Renesas RA8 Series

Getting Started with IPC on Dual Core MCU

Introduction

This application note introduces Inter-Processor Communication (IPC) on the RA8 dual-core MCU, emphasizing its importance in facilitating seamless coordination and data exchange between the two CPU cores. It explains the IPC architecture of the RA8 dual-core and describes key communication mechanisms, including hardware semaphores, message FIFOs, and interrupt handling through both maskable and non-maskable interrupts (NMIs).

The guide offers practical instructions for configuring, implementing, and debugging IPC using the Renesas Flexible Software Package (FSP). It also includes reference projects that demonstrate real-world use cases. Whether developing with a real-time operating system (RTOS) like FreeRTOS or working in a bare-metal environment, this document equips developers with the foundational knowledge and tools needed to create secure, reliable, and high-performance dual-core applications on the RA8 dual-core platform.

This application note covers:

- Dual Core System Architecture: Overview of the RA8 dual-core MCU structure, CPU selection, configuration, and resource allocation for efficient dual-core operation.
- IPC Mechanisms & Implementation: Explanation of different IPC communication methods, including semaphores, message FIFOs, and interrupts, with sample code.
- Synchronization and Data Exchange: Techniques for managing core communication using semaphores, NMIs, message FIFOs, and a user-level wrapper function to use IPC HAL drivers.
- Example Projects & Reference Applications: Practical implementations demonstrating IPC techniques, including FreeRTOS-based IPC, and semaphore/message FIFO examples.

Required Resources

- Flexible Software Package (FSP) v6.0.0 or later
  Note: FSP support for different boards is listed below.
  - For RA8P1(FSP 6.0.0 or later is required).
  - For RA8T2(FSP 6.1.0 or later is required).
  - For RA8D2 and RA8M2 (FSP 6.2.0 or later is required).

- Renesas e² studio - Version: 2025-04.1 or later is required.

Supported MCU Groups

At the time of the release, the supported MCU groups are:

- RA8P1 Group
- RA8T2 Group
- RA8D2 Group
- RA8M2 Group

Note: The App Note and the accompanying App Project are developed for the RA8P1 dual-core MCU as a reference. However, since the RA8T2, RA8D2, and RA8M2 MCUs share the same architecture, these resources can also serve as valuable references for other dual-core MCUs. The application projects can be used as a reference to recreate the application projects for other RA8 dual-core MCUs.

Contents

1. Introduction 4
2. Dual-Core System Architecture 4
2.1 Overview of RA8 Dual-Core MCU Architecture 4
2.2 Primary and Secondary CPU Project Selection 5
2.3 Setup and Configuration of the Two Cores 5
2.3.1 Dual-Core MCU Configuration Using FSP Configurator 5
2.3.2 Flexible Core Assignment 6
2.3.3 Boot Behavior on RA8 Dual-Core 6
2.4 MCU Resource Allocation in RA8 Dual-Core 6
2.4.1 Peripheral Sharing and Synchronization 6
2.4.2 Shared Memory Resources 6
2.4.3 Memory Configuration via solution.xml 6
3. IPC Module 7
3.1 Overview of IPC in RA8 Dual-Core MCU 7
3.1.1 Semaphore 7
3.1.2 Inter-Processor Interrupts 8
3.1.3 Message FIFO 8
3.1.4 IPC Register Architecture in RA8 Dual-Core MCU 9
3.2 IPC Module Implementation using FSP 11
3.2.1 IPC API Overview: Key Components and Structures 11
4. Synchronization and Data Exchange 12
4.1 Semaphore Mechanisms for Dual-Core Communication 12
4.2 Maskable and Non-Maskable Interrupts for IPC 12
4.3 Message FIFOs for Data Exchange Between Cores 12
5. FSP Support for IPC 12
5.1 Setting Up IPC Using FSP Configurator 13
5.2 Usage Notes: IPC Interrupts and Channel Configuration 15
5.2.1 Interrupt and Callback Behavior 15
5.2.2 Channel Matching Between Cores 15
5.2.3 IPC Semaphore Support 16
5.2.4 NMI Support for IPC 16
6. Example Projects and Reference Applications 17
6.1 Sample Code for IPC Implementation 17
6.2 Sample Example Code Implementation of IPC in RA8P1 MCUs 19
6.3 Using FreeRTOS for IPC in Dual-Core Environments 20
7. Running the Application Projects 21
7.1 Import the Projects 21

7.2 Build Projects 21
7.3 Download, Run, and Verify the Projects 22
7.4 Verifying the Application 24
8. Debugging and Troubleshooting 28
9. Next Steps 28
10. References 28
Revision History 31

1. Introduction

The IPC mechanism in the Renesas RA Dual-core MCUs enables efficient coordination between the two CPU cores by supporting hardware sharing, memory access synchronization, and data exchange. It provides multiple communication methods, including semaphores, FIFO buffers, and interrupt events, ensuring smooth and reliable interaction between CPU0 and CPU1 cores.

To prevent resource conflicts and maintain system integrity, the IPC mechanism uses multiple locks and semaphore controls to enforce mutually exclusive access to shared peripherals and memory. Depending on the application, developers can opt for either dedicated or shared hardware semaphores.

For data exchange, IPC includes FIFO-based communication channels that facilitate efficient and structured data flow:

- A write-only FIFO for CPU0 and a read only FIFO for CPU1, enabling CPU0 to send data to CPU1.
- A write-only FIFO for CPU1 and a read only FIFO for CPU0, enabling CPU1 to send data to CPU0.

Thereby supporting bidirectional communication between the two cores.

These features provide a robust framework for dual-core applications, allowing reliable resource division, synchronized operations, and efficient data sharing between CPU0 and CPU1.

2. Dual-Core System Architecture

RA Dual-core MCUs integrate two processing cores within a single chip to enhance performance, parallel processing, and power efficiency. These MCUs are commonly used in real-time applications, IoT devices, and industrial automation systems.

The Renesas RA8 dual-core MCU uses a heterogeneous architecture, with a Cortex-M85 (CM85) core configured as Core 0 and a Cortex-M33 (CM33) core configured as Core 1. Core 0 is intended for compute-intensive operations, while Core 1 manages system control and peripheral interaction.

2.1 Overview of RA8 Dual-Core MCU Architecture

The System architecture in a Renesas RA dual-core MCU is engineered to facilitate efficient communication and data exchange between the cores and connected peripherals. The system primarily employs a shared bus architecture, where both cores interface with a standard system bus linked to memory and peripherals. To maintain orderly access and prevent conflicts, arbitration mechanisms are implemented to manage resource sharing and prioritize requests fairly. In addition, dedicated buses are utilized for accessing Instruction Tightly Coupled Memory (ITCM) and Data Tightly Coupled Memory (DTCM), enabling low-latency and high-speed memory operations.

To manage shared resource access and coordinate operations between the two cores, the RA8 dual-core architecture utilizes IPC mechanisms. These include FIFO buffers, interrupt signaling, shared memory regions, and semaphores, all of which support synchronization and efficient data exchange. IPC ensures that both CPUs can work together effectively while preserving system stability and performance.

The underlying dual-core bus architecture is built on standard interconnects such as AXI, AHB, and APB buses, which facilitate communication between the Cortex-M85 and Cortex-M33 cores, as well as with memory and peripherals.

RA8 dual-core MCU features a high-performance Arm® Cortex®-M85 core operating at speeds of up to 1 GHz, alongside an Arm® Cortex®-M33 core running at up to 250 MHz, and offers the following key features:

- Up to 1 MB of MRAM for non-volatile memory storage
- 2 MB of SRAM, including:
- 256 KB TCM RAM for the Cortex-M85 core
- 128 KB TCM RAM for the Cortex-M33 core
- 1.664 MB of user-accessible SRAM

- Integrated Arm® Ethos™-U55 Neural Processing Unit (NPU)
- Octal Serial Peripheral Interface (OSPI) for high-speed flash memory access
- Layer 3 Ethernet Switch Module (ESWM), USB Full-Speed (USBFS), USB High-Speed (USBHS), and SD/MMC Host Interface
- Graphics LCD Controller (GLCDC) for advanced display support
- 2D Drawing Engine (DRW) for hardware-accelerated graphics rendering
- Support for MIPI DSI/CSI interfaces for display and camera connectivity
- Rich set of analog peripherals
- Comprehensive security and safety features for reliable operation

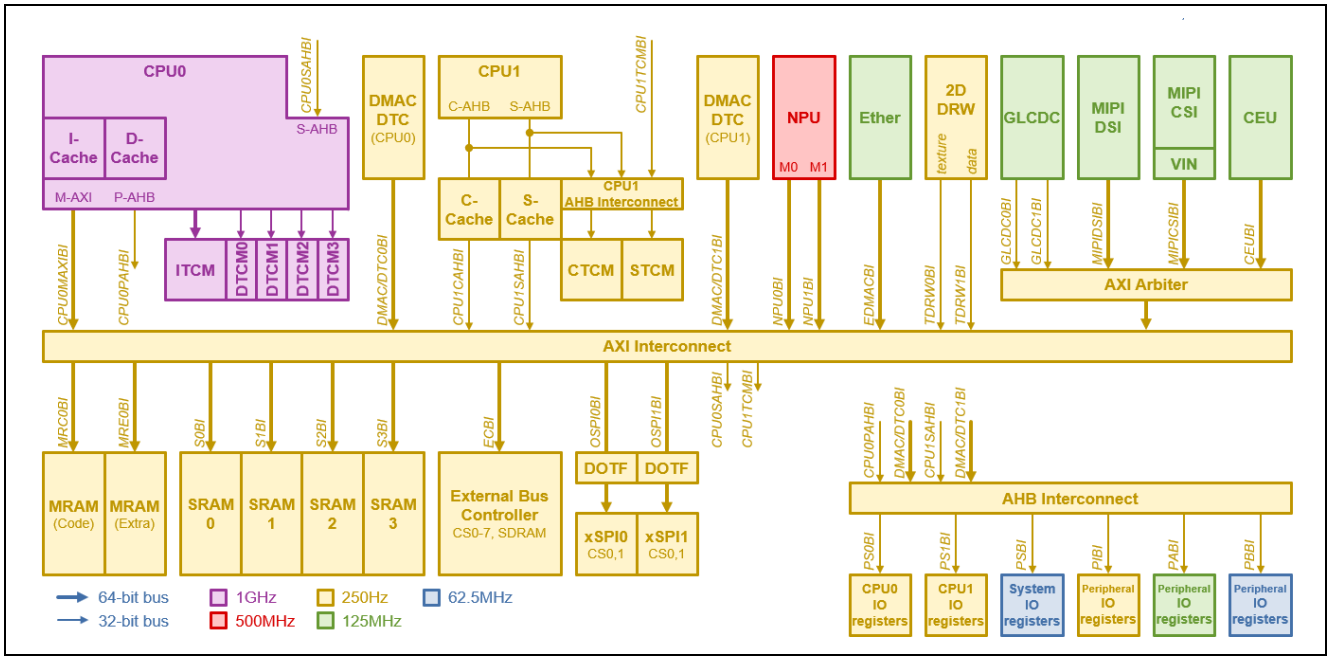

Figure 1. RA8 Dual Core System Architecture Overview

For more information on the Architecture of RA8 Dual Core MCU, please refer to the “Developing with RA8 Dual Core MCUs”(R01AN7881) App note.

2.2 Primary and Secondary CPU Project Selection

e² studio, in combination with the FSP and BSP, provides a streamlined way to create dual-core projects through a single run of the project creation wizard. This is accomplished using the “Renesas RA FSP Solution” project creation option.

When targeting a dual-core device, the wizard automatically generates three projects:

1. CPU0 Project (Primary Core): Contains the application code for the primary core.
2. CPU1 Project (Secondary Core): Contains the application code for the secondary core.
3. Solution (Container) Project: Acts as a top-level container that manages both CPU projects. It handles shared configuration, project organization, and overall solution structure.

The FSP and BSP together provide default configurations for both CPU0 and CPU1, allowing developers to focus on application development while ensuring the two cores are adequately integrated for deployment to the target MCU.

2.3 Setup and Configuration of the Two Cores

2.3.1 Dual-Core MCU Configuration Using FSP Configurator

The setup and configuration of a dual-core MCU in the Renesas RA family is managed through the FSP Configurator within e² studio. When a dual-core RA FSP Solution project is created, the tool automatically

generates two separate application projects by default, one for CPU0 (Cortex-M85) and another for CPU1 (Cortex-M33).

2.3.2 Flexible Core Assignment

While the default configuration assigns the Cortex-M85 (CPU0) as the Primary CPU and the Cortex-M33 (CPU1) as the Secondary CPU, developers can reassign core roles based on the application's needs.

2.3.3 Boot Behavior on RA8 Dual-Core

On the RA8 dual-core MCU, the primary CPU is configurable via a boot firmware command and starts execution first after a system reset. By default, CPU0 is configured as the primary core.

Once the system reset is released:

The secondary CPU remains in a power-gated state.

- The primary CPU must explicitly activate it through a write to the CPUnACTCSR register.
- Depending on the setting of the CPUnWAITCR.CPUWAIT bit, the secondary CPU will either:
  - Enter a wait state (CPUWAIT), allowing safe loading of code/data into its TCMs, or
  - Immediately begin program execution.

Note: If CPU0 is configured as the secondary core, its activation status must be verified before transferring data or releasing the wait state. The primary CPU is not affected by the CPUWAIT setting.

2.4 MCU Resource Allocation in RA8 Dual-Core

In the RA8 dual-core MCU, many peripherals and memory resources are shared between the two cores, CPU0 (Cortex-M85) and CPU1 (Cortex-M33). This shared architecture allows efficient hardware utilization but also requires careful resource management, particularly when both CPUs access the same peripherals or memory regions.

2.4.1 Peripheral Sharing and Synchronization

Most of the MCU's peripheral modules, such as timers, communication interfaces (UART, SPI, I2C), and system controllers, are common to both cores. While this enables flexible task distribution, developers must implement proper synchronization mechanisms to avoid race conditions and conflicts. This is typically achieved using hardware-supported IPC methods such as semaphores or locks, to coordinate access and ensure only one CPU interacts with a shared peripheral at a time.

2.4.2 Shared Memory Resources

Memory resources, including MRAM, SRAM, and external memory interfaces (e.g., OSPI or SDRAM), are also accessible by both cores. To support custom memory segmentation based on application requirements, the RA8 dual-core MCU allows flexible allocation of memory regions.

2.4.3 Memory Configuration via solution.xml

During project setup using the Renesas FSP, the solution.xml file in the solution project provides a GUI-based memory configuration interface. When opened in the FSP Configurator, this interface allows developers to:

- Define start and end addresses for selected memory blocks
- Allocate memory regions to either CPU based on their roles
- Fine-tune memory size and layout according to system constraints and performance needs

The memory configuration is shown in the snapshot below. This flexibility empowers developers to design applications that are balanced, efficient, and tailored to specific use cases, such as isolating memory for real-time tasks on CM33, while reserving high-speed memory for compute-intensive operations on CM85.

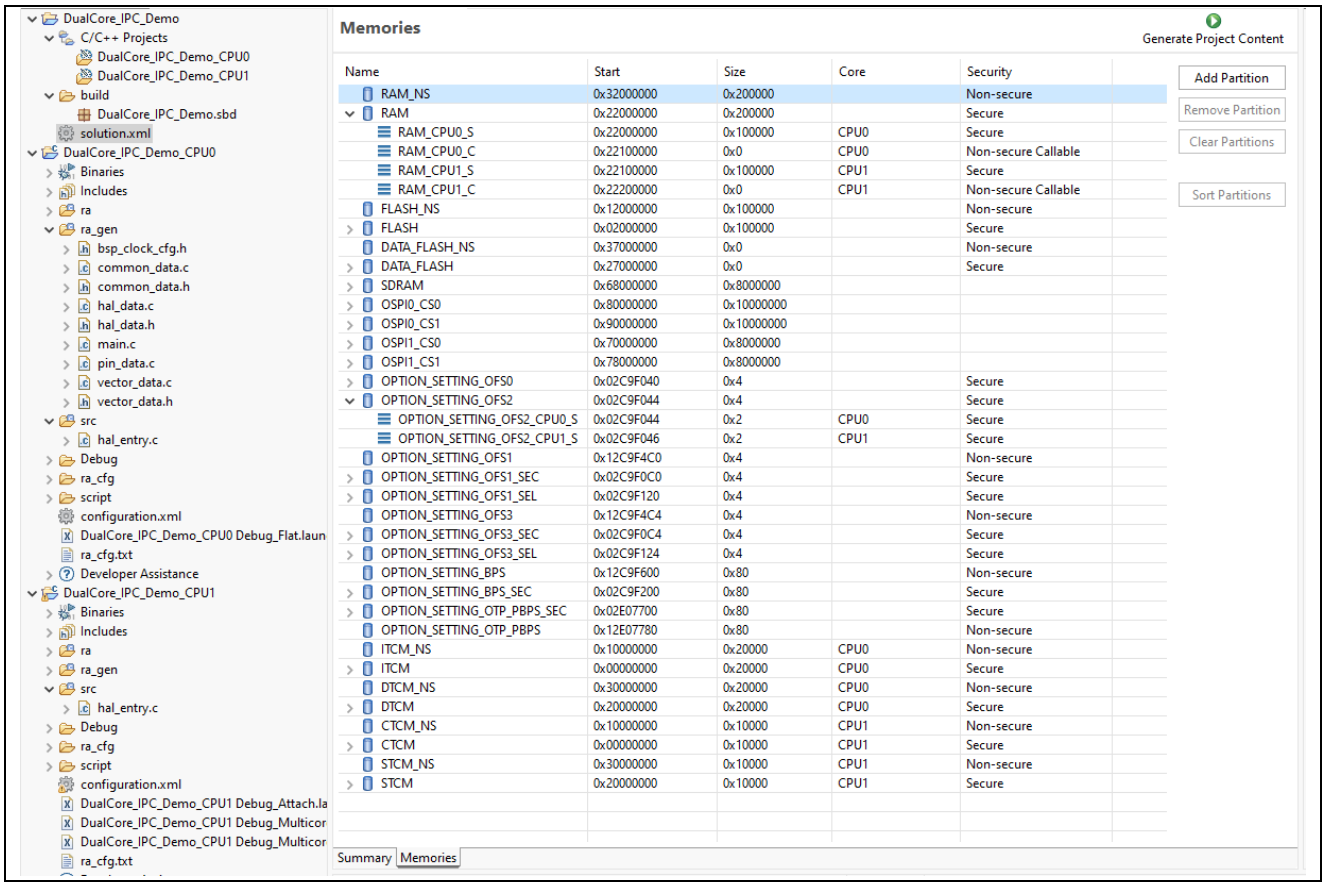

Figure 2. RA8 Dual-Core Memory configuration using solution.xml

3. IPC Module

The IPC mechanism within the RA MCU enables seamless resource sharing and communication between the two processor cores in the RA8 dual-core MCU. It plays a vital role in coordinating tasks, exchanging data, and ensuring safe access to shared hardware components.

IPC supports interrupt-driven communication, allowing one processor to trigger events in the other, which is especially useful for real-time synchronization and signaling.

To ensure mutually exclusive access to shared peripherals and memory, IPC implements a range of hardware-based semaphore mechanisms. These include multiple lock registers that prevent simultaneous access to critical resources, helping maintain data integrity and system stability. Developers can choose between dedicated (separate) or shared semaphore configurations, depending on the level of isolation or interaction required between the cores.

In addition to synchronization mechanisms, IPC also facilitates simple and efficient data exchange using FIFO buffers and shared memory. These are typically configured in a unidirectional or bidirectional setup.

3.1 Overview of IPC in RA8 Dual-Core MCU

The RA8 dual-core MCU features a robust IPC system within its CPUs, enabling efficient coordination, data sharing, and hardware resource management between the two processor cores, Cortex-M85 (CPU0) and Cortex-M33 (CPU1).

3.1.1 Semaphore

- IPC provides 16 semaphore locks to enforce mutual exclusion, allowing only one CPU to access a shared resource at a time.

- These semaphores are categorized by security and privilege attributes, grouped per 8 factors, enabling controlled and secure access based on application requirements.
- This ensures data consistency and avoids contention over shared peripherals or memory.

3.1.2 Inter-Processor Interrupts

IPC allows one CPU to trigger interrupt events on the other, supporting real-time signaling and event handling. These are divided into

- Inter-Processor Non Maskable Interrupts:
  - IPC0: 1 factor (CPU1 to CPU0)
  - IPC1: 1 factor (CPU0 to CPU1)
  - Each includes security and privilege configuration.
- Inter-Processor Maskable Interrupts:
  - IPC00, IPC01, IPC10, IPC11: Each supports 8 factors, allowing maskable interrupts.
  - Each factor is individually configurable with security and privilege attributes.

These interrupt types offer flexible notification mechanisms, useful for task synchronization, inter-core signaling, and exception handling.

3.1.3 Message FIFO

To support fast and structured data transfer between cores, IPC provides Message FIFO channels, each configured with:

- 4 FIFO stages per channel
- 32-bit data size per stage
- Associated maskable interrupt support
- Error detection mechanisms (FIFO Full/Empty status)

FIFO Channel Configurations:

- Message FIFO 00: CPU1 → CPU0 (write-only for CPU1, read-only for CPU0)
- Message FIFO 01: CPU1 → CPU0 (second channel)
- Message FIFO 10: CPU0 → CPU1 (write-only for CPU0, read-only for CPU1)
- Message FIFO 11: CPU0 → CPU1 (second channel)

These FIFO configurations allow bidirectional communication, ensuring that both CPUs can send and receive data asynchronously without disrupting each other’s execution flow.

Additionally, error flags such as FIFO FULL (write blocked) and FIFO EMPTY (read blocked) provide feedback for managing buffer states.

Item		Description
Semaphore	Lock information	• Total of 16 factors • Security attributes and privileged attributes are selected every 8 factors
Inter-Processor interrupt	Non-maskable	• IPC0 1 factor, one non-maskable interrupt, security attributes and privileged attributes are selected. • IPC1 1 factor, one non-maskable interrupt, security attributes and privileged attributes are selected.
	Maskable	• IPC00 8 factors, one maskable interrupt, security attributes and privileged attributes are selected*1 • IPC01 8 factors, one maskable interrupt, security attributes and privileged attributes are selected*1 • IPC10 8 factors, one maskable interrupt, security attributes and privileged attributes are selected*1 • IPC11 8 factors, one maskable interrupt, security attributes and privileged attributes are selected*1
Message-FIFO	FIFO	• Message FIFO 00 (CPU1 -> CPU0), one maskable interrupt, security attributes and privileged attributes are selected*1 • Message FIFO 01 (CPU1 -> CPU0), one maskable interrupt, security attributes and privileged attributes are selected*1 • Message FIFO 10 (CPU0 -> CPU1), one maskable interrupt, security attributes and privileged attributes are selected*1 • Message FIFO 11 (CPU0 -> CPU1), one maskable interrupt, security attributes and privileged attributes are selected*1 • 4 FIFO stages • Transfer data size 32 bits
	Error information	• Write with FIFO FULL*1 • Read with FIFO Empty*1

Figure 3. RA8 Dual-Core IPC Specification

3.1.4 IPC Register Architecture in RA8 Dual-Core MCU

The RA8 dual-core MCU provides a comprehensive IPC mechanism that enables secure, efficient communication and resource coordination between the dual-core system (CPU0: Cortex-M85 and CPU1: Cortex-M33). The register-level configuration supports semaphores, interrupt signaling, and data sharing through a dedicated IPC register map, as shown in the image.

Semaphore Control (IPCSEM0 – IPCSEM15)

- Addresses: 0x4002_0000 to 0x4002_003C
- 16 independent semaphore registers are available, allowing each core to lock access to shared hardware or memory.
- These registers help implement mutual exclusion by enabling one CPU to acquire a lock, preventing the other from accessing the same resource simultaneously.

Security and Privilege Control

- IPCSAR (IPC Security Arbitration Register) – 0x4000_8610
    - Manages access control and security levels (secure vs non-secure regions).
- IPCPAR (IPC Privileged Attribute Register) – 0x4000_8614

- Defines privilege levels (privileged vs unprivileged) for resource access.

NMI Control and Status Registers

These registers manage Non-Maskable Interrupt (NMI) signaling between the CPUs:

- IPC0NMICLR, IPC0NMISET, IPC0NMISTA – CPU0 NMI control/set/status
- IPC1NMICLR, IPC1NMISET, IPC1NMISTA – CPU1 NMI control/set/status (Addresses : 0x4002_0080 – 0x4002_009C)

They help trigger or clear NMI requests and check interrupt status for event-driven communication.

Inter-Processor Communication Channels

Each core has its own interrupt and FIFO-based message registers for data transfer and signaling.

FIFO Registers (TXD, RXD):

- Transmit Data Registers (Write-Only):
  - CPU1 → CPU0: IPC0TXD0, IPC0TXD1
  - CPU0 → CPU1: IPC1TXD0, IPC1TXD1
- Receive Data Registers (Read-Only):
  - CPU0: IPC0RXD0, IPC0RXD1
  - CPU1: IPC1RXD0, IPC1RXD1

FIFO Control and Status Registers:

- SET Registers: Used to send IRQ requests (e.g., IPC0ISET0, IPC1ISET1)
- CLR Registers: Clear IRQ or status (e.g., IPC0CLR0, IPC1CLR1)
- STA Registers: Provide FIFO status flags such as full/empty or error states (e.g., IPC0STA0, IPC1STA1)

Address Map Summary:

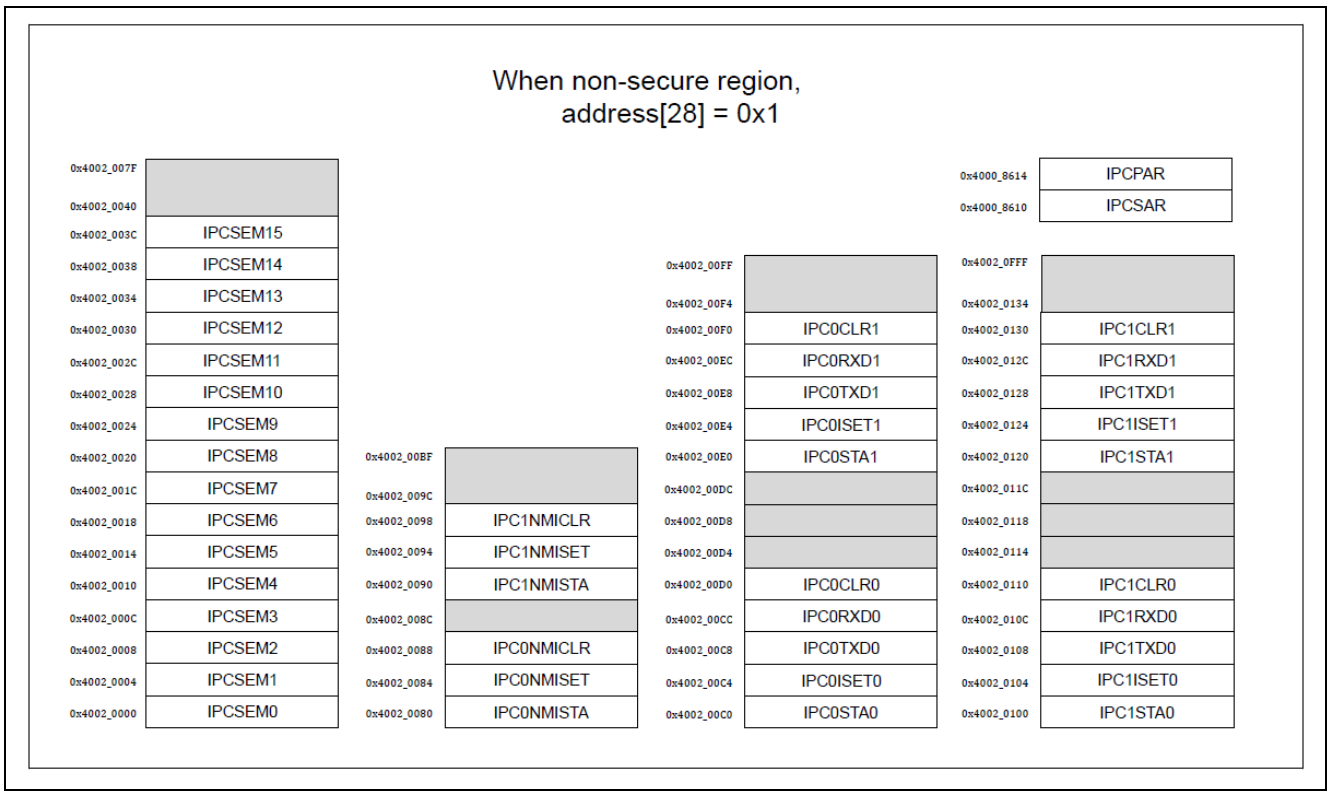

Figure 4. RA8 Dual-Core IPC Register Memory Map

3.2 IPC Module Implementation using FSP

For application developers working on the RA8 dual-core MCU, the Hardware Abstraction Layer (HAL) offers a dedicated IPC driver and API suite. This driver simplifies and standardizes the process of enabling communication between multiple CPU cores in a multicore system.

The IPC HAL provides a high-level interface that abstracts away the complexity of directly interacting with hardware registers, FIFOs, and interrupt control logic. Instead, developers can leverage a clean and consistent API to:

- Send and receive messages via hardware-backed FIFO queues, ensuring efficient, low-latency, one-way data transmission between cores.
- Generate and handle maskable interrupts, allowing one core to asynchronously notify another about an event or state change.
- Set user-defined callback functions to process incoming messages or event flags in a non-blocking, event-driven manner.

This design supports structured communication patterns, promoting better modularity and synchronization across CPU cores, while reducing the risk of race conditions and timing issues. By hiding hardware-level details, the IPC HAL enables faster development cycles and easier maintenance, especially in applications requiring reliable inter-core coordination, such as industrial automation, real-time control, and secure processing.

3.2.1 IPC API Overview: Key Components and Structures

Category	Item	Description
API Functions	R_IPC_Open ()	Initializes and configures an IPC instance.
	R_IPC_MessageSend ()	Sends a message through the hardware FIFO to the other core.
	R_IPC_EventGenerate()	Triggers an interrupt event to the target core.
	R_IPC_CallbackSet ()	Registers a user-defined callback for IPC events.
	R_IPC_Close ()	Closes and releases the IPC instance.
Data Structures	ipc_callback_args_t	Holds callback info: channel, message, event, and user context.
	ipc_cfg_t	Config for channel: ID, IRQ number, priority, callback function.
	ipc_api_t	Structure of function pointers for IPC operations.
	ipc_instance_t	Wrapper holding control block, config, and API pointers.
	ipc_ctrl_t	An opaque handle is used internally to track IPC instance state.
Handled Events	IPC_EVENT_IRQ0–7	IRQ events from the other core.
	IPC_EVENT_MESSAGE_RECEIVED	Indicates a message received in FIFO.
	IPC_EVENT_FIFO_ERROR_EMPTY	Error when reading from an empty FIFO.
	IPC_EVENT_FIFO_ERROR_FULL	Error when writing to a full FIFO.
Triggerable Events	IPC_GENERATE_EVENT_IRQ0–7	Used to trigger IRQ events to the other core.

Category	Item	Description
API Functions	R_BSP_IpcSemaphoreTake()	Attempts to take (lock) an IPC semaphore
	R_BSP_IpcSemaphoreGive()	Releases (unlocks) the specified IPC semaphore.
	R_BSP_IpcNmiEnable()	Enables NMI (Non-Maskable Interrupt) on current core.
	R_BSP_IpcNmiRequestSet()	Sends an NMI request to the opposing core.

4. Synchronization and Data Exchange

The RA8 dual-core MCU supports robust synchronization and data exchange between its dual cores. It provides hardware IPC mechanisms such as semaphores, interrupts, and message FIFOs to ensure safe and efficient inter-core communication. Semaphores are used to enforce mutual exclusion over shared resources. Maskable interrupts allow event-driven communication, while non-maskable interrupts (NMI) handle critical cross-core signaling. Message FIFOs enable buffered data transfer between cores without direct memory sharing. These mechanisms are abstracted via HAL APIs like R_IPC_MessageSend(), R_BSP_IpcSemaphoreTake(), and R_BSP_IpcNmiRequestSet(). FreeRTOS support on RA8 Dual-Core allows developers to integrate IPC with RTOS primitives for more structured coordination. Together, these features provide a flexible and secure framework for dual-core operation. The IPC system is essential for building high-performance, responsive multicore applications on RA8 dual-core MCU.

4.1 Semaphore Mechanisms for Dual-Core Communication

RA8 dual-core MCU provides hardware-based IPC semaphores (IPCSEMn), enabling safe synchronization between its dual-core architecture (e.g., Cortex-M85 and Cortex-M33). Semaphores prevent race conditions by allowing one core to take or give a semaphore, ensuring mutual exclusion. R_BSP_IpcSemaphoreTake() and R_BSP_IpcSemaphoreGive() APIs manage ownership across cores. Useful for protecting shared resources, such as memory buffers or peripherals.

4.2 Maskable and Non-Maskable Interrupts for IPC

The RA8 dual-core supports both maskable interrupts (via IPC events like IPC_EVENT_IRQ0) and NMI for inter-core signaling. Maskable interrupts allow signaling with flexible event bits and are processed through standard interrupt vectors. NMI is used for urgent, high-priority alerts. APIs like R_BSP_IpcNmiEnable() and R_BSP_IpcNmiRequestSet() facilitate NMI setup and request triggering between cores. Suitable for both routine coordination and critical cross-core alerts.

4.3 Message FIFOs for Data Exchange Between Cores

The IPC module on RA8 dual-core includes hardware FIFOs, enabling buffered, asynchronous message passing between cores. Functions such as R_IPC_MessageSend() and callbacks triggered by IPC_EVENT_MESSAGE_RECEIVED enable efficient unidirectional communication via a standardized HAL API. Ideal for sending control messages, commands, or status updates without blocking.

5. FSP Support for IPC

The FSP provides dedicated HAL-level driver modules for IPC, enabling developers to implement core-to-core communication. These drivers abstract the low-level register interactions required for semaphores, locks, message passing, and synchronization between processor cores. They can be easily added and configured using the FSP Configuration tool in e² studio.

Once added, these IPC drivers allow configuring parameters such as channel numbers, interrupt priorities, and callback functions. The following sections outline how to utilize these drivers effectively within application projects. This modular and configurable approach simplifies development while ensuring compatibility with the underlying hardware.

In addition to HAL-level support, the FSP's BSP includes settings for Semaphore Group Security Attribution. These settings define secure or non-secure access permissions for IPC-related hardware resources and are especially important for TrustZone-enabled MCUs. Correct configuration at both HAL and BSP levels is essential for safe and reliable IPC functionality across cores.

5.1 Setting Up IPC Using FSP Configurator

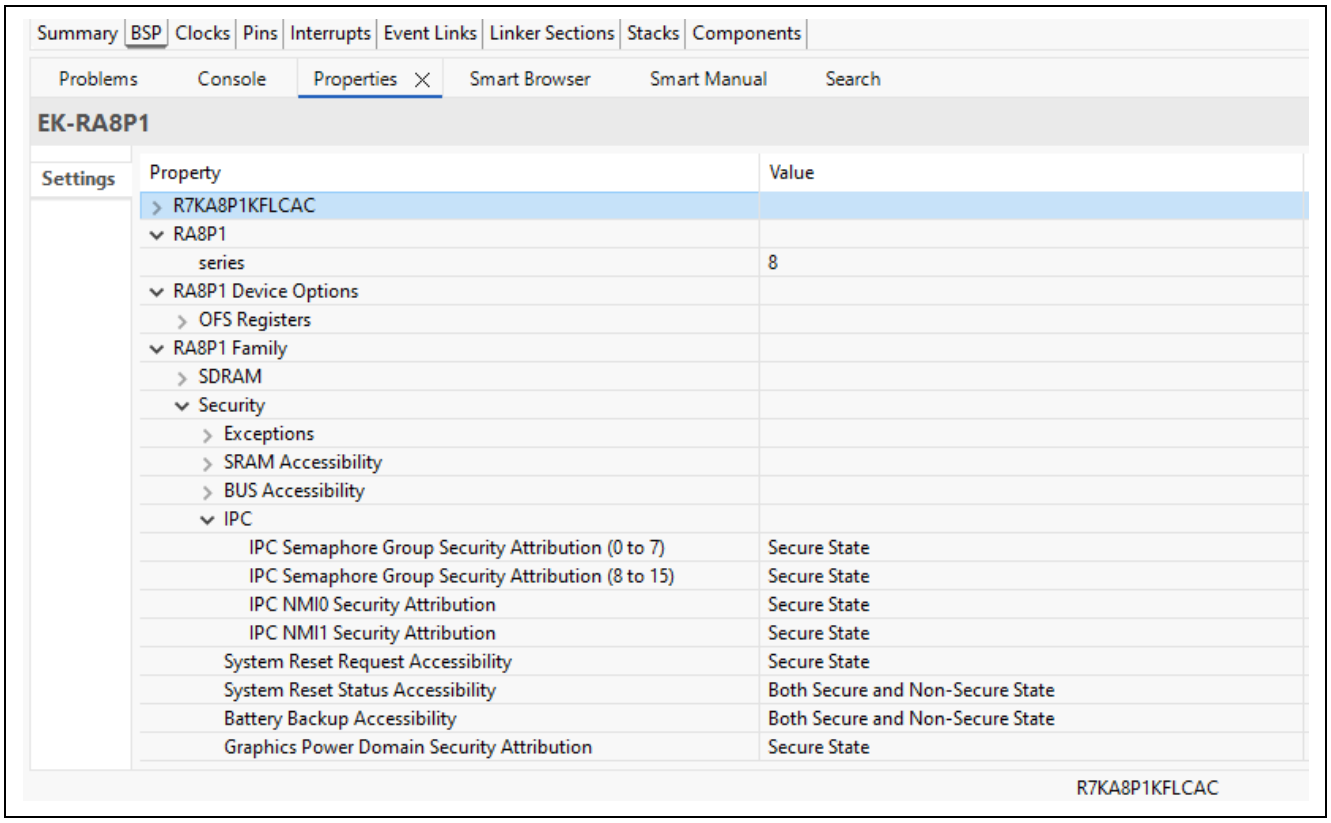

Figure 5. IPC NMI and Semaphore - Security Attribution Settings Overview

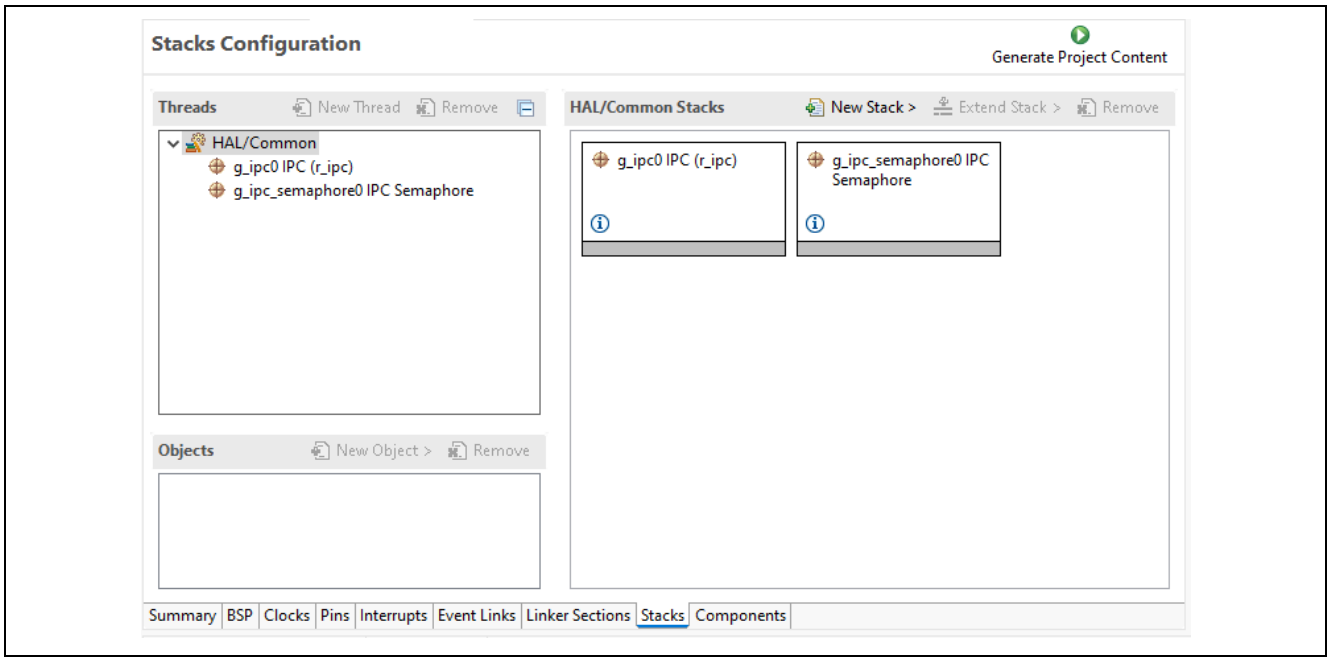

Figure 6. IPC and Semaphore Module Configuration

r_ipc module can be added to the stacks tab via New Stack → System → IPC (r_ipc). Non-secure callable guard functions can be generated for this module by right-clicking the module in the RA Configuration tool and checking the "Non-secure Callable" box.

The r_ipc module in the Renesas FSP facilitates inter-core communication between CPU0 and CPU1 on the RA8 dual-core MCU. Its primary functions include sending messages and generating interrupt events to notify the other core of specific conditions or data availability.

- The module allows either CPU core (CPU0 or CPU1) to send messages to the other using the R_IPC_MessageSend() API. This enables structured communication and data exchange between cores.
- It also supports event generation through the R_IPC_EventGenerate() function. This triggers a maskable interrupt on the receiving core, which is useful for alerting the other core about an event or newly available data.
- To handle these interrupt events properly, developers must configure the callback function under the driver property using the configurator. This registers a callback function on the receiving core that gets invoked when an IPC event is triggered.

Usage Consideration:

- If the interrupt/event feature is used for communication, configuring the callback function is essential to ensure the receiving core can properly process the incoming event.
- These APIs together enable efficient, low-latency signaling and data coordination between cores, which is vital for real-time and multi-threaded embedded applications.

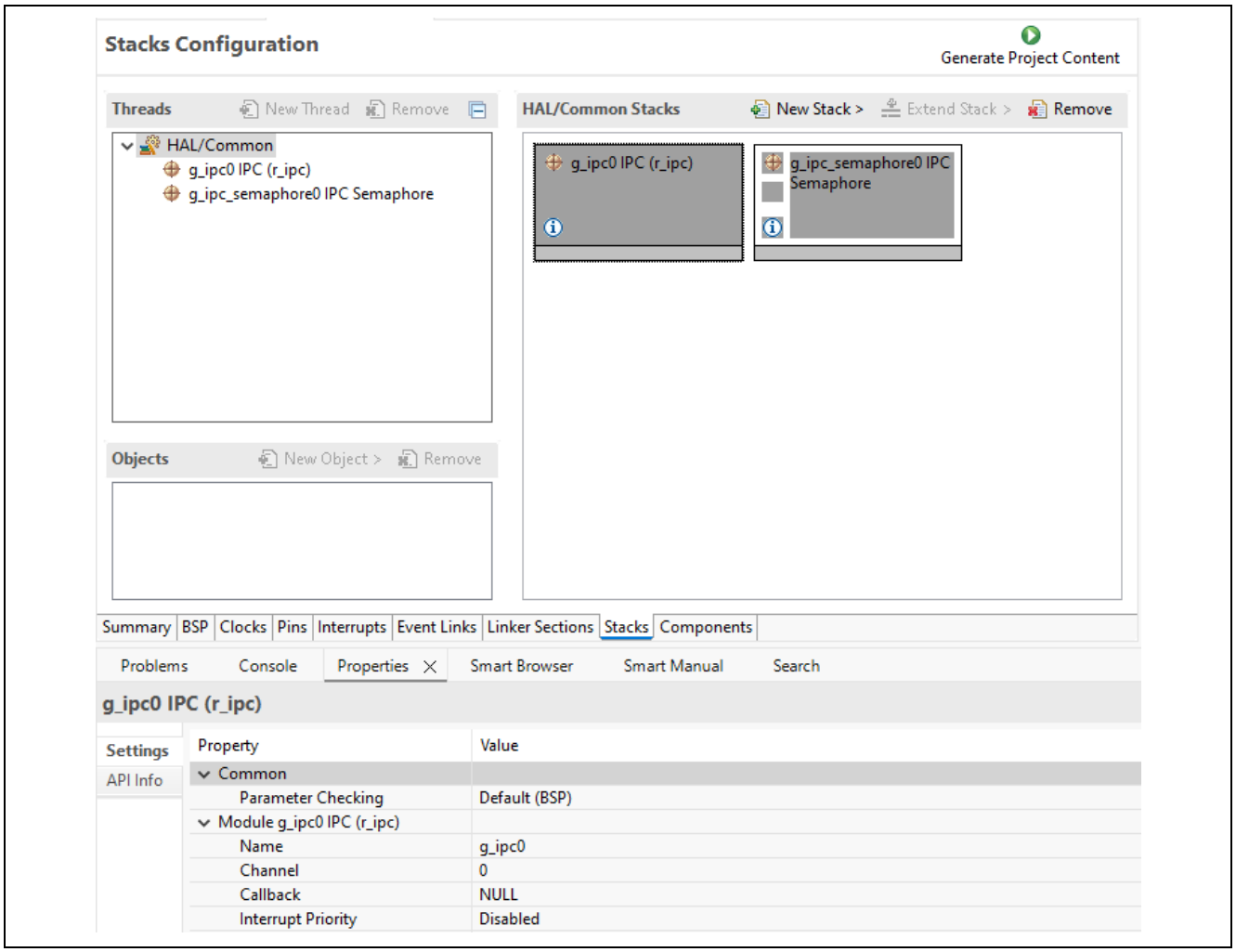

Figure 7. IPC Module Configuration

5.2 Usage Notes: IPC Interrupts and Channel Configuration

In the Renesas FSP IPC implementation, maskable interrupts and FIFOs share the same interrupt line. As a result, the IPC driver configures both mechanisms uniformly to ensure consistent behavior across communication types.

5.2.1 Interrupt and Callback Behavior

- An interrupt and callback function are only required when the IPC instance is configured for receiving messages or events.
- If the IPC is only used for sending, no interrupt or callback setup is necessary.
- When receiving, a properly configured callback function is essential. It will be invoked in response to interrupts, enabling the receiving core to process incoming messages or events.

5.2.2 Channel Matching Between Cores

- IPC channels must be consistently configured on both communicating cores. For reliable communication, the channel number used by one core must exactly match the configuration on the other.
- For example, to enable communication from Core 0 to Core 1 over channel 0:
    - Core 0 must have an IPC instance configured and opened on channel 0.
    - Core 1 must also have an IPC instance configured and opened on channel 0.
- Misalignment in channel configuration between cores will lead to communication failure or undefined behavior.

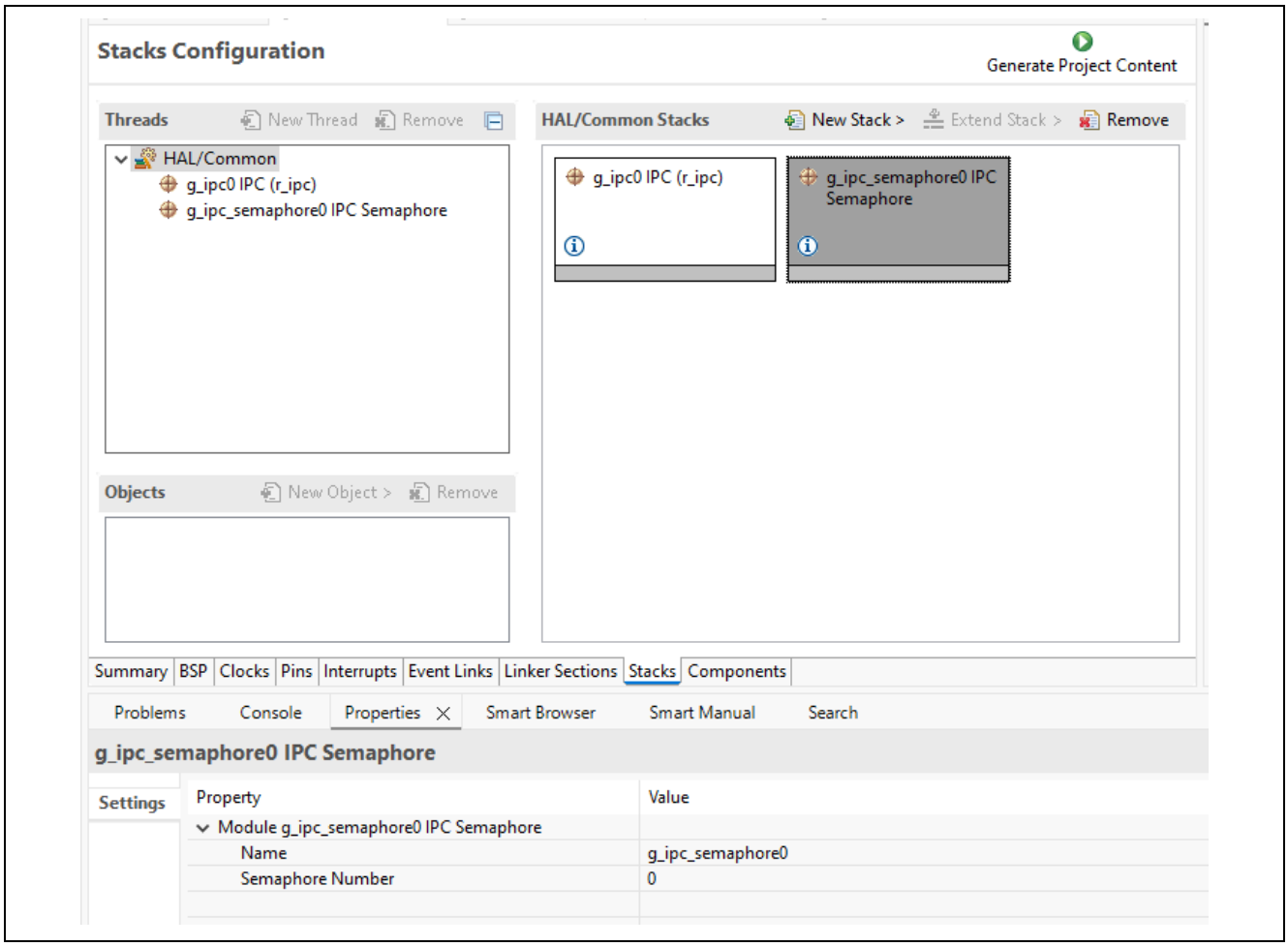

Figure 8. Semaphore Module Configuration

A dedicated IPC Semaphore Stack module is available to help manage the Semaphore and its configuration.

- This module can be added in e² studio via:
  Stacks tab → New Stack → System → IPC Semaphore
- When configuring the module, the user must select a specific semaphore number, which determines which hardware semaphore register is utilized.

5.2.3 IPC Semaphore Support

FSP-supported hardware-based IPC semaphores are useful for synchronizing access to shared resources between cores.

- The functions R_BSP_IpcSemaphoreTake() and R_BSP_IpcSemaphoreGive() are used to acquire and release semaphores, respectively.
- Each semaphore operation requires a semaphore_handle, which must be associated with a valid hardware semaphore number.

5.2.4 NMI Support for IPC

The Renesas FSP supports inter-core NMIs, allowing one core to send urgent, high-priority interrupt requests to another.

- To enable reception of IPC NMIs, the receiving core must call R_BSP_IpcNmiEnable(). This function enables the NMI and sets up a callback to handle the interrupt.
- To trigger an NMI on the opposing core, use R_BSP_IpcNmiRequestSet() from the sending core. This causes an immediate, non-maskable interrupt to occur on the target core.
- This mechanism is useful for signaling critical conditions or urgent messages that must not be delayed by normal interrupt masking.

6. Example Projects and Reference Applications

This section provides sample code, example implementations, and reference projects that demonstrate how to use the IPC feature on the RA8 dual-core MCU. These resources serve as practical guides for developers, helping them understand IPC setup and usage, and can be used directly or adapted for their own RA8 dual-core-based application development.

6.1 Sample Code for IPC Implementation

This Sample code illustrates how to use the IPC interface to send a simple message from Core 0 to Core 1. It demonstrates the fundamental steps involved: opening an IPC instance, sending a message through the FIFO from Core 0, and handling the received message via a callback on Core 1. This provides a clear starting point for implementing core-to-core communication in dual-core RA MCUs.

Core 0 Example

```
#define IPC_SEND_MESSAGE     0xABCD

void r_ipc_basic_messaging_core0_example (void)
{
    /* Open an IPC instance on the core 0 application configured to send a message to core 1.
     * @ref ipc_cfg_t::direction should be set to IPC_DIRECTION_SEND.
     * @ref ipc_cfg_t::channel should match for IPC instances on both cores.
     */
    fsp_err_t err = R_IPC_Open(&g_ipc0_ctrl, &g_ipc0_cfg);
    assert(FSP_SUCCESS == err);

    /* Send IPC_SEND_MESSAGE to core 1. */
    err = R_IPC_MessageSend(&g_ipc0_ctrl, IPC_SEND_MESSAGE);
    assert(FSP_SUCCESS == err);
}
```

Core 1 Example

```
#define RESET_FLAG 0x00
#define SET_FLAG   0x01

uint32_t g_received_message = RESET_FLAG;
uint32_t g_is_message_received = RESET_FLAG;

/***********************************************************************************************************************
 * @brief     IPC callback
 *
 * @param[in] ipc_callback_args_t: The callback arguments.
 * @return    None
 **********************************************************************************************************************/
void ipc_callback(ipc_callback_args_t *p_args)
{
    switch (p_args->event) {
      case IPC_EVENT_MESSAGE_RECEIVED:
        {
            g_received_message = (uint32_t)p_args->message;
            g_is_message_received = SET_FLAG;
            break;
        }
     default:
            break;
    }
}

void r_ipc_basic_messaging_core1_example (void)
{
    /* Open an IPC instance on the core 1 application configured to send a message to core 0.
     * @ref ipc_cfg_t::direction should be set to IPC_DIRECTION_RECEIVE .
     * @ref ipc_cfg_t::channel should match for IPC instances on both cores.
     */
    fsp_err_t err = R_IPC_Open(&g_ipc0_ctrl, &g_ipc0_cfg);
    assert(FSP_SUCCESS == err);

     /* Wait for event */
    while (!g_is_message_received)
    {
       ;
    }

    /* Message will be in g_message_received  */
    (void) g_received_message;
}
```

The code below demonstrates how to trigger an event via a maskable interrupt from Core 0 to Core 1 using the IPC interface. Core 0 sends an interrupt signal with a specified event bit, while Core 1 listens for and responds to the event via a registered callback. This technique is useful for signaling asynchronous events between cores without sending data.

Core 0 Example

```
void r_ipc_basic_interrupt_event_core0_example (void)
{
    /* Open an IPC instance on the core 0 application configured to send to core 1.
     * @ref ipc_cfg_t::direction should be set to IPC_DIRECTION_SEND.
     * @ref ipc_cfg_t::channel should match for IPC instances on both cores.
     */
    fsp_err_t err = R_IPC_Open(&g_ipc0_ctrl, &g_ipc0_cfg);
    assert(FSP_SUCCESS == err);

    /* Trigger interrupt on core 1 with event 0 bit set. */
    err = R_IPC_EventGenerate(&g_ipc0_ctrl, IPC_GENERATE_EVENT_IRQ0);
    assert(FSP_SUCCESS == err);
}
```

Core 1 Example

```
#define false 0x00
#define true  0x01


uint32_t g_event0_triggered = false;

void r_ipc_basic_interrupt_event_core1_example (void)
{
    /* Open an IPC instance on the core 1 application configured to receive.
     * @ref ipc_cfg_t::direction should be set to IPC_DIRECTION_RECEIVE.
     * @ref ipc_cfg_t::channel should match for IPC instances on both cores.
     */
    fsp_err_t err = R_IPC_Open(&g_ipc0_ctrl, &g_ipc0_cfg);
    assert(FSP_SUCCESS == err);

    /* Wait for event */
    while (!g_event0_triggered)
    {
        ;
    }
}

void ipc_event_example_callback (ipc_callback_args_t * p_args)
{
    /* Check for message received event */
    if (IPC_EVENT_IRQ0 & p_args->event)
    {
        g_event0_triggered = true;
    }
}
```

6.2 Sample Example Code Implementation of IPC in RA8P1 MCUs

The bundled sample projects included with this application note demonstrate multiple IPC mechanisms available on the device. These projects are designed to help developers understand and experiment with different methods of exchanging data and synchronizing operations between the two cores:

1. **Message FIFO-based IPC**
    - One project showcases the usage of the Message FIFO as an IPC mechanism.
    - In this method, one core places an event or 32-bit data into the FIFO, while the other core retrieves it in a synchronized manner.
    - This approach is particularly useful for ordered message exchange and ensures a structured flow of communication without requiring complex handshaking logic.
2. **Interrupt-based IPC**
    - Additional sample projects demonstrate the use of maskable interrupts (IRQ) and non-maskable interrupts (NMI) as IPC mechanisms.
    - Using interrupts enables one core to quickly notify or signal the other core about an event, eliminating the need for polling and ensuring a low-latency response.
    - The use of both maskable and non-maskable interrupts provides developers with flexibility to prioritize IPC events depending on system requirements.
3. **Semaphore-based IPC**
    - Another sample project highlights the semaphore feature of the IPC.
    - Semaphores are particularly useful for coordinating access to shared resources between the two cores.
    - By using semaphores, the project demonstrates how to implement mutual exclusion (mutex) and avoid race conditions when both cores attempt to access or update shared data simultaneously.

4. **Shared Memory-based IPC**
    - Finally, a project is included to showcase IPC through shared memory regions.
    - In this mechanism, both cores read from and write to a common memory area, enabling direct data sharing.
    - The project illustrates how shared memory can be combined with synchronization mechanisms like semaphores or interrupts to ensure data integrity and consistency across the cores.

Together, these projects not only demonstrate the individual IPC mechanisms but also provide developers with practical references on when and how to apply each method depending on the application's real-time requirements, data volume, and synchronization needs.

6.3 Using FreeRTOS for IPC in Dual-Core Environments

FreeRTOS provides standard RTOS-level IPC mechanisms (such as queues, semaphores, and message buffers) for communication within a single core. These middleware objects are commonly used in FreeRTOS-based applications to efficiently manage inter-task communication.

FreeRTOS IPC Across Dual Cores: Concept vs. Reality

While it is technically possible to extend FreeRTOS IPC mechanisms across dual-core systems, this is not currently supported on FSP for the Renesas RA8 dual-core MCU, due to its heterogeneous core architecture. In the RA8 dual-core MCU, the two CPU cores are different (e.g., Cortex-M85 and Cortex-M33), and FreeRTOS is not natively integrated to support shared IPC constructs across both.

As a result, direct communication between FreeRTOS instances running on each core is not feasible using standard RTOS constructs. Instead, inter-core communication must rely on the MCU's hardware IPC mechanisms (e.g., hardware semaphores, events, and FIFO messaging via FSP IPC drivers).

Alternative: Application-Level IPC Frameworks

To achieve a level of abstraction and ease of use similar to FreeRTOS IPC across cores, developers can create custom application-level IPC frameworks. These frameworks can wrap around the hardware-based IPC features to mimic FreeRTOS-style messaging and synchronization, enabling more intuitive inter-core communication in dual-core applications.

Such an approach helps bridge the gap between high-level RTOS design patterns and low-level hardware IPC, improving maintainability and scalability in complex embedded systems.

To simplify inter-core communication on the RA8 dual-core MCUs, a FreeRTOS-based wrapper framework can be developed. The sample application referenced in the "Developing with RA8 Dual Core MCU - R01AN7881EU" document contains a sample framework. This framework is designed to mimic standard FreeRTOS IPC objects (such as queues and semaphores) while internally utilizing the MCU's hardware IPC features.

Purpose and Benefits

- The wrapper allows developers to use a familiar FreeRTOS-like API for inter-core messaging and synchronization.
- It abstracts the complexity of configuring and managing hardware-level IPC (e.g., semaphores, events) and presents a cleaner, application-friendly interface.
- This approach helps integrate dual-core communication seamlessly into FreeRTOS-based applications without needing to modify FreeRTOS internals.

Usage

- Developers can refer to the wrapper framework as a reference or foundation for building application-level IPC solutions.
- It can be included as middleware in the project and used alongside FreeRTOS tasks to coordinate activities between cores.

This framework bridges the gap between FreeRTOS and hardware-level IPC, providing an efficient, scalable solution for dual-core embedded development.

7. Running the Application Projects

7.1 Import the Projects

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

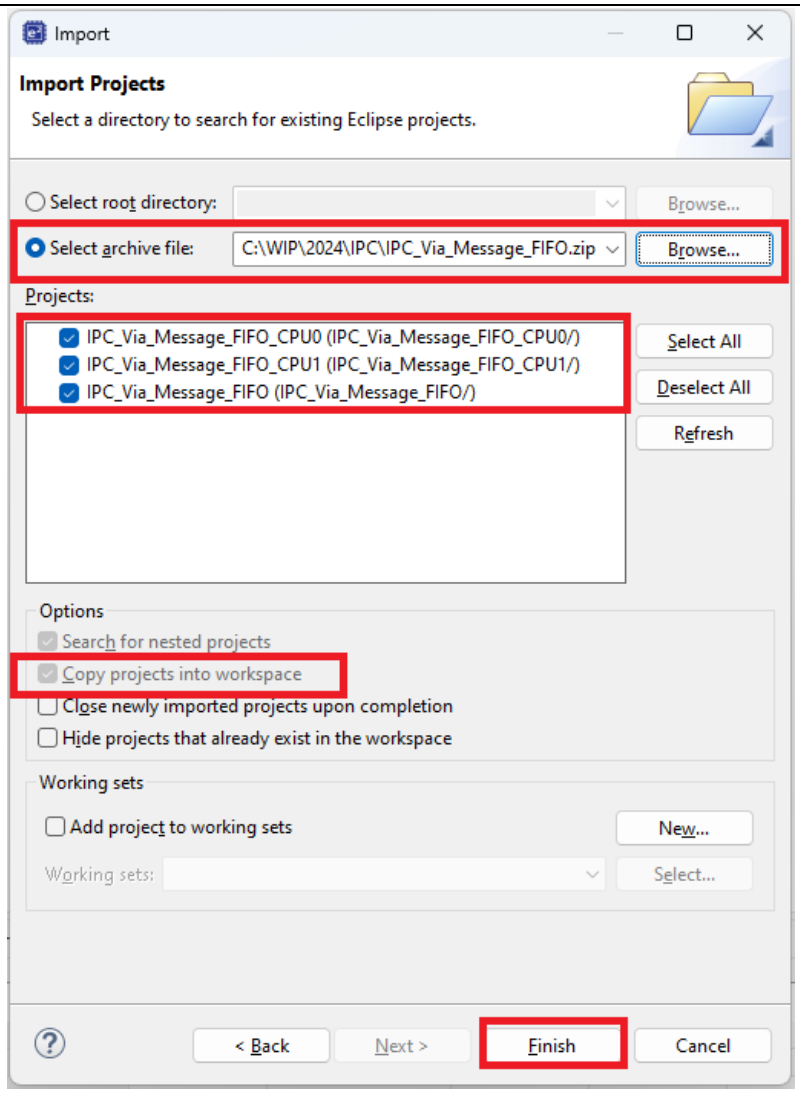

Figure 9. Importing IPC Via Message FIFO-based Project.

7.2 Build Projects

Build the solution project “IPC_Via_Message_FIFO” from the imported bundle; this will build the CPU0 project first, followed by the CPU1 project.

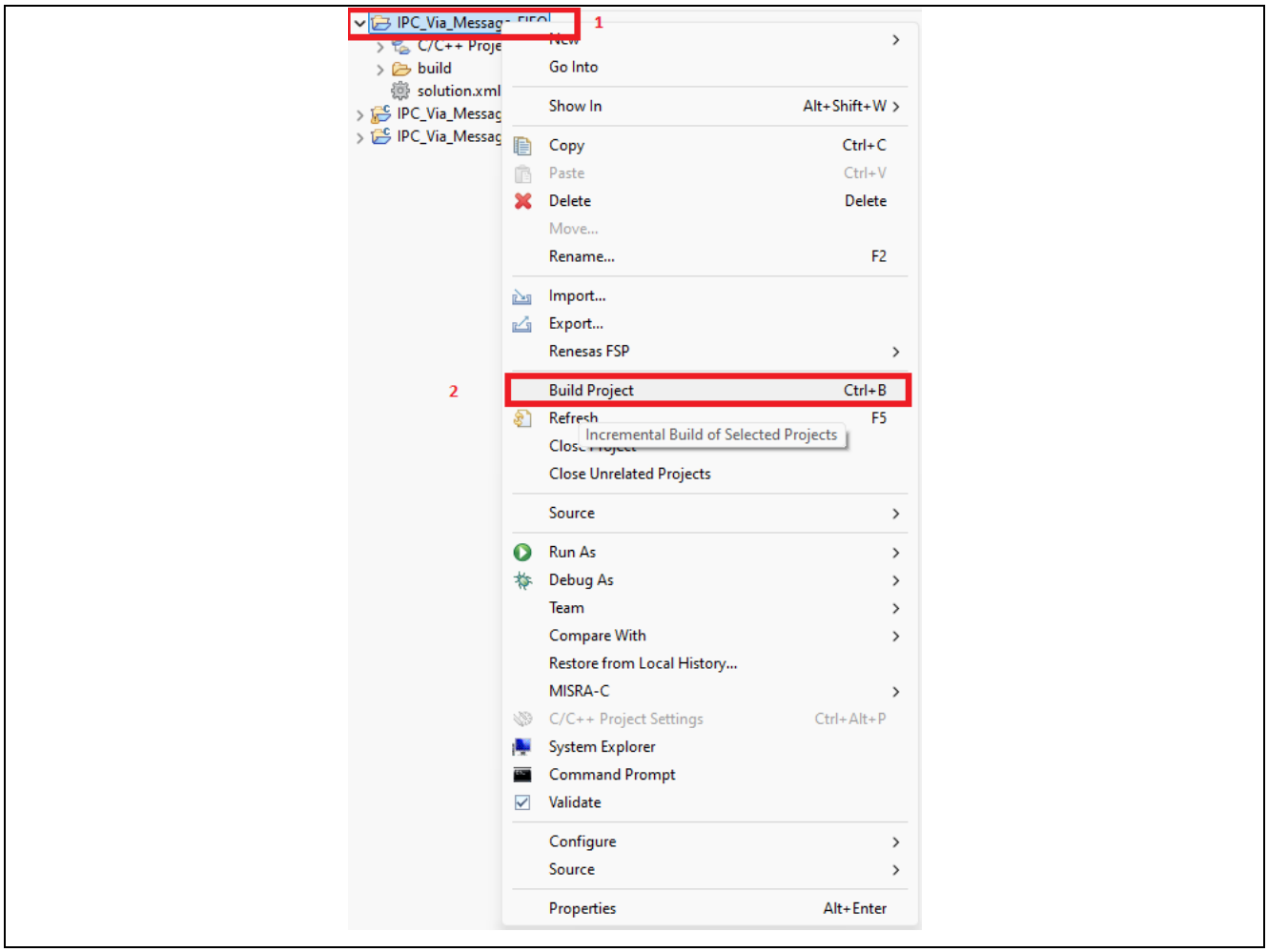

Figure 10. Example of Building IPC Via Message FIFO-based Project.

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, Run, and Verify the Projects

Initially, the device must be initialized in the OEM_PL2 state, with no TrustZone boundary settings required.

To start debugging both cores simultaneously:

1. Connect the USB-C cable to Debug connector(J10) and another end to host PC. This will be a JLink connection for downloading the image and for the virtual console (JLink CDC UART Port) for user interaction via a serial terminal.
2. Open Debug Configurations as shown in Figure 11.
3. Select “IPC_Via_Message_FIFO_CPU1 Debug_Multicore” as shown in Figure 12.
4. Click Debug to launch the dual-core debug session.

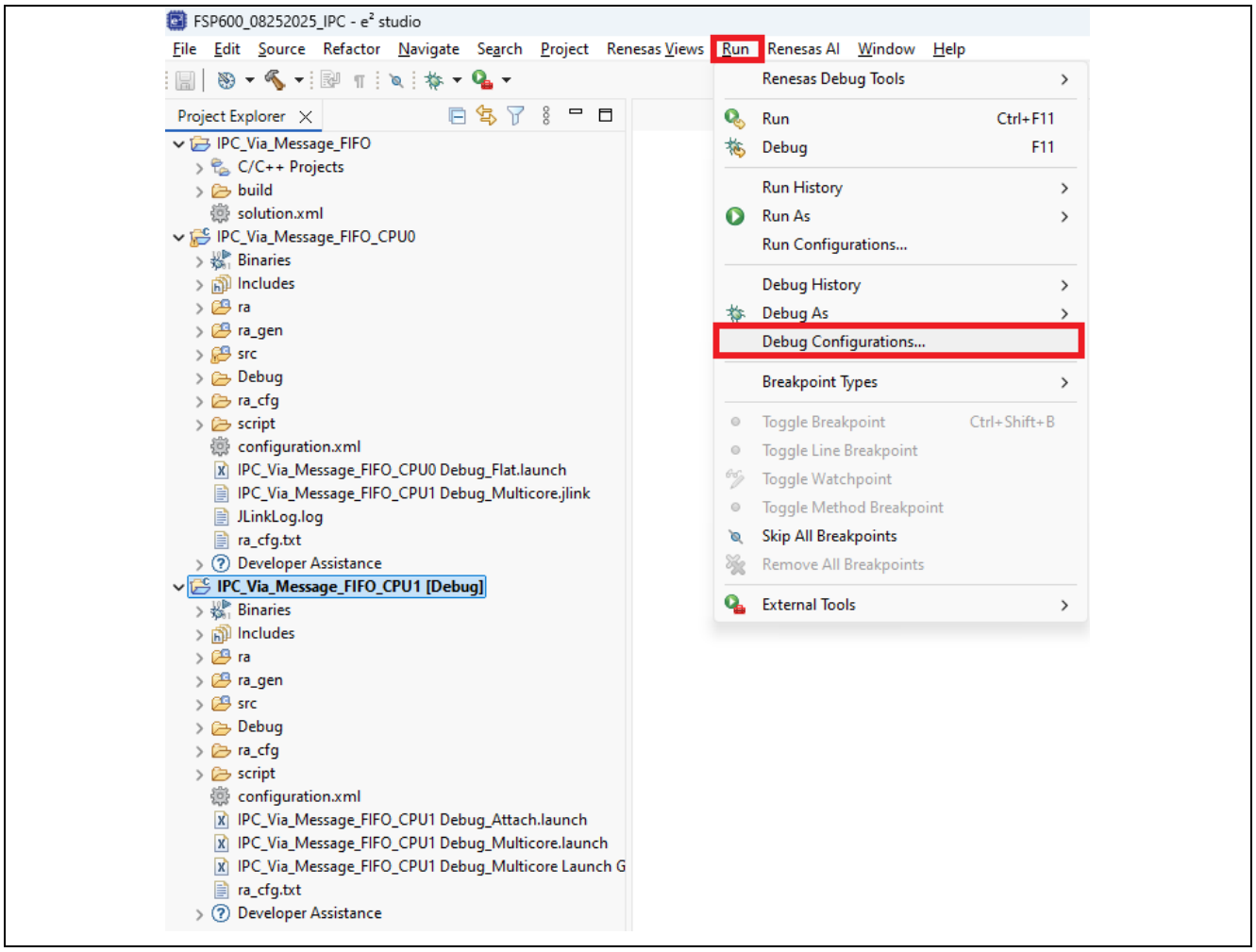

Figure 11. Example of Open Debug Configuration in IPC Via Message FIFO Example

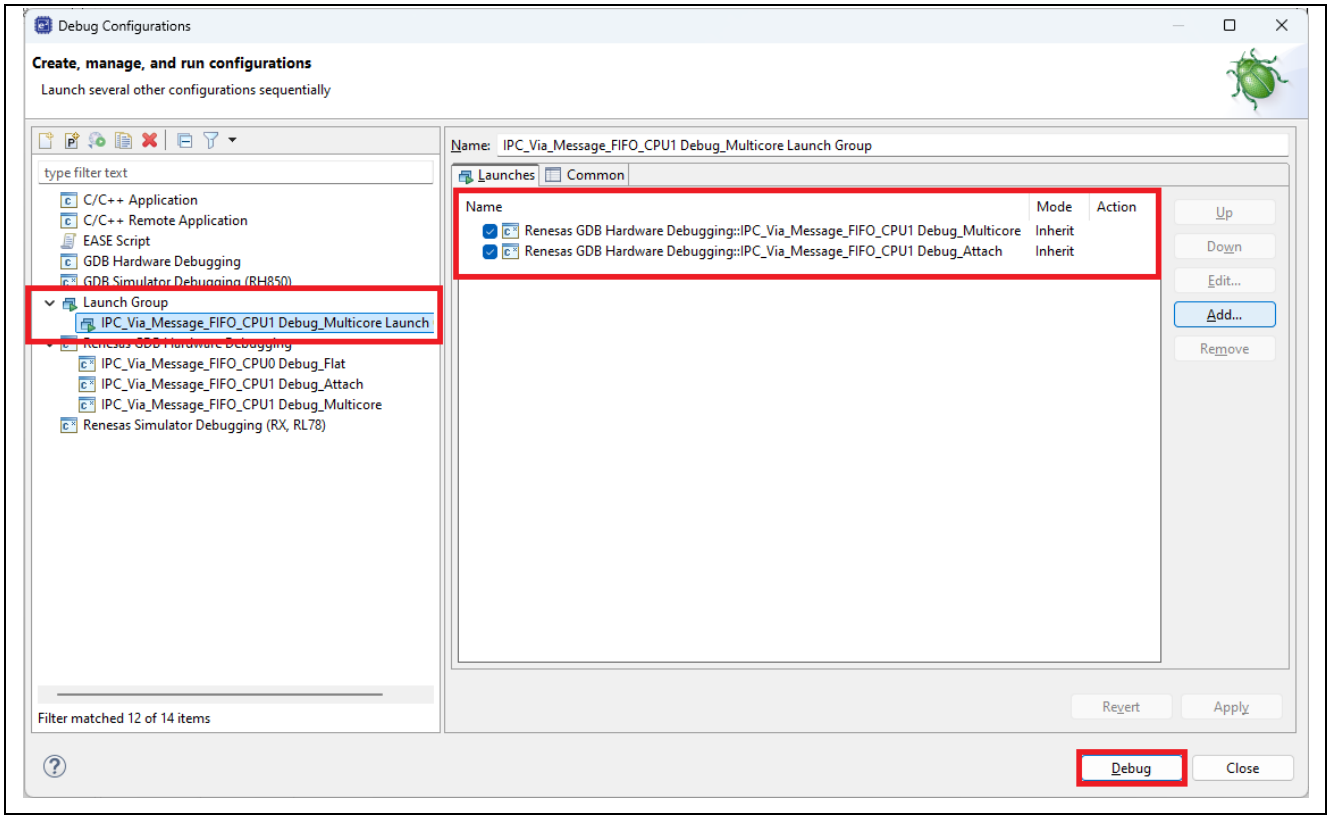

Figure 12. Example of Debug IPC Via Message FIFO Example

7.4 Verifying the Application

Open the serial terminal (File→New connection→serial) and connect to the J-Link CDC UART Port with the following settings: Baud rate 115200 bps, 8-bit data, parity none, one stop bit, and no flow control. Figures 13 and 14 illustrate the connection process and configuration steps using the Tera Term terminal.

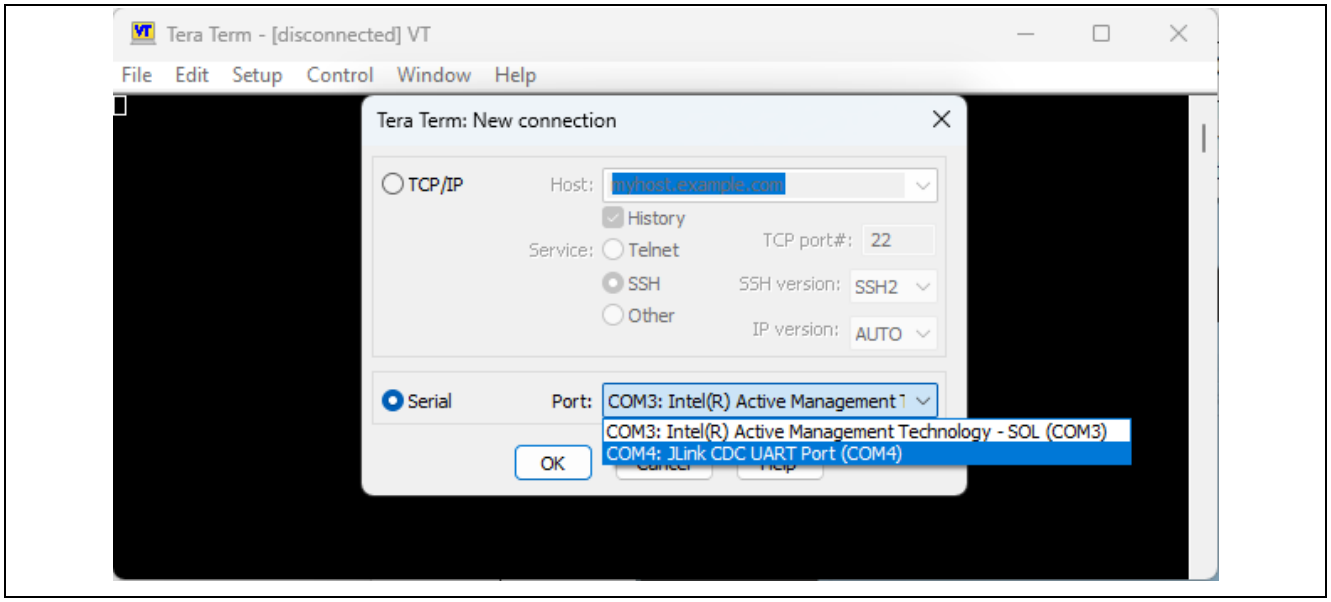

Figure 13. Example of Connect to JLink CDC UART Port

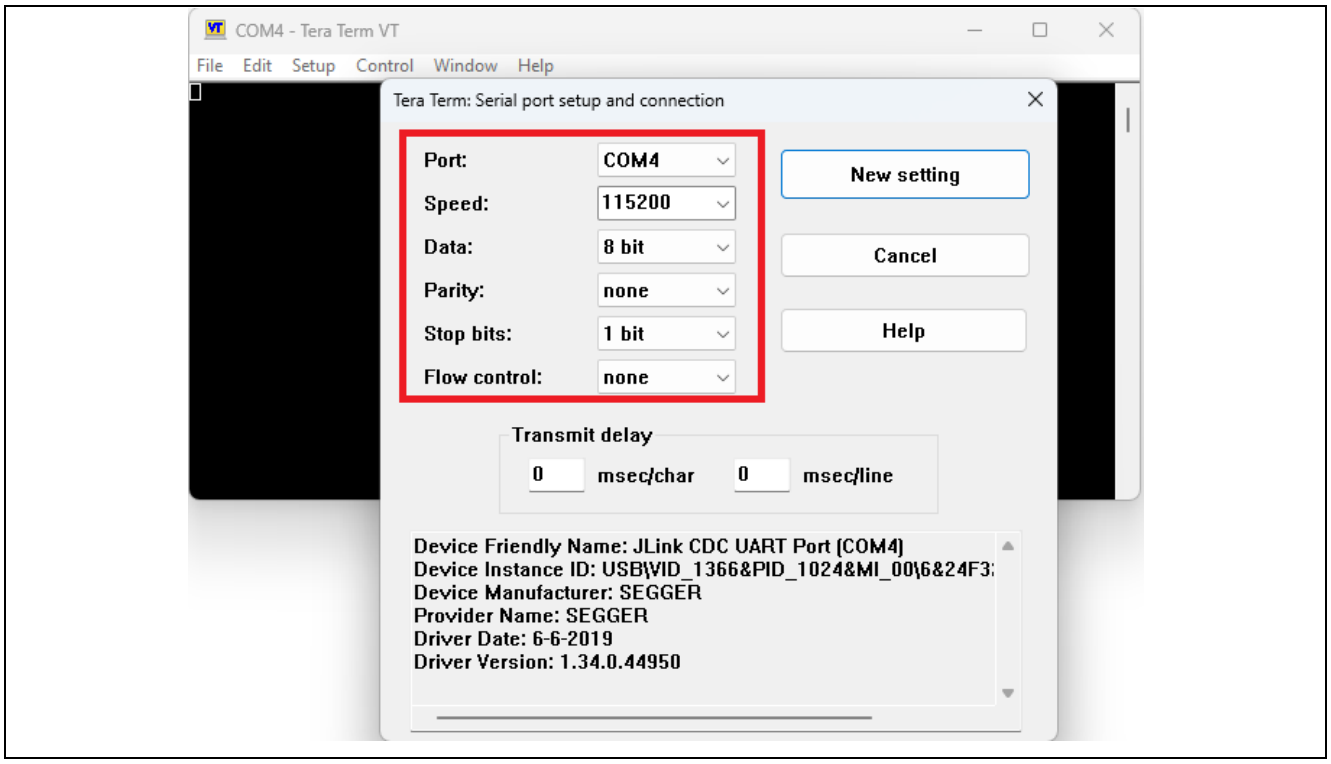

Figure 14. Example of Set Up Tera Term Serial Terminal

Return to e² studio and press Resume three times to execute the application across both cores. Upon completion, the example IPC via Message FIFO will output its status to the terminal, as illustrated in Figure 15.

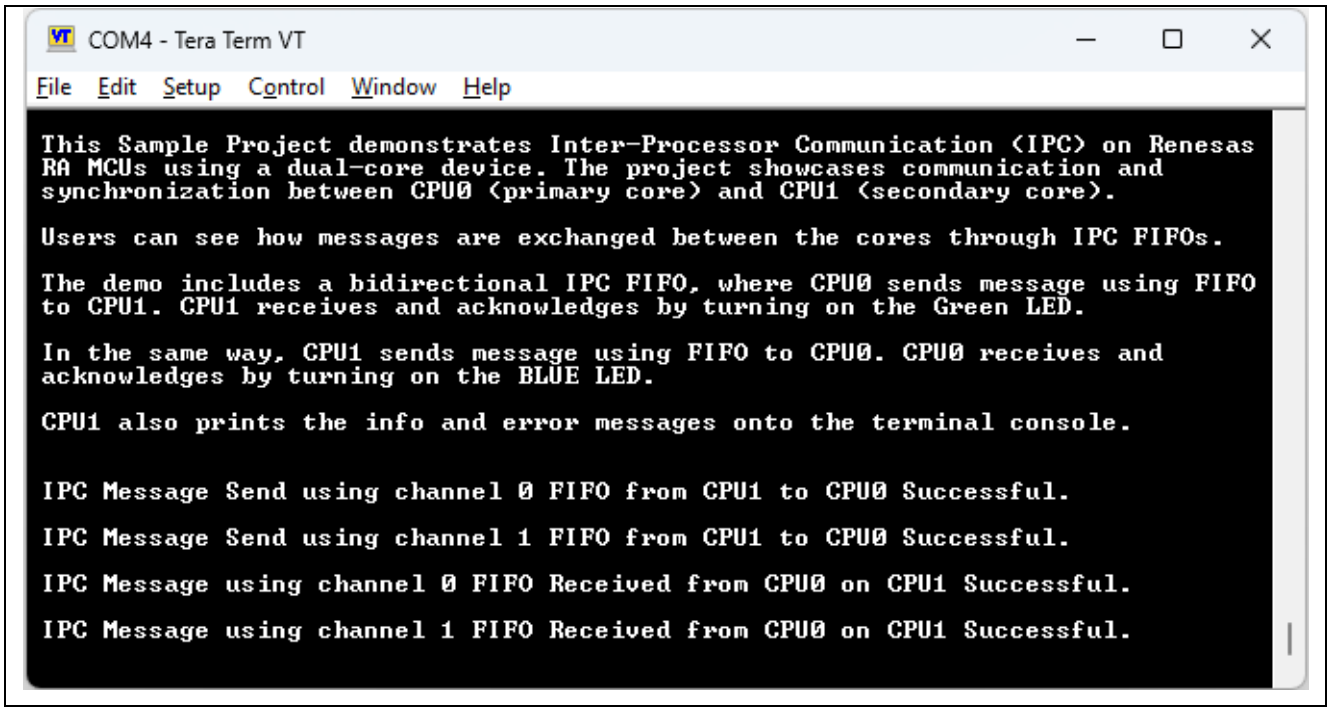

Figure 15. Console log upon successful Demonstration of IPC Via the Message FIFO example

Note: The Project bundle contains 4 projects to demonstrate various IPC features. Users can follow the above-mentioned steps to Import, build, run, and verify the projects. Each Project demonstrates the distinctive features of the IPC available in RA8P1.

The following sections provide a snapshot of the sample projects along with their expected outcomes. Each project highlights a unique IPC method for dual-core RA MCUs. While the overall objective is to demonstrate communication between the two cores, each project focuses on a different IPC mechanism, allowing users to understand the versatility of available options.

1. IPC_Via_Event_Interrupt
This project demonstrates the use of standard interrupt events as an IPC mechanism.

- CPU0 can trigger an interrupt event, which is serviced by CPU1, and vice versa.
- This approach is useful for event-driven synchronization, where one core signals the other upon completion of a specific task.
- The project shows how event-based signaling enables timely communication with low overhead.

2. IPC_Via_Message_FIFO
This project utilizes message FIFOs for inter-core data transfer.

- A 32-bit message is written by one CPU and read by the other, ensuring orderly communication.
- FIFOs are particularly useful for scenarios where multiple messages need to be queued and processed in sequence.
- This mechanism provides a reliable way of exchanging structured data streams between cores.

3. IPC_Via_NMI_Interrupt
Here, Non-Maskable Interrupts are employed as the IPC signaling method.

- Similar to event interrupts, NMIs allow one CPU to alert the other.
- However, unlike regular interrupts, NMIs cannot be disabled or masked by the CPU NVIC, ensuring that critical inter-core signals are always serviced.
- This mechanism is valuable for urgent communication, such as fault notifications or safety-critical signaling.

4. IPC_Via_Shared_Memory

In this project, a designated region of system RAM is reserved as shared memory, accessible by both cores.

- CPU0 and CPU1 can directly read and write to this shared region to exchange data.
- Shared memory provides the highest flexibility, enabling bulk data transfer, buffer sharing, and complex data structures.
- Synchronization mechanisms can be layered on top to ensure data consistency and avoid conflicts.

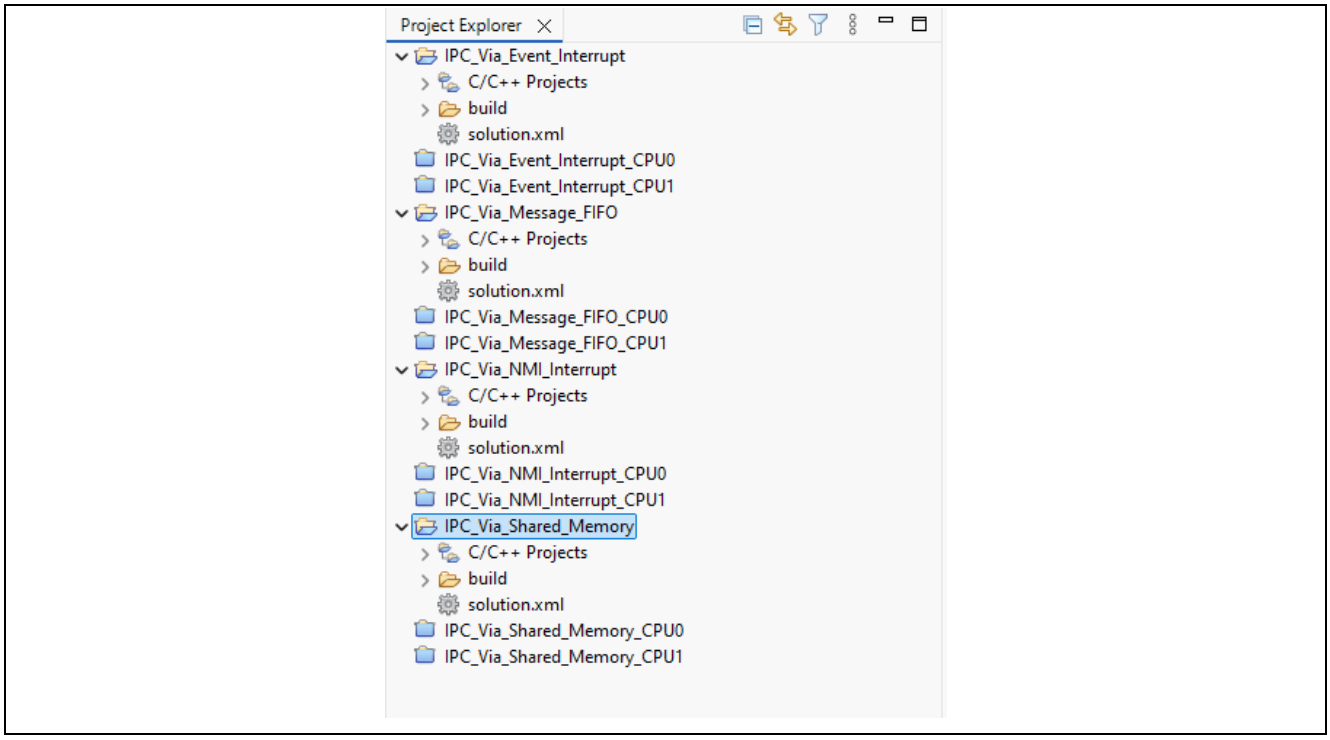

Figure 16. List of Projects in the bundle

Users can run and validate the results similar to the “IPC Via Message FIFO” Example. The following snapshots show the expected results of the Sample Projects.

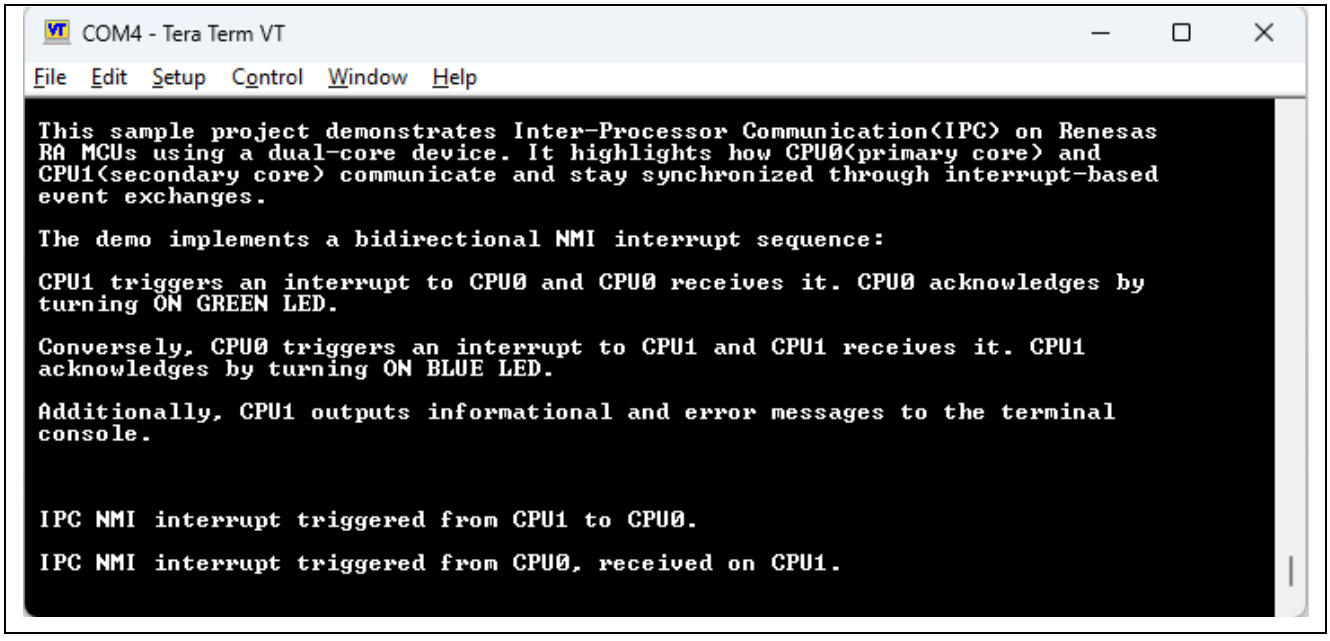

Figure 17. Console log upon successful Demonstration of IPC Via NMI Interrupt example

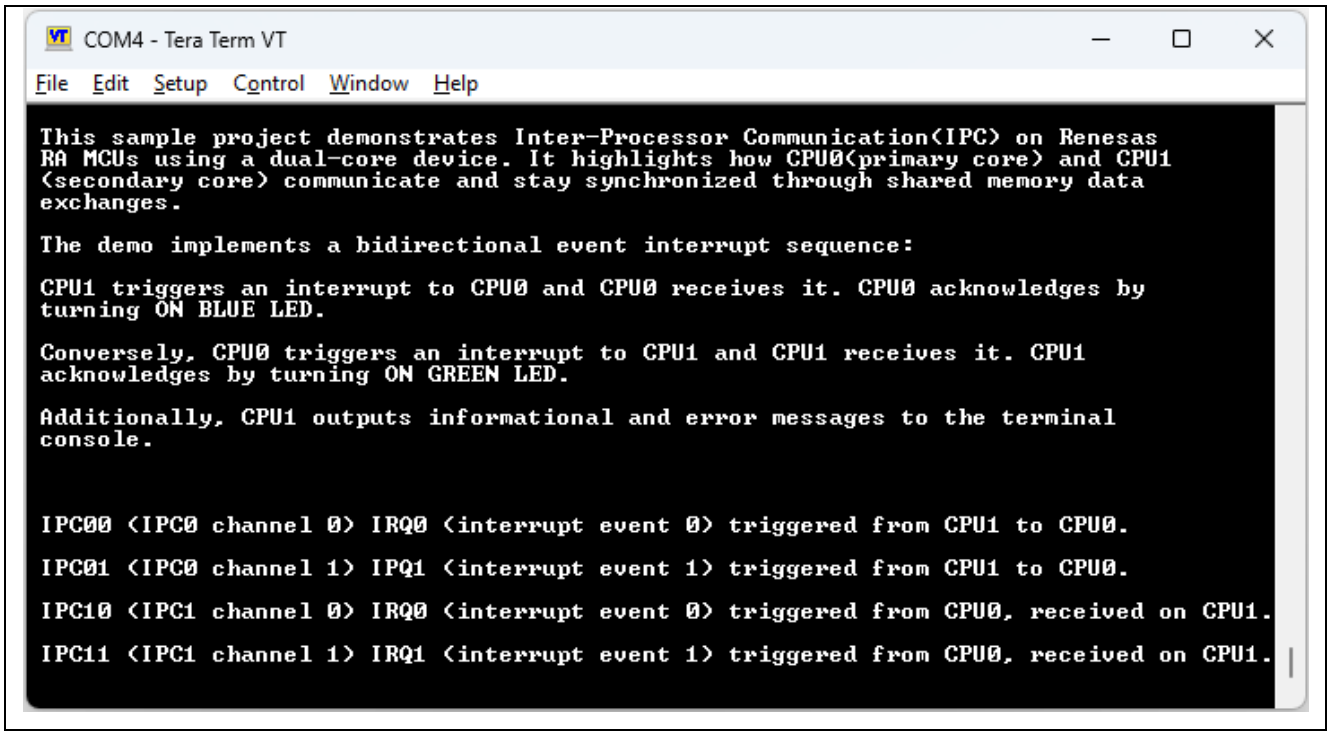

Figure 18. Console log upon successful Demonstration of IPC Via Event Interrupt example

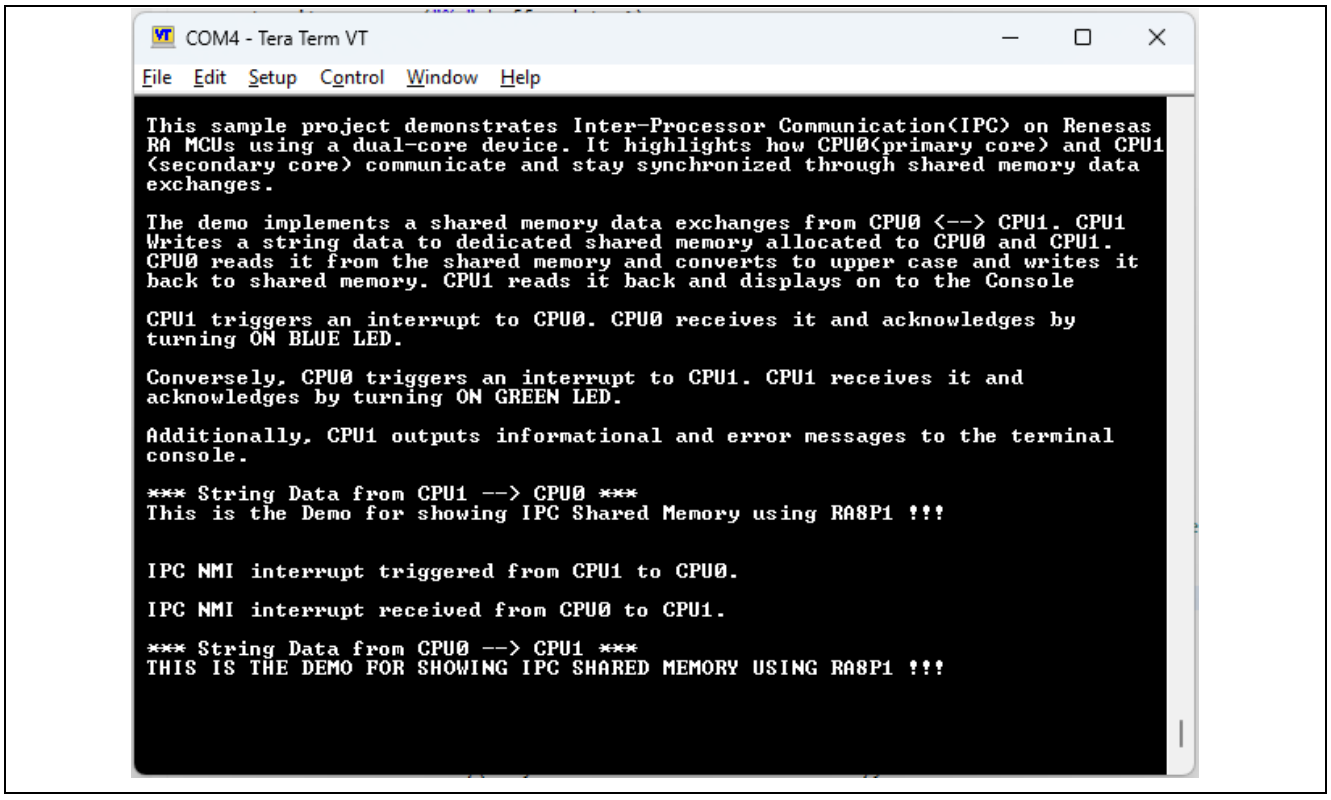

Figure 19. Console log upon successful Demonstration of IPC via Shared Memory example

8. Debugging and Troubleshooting

1. To debug both cores simultaneously, it is required to begin by initializing your MCU using either the Renesas Flash Programmer or the Renesas Device Partition Manager. This step ensures that the device is set to Protection Level 2 and that the TrustZone boundary has not already been configured. If the boundary has been previously set, you must reset it to establish a suitable environment for debugging.

Completing this initialization is essential; without it, you may encounter issues when downloading project images or starting the debug session.

2. Also, using the Launch group, which combines individual launch configurations, helps to run and debug the dual-core projects.

3. Additionally, make sure “R_BSP_SecondaryCoreStart()” gets called from the CPU0 project to run the CPU1 core. For debugging purposes, add a breakpoint to make sure this code is invoked to confirm the CPU1 code gets invoked.

9. Next Steps

• To learn more about the EK-RA8P1 kit, refer to the EK-RA8P1 user’s manual and design package available in the Documents and Download tabs, respectively, of the EK-RA8P1 webpage at renesas.com/ek-ra8p1. Similarly, to know about MCK-RA8T2, EK-RA8M2, and EK-RA8D2, refer to their respective webpages, similar to RA8P1.

• To learn how to create a new e^2 studio project from scratch, refer to Chapter 2, Starting Development, in the FSP User Manual (renesas.com/ra/fsp). To learn how to use e^2 studio, refer to the user manual provided on the e^2 studio webpage (renesas.com/software-tool/e-studio).

• Renesas provides several example projects that demonstrate different capabilities of the RA and the MCUs. These example projects can serve as a good starting point for users to develop custom applications.

• Refer to the application note “Developing with RA8 Dual Core MCU” (Document No. R01AN7881EU), which demonstrates the usage of various IPC features within a single project. In this example, the RTC and ADC peripherals are used to demonstrate various IPC capabilities on the RA8P1 MCU. This application project serves as an excellent starting point for users looking to develop their own custom applications.

10. References

A collection of essential references, datasheets, and documentation related to dual-core memory architecture and system design.

The following documents can be referred to for a more detailed understanding of RA8 dual-core and its support

- Renesas FSP User’s Manual: https://renesas.github.io/fsp
- Renesas RA Flash Memory Programming: https://www.renesas.com/us/en/document/apn/flash-memory-programming
- Renesas RA MCU Datasheets: See http://renesas.com/ra and select the relevant MCU
- RA8P1, RA8M2, RA8T2, and RA8D2 Example Projects on Renesas RA GitHub: https://github.com/renesas/ra-fsp-examples
- Getting Started with IPC on Dual Core MCU
- Renesas RA Family RA8P1 User’s Manual: Hardware (R01UH1064)
- Renesas RA Family RA8M2 User’s Manual: Hardware (R01UH1066)
- Renesas RA Family RA8D2 User’s Manual: Hardware (R01UH1065)
- Renesas RA Family RA8T2 User’s Manual: Hardware (R01UH1067)

- Renesas RA Family RA8 Quick Design Guide (R01AN7087).
- *High Performance with RA8 MCU using CM85 core with Helium™-(R01AN7127)*
- EK-RA8P1, MCK-RA8T2, EK-RA8M2, and EK-RA8D2 Board Design Schematics.
- Developing with RA8 Dual-Core MCU (R01AN7881)
- RA8P1 Multicore setup and running Hello World on dual core (R01AN7982)
- RA8P1 MCU Quick Design Guide (R01AN7883)
- Getting Started with RA8 Dual-Core MCU Memory Architecture, Configurations, and Topologies (R01AN7880)
- 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.

EK-RA8P1 Resources	www.renesas.com/ek-ra8p1
MCK-RA8T2 Resources	www.renesas.com/mck-ra8t2
EK-RA8M2 Resources	www.renesas.com/ek-ra8m2
EK-RA8D2 Resources	www.renesas.com/ek-ra8d2
RA Product Information	renesas.com/ra
RA Product Support Forum	renesas.com/ra/forum
RA Flexible Software Package	renesas.com/FSP
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jul.30.25	-	Initial version
1.01	Oct.16.25	-	Added Support for RA8T2,RA8M2 and RA8D2.

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/.