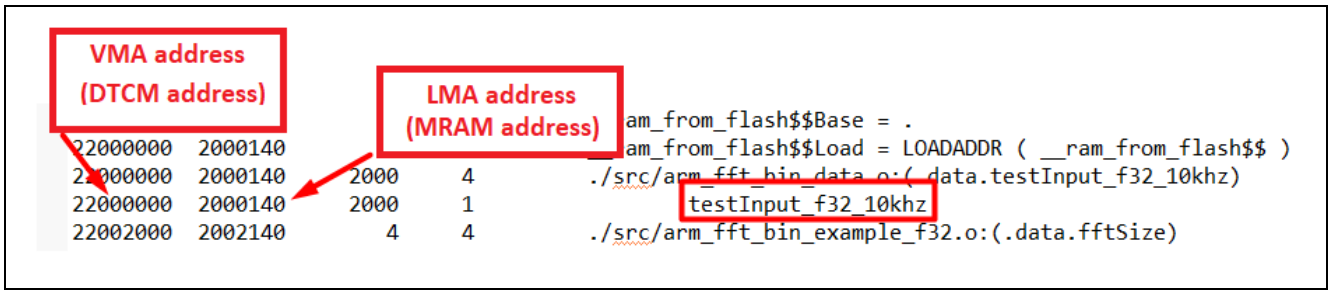

Figure 32. Example of Verifying Test Input Data Placement in DTCM

4.4.3 Improve Performance Using CTCM

To achieve optimal performance for time-critical functions on CPU1, specific program instructions can be placed in CTCM. This configuration can be performed using the Linker Sections settings available in the configuration.xml of the CPU1 project within e²studio. By explicitly mapping selected functions to CTCM, the application benefits from reduced instruction fetch latency and improved execution speed.

The following procedure demonstrates how to place the `ipc_callback()` function into CTCM on CPU1 using the Linker Sections configuration interface in e²studio. This approach ensures that the callback routine executes at high speed and has deterministic memory access, thereby enhancing response time during inter-processor communication events.

1. Open the project in **e² studio**.
2. In the **Project Explorer**, double-click on configuration.xml.
3. Navigate to the **Linker Sections** tab.
4. Assign the function `ipc_callback()` to the designated CTCM section (Figure 33 and Figure 34).

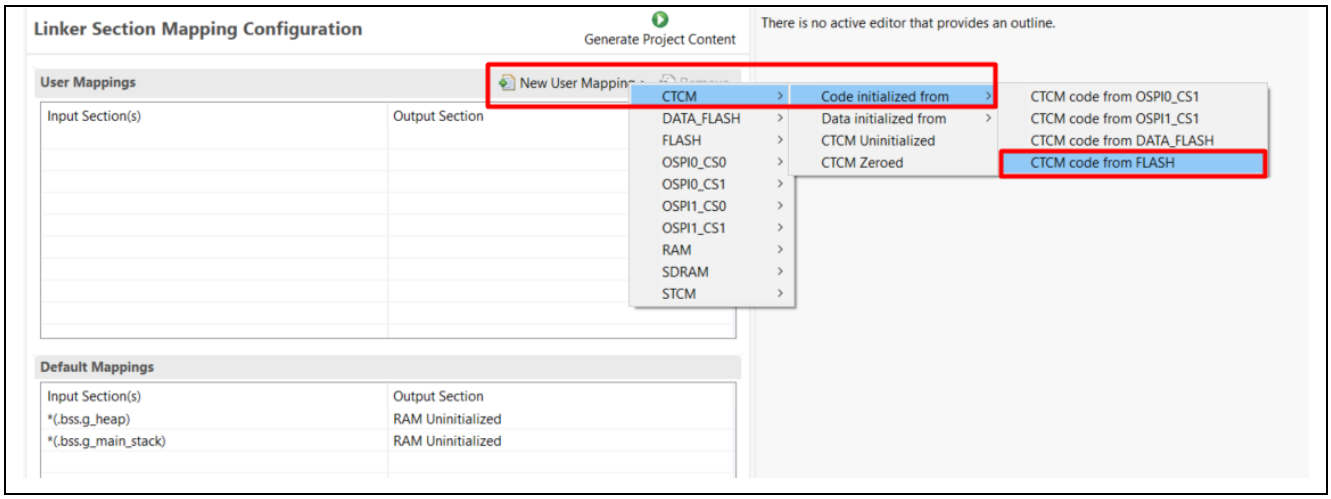

Figure 33. Example of Defining a New Section Mapping for CTCM

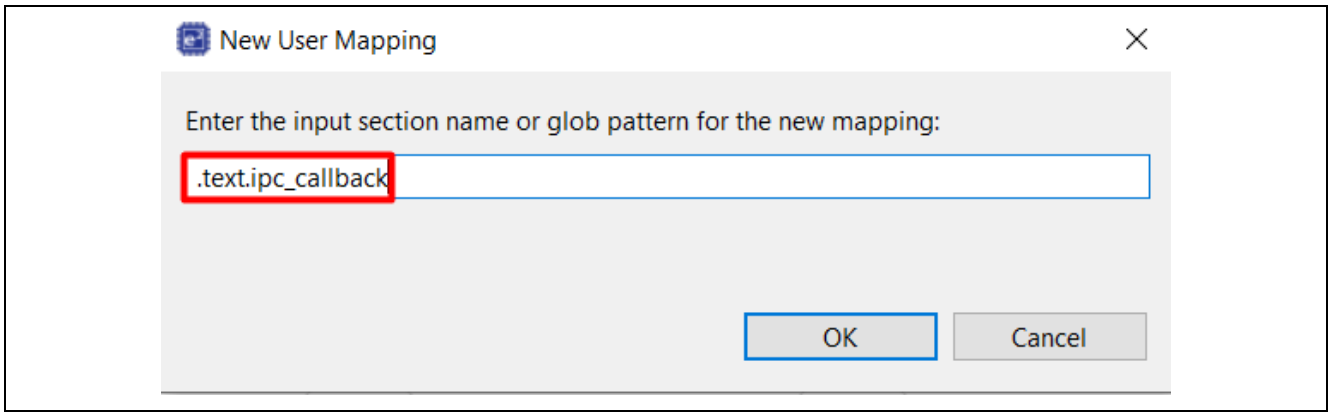

Figure 34. Example of Defining Input Section Name for Instruction Code on CPU1

After completing the configuration, click on **Generate Project Content** and then select **Build Project** to generate the code and compile the updated project configuration. Once the build is successful, you can verify the placement of code or data in the specified memory section by inspecting the map file located at `Debug/*.map`, as illustrated in Figure 35.

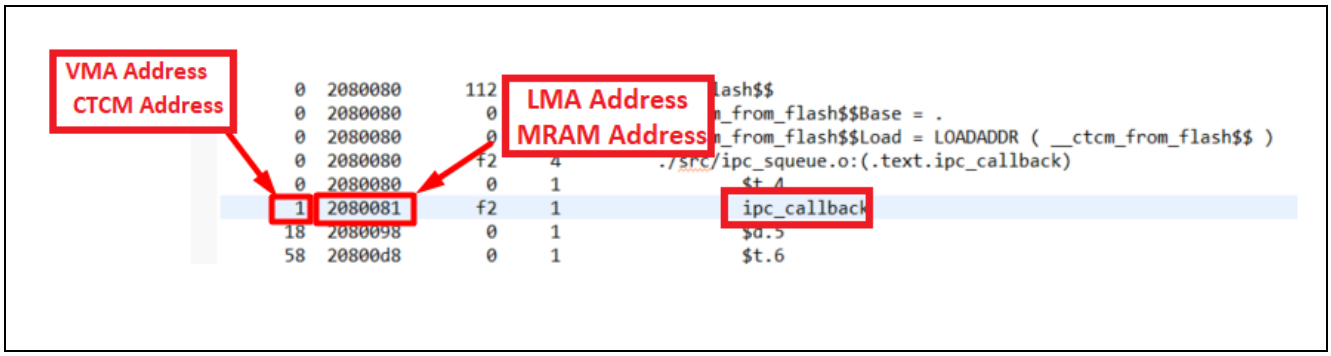

Figure 35. Example of Successfully Allocate ipc_callback to CTCM

4.4.4 Improve Performance by Utilizing Data Cache Cortex®-CM85 Core

In the default configuration for RA8 devices, the FSP always enables the CM85 Instruction Cache (I-Cache) and manages its coherency as necessary. The FSP also allows for the optional enabling of the CM85 Data Cache (D-Cache) in the BSP configuration settings, as illustrated in Figure 36, although it is disabled by default. When utilizing any type of cache within a system, it's worth thinking about coherency. For further details, refer to the Cortex-M85 Caches documentation.

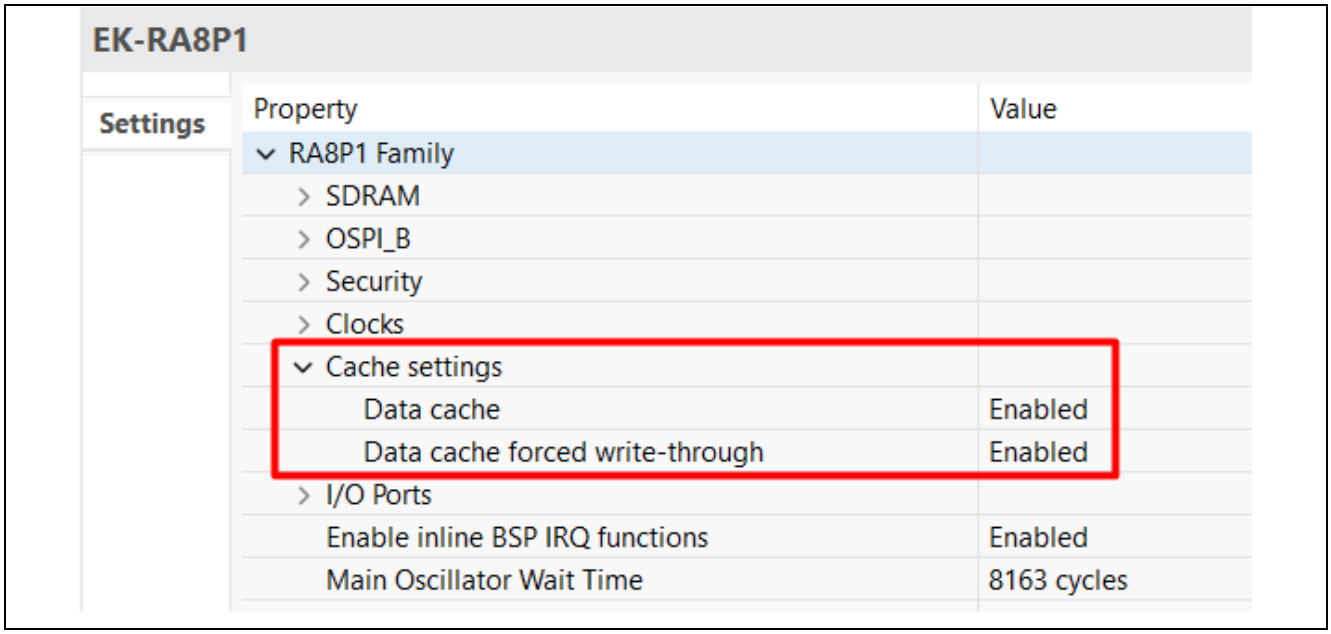

Figure 36. Example of CM85 Data Cache (D-Cache) Enabled

4.4.5 Using Neural Processing Unit (NPU)

The RA8P1 dual core MCU integrates Ethos-U55, which offers support for transformer-based models at the edge, the foundation for newer language and vision models, and scales from 32 to 256 MAC units, enabling higher-performance edge AI use cases in a sustainable way. Offering the same toolchain as previous Ethos-U generations, partners can benefit from seamless migration and leverage investments in Arm-based machine learning (ML) tools. The Ethos-U NPU works alongside the host CPU, providing efficient AI/ML acceleration for applications like computer vision, speech recognition, and anomaly detection.

By offloading ML computations from the CPU to the Ethos NPU, the system achieves higher performance and energy efficiency, enabling AI at the edge without a significant power overhead. Figure 37 and Figure 38 are an NPU block diagram and an example of Ethos configuration in Renesas FSP.

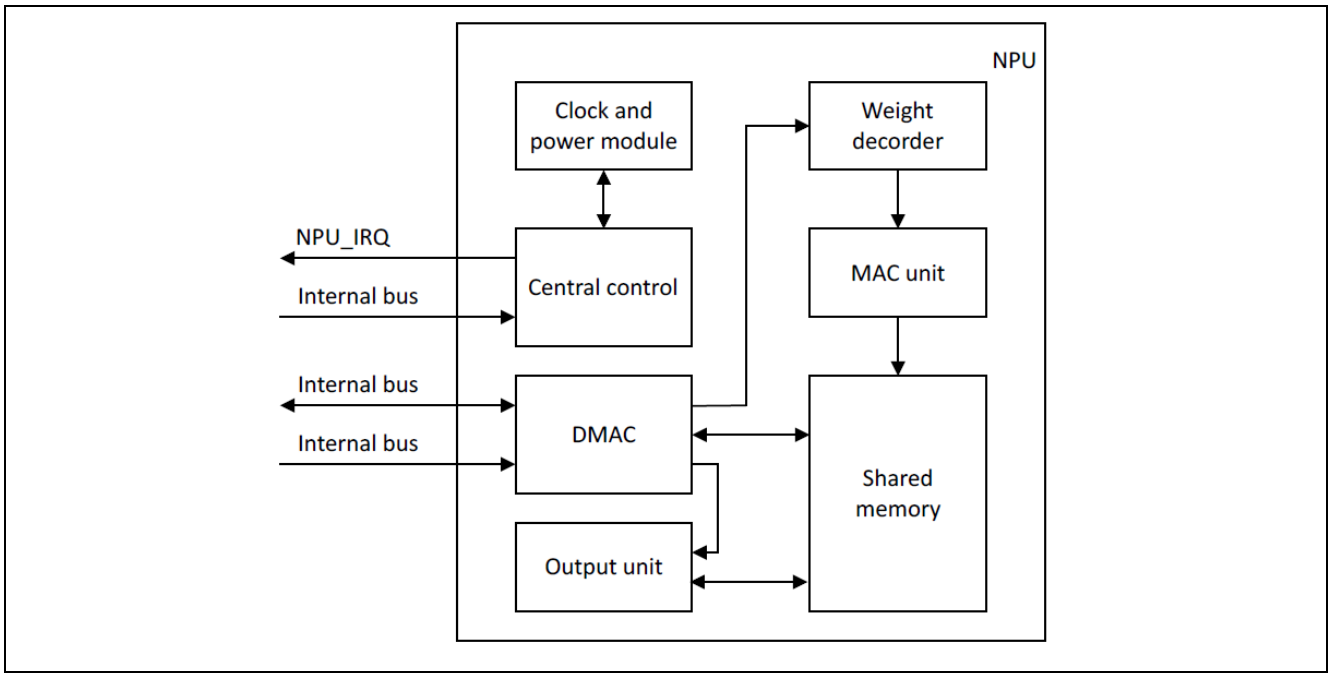

Figure 37. NPU Block Diagram

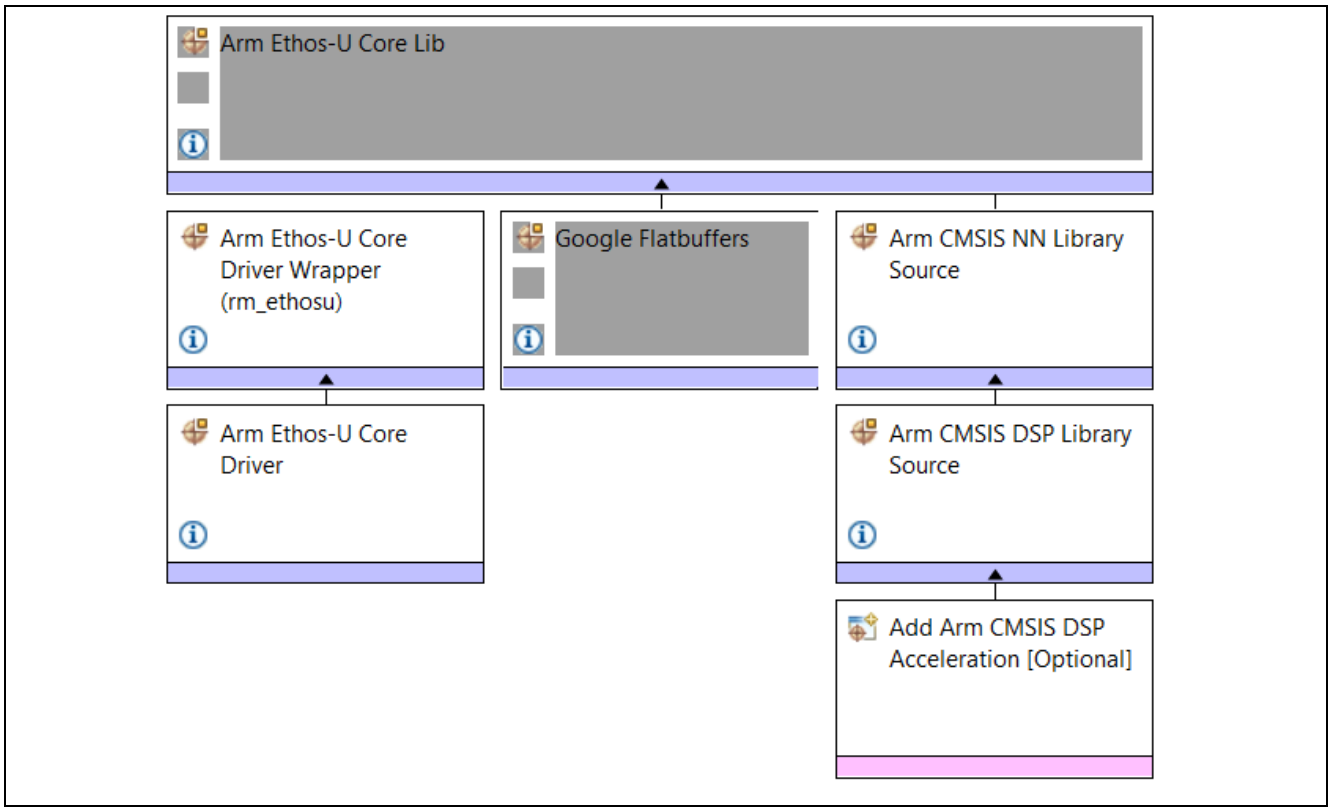

Figure 38. Example of Ethos Support in Renesas FSP

Refer to the following application notes for developing AI applications on RA8 Dual Core MCUs: "Reference System Design for Vision AI Design using Ethos-U NPU" document No. R11AN0995, and "Using the Ethos-U NPU with RA8 MCUs" document No. R01AN7712.

5. Application Projects

5.1 IPC - Share Memory Project

The implementation of this application project adheres to the following specifications.

CPU1 is responsible for managing the user interface and logging data to the terminal. CPU0 will handle real-time control and sensor data processing.

The task distribution across both cores is depicted in Figure 39.

The peripheral resources utilized in this application project are listed in Table 1 below.

Table 1. List of the resources used in the project

CPU0	CPU1
Inter-Processor Communication (IPC0 & IPC1)	Inter-Processor Communication (IPC0 & IPC1)
Hardware Semaphore	Hardware Semaphore
Real-Time-Clock control	Serial communication interface UART
12bit-A/D Converter – Temprature Sensor	I/O Port control
I/O Port control – User LEDs	

In this sample application, all runtime information is transmitted to the Tera Term terminal console via the SCI-UART interface, which is managed by CPU1.

Meanwhile, CPU0 is responsible for handling the real-time clock (RTC), sensor data acquisition, and user LED control.

Figure 40 shows the functionality of this application.

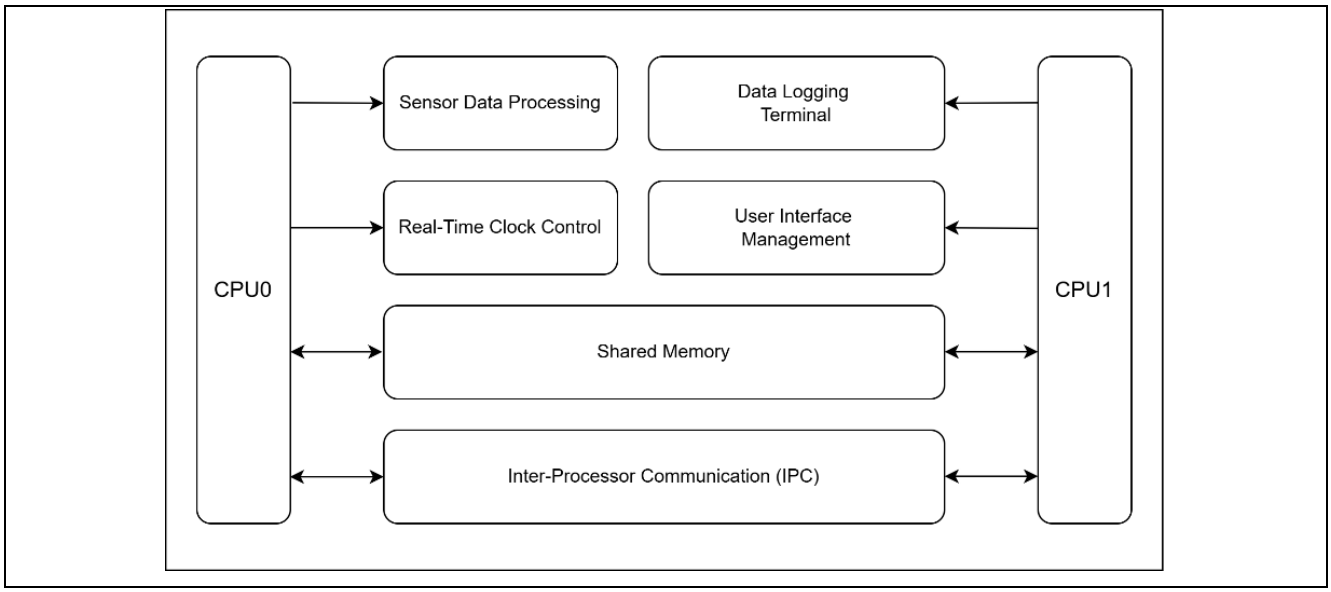

Figure 39. Task Partition on Dual Core BareMetal Example

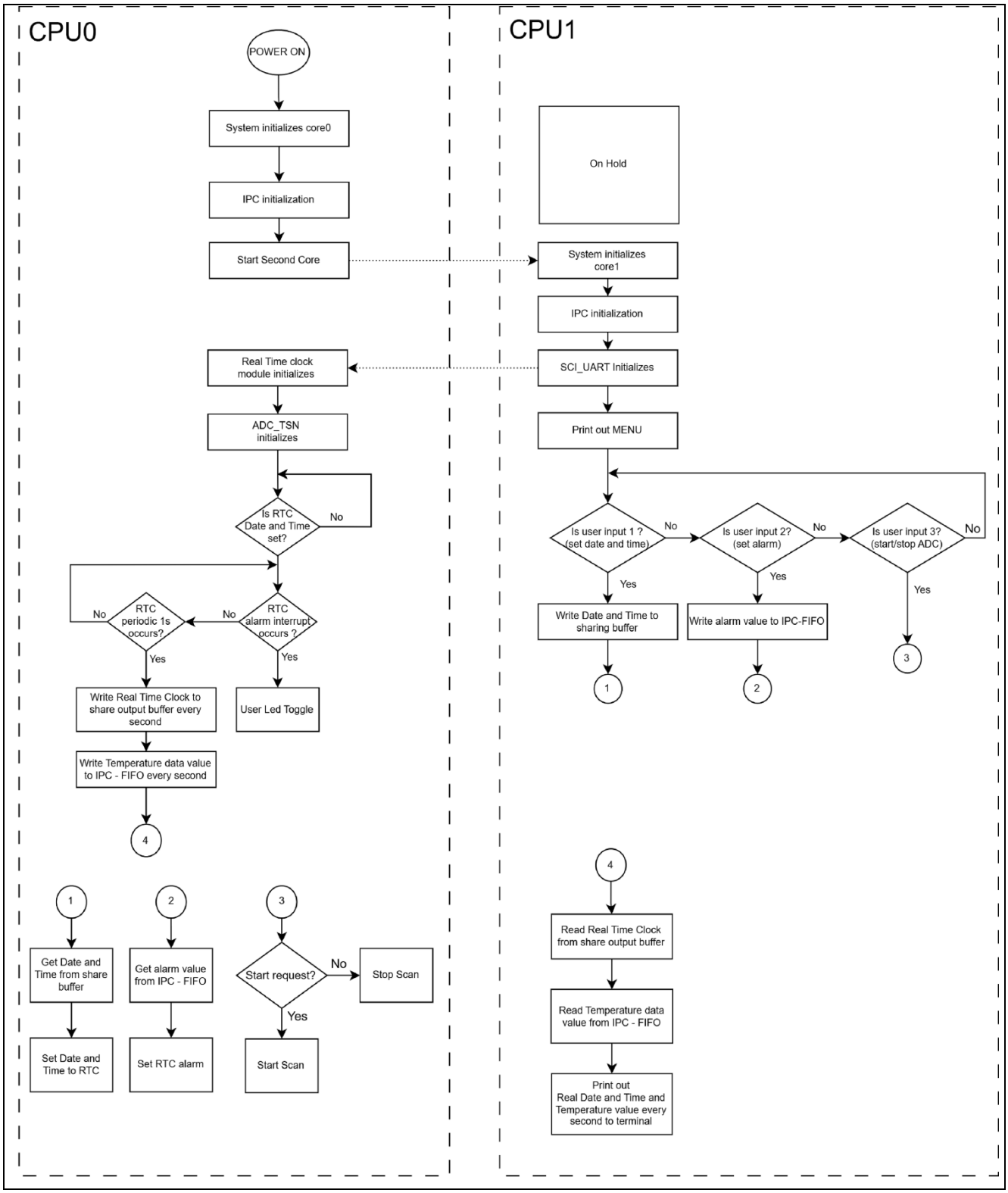

Figure 40. Application Using IPC/semaphore and shared-memory Feature.

Table 2 summarizes the IPC and hardware semaphore APIs utilized in this application. These APIs facilitate inter-processor communication and synchronization between the processors in the dual-core system.

Table 2. Inter Processor Comunication APIs

Functions	Description
R_IPC_Open	Configure an IPC instance
R_BSP_IpcNmiEnable	Assign the user callback for IPC-NMI
R_IPC_MessageSend	Send the message to IPC-Message FIFO
R_IPC_EventGenerate	Generate IPC maskable interrupt to another core
R_BSP_IpcSemaphoreTake	Take hardware semaphore
R_BSP_IpcSemaphoreGive	Release hardware semaphore
R_BSP_IpcNmiRequestSet	Trigger non-maskable interrupt to another core

5.1.1 Implement Inter-Processor Communication in Application.

Inter-Processor Communication (IPC) facilitates both hardware resource sharing and data exchange between two processors within the same CPU system.

The IPC module supports up to 16 hardware semaphores, enabling efficient synchronization between cores.

Additionally, IPC is capable of generating interrupt-driven events to support inter-processor signaling, including both maskable and non-maskable interrupts (NMIs).

Figure 41 and Figure 42 illustrate the implementation of IPC maskable interrupts and IPC non-maskable interrupts in an application.

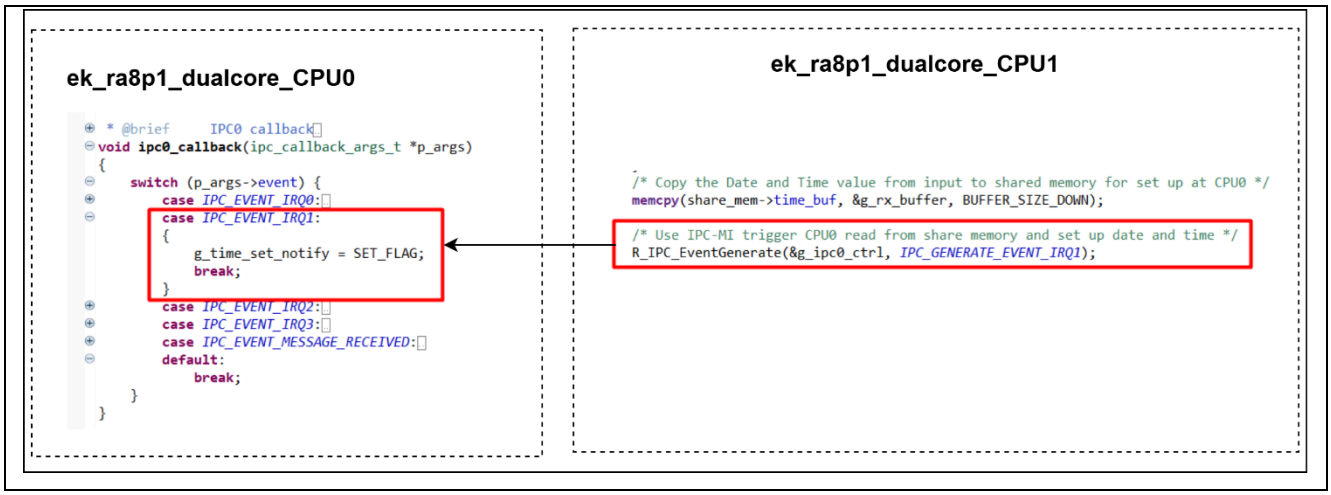

Figure 41. Example of IPC Maskable Interrupt Sample Application

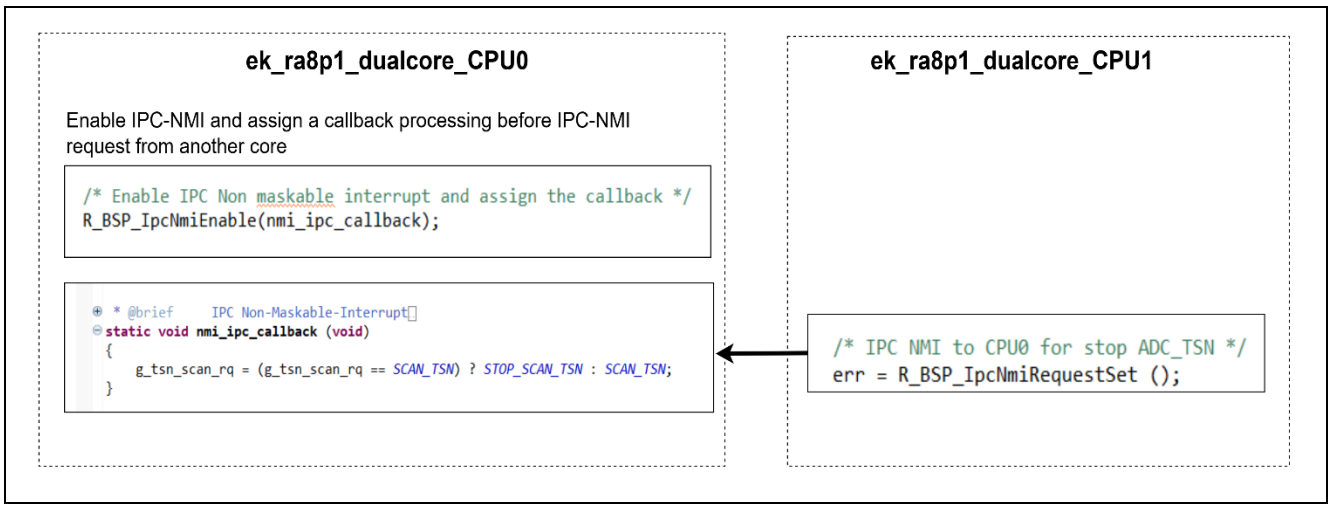

Figure 42. Example of IPC Non-Maskable Interrupt Sample Application.

In this application, one-way FIFOs are also used to exchange simple data between the two CPUs. Specifically, temperature data is sent from CPU0 to CPU1, while the user alarm setting value is sent from CPU1 to CPU0, as illustrated in Figure 43.

Figure 43. Example of IPC Message-FIFO Sample Application

5.1.2 Implement Share Memory between Two Cores in Application.

To implement a shared memory mechanism between two cores, a dedicated memory region must be explicitly defined within the solution project. In this application, the shared memory is allocated to the last 32 KB of CPU0's RAM. The configuration procedure for defining this memory region is described in the following steps:

Change the RAM size of **RAM_CPU0_S** to 0xE2000.

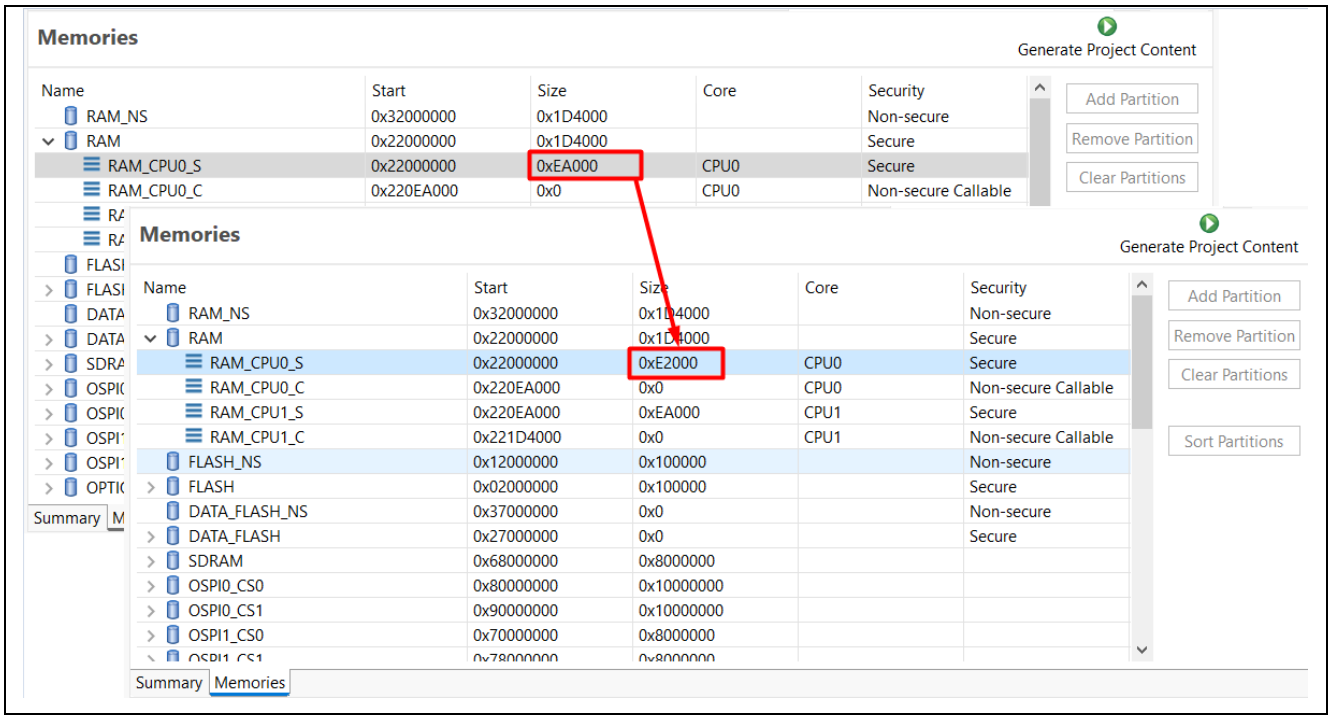

Figure 44. Example of RAM_CPU0_S Size Modification

Click to **RAM** > **Add Partition**.

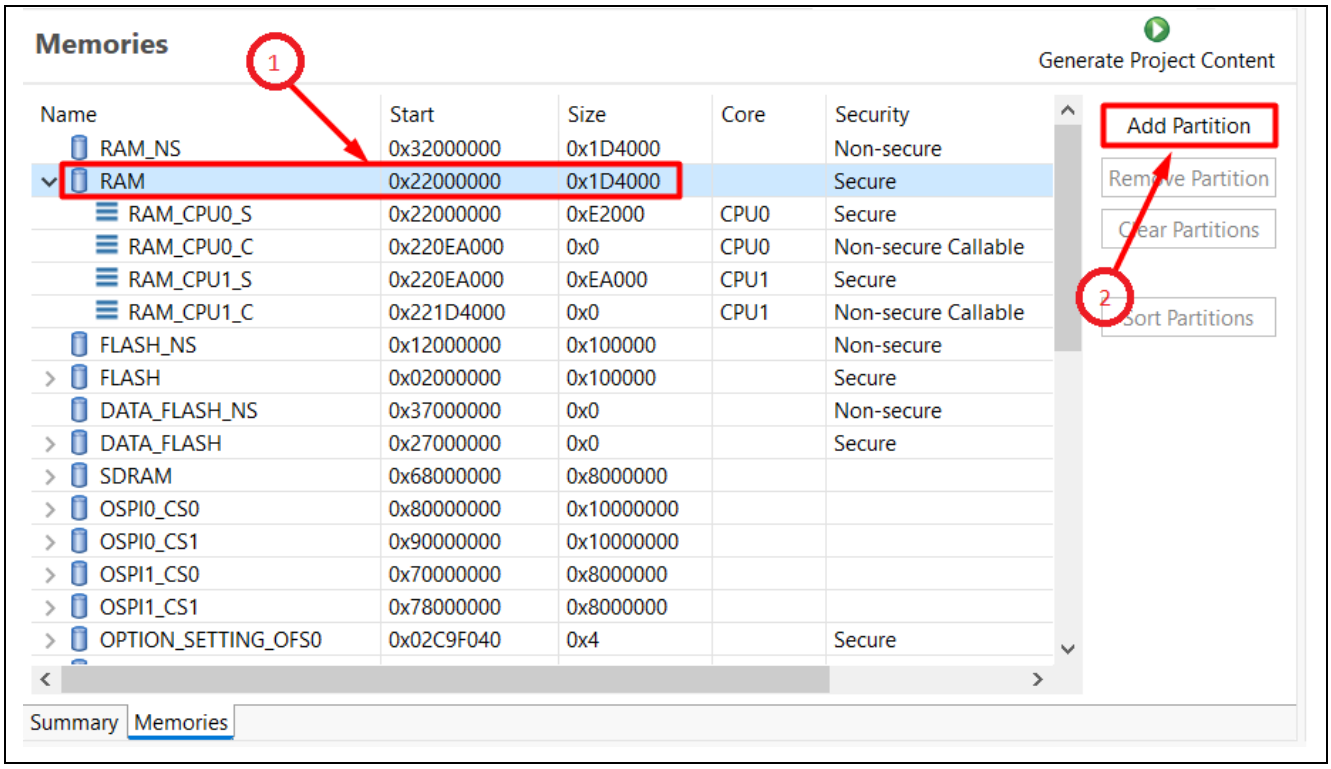

Figure 45. Example of Adding Partition for Shared Memory Region

In the **New Partition** pop-up window, as in Figure 46, complete all required fields and click OK.

For the Multicore Flat Bare-Metal Project, configure the fields as follows:

- **Name:** SHARED_MEM
- **Start:** 0x220E2000
- **Size:** 0x8000

- **Core:** CPU0
- **Security:** Secure

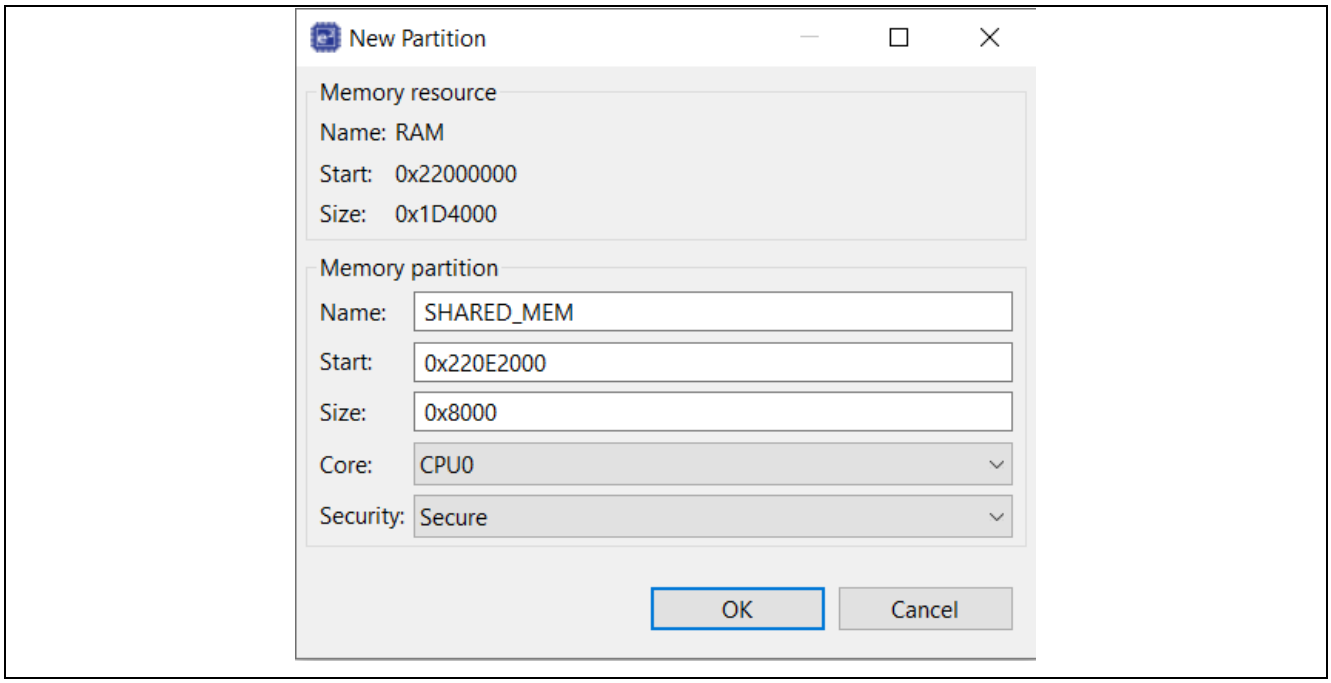

Figure 46. Example of Defining a Shared Memory Partition

If an error message appears indicating, "Partition does not follow address order" this means that the memory partitions are not arranged in a strictly increasing address sequence.

To resolve this issue, click Sort Partitions to reorder them correctly, as illustrated in Figure 47. This ensures proper memory mapping and prevents address conflicts during the build process.

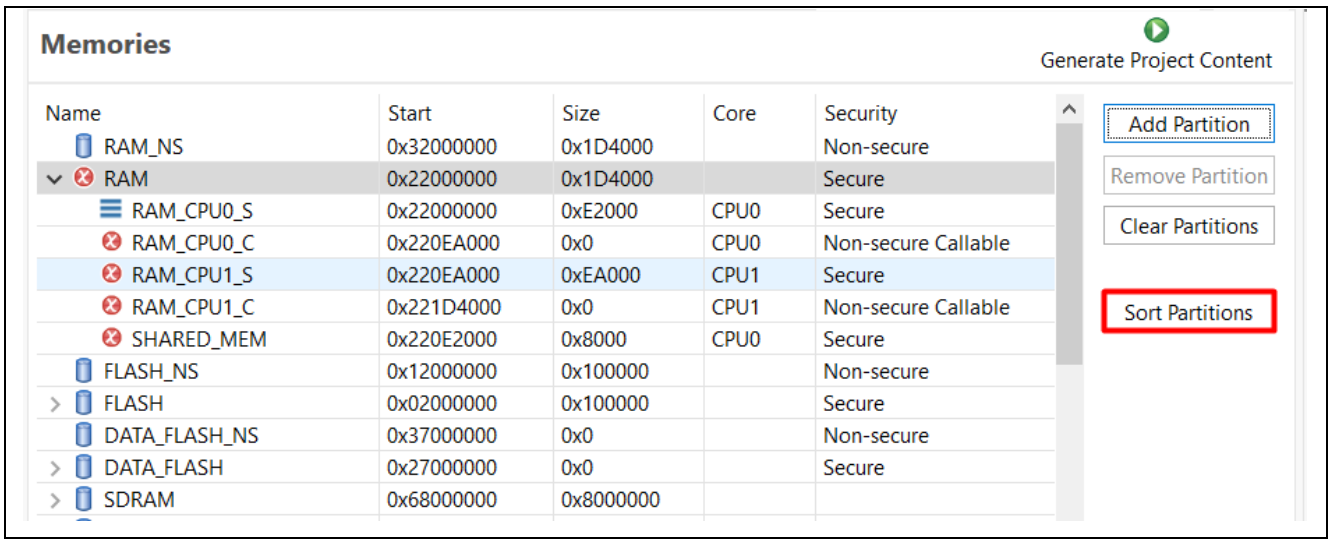

Figure 47. Example of Shortening Partitions in RAM

The successfully set up partitions are shown in Figure 48 below.

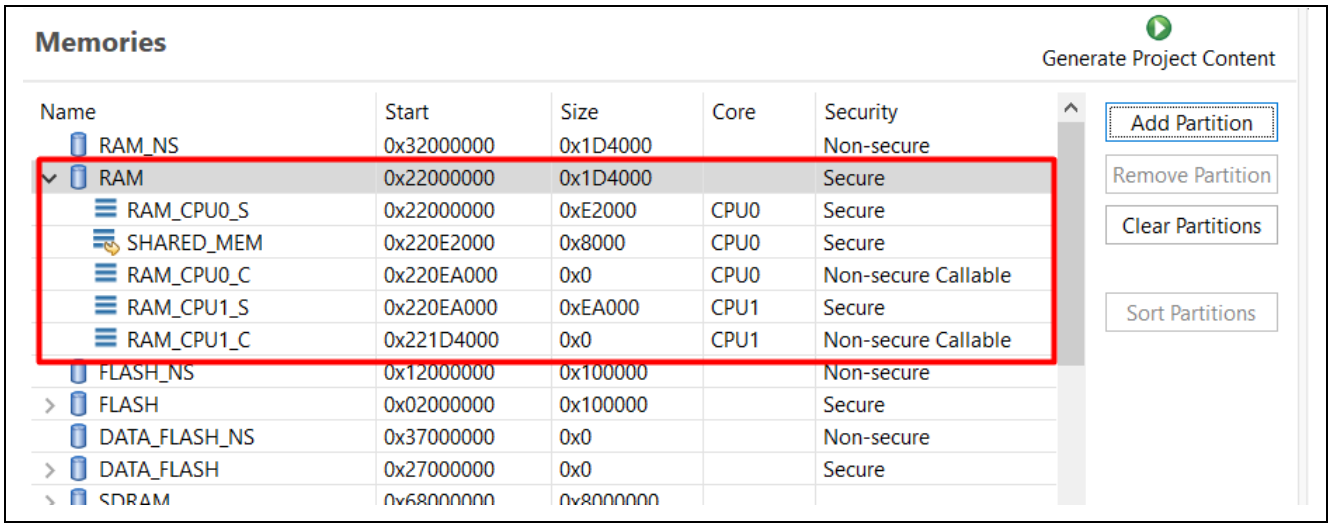

Figure 48. Example of Successful Setup of Shared Memory Partitions

After completing the configuration, right-click on the solution project and select Build Project to compile the changes.

Upon a successful build, verify the creation of the shared memory partition by double-clicking the `.sbd` file located at `<solution_project>/build/*.sbd`. This file confirms the correct partitioning of the shared memory region.

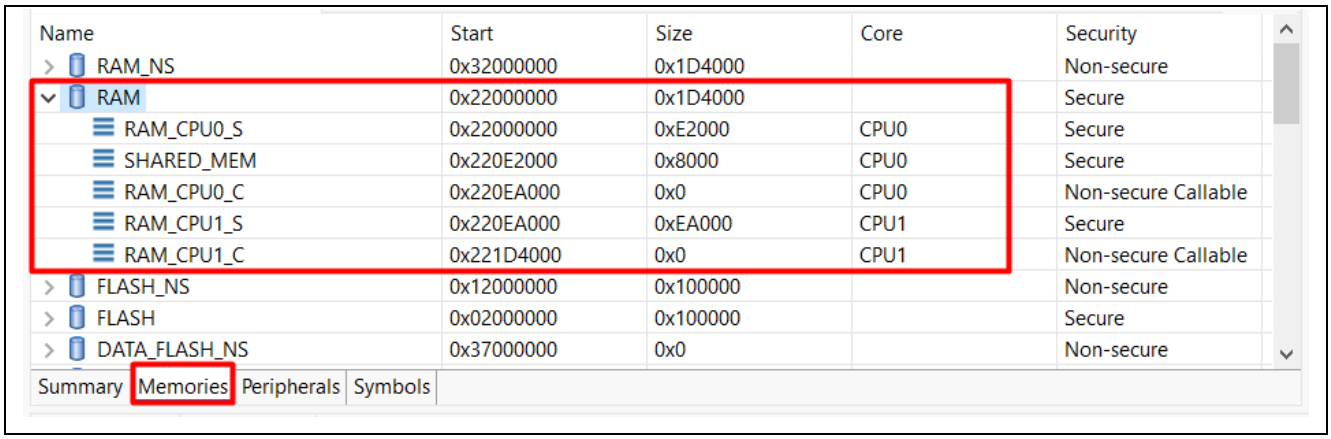

Figure 49. Example of Successful Creation of Shared Memory Partitions

From this point forward, the application can allocate buffers within the shared memory area. The implementation process is illustrated in Figure 50. This configuration guarantees that buffers reside in the shared memory space, enabling seamless data exchange between the two cores.

```
/**********************************************************************
 * Global Variables
 **********************************************************************/
uint8_t g_periodic_irq_flag = RESET_FLAG;
uint8_t g_alarm_irq_flag = RESET_FLAG;
share_mem_t share_memory BSP_PLACE_IN_SECTION(".shared_mem") = {.buf_out = {RESET_VALUE},
                                                               .length = RESET_VALUE,
                                                               .time_buf = {RESET_VALUE}};
```

Figure 50. Example of Placing the Sharing Buffer in the Shared Memory

The fundamental principle of shared memory management is to guarantee exclusive access by a single CPU at any given time. In this application, hardware semaphores and inter-processor maskable interrupts (MIs) are used together to coordinate and arbitrate access between the two cores. This mechanism follows the control flow illustrated in Figure 20.

5.2 RTOS/IPC/Share Memory/TCM Projects

These example projects leverage the RTOS, IPC module, IO interfaces, and TCM memory with data cache enabled to facilitate communication between two CPU cores.

The application is adapted from the official Arm CMSIS-DSP FFT example to run on CPU0 (CM85 core), with computational tasks executed within TCM to maximize performance. FFT results are transmitted to the second core through an RTOS queue based on the on-chip IPC module. The second core then outputs the results via UART to a terminal interface.

When porting CMSIS-DSP examples from Arm with RA FSP, refer to application note R01AN5865 Arm® DSP Examples for detailed integration guidelines.

In this implementation, a FreeRTOS-based queue is integrated with a hardware IPC message FIFO to manage message transactions between the two cores, as illustrated in Figure **51**.

Alternatively, FreeRTOS message buffers or stream buffers can also be employed for inter-processor communication depending on the use case and data flow requirements.

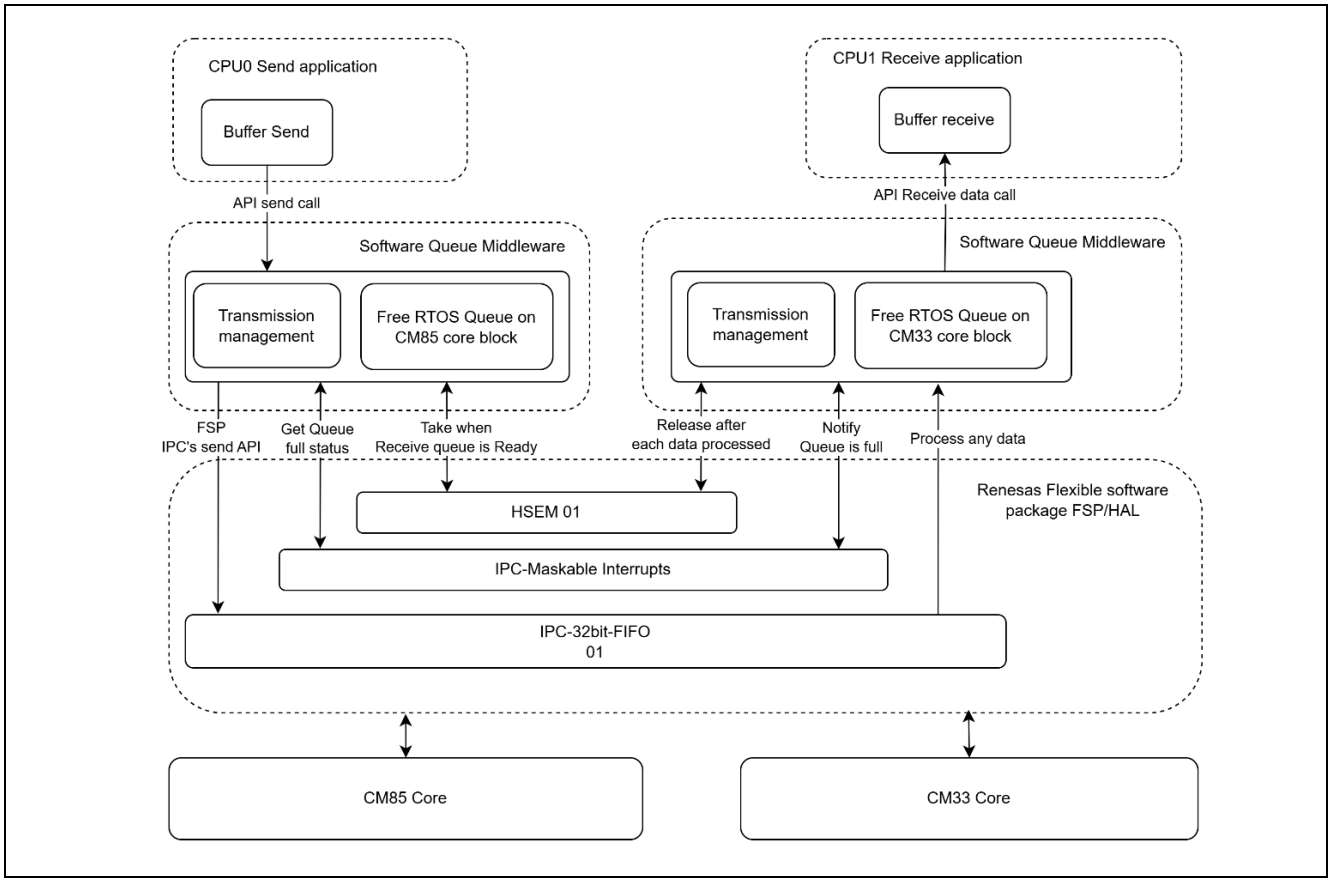

Figure 51. **FreeRTOS-based Inter-processor Messaging via IPC Hardware**

6. Verify the Multicore Flat Bare-Metal Project

The e²studio project employs a split project development model to support dual-core application development. Each core project is created as a separate one, utilizing Inter-Processor Communication (IPC) to enable efficient communication and shared memory management between the two cores.

To build and run the example application project ek_ra8p1_dualcore in e²studio, please follow the procedure outlined below.

6.1 Import The Projects

1. Launch e² studio IDE.
2. Select any workspace in Workspace launcher.
3. Close the **Welcome** window.
4. Select **File > Import**.
5. Select **Existing Projects into Workspace** from the **Import** dialog box.
6. Select archive file “ek_ra8p1_dualcore.zip” in the file named “Developing_with_RA8_Dual_Core_MCU.zip”.
7. Select solution project and developed project samples on each core as shown below, click **Finish**

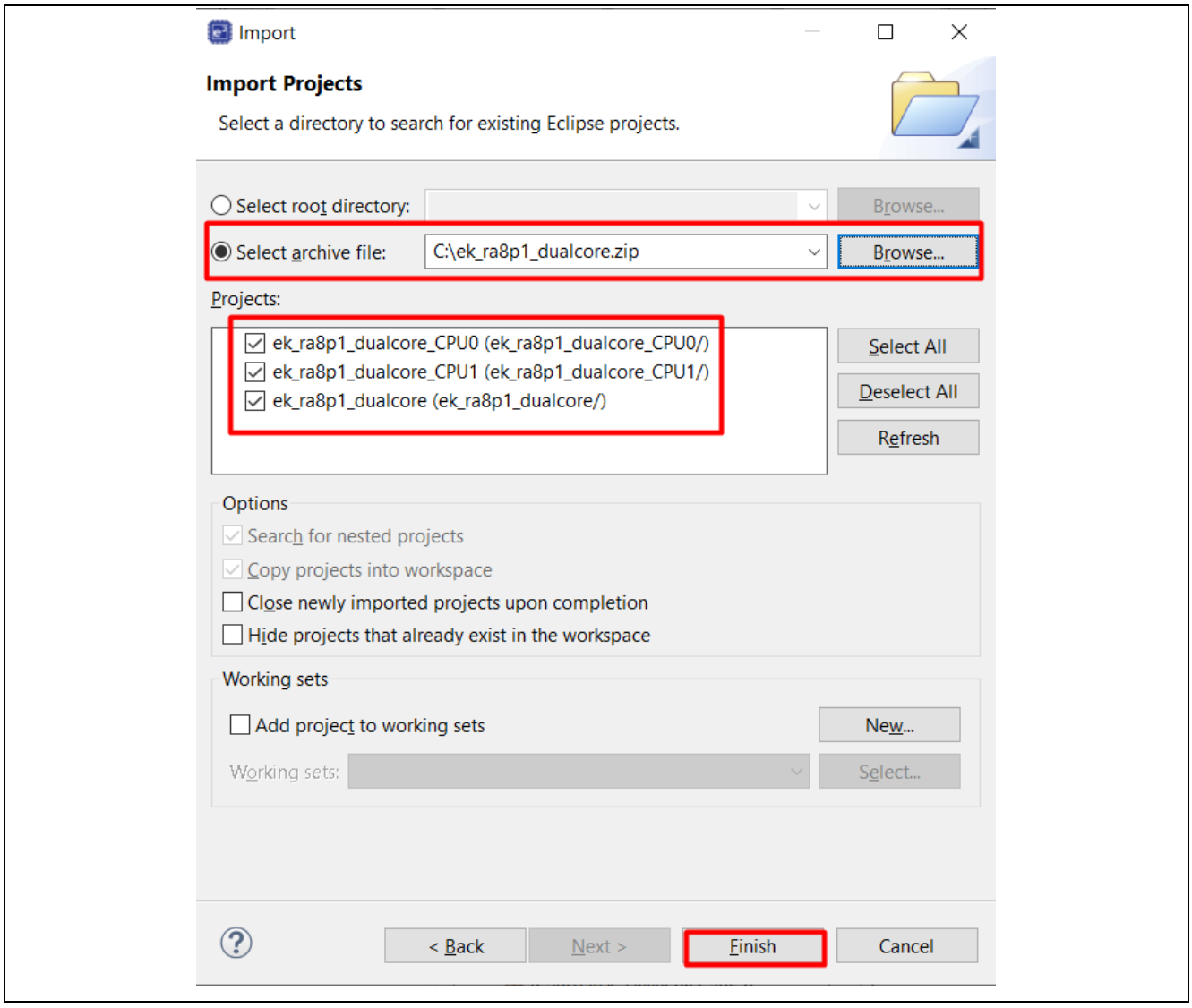

Figure 52. Example of Importing Projects into Workspace.

6.2 Build Projects

The easiest method is to right-click the solution project **ek_ra8p1_dualcore** and select **Build Project**. When building the solution project, both CPU projects are built automatically in sequence: CPU0 is built first, followed by CPU1.

If building the projects individually, ensure that the CPU0 project is built first, and then build the CPU1 project.

6.2.1 Compile Project Developed on CM85 Core

Double-click on `configuration.xml` located in the `ek_ra8p1_dualcore_CPU0` project > Click “Generate Project Content.

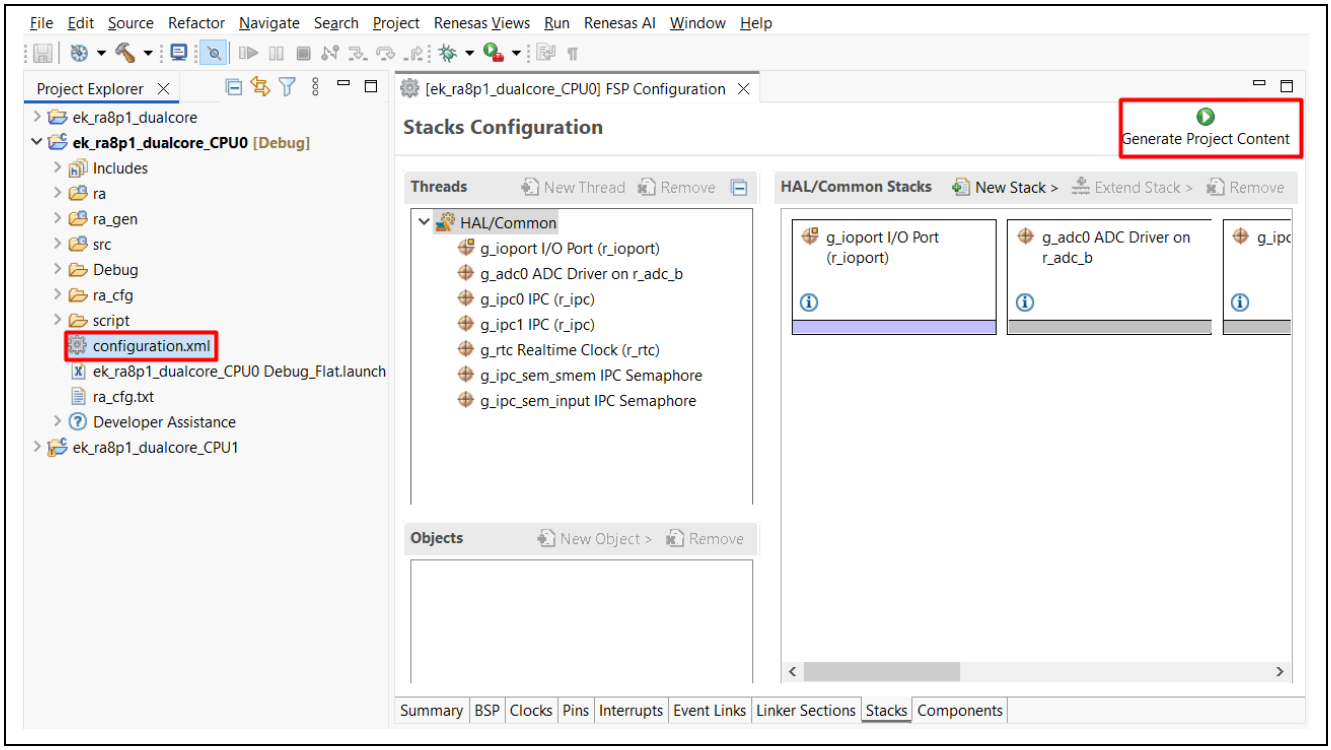

Figure 53. Example of Generating Project Content on CPU0 Project

After generating the project content, right-click on ek_ra8p1_dualcore_CPU0, then select Build Project to compile the CPU0 core application.

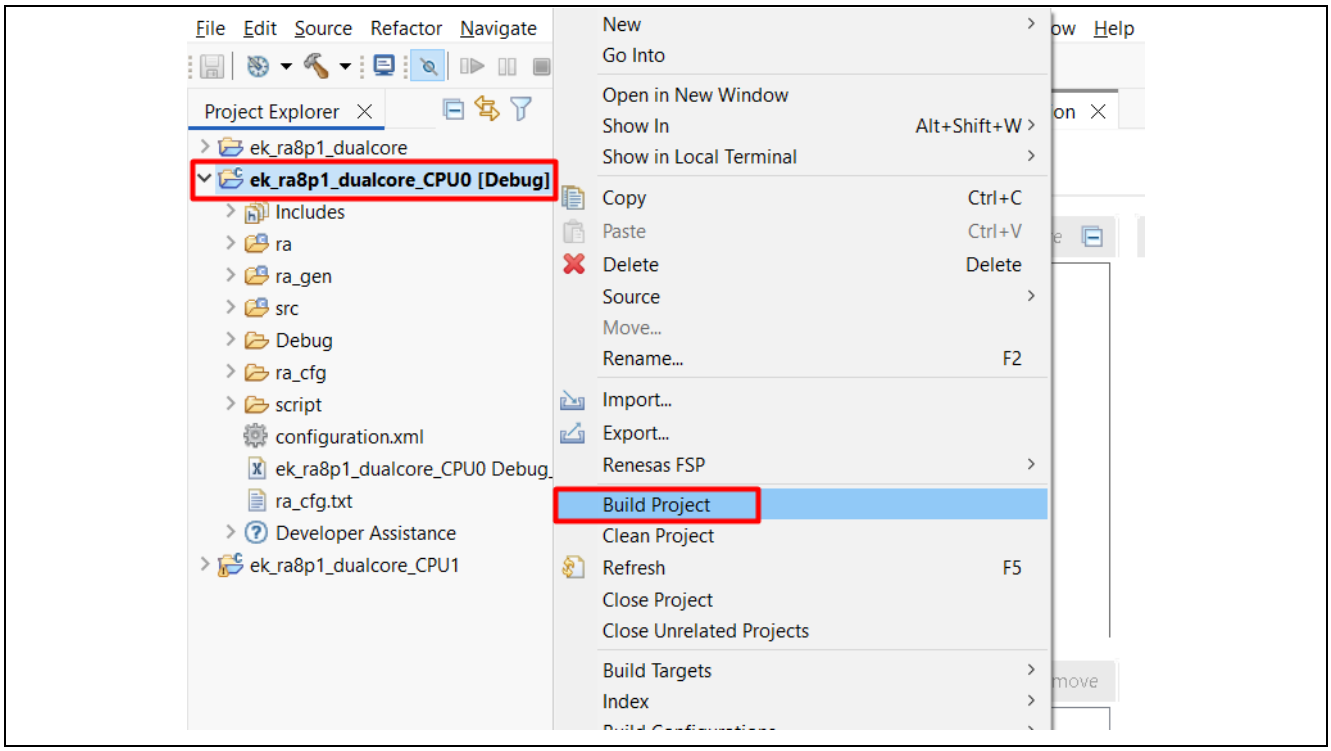

Figure 54. Example of Building CPU0 Dual-Core Project

Verify that the build completes successfully by checking the output in the Build Log console.

6.2.2 Compile Project Developed on CM33 Core

In the `ek_ra8p1_dualcore_CPU1` project, double-click on `configuration.xml`, then click `Generate Project Content` to apply the configuration settings for the CPU1 core.

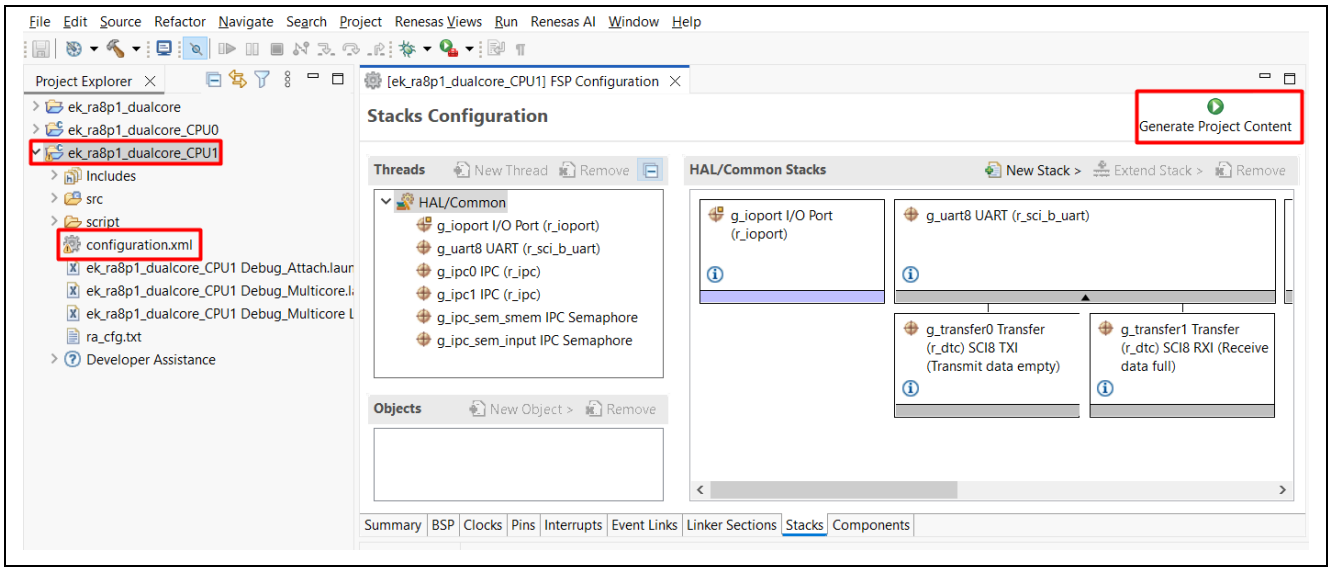

Figure 55. Example of Generate Project Content on CPU1 Project

After the content is generated, right-click on `ek_ra8p1_dualcore_CPU1`, then select `Build Project` to compile the CPU1 core application.

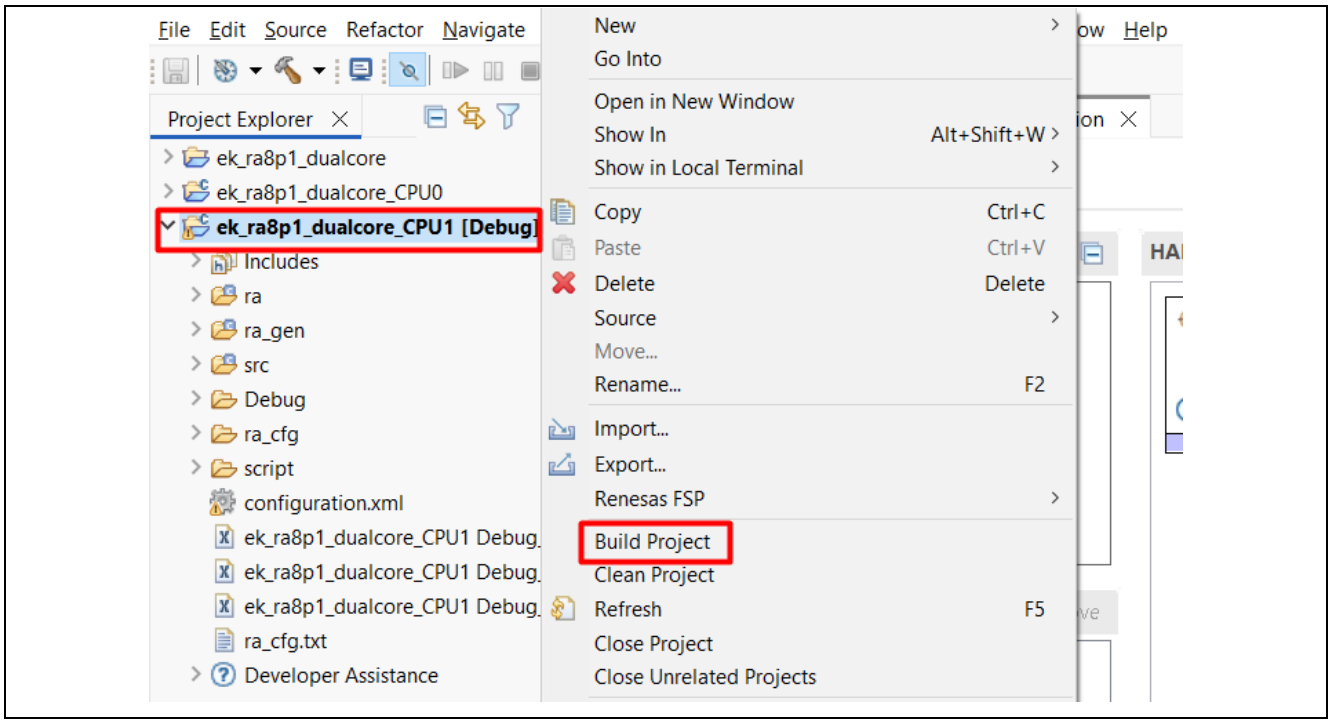

Figure 56. Example of Build CPU1 Dual Core Project

Ensure that the build completes successfully by checking the output in the Build Log console.

Alternatively, the build process for both cores can be executed through the solution project. Right-click on the `ek_ra8p1_dualcore` solution project and select `Build Project`, as shown in Figure 57. This command will sequentially build all projects within the solution, following the order: CPU0 → CPU1.

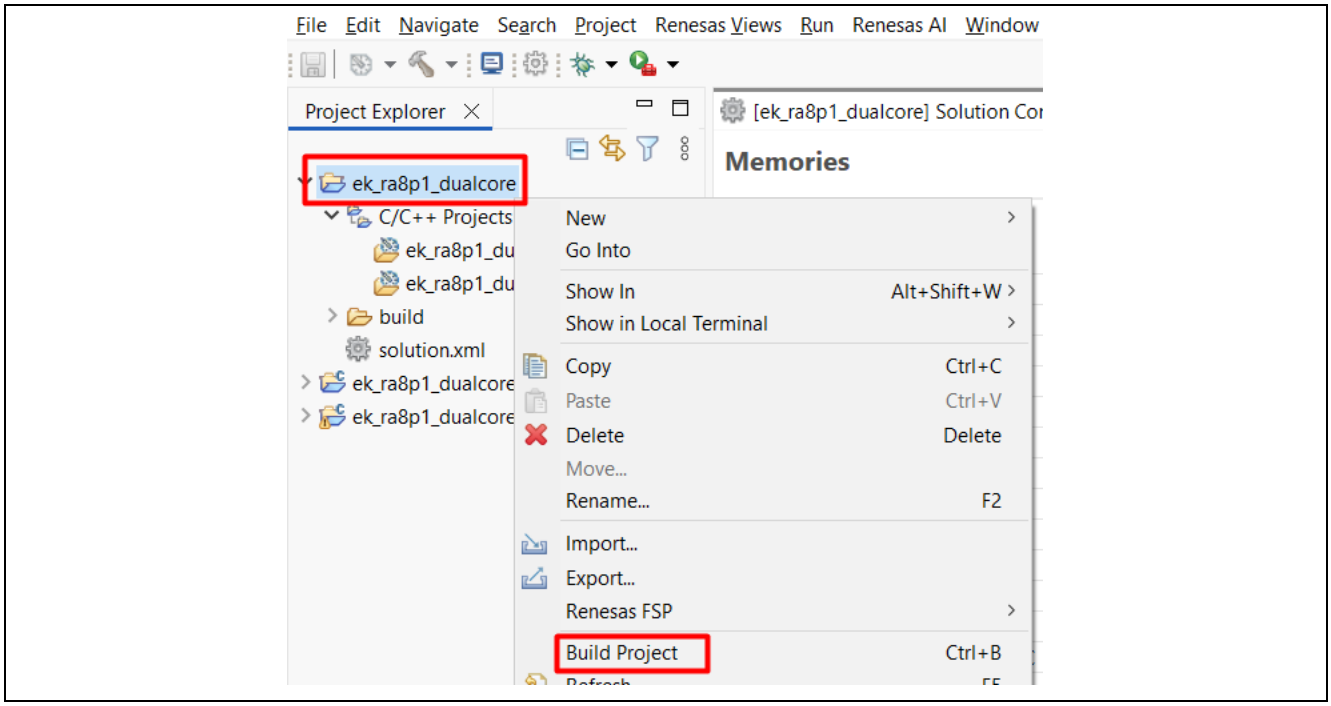

Figure 57. Example of Build RA Solution Project Dual Core

6.3 Download and Run Projects

As described in the debug settings in Section 3.2, the device must be initialized in the OEM_PL2 state, and TrustZone boundary settings are not required for this configuration.

To start a dual-core debug session, open the Debug Configurations dialog as shown in Figure 58.

Select "ek_ra8p1_dualcore_CPU1 Debug_Multicore Launch Group," then click Debug as illustrated in Figure 59 to simultaneously launch debug sessions for both cores.

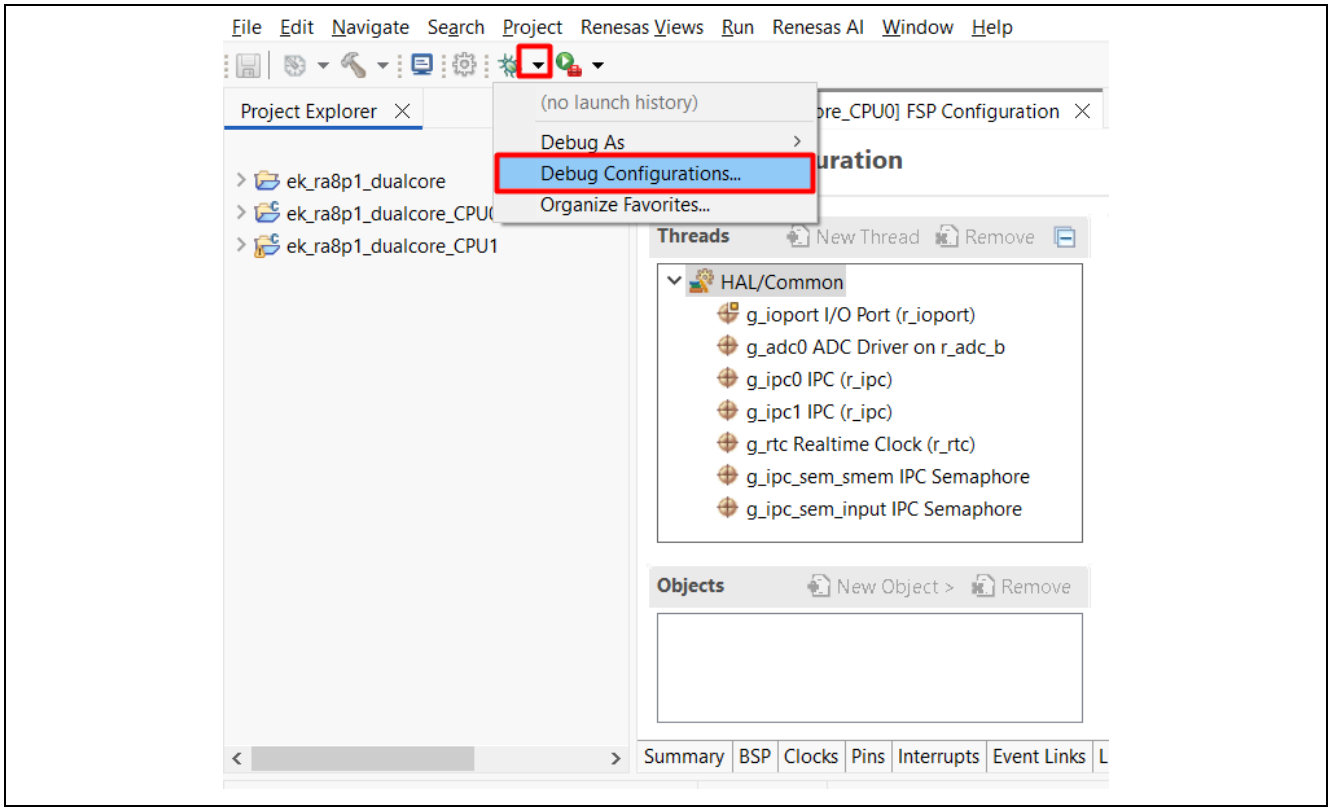

Figure 58. Example of Select Debug configuration on Toolbars

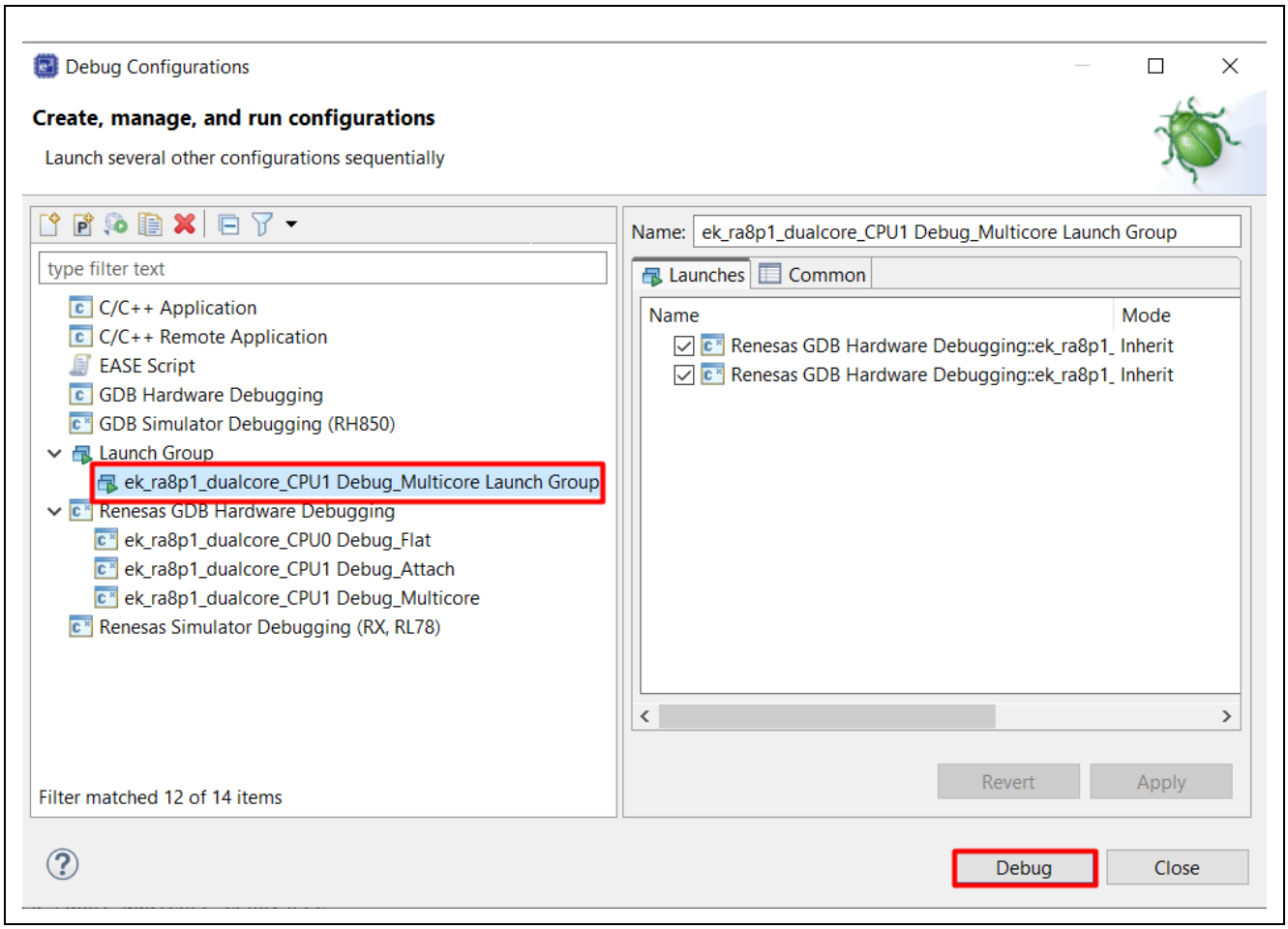

Figure 59. Example of Debug with Multicore Launch Group

The debugging information will appear on the debug console. Before clicking Resume to start application execution, it is necessary to configure the Tera Term terminal to monitor the output and verify application behavior.

To set up the terminal, follow the steps below:

1. Open Tera Term (Figure 60).
2. Select the appropriate J-Link CDC UART Port from the serial connection options to establish communication with the target device (Figure 61).
3. Enable Local Echo by navigating to Setup > Terminal and checking the Local Echo option (Figure 61).
4. Configure the terminal settings by selecting Setup > Serial Port, then adjust the baud rate and other serial parameters according to the project requirements (Figure 62).

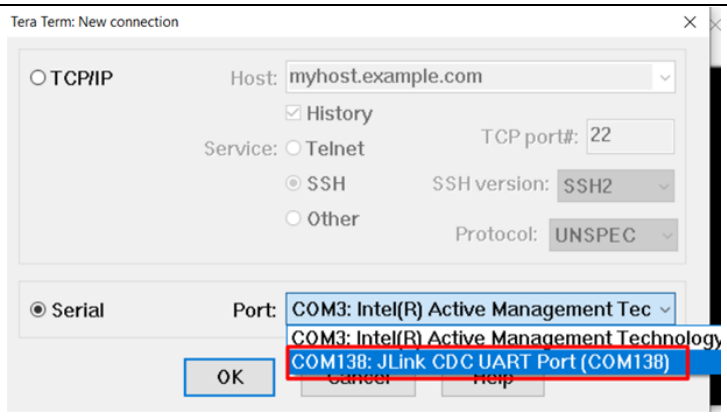

Figure 60. Example of Connect to Tera Term terminal

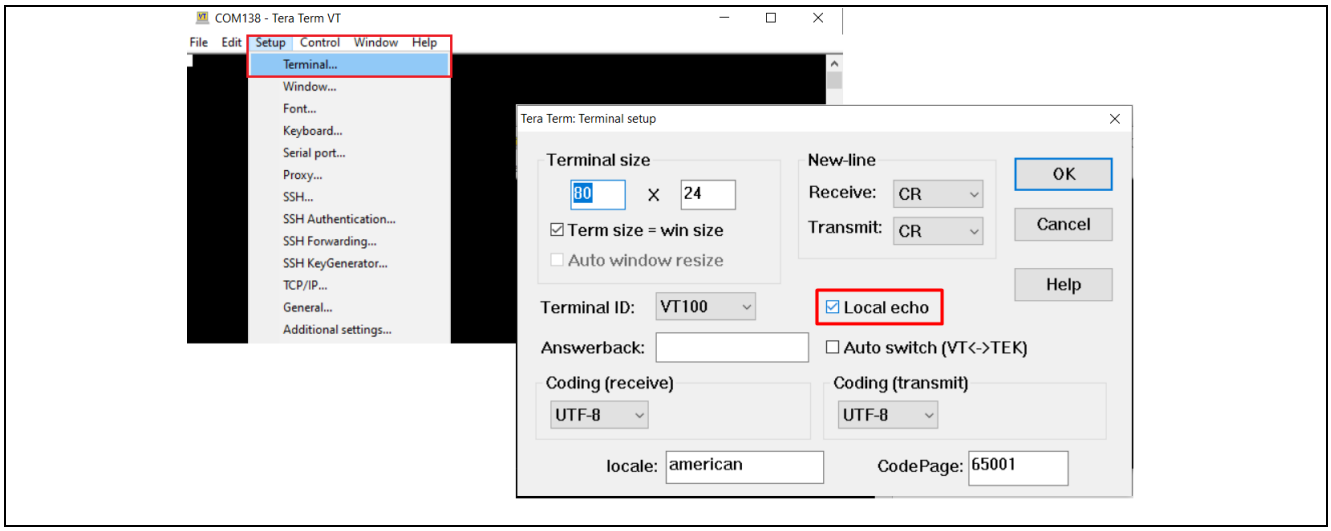

Figure 61. Example of Enable Local echo Tera Term Terminal

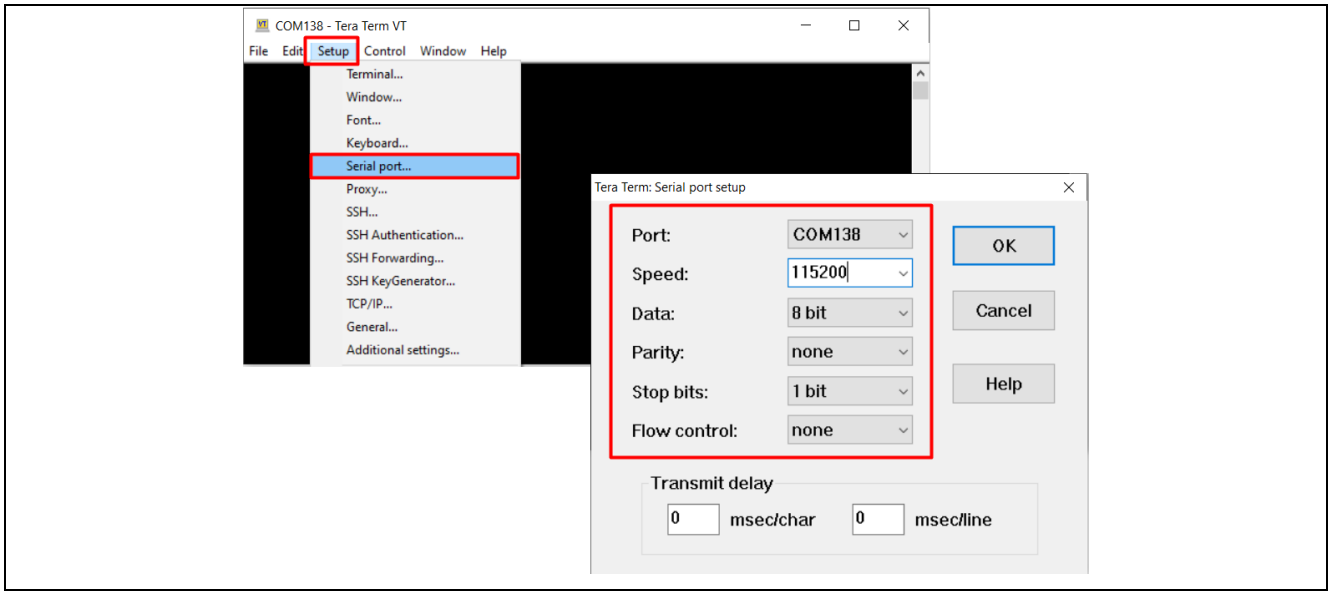

Figure 62. Example of Serial Port Set Up in Tera Term Terminal

After completing the TeraTerm terminal setup, return to e²studio and click Resume three times to start the application execution.

The initial screen layout should appear as shown in Figure 63.

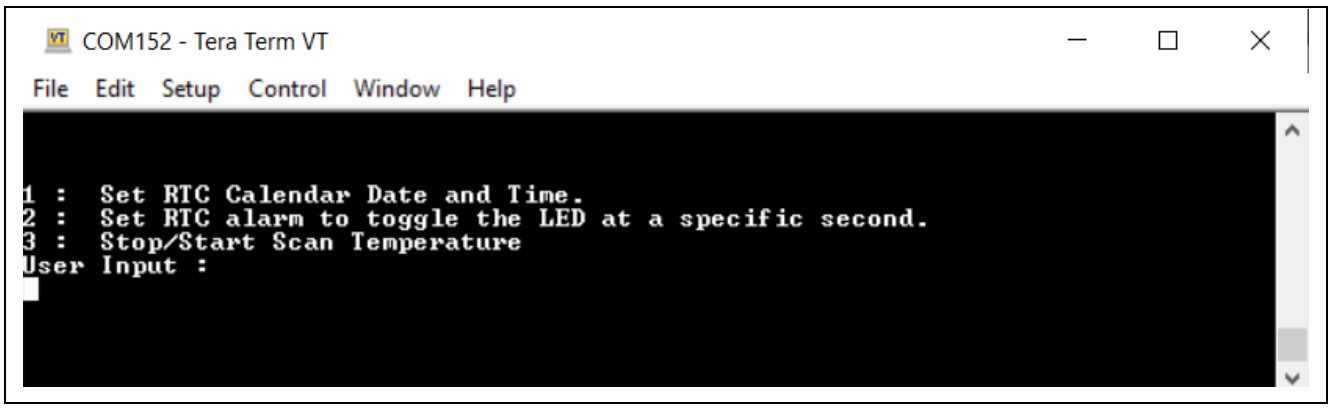

Figure 63. Initial Terminal Layout of the Application

Depending on the selected user input options, the terminal layout will dynamically change. Figure 64 illustrates the various terminal layouts corresponding to different user inputs.

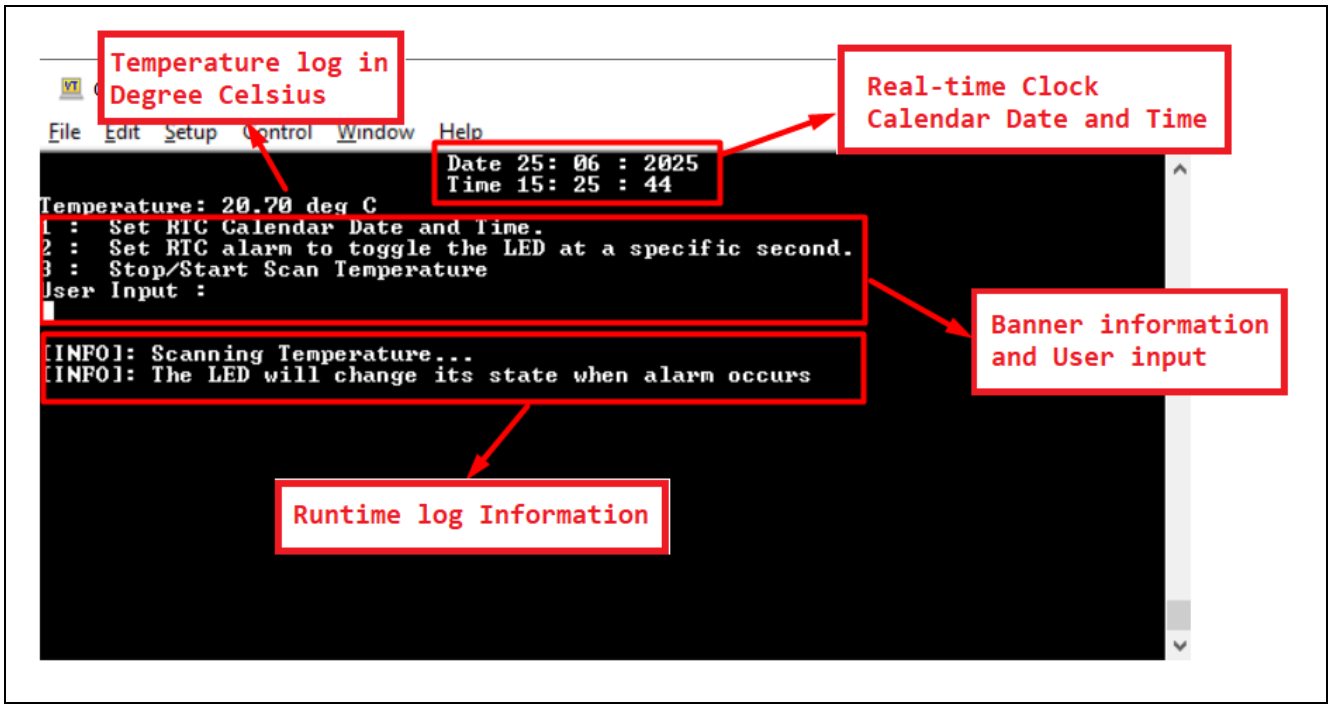

Figure 64. Overview of Application Terminal Layout

Note: The RTC alarm will be activated when the alarm setting for seconds matches the second counter in the RTC.

To operate the application project, follow this sequence: option 1 → option 2 → option 3. This sequence is necessary because both the temperature printing and the RTC alarm are triggered by the RTC timer.

7. Verify the FreeRTOS-Based Projects

7.1 Import The Projects

1. Launch e² studio IDE.
2. Select any workspace in Workspace launcher.
3. Close the **Welcome** window.
4. Select **File > Import**.
5. Select **Existing Projects into Workspace** from the **Import** dialog box.
6. Select archive file “dsp_example_dual_core.zip” in the file named “Developing_with_RA8_Dual_Core_MCU.zip”.
7. Select solution project and developed project samples on each core as shown below, click **Finish.**

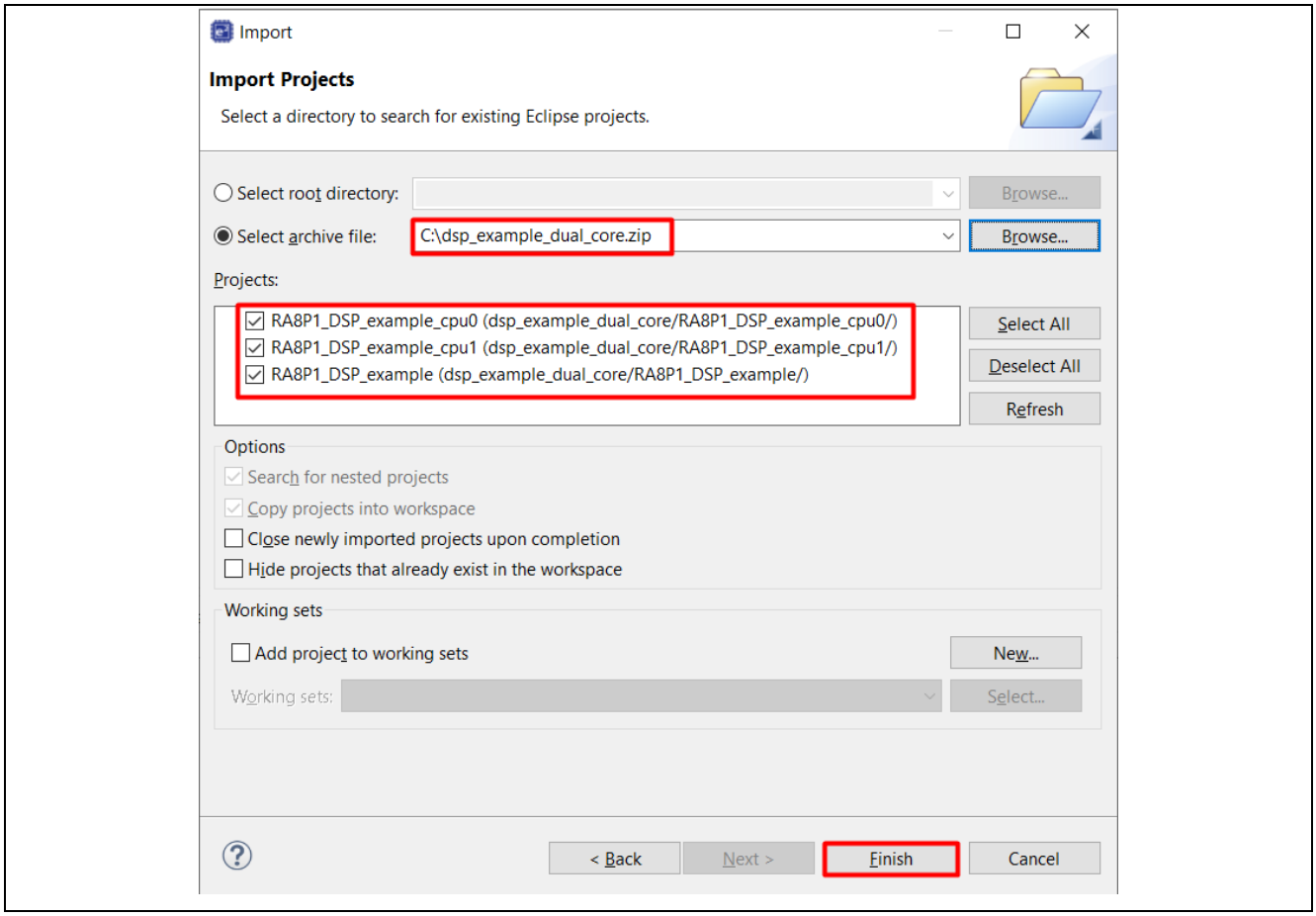

Figure 65. Example of Build DSP Example Project.

7.2 Build Projects

Build the projects sequentially in the order CPU0 → CPU1, as detailed in Section 6.2. Alternatively, right-click on the solution project and select Build Project to compile all contained projects at once, as illustrated in Figure 66.

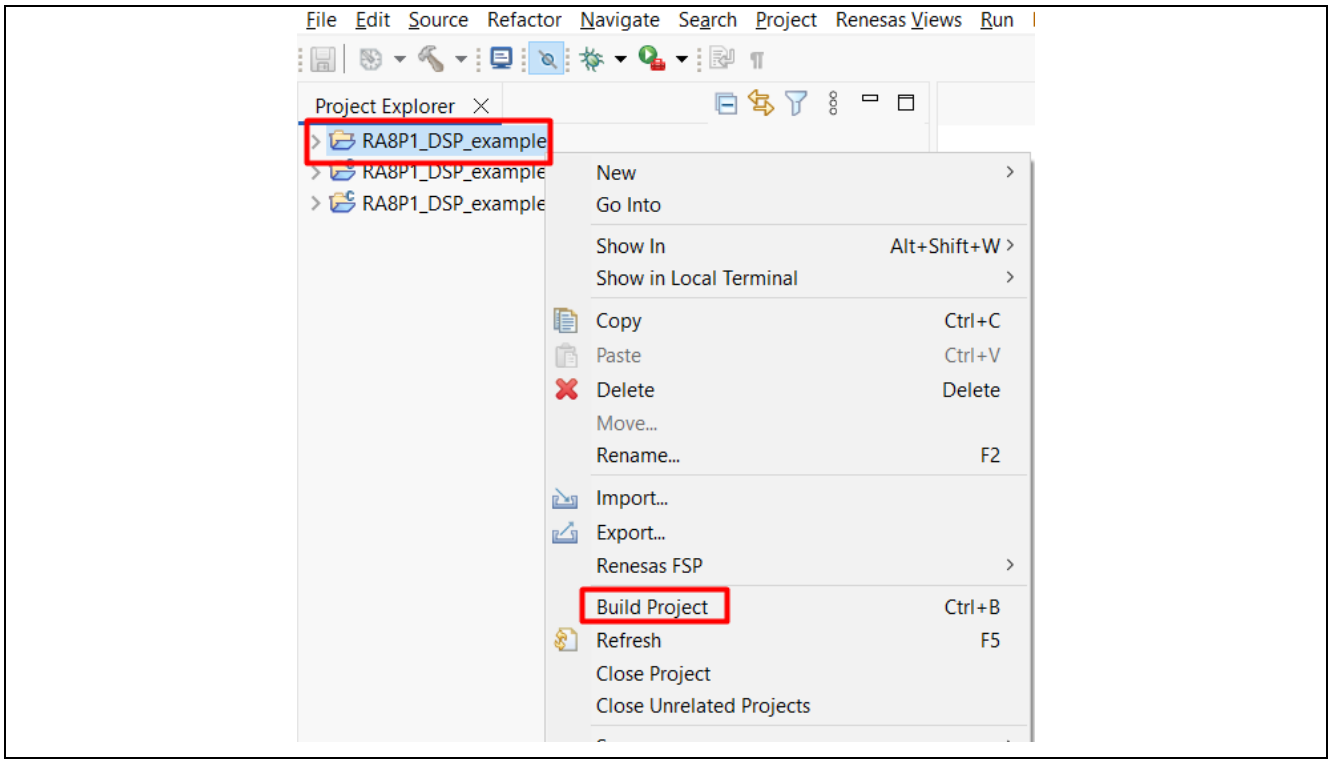

Figure 66. Example of Build DSP Example Project.

Ensure that the build completes successfully for both CPU0 and CPU1 projects by verifying the build status in the Build Log console.

7.3 Download and Run Projects

As described in Section 3.2, the device must be initialized in the OEM_PL2 state, with no TrustZone boundary settings required.

To start debugging both cores simultaneously:

1. Open Debug Configurations as shown in Figure 67.
2. Select “RA8P1_DSP_example_cpu1 Debug_Multicore Launch Group.” as shown in Figure 68.
3. Click Debug to launch the dual-core debug session.

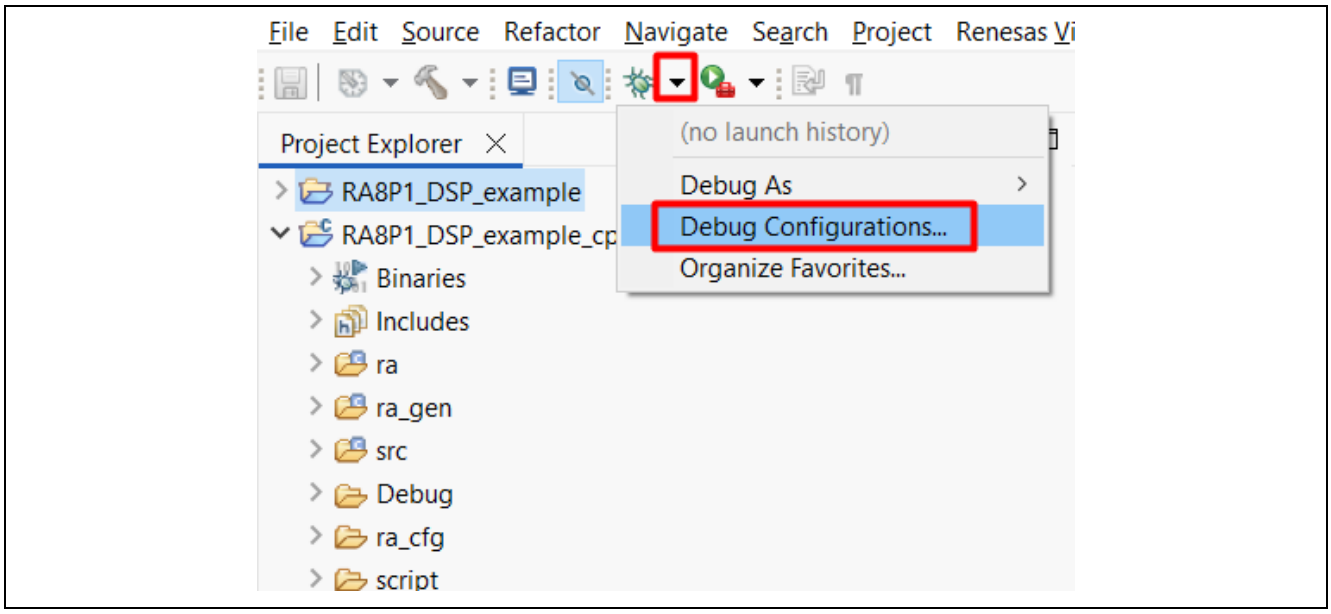

Figure 67. Example of Open Debug Configuration in DSP Example

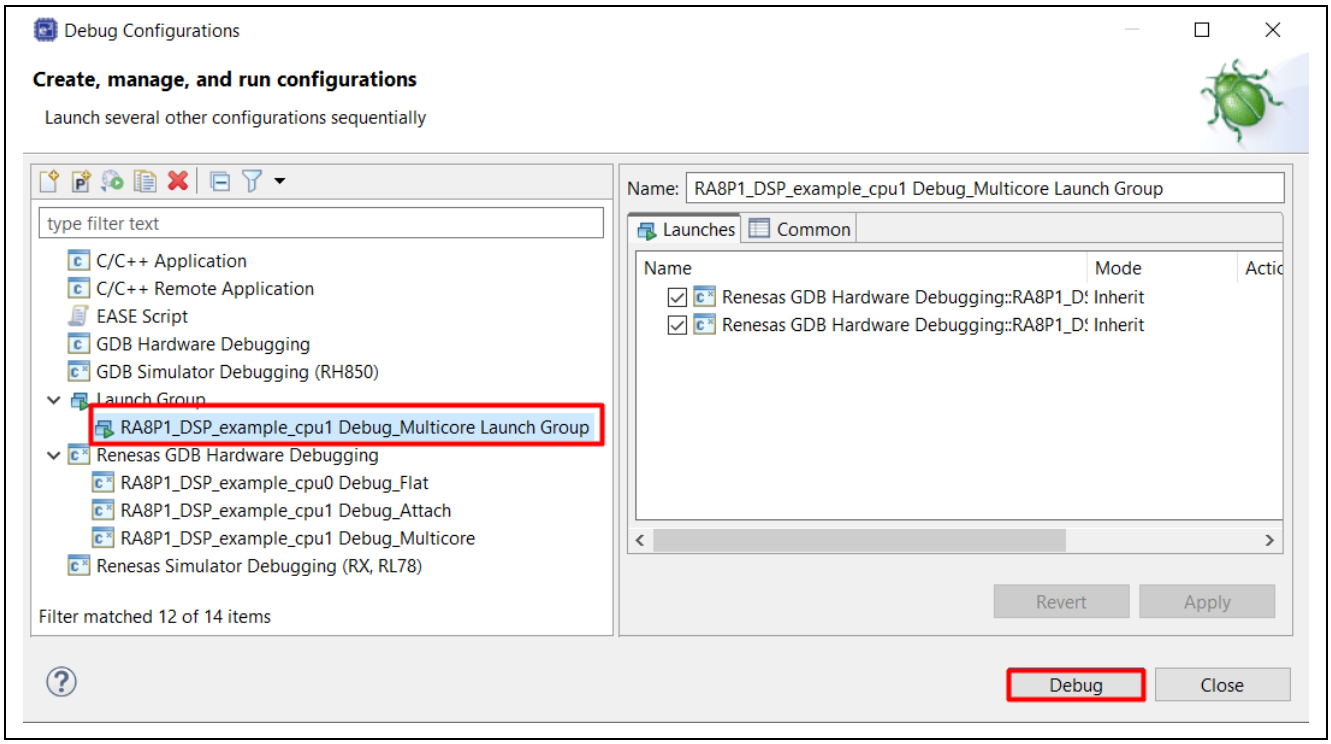

Figure 68. Example of Debug DSP Dual Core DSP Example

Open the serial terminal and connect to the J-Link CDC UART Port with the following settings: Baud rate 115200 bps, 8-bit data, none parity, 1 stop bit, and no flow control. The Figure 69 and Figure 70 illustrate the connection process and configuration steps using Tera Term terminal.

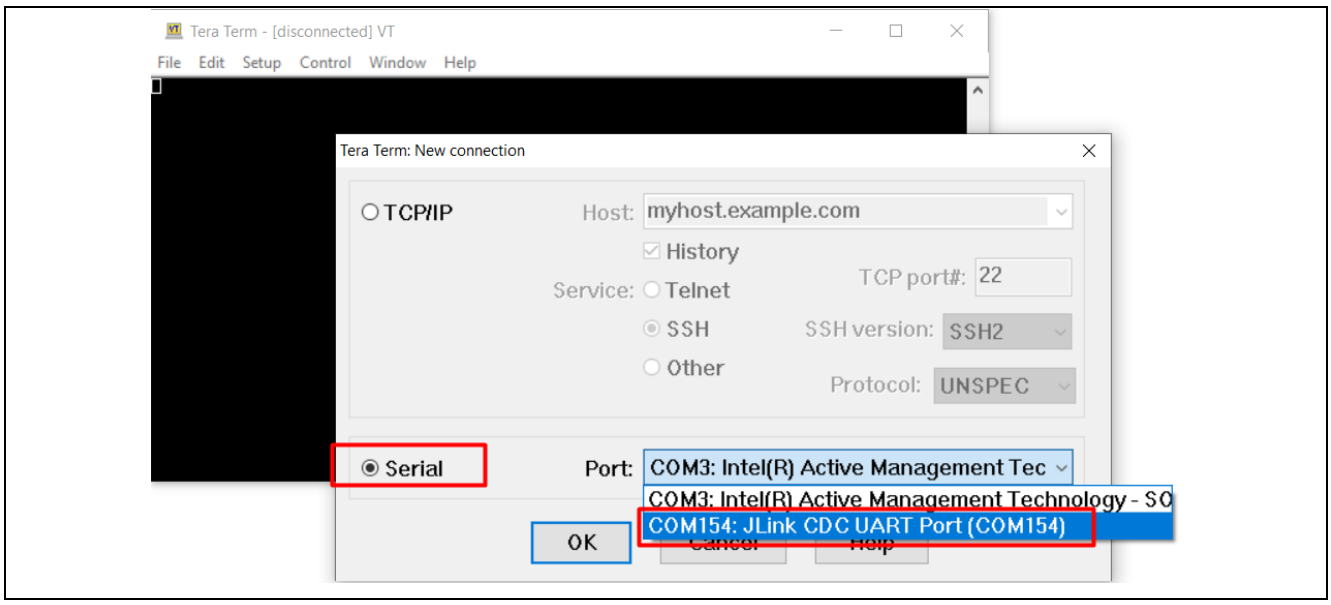

Figure 69. Example of Connect to JLink CDC UART Port

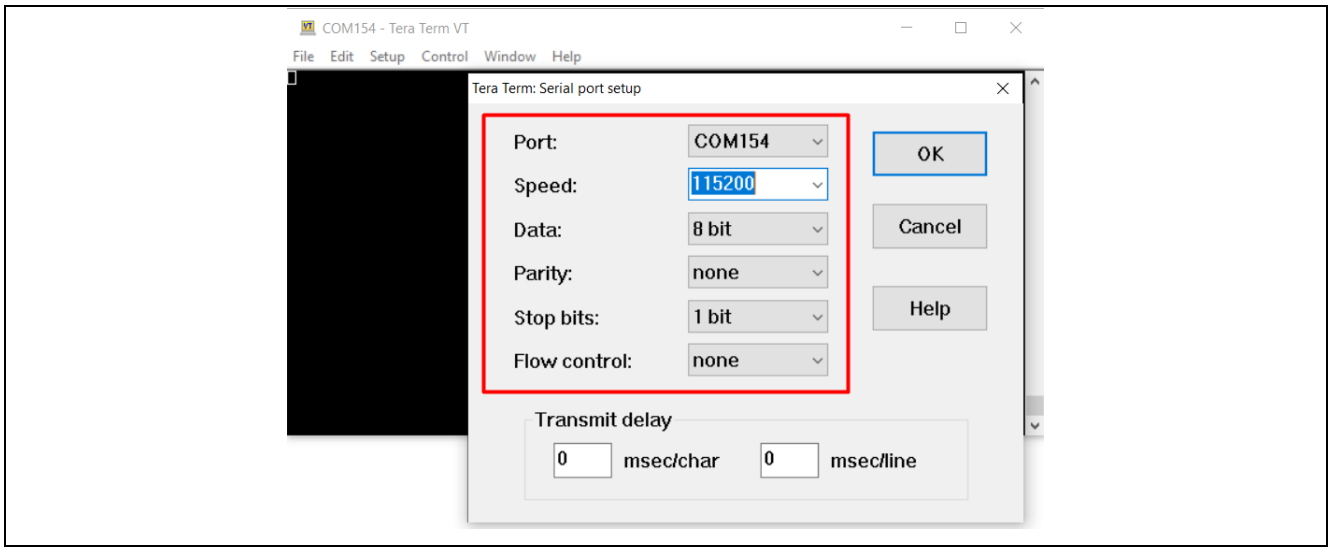

Figure 70. Example of Set Up Tera Term Serial Terminal

Return to e² studio and press Resume three times to execute the application across both cores. Upon completion, the CMSIS-DSP FFT example will output its status to the terminal, as illustrated in Figure 71.

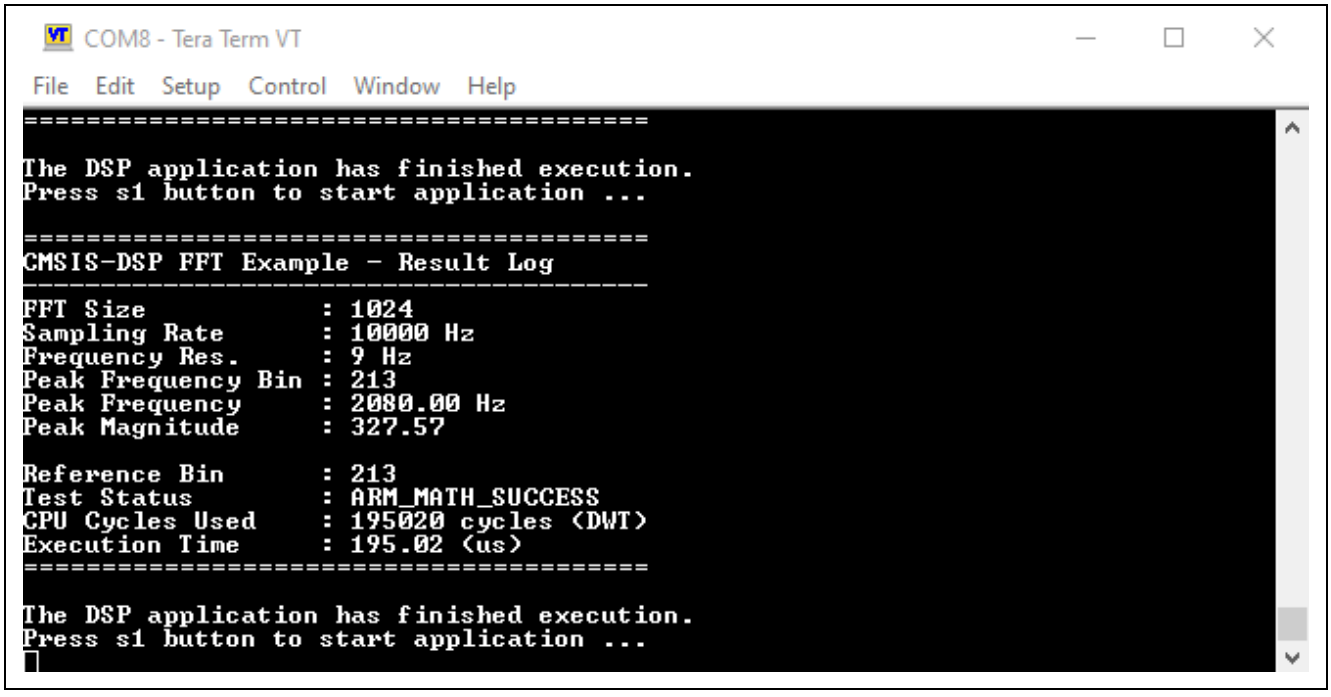

Figure 71. The Successfully Result Log CMSIS-DSP FFT Example

8. Migrate Dual Core Example Projects to a New Dual Core MCU

The steps described in this chapter illustrate the migration process of the `dsp_example_dual_core` application on RA8P1 to another RA8 dual-core MCU, such as the RA8D2, RA8M2, or RA8T2.

The RA8D2 device is used as the demonstration example in this section. However, the same procedure applies to other dual-core MCUs with minor adjustments as noted in the following descriptions.

The same migration procedure can also be applied to the `ek_ra8p1_dualcore` project included in this application note.

The overall migration process can be divided into three main parts:

- **Migrate the Solution Project:** It is recommended to create a new **Solution project** by selecting the target board, and then update the memory partitions, pin configuration, and clock settings in the `solution.xml` file to match the target MCU.

- **Migrate the CPU0 Project:** Adjust the BSP configuration and FSP modules and implement the necessary source code modifications to meet the requirements of the migrated project on CPU0.
- **Migrate the CPU1 Project:** Adjust the BSP configuration and FSP modules and implement the necessary source code modifications to meet the requirements of the migrated project on CPU1.

Step 1: Create a New Dual-Core Solution Project

Refer to the procedure described in Section 3.1 to create a new **Solution project** `RA8D2_DSP_example` using **Multicore Flat FreeRTOS** template.

In the **Device Selection** dialog, select **EK-RA8D2** as the target board, and choose **LLVM** as the toolchain, as shown in Figure 72.

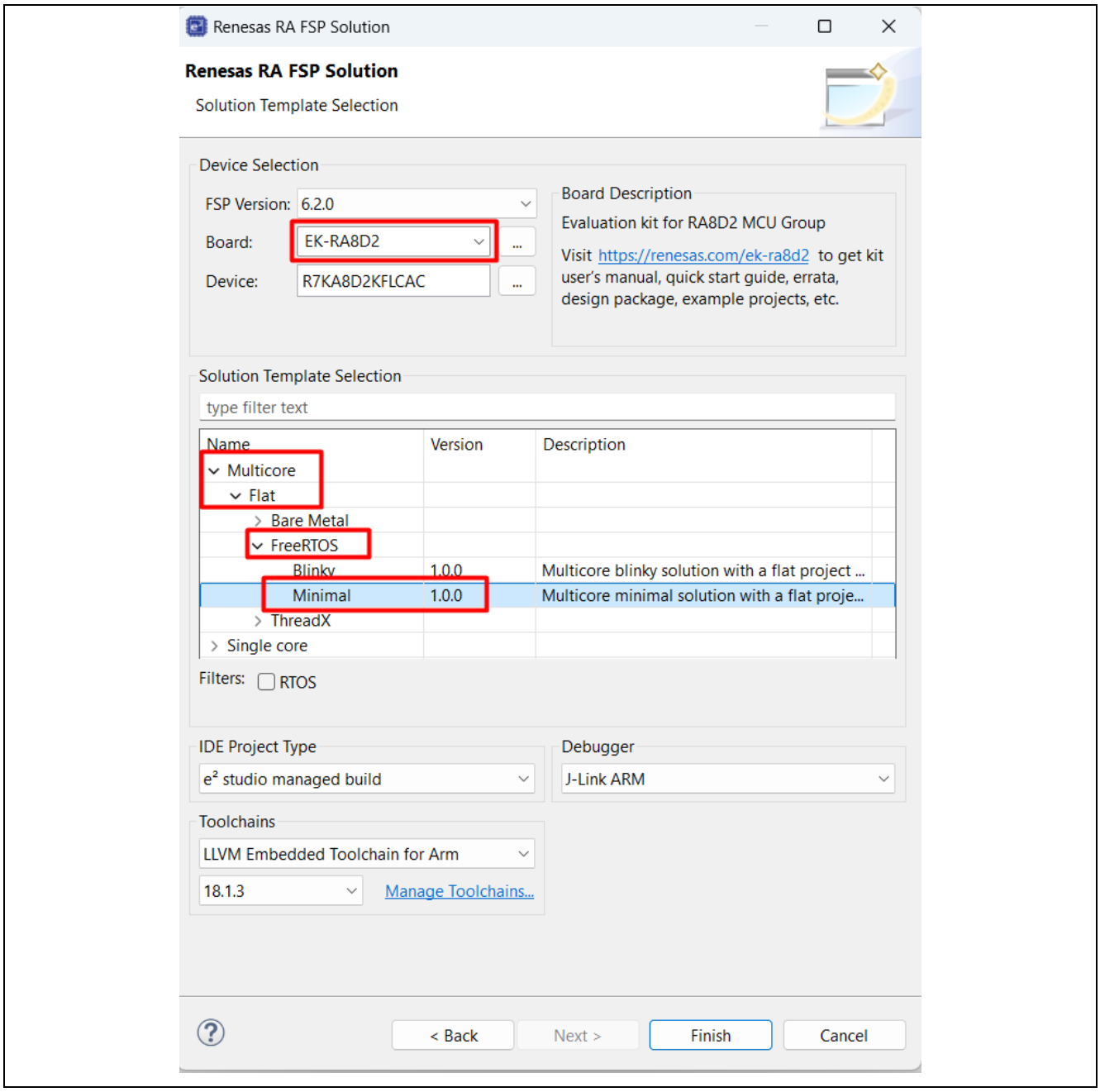

Figure 72. Example of Solution Template Selection.

Step 2: Define Memory Partitions, Configure Pin and Clocks at the Solution Project.

- Add partitions (e.g., SHARED_MEM).
- Assign Start/Size/Core/Security for the SHARED_MEM partition as follows.

For the migration based on `ek_ra8p1_dualcore`, these two configuration steps can be referenced in Section 5.1.2 for setup. In the `RA8P1_DSP_example` project, the memory map remains in its default configuration. Therefore, no modification to the memory map settings is required in the migrated Solution project `RA8D2_DSP_example`.

- **Clocks** configuration for entire Multicore Project. With `RA8D2_DSP_example` enable SCICLK as 120Mhz to use VCOM UART for log terminal output on `RA8D2_DSP_example_CPU1`. Navigate to the **Clocks** tab in the `solution.xml` of the `RA8D2_DSP_example` project to apply this configuration as shown in Figure 73. Disable any unused clocks to optimize system performance and reduce power consumption.

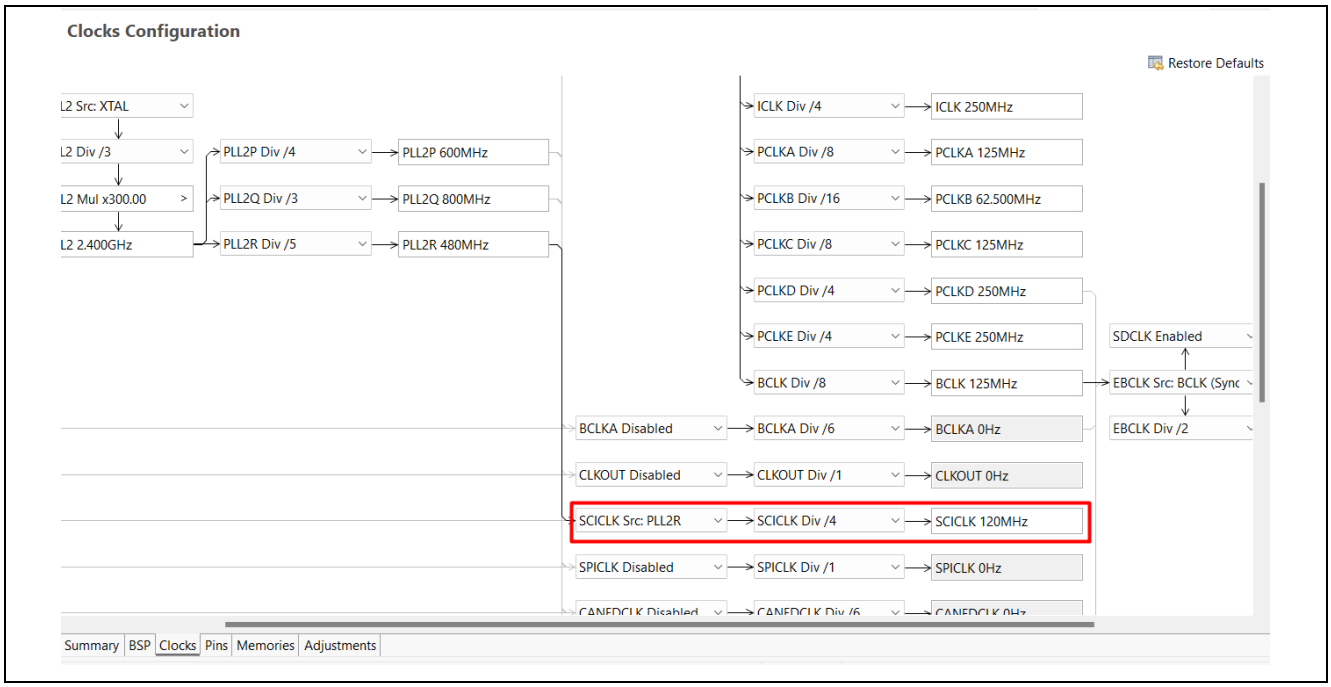

Figure 73. Example of Clock Configuration in Solution Project

- **Pins** Configuration: The SCI UART pins PD02 (RXD8), PD03 (TXD8), and external interrupt pin P009 (IRQ13-DS) are allocated to the CPU1 project for VCOM communication with the terminal and for receiving button interrupt signals. Therefore, these pins will be configured later in the CPU1 project. Disable all related pin configurations in the Solution project and CPU0 project.

  Navigate to the **Pins** tab, expand the **Connectivity: SCI** category, and select **SCI8**. Then, set the **Operation Mode** to **Disable.**

  Next, in the **IRQ**, scroll down and set **IRQ13-DS** to **None.**

Note: Make sure that any configuration changes made in the **Solution project** are applied to all projects in the chain by right-clicking on the Solution project `RA8D2_DSP_example` and selecting **Build Project**.

Step 3: Add and Configure FSP Stack for CPU0 Project

Configure IPC and set up IPC Maskable Interrupts in `RA8D2_DSP_example_CPU0`. The inter-processor communication (IPC) is handled in a dedicated IPC Thread that runs with the highest priority to ensure timely message exchange between the two cores. A mutex is also used in this application during IPC operations.

Create a new thread named **Ipc Thread** by clicking **New Thread** at the **Stacks** tab in `RA8D2_DSP_example_CPU0/configuration.xml`. The configuration for this thread is shown in Figure 74.

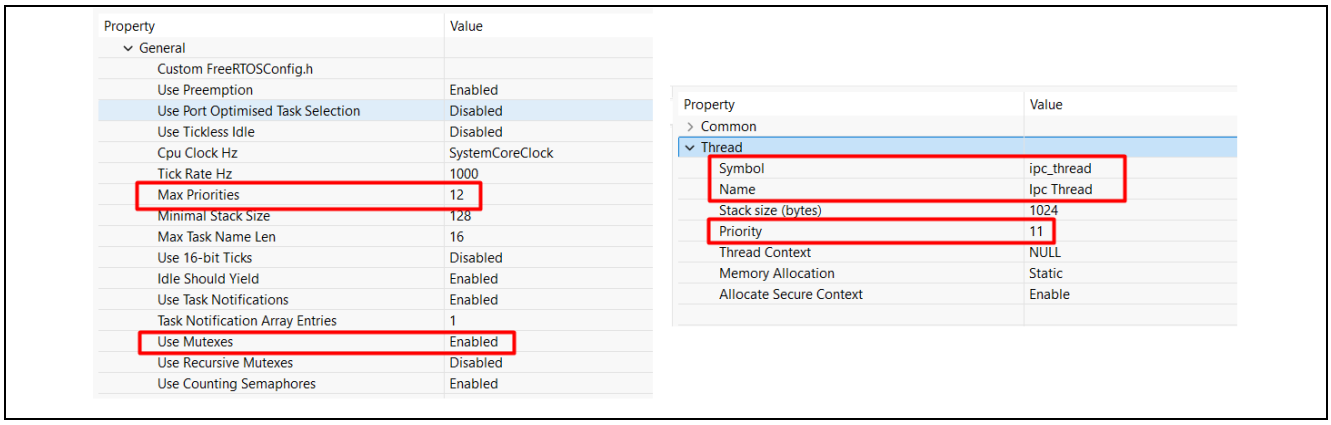

Figure 74. Example of Ipc Thread settings in RA8D2_DSP_example_CPU0

Add two IPC stacks under the **Ipc Thread** by selecting **New Stack > System > IPC (r_ipc)**. The **IPC configurations** are shown in Figure 75.

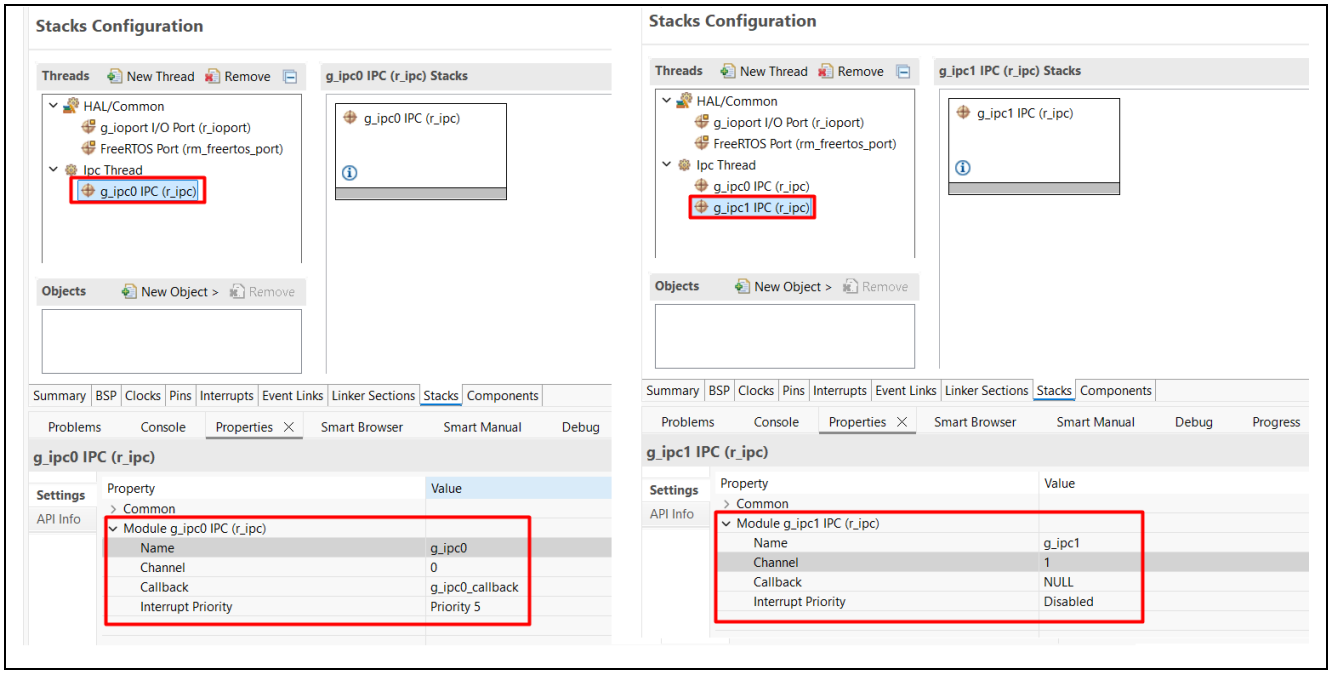

Figure 75. Example of IPC Configuration in RA8D2_DSP_example_CPU0

Add two hardware semaphores 0 and 1 in CPU0 project under **Ipc Thread** by click **New Stack > System > IPC Semaphore**. **The IPC semaphore** configurations is shown in Figure 76.

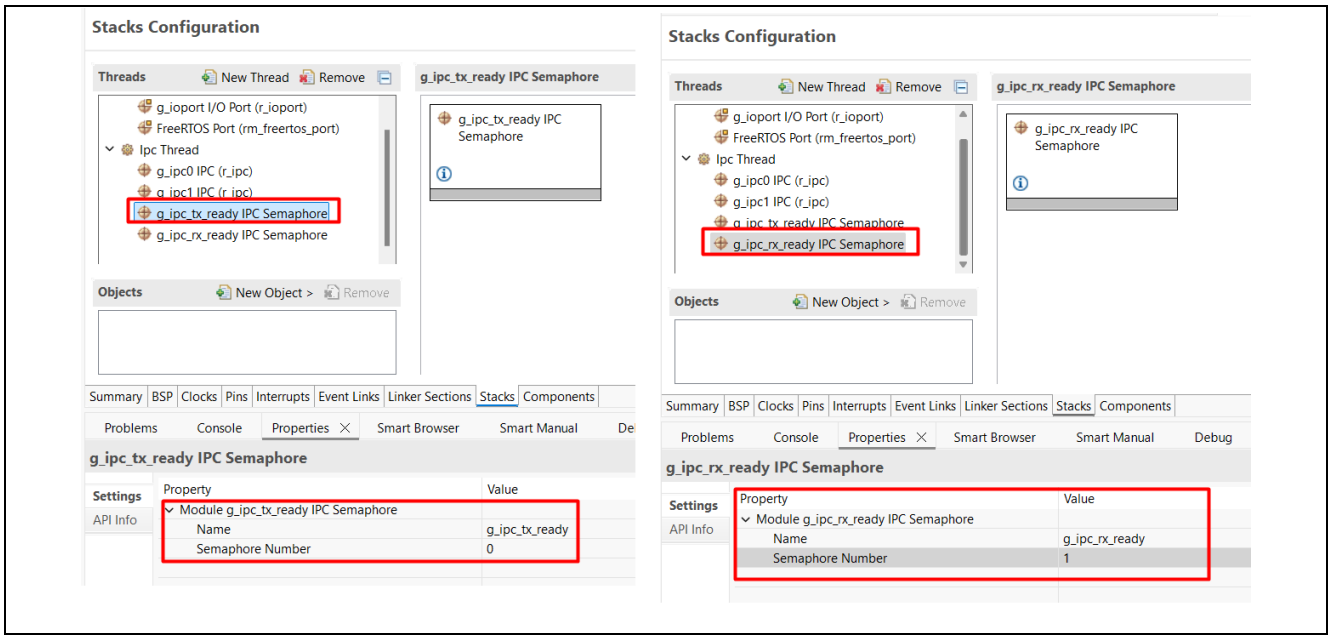

Figure 76. Example of Hardware Semaphores configuration in RA8D2_DSP_example_CPU0 Project

In `RA8D2_DSP_example_CPU0`, the Arm DSP example runs in a separate thread. Open `RA8D2_DSP_example_CPU0/configuration.xml`, then click add **New Thread** and create a thread named **DSP Thread**. The configuration for this thread is shown in Figure 77:

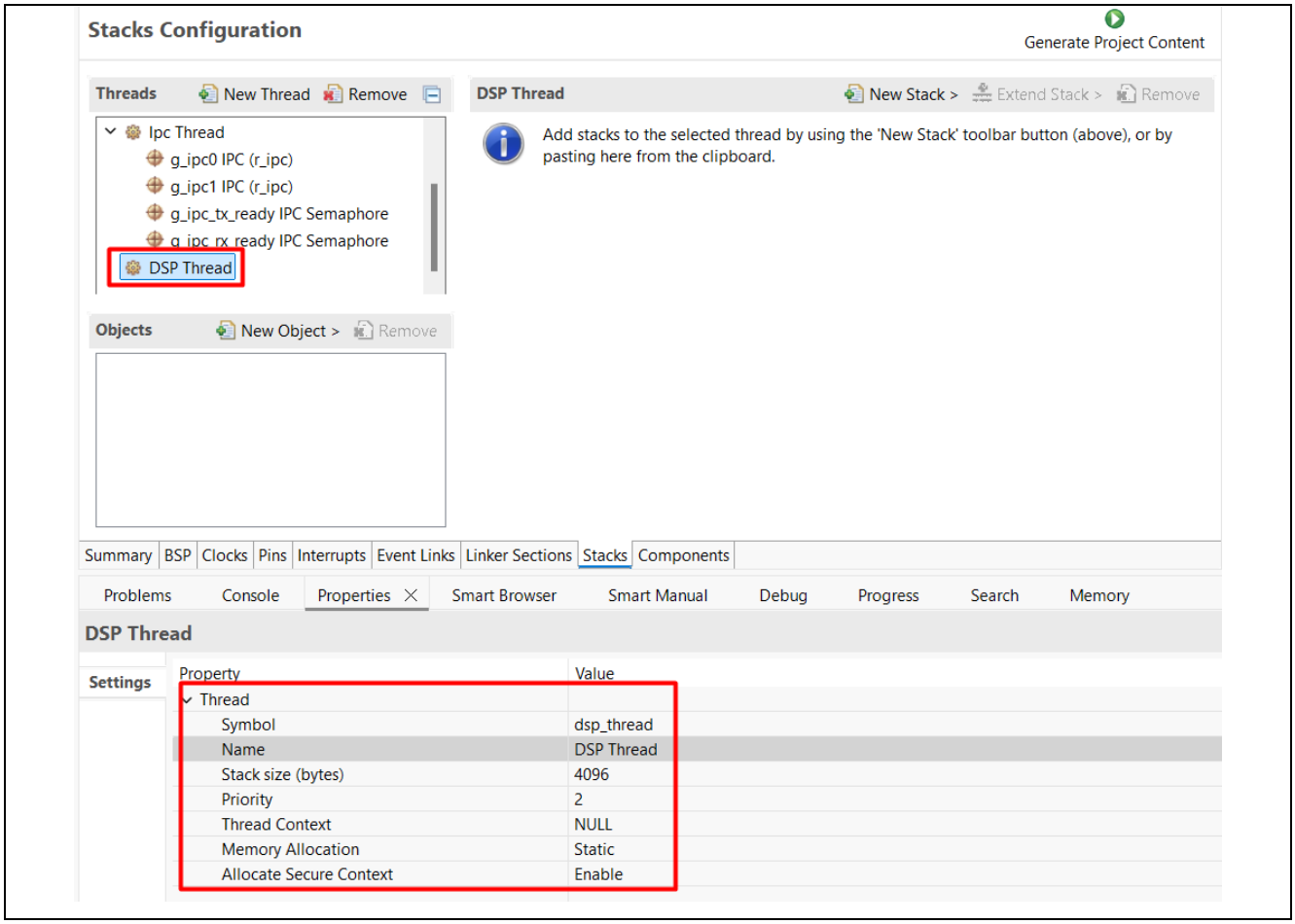

Figure 77. Example of Add DSP Thread in RA8D2_DSP_example_CPU0 Project

Then add the Arm DSP library to the `RA8D2_DSP_example_CPU0` project under **DSP Thread** as follows: Click **New Stack > DSP > Arm CMSIS DSP Library Source.** The complete FSP stacks in `RA8D2_DSP_example_CPU0` as shown in Figure 78.

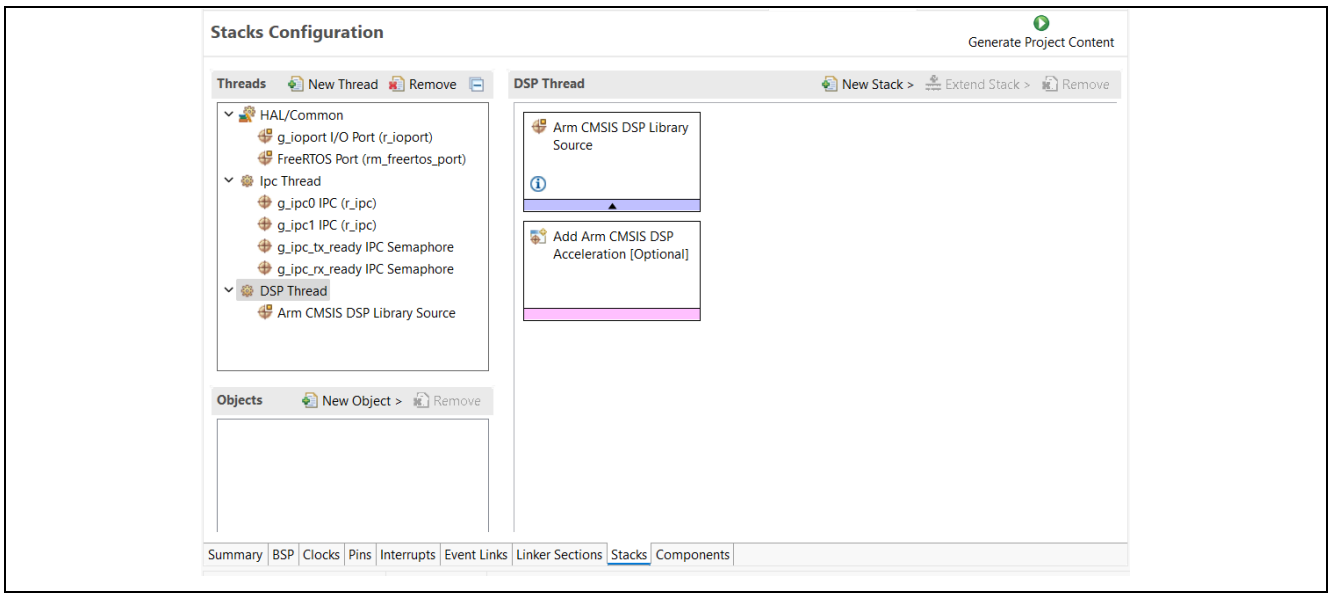

Figure 78. Complete FSP Stack Configuration in RA8D2_DSP_example_CPU0 Project

Step 4: Pins Configuration in RA8D2_DSP_example_CPU0 project.

As mentioned in **Step 1**, the SCI UART channel 8 pins (PD02, PD03) and the external interrupt pin (P009) must also be disabled in the CPU0 project as follows.

1. Navigate to the **Pins** tab, expand the **Connectivity: SCI** category, and select **SCI8**. Then, set the **Operation Mode** to **Disable**
2. In the **IRQ**, scroll down and set **IRQ13-DS** to **None.**

Step 5: Place the `arm_cmplx_mag_f32` function in ITCM and the `testInput_f32_10khz` buffer in DTCM.

Refer Section 4.4.1 to allocate function `arm_cmplx_mag_f32` to ITCM and Section 4.4.2 to allocate `testInput_f32_10khz buffer` to DTCM. The successful configuration is shown in Figure 79.

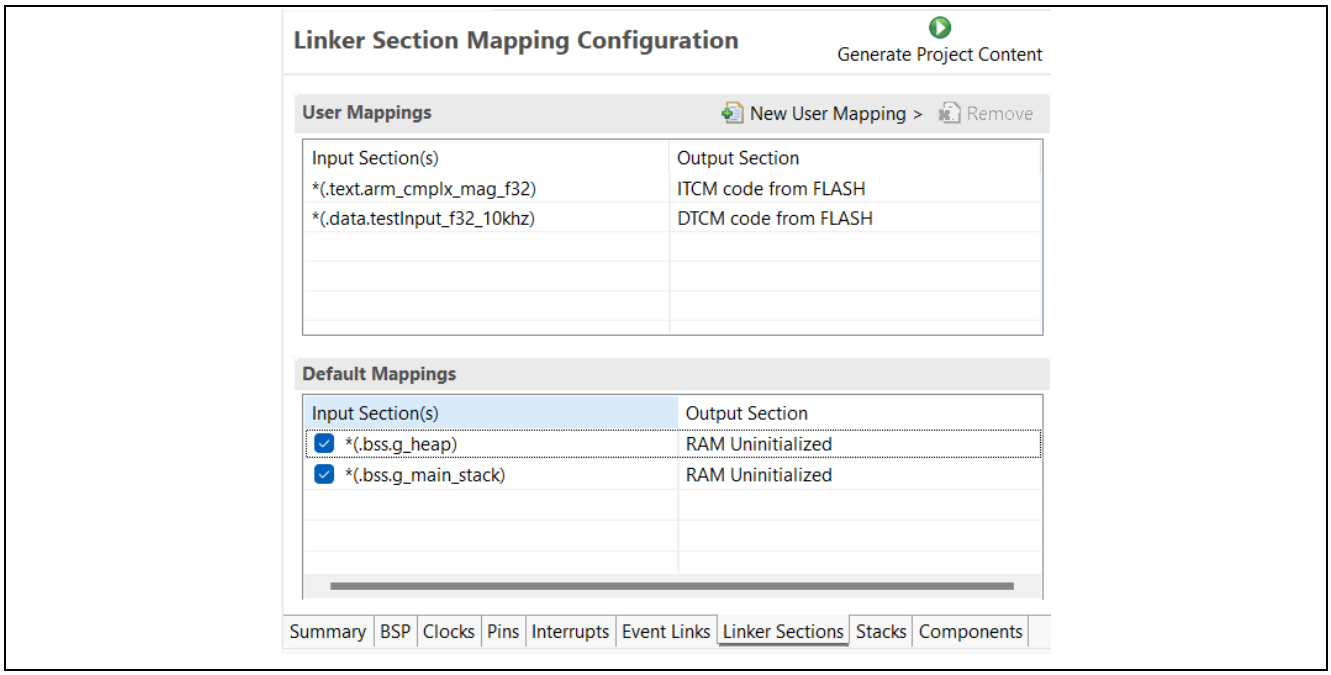

Figure 79. Example of Successfully Setup Data and Function in DTCM and ITCM

Step 6: Generate Project Content in RA8D2_DSP_example_CPU0 project

Under `RA8D2_DSP_example_CPU0/configuration.xml` click **Generate Project Content** to apply all the FSP configuration in CPU0 project.

Step 7: Implement CPU0-Specific Tasks.

All application source files implemented under the `/src` folder of the `RA8P1_DSP_example_cpu0` project are compatible with the `RA8D2_DSP_example_CPU0` project. Therefore, simply copy all contents from `RA8D1_DSP_example_CPU0/src` to `RA8D2_DSP_example_CPU0/src`.

Step 8: Build CPU0 project.

Right click on `RA8D2_DSP_example_CPU0` project and select **Build Project.**

Step 9: Add FSP Stacks in CPU1 Project

This step also applies the FSP configuration from the `RA8P1_DSP_example_cpu1` project to the `RA8D2_DSP_example_CPU1` project.

Unlike the manual configuration performed in **Step 3**, this step utilizes the **Import/Export FSP Stacks** feature provided by e² studio, which simplifies the porting or migration of an existing project to a different target MCU.

Export the configuration from the `RA8P1_DSP_example_cpu1` project:

1. Open the **configuration.xml** file in the `RA8P1_DSP_example_cpu1` project.
2. Under the **Ipc Thread**, right-click each module and select **Export**, as shown in Figure 80.
3. Save the exported configuration as a new `.xml` file (for example, `ipc_thread_config.xml`), ensuring that the **Include Common Properties** option is selected then click **Finish**, as illustrated in Figure 81.

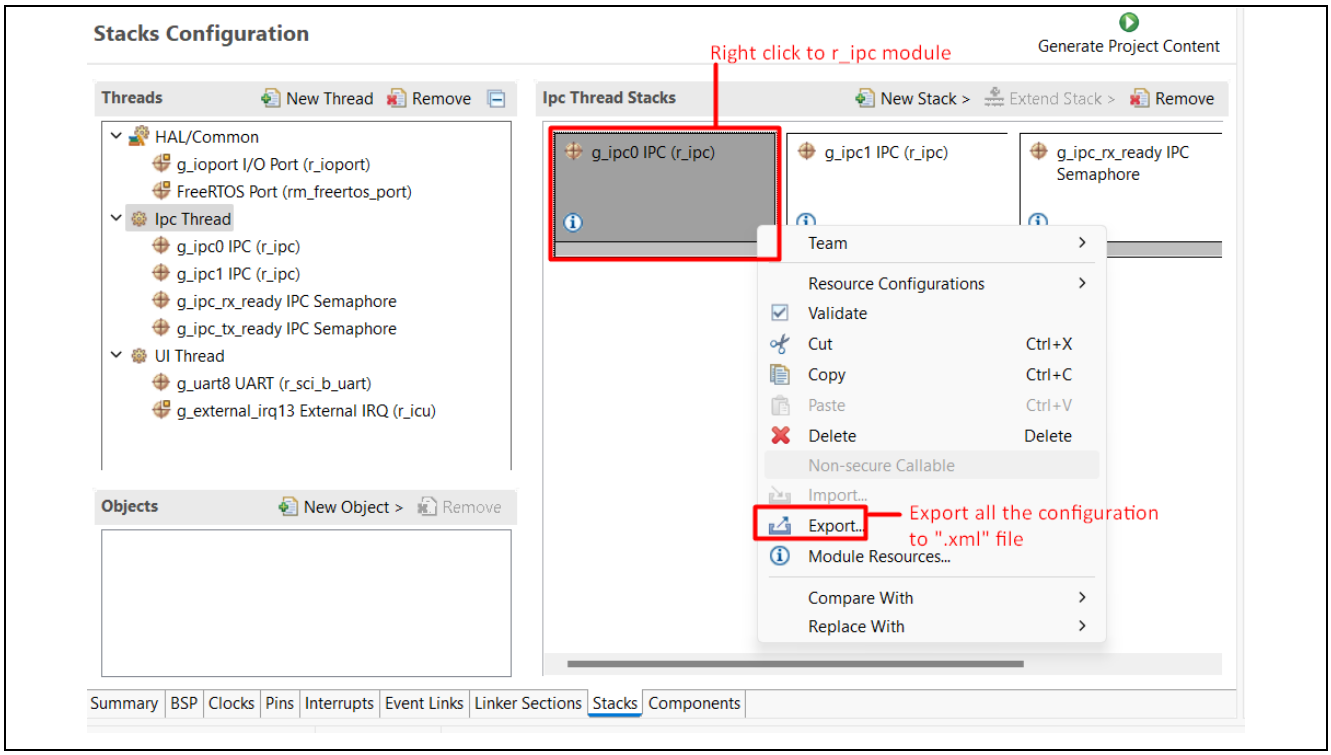

Figure 80. Export the FSP Stack in RA8P1_DSP_example_cpu1

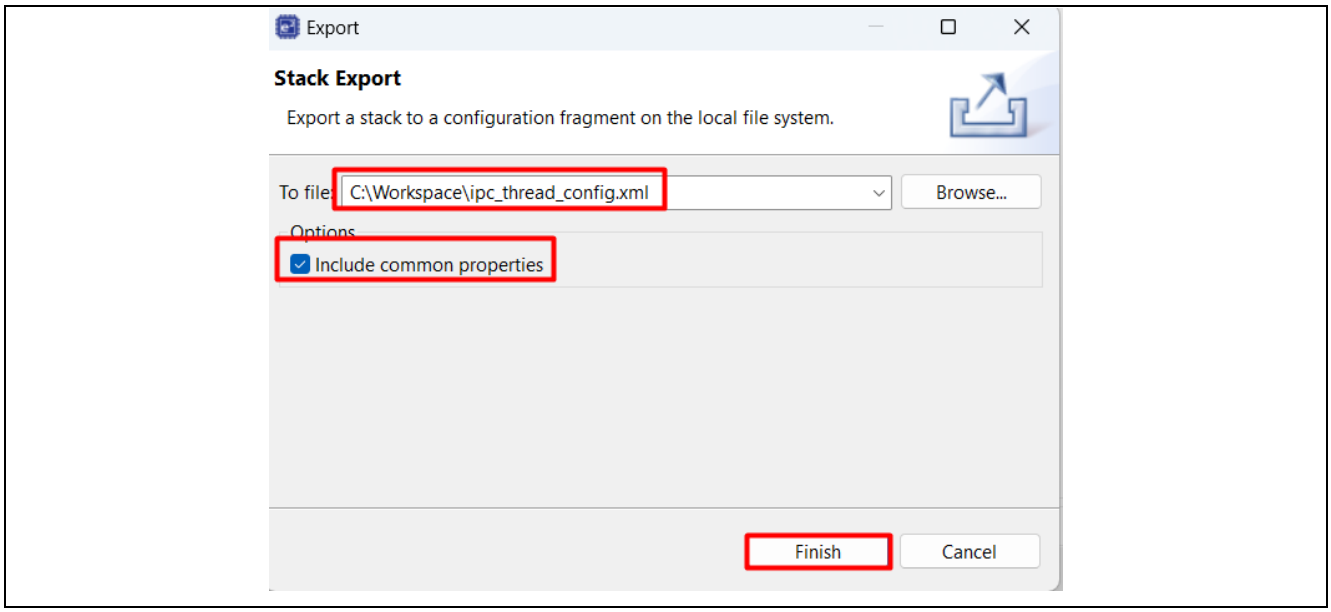

Figure 81. Example of Settings in Stack Export Dialog

Repeat the same export process for all **FSP stacks** used in the `RA8P1_DSP_example_cpu1` project.
It is recommended to export each **FSP stack** separately for each thread to make configuration management and re-importing easier during project porting or migration to another MCU. In this example, the stacks under the **Ipc Thread** are saved in `ipc_thread_config.xml`, while the stacks under the **UI Thread** are saved in `ui_thread_config.xml`.
After successfully exporting all FSP stacks from the `RA8P1_DSP_example_cpu1` project, proceed to the next step to import the FSP stack into the `RA8D2_DSP_example_CPU1` project.

Step 10: Import the FSP stacks to RA8D2_DSP_example_CPU1 project

Double click to `configuration.xml` from `RA8D2_DSP_example_CPU1` project, then navigate to **Stacks** tab and manually create two threads **Ipc Thread** and **UI Thread** which the general setting as shown in Figure 82 and detail each thread configuration as shown in Figure 83.

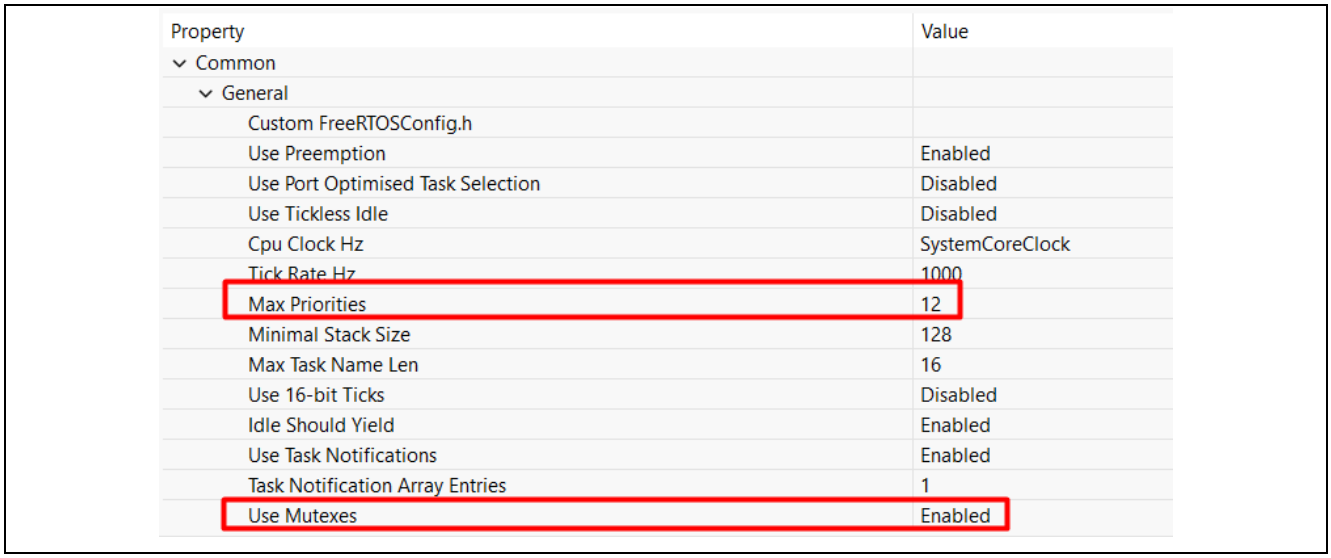

Figure 82. Example of FreeRTOS Thread setting in RA8D2_DSP_example_CPU1 project

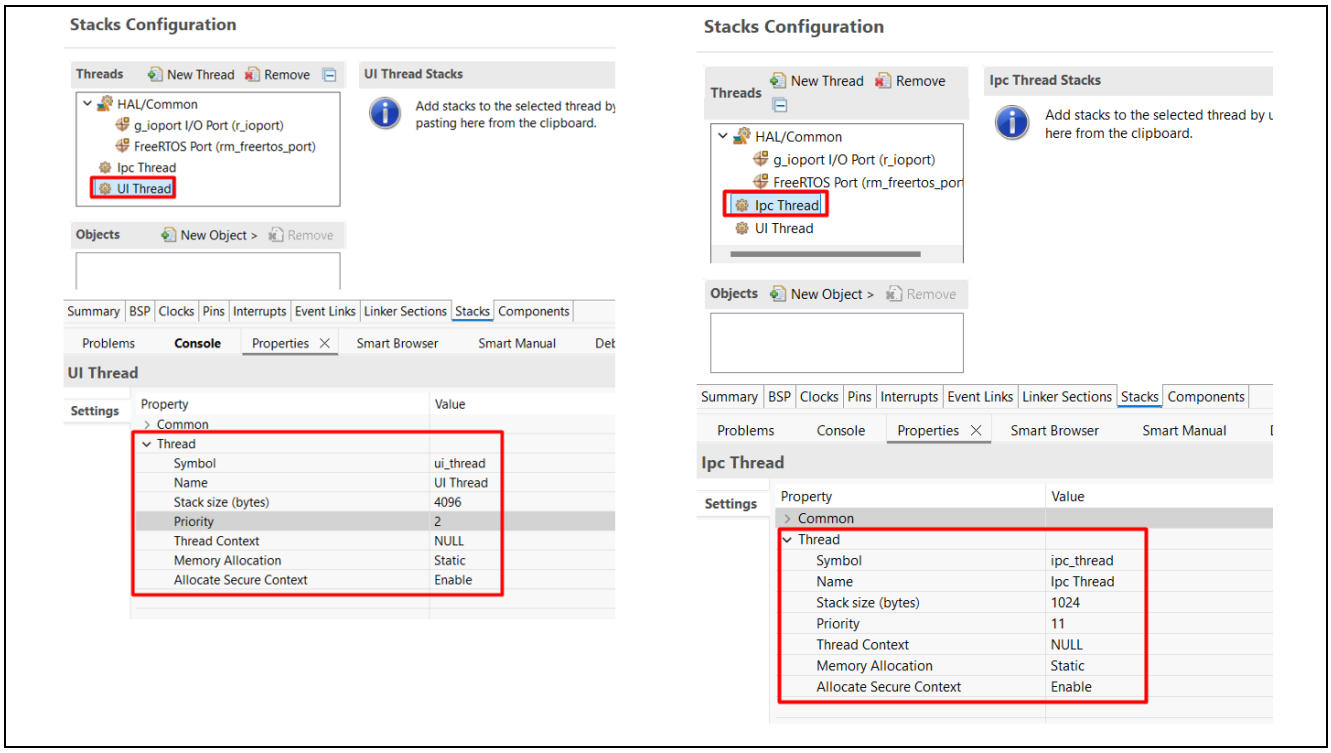

Figure 83. Example of Thread Configuration in RA8D2_DSP_example_CPU1 project

Import the previously saved **FSP stacks** from the `RA8P1_DSP_example_cpu1` project:

1. In the **Ipc Thread Stacks**, right-click and select **Import**.
2. In the **Stack Import** dialog, navigate to the `ipc_thread_config.xml` file exported in **Step 9**, then click **Select All** and enable the **Include Common Properties** option. Finally, click **Finish** to complete the import process.

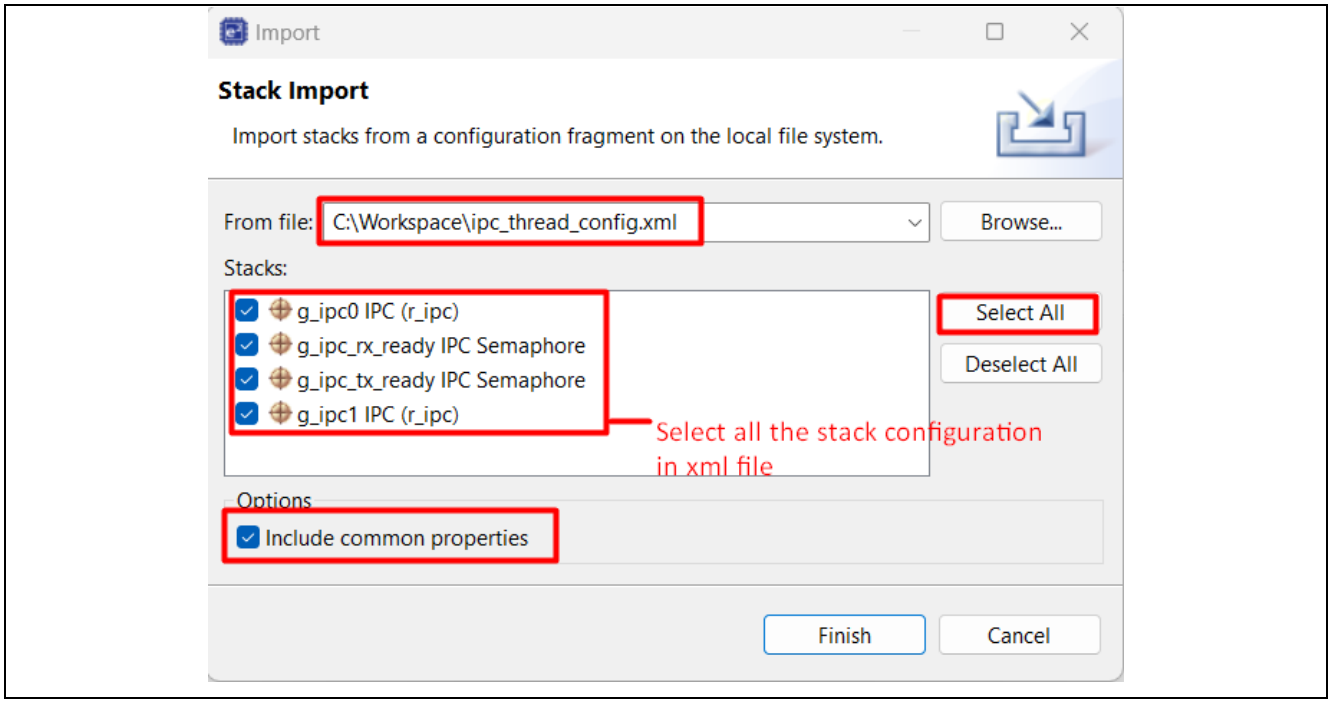

Figure 84. Example of Stack Import Dialog for Ipc Thread in RA8D2_DSP_example_CPU1 project

Perform the same procedure to import the FSP configuration under the **UI Thread**:

1. In the **UI Thread Stacks**, right-click and select **Import**.

2. In the **Stack Import** dialog, navigate to the `ui_thread_config.xml` file exported in **Step 9**, then click **Select All** and enable the **Include Common Properties** option. Finally, click **Finish** to complete the import process.

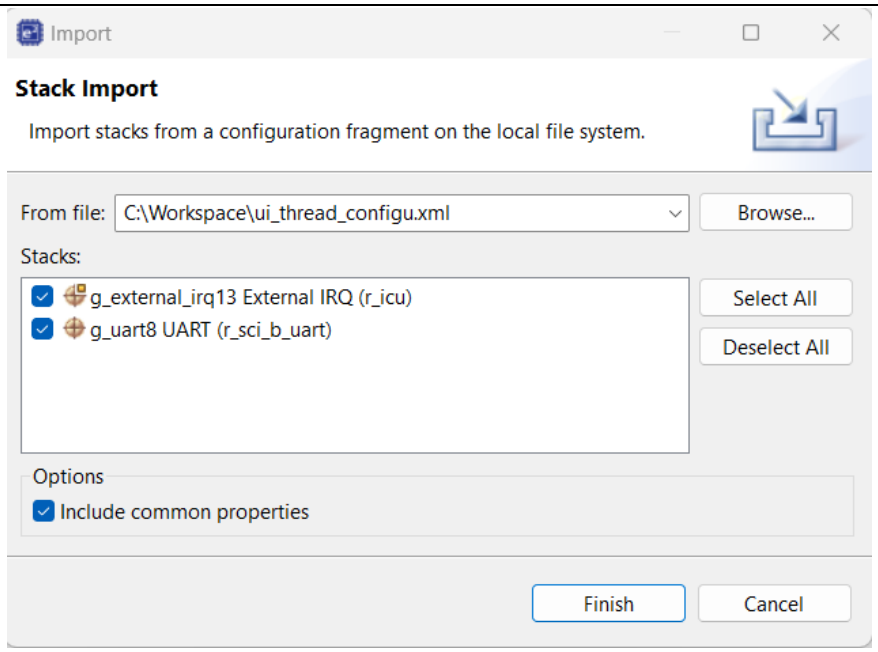

Figure 85. Example of Stack Import Dialog for UI Thread in RA8D2_DSP_example_CPU1 project

Step 11: In the FSP Configurator of the `RA8D2_DSP_example_CPU1` project, navigate to the **Pins** tab and ensure that both the SCI UART channel 8 pins (PD02, PD03) and the external interrupt pin (P009) are enabled, as shown Figure 86 and Figure 87.

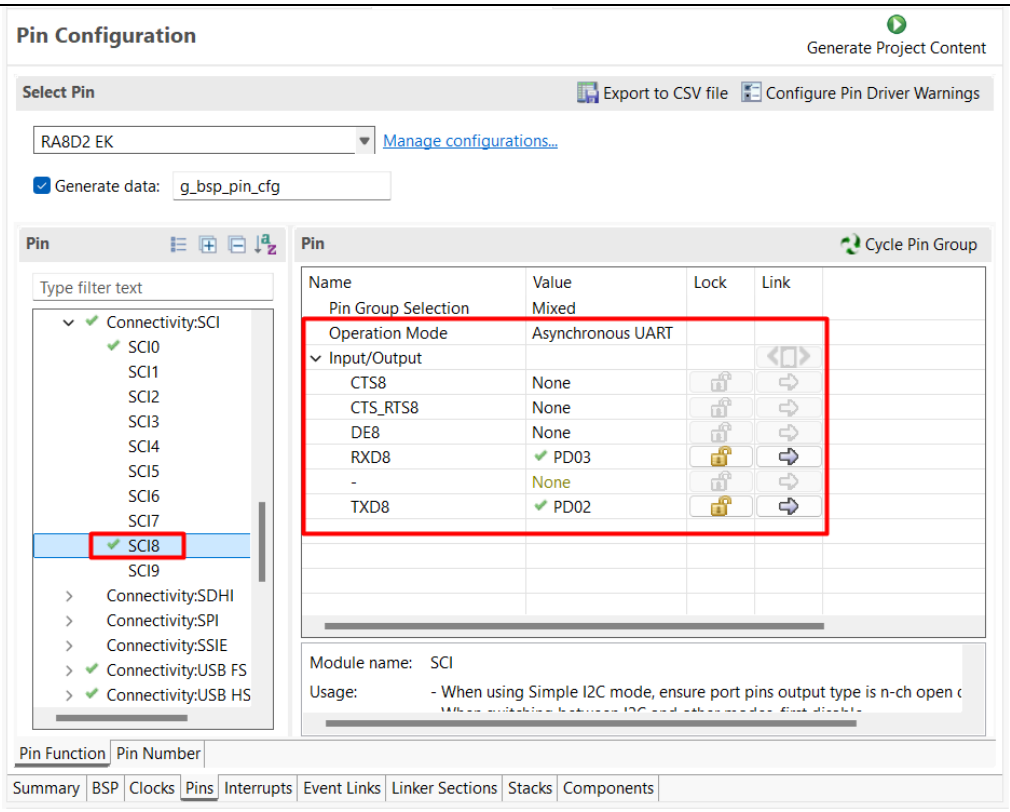

Figure 86. Example of SCI UART Pin Configuration in RA8D2_DSP_example_CPU1 project

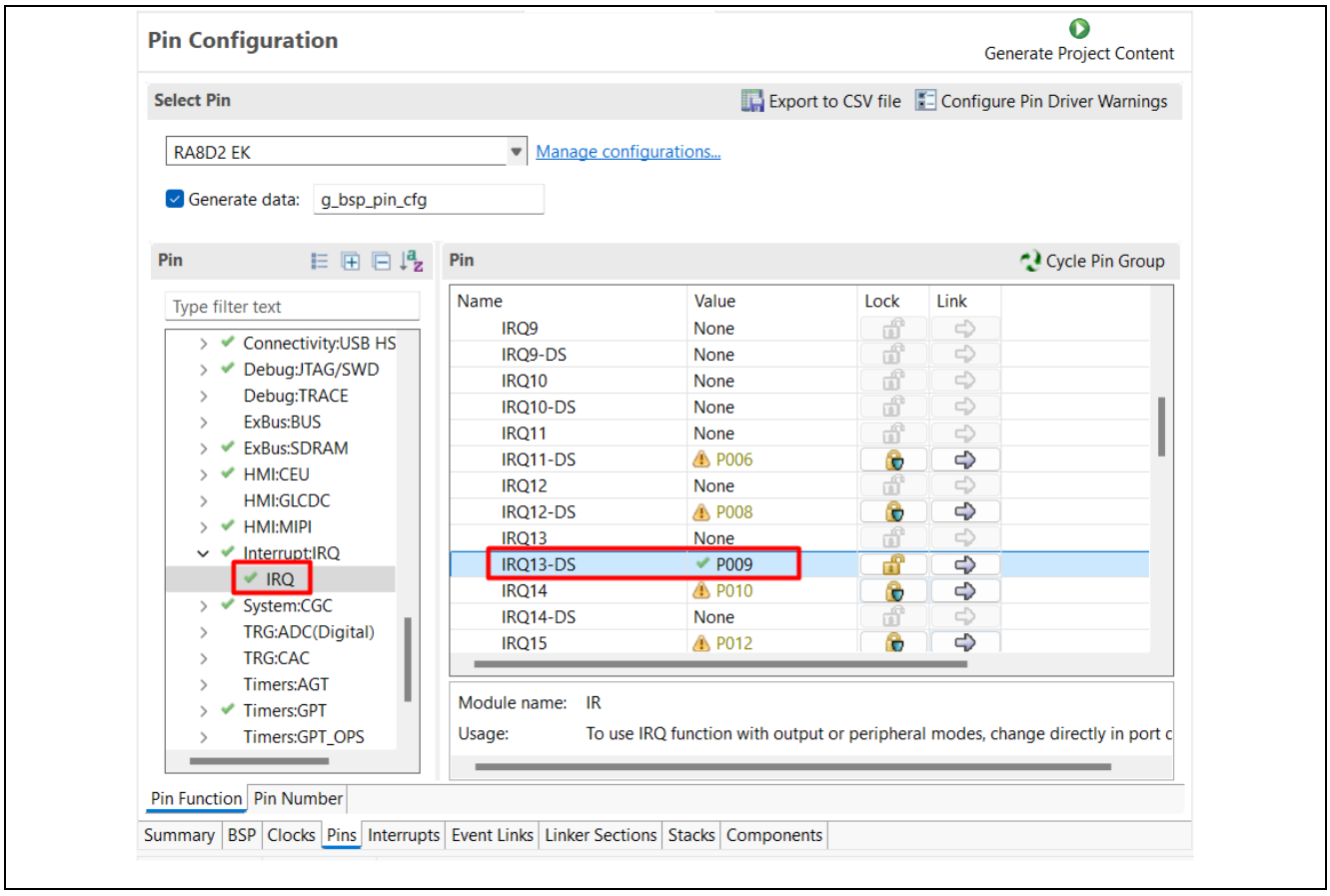

Figure 87. Example of External Interrupt Pin Configuration in RA8D2_DSP_example_CPU1 project

Step 12: Place the `ipc_callback` to CTCM.

Refer to Section 4.4.3 for details on allocating the `ipc_callback` function to CTCM.

Step 13: Click **Generate Project Content** to apply all configuration settings.

Step 14: Implement CPU1-Specific Tasks.

Copy the files under `RA8P1_DSP_example_cpu1/src` to `RA8D2_DSP_example_CPU1/src`.

Step 15: Build the **Solution project** to apply the build chain by right-clicking on `RA8D2_DSP_example` and selecting **Build Project**.

Step 16: Debug and verify the project as described in Section 7.3.

9. References

The following documents were used in creating this Quick Design Guide:

- Renesas RA8P1 Group User's Manual Hardware, document No. R01UH1064
- Renesas RA8D2 Group User's Manual Hardware, document No. R01UH1065
- Renesas RA8M2 Group User's Manual Hardware, document No. R01UH1066
- Renesas RA8T2 Group User's Manual Hardware, document No. R01UH1067
- RA Flexible Software Package Documentation Release v6.2.0.
- RA8P1 Memory Architecture App Note, document No R01AN7880
- Reference System Design for Vision AI Design using Ethos-U NPU, document No. R11AN0995
- Using the Ethos-U NPU with RA8 MCUs, document No. R01AN7712
- Arm Cortex®-M85 Processor Technical Reference Manual, document No. 101924, available from Arm

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	June.30.25	-	Initial release
1.10	Jan.30.26	-	Update to FSP version 6.2.0

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/.