Renesas RA Family

RA2 MCU Advanced Secure Bootloader Design using MCUboot Internal Code Flash and Memory Mirror Function

Introduction

MCUboot is a secure bootloader for 32-bit MCUs. It defines a common infrastructure for the bootloader, defines system flash layout on microcontroller systems, and provides a secure bootloader that enables easy software update. MCUboot is an operating system and hardware independent and relies on hardware porting layers from the operating system it works with. Users can benefit from using the FSP MCUboot Module to create a Root of Trust (RoT) for the system and perform secure booting and fail-safe application updates.

MCUboot is maintained by Linaro in the GitHub mcu-tools page https://github.com/mcu-tools/mcuboot. There is a `\docs` folder that holds the documentation for MCUboot in .md file format. This application note refers to the above-mentioned documents wherever possible and is intended to provide additional information that is related to using the Renesas FSP MCUboot Module.

For the RA family, RA2A2 and RA2A1 MCU Groups support the Memory Mirror Function (MMF). This application explains how to design a secure bootloader using the MMF feature and demonstrates the benefits of combining MCUboot with MMF for the RA2 MCU Series.

Furthermore, MCUboot is a secure bootloader solution. Therefore, the implementation of MCUboot with the MMF feature is inherently designed as a security solution by default.

Example projects using the EK-RA2A2 evaluation kit are provided in this application project. Users can review the flash layout for other RA2 MCUs and port the application. In addition, steps for how to master an application to use with the bootloader and how to download and update to a new application are provided. Users can follow these steps to recreate the reference bootloader and link the example application projects included in this application project to use the bootloader.

If you are interested in the basic secure bootloader design using the MCUboot module with RA2 internal code flash in linear mode, please refer to https://www.renesas.com/en/document/apn/secure-bootloader-ra2-mcu-series-application-project?r=1470181.

For RA6 MCU group secure bootloader design using MCUboot and code flash linear mode, please refer to https://www.renesas.com/en/document/apn/ra6-basic-secure-bootloader-using-mcuboot-and-internal-code-flash?r=1353811.

For RA8 MCU group secure bootloader design using MCUboot and code flash linear mode, please refer to https://www.renesas.com/en/document/apn/ra8-basic-secure-bootloader-using-mcuboot-and-internal-code-flash?r=25448206.

Required Resources

Development tools and software

- The e^2 studio IDE v2025-01
- Renesas Flexible Software Package (FSP) v5.8.0
- SEGGER J-link® USB driver v8.12

The above three software components: the FSP, J-Link USB drivers and e^2 studio are bundled in a downloadable platform installer available on the FSP webpage at renesas.com/ra/fsp.

- Python v3.9 or later - https://www.python.org/downloads/

Hardware

- EK-RA2A2, Evaluation Kit for RA2A2 MCU Group http://www.renesas.com/ra/ek-ra2a2
- Workstation running Windows® 10
- One USB device cables (type-A male to micro-B male)

Prerequisites and Intended Audience

Users of this application project should have some experience with the Renesas e^2 studio. Users should read the **MCUboot Port** section of the FSP User's Manual as well as the MCU Hardware User's manual **Flash Memory** and **Memory Mirror Function (MMF)** sections prior to working with this application project. Users should also have some knowledge of cryptography. Prior knowledge of Python usage is also helpful.

The intended audience includes product developers, product manufacturers, product support, or end users who are involved with designing application systems involving usage of a secure bootloader.

Contents

1. RA2 MCU Group Memory Layout

For RA2A2 and RA2A1 MCU groups, the internal flash memory can operate in linear mode or dual bank mode. In this application, we use the linear mode to demonstrate. In this mode, the code flash memory is used as one area to boot a new application for a system that includes a bootloader.

1.1 RA2A2 MCU Code Flash Configuration

Based on the code flash memory size, as shown in Figure 1, users can easily calculate the bootloader size and image size in the bootloader project.

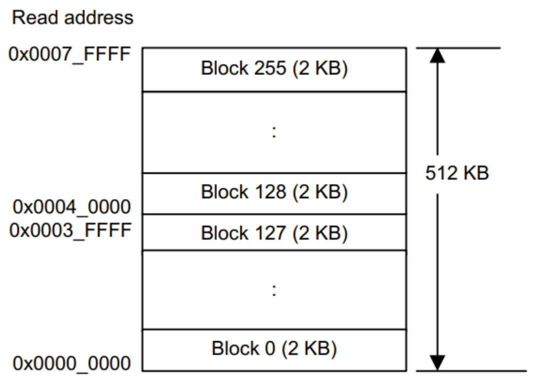

Figure 1. Mapping of the code flash memory

1.2 RA2A2 MCU Memory Mirror Address Mapping

The Memory Mirror Function (MMF) links the memory mirror space (0x0200_0000 to 0x027F_FFFF) to the code flash area, as show in Figure 2.

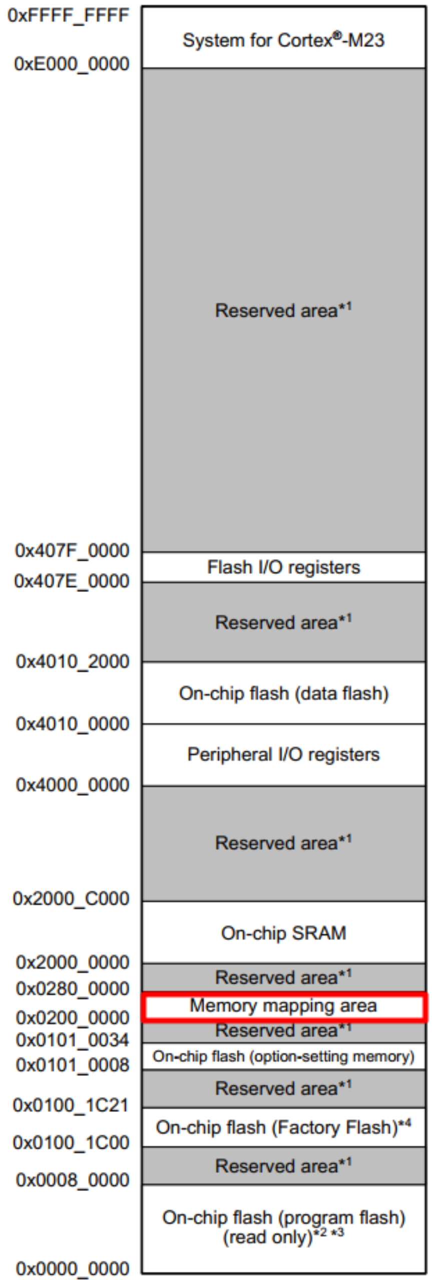

Figure 2. Memory Mapping Area

1.3 Security Memory Protection Unit and Flash Access Window

The MCU incorporates a security MPU with four secure regions that include the code flash, SRAM, and two security functions. In this application, we only use code flash security region. The secure regions can be protected from non-secure program access. A non-secure program cannot access a protected region.

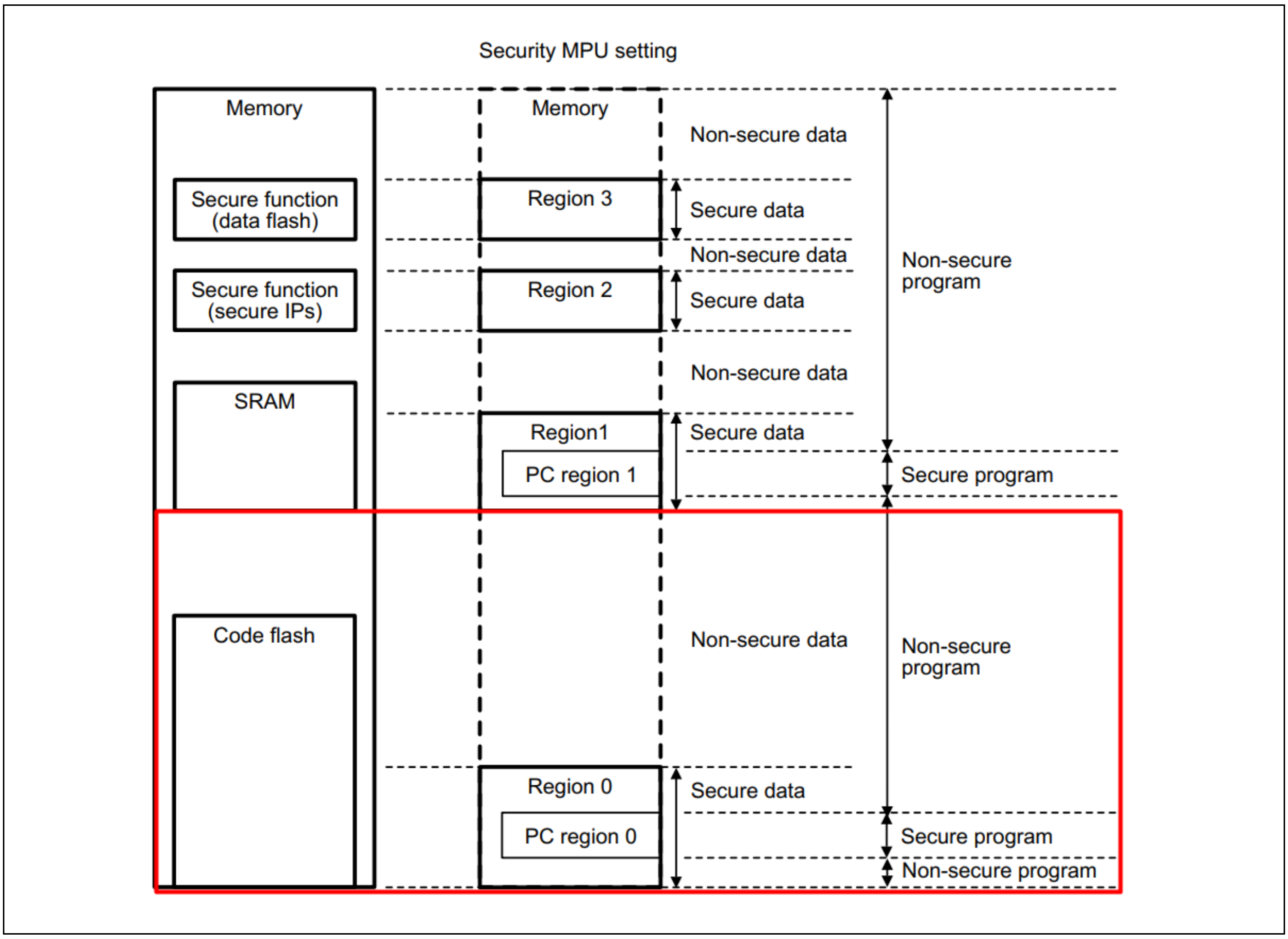

Figure 3. Security MPU Secure Regions

The Flash Access Window (FAW) defines one contiguous flash region within the MCU flash space. Within this region, flash erase and write operations are allowed from both secure and non-secure programs. The access window is only valid in the program flash area.

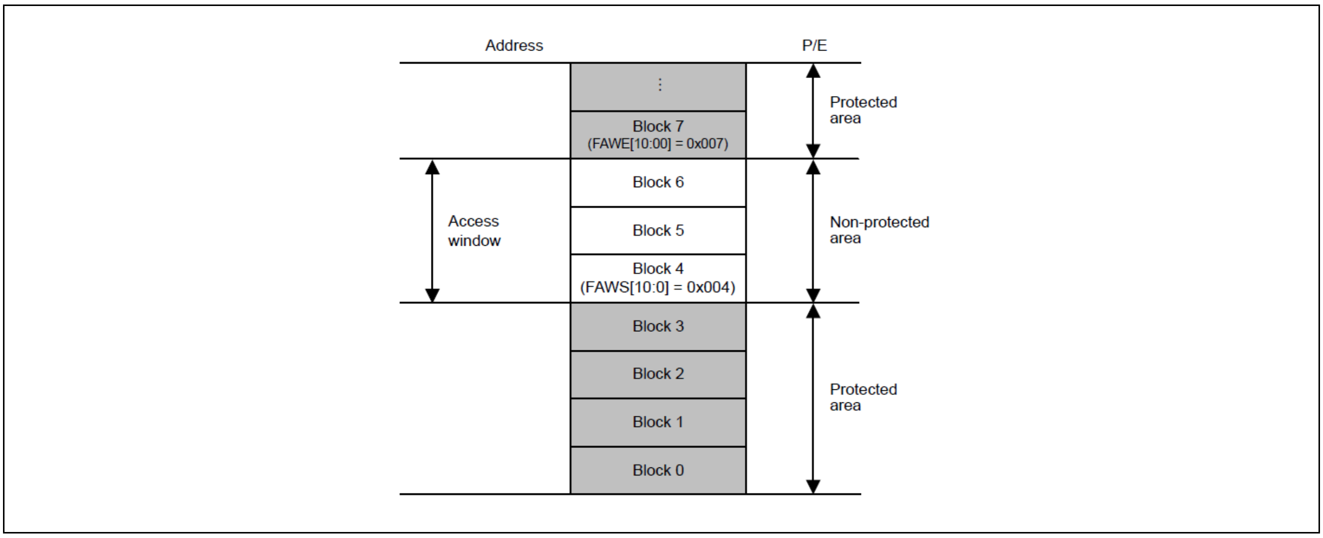

Figure 4. Flash Access Window

By combining the MPU and FAW, the bootloader's protection will be enhanced. To help users easily visualize the implementation, we will provide the diagram as shown in Figure 5.

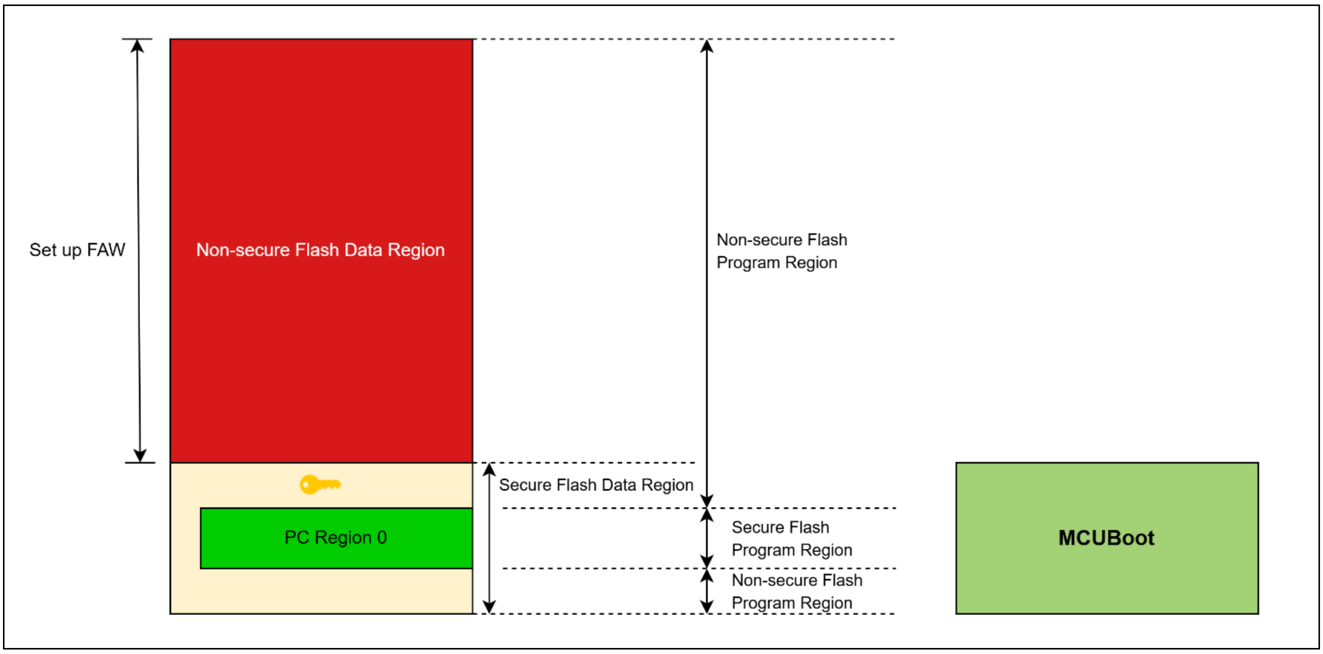

Figure 5. Combining the MPU and FAW to protect the bootloader

In addition, for manufacturing usage permanently locking the FAW region prevents a user from updating the FAW region. Users can refer to **Permanent Locking of the FAW Region** section in Application Project R11AN0416.

Note: When permanently locking the FAW region, this action is irreversible.

2. Using the Code Flash Linear Mode and MMF Feature with MCUboot Overview

MCUboot evolved out of the Apache Mynewt bootloader, which was created by runtime.io. MCUboot was then acquired by JuulLabs in November 2018. The MCUboot github repo was later migrated from JuulLabs to the mcu-tools github project. In the year 2020, MCUboot was moved under the Linaro Community Project umbrella as an open-source project.

2.1 MCUboot Functionalities Overview

MCUboot handles the firmware authenticity check after start-up and the firmware switch part of the firmware update process. Downloading the new version of the firmware is out-of-scope for MCUboot. Typically, downloading the new version of the firmware is functionality that is provided by the application project itself. This application project provides an example of downloading a new image using the XModem protocol from the application project.

The functionality of MCUboot during booting and updating follows the process below:

The bootloader starts when the CPU is released from reset. If there are images in the Secondary App memory marked as to be updated, the bootloader performs the following actions:

1. The bootloader authenticates the Secondary image.
2. Upon successful authentication, the bootloader switches to the new image based on the update method selected. Available update methods supported by FSP are overwrite, swap, and direct XIP.
3. The bootloader boots the new image.

If there is no new image in the Secondary App memory region, the bootloader authenticates the Primary applications and boots the Primary image.

The authentication of the application is configurable in terms of the authentication methods and whether the authentication is to be performed with MCUboot. If authentication is to be performed, the available methods are RSA or ECDSA. The firmware image is authenticated by hash (SHA-256) and digital signature validation. The public key used for digital signature validation can be built into the bootloader image or provisioned into the MCU during manufacturing. In the examples included in this application project, the public key is built into the bootloader images.

There is a signing tool included with MCUboot: `imgtool.py`. This tool provides services for creating Root keys, key management, and signing and packaging an image with version controls. Read the MCUboot documentation to understand and use hese operations.

2.2 Use Direct XIP Firmware Update Mode

When using direct XIP mode with code flash in linear mode, the active image slot alternates with each firmware update. If this update method is used, then two firmware update images must be generated: one of them is linked to be executed from the primary slot memory region, and the other is linked to be executed from the secondary slot. Direct XIP is supported in FSP versions 3.6.0 and later.

- Advantages:
    - Faster boot time, as there is no overwrite or swap of application images needed.
    - Fail-safe and resistant to power-cut failures.
- Disadvantages:
    - Added application-level complexity to determine which firmware image needs to be downloaded.
    - Encrypted image support is not available.

For overview and usage of other update modes, refer to https://www.renesas.com/en/document/apn/ra8-basic-secure-bootloader-using-mcuboot-and-internal-code-flash?r=25448206 and the MCUboot design page: https://github.com/mcu-tools/mcuboot/blob/master/docs/design.md.

Note: When using the Direct XIP upgrade mode, the update image needs to have a version number higher than the current primary image.

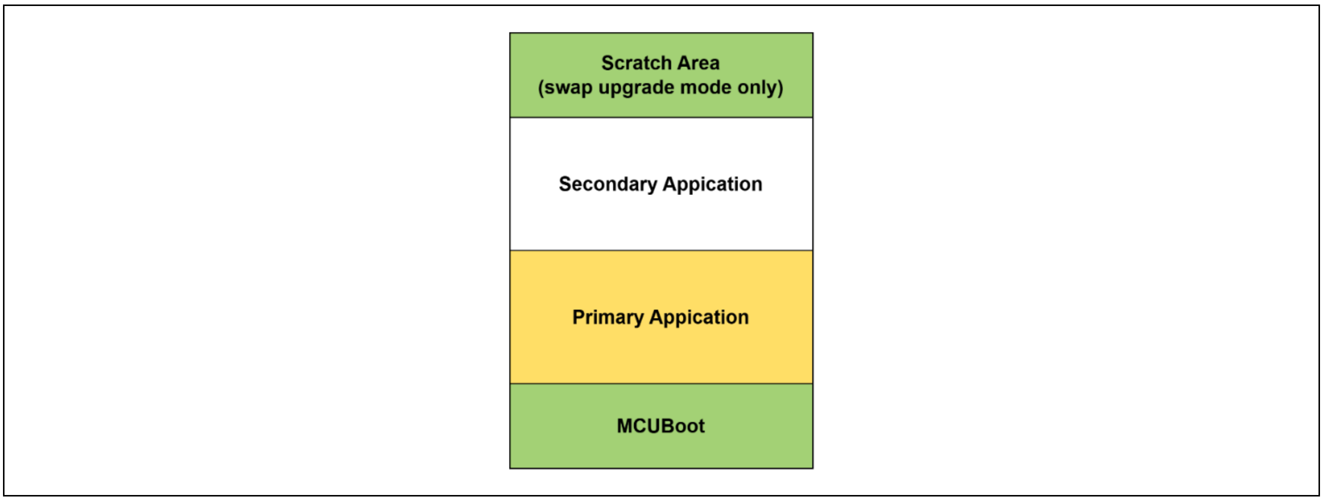

Figure 6. MCUboot Memory Flash Map: DXIP Upgrade Mode

2.3 Memory Mirror Function

The Memory Mirror Function (MMF) maps the load address of an application image in the code flash memory to its link address in the unused 23-bit memory mirror space. The user application code is developed and linked to run from this MMF destination address. The user application code is not required to know the load location where it is stored in the code flash memory.

For more details, users can refer to **Memory Mirror Function (MMF)** section in RA2A2 Hardware User's Manual.

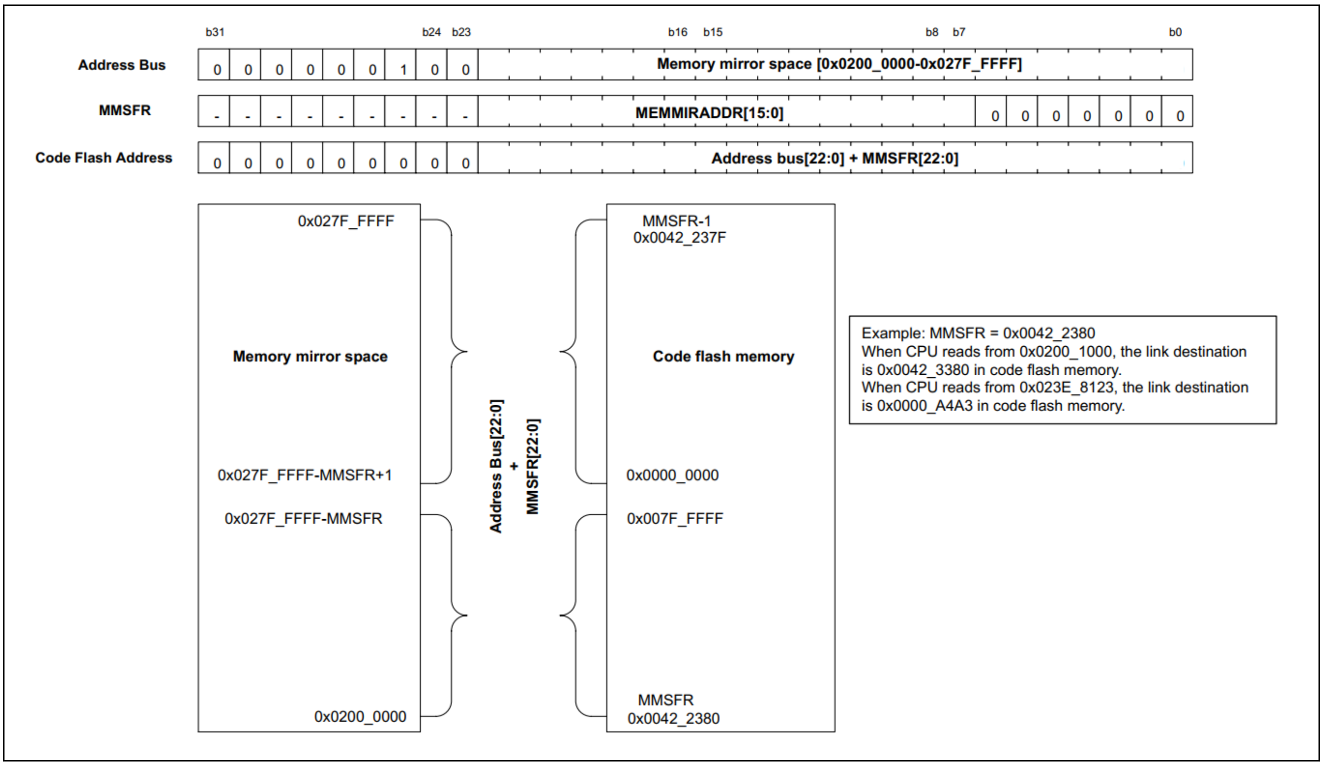

Figure 7. MMF Overview

2.4 Using Direct XIP Upgrade Mode with MMF

When the MCUboot has MMF enabled, MCUboot decides which image will be “mirrored”. After the image is reflected on the MMF region, the MCUboot will start booting the application from start address of MMF (0x0200_0000).

In other words, MCUboot always jumps to start address of MMF to execute the application.

In addition, the benefits of using Direct XIP Upgrade Mode with MMF include:

- A simplified transition from bootloader execution to application execution using a fixed application image vector table.
- Faster image updates due to Direct XIP mode.

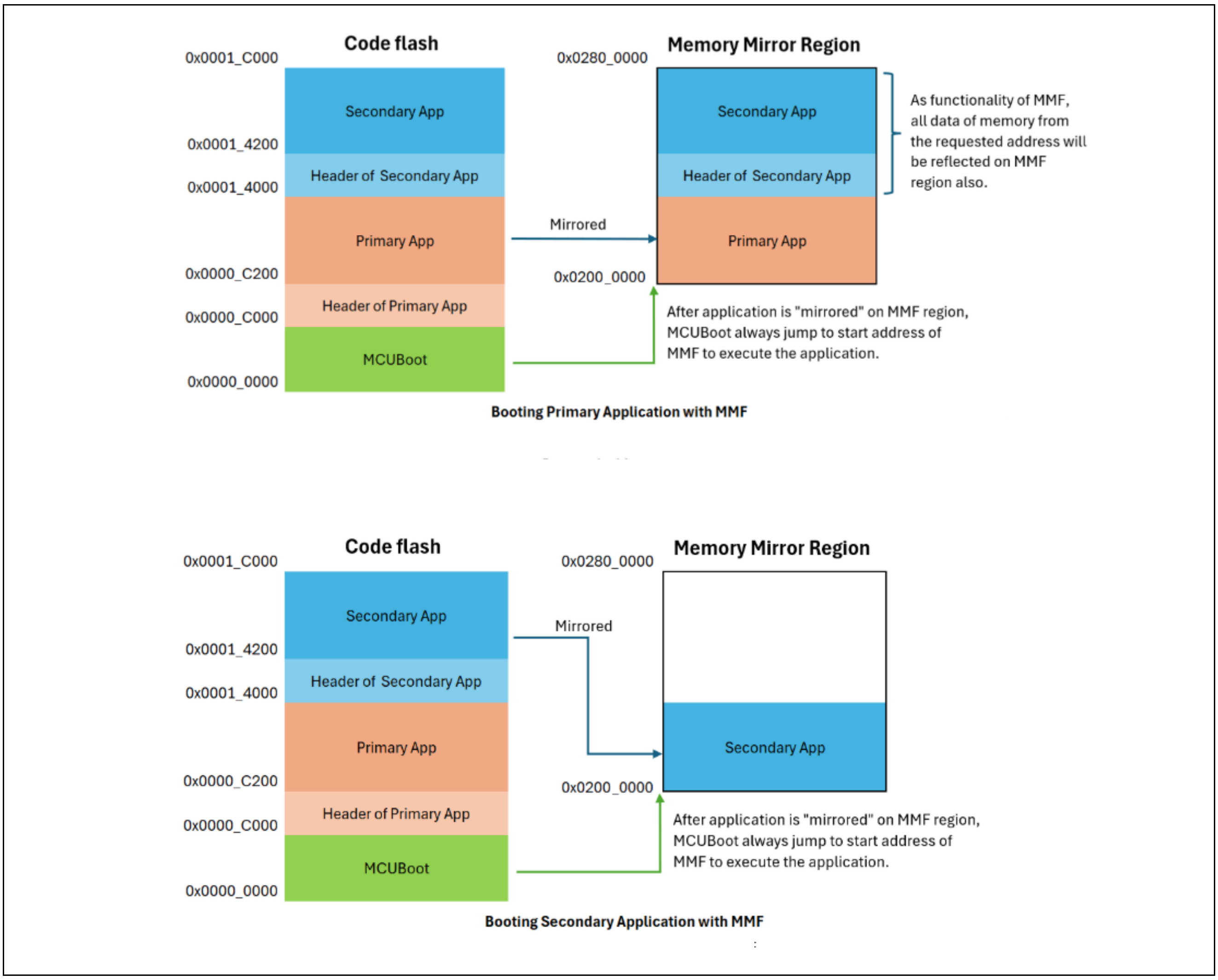

Figure 8. Booting Application with MMF

2.5 Designing Bootloader and Initial Primary Application Overview

A bootloader is typically designed with the initial primary application. The following general guidelines apply to designing the bootloader and the initial primary application:

- Develop the bootloader and analyze the MCU memory resource allocation needed for the bootloader and the application. The bootloader memory usage is influenced by the application image update mode, signature type, and whether to validate the Primary Image as well as the cryptographic library used.
- Develop the initial primary application, perform the memory usage analysis, and compare with the bootloader memory allocation for consistency and adjust as needed.
- Determine the bootloader configurations in terms of image authentication and new image update mode. This may result in adjustment of the memory allocated definition in the bootloader project.
- Sign the application image. The signing command is output to the `<bootloader project>\Debug\>bootloader project>.sbd` file. The application image can use a Build Variable to access this `.sbd` file. The IDE tools use the signing command to sign the application and generate a binary file for downloading to the MCU.
- Test the bootloader and the initial primary application.

The above guidelines are demonstrated in the walk-through sections in this application note.

3. System Overview

This section provides information on the major events involved in embedded system design using MCUboot as the secure bootloader. An XModem-based image downloader is included in this application project, and its main design flow is described here. In addition, the included example projects are outlined, along with guidelines for quick evaluation.

3.1 System-Level Major Events

Figure 9. High-Level Design for the Application Project High-Level Design for the Application Project describes a high-level overview of the key events within the system.

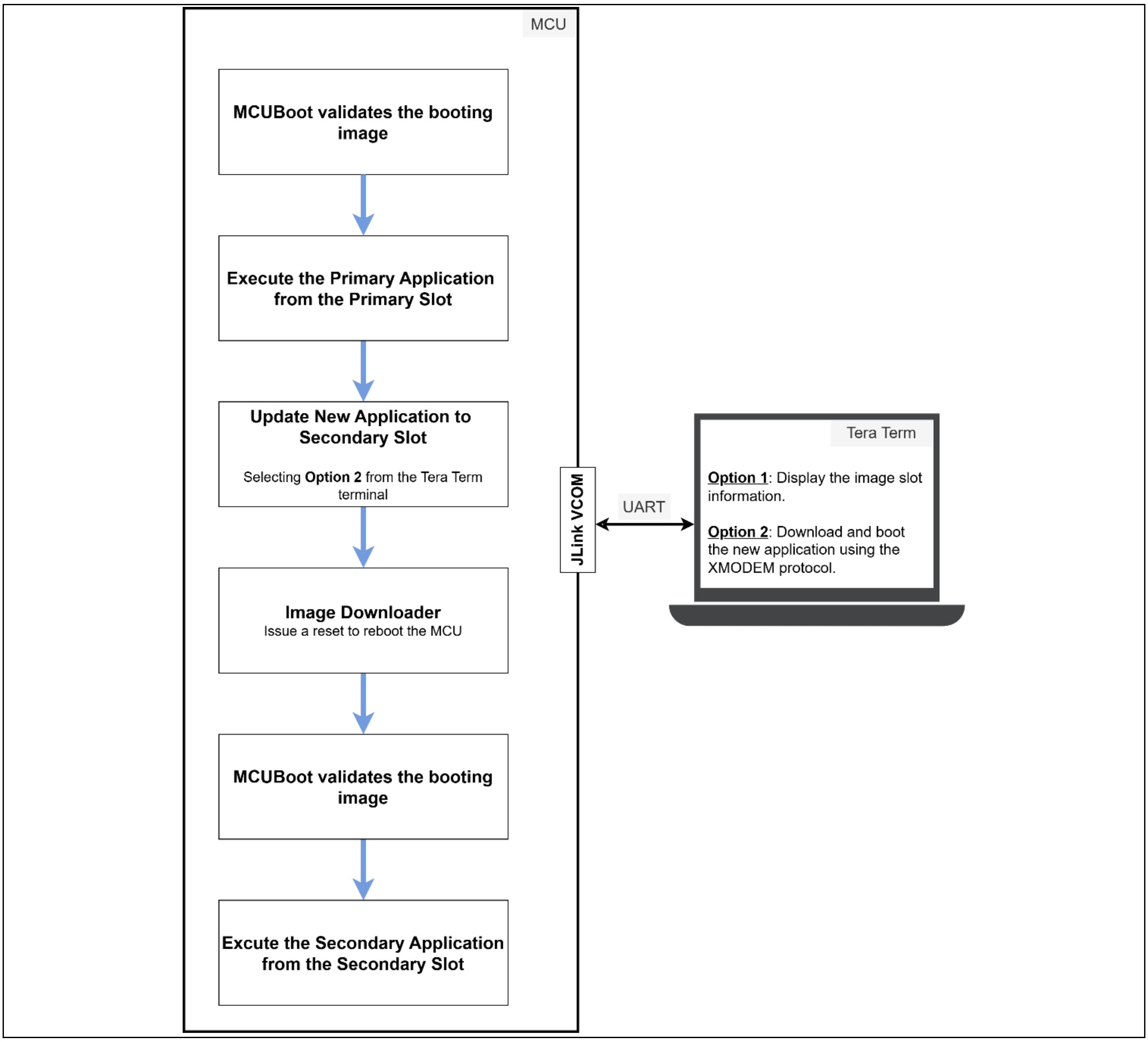

Figure 9. High-Level Design for the Application Project

3.2 XModem Based Image Downloader

Furthermore, to support updating a new application from the PC to the MCU, this application note demonstrates the use of the XMODEM protocol over UART.

The XMODEM protocol is a simple file transfer protocol designed for reliable serial communication (UART). It divides the application image into fixed-size blocks (128 bytes), each accompanied by a checksum to ensure data integrity during transmission. Each transfer block is as follows:

Start of Header	Packet Number	1's complement of the Packet Number	The packet	Checksum

Description	Value	Length	Function
Start of Header (SOH)	0x01	1 byte	Signifies the start of the block.
Packet Number		1 byte	Represents the block number, starting from 1 and incrementing by 1 for each subsequent block.

1's complement of the Packet Number	255 – "Packet Number"	1 byte	Ensuring the block is received correctly.
The packet		128 bytes	The actual data being transmitted.
Checksum		1 byte	A checksum of the 128 packet bytes is used for error detection.

During the transfer, each block must be acknowledged (ACK) by the receiver before the next block is sent. In the case of a failed transmission, the receiver responds with a negative acknowledgement (NAK), and the block is retransmitted, as shown in Figure 9 XMODEM Transfer Example.

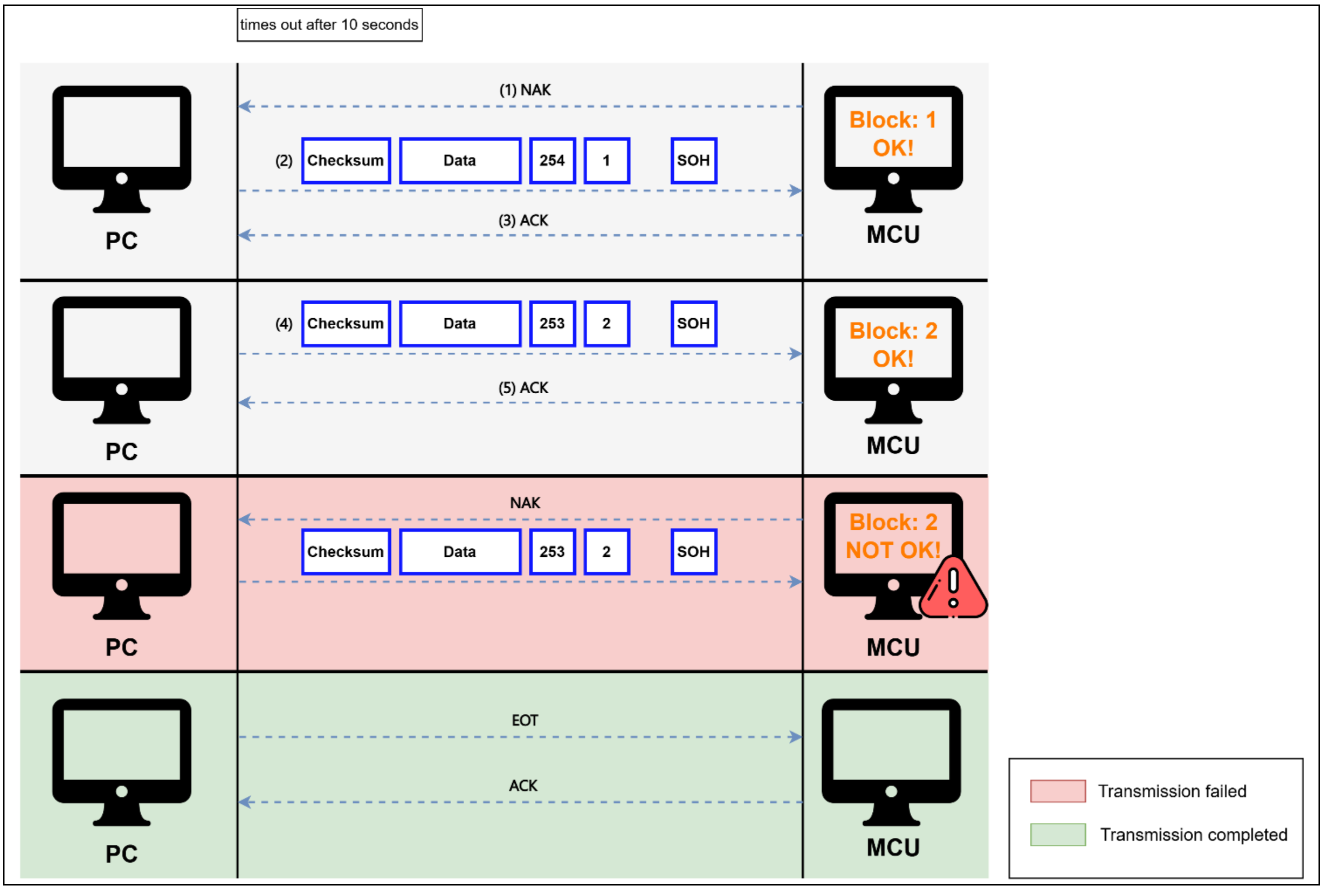

Figure 9. XMODEM Transfer Example

Note that the values of ACK, NAK, SOH, and EOT are defined in the `xmodem.h` file.

To update the new application to the secondary or primary slot, users need to refer to Figure 27 MCUboot Memory Map with DXIP Update Mode to define the start and end addresses of the image in the `header.h` file, as shown in Figure 10 Image Address Configuration in header.h file.

```
#include "downloader_thread.h"

#define PRIMARY_IMAGE_START_ADDRESS      0x00005000
#define PRIMARY_IMAGE_END_ADDRESS        0x000427FF

#define SECONDARY_IMAGE_START_ADDRESS    0x00042800
#define SECONDARY_IMAGE_END_ADDRESS      0x0007FFFF

#define FLASH_BLOCK_SIZE                 (2 * 1024)

#define PRIMARY_IMAGE_NUM_BLOCKS         ((PRIMARY_IMAGE_END_ADDRESS - PRIMARY_IMAGE_START_ADDRESS + 1U) / FLASH_BLOCK_SIZE)
#define SECONDARY_IMAGE_NUM_BLOCKS       ((SECONDARY_IMAGE_END_ADDRESS - SECONDARY_IMAGE_START_ADDRESS + 1U) / FLASH_BLOCK_SIZE)
```

Figure 10. Image Address Configuration in header.h file

3.3 Linker Script Update When MMF is Enabled

Enabling the Memory Mirror Function (MMF) updates the `\Debug\memory_regions.ld` linker script of the bootloader project. The comparison of the `memory_regions.ld` between configurations with and without MMF using the bootloader included in this application project is presented in Figure 11 BOOT_IMAGE_FROM_MMF_REGION with and without MMF (memory_regions.ld).

Without MMF	With MMF
FLASH_BOOTLOADER_LENGTH = 0x5000; FLASH_BOOTLOADER_HEADER_LENGTH = 0x200; FLASH_BOOTLOADER_HEADER_LENGTH_2 = 0x200; FLASH_BOOTLOADER_SCRATCH_LENGTH = 0x0; FLASH_APPLICATION_S_LENGTH = 0x3D800; FLASH_APPLICATION_NSC_LENGTH = 0x0; FLASH_APPLICATION_NS_LENGTH = 0x0; RAM_APPLICATION_NSC_LENGTH = 0x0; RAM_APPLICATION_NS_LENGTH = 0x0; FLASH_APPLICATION_IMAGE_NUMBER = 0x1; BOOT_IMAGE_FROM_MMF_REGION = 0x0; MMF_REGION_START_ADDR = 0x2000000;	FLASH_BOOTLOADER_LENGTH = 0x5000; FLASH_BOOTLOADER_HEADER_LENGTH = 0x200; FLASH_BOOTLOADER_HEADER_LENGTH_2 = 0x200; FLASH_BOOTLOADER_SCRATCH_LENGTH = 0x0; FLASH_APPLICATION_S_LENGTH = 0x3D800; FLASH_APPLICATION_NSC_LENGTH = 0x0; FLASH_APPLICATION_NS_LENGTH = 0x0; RAM_APPLICATION_NSC_LENGTH = 0x0; RAM_APPLICATION_NS_LENGTH = 0x0; FLASH_APPLICATION_IMAGE_NUMBER = 0x1; BOOT_IMAGE_FROM_MMF_REGION = 0x1; MMF_REGION_START_ADDR = 0x2000000;
Note: The value of **BOOT_IMAGE_FROM_MMF_REGION** is defined in the `memory_regions.ld` file of the bootloader project.	

Figure 11. BOOT_IMAGE_FROM_MMF_REGION with and without MMF (memory_regions.ld)

When the MMF is enabled, as shown in Figure 29 Enable the MMF, the value of **BOOT_IMAGE_FROM_MMF_REGION** is set to 1. This value will be assigned to __bl_FLASH_IMAGE_START_FROM_MMF_REGION in the linker script `\script\fsp.ld` of the bootloader and the application project.

```
__bl_XIP_SECONDARY_FLASH_IMAGE_END = __bl_XIP_SECONDARY_FLASH_IMAGE_START + __bl_FLASH_IMAGE_LENGTH;
__bl_FLASH_IMAGE_START_FROM_MMF_REGION = DEFINED(BOOT_IMAGE_FROM_MMF_REGION) ? BOOT_IMAGE_FROM_MMF_REGION : 0;
__bl_MEMORY_MIRROR_REGION_START = DEFINED(MMF_REGION_START_ADDR) ? MMF_REGION_START_ADDR : 0;
__bl_FLASH_NS_START = !DEFINED(FLASH_BOOTLOADER_LENGTH) ? 0 :
                      FLASH_APPLICATION_NS_LENGTH == 0 ? __bl_FLASH_IMAGE_END :
                      __bl_FLASH_IMAGE_START - FLASH_BOOTLOADER_HEADER_LENGTH + FLASH_APPLICATION_S_LENGTH;
```

Figure 12. __bl_FLASH_IMAGE_START_FROM_MMF_REGION in the linker script

Based on the value of __bl_FLASH_IMAGE_START_FROM_MMF_REGION, the value of **FLASH_IMAGE_START_FROM_MMF_REGION** is defined in the application project linker script `memory_regions.ld`, as shown in Figure 13 **FLASH_IMAGE_START_FROM_MMF_REGION** with and without MMF.

Without MMF	With MMF
FLASH_NS_START = 0x42800; FLASH_IMAGE_END = 0x42800; FLASH_NS_IMAGE_START = 0x42800; FLASH_NSC_START = 0x42800; FLASH_IMAGE_START_FROM_MMF_REGION = 0x0; MEMORY_MIRROR_REGION_START = 0x2000000; RAM_NSC_START = 0x2000C000; FLASH_IMAGE_LENGTH = 0x3D600; XIP_SECONDARY_FLASH_IMAGE_END = 0x80000; XIP_SECONDARY_FLASH_IMAGE_START = 0x42A00; FLASH_IMAGE_START = 0x5200;	FLASH_NS_START = 0x42800; FLASH_IMAGE_END = 0x42800; FLASH_NS_IMAGE_START = 0x42800; FLASH_NSC_START = 0x42800; FLASH_IMAGE_START_FROM_MMF_REGION = 0x1; MEMORY_MIRROR_REGION_START = 0x2000000; RAM_NSC_START = 0x2000C000; FLASH_IMAGE_LENGTH = 0x3D600; XIP_SECONDARY_FLASH_IMAGE_END = 0x80000; XIP_SECONDARY_FLASH_IMAGE_START = 0x42A00; FLASH_IMAGE_START = 0x5200;
Note: The value of **FLASH_IMAGE_START_FROM_MMF_REGION** is defined in the `memory_regions.ld` file of the primary or secondary project.	

Figure 13. FLASH_IMAGE_START_FROM_MMF_REGION with and without MMF

The symbol **FLASH_IMAGE_START_FROM_MMF_REGION** affects the linker script file (`fsp.ld`) in the primary or secondary project. It determines whether the application image will be executed from **MEMORY_MIRROR_REGION_START** or **FLASH_IMAGE_START**, as shown in Figure 14 FLASH_IMAGE_START_FROM_MMF_REGION in the linker script.

```
FLASH_ORIGIN = !DEFINED(FLASH_IMAGE_START) ? FLASH_START :
               XIP_SECONDARY_SLOT_IMAGE == 1 ? XIP_SECONDARY_FLASH_IMAGE_START :
               FLASH_IMAGE_START_FROM_MMF_REGION == 1 ? MEMORY_MIRROR_REGION_START : FLASH_IMAGE_START;

LIMITED_FLASH_LENGTH = DEFINED(FLASH_IMAGE_LENGTH) ? FLASH_IMAGE_LENGTH :
                       DEFINED(FLASH_BOOTLOADER_LENGTH) ? FLASH_BOOTLOADER_LENGTH :
                       FLASH_LENGTH;
```

Figure 14. FLASH_IMAGE_START_FROM_MMF_REGION in the linker script

3.4 Introduction to the Included Example Projects

Unzip `ra2-secure-bootloader-using-mcuboot-internal-code-flash-mmf.zip` to unpack the example projects included in this application project.

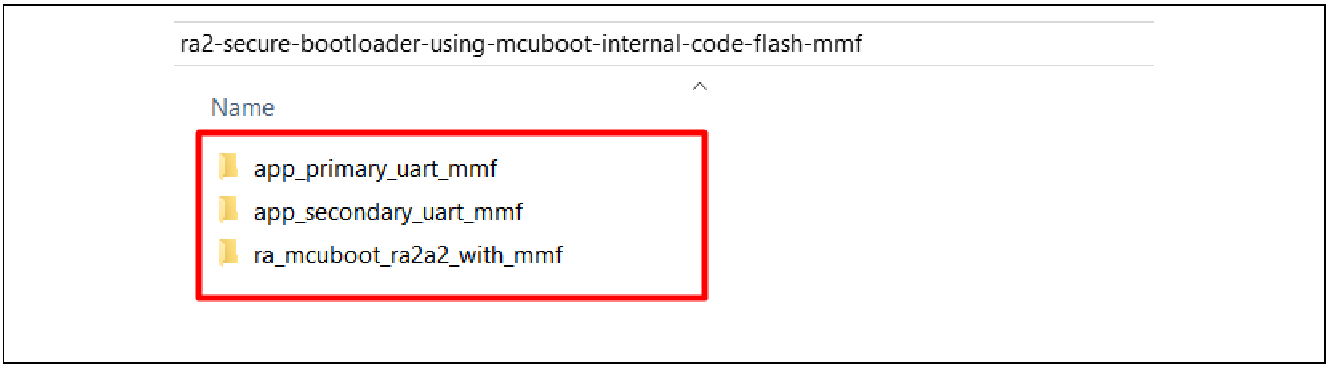

Figure 15. Example Projects Included

- `ra_mcuboot_ra2a2_with_mmf`: Bootloader, which enables MMF and Direct XIP upgrade mode.
- `app_primary_uart_mmf`: Primary application, which is configured to work with the bootloader and implements XModem over UART to download a new application image. FreeRTOS is used with two threads, one thread blinks the three LEDs on EK-RA2A2 while the other thread downloads the new application image concurrently.
- `app_secondary_uart_mmf`: Secondary application, which implements the same functionality as `app_primary_uart_mmf` except only the blue LED is blinking.

Refer to the section 9 for instructions on quickly evaluating the projects.

4. Creating the Bootloader Project using Code Flash Linear Mode and MMF

This section demonstrates the creation process of the bootloader project utilizing MCUboot, the Flash Linear Mode, and MMF.

4.1 Include the MCUboot Module in the Bootloader Project

Follow below steps to start the bootloader project creation and include the MCUboot module in the project:

1. Launch e^2 studio and start a new C/C++ Project. Click **File** > **New** > **C/C++ Project.**

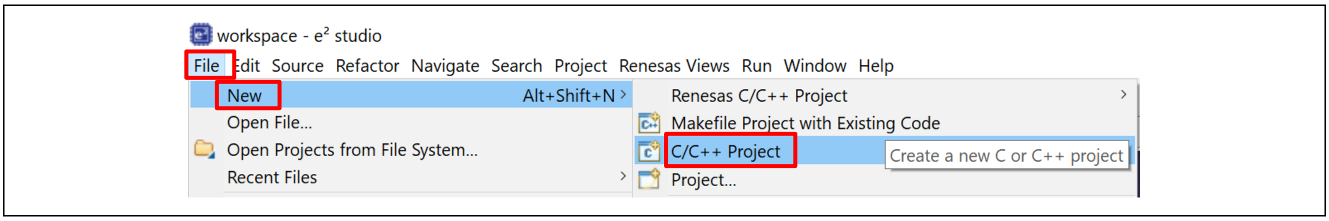

Figure 16. Start a New Project

2. Choose **Renesas RA > Renesas RA C/C++ Project**. Click **Next**.

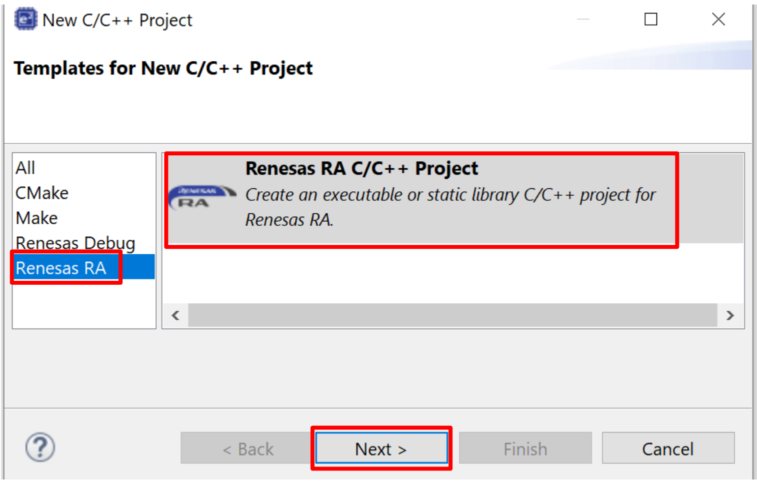

Figure 17. Choose Renesas RA C/C++ Project

3. Provide the project name `ra_mcuboot_ra2a2_with_mmf` on the next screen. Click **Next**.
4. In the next screen, choose **EK-RA2A2** for **Board** and click **Next**.

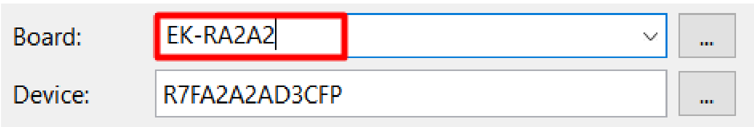

Figure 18. Select the Board

5. When the following screen appears, select **Flat (Non-TrustZone) Project**.

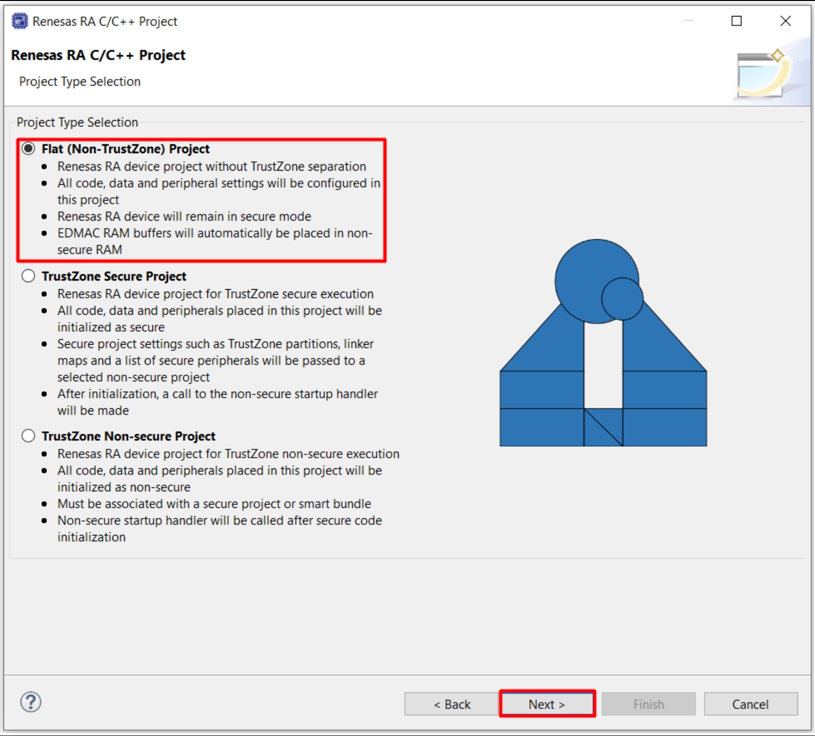

Figure 19. Choose Flat Project as Project Type

6. Choose **Executable** for **Build Artifact Selection** and **No RTOS**. Click **Next**.

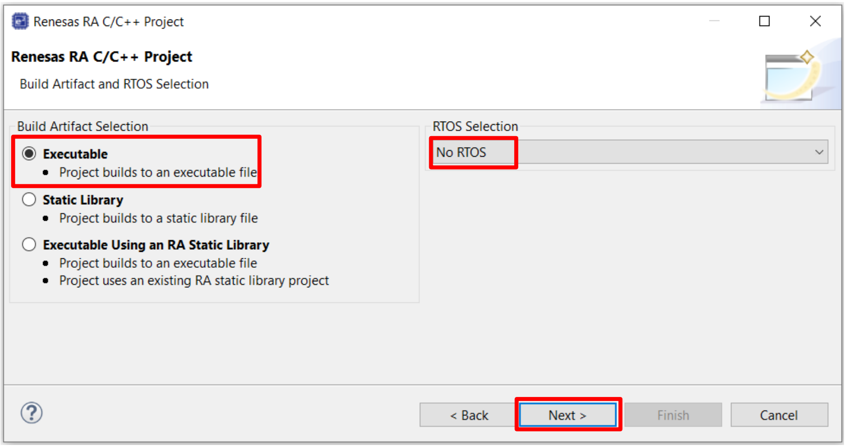

Figure 20. Choose to Build Executable and No RTOS

7. Choose **Bare Metal – Minimal** for the Project Template in the next screen and click **Finish** to establish the initial project.

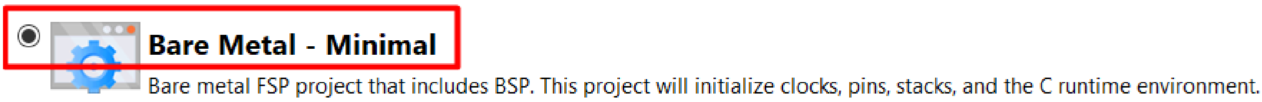

Figure 21. Choose the Project Template

8. When the following prompt opens, click **Open Perspective**.

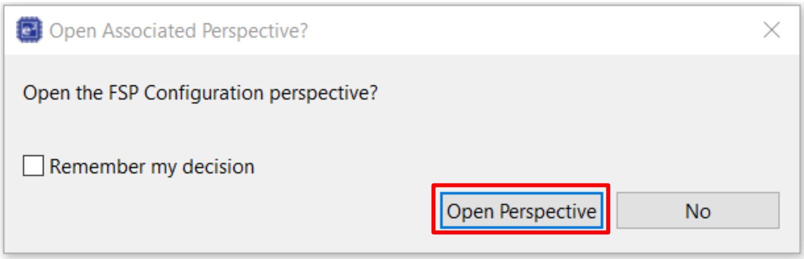

Figure 22. Choose Open the FSP Configuration Perspective

The project is then created, and the bootloader project configuration is displayed.

9. Select the **Pins** tab and uncheck **Generate data** for **RA2A2 EK.**

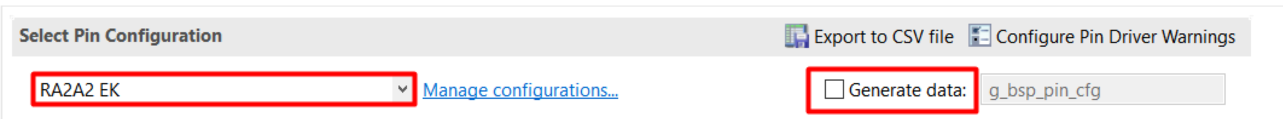

Figure 23. Uncheck Generate data for RA2A2 EK Pin Configuration

Use the pull-down menu to switch from **RA2A2 EK** to **R7FA2A2AD3CFP.pincfg** for the **Select Pin Configuration** option, then select the **Generate data** check box and enter **g_bsp_pin_cfg**. Note that here we choose to use this configuration which has fewer peripherals/pins configured since the bootloader does not use the extra peripheral or GPIO pins configured in the **RA2A2 EK** configuration. This also reduces some memory usage for the bootloader project.

Figure 24. Select g_bsp_pin_cfg and Generate data g_bsp_pin_cfg

10. Once the project is created, click the **Stacks** tab on the RA configurator. Add **New Stack > Bootloader > MCUboot**.

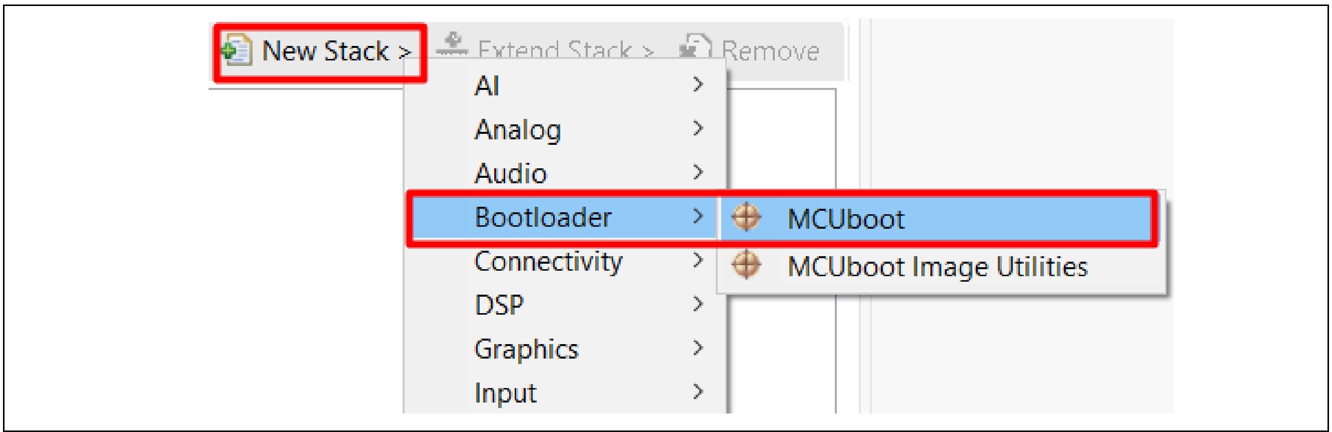

Figure 25. Add the MCUboot Port

11. Next, configure the **General** properties of **MCUboot**. We will resolve the errors in the configurator in the following steps. Currently, the FSP only supports DXIP mode with MMF feature. Therefore, users need to configure the **Update Mode** to **Direct XIP**.

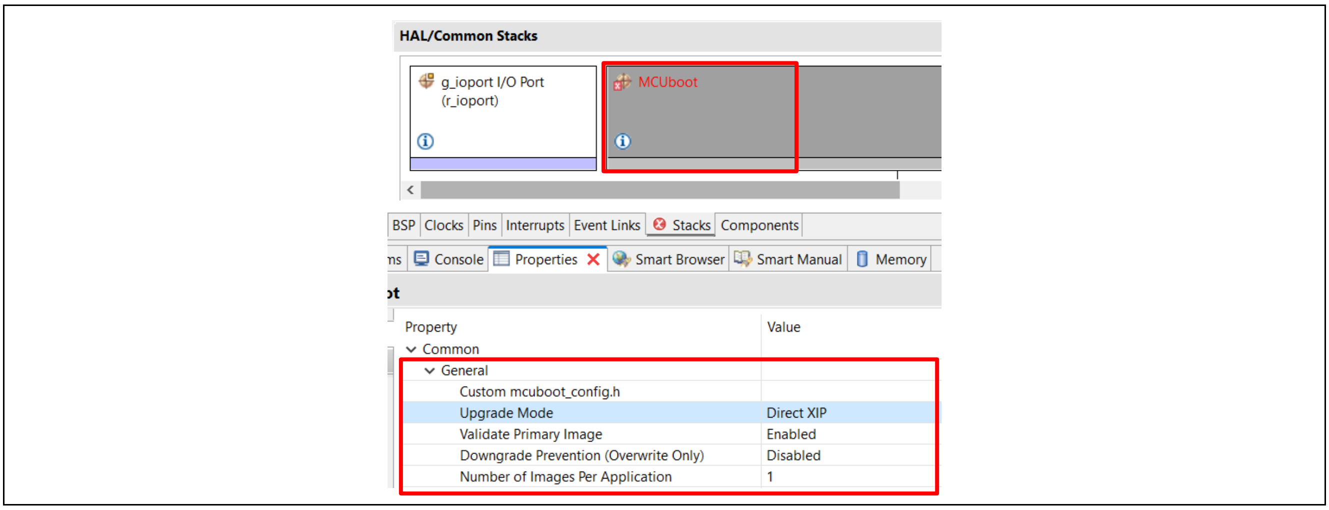

Figure 26. General Configuration for MCUboot Module

The properties configured are:

- **Custom mcuboot_config.h**: The default `mcuboot_config.h` file contains the MCUboot Module configuration that the user selected from the RA configurator. The user can create a custom version of this file to achieve additional bootloader functionalities available in MCUboot.
- **Upgrade Mode**: This property configures the application image upgrade method. The available options are Overwrite Only, Overwrite Only Fast, Swap and Direct XIP. Only Direct XIP is supported for MMF.
- **Validate Primary Image**: When enabled, the bootloader will perform a hash or signature verification, depending on the verification method chosen, in addition to the MCUboot magic number-based sanity check. When disabled, only a sanity check is performed based on the MCUboot magic number.
- **Number of Images Per Application:** This property allows users to choose one image for non-TrustZone-based applications and two images for TrustZone-based applications. Set this property to 1.
- **Downgrade Prevention (Overwrite Only):** This property applies to Overwrite upgrade mode only. When this property is **Enabled**, a new firmware with a lower version number will not overwrite the existing application.

4.2 Configure the Memory Configuration and Authentication Method

Configure the Signing Options and Flash Layout of the MCUboot module. Based on the internal code flash memory described in section 1.1, the MCUBoot memory map is calculated, as shown in Figure 27.

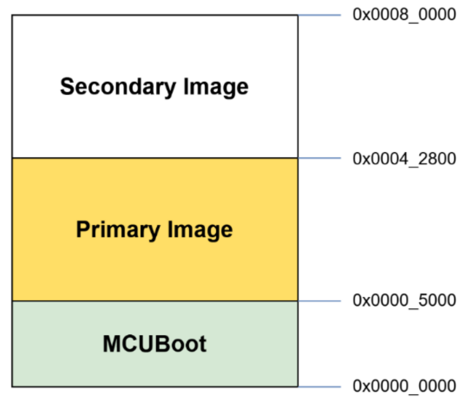

Figure 27. MCUboot Memory Map with DXIP Update Mode

Follow Figure 27 to update the **Properties** for the Flash Layout to match with the MCUboot memory map, as shown in Figure 28.

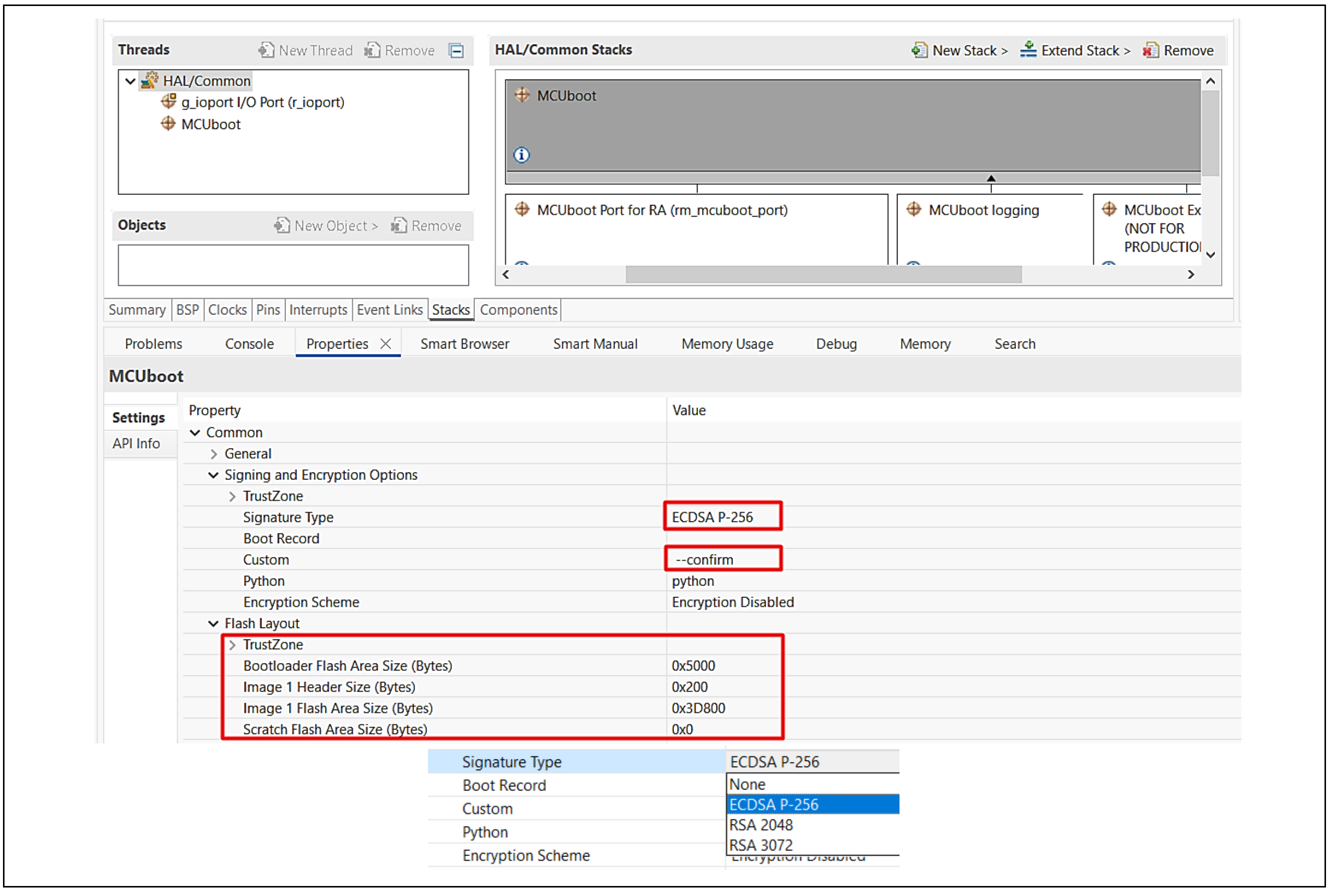

Figure 28. Configure the Flash Layout and Signing Options

Explanation of the Above Configurations:

- **Bootloader Flash Area**: Size of the flash area allocated for the bootloader, with a boundary of 0x800 since 0x800 is the minimum erase size for RA2A2 code flash.
- **Image 1 Header Size**: Size of the code flash reserved for the application image header. It must meet minimum VTOR alignment requirements based on the number of interrupts implemented on the RA2A2.
- **Image 1 Flash Area Size**: Size of application image 1, including the header and trailer. For the RA2A2, this size needs to be on a boundary of 0x800 which is the smallest flash erase size.
- **Scratch Flash Area Size**: This property is only needed for Swap mode. The Scratch Area must be large enough to store the largest sector that is going to be swapped. For all RA2 MCUs, the Scratch The area should be set up to 0x800 when Swap mode is used.
- **Signature Type:** Signing algorithm selection. The choices are:
  - **NONE:** Select this option for bootloaders that do not support signature verification.
  - **ECDSA P-256**: Select this option for this example bootloader design.
  - **RSA 2048 and RSA 3072:** Not supported.
- **Custom:** Use the default `--confirm` for this bootloader design. Switching to a new image is always confirmed, and the new image will be booted after a subsequent system reset. Reverting the image with Direct XIP is not supported with the current FSP version.
- **Encryption Scheme:** Encryption is disabled in this example implementation.

4.3 Enable the Memory Mirror Function Support

Click on the **MCUboot stack** > **Properties** > **Flash Configuration**. Then, enable the **Memory Mirror Function**, as shown in Figure 29 Enable the MMF.

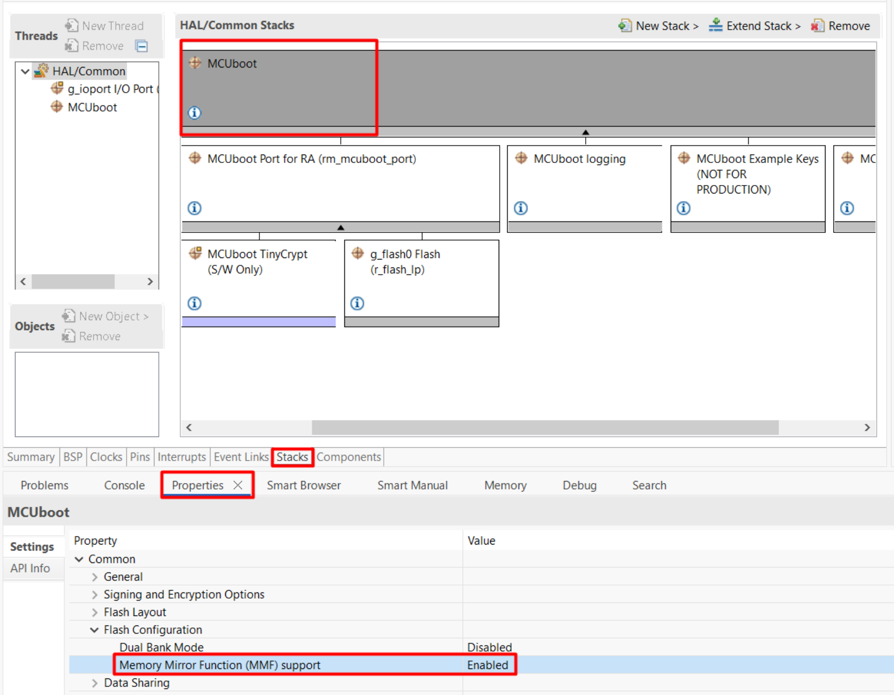

Figure 29. Enable the MMF

4.4 Configure the TinyCrypt Module and the Flash Driver

Follow steps below to configure the TinyCrypt module and the flash driver:

1. Click on **Add Crypto Stack** and select the **MCUboot TinyCrypt (S/W Only)** module.

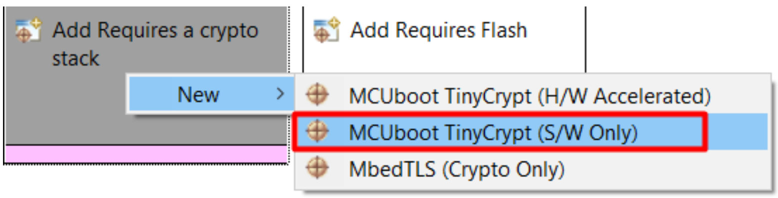

Figure 30. Select TinyCrypt Module

2. If the user is creating a bootloader with signature verification support, then the **ASN.1 Parser** stack and the **MCUboot Example Keys** stack will be required.

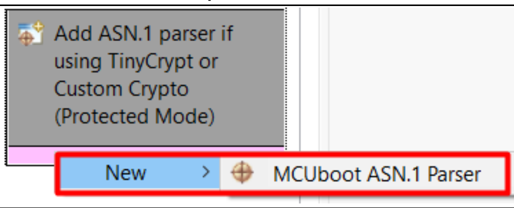

Figure 31. Add the ASN.1 Parser

Click on the **Add [Optional] Add Example Keys** stack and choose **New > MCUboot Example Keys [NOT FOR PRODUCTION]**.

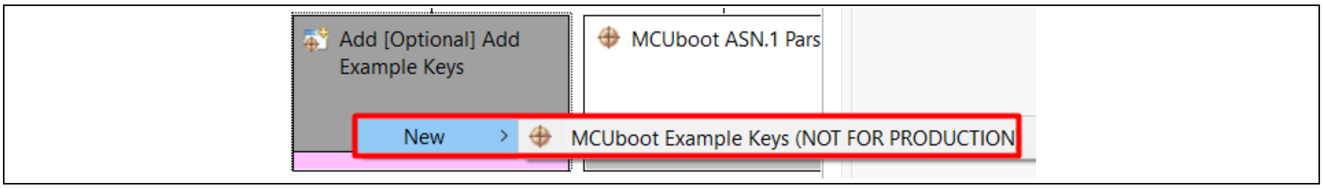

Figure 32. Add the Example Image Signing Key

Note: The example key is open to public access from MCUboot port, customers should not use them for production purposes. Customers can follow the procedure in section 3.6.1 in Application Project R11AN0516 to create and use customized signing key.

3. Click on **Add Requires Flash** stack and select Flash (r_flash_lp) stack.

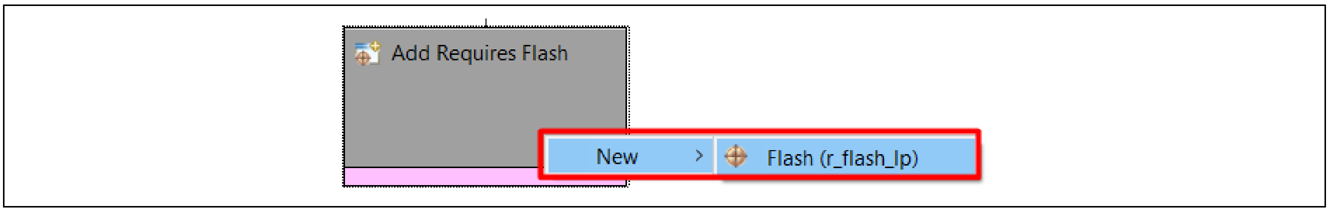

Figure 33. Add the Flash Driver

4. Next, set the **Code Flash Programming** to **Enabled**. As **Data Flash Programming** is not used in the bootloader, select **Disabled** for the **Data Flash Programming** to reduce the bootloader memory footprint.

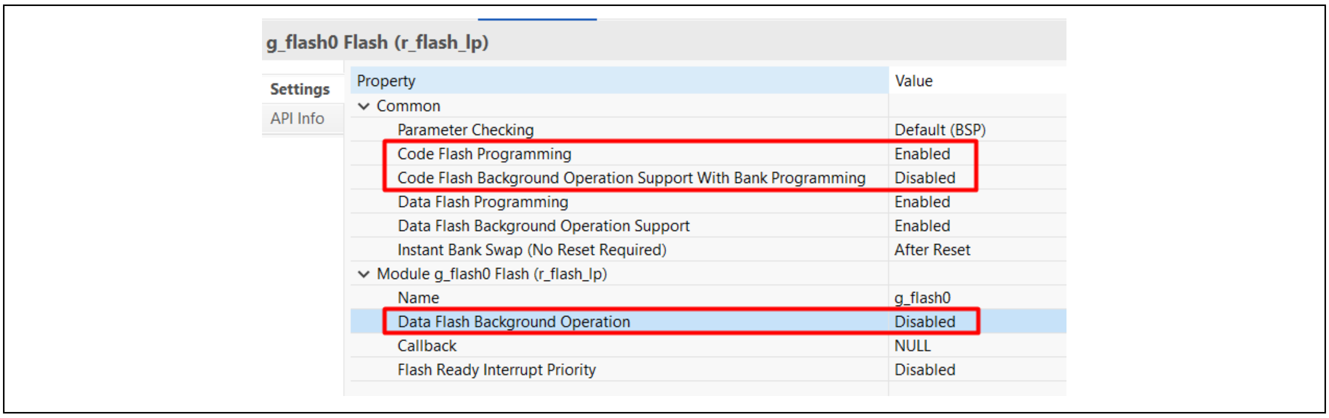

Figure 34. Configure the Flash Driver

5. Update the **BSP > Main Stack Size** to 0x1000

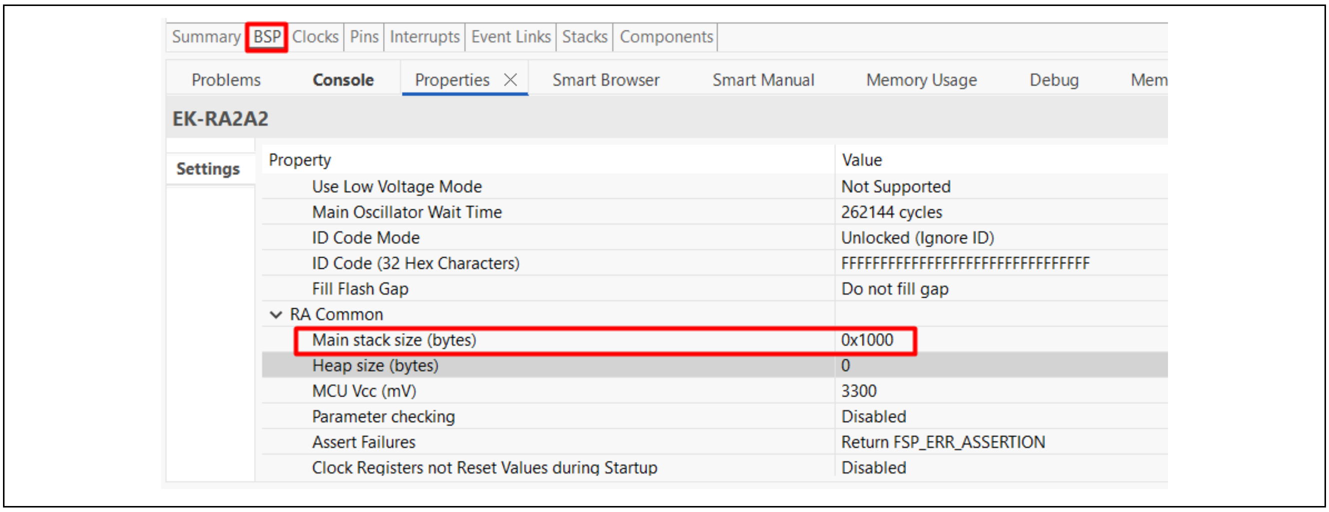

Figure 35. Update the Main Stack Size

4.5 Add the Boot Code

Save `configuration.xml` and click **Generate Project Content**. Then, expand the `Developer Assistance>HAL/Common>MCUboot>Quick Setup` and drag `Call Quick Setup` to the top of the `hal_entry.c` of the bootloader project.

Add the following function call to the top of the `hal_entry()` function:

```
mcuboot_quick_setup();
```

4.6 Configure the Python Signing Environment

Signing the application image can be done using a post-build step in e² studio, using the image signing tool `imgtool.py`, which is included with MCUboot. This tool is integrated as a post-build tool in e² studio to sign the application image. If this is **NOT** the first time you have used the Python script signing tool on your computer, you can skip to section 5.

If this is the first time you are using the Python script signing tool on your system, you will need to install the dependencies required for the script to work. Navigate to the **ra_mcuboot_ra2a2_with_mmf > ra > mcu-tools > MCUboot** folder in the **Project Explorer**, right click and select **Command Prompt**. This will open a command window with the path set to the `\mcu-tools\MCUboot` folder.

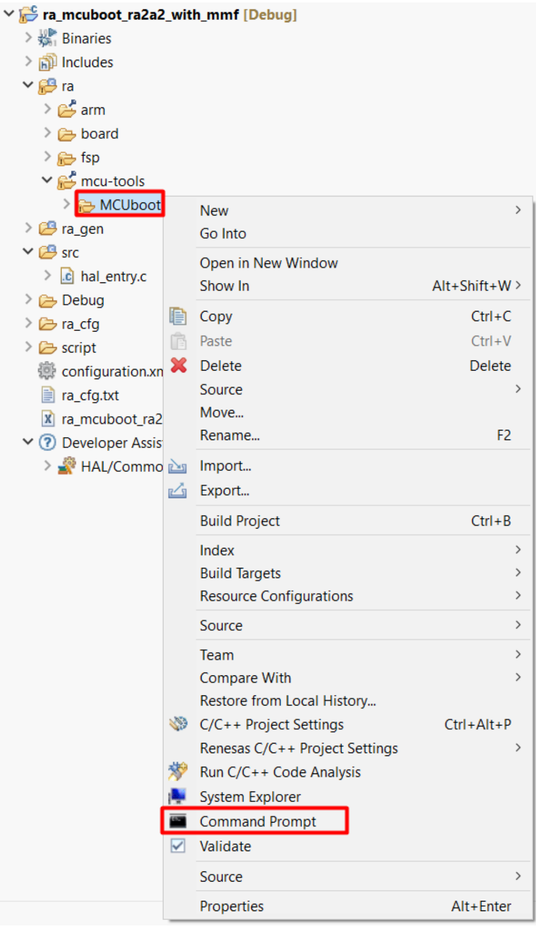

Figure 36. Open the Command Prompt

We recommend upgrading pip prior to installing the dependencies. Enter the following command to update pip:

```
python -m pip install --upgrade pip
```

Next, in the command window, enter the following command line to install all the MCUboot dependencies:

```
pip3 install --user -r scripts/requirements.txt
```

This will verify and install any dependencies that are required.

4.7 Compile the Bootloader Project

In the RA configurator, click **Generate Project Content**, then compile the project.

```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2a2_with_mmf.elf"  "ra_mcuboot_ra2a2_with_mmf.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2a2_with_mmf.elf"
   text    data     bss     dec     hex filename
  17924       0    4668   22592    5840 ra_mcuboot_ra2a2_with_mmf.elf

14:31:12 Build Finished. 0 errors, 1 warnings. (took 7s.769ms)
```

Figure 37. Compile the bootloader ra_mcuboot_ra2a2_with_mmf

There are warnings from third-party code.

4.8 Optimizing the Bootloader Project Size

To further optimize the bootloader project for size, users can follow several optimization methods, such as:

Bootloder Size Initial (bytes): 17924

No.	Optimization methods	Actions	Bootloader Size (bytes)
1	Put some functions into gap area (.flash_gap)	**void R_BSP_WarmStart**(bsp_warm_start_event_t event) BSP_PLACE_IN_SECTION(".flash_gap*"); **void mcuboot_quick_setup**() BSP_PLACE_IN_SECTION(".flash_gap*"); fih_ret **context_boot_go**(**struct** boot_loader_state *state, **struct** boot_rsp *rsp) BSP_PLACE_IN_SECTION(".flash_gap*");	17572
2	Change the compiling optimization to **Optimize Size (-Os)**	**Optimize Size (-Os)**	13532
3	Combining methods 1 and 2		13180

Figure 38. Several methods to optimize the bootloader size

For more details, users can refer to section 3.2 in Application Project R11AN0516.

5. Configuring and Signing an Application Project

Developing an initial application to use a bootloader starts with developing and testing the application and the bootloader independently. Using the bootloader with an existing application or developing a new application to use the bootloader involves the following common steps:

- Adjust the memory map of the bootloader to allow the application and bootloader to fit the available MCU memory area.
- Configure the application to use the bootloader.
- Sign the application image.
- Developing an application to use a bootloader typically requires the application to have the capability to download a new application. This application project demonstrates how to download a new application using the UART interfaces as examples. Users typically have custom methods to download new application images.

5.1 Configure the Application Project to Use the Bootloader

Users can follow *FSP User's Manual* section Tutorial: Your First RA MCU Project – Blinky to establish a new project. This application note uses the included example project as the initial application project and guides the user through the procedures to configure the example project to use the bootloader established in section 4.

Note that the steps described in this section can be applied to other existing application projects to configure the application project to use the bootloader. Be sure to consider the size of the application project. When using the bootloader with a different application project, the **Image 1 Flash Area Size** property should be adjusted accordingly.

Import the desired application projects to the workspace where the bootloader is created.

Right-click on the application project folder `app_primary_uart_mmf` in the **Project Explorer** and select **Properties**. Select **C/C++ Build > Build Variables**, click **Add** and set the **Variable name** to **BootloaderDataFile,** and check the **Apply to all configurations** box. Change the **Type** to **File** and enter the path to the `.sbd` file for the bootloader project **`ra_mcuboot_ra2a2_with_mmf`**:

- Set `${workspace_loc:ra_mcuboot_ra2a2_with_mmf}/Debug/ra_mcuboot_ra2a2_with_mmf.sbd` for the value.

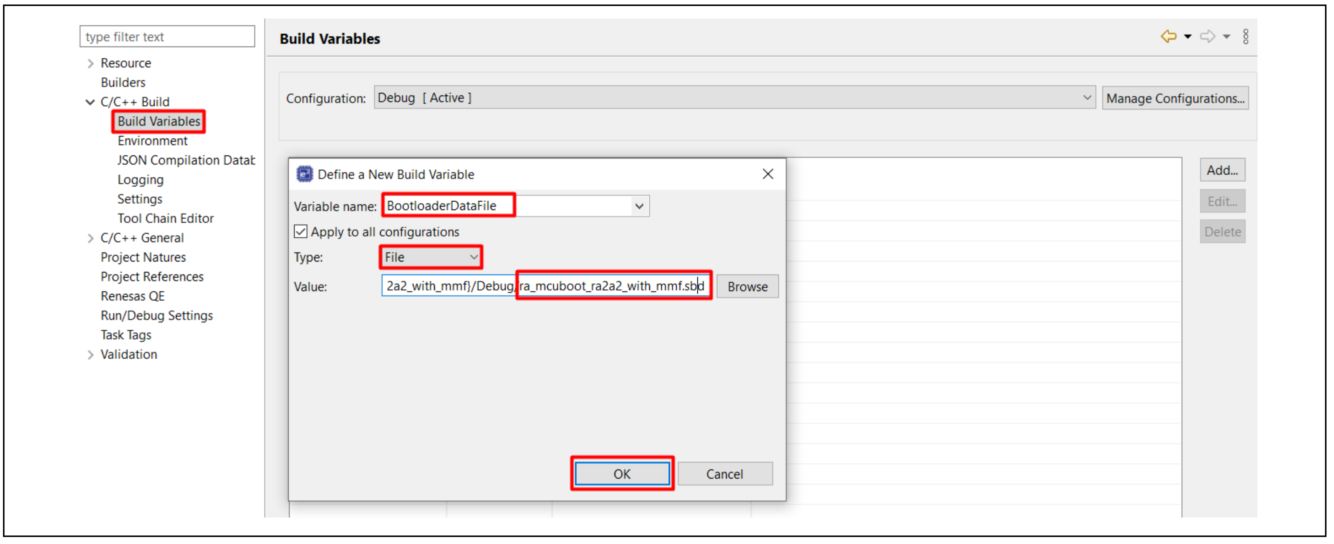

Figure 39. Configure the Build Variable to Use the Bootloader

Click **OK**, then **Apply** and **Apply and Close** in the next screen.

5.2 Signing the Application Image

Note: If you rebuild the bootloader project after changing any of the signing and signature **Properties** of the MCUboot module, you will need to **Generate Project Content** again to bring in the updated `.sbd` file.

When using Direct XIP mode, each application can define a version number. This is achieved by defining an Environment Variable: **MCUBOOT_IMAGE_VERSION**.

For applications that support signature verification, the signing key can be configured using Environment Variable **MCUBOOT_IMAGE_SIGNING_KEY**. If there is no signature verification, then it is not necessary to set Environment Variable **MCUBOOT_IMAGE_SIGNING_KEY**.

Open the **Properties** page of the project `app_primary_uart_mmf`, under **Environment**, click **Add** and configure **MCUBOOT_IMAGE_VERSION**.

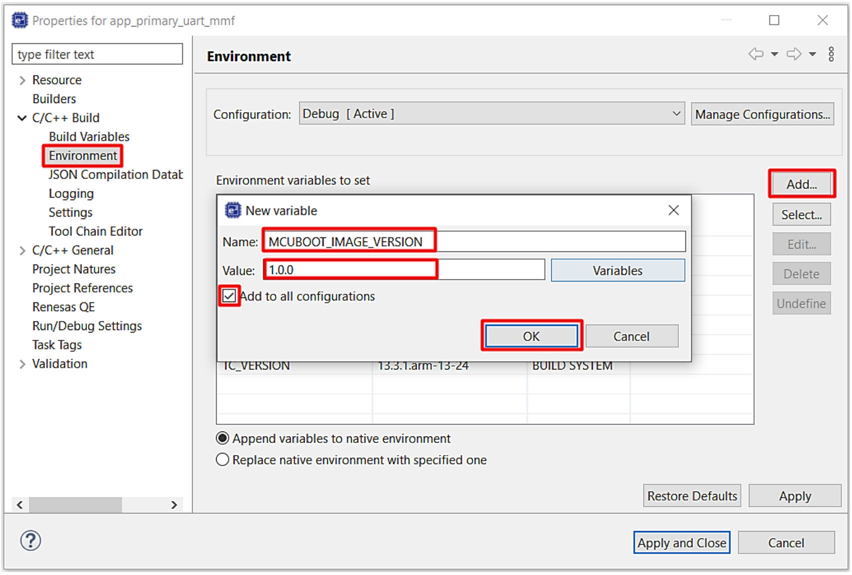

Figure 40. Configure the Application Version

Similarly, add the new variable for **MCUBOOT_IMAGE_SIGNING_KEY**.

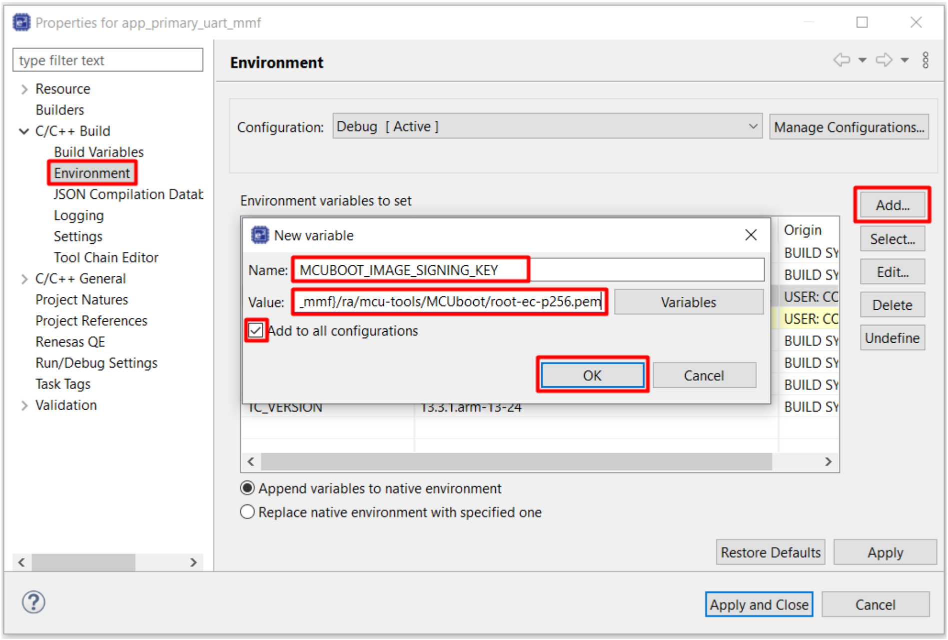

Figure 41. Configure the Private Signing Key

Note that the private key used for signing the application image is indicated in the signing command. `${workspace_loc:ra_mcuboot_ra2a2_with_mmf}/ra/mcu-tools/MCUboot/root-ec-p256.pem` is used for the example bootloaders. This key is used for testing purposes only. For real world use case and production support, users MUST change this to the private key of their choice.

Figure 42 is the result of the above configuration. Click **Apply and Close**.

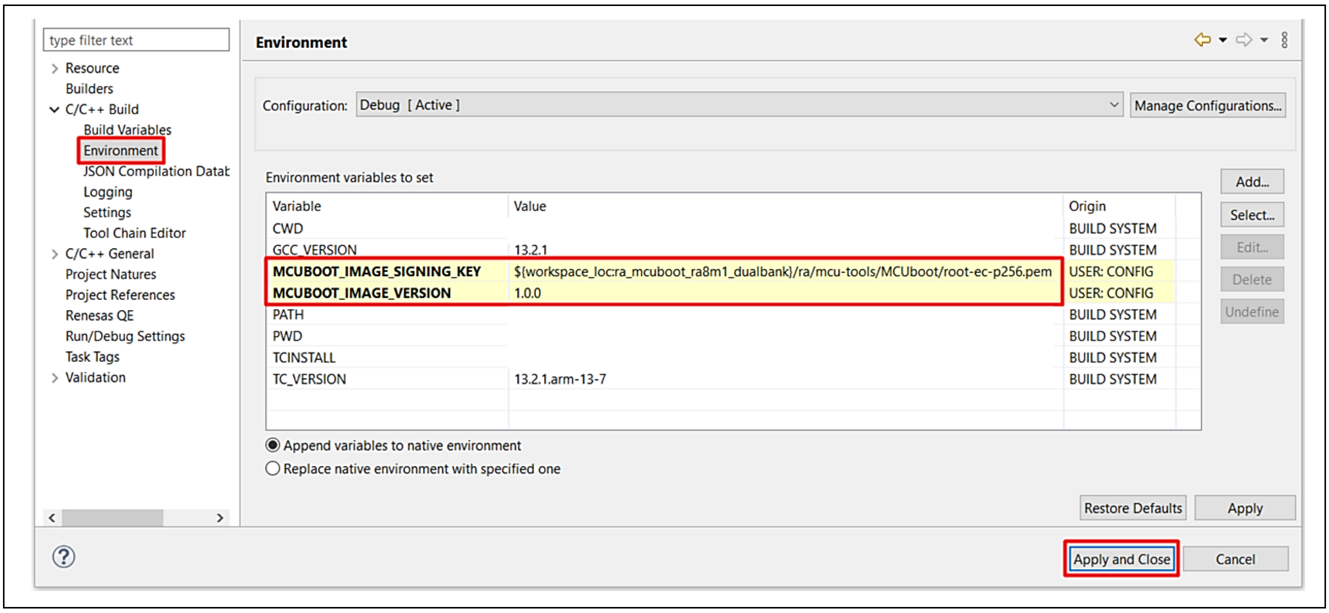

Figure 42. Configure the Application Image version number and Signing Key

To be able to recompile the project whenever the Environment Variables are updated, it is recommended add a Pre-build step to always delete the `.elf` file, as shown in Figure 43, so the application project is always recompiled.

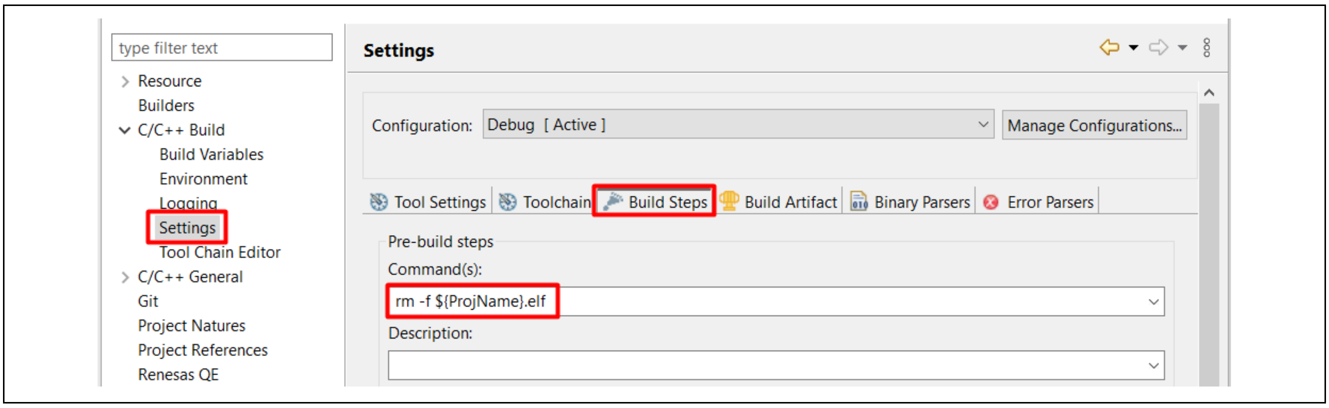

Figure 43. Configure the Pre-build Command

At this point, a user can click **Generate Project Content** and compile the newly created application project and ensure that `\Debug\app_primary_uart_mmf.bin.signed` is generated.

In addition, users need to link the primary application to the primary slot by adding a configuration. "`--defsym=XIP_SECONDARY_SLOT_IMAGE=0`" as shown in Figure 44.

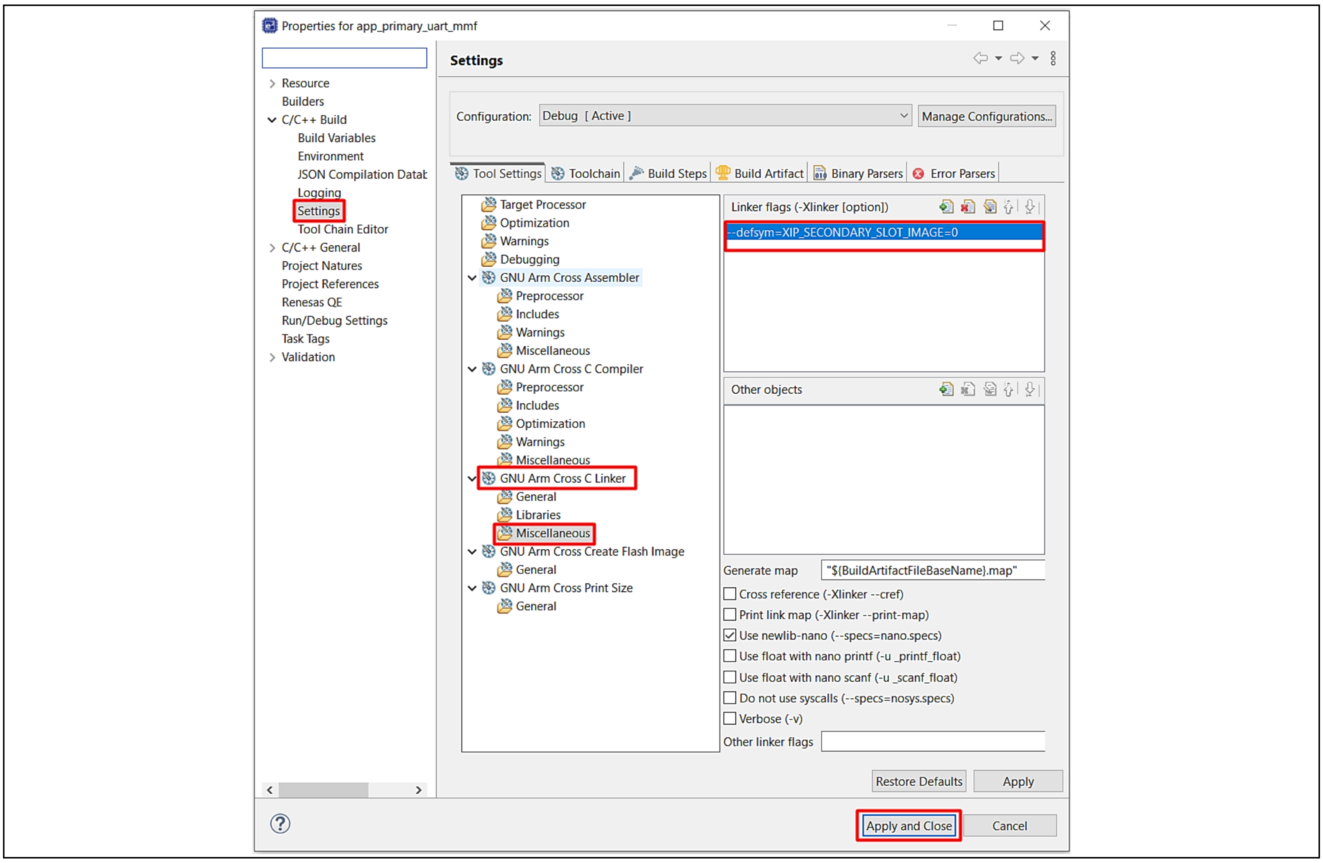

Figure 44. Configure linker flags to link an application to the primary slot

6. Booting the Primary Application and Updating to a New Image

To update the application, the primary application needs to provide an image downloader. A new image will also need to be prepared to test the image downloader function.

6.1 Prepare a Secondary Image

In this project, a secondary image is created to test the downloading functionality of the primary application. The new application can be created by either modifying the existing application or creating a new application project. If a new application project is used, the user needs to establish the linkage to the bootloader by following section 5. The newly created application project must also provide a method to download the new application to the upper bank.

In this application project, we will import the initial application project to the same workspace, rename the new project, and perform minor updates.

Right-click in the white space in the **Project Explorer** area and select **Import** and choose **Rename & Import Existing C/C++ Project into Workspace**.

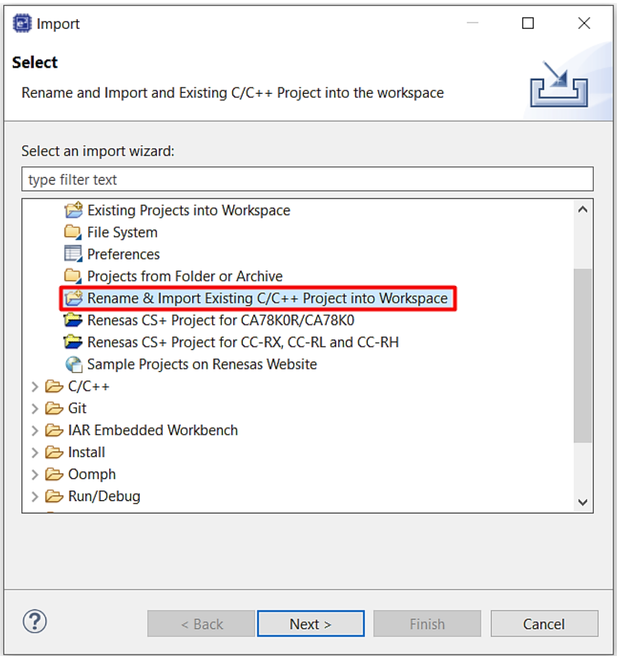

Figure 45. Import the Initial Application

Once the **Import** window opens, name the project `app_secondary_uart_mmf`, check **Select root directory**, and click **Browse**:

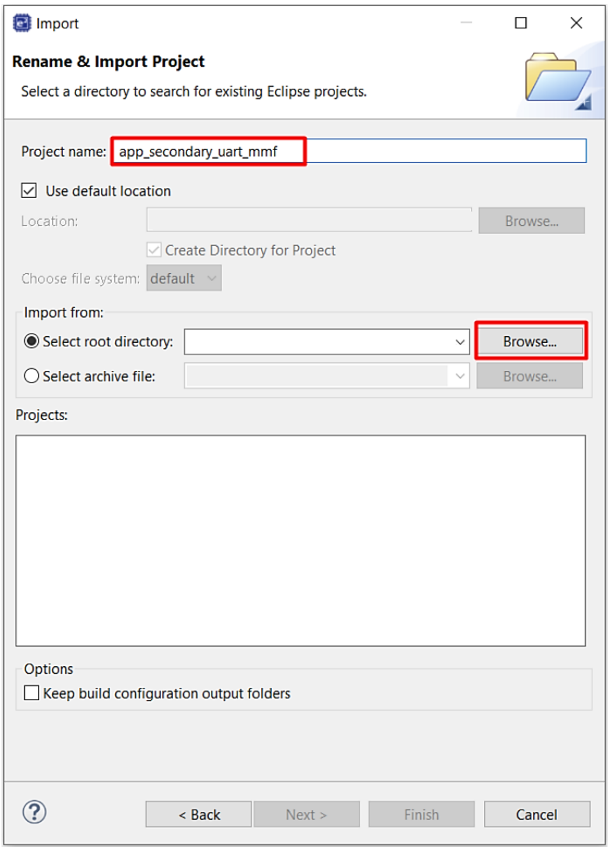

Figure 46. Name the New Application

Browse into the Workspace folder and select `app_secondary_uart_mmf`.

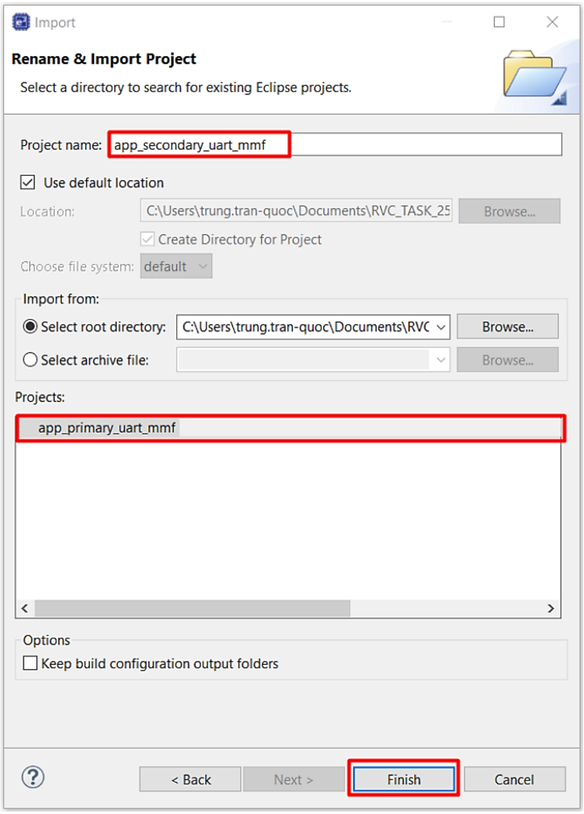

Figure 47. Select to Initial Primary Application

Click **Finish**. The new application project will be created with the following attributes:

- When importing the primary application, the **Build Variable** and **Environment Variables** are automatically imported.
- The linker flags of the primary application, as shown in Figure 44, are also automatically imported.

Change the Environment variable for the Secondary Image version, as shown in Figure 48. In DXIP mode, users must ensure that the version number of the secondary image is higher than that of the primary image.

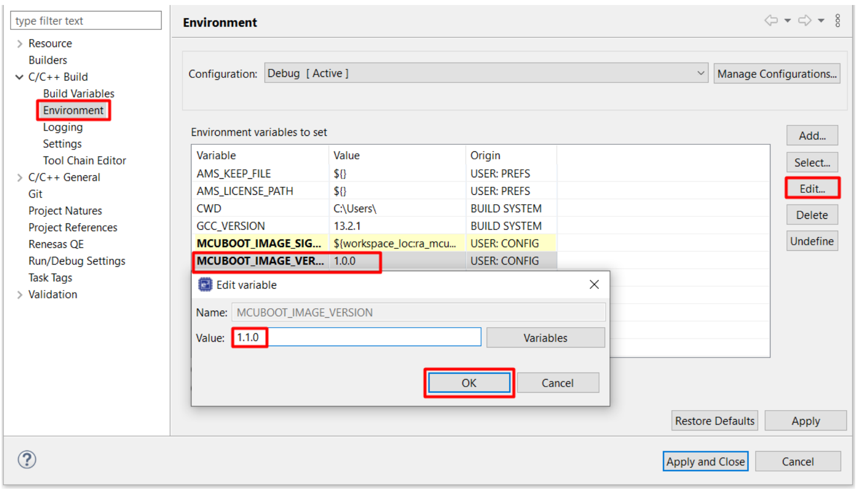

Figure 48. Change MCUBOOT_IMAGE_VERSION Variable

In addition, users need to link the secondary application to the secondary slot by adding a configuration.

"`--defsym=XIP_SECONDARY_SLOT_IMAGE=1`"

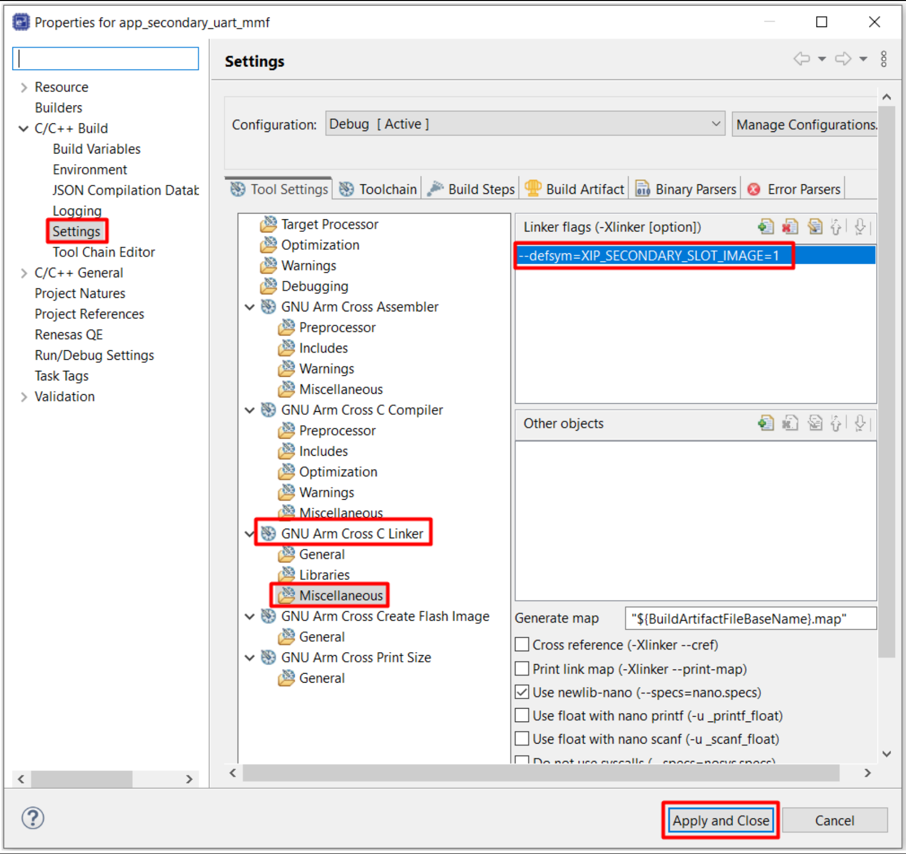

Figure 49. Update linker flags to link an application to the secondary slot

Update Existing Application to a New Application

To demonstrate the application update, update the application to blink only the blue LED.

Perform the following code updates in `blinky_thread_entry.c`:

Change below section of code in blinky_thread_entry:

```
/* 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);
 }
```

To:

```
/* update the blue led */
 R_BSP_PinWrite(leds.p_leds[0], pin_level);
```

Figure 50. Update the LED Control

Save the updated source file, click **Generate Project Content**, then compile the new project.

If you create a new application project and would like to debug the new project with the bootloader, follow the instructions in section 5. When debugging an update image with the bootloader, you can treat the update image as the primary application.

6.2 Set Up the Hardware

If using `app_primary_uart_mmf` as the initial application project:

- Connect J10 (USB Debug) using a USB micro to B cable from the EK-RA2A2 to the development PC to provide power and debug connection using the on-board debugger.

Note: On the EK-RA2A2 board, the user can use the TX and RX pins available on the debugger chip without using the UART to USB converter module.

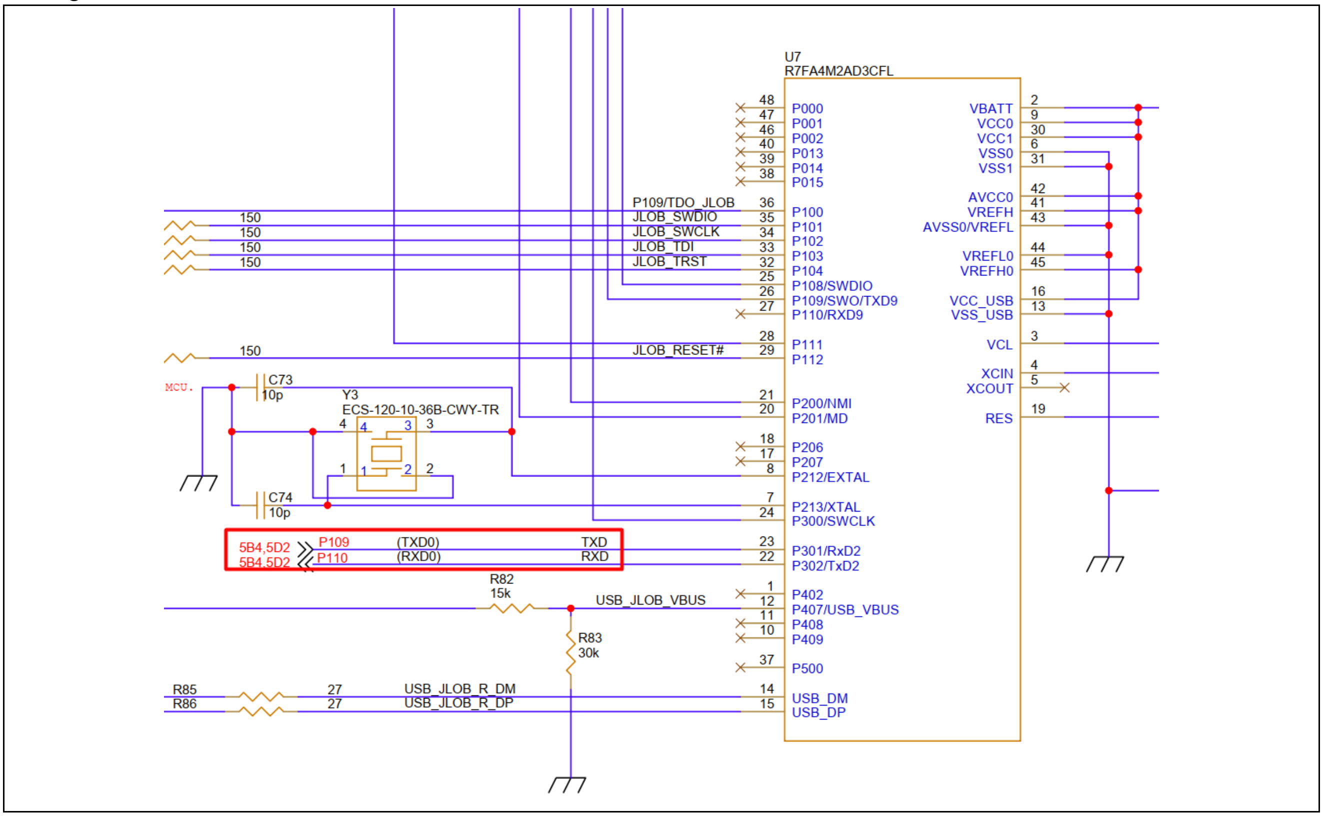

Figure 51. The TX and RX pins on the debugger chip

6.3 Erase the MCU

Once the EK-RA2A2 is powered up, the user needs to initialize the MCU prior to exercising the bootloader project. This will create a clean environment to start the bootloader project verification.

In this application project, we use J-Flash Lite to erase the entire MCU flash.

J-Flash Lite is a free, simple graphical user interface which allows downloading into flash memory of target systems. J-Flash Lite is part of the J-Link Software and Documentation package that is installed when the J-Link software & documentation pack is installed.

To use J-Flash Lite, connect the USB Debug port J10 to the PC and launch J-Flash Lite. Select the **Target Device**, debug **Target Interface**, and communication **Speed**.

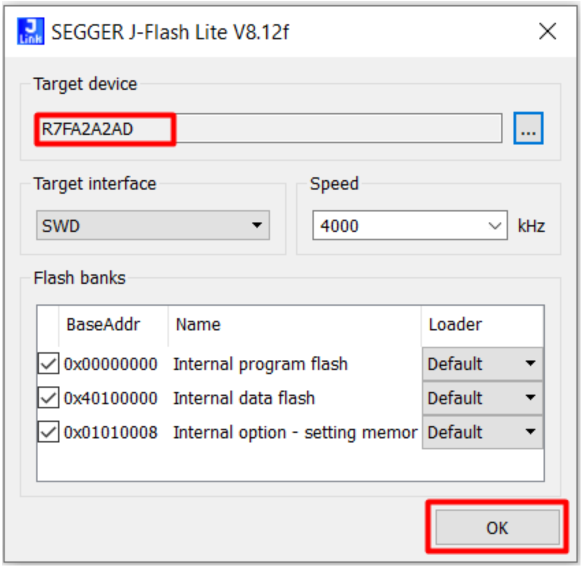

Figure 52. Launch the J-Flash Lite

Click **OK**. In the next screen, select **Erase Chip**.

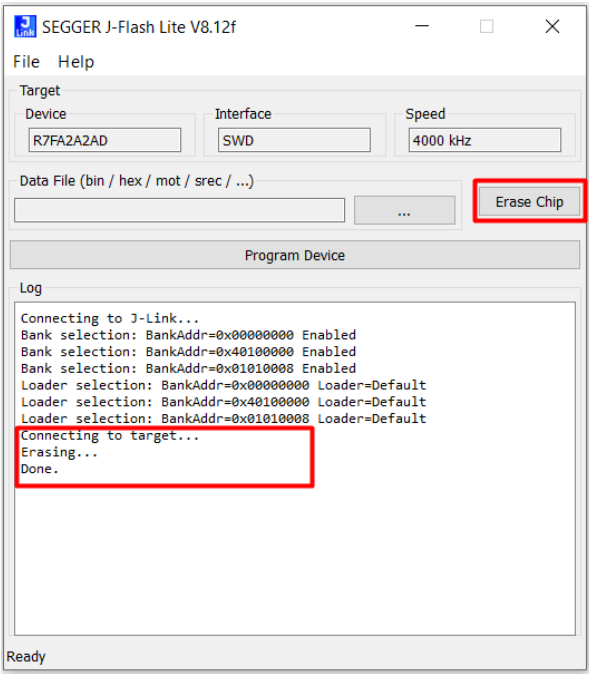

Figure 53. Erase the MCU using J-Flash Lite

6.4 Boot the Primary Application

Follow the steps below to start the debug session:

1. Disable flash content caching from the **Debugger** setting.
   Right-click on project **app_primary_uart_mmf** > **Debug As** > **Debug Configurations**, navigate to **Debugger** > **Debug Tool Settings**, and uncheck **Allow caching of flash contents**. Otherwise, when debugging bootloader applications, the memory window may show wrong information.

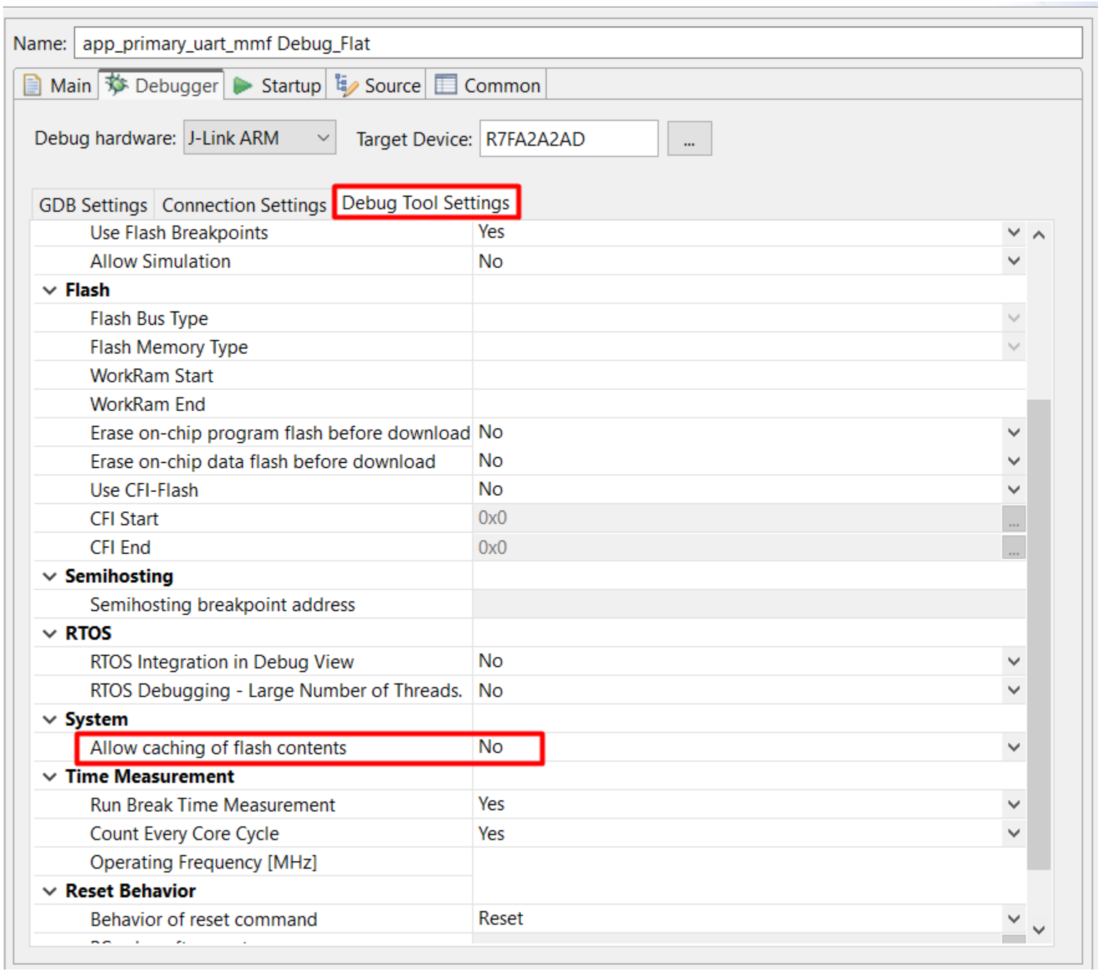

Figure 54. Disable Flash Content Caching

2. Configure the load image and symbols properties.
   Open the **Debug Configurations**: **app_primary_uart_mmf > Debug As > Debug Configurations**. Make sure **app_primary_uart_mmf Debug_Flat** is selected and select the **Startup** tab, then confirm that the following configuration exists.

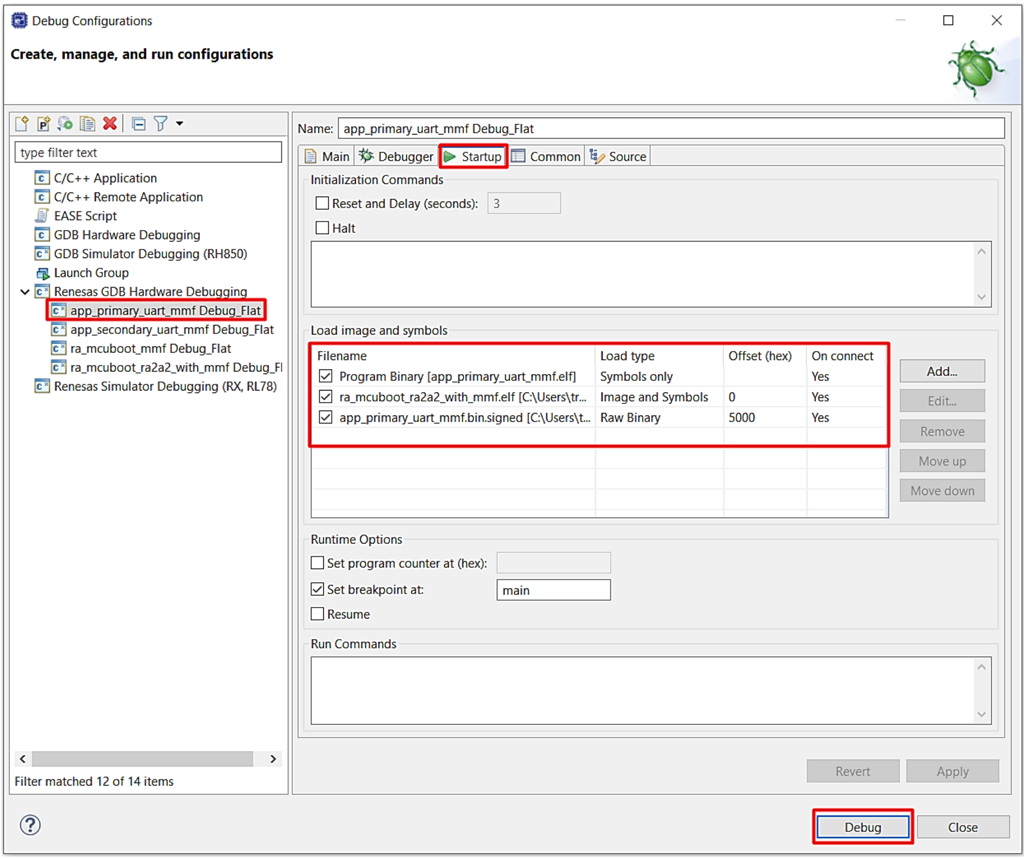

Figure 55. Debug Configurations

- Under the Startup configuration, verify the Load type of `app_primary_uart_mmf.elf` is **Symbols only** rather than **Image and Symbols**.
- The `ra_mcuboot_ra2a2_with_mmf.elf` is added with Load type as **Image and Symbols** with an Offest 0 since the bootloader starts form 0x0.
- The `app_primary_uart_mmf.bin.signed` entry exists with Load type as **Raw Binary** and the Offset is set to 0x5000 since that is the beginning of the primary application, as shown in Figure 27.

3. Click **Debug**. Choose **Remember my decision** and click **Switch** if prompted to switch the perspective.

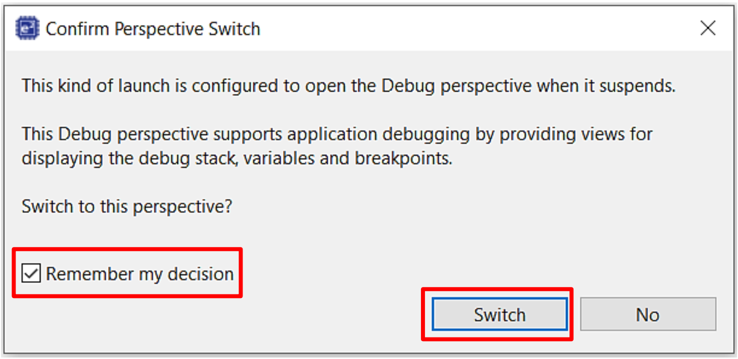

Figure 56. Start the Application Execution

4. The debugger should hit the reset handler in the bootloader.

```
BSP_SECTION_FLASH_GAP void Reset_Handler (void)
{
    /* Initialize system using BSP. */
    SystemInit();

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

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

Figure 57. Switch the Perspective

5. Click **Resume** to run the project.
   The program should now be paused in `main` at the `hal_entry()` call in the bootloader.

```
/* generated main source file - do not edit */
#include "hal_data.h"
int main(void)
{
    hal_entry ();
    return 0;
}
```

Figure 58. Start the Application Execution

6. Click to run again.
   The red, blue, and green LEDs on the EK-RA2A2 should now be blinking while the blinky application is running.

6.5 Program the New Application Using the Primary Application Downloader

Follow the steps below to program the new application created in section 6.1:
Note that when using the UART interface, users need to open the Tera Term and configure the Baud Rate first, as shown in Figure 59 and Figure 60. Then, the debug session can be started from the primary application.

1. Open Tera Term and choose the JLink CDC UART Port (COM number may be different for your setup), as shown in Figure 59. Then click **OK**.

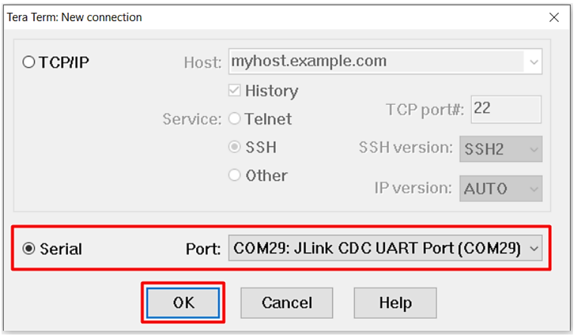

Figure 59. Open the USB COM Port

2. Select the Serial Terminal and set the **Speed** to 115200, as shown in Figure 60. Then click **New setting**.

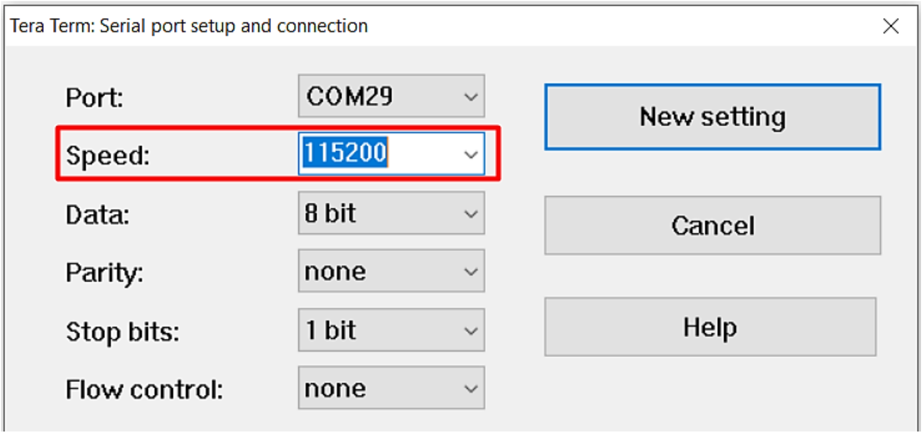

Figure 60. Configure the Baud Rate

The menu in Figure 61 Tera Term Menu will be displayed on the Tera Term.

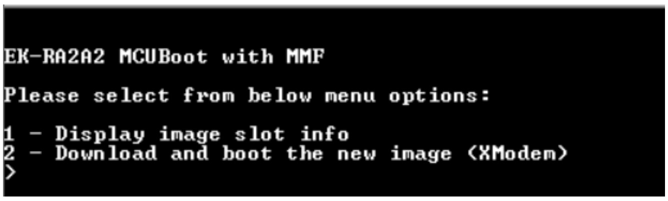

Figure 61. Tera Term Menu

3. Select option **1** to print the image slot information**.**

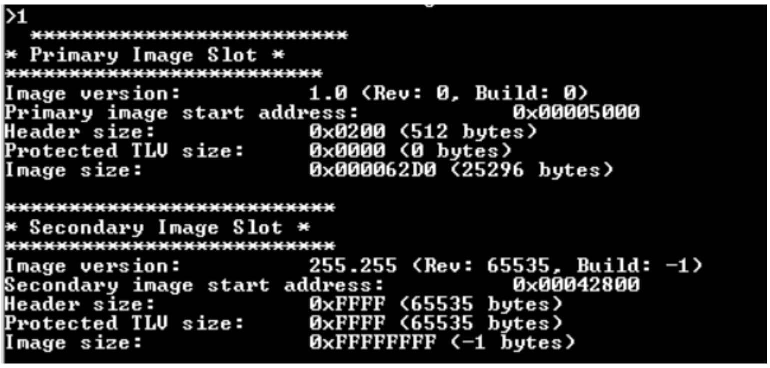

Figure 62. Print the Image Slot Information

4. Select option **2** to download the secondary image using the primary image downloader.

Figure 63. Choose Option 2 to Download the New Image using XModem

5. Open the **Transfer** interface of the Tera Term.

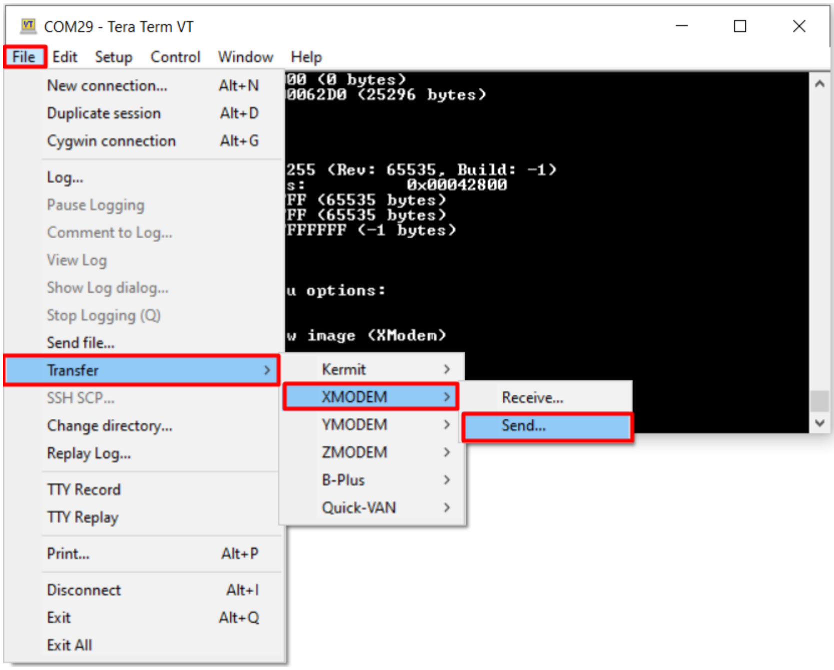

Figure 64. Start Transfer from Tera Term

6. Choose `\app_secondary_uart_mmf\Debug\app_secondary_uart_mmf.bin.signed`, then click **Open**.

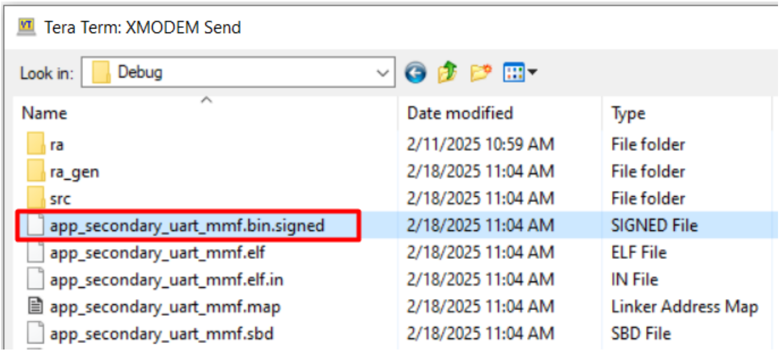

Figure 65. Choose the Signed Secondary Image

The secondary image is then downloaded and programmed to the secondary slot.

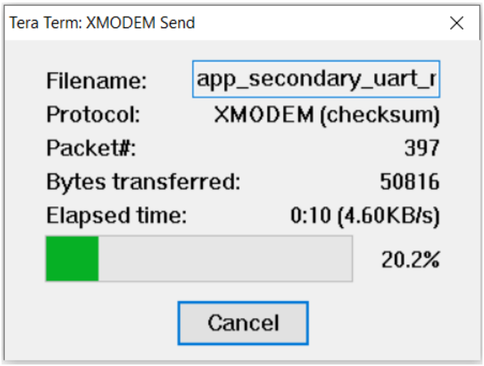

Figure 66. Download the New Image via XModem

6.6 Boot the New Application

The system will automatically reboot after the new image is downloaded.

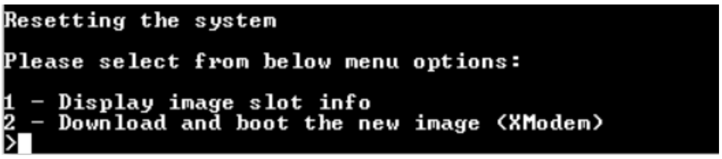

Figure 67. The New Image is Booted

Select option **1** to read the swapped memory layout.

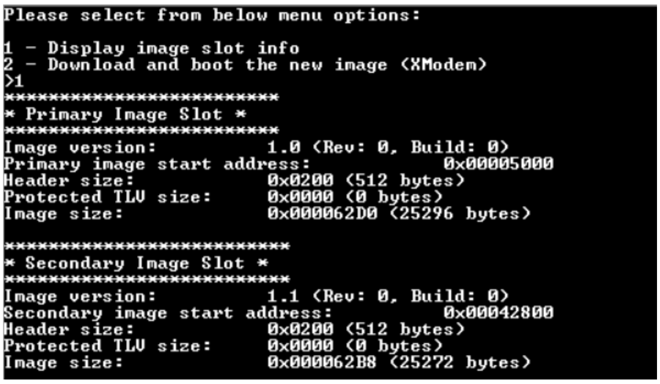

Figure 68. The Slot Layout After New Image is Booted

Note that even though the secondary image is booted, it cannot be debugged as the symbol downloaded to the debugger is for the primary image.

Also, if you want to perform further update, the new image must have a version of higher than the current image in the primary slot.

7. Memory Mirror Address When Booting Image

In this section, we will help users better understand the operation of the Memory Mirror Function (MMF) when combining the MCUboot.

The MMSFR register will contain the start address of the image on code flash. For more details about the MMSFR register, users can refer to **section 5.2: Register Descriptions** in the RA2A2 Hardware User's Manual.

5.2.1 MMSFR : MemMirror Special Function Register

Base address: MMF = 0x4000_1000

Offset address: 0x00

Bit position:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Bit field:	KEY[7:0]								—	MEMMIRADDR[15:0]						
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit position:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit field:	MEMMIRADDR[15:0]									—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 69. MemMirror Special Function Register

When booting the primary image, the MMSFR register will store the start address of the primary image (it includes the image header size), as shown in Figure 70 MMSFR Register when booting the primary image.

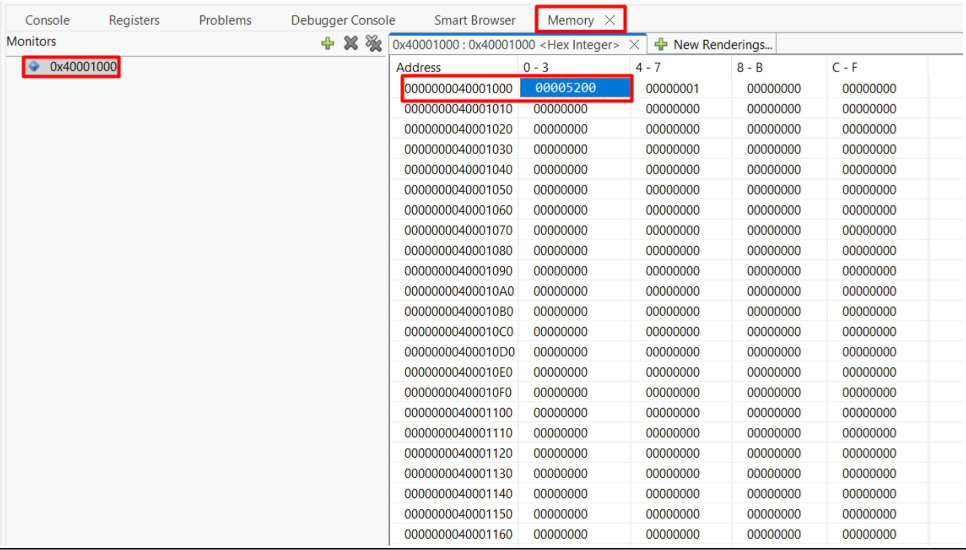

Figure 70. MMSFR Register when booting the primary image

After the secondary image is downloaded, the MMSFR register will also store the start address of the secondary (it includes the image header size), as shown in Figure **71**.

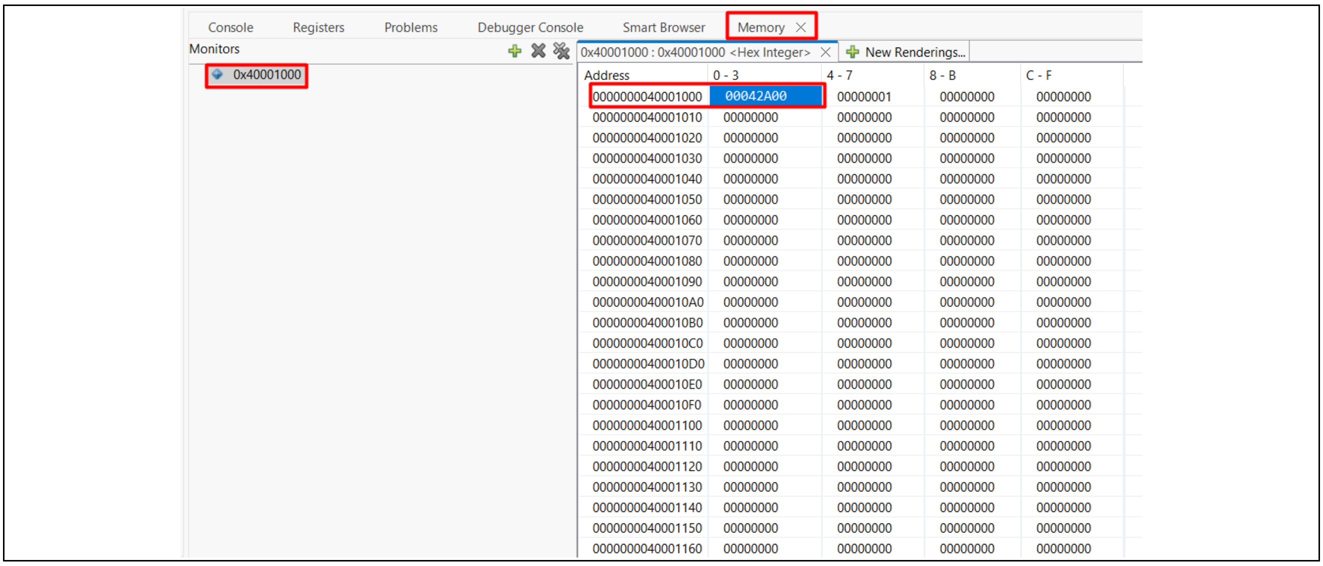

Figure 71. MMSFR Register when booting the secondary image

8. Production Support Considerations

This section describes one possible flow of production flow. Users may adapt this procedure to their own needs wherever possible.

8.1 Protect the Bootloader using Memory Protection Unit and Flash Access Window

In this application, we only need to focus on the secure flash program and data regions.

Users need to determine the bootloader size and set the boundaries for both the secure flash data and program regions within this area, as shown in Figure 5.

For the secure flash program region, users can configure the Security MPU Regions in the `ra_mcuboot_ra2a2_with_mmf` project under the **BSP** tab.

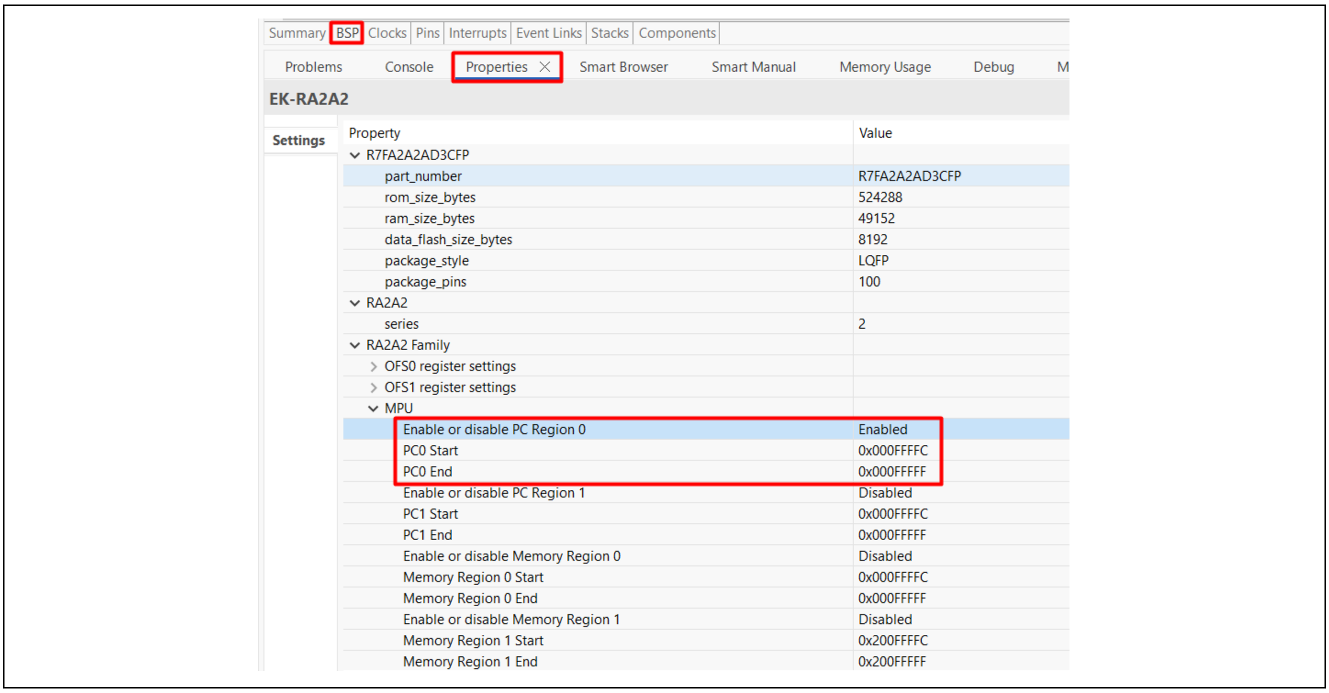

Figure 72. Security MPU Configuration

- Secure flash program
  — **Enable or disable PC Region 0:** enable or disable the secure flash program.
  — **PC0 Start** and **PC0 End**: program counter region for the secure flash program.

For the secure flash data region, users need to create a customized linker script to define it, as shown in Figure 73. For more details, users can refer to section 5 in Application Project R11AN0416.

```
/* Linker script to configure memory regions. */
MEMORY
{
        VECTOR_TABLE (rx)       : ORIGIN = 0x00000000, LENGTH = 0x00000400   /*1024 bytes */
        SECURE_PROGRAM (rx)     : ORIGIN = 0x00000400, LENGTH = 0x0007FC00   /*  512K - 1024 bytes  */
        SECURE_DATA   (rw)      : ORIGIN = 0x00080000, LENGTH = 0x00080000    /*  512K bytes */
        FLASH (rx)              : ORIGIN = 0x00100000, LENGTH = 0x00100000   /*  1MB */
        SECURE_RAM_PROGRAM (rwx)    : ORIGIN = 0x1FFE0000, LENGTH = 0x00010000 /*    64K     */
        SECURE_RAM (rw)         : ORIGIN = 0x1FFF0000, LENGTH = 0x00040000   /*      256K    */
        RAM (rwx)               : ORIGIN = 0x20030000, LENGTH = 0x00050000   /*      320K    */
        DATA_FLASH (rx)         : ORIGIN = 0x40100000, LENGTH = 0x00008000   /*  32K         */
        QSPI_FLASH (rx)         : ORIGIN = 0x60000000, LENGTH = 0x04000000   /*  64M         */
        SDRAM (rwx)             : ORIGIN = 0x90000000, LENGTH = 0x02000000   /*  32M         */

}
```

Figure 73. Customized Linker Script for Secure Data Region

Note that this is an example of the customized linker script, users need to calculate the memory regions to fit their application project.

For the FAW region, users only need to call the FSP FAW API:

```
err = R_FLASH_LP_AccessWindowSet(&g_flash0_ctrl, FAW_START, FAW_END);
```

where:

- `g_flash0_ctrl` is the instance of this flash HAL driver.
- `FAW_START` is the start address of the FAW window.
- `FAW_END` is the address of the next block acceptable for programming and erasure defined by the access window.

Note:

- If the FAW is permanently locked before running this API, the FAW region cannot be updated using this API.
- It is always recommended to set up the FAW region outside of Security MPU Regions, as shown in Figure 5.

9. Appendix: Compile and Exercise the Included Example Bootloader and Application Projects

There are three projects:

- `ra_mcuboot_ra2a2_with_mmf`
- `app_primary_uart_mmf`
- `app_secondary_uart_mmf`

Users can follow the steps below to run the example projects in the folder `\ra2-secure-bootloader-using-mcuboot-internal-code-flash-mmf`.

1. Follow the instructions in section 6.2 to set the hardware.
2. Import the above-mentioned three projects to a workspace.
3. Open the `configuration.xml` file from project `ra_mcuboot_ra2a2_with_mmf`.
4. Click **Generate Project Content**.
5. Follow section 4.6 to set up the Python dependencies. Skip this step if the dependencies are already met.
6. Compile the project `ra_mcuboot_ra2a2_with_mmf`.
7. Open the `configuration.xml` file from project `app_primary_uart_mmf`.
8. Click **Generate Project Content**.
9. Compile the project `app_primary_uart_mmf`.
10. Open the `configuration.xml` file from project `app_secondary_uart_mmf`.
11. Click **Generate Project Content**.
12. Compile the project `app_secondary_uart_mmf`.
13. Erase the entire chip following the instructions in section 6.3.
14. Debug the application from project `app_primary_uart_mmf` in the e^2 studio environment.
15. Resume the program execution twice. All three LEDs should be blinking.
16. Open the Tera Term with the enumerated COM port and set up the baud rate as 115200.
17. Use Tera Term to send the `\app_secondary_uart_mmf\Debug\app_secondary_uart_mmf.bin.signed` to the MCU by following section 6.6. This will take around 50 seconds.
18. The system will reset automatically after downloading.
19. Blue LED should be blinking.
20. Enter menu item 1 to confirm the image with version 1.1.0 is located in the secondary slot and the image with version 1.0.0 is located in the primary slot.

10. References

1. Renesas RA Family RA2 Series MCU Secure Bootloader Design using MCUboot Application Project (R11AN0516)
2. Renesas RA Family RA6 Series MCU Basic Secure Bootloader Design using MCUboot with Code Flash Linear Mode Application Project (R11AN0497)
3. Renesas RA Family RA8 Series MCU Basic Secure Bootloader Using MCUboot and Internal Code Flash (R11AN0909)
4. Renesas RA Family MCU Securing Data at Rest using Security MPU Application Project (R11AN0416)

11. Website and Support

Visit the following URLs to learn about the RA family of microcontrollers, download tools and documentation, and get support.

EK-RA2A2 Resources	renesas.com/ra/ek-ra2a2
RA Product Information	renesas.com/ra
Flexible Software Package (FSP)	renesas.com/ra/fsp
RA Product Support Forum	renesas.com/ra/forum
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	May 20. 25	-	Initialize release

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

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

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

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

Contact information

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