Renesas RA Family

Secure Bootloader using MCUboot with the Ocrypto Cryptographic Library

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 operating system and hardware independent and relies on hardware porting layers. MCUboot is maintained by TrustedFirmware 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 those documents wherever possible.

The Renesas Flexible Software Package (FSP) integrates an MCUboot port across the entire RA MCU Family starting from FSP v3.0.0. The Renesas RA MCU family, based on the Arm® Cortex®-M cores, includes devices with varying levels of flash and RAM resources. This application project is designed to address the challenges of limited memory environments and provide guidelines for optimizing bootloader memory usage. For the MCUboot cryptographic support for select MCU groups, Ocrypto is integrated with the FSP MCUboot module to offer an optimized memory footprint. Licensed from Oberon Microsystems AG, Ocrypto is supported by select MCU Groups.

This application note guides you through secure bootloader creation using the MCUboot Module with Ocrypto for enhanced security on the Renesas EK-RA2E1 kit. In addition, examples of how to configure the application project to use the bootloader are provided.

For the Renesas RA6 and RA4 MCU Series, Renesas provides an application project “Using MCUboot with Renesas RA MCU Application Project”, which guides you through using MCUboot with RA6 and RA4 MCU groups. See the References section for information on that application project.

Required Resources

Development Tools and Software

- e² studio IDE v2025-07
- Renesas Flexible Software Package (FSP) v6.1.0
- SEGGER J-link® USB driver v8.58

The above three software components: the FSP, J-Link USB drivers, and e² studio are bundled in a downloadable platform installer available on the FSP webpage at renesas.com/ra/fsp.

Hardware

- EK-RA2E1 Evaluation Kit for RA2E1 MCU Group (http://www.renesas.com/ra/ek-ra2e1).
- Workstation running Windows® 10.
- One USB device cable (type-A male to micro-B male).

Prerequisites and Intended Audience

This application note assumes that you have some experience with the Renesas e² studio IDE. You should read the entire MCUboot Port section in the *FSP User’s Manual* prior to moving forward with this application project. In addition, the application note assumes that you have some knowledge of cryptography. Prior knowledge of Python usage is also helpful.

The intended audience are product developers, product manufacturers, product support, and end users who are involved with designing application systems involving use of a secure bootloader with the Renesas RA2 MCU family.

Using this Application Note

Section 1 presents a general overview of MCUboot and the application upgrade methods supported by MCUboot.

Section 2 describes the general flow of using the FSP MCUboot module to establish bootloader-based application systems.

Section 2.4 to Section 7 are the walk-throughs of how to create bootloader projects using Overwrite, and Swap upgrade modes, how to configure the application projects to use the bootloader, and how to boot the primary and secondary images.

Section 8 provides instructions on how to directly run the included example projects without going through sections 3 to 7. For a quick evaluation of the included example projects, users can go directly to Section 8.

Section 9 provides several solutions for MCUboot and the Application Project in the production phase.

Contents

1. Overview of MCUboot

1.1 History of MCUboot

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 2020, MCUboot was moved under the Linaro Community Project umbrella as an open-source project, and it now resides under TrustedFirmware (https://www.trustedfirmware.org/projects/mcuboot).

1.2 MCUboot Functionalities Overview

MCUboot handles the firmware integrity and authenticity check after startup and the firmware switch part of the firmware update process. The operation of switching the firmware from the original image to a new image depends on the image upgrade method. The image upgrade methods are described in section 1.2.2. 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.

1.2.1 Overview of Application Booting Process

For applications using MCUboot, the MCU memory is separated into MCUboot, Primary App, Secondary App, and the Scratch Area. The following is an example of the single-image MCUboot memory map. For more information on the MCUboot memory layout, refer to the Flash Map Section of the MCUboot website.

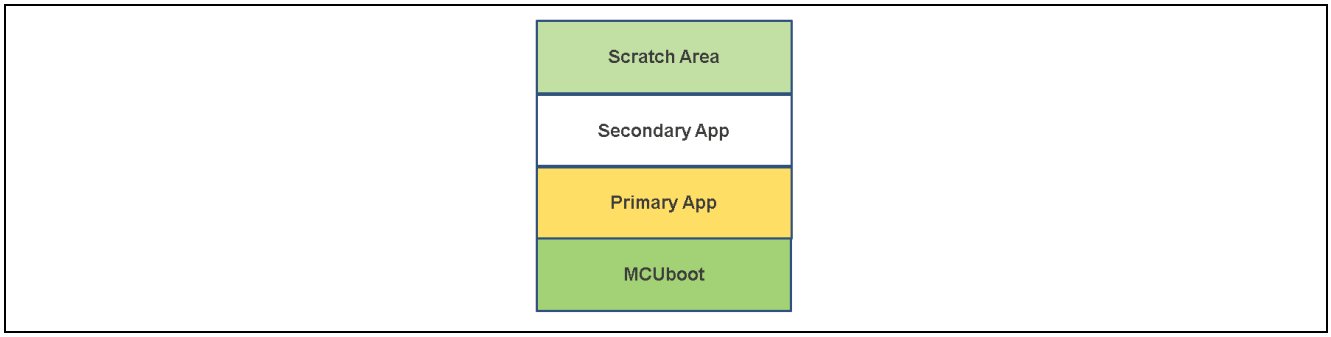

Figure 1. Single Image MCUboot Memory Flash Map

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

1. The bootloader is started when the CPU is released from reset.
2. If there are images in the Secondary App memory marked as to be updated, the bootloader performs the following actions:
   A. The bootloader verifies the integrity and authenticity of the Secondary image.
   B. Upon successful authentication, the bootloader switches to the new image based on the update method selected.
   C. The bootloader boots the new image.
3. 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. The firmware image can be authenticated by hash (SHA-256) and digital signature validation.

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 these operations.

1.2.2 Application Update Strategies

The following update strategies are supported by MCUboot. The Renesas FSP MCUboot Module supports one or more of the following strategies depending on the FSP version. The analysis of pros and cons is based on the MCUboot functionality, not the FSP version specific MCUboot Module functionality. In addition, this application note is not intended to provide all details on the MCUboot application update strategies. We recommend acquiring more details on these update strategies by referring to the MCUboot design page:

https://github.com/mcu-tools/MCUboot/blob/master/docs/design.md

- **Overwrite**
  In the Overwrite update mode, the active firmware image is always executed from the Primary slot, and the Secondary slot is a staging area for new images. Before the new firmware image is executed, the entire contents of the Primary slot are overwritten with the contents of the Secondary slot (the new firmware image).
  - Pros:
    - Fail-safe and resistant to power-cut failures.
    - Less memory overhead, with a smaller MCUboot trailer and no Scratch Area.
    - Encrypted image support is available when using external flash.
  - Cons:
    - Does not support pre-testing of the new image prior to overwrite.
    - Does not support automatic application fallback mechanism.

  Overwrite upgrade mode is supported by Renesas RA FSP v3.0.0 or later.

- **Swap**
  In the Swap image upgrade mode, the active image is also stored in the Primary slot and is always started by the bootloader. If the bootloader finds a valid image in the Secondary slot that is marked for upgrade, then contents of the Primary slot and the Secondary slot are swapped. The new image then starts from the Primary slot.
  - Pros:
    - The bootloader can revert the swapping as a fallback mechanism to recover the previous working firmware version after a faulty update.
    - The application can perform a self-test to mark itself permanent.
    - Fail-safe and resistant to power-cut failures.
    - Encrypted image support is available when using external flash.
  - Cons:
    - Need to allocate a Scratch Area.
    - Larger memory overhead, due to a larger image trailer and additional Scratch Area.
    - Larger number of write cycles in the Scratch Area, wearing the Scratch sectors out faster.

  Swap upgrade mode is supported by Renesas RA FSP v3.0.0 or later. Runtime image testing is supported from FSP v3.4.0 or later.

- **Direct execute-in-place (XIP)**
  In the direct execute-in-place 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.
  - Pros:
    - Faster boot time, as there is no overwrite or swap of application images needed.
    - Fail-safe and resistant to power-cut failures.
  - Cons:
    - Added application-level complexity to determine which firmware image needs to be downloaded.
    - Encrypted image support is not available.

  Direct execute-in-place is supported by Renesas FSP v3.4.0 or later.

- **RAM loading firmware update**
  Like the direct execute-in-place mode, RAM loading firmware update mode selects the newest image by reading the image version numbers in the image headers. However, instead of executing it in place, the newest image is copied to RAM for execution. The load address (the location in RAM where the image is copied to) is stored in the image header. This upgrade method is not typically used in an MCU environment. This image update mode does not support encrypted images. Refer to the MCUboot Design Page for more information on this update strategy.
  RAM loading update mode is not supported by the Renesas RA FSP.

2. Architecting an Application with MCUboot Module using FSP for RA2 MCUs

This section provides an overview of the FSP MCUboot Module, the available application image upgrade modes, memory architecture design, and guidelines for mastering the new image. In addition, this section describes how the lightweight Ocrypto is used in the RA2E1 bootloader design. We recommend reviewing the MCUboot Port section of the *FSP User's Manual* to understand the build time configurations for MCUboot.

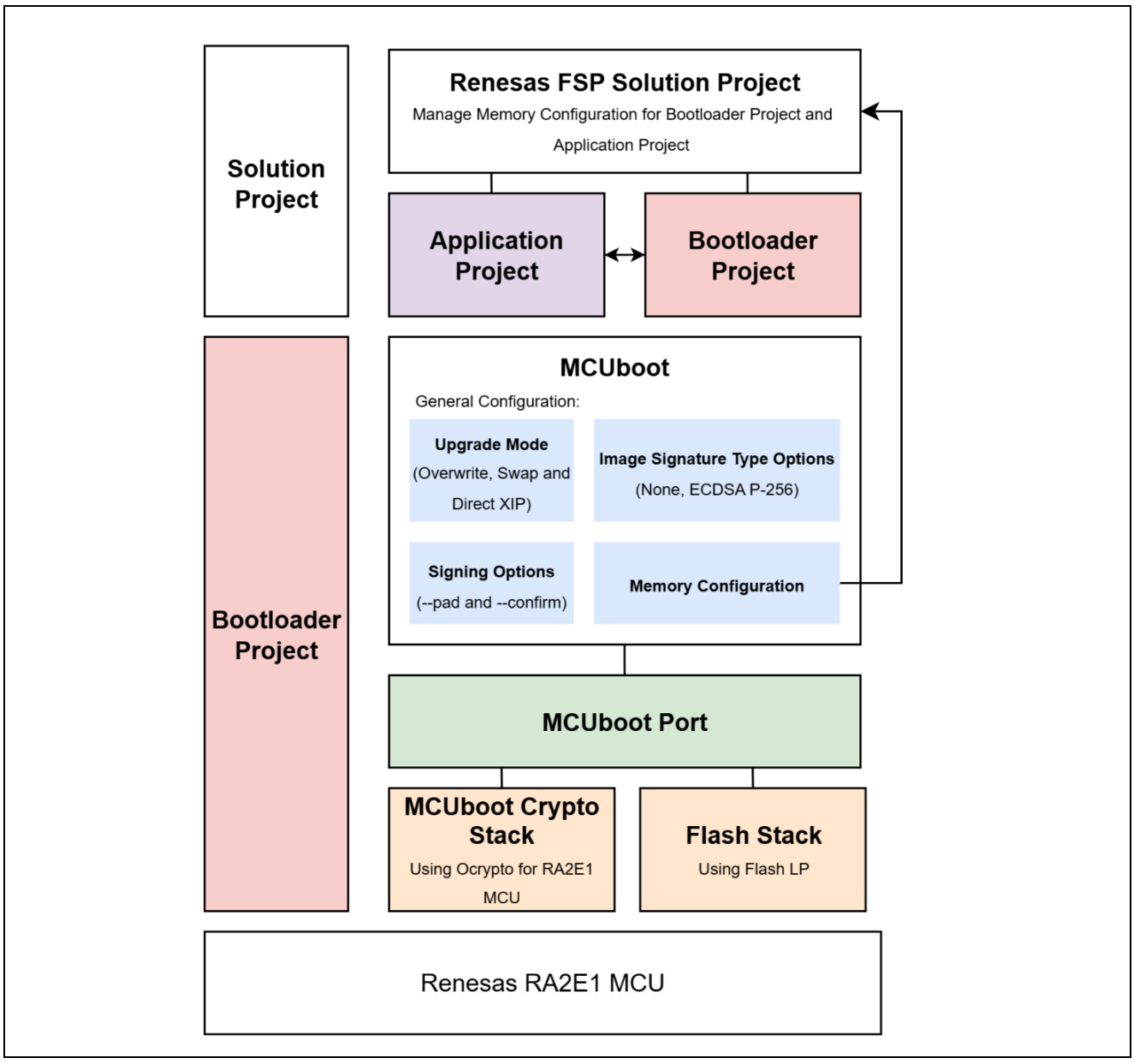

Figure 2. MCUboot Architecture and Project Relationship in Renesas FSP

For the **MCUboot Module General Configuration**, users can refer to the section 2.4 for more details.

In addition, starting from FSP v6.0.0 with e^{2}studio v2025-04.1, the MCUboot Memory Configuration is managed under the **Renesas FSP Solution Project**. Refer to section 5 for more details.

2.1 Secure Booting with Ocrypto

Ocrypto is a lightweight crypto software targeting constrained devices. Its minimal set of standard cryptographic primitives are designed to provide secure messages, basic encryption, and random number generation, which are all needed to secure the small footprint of IoT devices. For the RA2E1 bootloader

design, SHA256 and ECDSA from Ocrypto are used to ensure the application image integrity and authenticity.

The FSP Ocrypto Port Module does not provide any interfaces to the user. Consult the documentation at https://www.oberon.ch/products/ocrypto/algorithms/ for further information on the use of the Ocrypto port module. The Ocrypto Stack module on RA2E1 MCU is available in two forms:

- Ocrypto (S/W Only) - suitable for use cases where image encryption is not required.
- Ocrypto (H/W Accelerated) - suitable for use cases where image encryption is required. The encrypted image is typically used when image needs to be stored in external storage. For the EK-RA2E1 board, external storage (external flash) is not supported, so internal code flash is used for demonstration purposes.

In addition, Ocrypto helps you reduce:

- The memory footprint (small code, small stack and no heap).
- The execution time and energy consumption.
- The time to market.
- The cost of maintenance.
- The vulnerability to common side-channel attacks.

Note: RSA is currently not supported as a **Signature Type** in the **FSP Ocrypto Port Module**.

2.2 Designing Bootloader and the Initial Primary Application Overview

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

1. Bootloader development and memory allocation
   - Analyze the MCU memory resource allocation requirements for both the bootloader and the application.
   - Consider factors such as update mode, signature type, and whether to validate the Primary Image, as well as the cryptographic library used. These factors is influence the bootloader memory usage.
2. Initial primary application development
   - Develop the initial primary application, perform the memory usage analysis.
   - Compare with the bootloader memory allocation to ensure consistency, and adjust if necessary.
3. Bootloader configuration
   - Configure image authentication and new image update mode.
   - Update memory allocation definitions in the bootloader project if required.
4. Application image signing
   - The IDE tools will use the signing command to sign the application and generate a binary file for downloading to the MCU.
5. Testing
   - Verify the bootloader and the initial primary application to ensure correct operation.

2.3 Guidelines for Using the MCUboot Module with RA2 Series MCUs

The MCUboot Module is supported on all RA Family MCUs. For the Renesas RA2 Cortex-M23 MCU series, image hashing and image authentication are supported in FSP v3.4.0 and later.

2.3.1 Customizing the RA2 Bootloader

Customizing the bootloader involves the following main aspects:

- Customized method to download the application. This is very application specific and is not discussed in this application project.

- Bootloader size optimization.
  Some of the bootloader size optimization actions that can be taken are summarized as follows:
  - Disable application image validation to reduce code size.
  - Disable image signing to reduce code size.
  - Disable unused FSP components to reduce code size.
  - Use pin configurations that initialize fewer peripheral and I/O pins.

  Details on the operational flow of these optimization are described in section 2.4.

2.3.2 Time Usage in an Application Image Update

Several factors can significantly influence how long an application image update takes. The main factors include:

1. **Image Verification (ECC/RSA)**
   - Larger image sizes require more time to verify.
   - The verification time depends on both **image size** and the chosen **cryptographic algorithm**.
2. **Flash Erase and Programing Time**
   - The larger the application image, the longer the erase and programming time during the upgrade process.
   - This applies to **Overwrite** and **Swap** modes where flash operations are involved.
   - The erase/programming time can be estimated from the **Flash Memory Characteristics** section in the MCU Hardware User's Manual.
3. **Upgrade Mode**
   - The selected upgrade mode (Overwrite or Swap) directly impacts the number of erase and programming operations:

   Overwrite Mode
   - 2 x erase time (primary + secondary slot).
   - 1 x programming time (primary slot only).

   Swap Mode
   - 2 x erase time (primary + secondary slot).
   - Multiple erase/program operations in the scratch area (total equivalent to image size, aligned to scratch area size).
   - 2 x programming time (primary + secondary slot).
4. **Signature Algorithms Used**
   - The choice of signature algorithm also affects verification speed.
   - ECC offers faster verification compared to RSA for the same image size.
   - On RA2 MCU, only ECC is supported for signature verification.
5. **Image Signing Option**
   - If image signing is enabled, additional cryptographic operations are required, which increase the update time.
   - If image signing is disabled, the signature verification step is skipped, resulting in faster update times.

We also measured the boot time with and without the **Image Signing Option**, using an application size of approximately 51 KB, as shown in Figure 3.

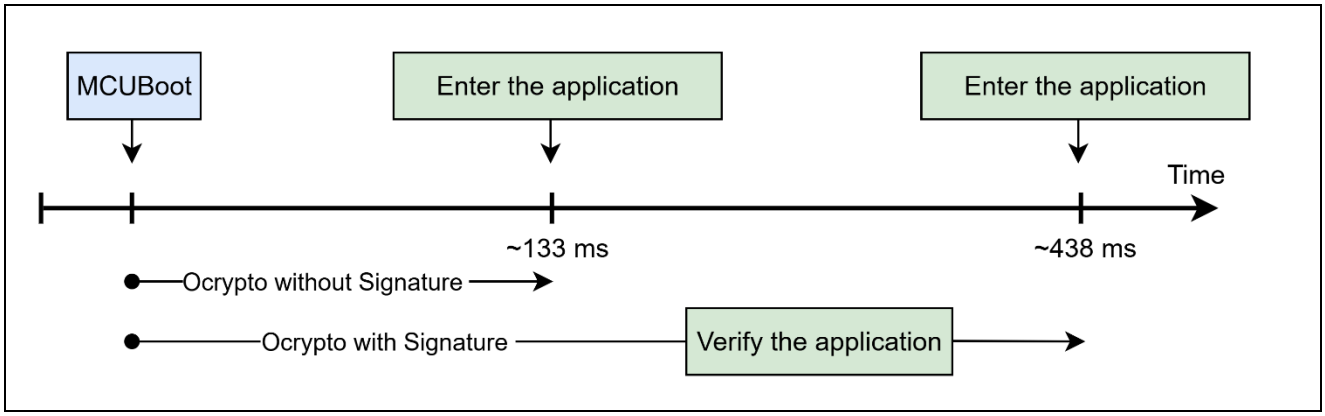

Figure 3. Boot Time with and without the Image Signing Option

2.4 Introduction to the Included Example Projects

Unzip `RA2_Secure_Bootloader_using_Ocrypto.zip` to unpack the example projects included in this application project.

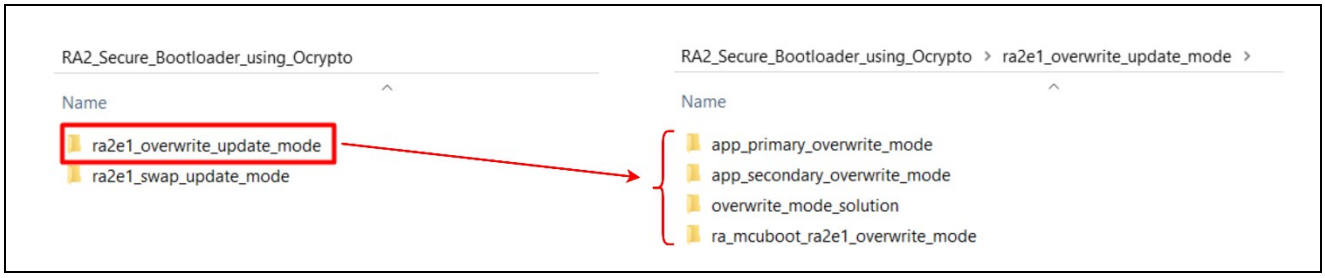

Figure 4. Example Projects Included

- `ra_mcuboot_ra2e1_overwrite_mode`: Bootloader Project, which enables Overwrite Upgrade Mode with signature verification.
- `app_primary_overwrite_mode`: Primary Application, which is configured to work with bootloader project, will blink three LEDs on EK-RA2E1 board when it boots successfully.
- `app_secondary_overwrite_mode`: Secondary Application, which implements the same functionality as `app_primary_overwrite_mode`, except that only the blue LED blinks.
- `overwrite_mode_solution`: Renesas Solution Project, which manages the MCUboot Memory Configuration for both the Bootloader Project and the Primary Application Project.

The descriptions above are also applicable to the `ra2e1_swap_update_mode` folder.

Refer to section 8 for instructions on quickly evaluating the projects.

3. Creating the Bootloader Project

This section guides you through the creation process of the RA2 bootloader provided in this application project.

The example bootloader that you will create by following this section is provided in the `RA2_Secure_Bootloader_using_Ocrypto.zip`. Users can follow section 8 to exercise the example bootloader and application projects without going through the creation process in this section.

3.1 Including the MCUboot Module in the Bootloader Project

1. Launch e² studio and start to establish a new C/C++ Project. Click **File > New > Renesas C/C++ Project > Renesas RA.**

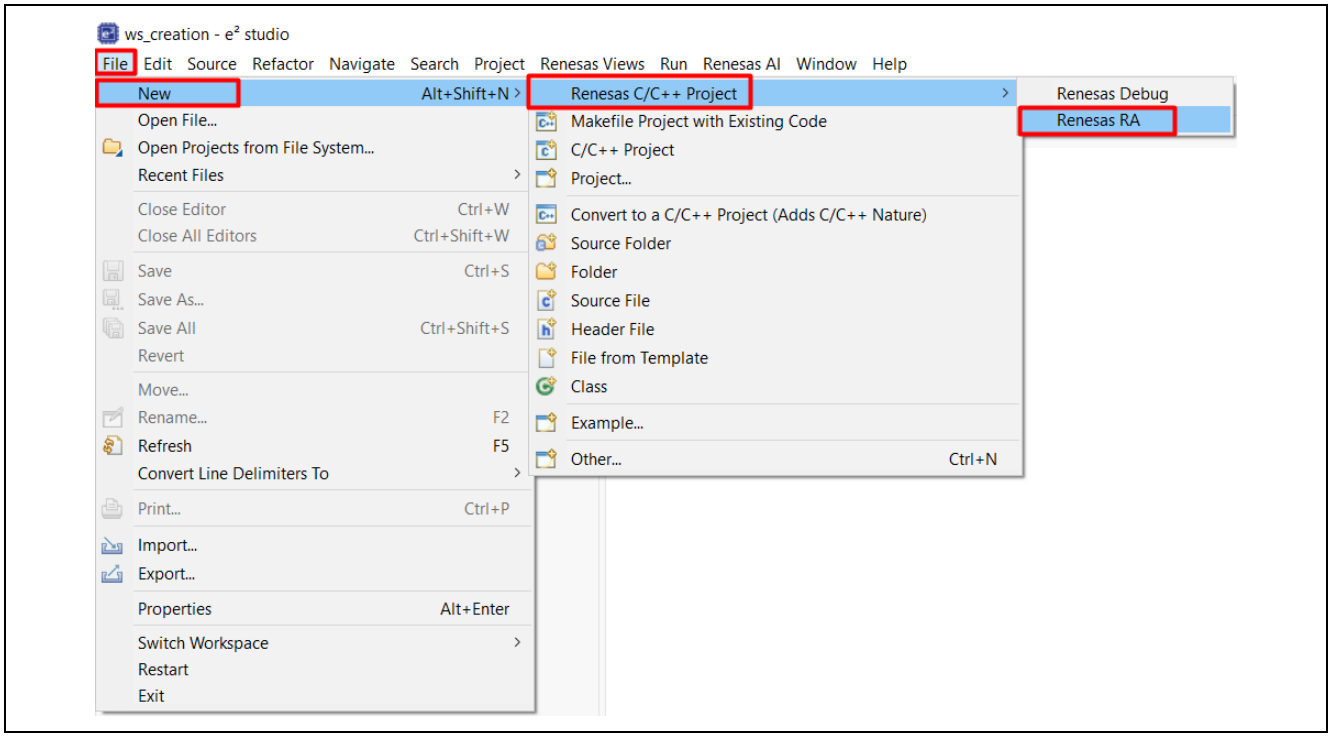

Figure 5. Start a New Project

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

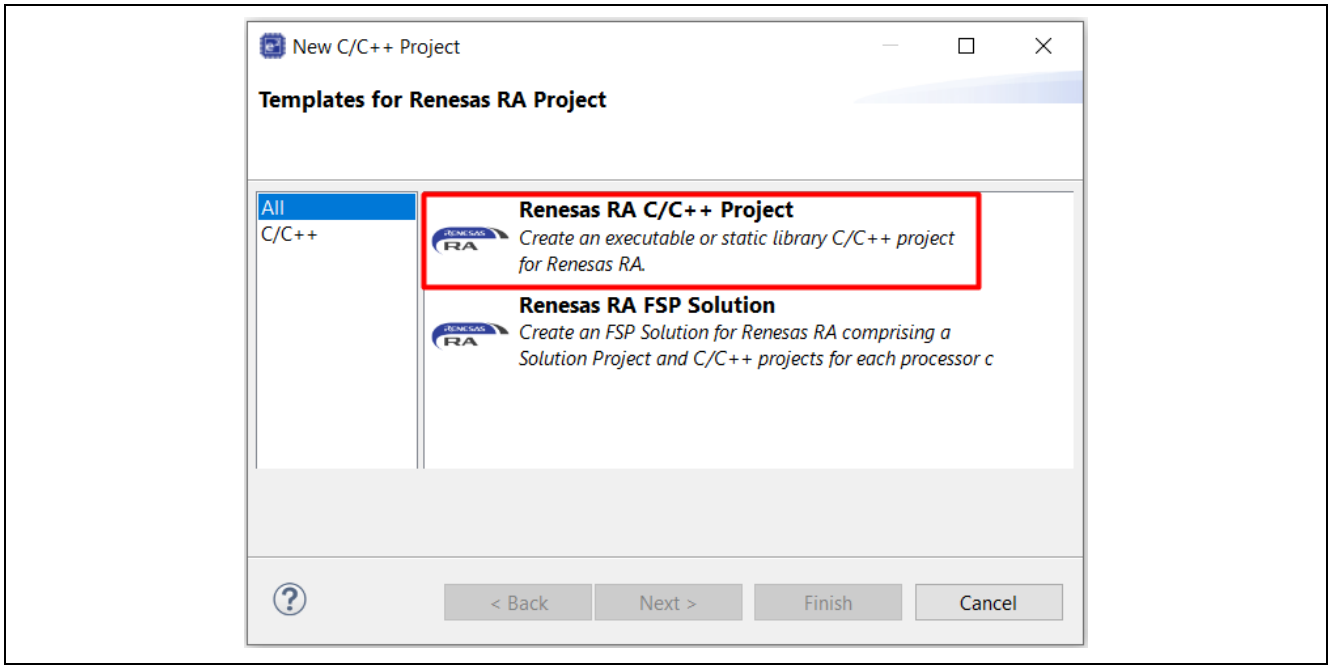

Figure 6. Choose Renesas RA C/C++ Project

3. Provide a project name in the next screen. Select a project name based on the upgrade mode and authentication method. The name will persist in the instructions used in this application note. Table 1 shows the name and intended application image update strategy of each bootloader project. Note that magic number and SHA256 integrity check are included in all of the systems.

Table 1. Description of the Bootloader Projects

Name of the project to be used	Intended application update strategy
`ra_mcuboot_ra2e1_overwrite_mode`	Overwrite update mode with signature verification.
`ra_mcuboot_ra2e1_swap_mode`	Swap update mode with signature verification.

Figure 7 is an example of setting the project name to **`ra_mcuboot_ra2e1_overwrite_mode`**.

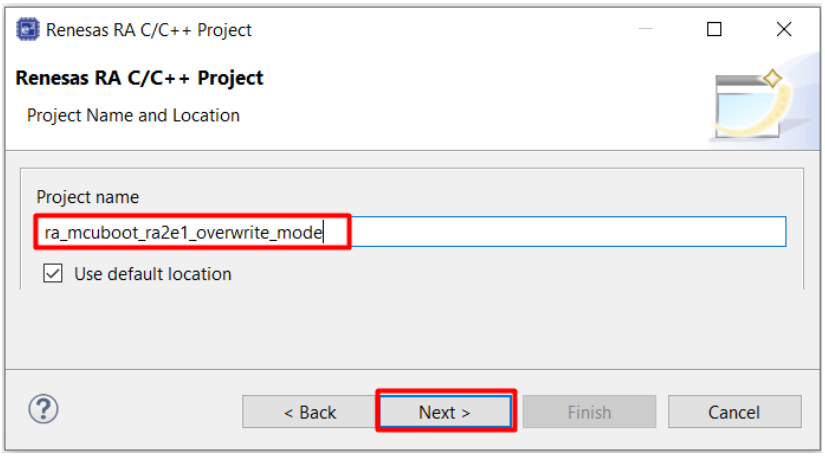

Figure 7. Name the Bootloader Project

Click **Next**. If users choose another name for the bootloader, adapt the corresponding instructions in this application note to the project name used.

4. In the next, select **FSP Version 6.1.0** and the **EK-RA2E1** for **Board** and click **Next**.

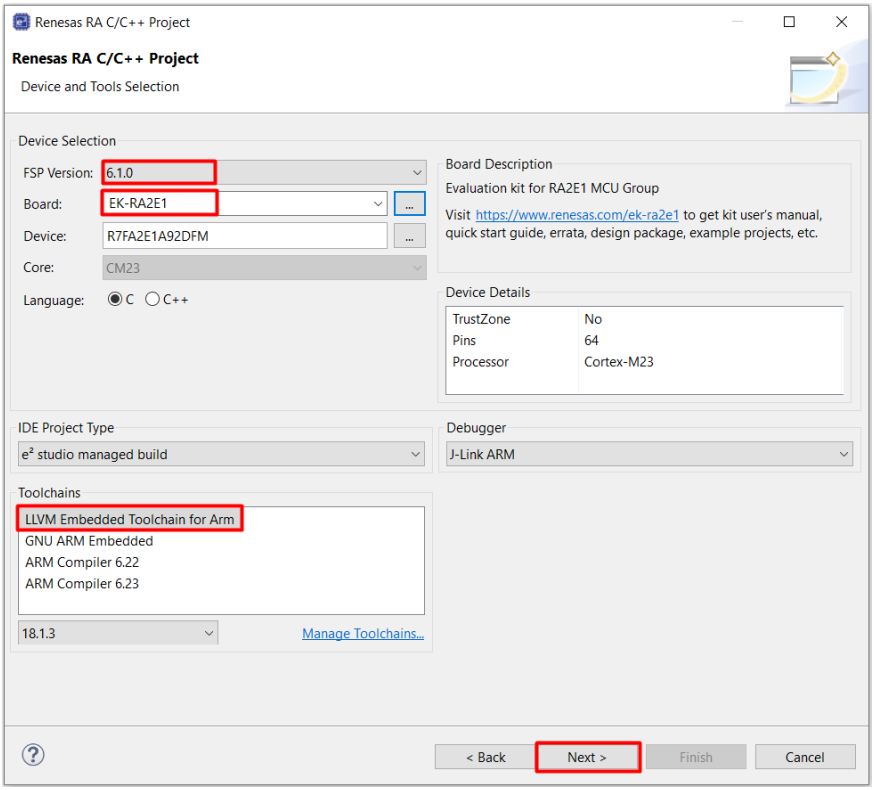

Figure 8. Select the FSP Version and Board

5. Select **None** for Preceding Project or Smart Bundle Selection screen and click **Next**.

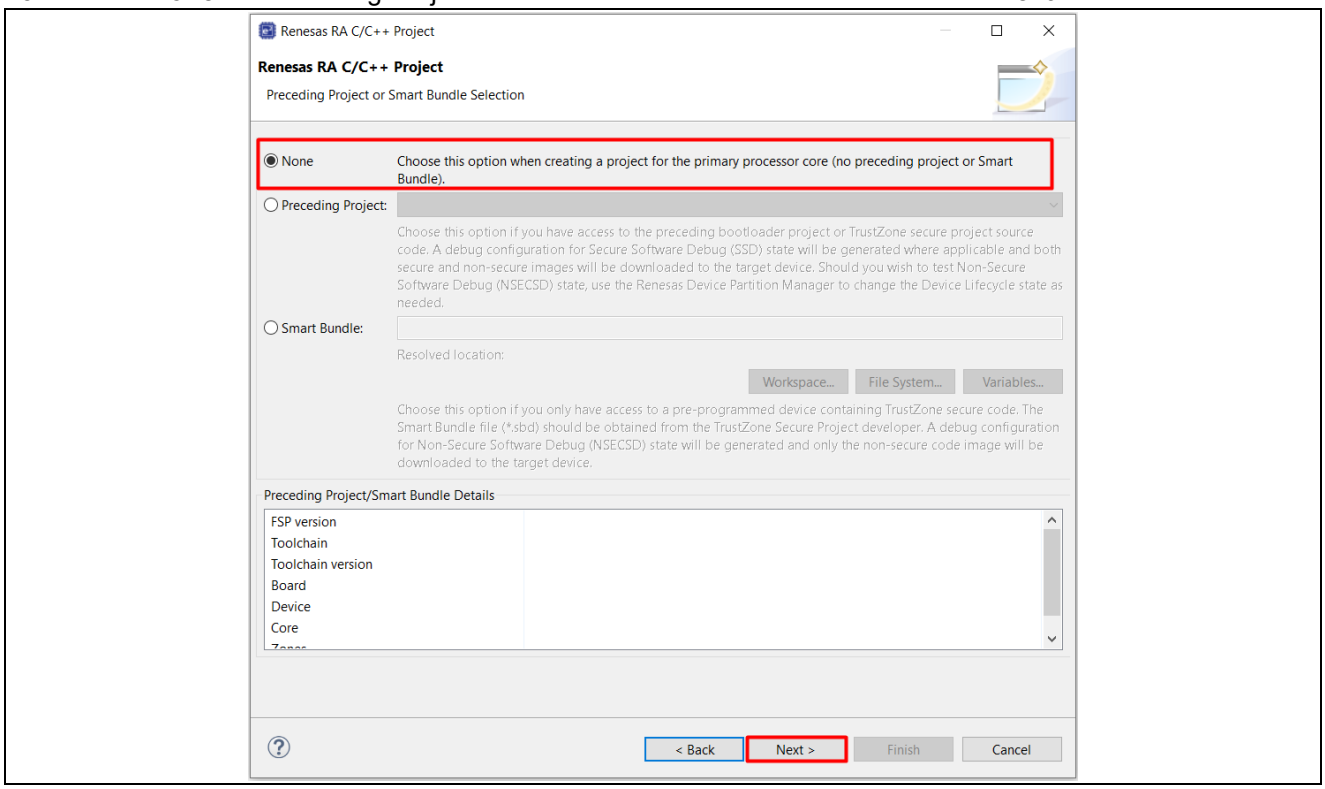

Figure 9. Select the None option for Preceding Project or Smart Bundle Selection screen

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

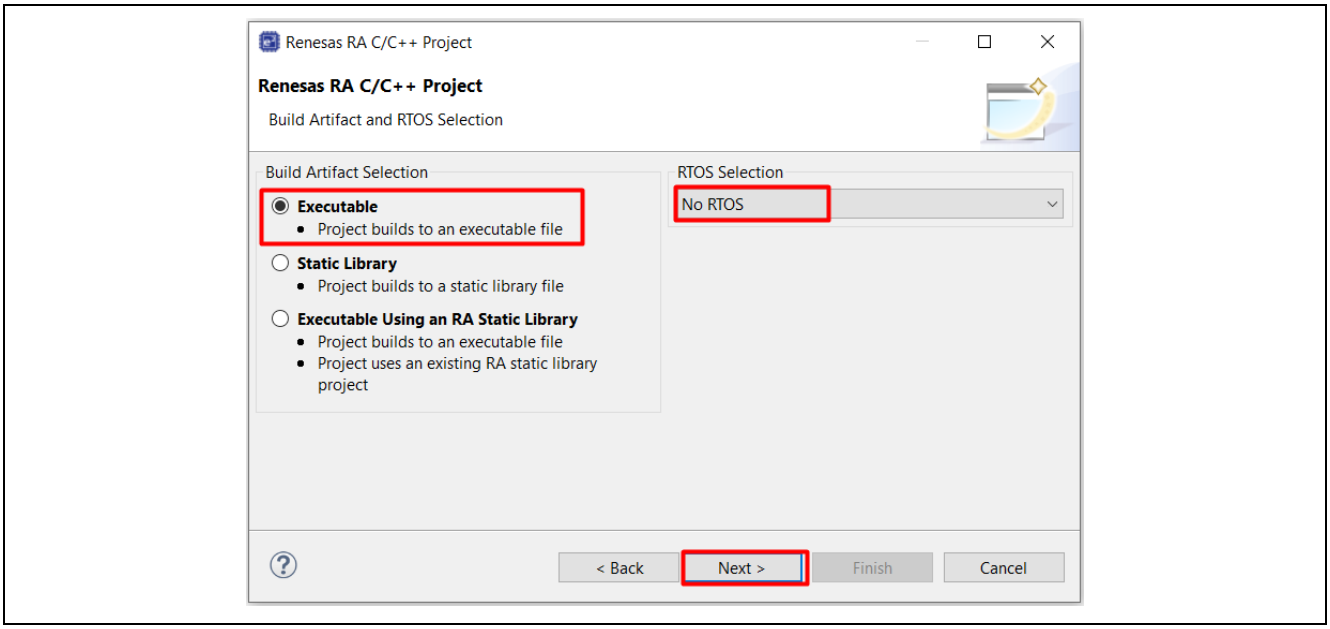

Figure 10. Choose to Build Executable and No RTOS

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

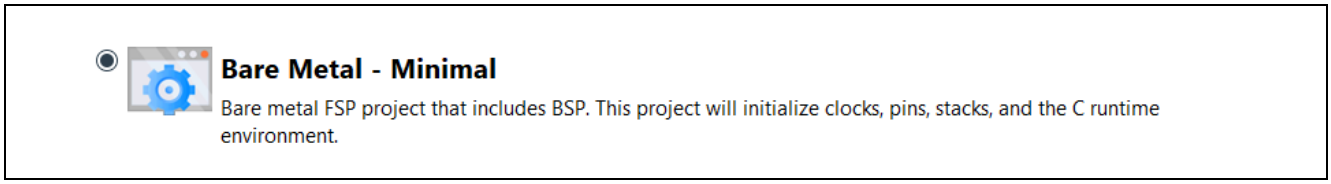

Figure 11. Choose the Project Template

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

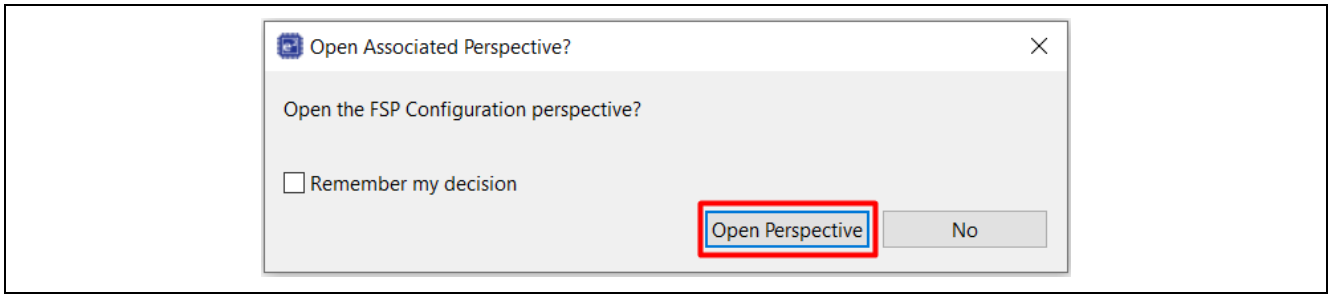

Figure 12. Choose Open the FSP Configuration Perspective

9. The project is now created, and the bootloader project configuration is displayed. Select the **Pins** tab and uncheck **Generate data** for **RA2E1 EK.**

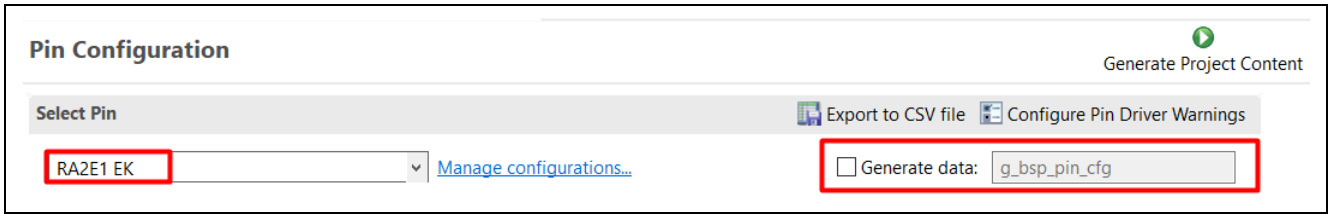

Figure 13. Uncheck Generate data for RA2E1 EK Pin Configuration

Use the pull-down menu to switch from **RA2E1 EK** to **R7FA2E1A92DFM.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 **RA2E1 EK** configuration. This change also reduces the bootloader memory usage and is highly recommended.

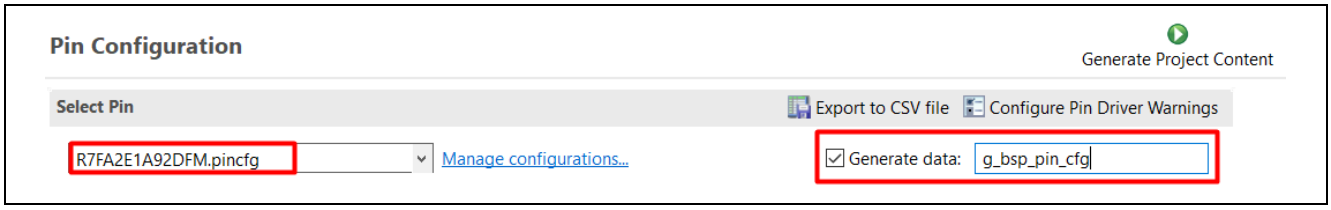

Figure 14. Select R7FA2E1A92DFM.pincfg and Generate data g_bsp_pin_cfg

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

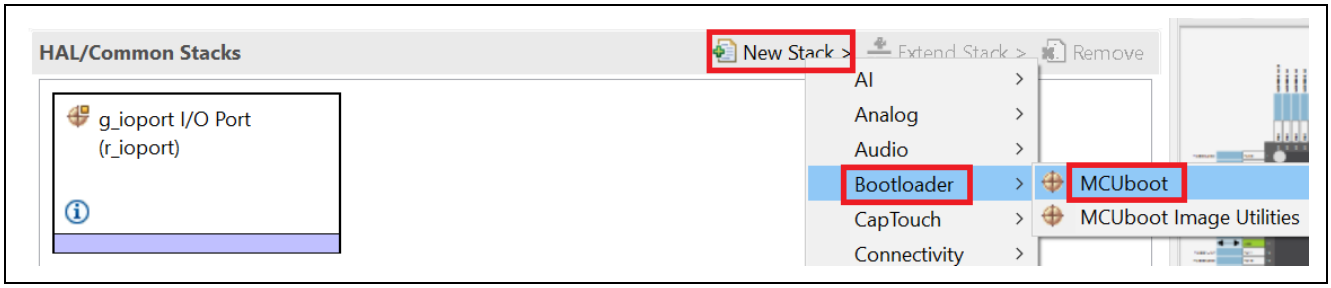

Figure 15. Add the MCUboot Port

11. Next, configure the **General** properties of **MCUboot**.
    - For project **`ra_mcuboot_ra2e1_overwrite_mode`**, use the settings in Figure 16.
    - For project **`ra_mcuboot_ra2e1_swap_mode`** update the following properties in Figure 17:
        - Change the **Upgrade Mode** to **Swap**.
        - Set the **Downgrade Prevention (Overwrite Only)** to **Disabled**.

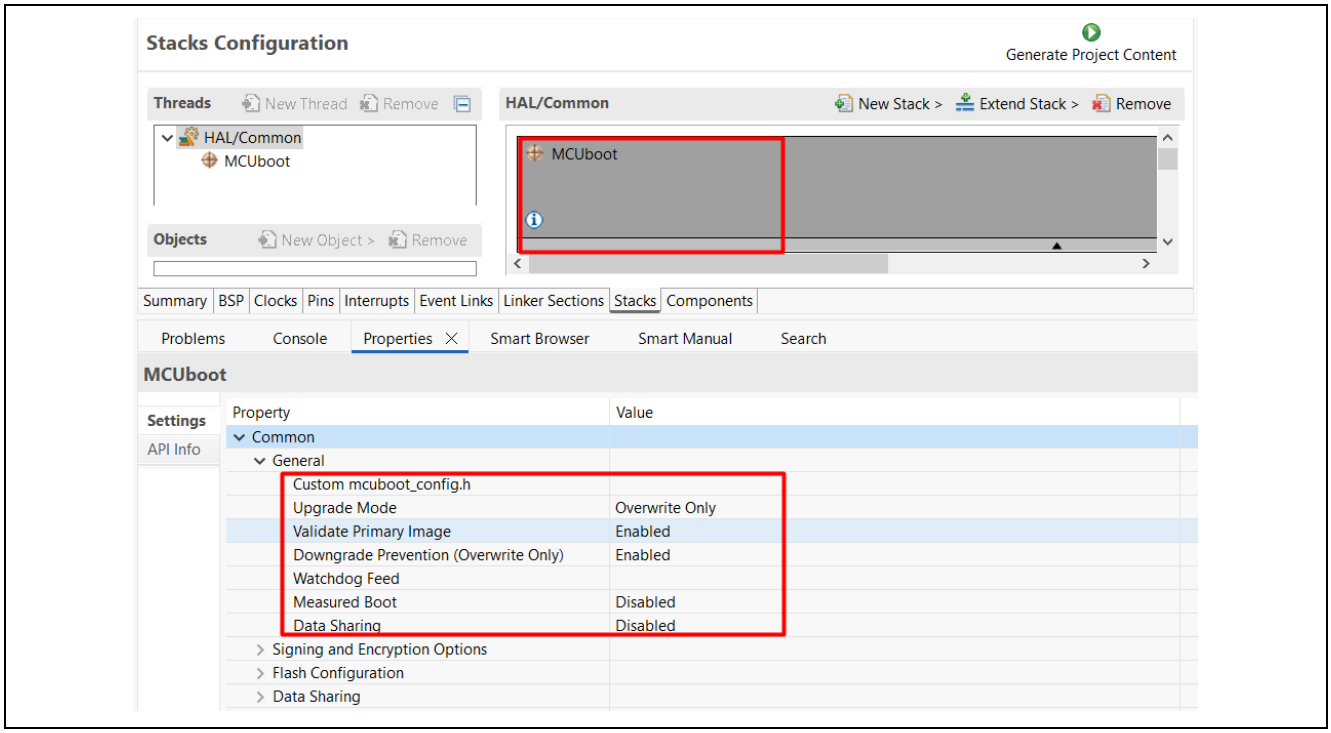

Figure 16. General Properties for MCUboot with Overwrite Upgrade Mode

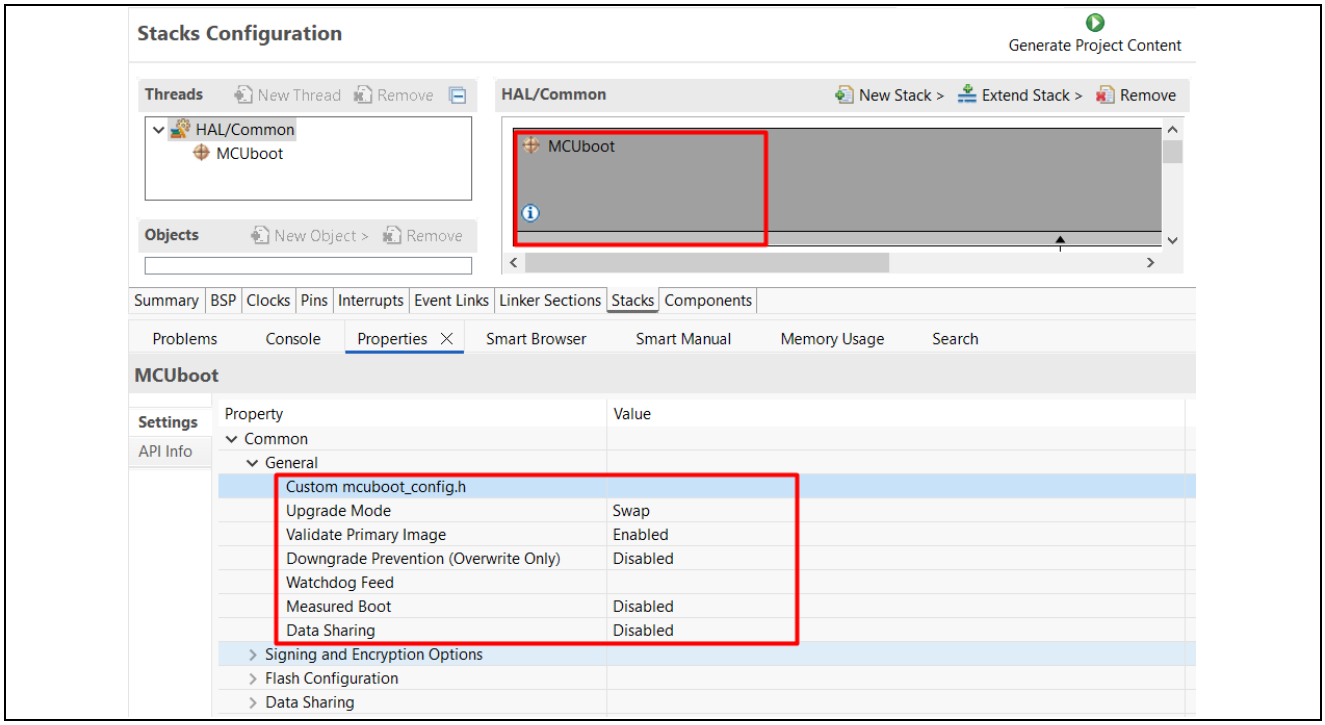

Figure 17. General Properties for MCUboot with Swap Upgrade Mode

Figure 18 is a more detailed application image format that can be referenced to understand the various MCUboot property definitions.

- The header magic number is used for image validation sanity check (refer to the description of **Validate Primary Image).**
- The **image_ok** byte is a flag used by the bootloader for swap test mode confirmation (refer to section 7.2 for more details).
- The trailer magic number is written after the image upgrade is finished.

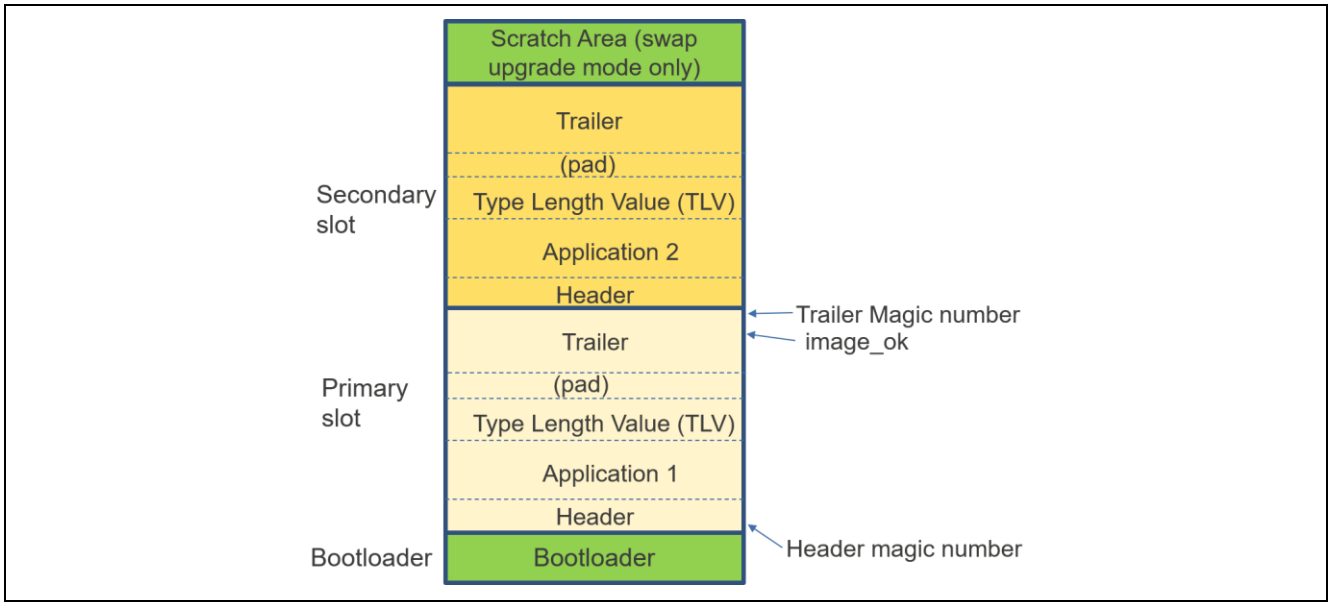

Figure 18. General Configuration for MCUboot Module

The properties configured include:

- **Custom mcuboot_config.h**: The default `mcuboot_config.h` file contains the MCUboot Module configuration that you select from the RA configurator. You 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.
- **Validate Primary Image**:
  When Enabled
  - The bootloader performs:
    - Hash or signature verification (depending on the selected verification method).
    - MCUboot sanity check (based on the image header magic number).
  - The header magic number is always checked as part of the sanity checking prior to the integrity checking and the signature verification.

  When Disabled
  - Only sanity check is performed based on the MCUboot header magic numbers.

  Note
  - It is **strongly recommended** to enable this property, as it adds critical security handling to the bootloader while requiring less than 32 bytes of additional code.
  - The image magic number is not part of the image validation.
  - It is a **reference value**, useful for sanity checks during the application upgrade debugging process.
  - The image magic number is written to flash only after a successful image upgrade.
- **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. To see how to set the version number of an image, refer to Figure 50.

12. Configure the Signing and Encryption Options

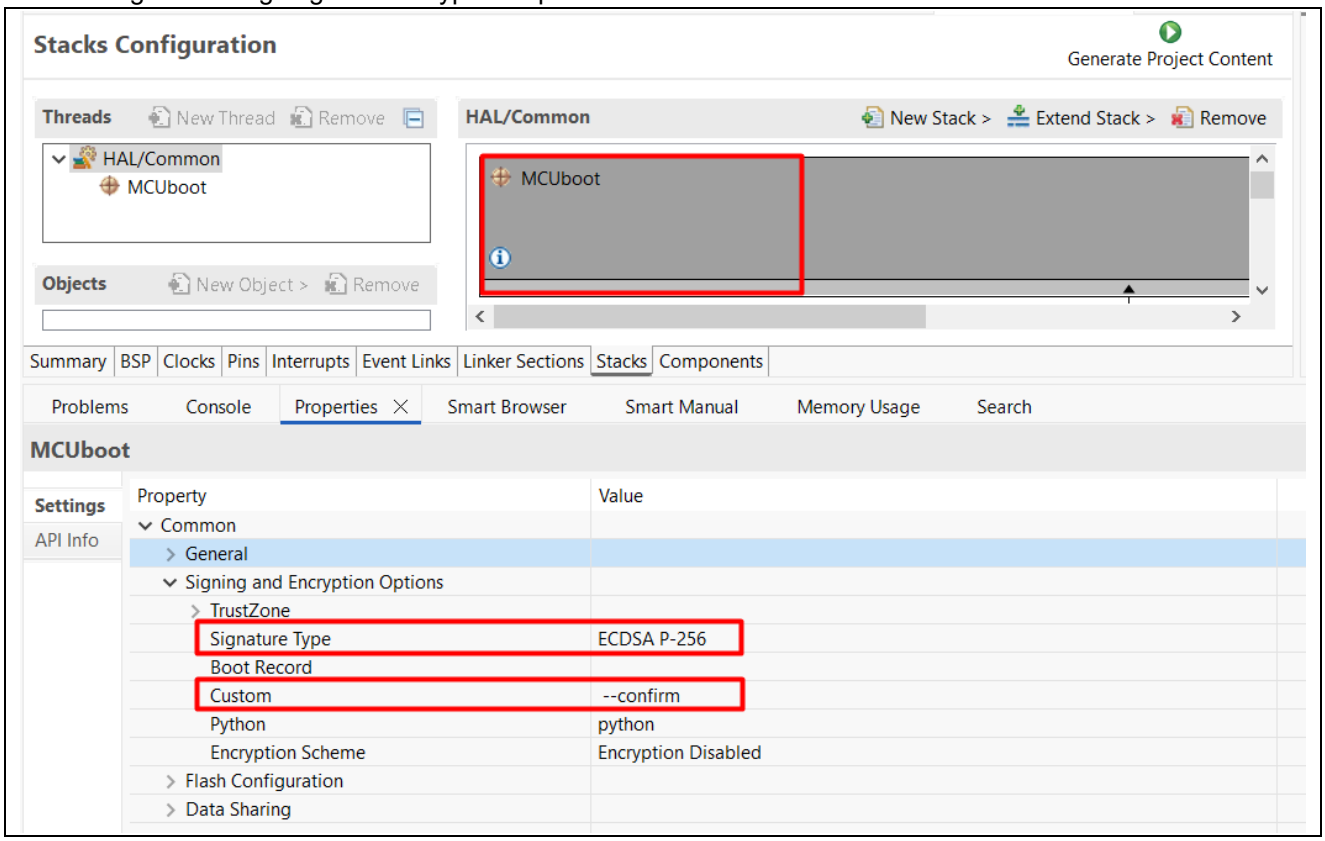

Figure 19. Application Image Signature Type and Signature Options

Explanation of the Above Configurations

For both single-image and two-image configurations, the following properties need to be defined:

- **Signature Type** is the signing algorithm selection. Application images using MCUboot must be signed to work with MCUboot. At a minimum, this involves adding a hash and an MCUboot-specific constant value in the image trailer.
  Note that when using Ocrypto as the cryptographic support for MCUboot, RSA signature verification is not supported. The choices are:
  - **NONE:** This option is selected for the bootloaders that do not support signature verification
  - **ECDSA P-256**: This option is selected for the example bootloaders that support signature verification included in this application project.
  - **RSA 2048 and RSA 3072:** Not supported.
- **Custom:** This property allows you to input any specific arguments for the signing command. By default `--confirm` is set for this property, which has the following influence on the Secondary image:
  - For Overwrite upgrade mode, the new image will always overwrite the original application image upon successful verification.
  - For Swap upgrade mode, the Primary image slot will be marked as Confirmed after the swap update. No swap happens upon the next reset after the swap update.

  If the **Custom** property is set to `--pad`, the system behavior is:
  - For Overwrite upgrade mode, the system behavior is same as when `--confirm` is set.
  - For Swap upgrade mode, the system behavior depends on whether the application has routines to mark the Primary image slot as Confirmed. The details about the system behavior are explained in section 7.2.2.

  The Primary image boot behavior is not influenced by the choice between `--confirm` or `--pad`.

Note that the **Signature Type** is set to **NONE**, the bootloader size decreases to approximately 8KB.

The properties under **TrustZone** are not used for RA2 MCUs since they do not have TrustZone. For other properties shown in this step, refer to the *FSP User's Manual* section on MCUboot port.

13. Next, add the **Ocrypto** module under **MCUboot Port for RA**. **Ocrypto (S/W Only)** is used. The **MbedTLS (Crypto Only)** and **TinyCrypt (S/W Only)** modules have a larger memory footprint compared with **Ocrypto** and are not used in this bootloader design.

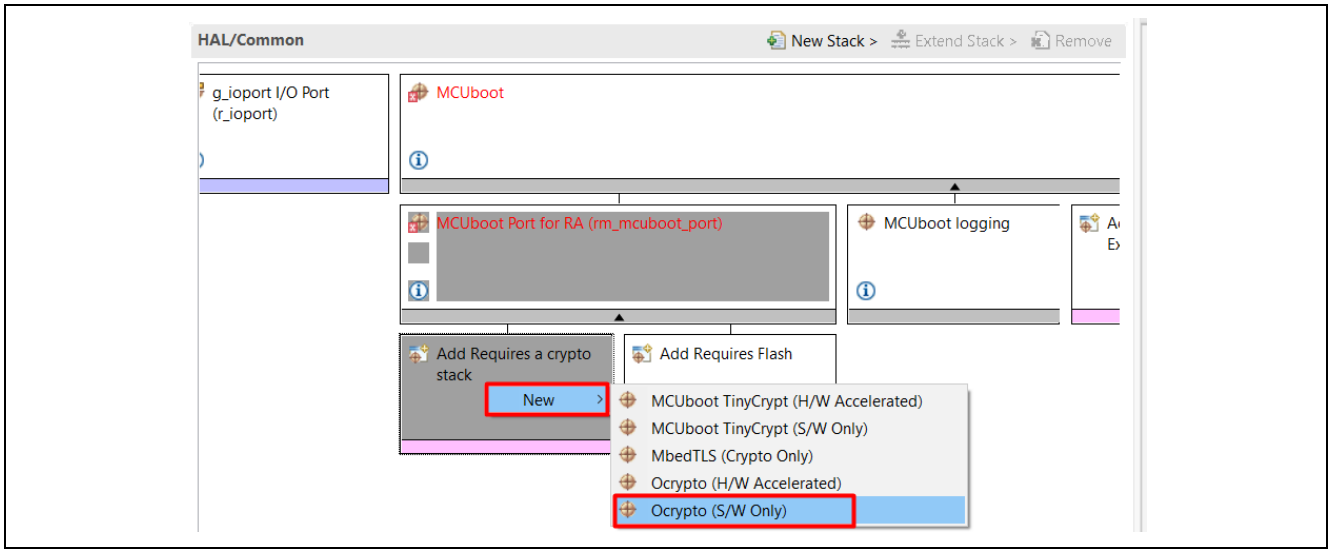

Figure 20. Select Ocrypto Module

14. 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. For example, if user wants to recreate the following example bootloaders, user needs to add the **ASN.1 Parser** stack and the **MCUboot Example Keys** stack.

Click on the **Add ASN.1 parser** stack and select **New** to add the ASN.1 Parser.

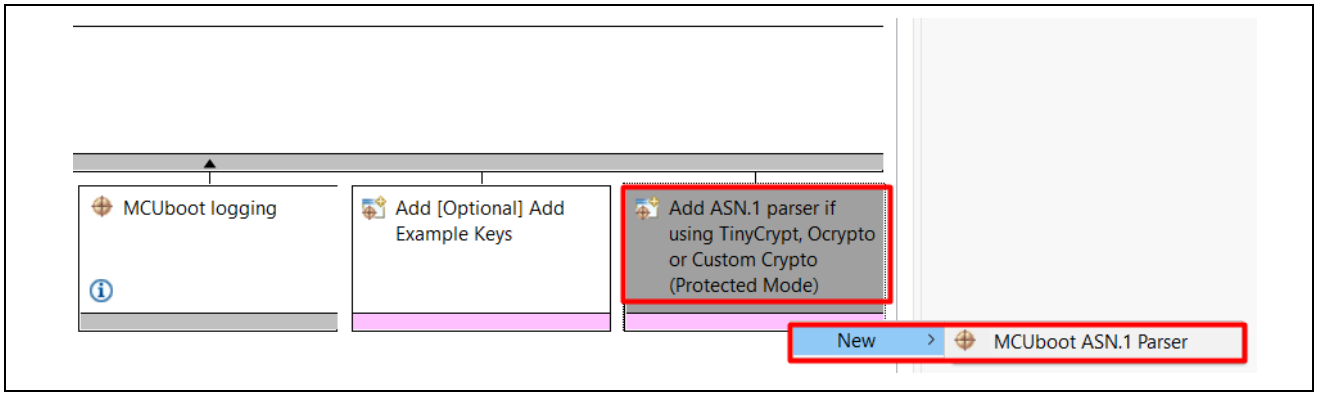

Figure 21. Add the ASN.1 Parser

Click on the **Add [Optional] Add Example Keys** stack and choose **New** to add **MCUboot Example Keys (NOT FOR PRODUCTION)**.

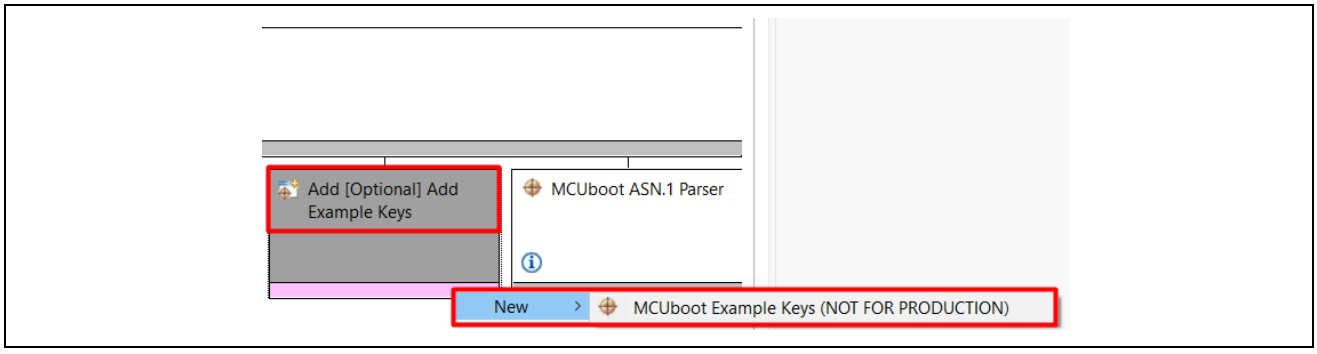

Figure 22. Add the Example Image Signing Key

Note that the example key is open to public access from MCUboot port, customers should not use them for production purposes. Customer can follow the procedure in section 9.1 to create and use a customized signing key.

15. Update the **BSP Main Stack size** to **0x800**.

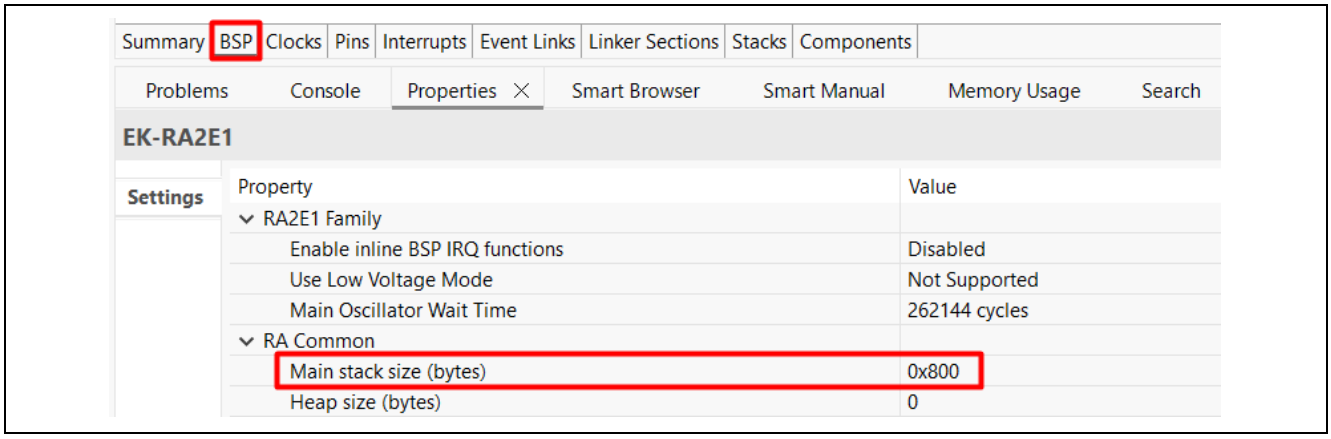

Figure 23. Update the BSP Main Stack Size

16. Click on **Add Required Flash** stack and add **Flash (r_flash_lp)**.
17. Click on the **Flash Driver** block and set the **Code Flash Programming** to **Enabled**. As **Data Flash Programming, Data Flash Background Operation Support,** and **Data Flash Background Operation**

are not used in the bootloader, select **Disabled** for these three properties to reduce the bootloader memory footprint.

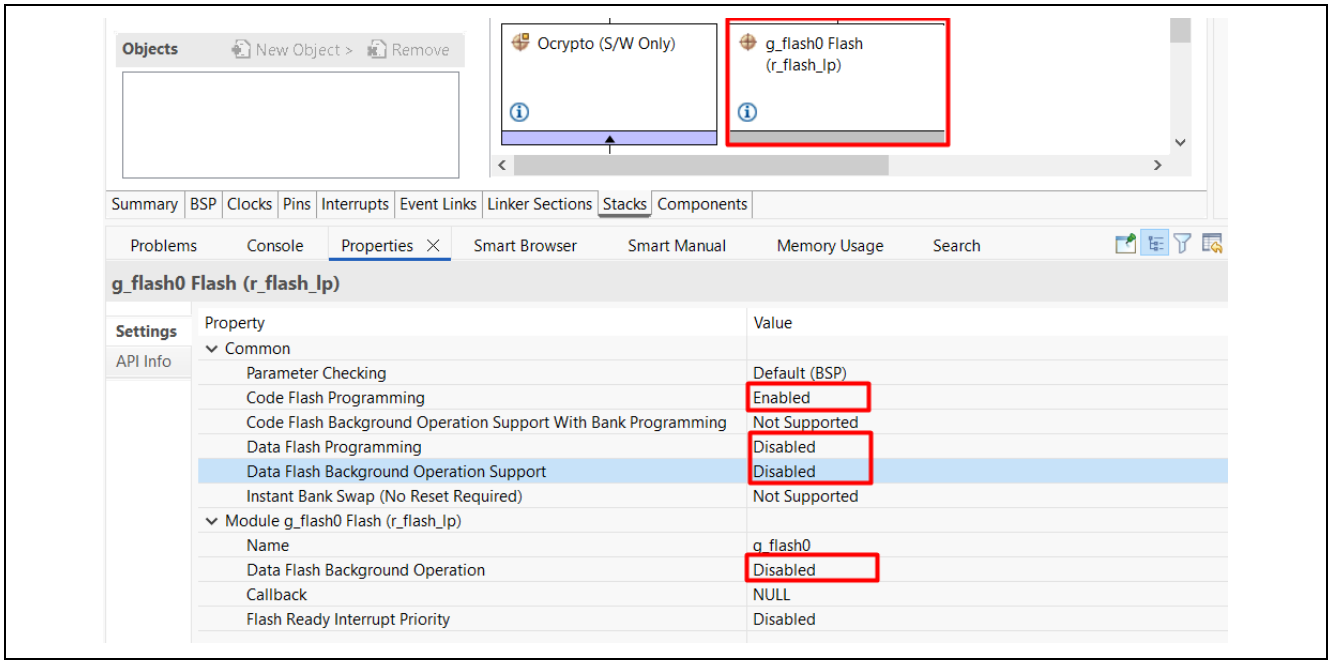

Figure 24. Enable Code Flash programming

18. Enable the HOCO Oscillation. The HOCO is used to provide the system clock to the Application Image after successful verification by MCUboot. Navigate to **BSP** tab and select it as shown in Figure 25.

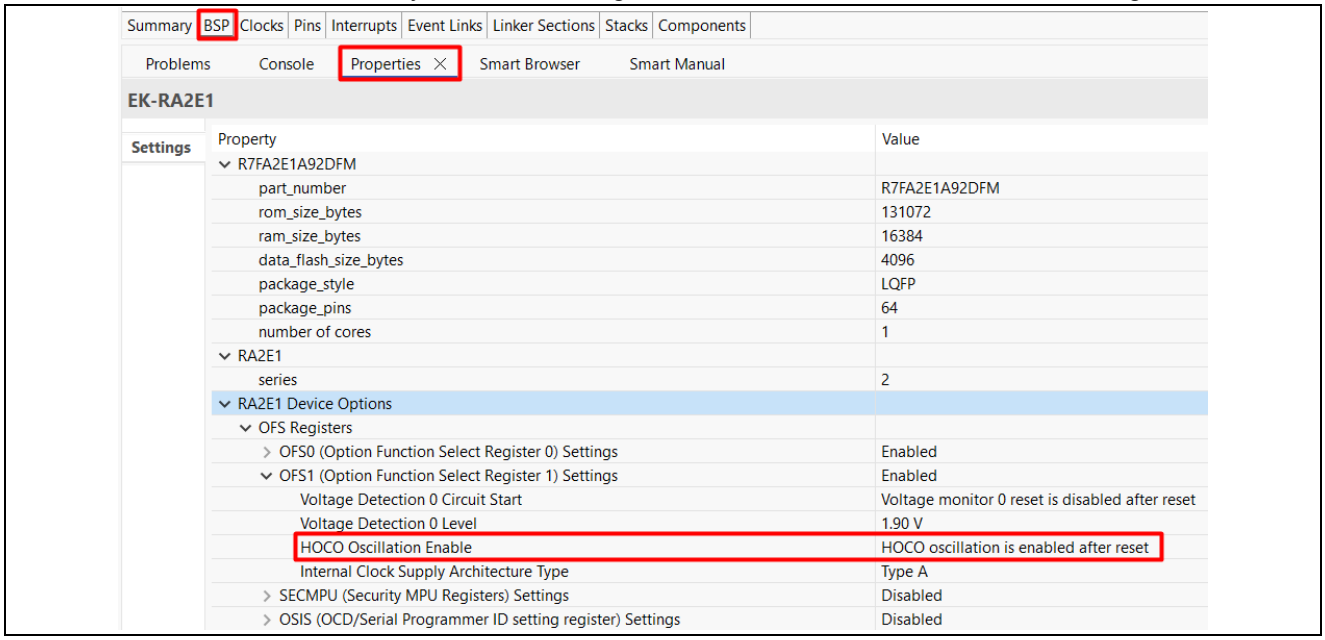

Figure 25. Enable the HOCO Oscillation

19. Save `configuration.xml` and click **Generate Project Content**. Next, 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();`

20. Notice that by default the **I/O Port** Driver is brought into the project when the project is established. Because the **I/O Port Driver** is not used in the bootloader project, this stack can be removed to reduce the bootloader project size. Right click on the **I/O Port** stack and choose **Delete**.

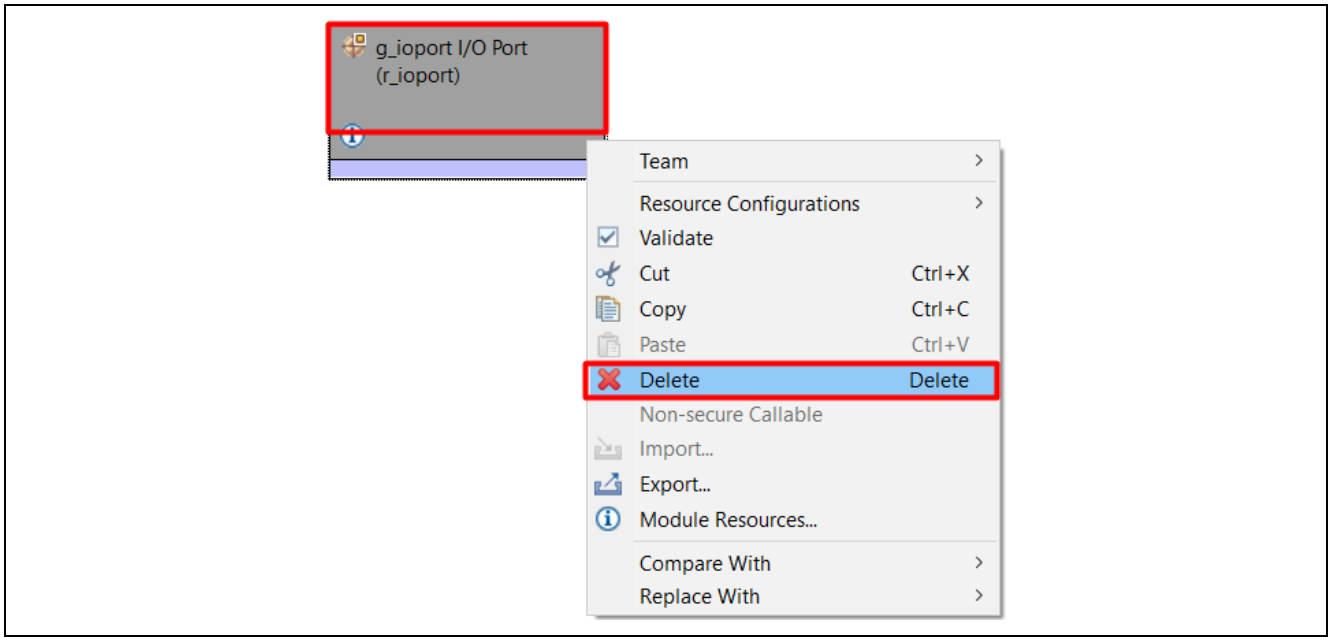

Figure 26. Remove the I/O Port Stack

After the I/O Port is deleted, remove all sections of code referencing the I/O Port API. For example, remove the two sections of the code in the red boxes in the function `R_BSP_WarmStart` in `hal_warmstart.c` file, as shown in Figure 27.

```
/*
* Copyright (c) 2020 - 2025 Renesas Electronics Corporation and/or its affiliates
*
* SPDX-License-Identifier: BSD-3-Clause
*/

#include "hal_data.h"

FSP_CPP_HEADER
void R_BSP_WarmStart(bsp_warm_start_event_t event);

FSP_CPP_FOOTER

 * This function is called at various points during the startup process.  This implementation uses the event that is
void R_BSP_WarmStart (bsp_warm_start_event_t event)
{
    if (BSP_WARM_START_RESET == event)
    {
#if BSP_FEATURE_FLASH_LP_VERSION != 0

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

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

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

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

#if BSP_CFG_SDRAM_ENABLED

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

Figure 27. Remove Unused Code in hal_warmstart.c

3.2 Configure the Bootloader for Encryption Support

In this section, we will add encryption to the application image, specifically to the secondary image. This feature is intended for use with external storage, particularly in cases involving large image sizes.

For the EK-RA2E1 board, external storage (external flash) is not supported; therefore, we will demonstrate the encrypted image using internal code flash.

The system will go through these following stages, as shown in Figure 28.

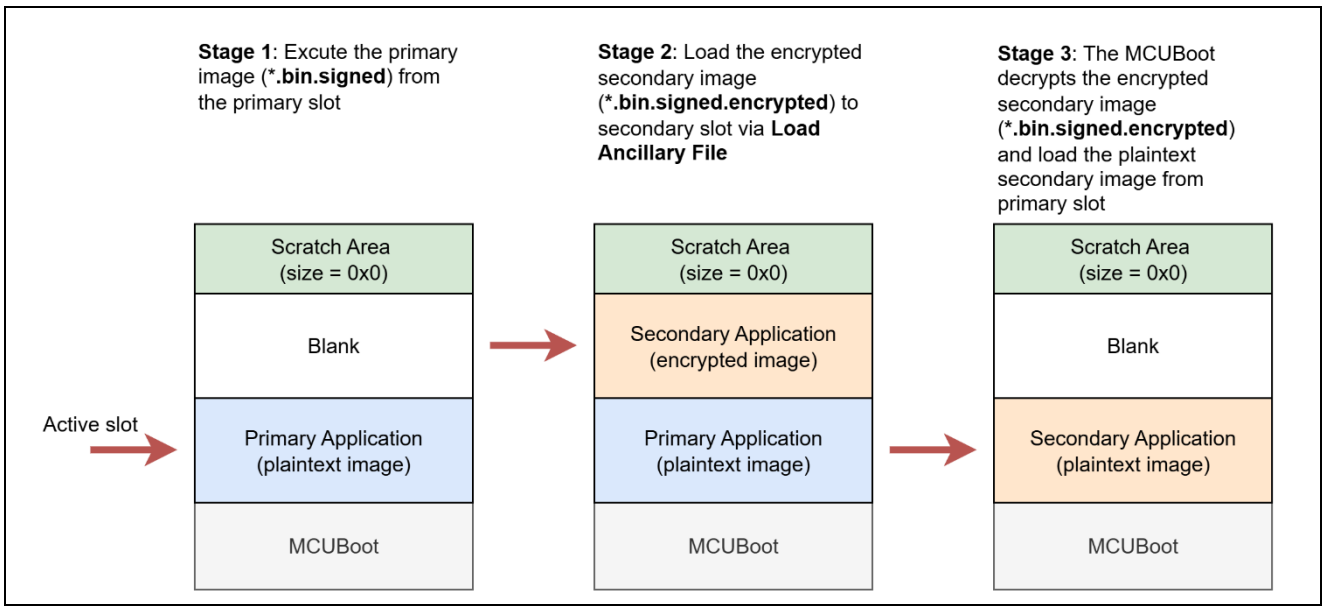

Figure 28. Booting Encrypted Image (Secondary Image Stored in Code Flash) in Overwrite Mode

As described in section 2.1, image encryption requires the Crypto Stack to be configured as **Ocrypto (HW Accelerated)**. The **Ocrypto (S/W Only)** configuration set in section 3.1 must first be removed, and then **Ocrypto (HW Accelerated)** should be added under **MCUboot Port for RA**.

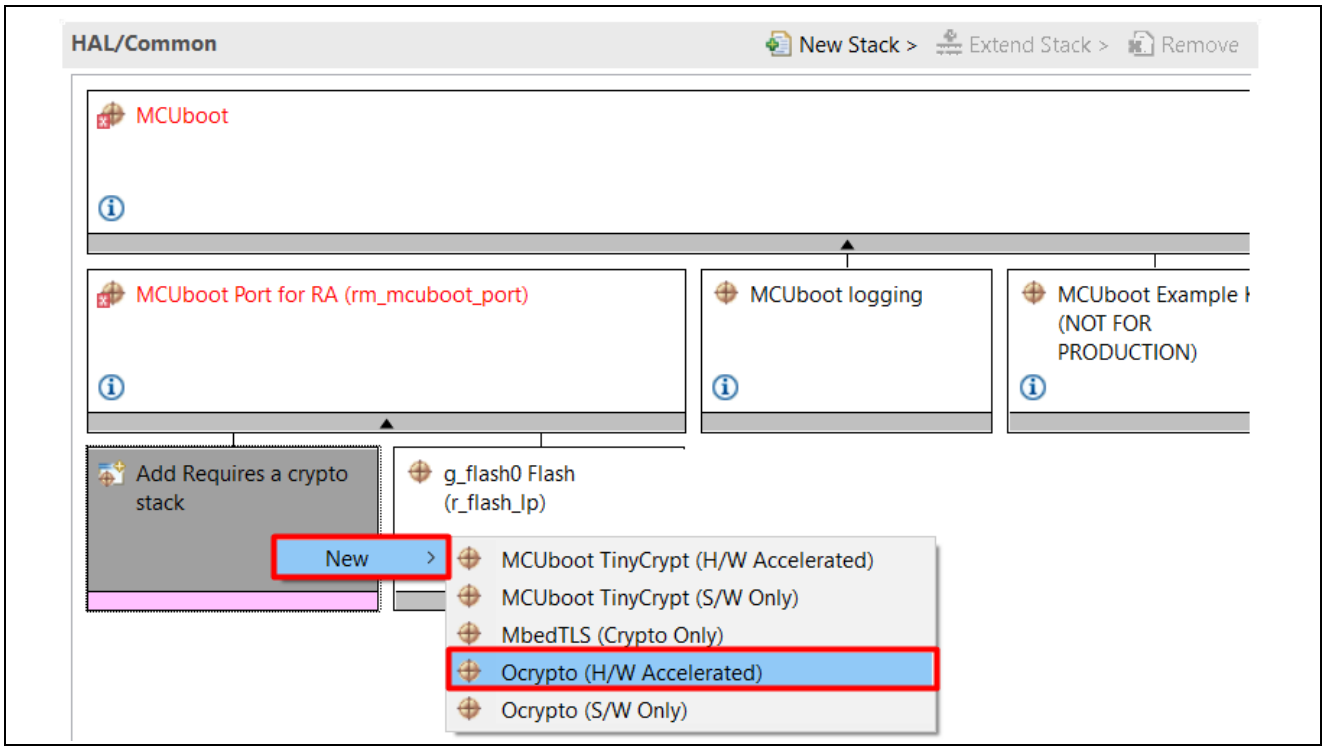

Figure 29. Select Ocrypto (HW Accelerated) Module

Navigate to the **Stacks** tab, select **MCUboot > Settings > Property > Common > Signing and Encryption Options > Encryption Scheme > ECIES-P256**

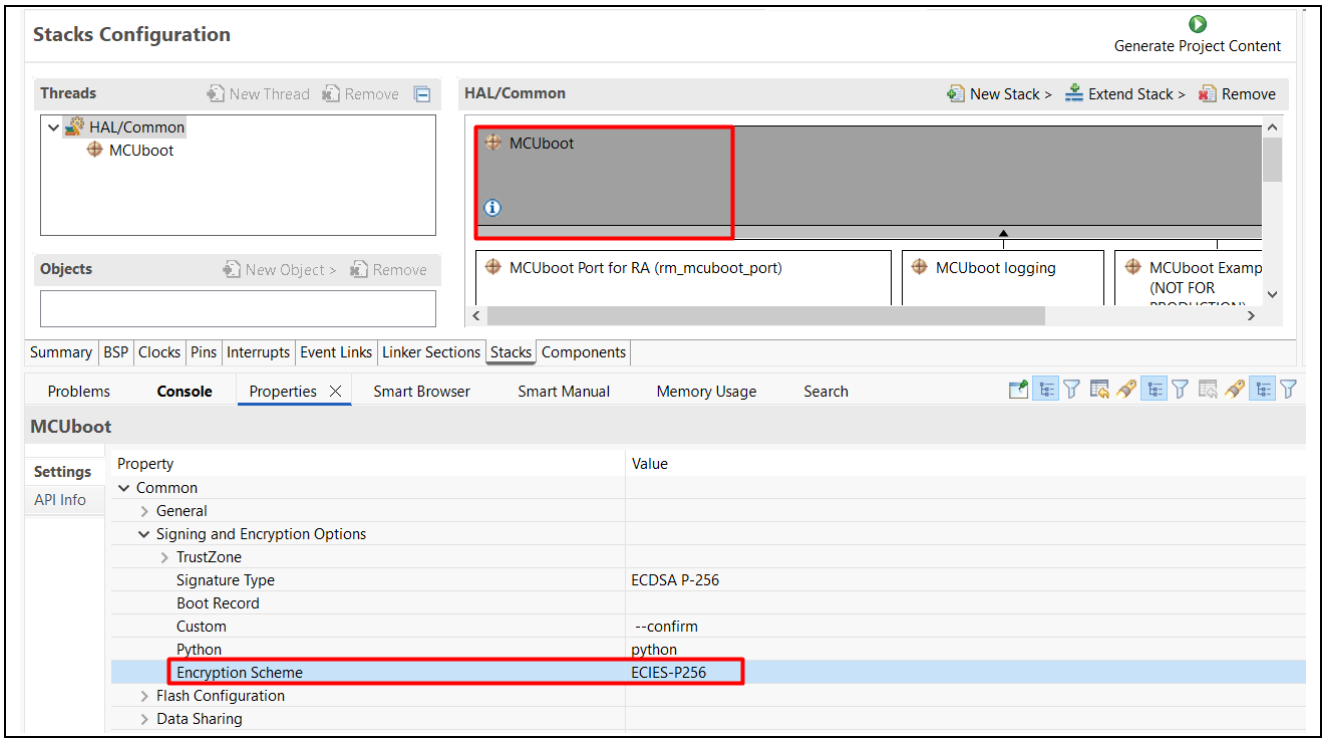

Figure 30. Choose Encryption Scheme: ECIES-P256

Note that when image encryption is enabled and **Ocrypto (HW Accelerated)** is selected, the bootloader image size increases to around 18 KB. Since the code flash boundary on the RA2E1 MCU is 2 KB (the largest erase size), the bootloader image is allocated 20 KB.

3.3 Further Optimizing for the Bootloader Project Size

To further optimize the bootloader project for size, users can put some application code in the gap area between the interrupt vector and the RA2E1 ROM registers (Option-Setting Memory Registers). We can use a section (`.flash_gap`) in the linker script to store some application code in this section.

Note that the bootloader image size optimization methods introduced in this section apply to any application project, regardless of whether a bootloader is used. Users can use the methods described in this section to save code space for any RA2 application.

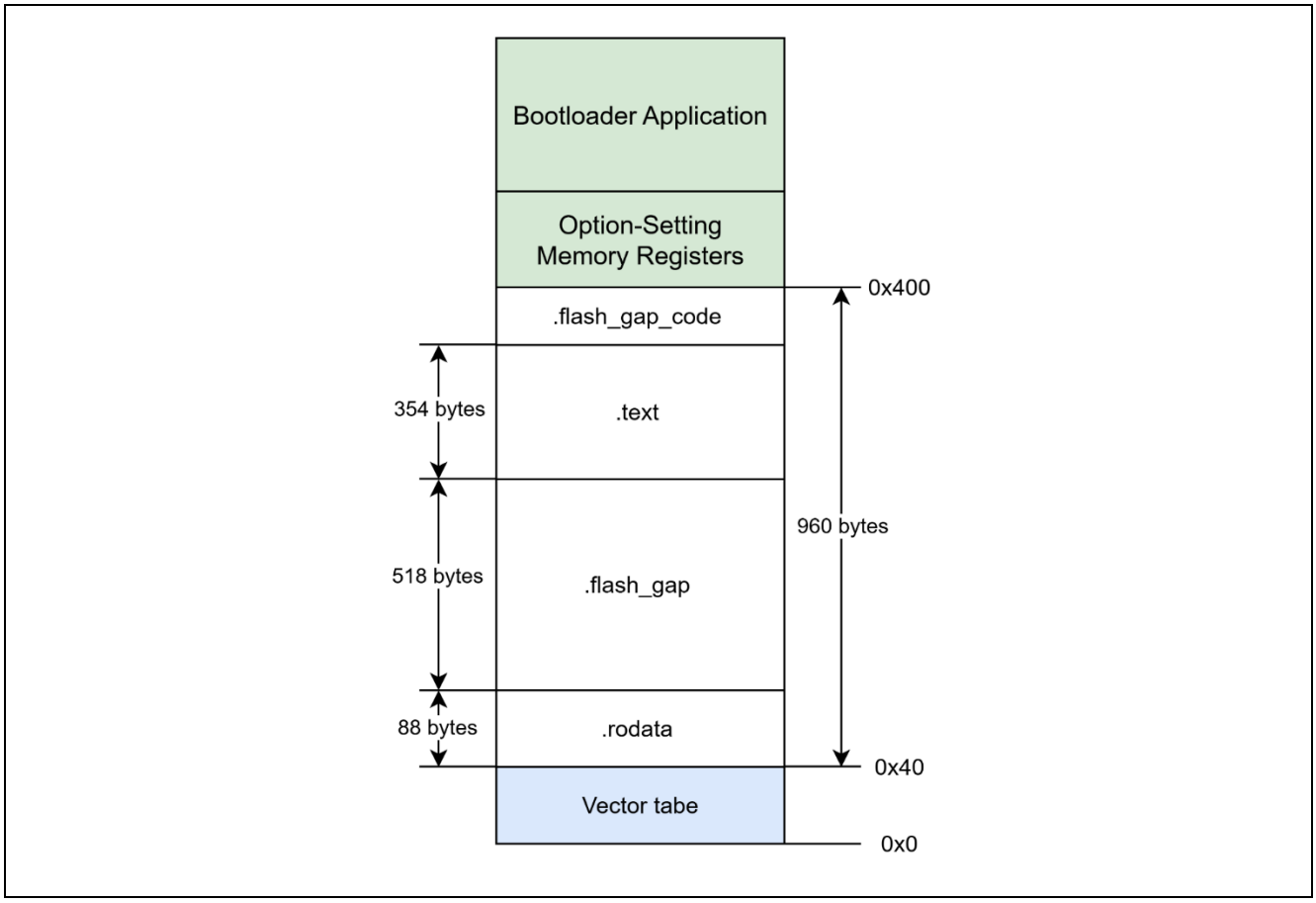

Figure 31. Example: First Flash Sector in Overwrite Update Mode

Next, users can choose some functions to put in the `.flash_gap` section to reduce the flash usage. Which functions are placed there is determined by the user.

In addition, users can also use the `.flash_gap_code` section for the same purpose. It is recommended that users choose only one section, either `.flash_gap` or `.flash_gap_code` to make management easier.

We also evaluated the size of `.flash_gap`, which is around 518 bytes. To make this evaluation, we use the **Memory Usage Window** in e^2 studio.

To launch the **Memory Usage Window**, users need to click **Window > Show View > Other… > C/C++ > Memory Usage,** as shown in Figure 32.

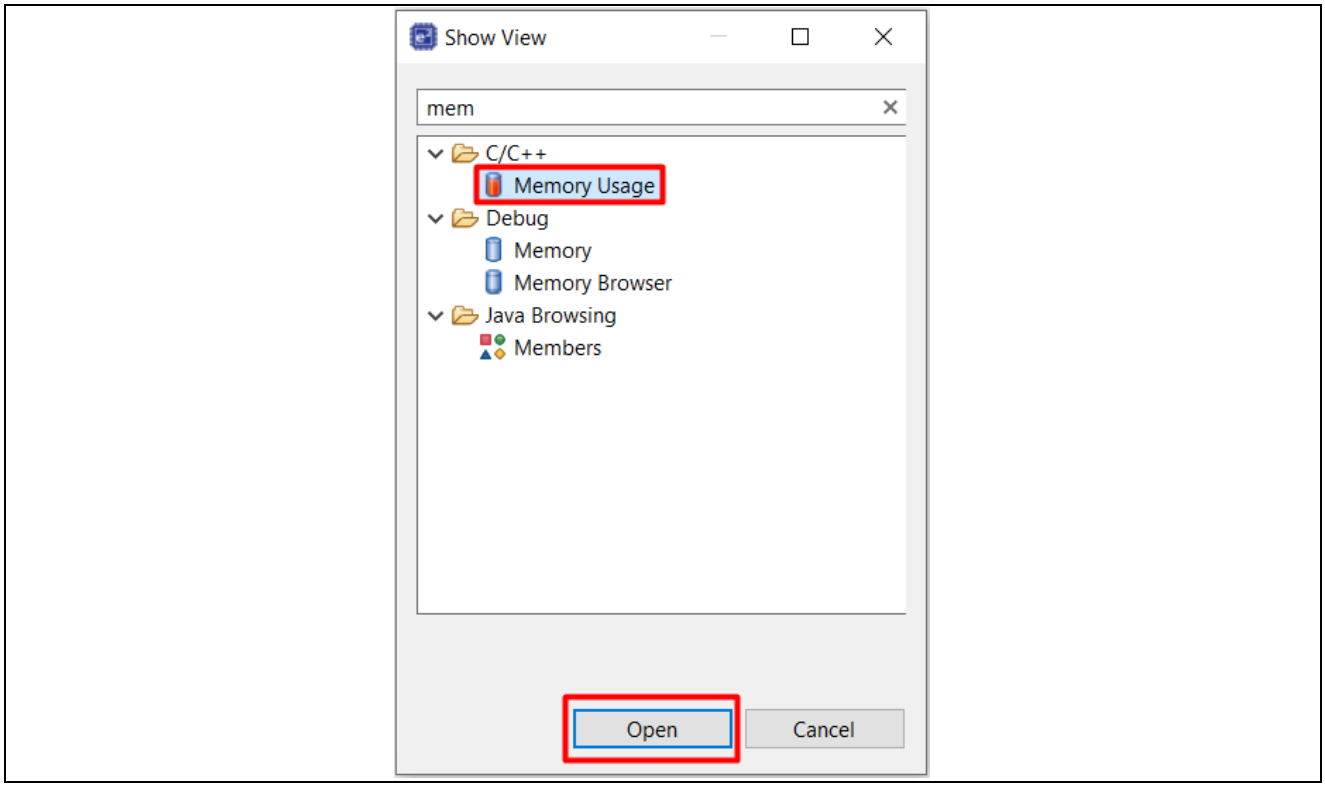

Figure 32. Open the Memory Usage

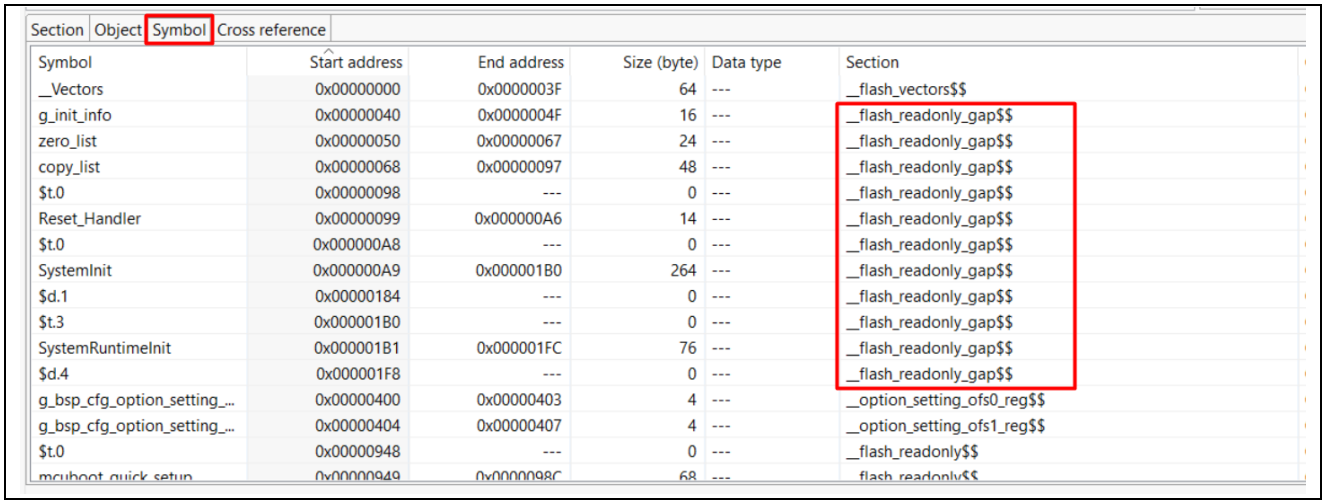

Figure 33. Memory Usage View

In addition, users can use the **Memory Usage** view to identify and select functions of suitable size to place in the `.flash_gap` section. Based on the **Memory Usage** results, we also placed some functions into the `.flash_gap` section, as shown in Figure 34 and Figure 35.

```
In \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\bootutil.h
int bootutil_tlv_iter_begin(struct image_tlv_iter *it,
                            const struct image_header *hdr,
                            const struct flash_area *fap, uint16_t type,
                            bool prot) BSP_PLACE_IN_SECTION(".flash_gap");

In \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\src\bootutil_priv.h
fih_ret bootutil_verify_sig(uint8_t *hash, uint32_t hlen, uint8_t *sig,
                            size_t slen, uint8_t key_id) BSP_PLACE_IN_SECTION(".flash_gap");

In \ra_MCUboot_ra2e1_overwrite_mode\src\hal_entry.c
void MCUboot_quick_setup() BSP_PLACE_IN_SECTION(".flash_gap");
```

Figure 34. Functions to put in the .flash_gap section in Overwrite Update Mode

```
In \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\bootutil_public.h
int boot_swap_type_multi(int image_index) BSP_PLACE_IN_SECTION(".flash_gap");

In \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\src\loader.c
static int boot_swap_image(struct boot_loader_state *state, struct boot_status *bs)
BSP_PLACE_IN_SECTION(".flash_gap"){}

In \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\src\bootutil_priv.h
int boot_write_trailer(const struct flash_area *fap, uint32_t off,
                       const uint8_t *inbuf, uint8_t inlen) BSP_PLACE_IN_SECTION(".flash_gap");
```

Figure 35. Functions to put in the .flash_gap section in Swap Update Mode

After placing functions into `.flash_gap` section, users can check whether functions were placed there using the **Memory Usage**.

Section | Object | Symbol | Cross reference

Symbol	Start address	End address	Size (byte)	Data type	Declaratio...	Section
__Vectors	0x00000000	0x0000003F	64	---	---	__flash_vectors$$
g_init_info	0x00000040	0x0000004F	16	---	---	__flash_readonly_gap$$
zero_list	0x00000050	0x00000067	24	---	---	__flash_readonly_gap$$
copy_list	0x00000068	0x00000097	48	---	---	__flash_readonly_gap$$
$t.0	0x00000098	---	0	---	---	__flash_readonly_gap$$
mcuboot_quick_setup	0x00000099	0x000000C4	44	---	---	__flash_readonly_gap$$
$d.1	0x000000B8	---	0	---	---	__flash_readonly_gap$$
$t.0	0x000000C4	---	0	---	---	__flash_readonly_gap$$
bootutil_verify_sig	0x000000C5	0x000001D0	268	---	---	__flash_readonly_gap$$
$d.1	0x000001BC	---	0	---	---	__flash_readonly_gap$$
$t.0	0x000001D0	---	0	---	---	__flash_readonly_gap$$
bootutil_tlv_iter_begin	0x000001D1	0x00000298	200	---	---	__flash_readonly_gap$$
$d.1	0x00000294	---	0	---	---	__flash_readonly_gap$$
$t.0	0x00000298	---	0	---	---	__flash_readonly_gap$$
Reset_Handler	0x00000299	0x000002A6	14	---	---	__flash_readonly_gap$$
$t.0	0x000002A8	---	0	---	---	__flash_readonly_gap$$
SystemInit	0x000002A9	0x000003B0	264	---	---	__flash_readonly_gap$$
$d.1	0x00000384	---	0	---	---	__flash_readonly_gap$$
$t.3	0x000003B0	---	0	---	---	__flash_readonly_gap$$
SystemRuntimeInit	0x000003B1	0x000003FC	76	---	---	__flash_readonly_gap$$
$d.4	0x000003F8	---	0	---	---	__flash_readonly_gap$$

Figure 36. Memory Usage View showing functions placed in the .flash_gap section

Furthermore, users can also observe that **Flash Usage** decreases, while **Flash Gap Usage** increases, as shown in Figure 37.

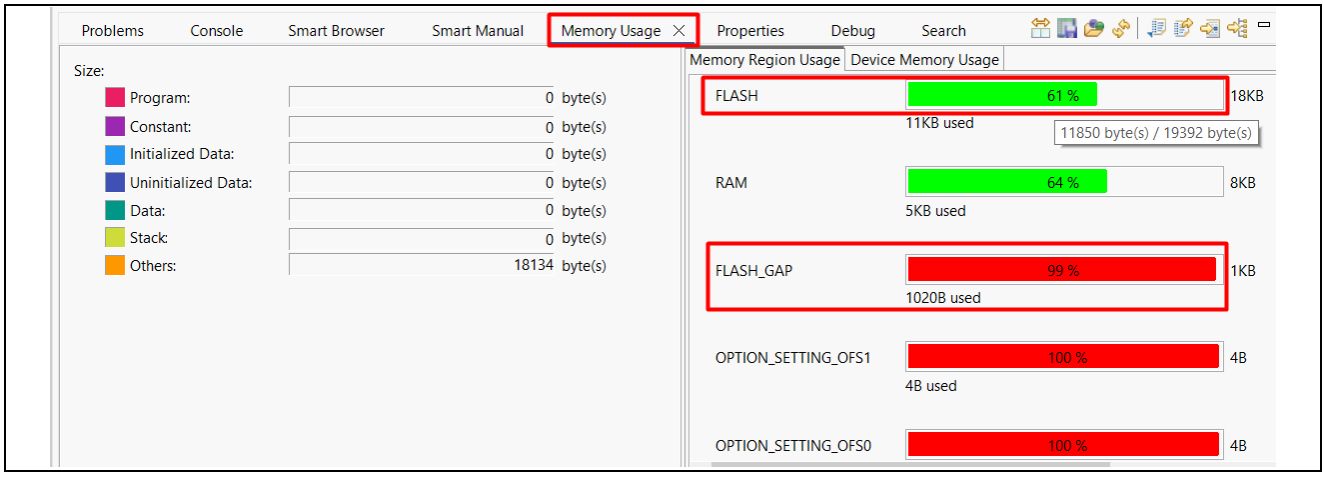

Figure 37. Effect of placing functions in the .flash_gap section on Flash and Flash Gap usage

3.4 Compiling the Bootloader Project

Depending on the upgrade mode selected, the size may differ slightly. In addition, migrating projects to a later FSP version may result in minor size variations.

```
llvm-size --format=berkeley "ra_mcuboot_ra2e1_overwrite_mode.elf"
   text    data     bss     dec     hex filename
  14094       0    4040   18134    46d6 ra_mcuboot_ra2e1_overwrite_mode.elf
llvm-objcopy "ra_mcuboot_ra2e1_overwrite_mode.elf" -O srec  "ra_mcuboot_ra2e1_overwrite_mode.srec"

13:27:03 Build Finished. 0 errors, 0 warnings. (took 3s.748ms)
```

Figure 38. Compilation Result in Overwrite Update Mode

```
llvm-size --format=berkeley "ra_mcuboot_ra2e1_swap_mode.elf"
   text    data     bss     dec     hex filename
  18122       0    4248   22370    5762 ra_mcuboot_ra2e1_swap_mode.elf
llvm-objcopy "ra_mcuboot_ra2e1_swap_mode.elf" -O srec  "ra_mcuboot_ra2e1_swap_mode.srec"

13:38:54 Build Finished. 0 errors, 0 warnings. (took 3s.245ms)
```

Figure 39. Compilation Result in Swap Update Mode

These outcomes are also related to the bootloader size configuration described in section 5.

3.5 Configuring 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 4.

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 `<bootloader_project>\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.

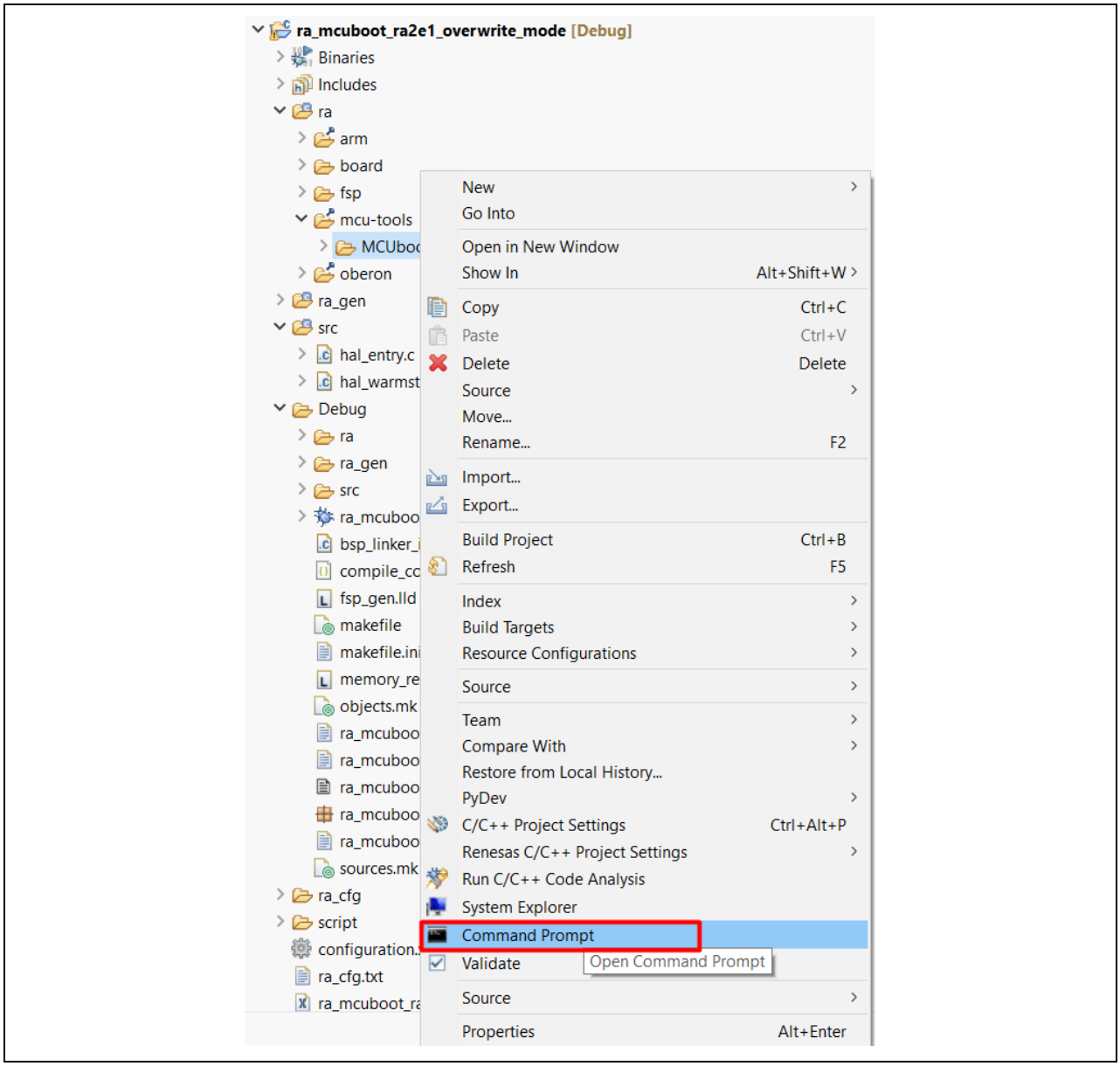

Figure 40. Open the Command Prompt

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. Using the Bootloader with a New Application or Existing Application

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 aspect is not demonstrated in this application project. Customers typically have customized image download method which differs from one customer to another.

This section uses a simple blinky project to demonstrate how to use the bootloader with the blinky application. After the initial blinky project is established, we need to configure the blinky project to the use of the bootloader project generated in the previous section. We also need to sign the blinky project using the signing command generated in the bootloader project. Detailed instructions are provided in this section.

Note: Users can also follow section 8 to exercise the example bootloader and application projects without going through the application creation and configuration process to use with the bootloader. This section provides references for users to understand how to customize the project for their specific application.

4.1 Generate the Initial Application Project

Follow the steps below to create a blinky application project as the Initial Application Project. The steps in section 4.1 are identical when generating a blinky project whether the application uses a bootloader or not. Launch e^2 studio and open a Workspace, click **File > New > Renesas C/C++ Project > Renesas RA**, and choose **Renesas RA C/C++ Project**. Click **Next**.

1. Assign the project name based on Table 2.

Table 2. Name the Initial Application Project

Bootloader project name	Initial application project name
`ra_mcuboot_ra2e1_overwrite_mode`	`app_primary_overwrite_mode`
`ra_mcuboot_ra2e1_swap_mode`	`app_primary_swap_mode`

2. Click **Next**, choose **EK-RA2E1** as the **Board** from the drop-down menu, and choose the **LLVM Embedded Toolchain**. Then click **Next**.
3. Linking the Application Project to Bootloader Project.

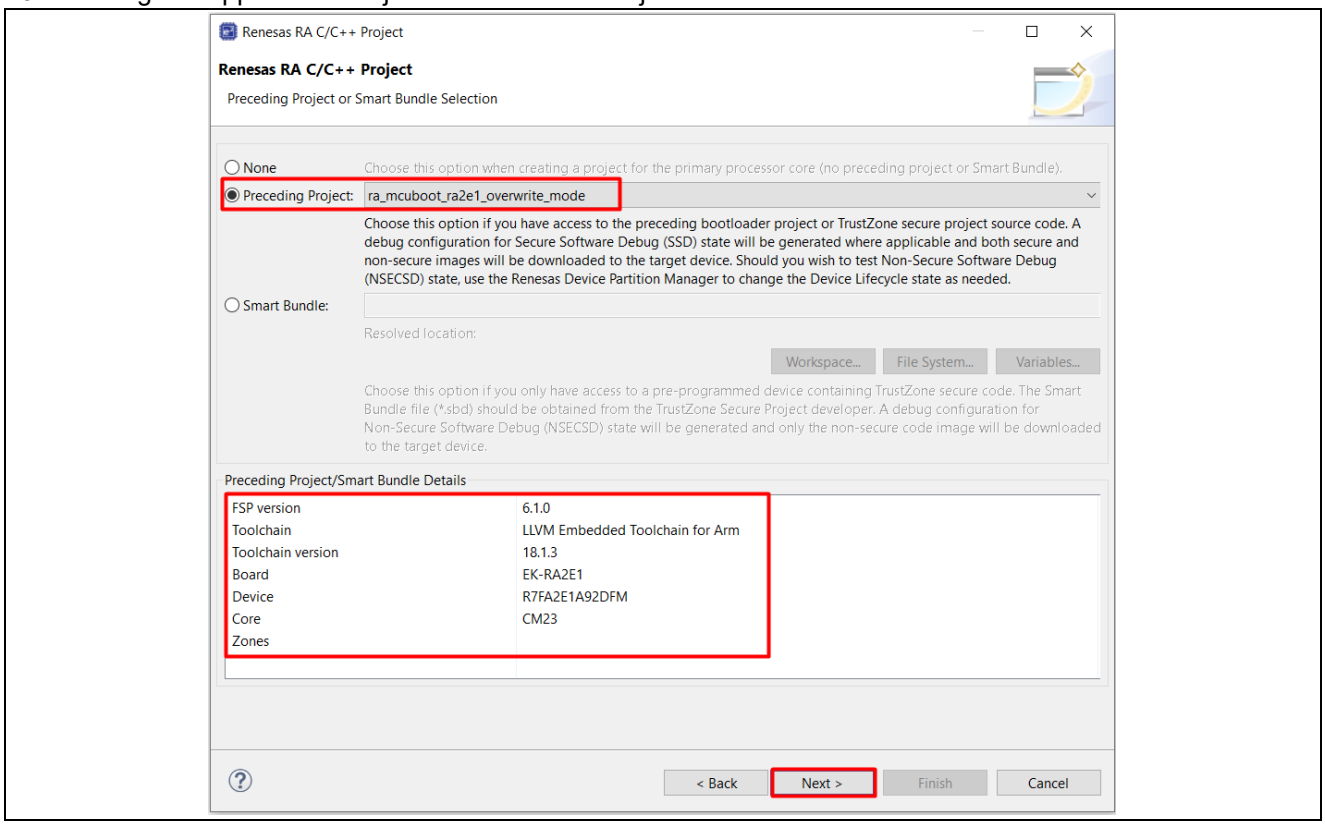

Figure 41. Selecting the Preceding Project

4. In the next screen, select **Executable** as the **Build Artifact** and **No RTOS** for the **RTOS Selection**. Then click **Next**.

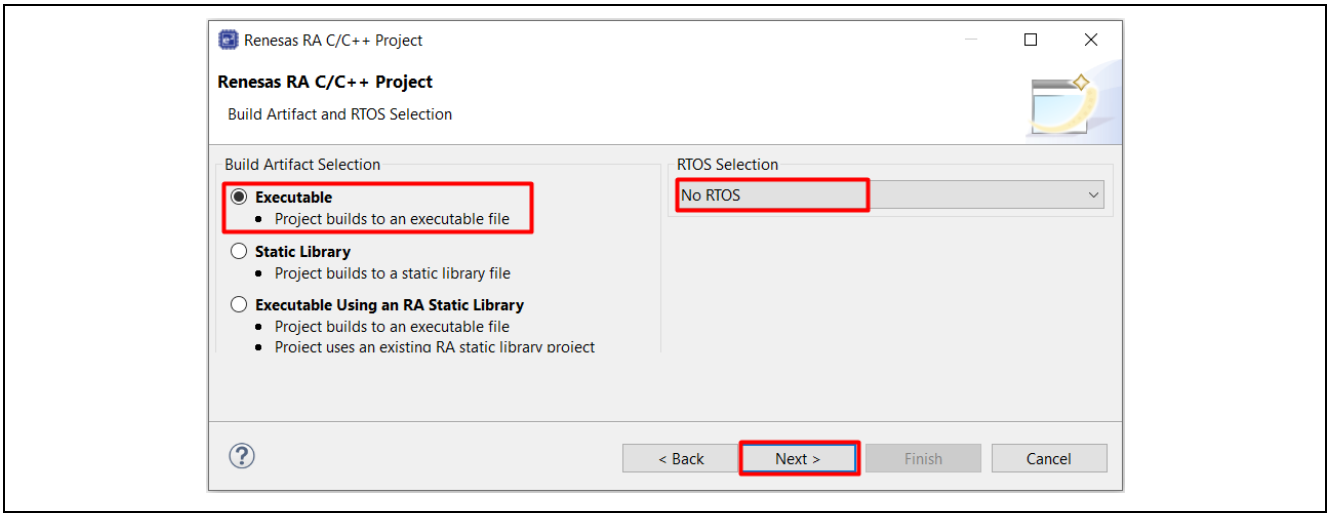

Figure 42. Choose to Build Executable with No RTOS

5. Select the **Bare Metal - Blinky** as the **Project Template** for the board and click **Finish**. The initial application project is now created.

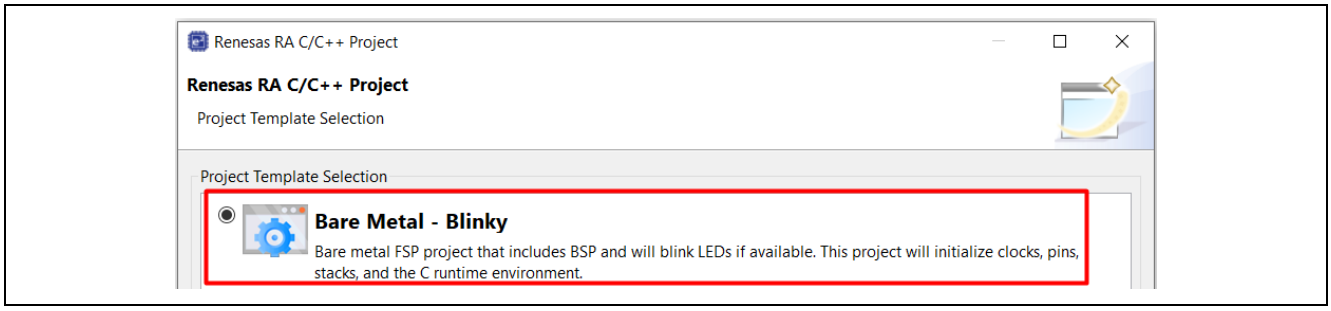

Figure 43. Choose Bare Metal – Blinky as Project Template

4.2 Import the Existing Application Project

In case users want to use the Bootloader Project with an existing application project, users can follow the steps below:

1. Import the existing application project to a workspace, along with the bootloader project.
2. Build both the existing application project and bootloader project.
3. Right-click on the existing application project and select the **Properties**.
4. In the next screen, navigate to the **C/C++ Build > Build Variables**.
5. Add the **Build Variables:**
   - Variable Name: **SmartBundle**
   - Value: `${workspace_loc:/<bootloader_project_name>}/${ConfigName}/bootloader_project_name.sbd`

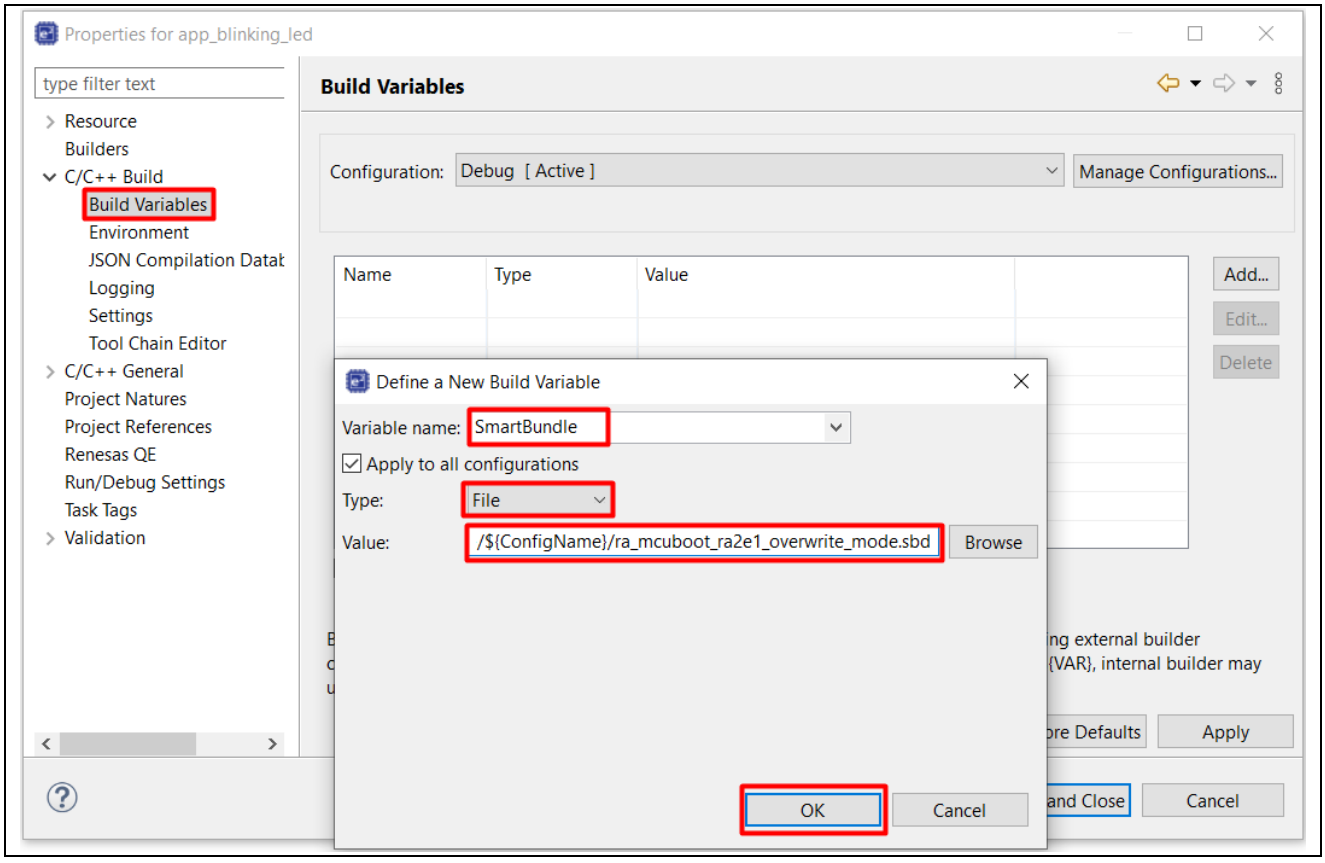

Figure 44. Adding the SmartBundle variable for existing application project

4.3 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 must select **Generate Project Content** again to update the `.sbd` file, because the application project is linked to the bootloader project through this file.

When you link the application project to the bootloader project at step 2 in section 4.1, the IDE automatically manages the dependency by updating the bootloader project's `.sbd` file with the application project information.

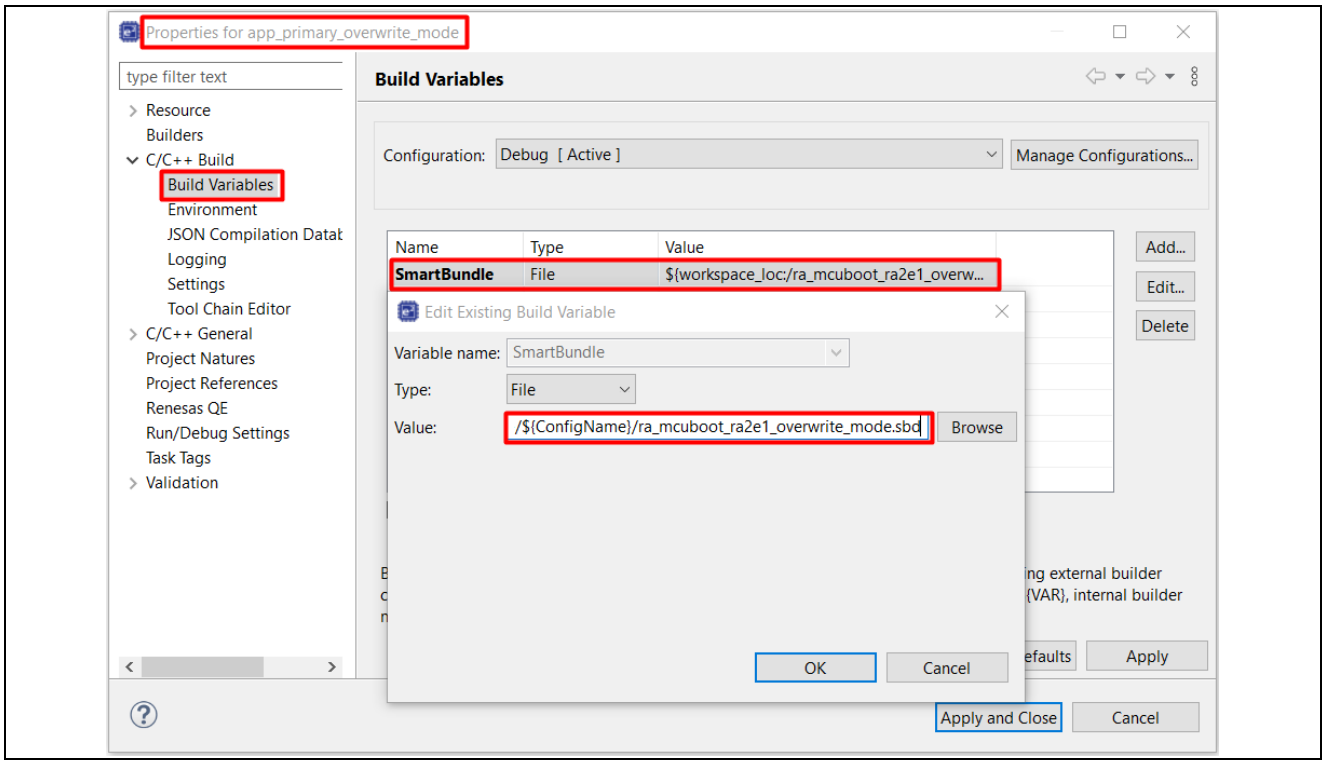

Figure 45. Access the .sbd file in the Application Project

Each application can have a defined version number. This version number can be used in the overwrite upgrade mode when **Downgrade Prevention** is **Enabled**. This is achieved by defining an Environment Variable: **MCUBOOT_IMAGE_VERSION**.

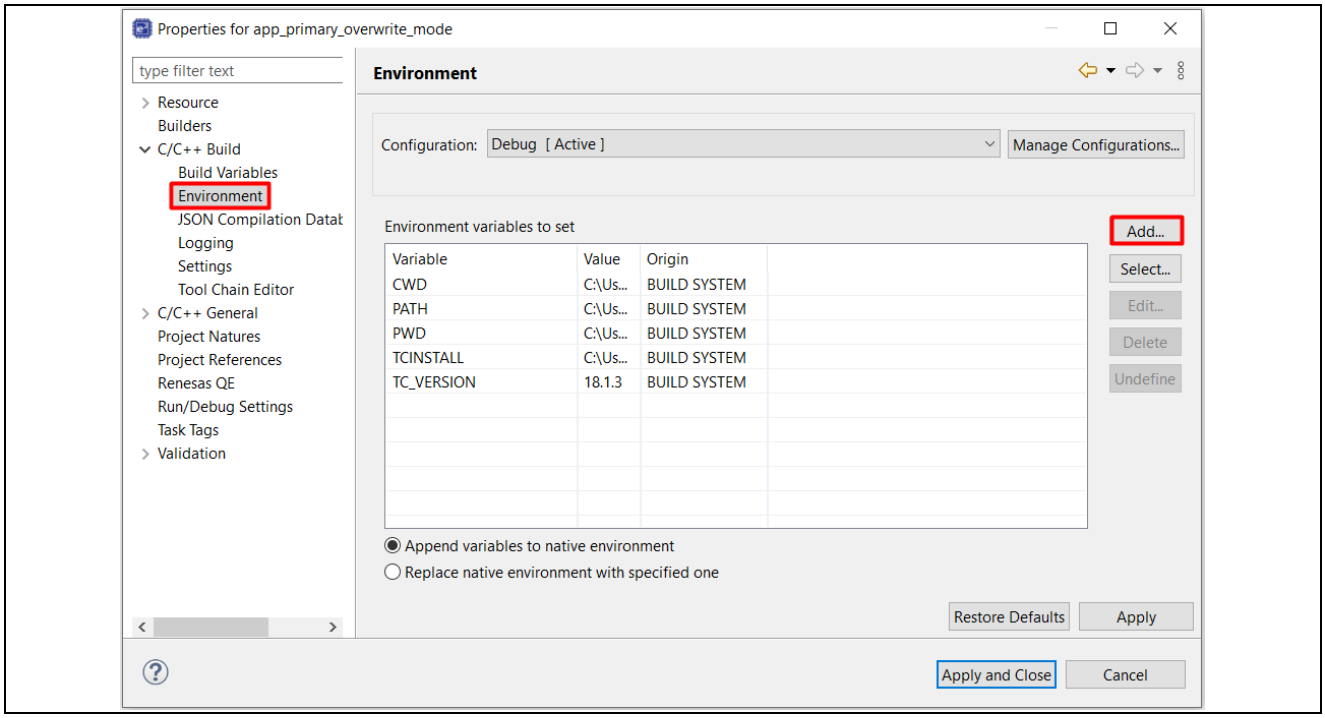

Figure 46. Add New Environment Variable

Add the Environment Variable: **MCUBOOT_IMAGE_VERSION** for the application image version.

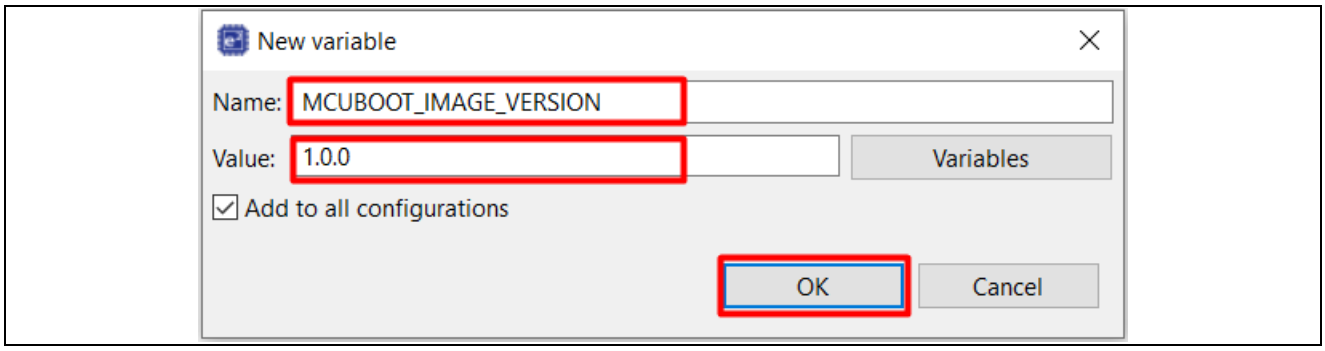

Figure 47. Add MCUBOOT_IMAGE_VERSION Variable

If there is signature verification, then it is necessary to set the Environment Variable: **MCUBOOT_IMAGE_SIGNING_KEY**

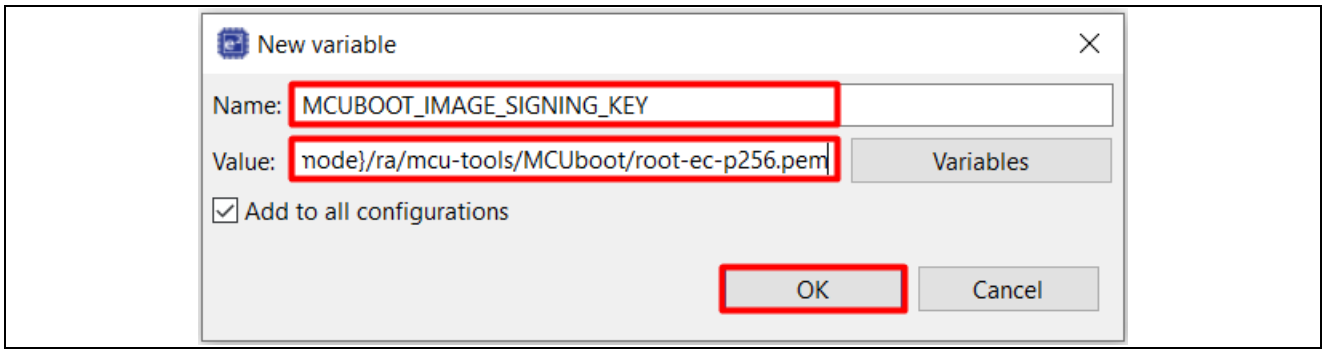

Figure 48. Add MCUBOOT_IMAGE_SIGNING_KEY Variable

When using the **LLVM Embedded Toolchain**, users need to add the Environment Variable: **MCUBOOT_APP_BIN_CONVERTER** in order to sign the application image.

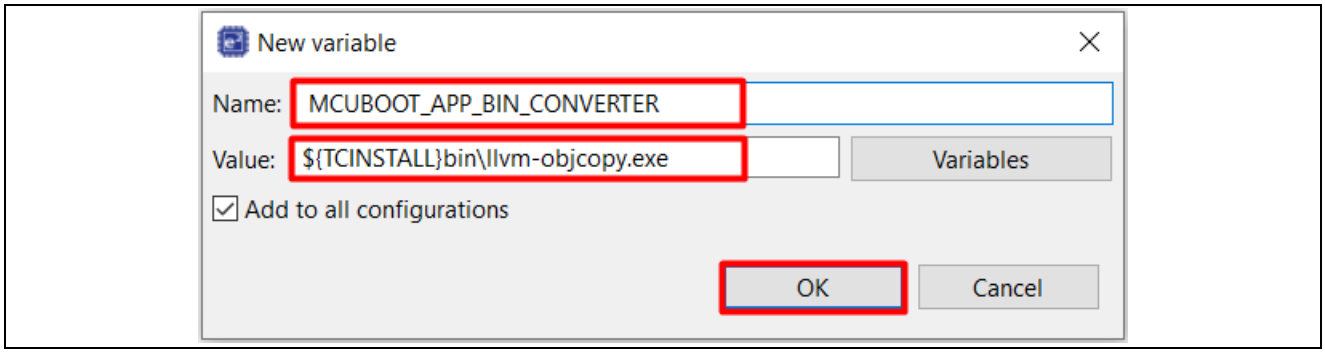

Figure 49. Add MCUBOOT_APP_BIN_CONVERTER Variable

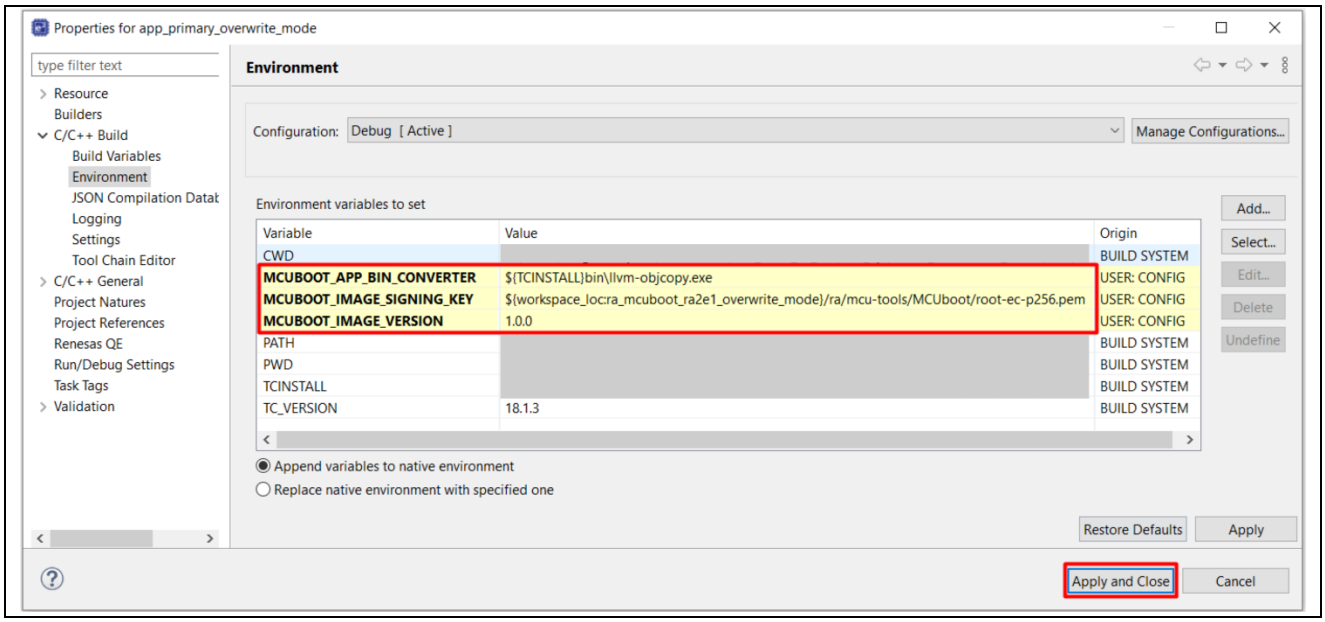

Figure 50. Configuration result

Note:

- The private key used for signing the application image is indicated in the signing command.

  `/ra/mcu-tools/MCUboot/root-ec-p256.pem` is used as an example bootloader. This key is used for testing purposes only. For real-world use cases and production support, users **MUST** change this to the private key of their choice. We will also guide users on how to use customized image signing key in section 9.

- The value of the **MCUBOOT_APP_BIN_CONVERTER** variable corresponds to the installation path of the LLVM toolchain - specifically, `${TCINSTALL}bin\llvm-objcopy.exe`.

To be able to always recompile the project when the Environment Variables or the linker script are updated, it is recommended to add a **Pre-build** step to always delete the `.elf` file as shown in Figure 51.

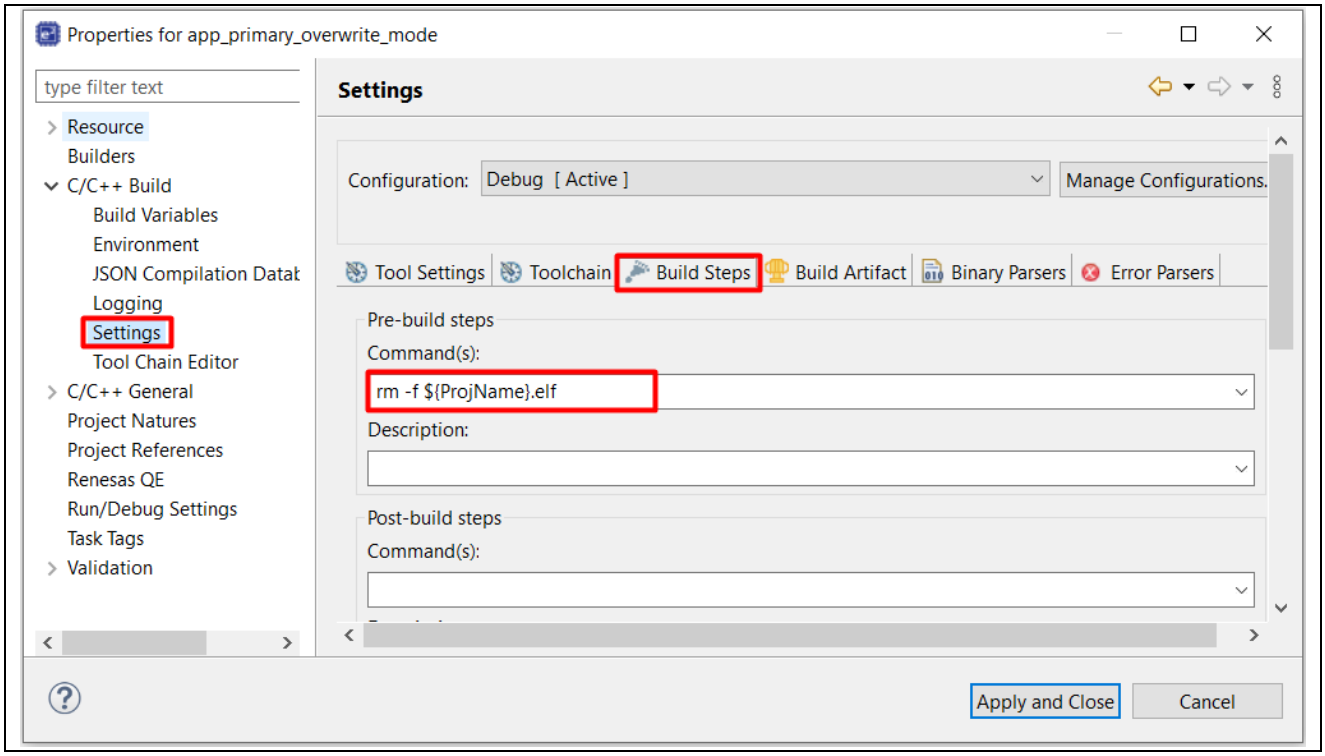

Figure 51. Configuration the Pre-build Command

Next, users can add the RTT Viewer usage related application code to the primary application project. Uzip the `RA2_Secure_Bootloader_using_Ocrypto.zip`, open the `ra2e1_overwrite_update_mode\app_primary_overwrite_mode\src` folder and copy all files under `\src` to the `\src` folder for the newly established project.

In addition, when using the Image Encryption with Bootloader Project, as described in section 3.2. Users need to add the Environment Variable: **MCUBOOT_IMAGE_ENC_KEY**.

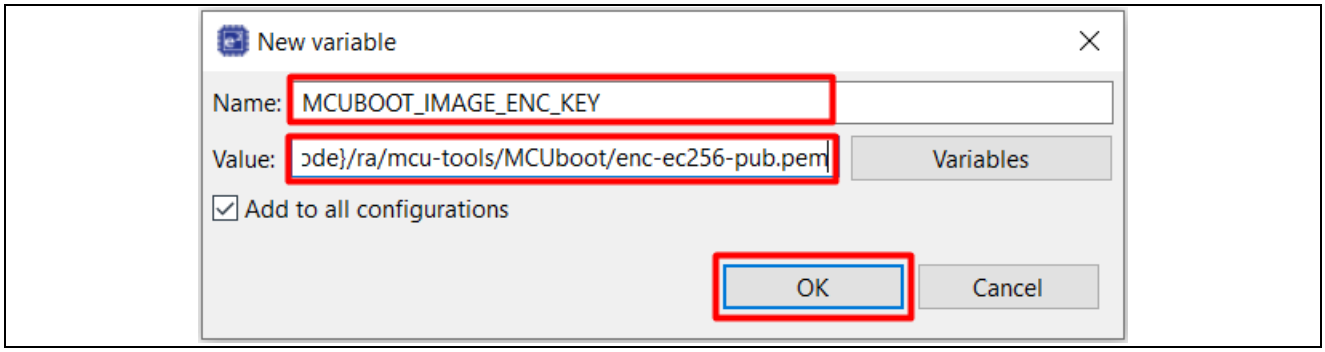

Figure 52. Add MCUBOOT_IMAGE_ENC_KEY Variable

Define the **Value** as:

```
${workspace_loc:ra_mcuboot_ra2e1_overwrite_mode}/ra/mcu-tools/MCUboot/enc-ec256-pub.pem
```

When building the application project, an error will appear in the **Console** tab, as shown in Figure 53.

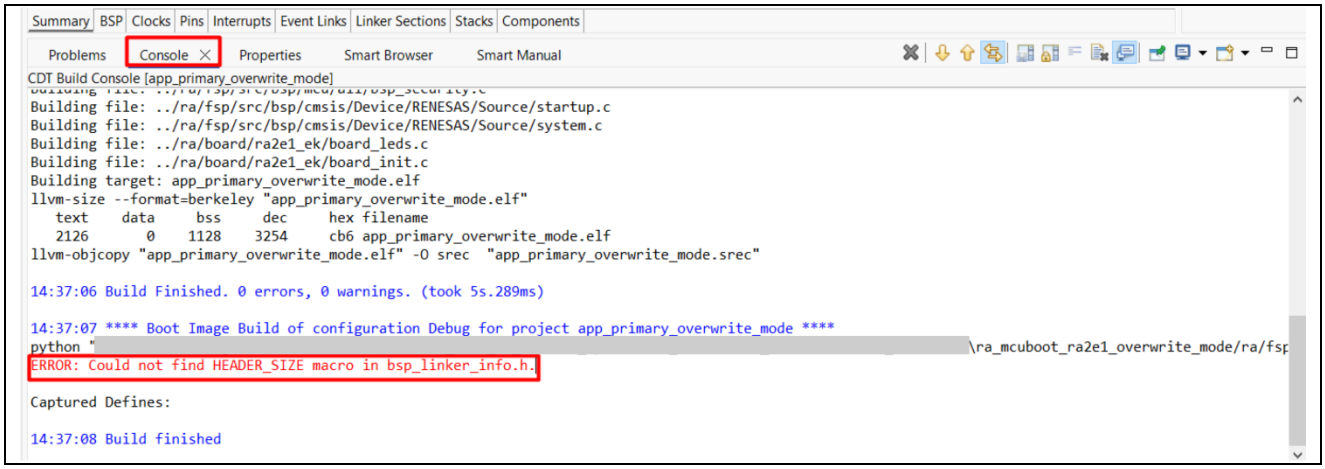

Figure 53. Error message when building the application project

To resolve this issue, users should refer to section 5 for the detailed procedure.

5. MCUboot Memory Configuration with Renesas FSP Solution Project

Starting from FSP v6.0.0 with e²studio v2025-04.1, when creating the Bootloader Project, users must use the **Renesas FSP Solution Project (Advanced)**. This enhancement also provides more effective and intuitive bootloader memory management by allowing users to edit the "`solution.xml`" file within the **Solution Project**.

In other words, the **FSP Solution Project (Advanced)** is a chain of projects, specifically consisting of the bootloader and the application project that are created in the previous steps.

5.1 How to Set Up the Renesas FSP Solution Project

Click **File** > **New** > **C/C++ Project > All** and select **Renesas FSP Solution Project (Advanced)**, as shown in Figure 54.

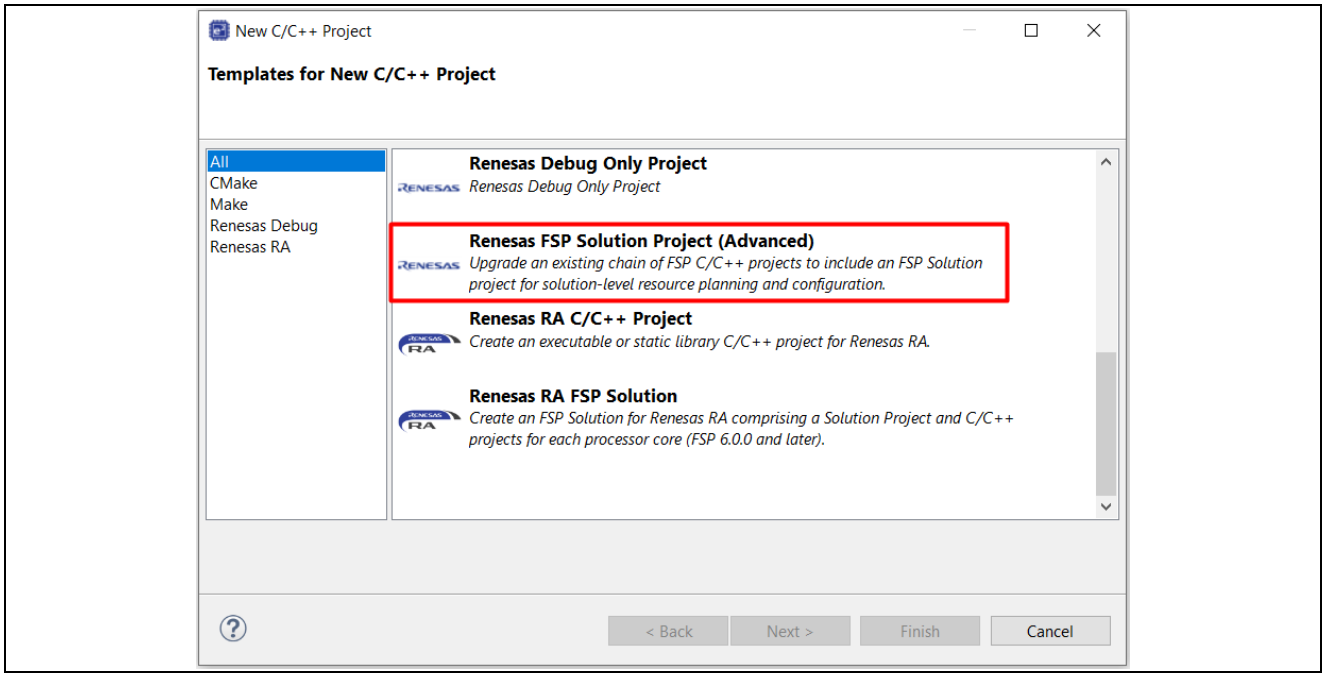

Figure 54. Renesas FSP Solution Project (Advanced)

1. Assign the project name based on Table 3.

Table 3. Name the Renesas Solution Project

Bootloader project name	Initial application project name	Renesas Solution Project name
ra_mcuboot_ra2e1_overwrite_mode	app_primary_overwrite_mode	overwrite_mode_solution
ra_mcuboot_ra2e1_swap_mode	app_primary_swap_mode	swap_mode_solution

2. Select the final project in a chain of projects.

In the **Project** option, select the `app_primary_overwrite_mode`, which is created in section 4.

Click **Finish**.

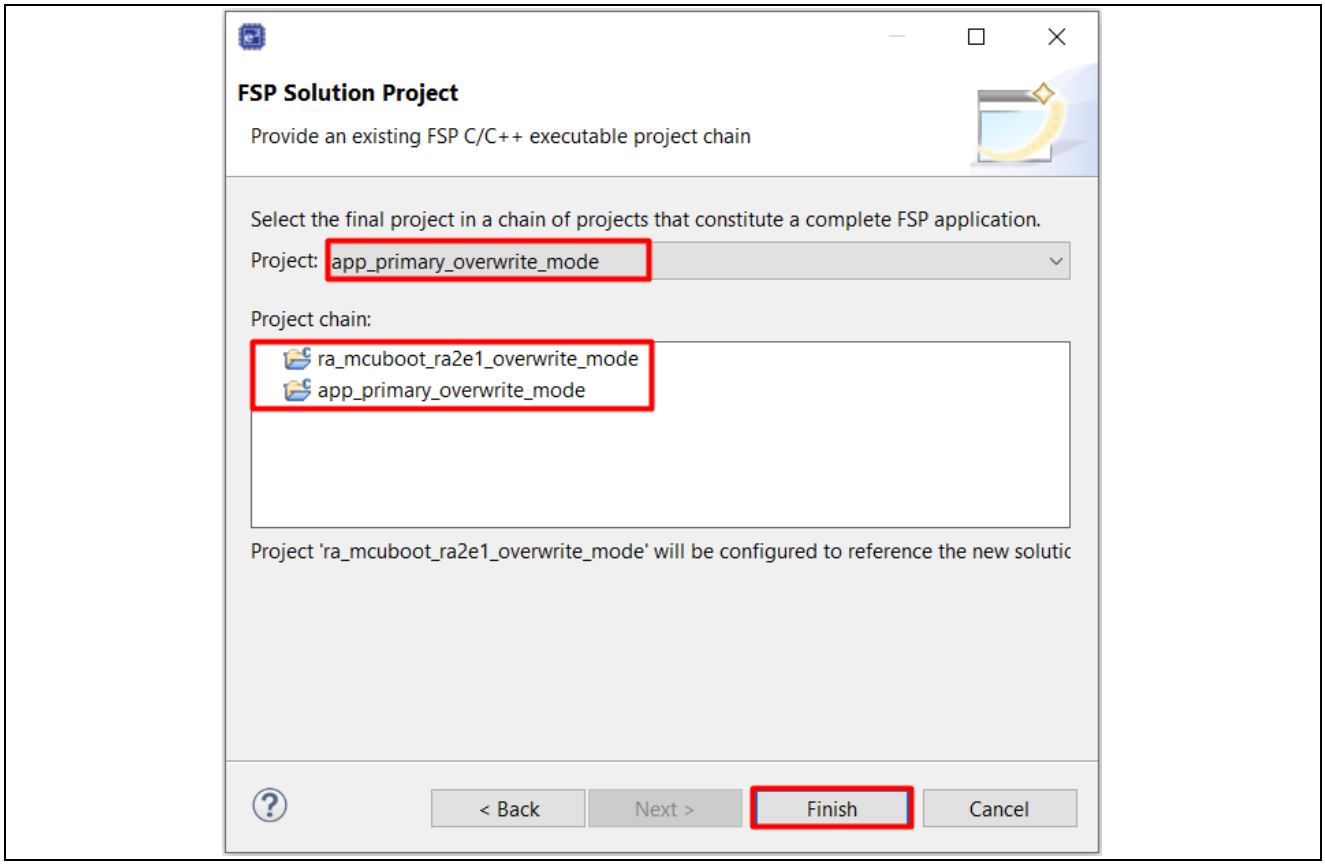

Figure 55. Select the application project in chain of projects

Since the application project already contains the information of the MCUboot project through the `.sbd` file, the **Project Chain** therefore also includes the bootloader project, as shown in Figure 55.

5.2 Managing the MCUboot Memory Configuration

Prior to configuring the MCUboot memory settings within the **Solution project**, we will present a flash map illustration to help users visualize it more easily, as shown in Figure 56 and Figure 57.

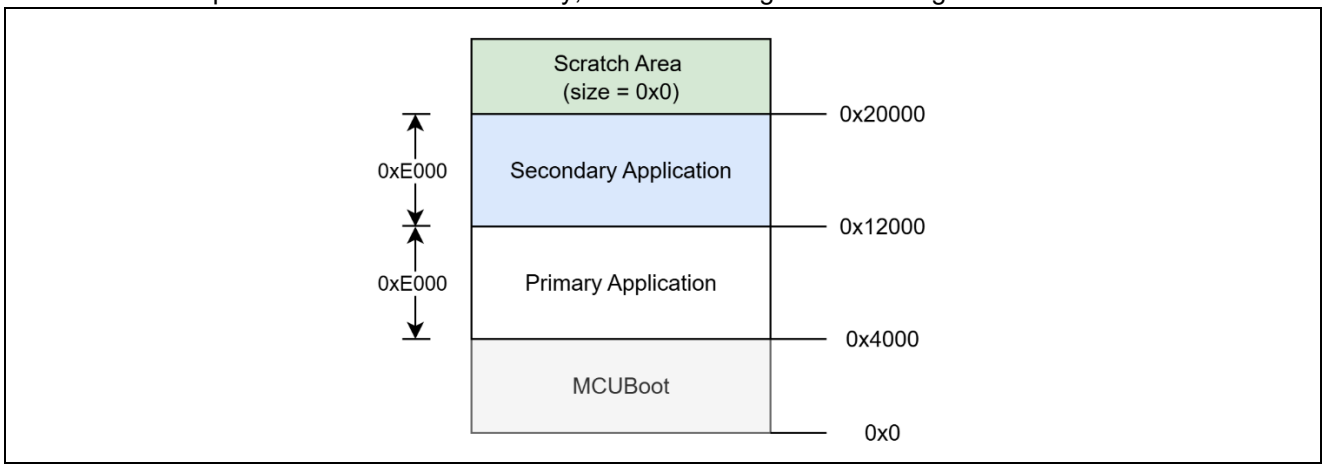

Figure 56. MCUboot Flash Map in Overwrite Update Mode

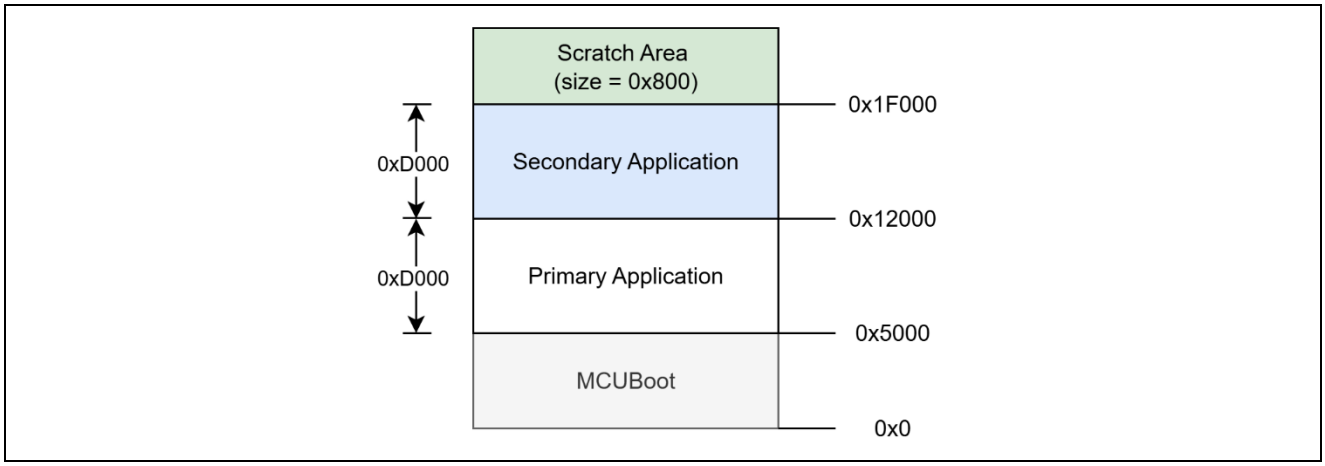

Figure 57. MCUboot Flash Map in Swap Update Mode

1. Memory Partition in Solution Project

Doule-click `solution.xml` in the `overwrite_mode_solution` project. Then, navigate to the **Memories** tab to configure the MCUboot flash map.

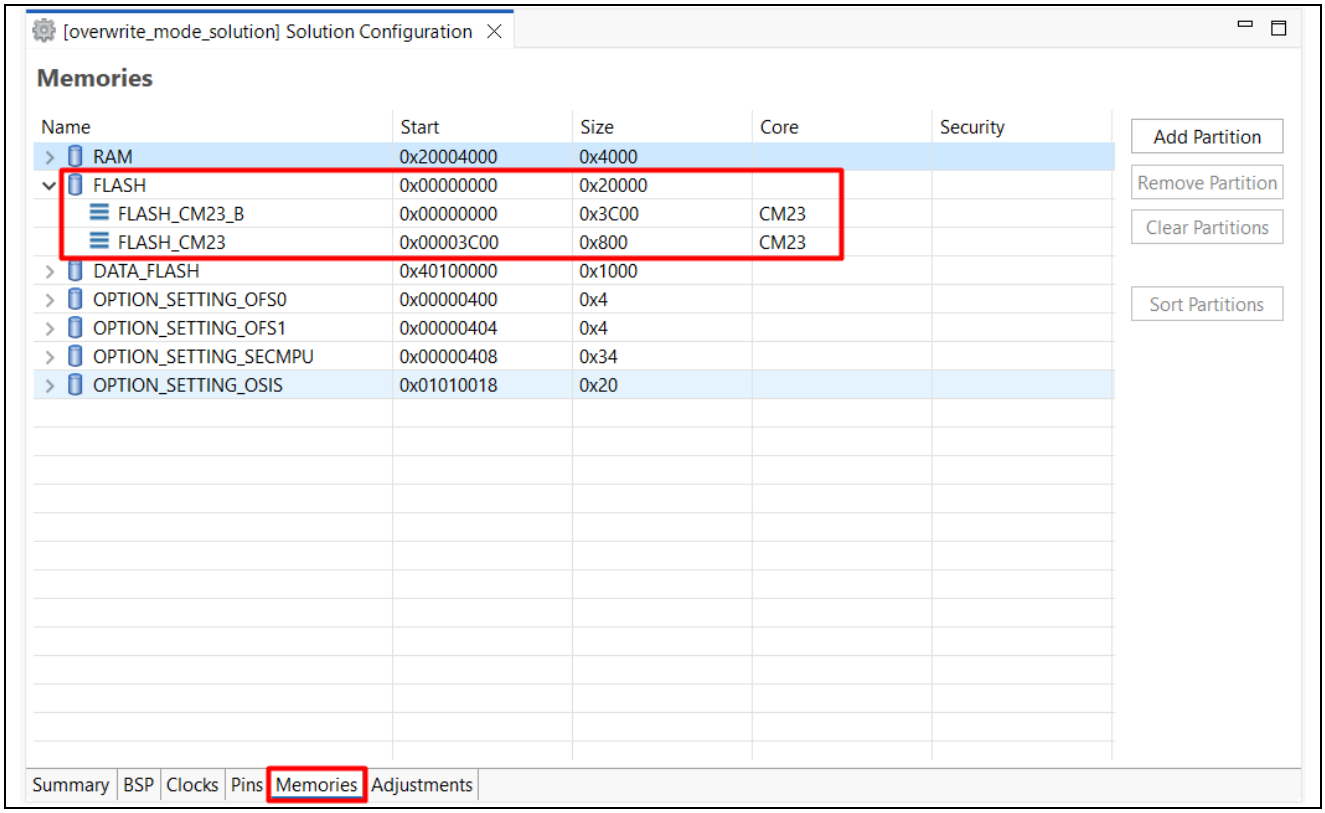

Figure 58. Memory Partition in Solution Project

When configuring information related to the bootloader size, image size, and header size, users must name it as described below in the memory partition of the **Solution Project.**

Table 4. Memory Partition Labels for MCUboot Flash Map

Memory Partition Labels	Flash Map	Definition of Memory Partition Labels
__BL_S	Scratch Area	Scratch area
__BL_0_S_I	Trailer TLV app_secondary.bin	Image 0 Secondary Image
__BL_0_S_H	Header	Image 0 Secondary Header
FLASH_CM23	Trailer TLV app_primary.bin	Image 0 Primary Image
__BL_0_P_H	Header	Image 0 Primary Header
FLASH_CM23_B	MCUboot	Bootloader area

For more details about the MCUboot Memory Partition Labels, users can refer to the [renesas.github.io: MCUboot Port]

Based on information in Figure 56 and Figure 67, users can configure the MCUboot Flash Map in the memory partition of the **Solution Project** as follows.

Add the MCUboot Memory Partition Labels, as shown in Figure 59.

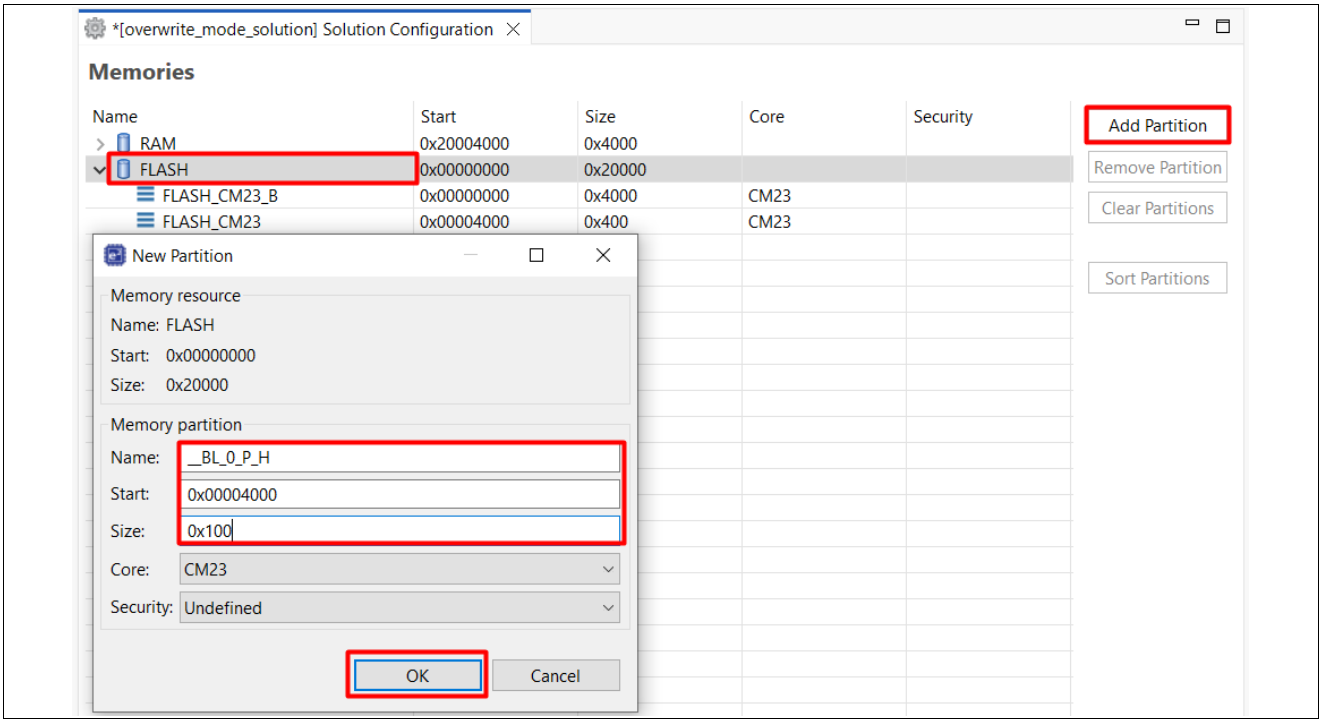

Figure 59. Add the MCUboot Memory Partition Labels

Users need to sequentially copy the labels from Table 4 into the Memory Partition of the **Solution Project**, and then press **Ctrl + S** to save the settings.

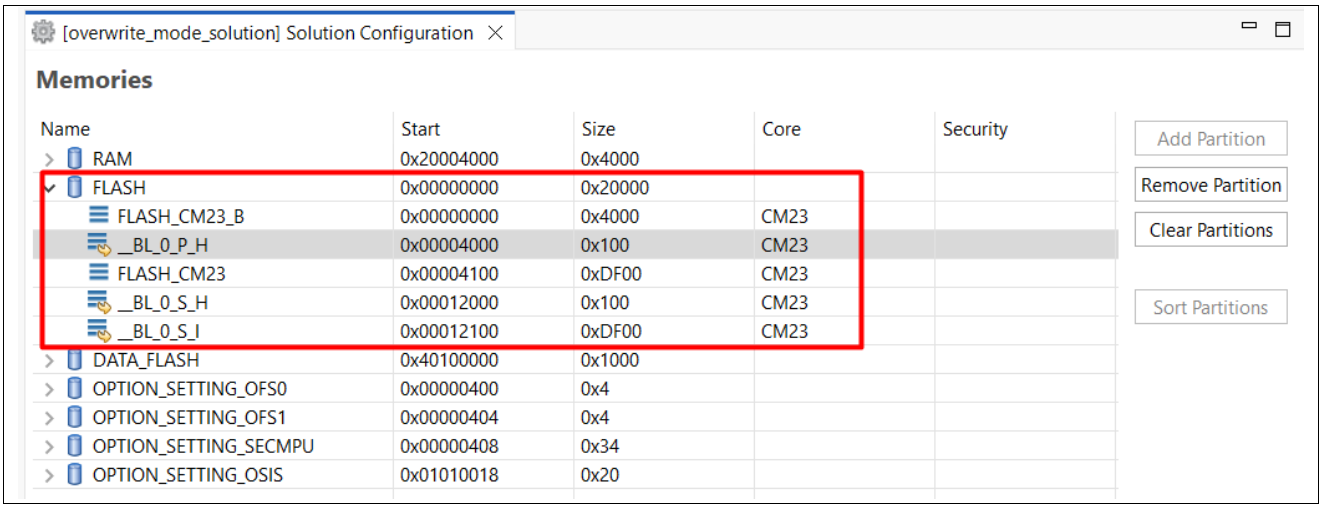

Figure 60. The MCUboot Memory Partition in Overwrite Update Mode

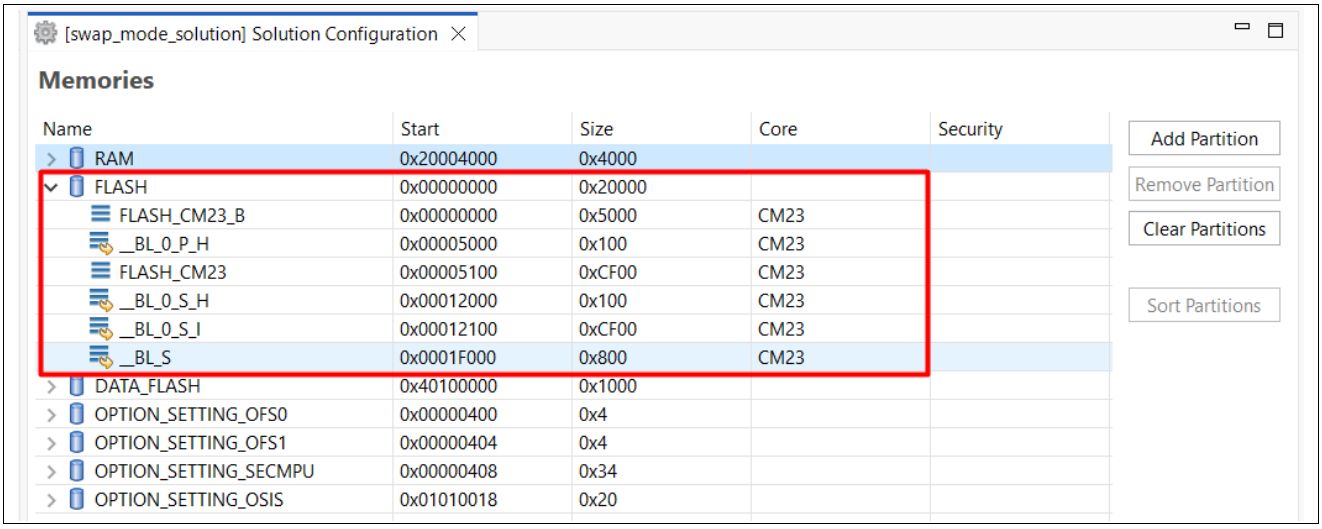

Figure 61. The MCUboot Memory Partition in Swap Update Mode

2. Build the **Solution Project**

Right-click on the **Solution Project**, and select the **Build Project**. This command will build all projects within the **Solution Project**.

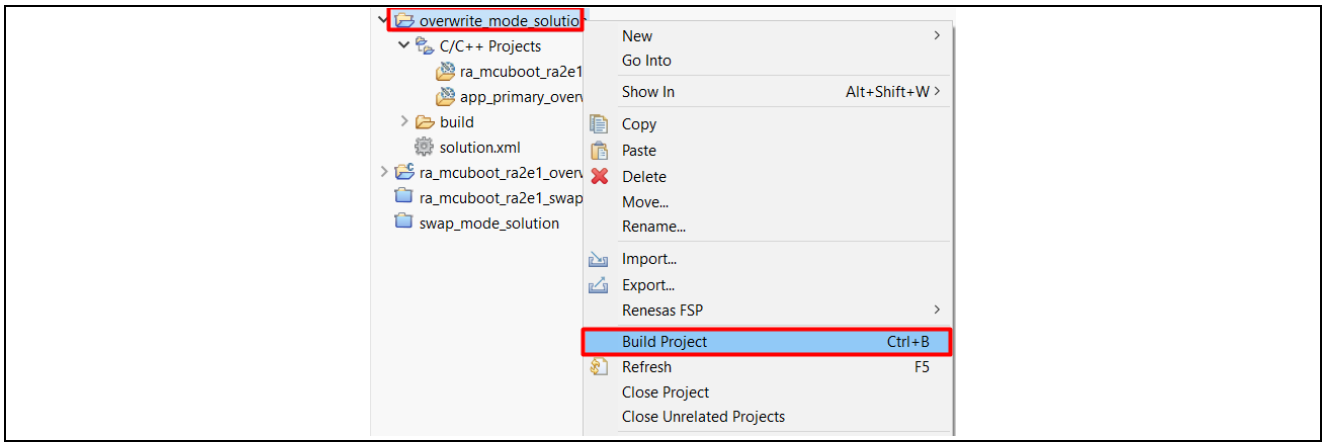

Figure 62. Build all projects within the Solution Project

At this point, when Image Encryption is disabled the `\debug\<Initial application project name>.bin.signed` file is generated.

When Image Encryption is enabled, `\debug\<Initial application project name>.bin.signed.encrypted` file is generated.

Note: If an error related to RAM region overflow appears in the **Console** tab, users need to adjust the RAM region in the **Memory Partition** settings of the **Solution Project**.

- For the Bootloader Project, adjust the **RAM_CM23_B** size.
- For the Application Project, adjust the **RAM_CM23** size.

In addition, users can refer to 9.4 to maximize the SRAM usage for the application project.

6. Booting the Initial Application Project.

6.1 Set Up the Hardware

Connect J10 using a USB micro to B cable from EK-RA2E1 to the development PC to provide power and debug connection using the on-board debugger.

6.2 Erase the MCU

To create a clean environment for starting the initial application project. Users should erase the MCU using the third-party tools, like J-Flash Lite.

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

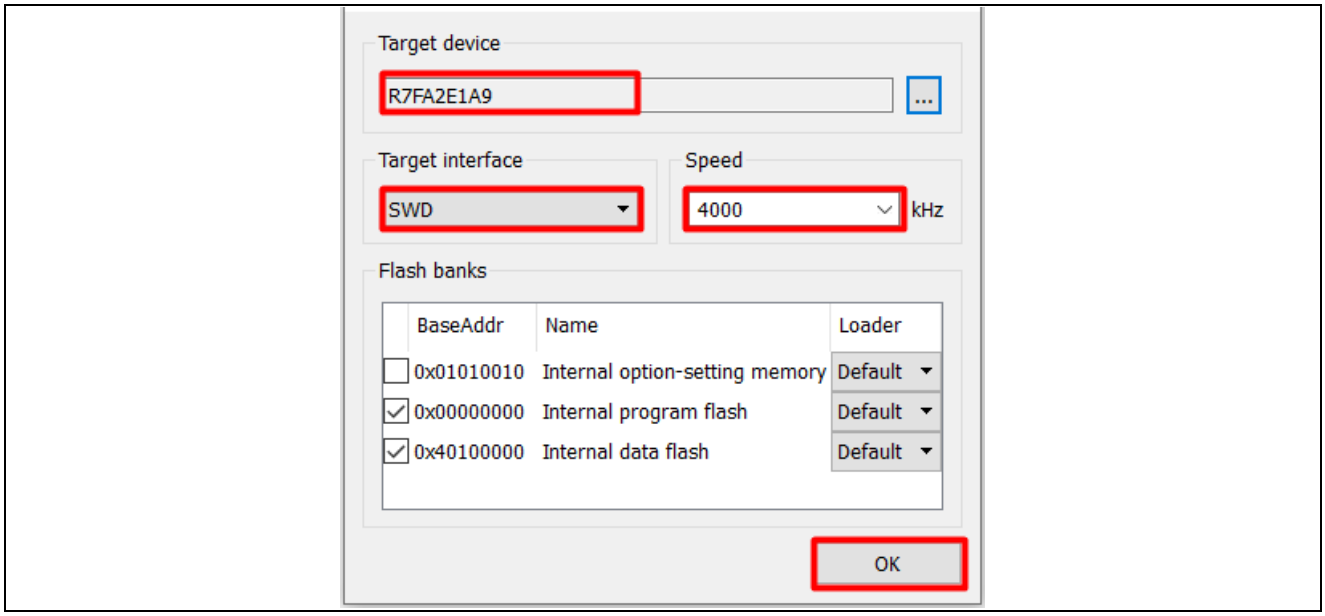

Figure 63. Launch J-Flash Lite

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

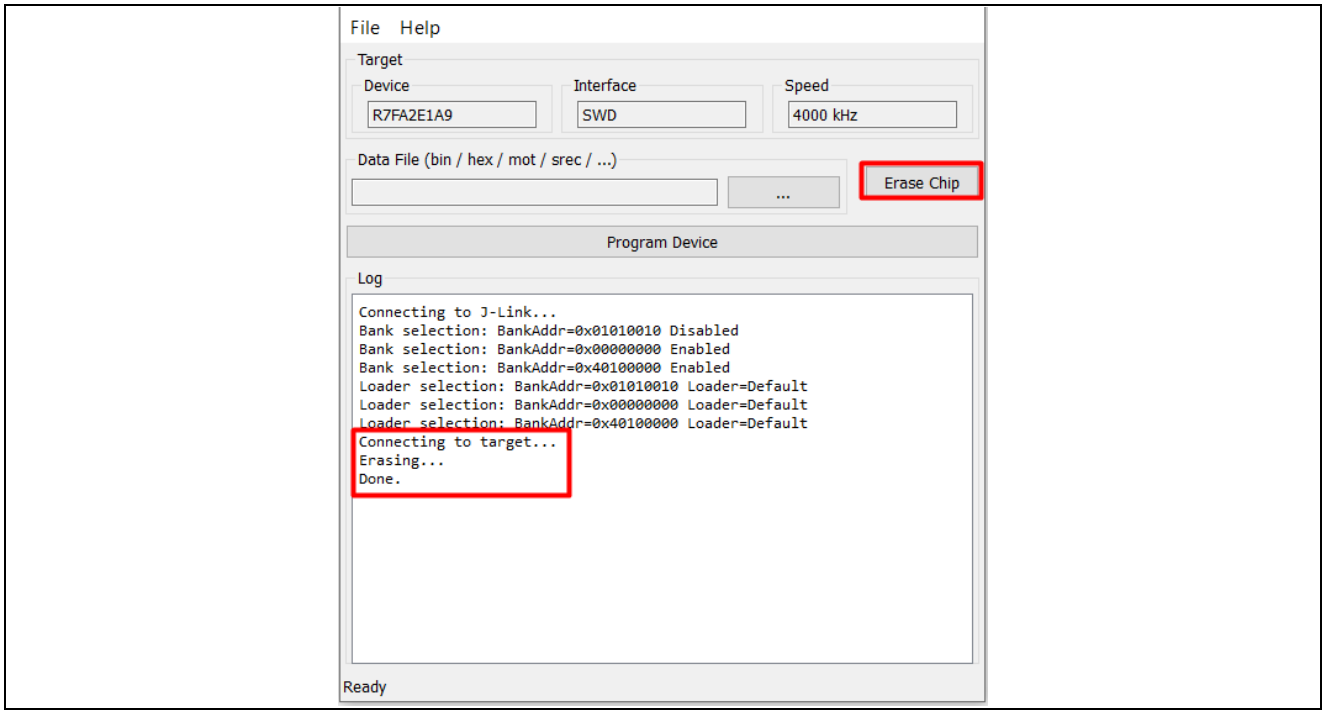

Figure 64. Erase MCU Succeeded

6.3 Configure the Debugger

Right-click on the **<initial_application_project_name> project,** and open the **Debug Configurations**: **<initial_application_project_name> > Debug As > Debug Configurations**

Optional Step: Set Allow caching of flash contents to **No**, as shown in Figure 65. Otherwise, the debugging bootloader applications memory window information may show wrong information.

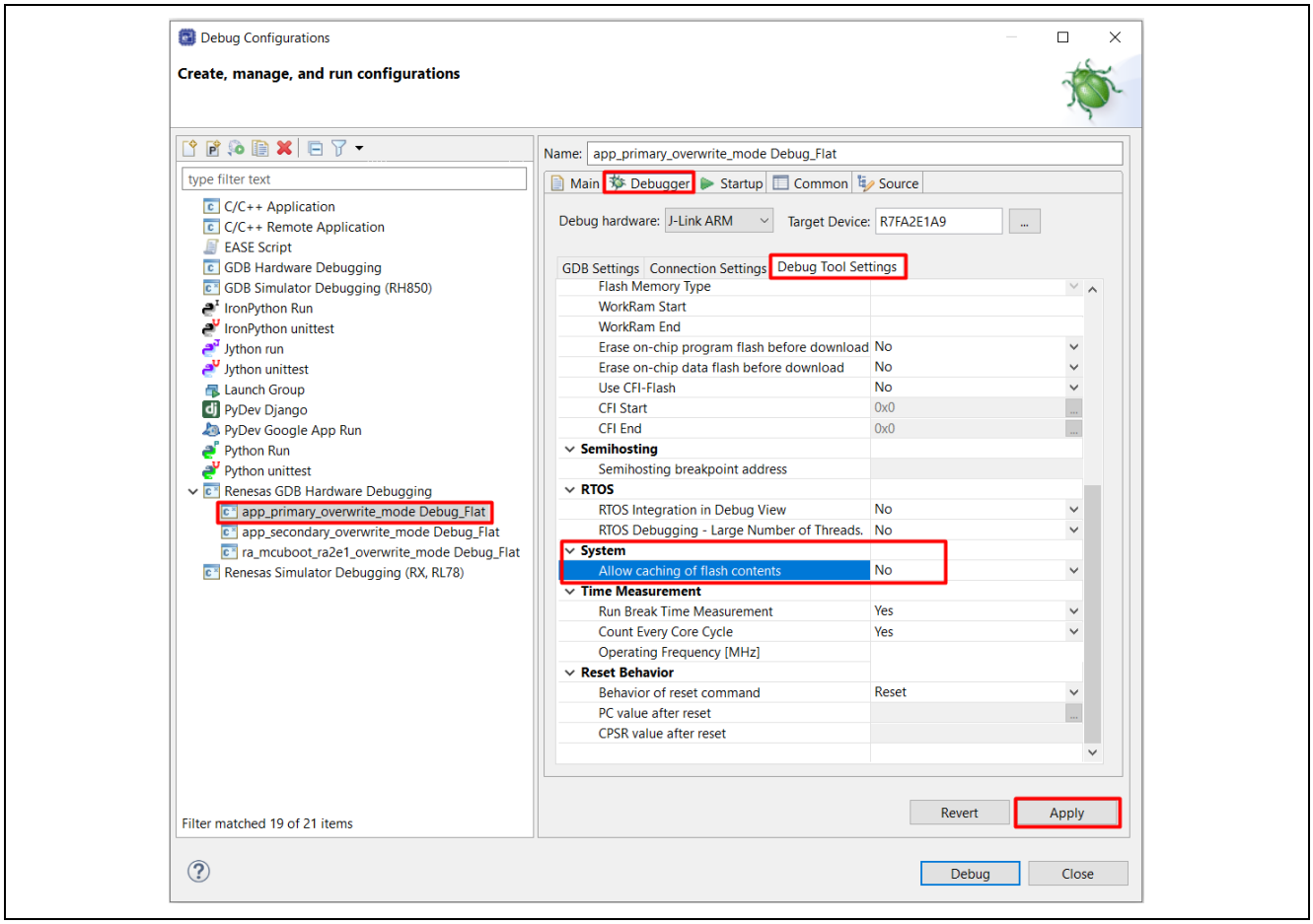

Figure 65. Disable Flash Content Caching

Make sure the **<initial_application_project_name > Debug_Flat** is selected and select the **Startup** tab.

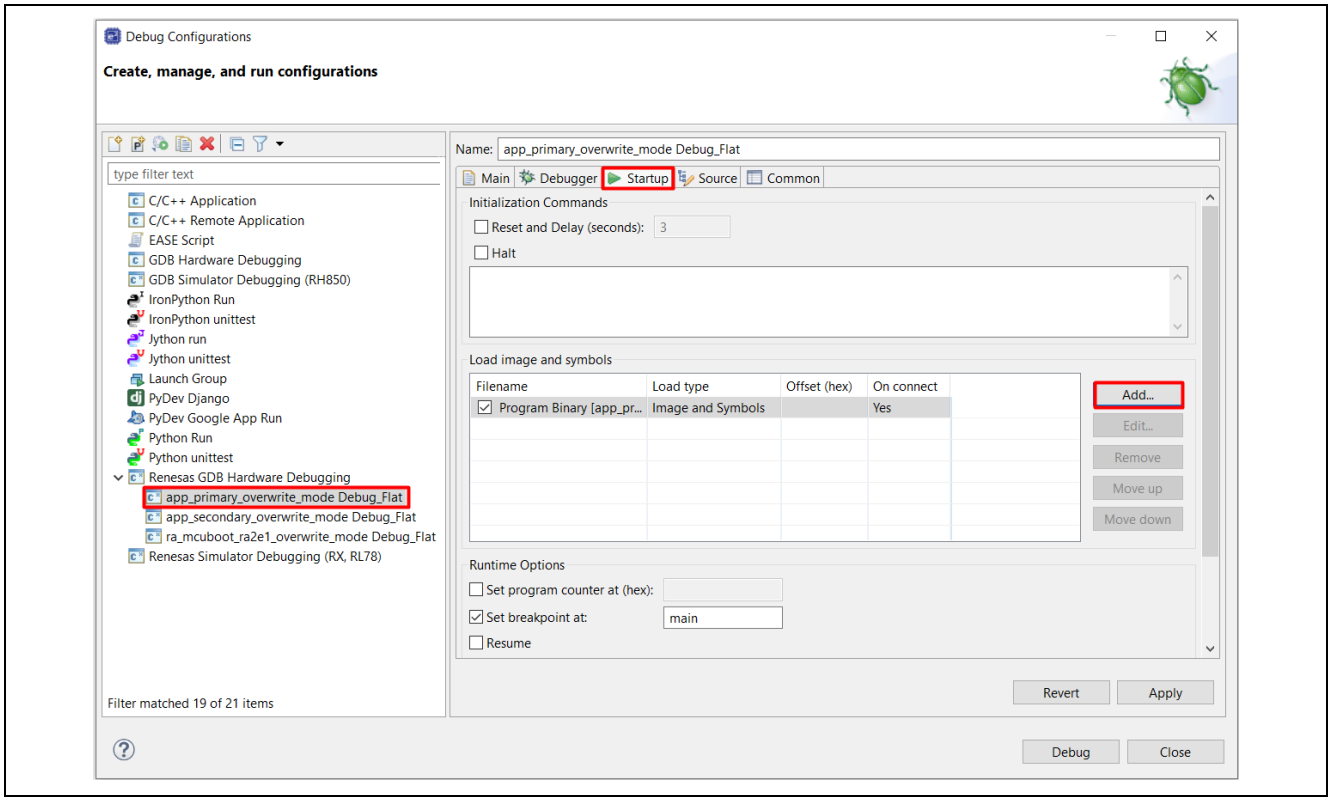

Figure 66. Configure the Primary Project Debug Startup

Click **Add…** and then **Workspace** and navigate to the **<bootloader_project_name>** and select the **<bootloader_project_name>.elf** file from the debug folder. Click **OK**.

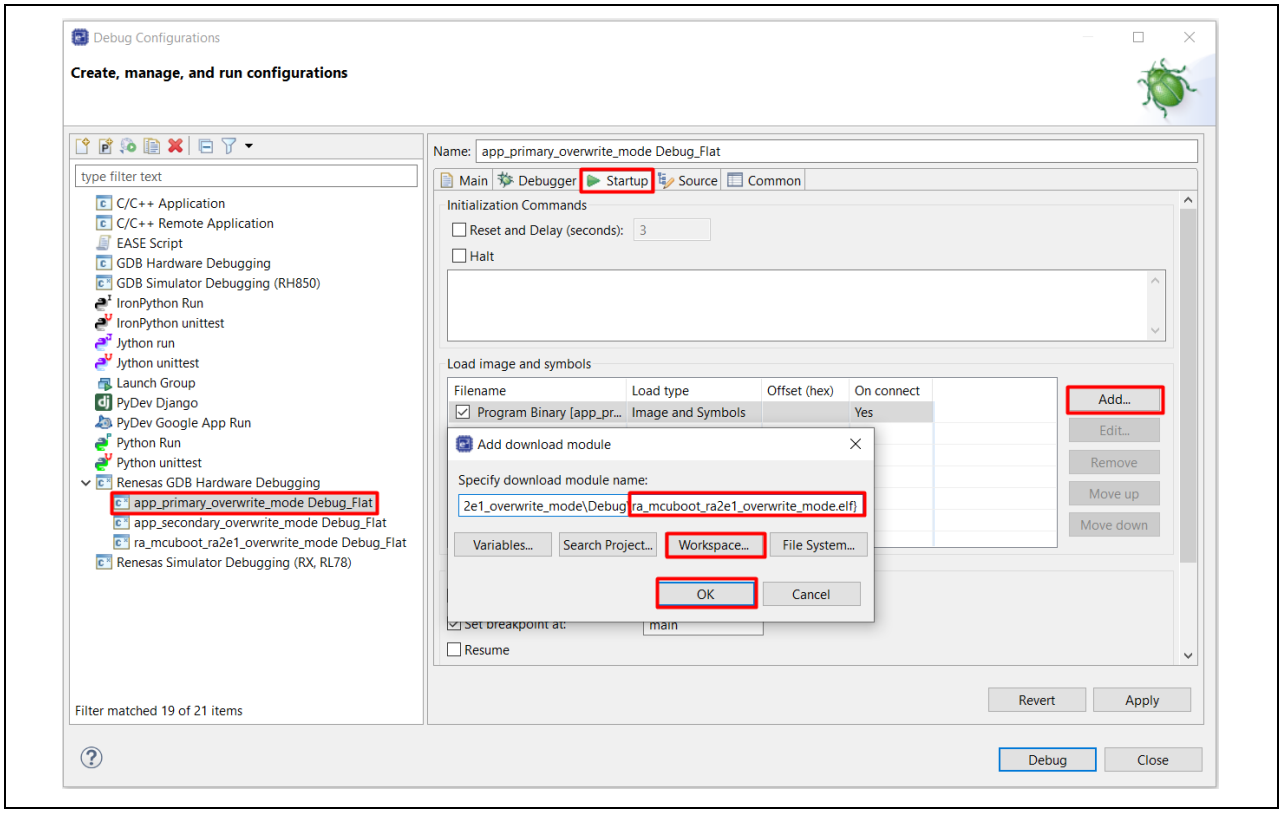

Figure 67. Add the Bootloader Project to Debug Configuration

Change the load type of the Program Binaries for the **<initial_application_project_name>** project to **Symbols only** by clicking on the cell for load type and selecting **Symbols only** from the drop-down menu.

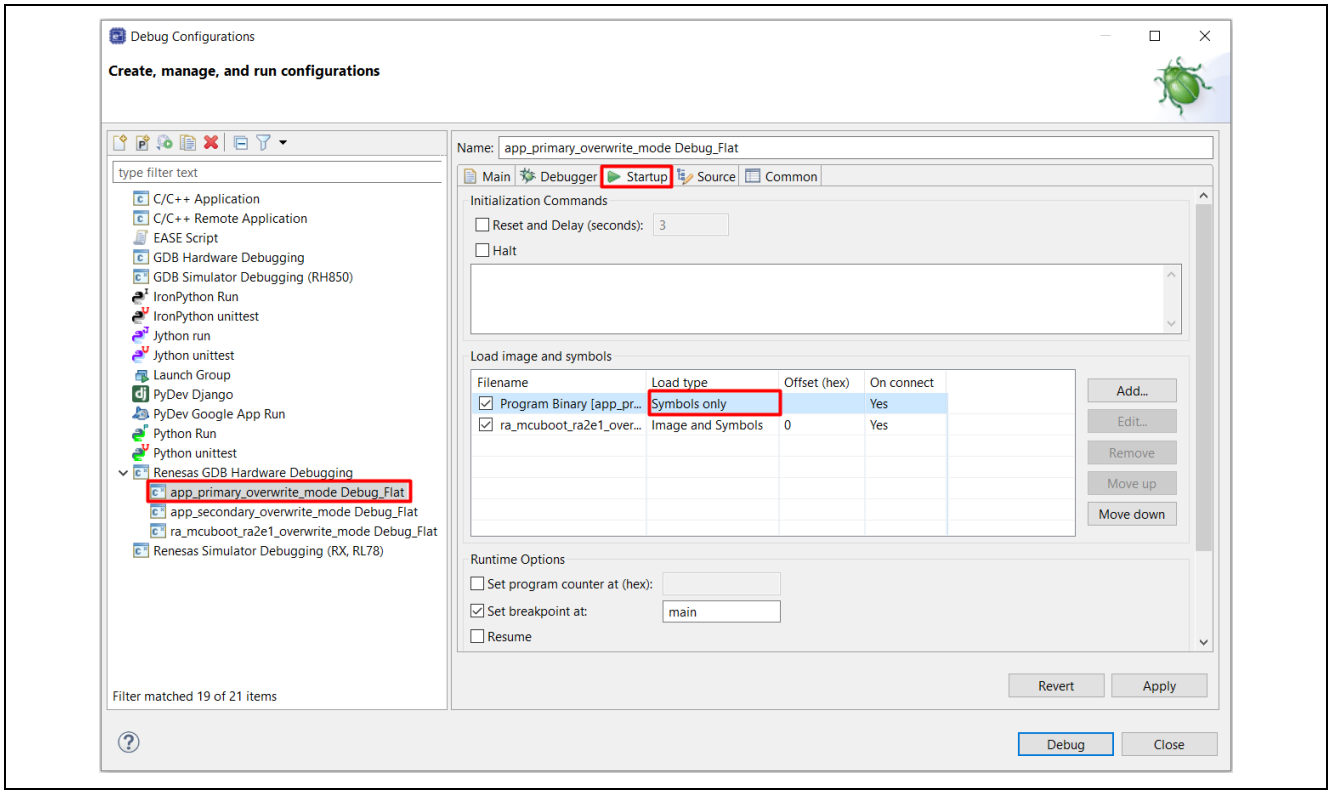

Figure 68. Select to Load Symbols Only for the Application Project

Next, configure the Debug Configuration to include the **Raw Binary** of the signed primary application for download. Click **Add…** and then **Workspace** and navigate to the **<initial_application_project_name>** and select the **<initial_application_project_name>.bin.signed** file from the debug folder. Click **OK**. Then, change the **Load type** to **Raw Binary**.

Note that the **Offset (hex)** setting of the signed primary image is the size of the bootloader (refer to Figure 56). Figure 69 is an example of downloading the signed primary image for the overwrite project.

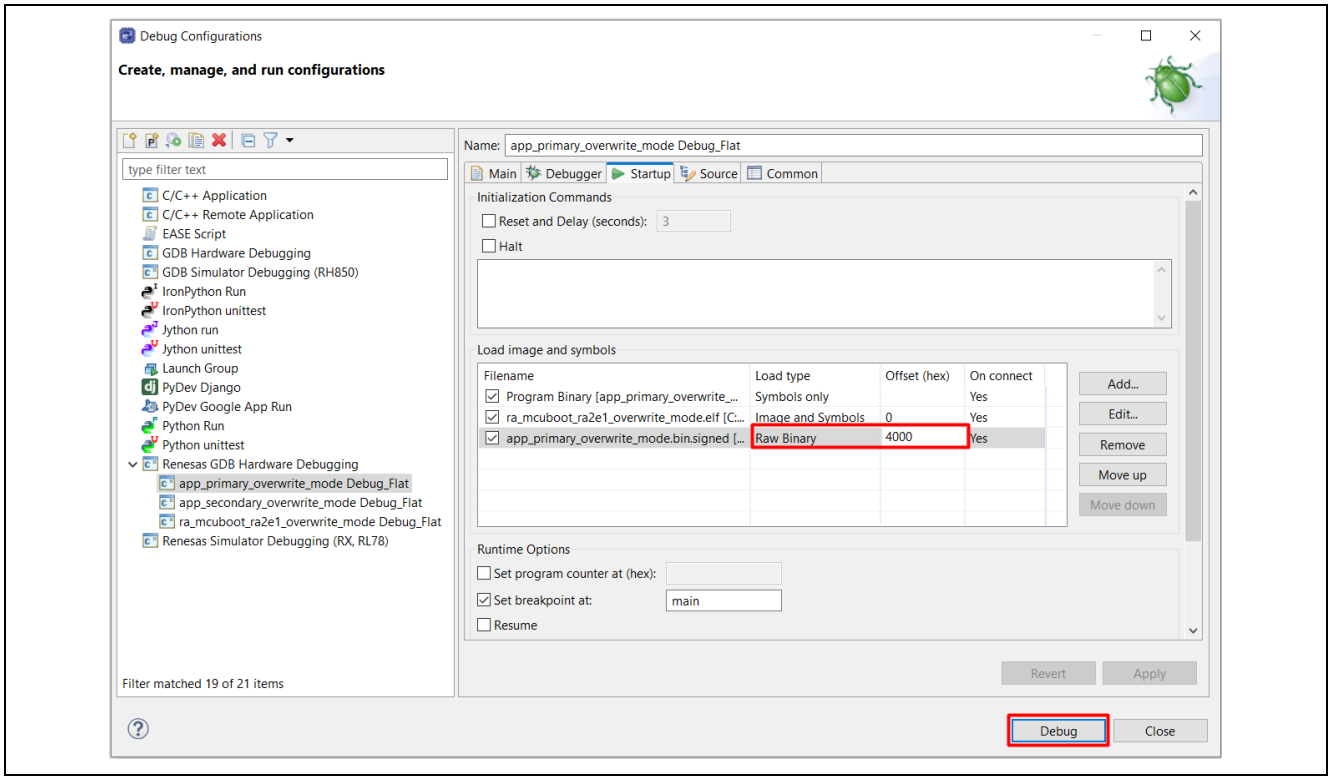

Figure 69. Include the Raw Binary of the signed image in the download

Click **Debug**. The debugger should hit the `Reset_Handler()` in the bootloader project.

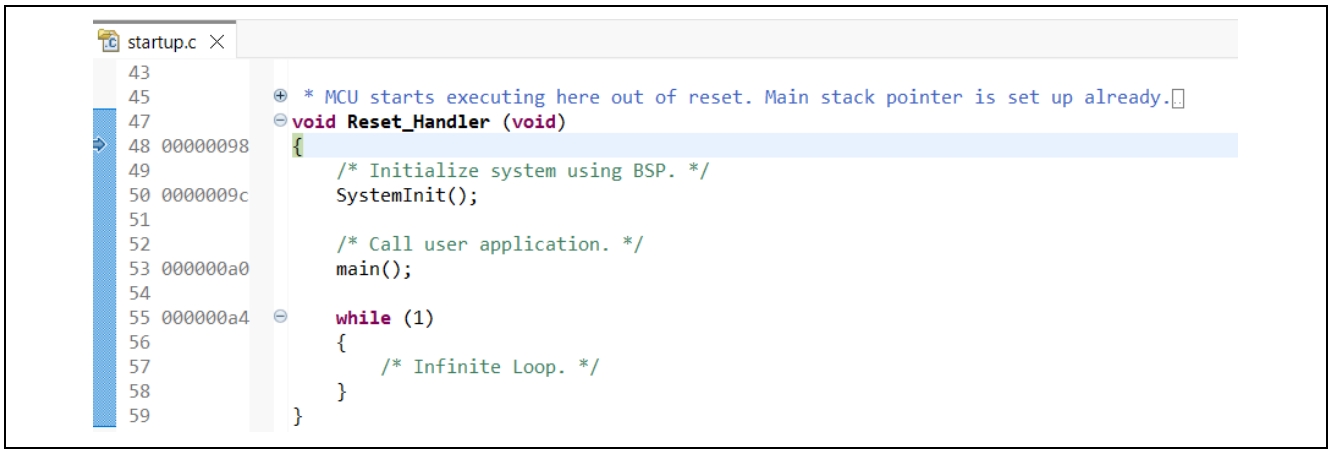

Figure 70. Start the Application Execution

Click **Resume** twice to run the project. The bootloader and the primary application project will be programmed and then the primary application project will be booted, the Red, Blue, and Green LEDs on the EK-RA2E1 should now be blinking.

Press to pause the program. Note that the program counter is in the application image. Click Resume to run again.

6.4 Open the J-Link RTT Viewer

Open the J-Link RTT Viewer and set up the following configurations. Set up the search range as 0x20000000 0x8000.

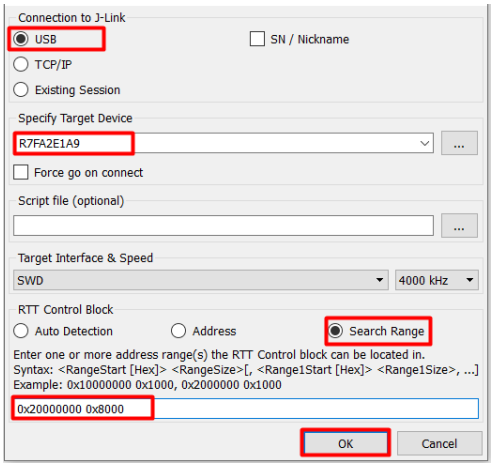

Figure 71. Configure the RTT Viewer

Click **OK** and observe the following output on the RTT Viewer. This output shows that the Primary application is being executed and all three LEDs are blinking. The message displayed indicates the upgrade mode and whether the Primary or the Secondary image is running.

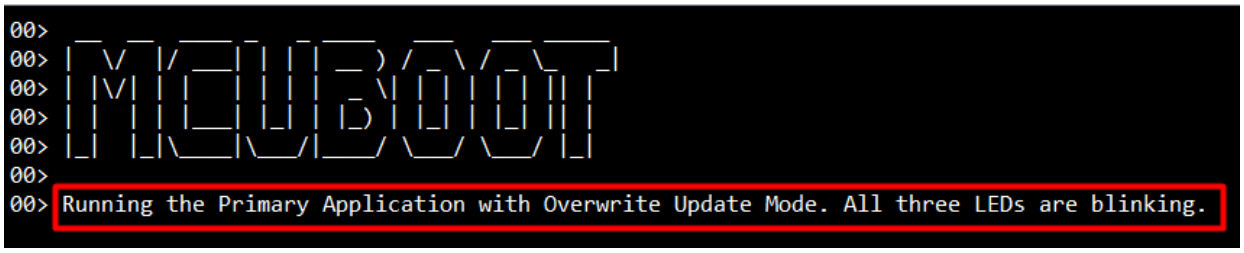

Figure 72. RTT Viewer Output from the Primary Application

7. Mastering and Delivering a New Application

This section provides instructions on how to master and deliver a new application that will be loaded into the Secondary image slot.

Note that the example bootloader, the example Primary application as well the example secondary applications are provided in the `RA2_Secure_Bootloader_using_Ocrypto.zip`. You can also follow section 8 to exercise these projects without going through the new application creation and mastering process described in this section if desired.

7.1 Create a New Application

The new application can be created by modifying the existing application. Import the initial project to the same workspace and rename the new project.

Click **File** > **Import…** and select **General > Rename & Import Existing C/C++ Project into Workspace**.

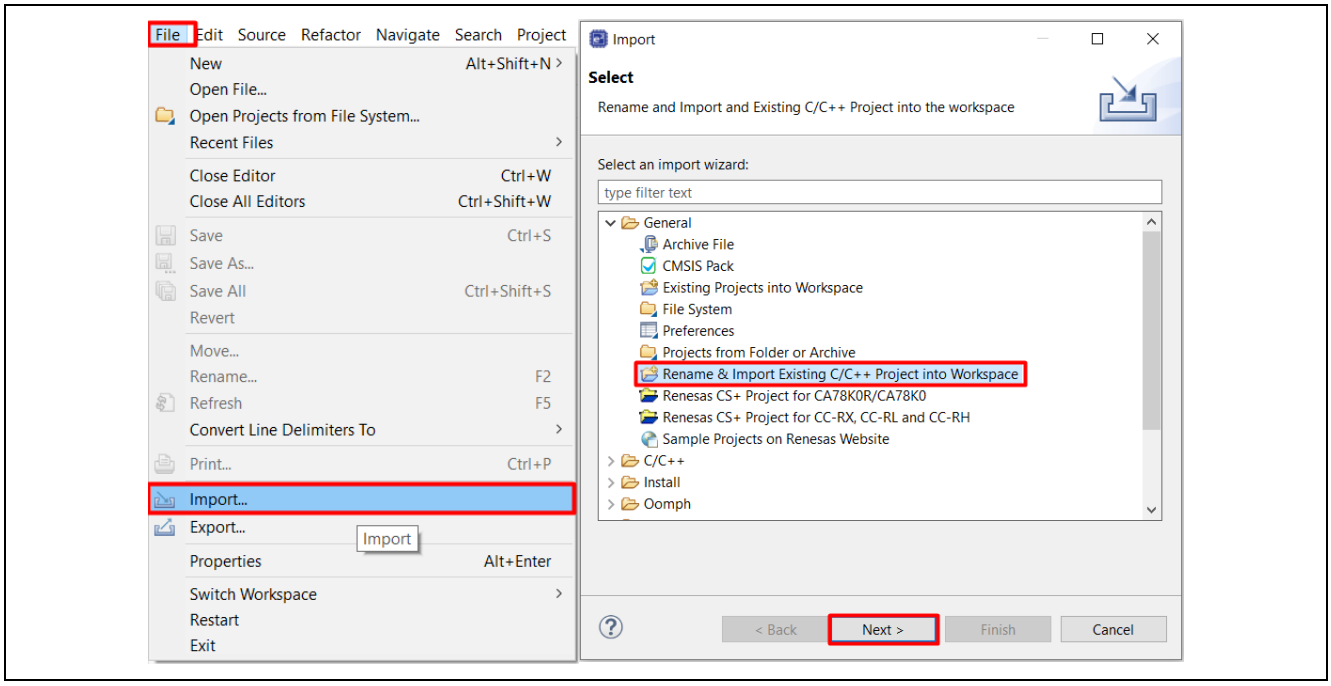

Figure 73. Select Rename and Import the Primary Application

Click **Next**, once the **Import** window opens, name the project and click **Browse** for **Select root directory** as shown in Figure 74.

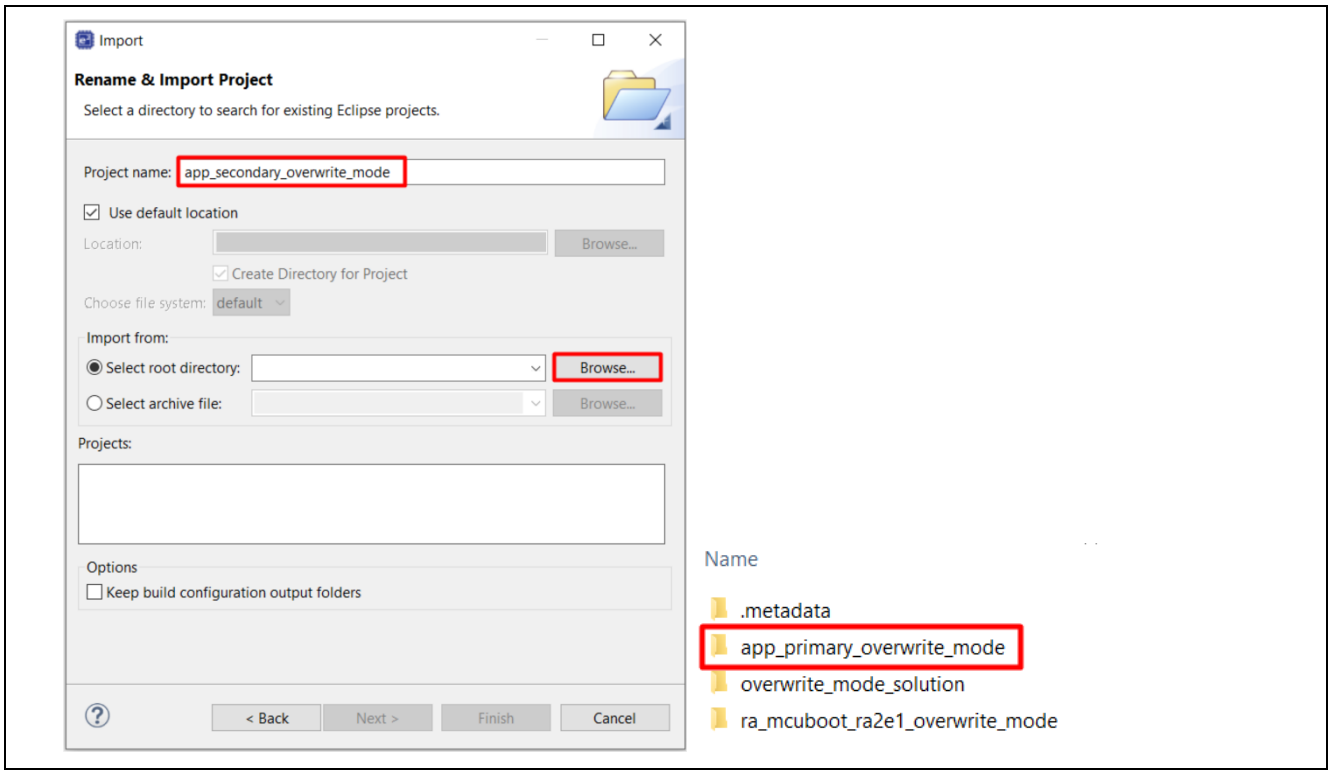

Figure 74. Rename the Project

Name the new project based on Table 5.

Table 5. Project Naming

Bootloader Project Name	Initial Application Project Name	New Application Project Name
`ra_mcuboot_ra2e1__overwrite_mode`	`app_primary_overwrite_mode`	`app_secondary_overwrite_mode`
`ra_mcuboot_ra2e1_swap_mode`	`app_primary_swap_mode`	`app_secondary_swap_mode`

Figure 75 is an example screenshot when importing the secondary project as **app_secondary_overwrite_mode**.

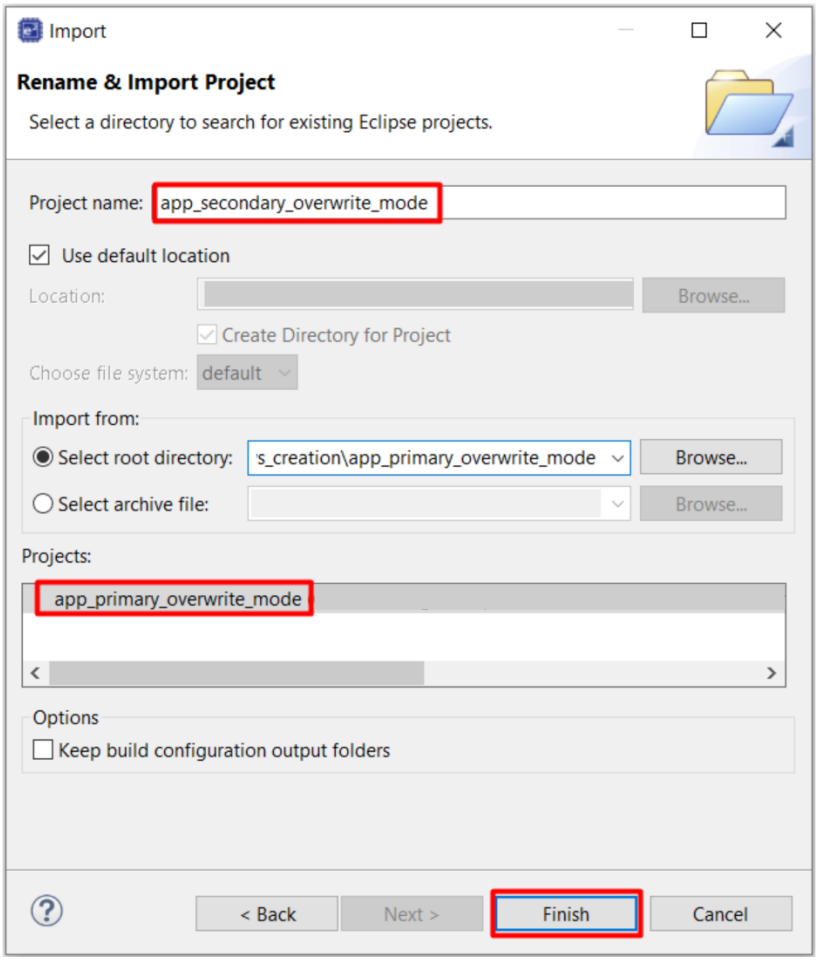

Figure 75. Import Secondary Project as app_secondary_overwrite_mode

Click **Finish,** and the new application project will be created.

Update Existing Application to a New Application

To demonstrate that the application is updated, portions of the code can be updated, for example:

- Update the application to blink one blue LED only.
- Update the RTT Viewer message to show this is the update image.

For simplicity, users can unzip `RA2_Secure_Bootloader_using_Ocrypto.zip`, open the `ra2e1_overwrite_update_mode\app_secondary_overwrite_mode\src` folder and copy all files under `\src` to the `\src` folder for the newly established project.

When importing the primary application, the **Build Variable** and the **Environment Variables** as well as the Debug configurations are automatically imported.

Click **Generate Project Content** and compile the new application. The signed binary for the new application is now created. In this example, `app_secondary_overwrite_mode.bin.signed` will be created.

For cleanness of the project, users can delete the `.jlink` file of the old project under the root of the newly created project structure.

Debug the New Application

To boot the new image, there is no need to update the debug configuration.

However, in most cases, user needs to debug the new application. It is recommended that the user debug the new application as a primary application, which means to initiate the debug process using the debug configuration of the new application. To debug the new image as a primary image, we need to update the debug configuration of the newly created application to use the signed binary of the `app_secondary_overwrite_mode` application rather than the signed binary of the `app_primary_overwrite_mode` application.

For example, when using the application projects for the `ra_mcuboot_ra2e1_overwrite_mode`, we want to change the debug configuration of the `app_secondary_overwrite_mode` project from the imported result shown in Figure 77.

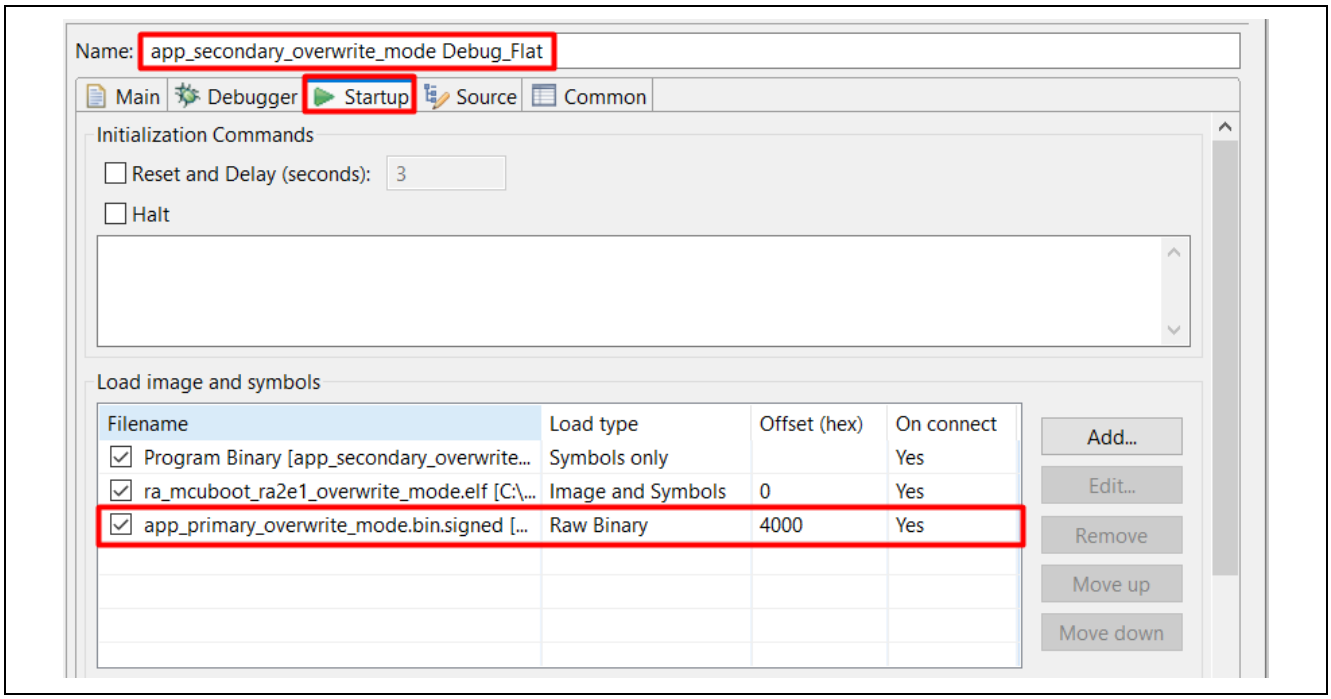

Figure 76. Debug Configuration of app_primary_overwrite_mode imported

In the imported configuration, the signed binary of the primary project is used. We need to change that to the signed binary of `app_secondary_overwrite_mode` as shown in Figure 77.

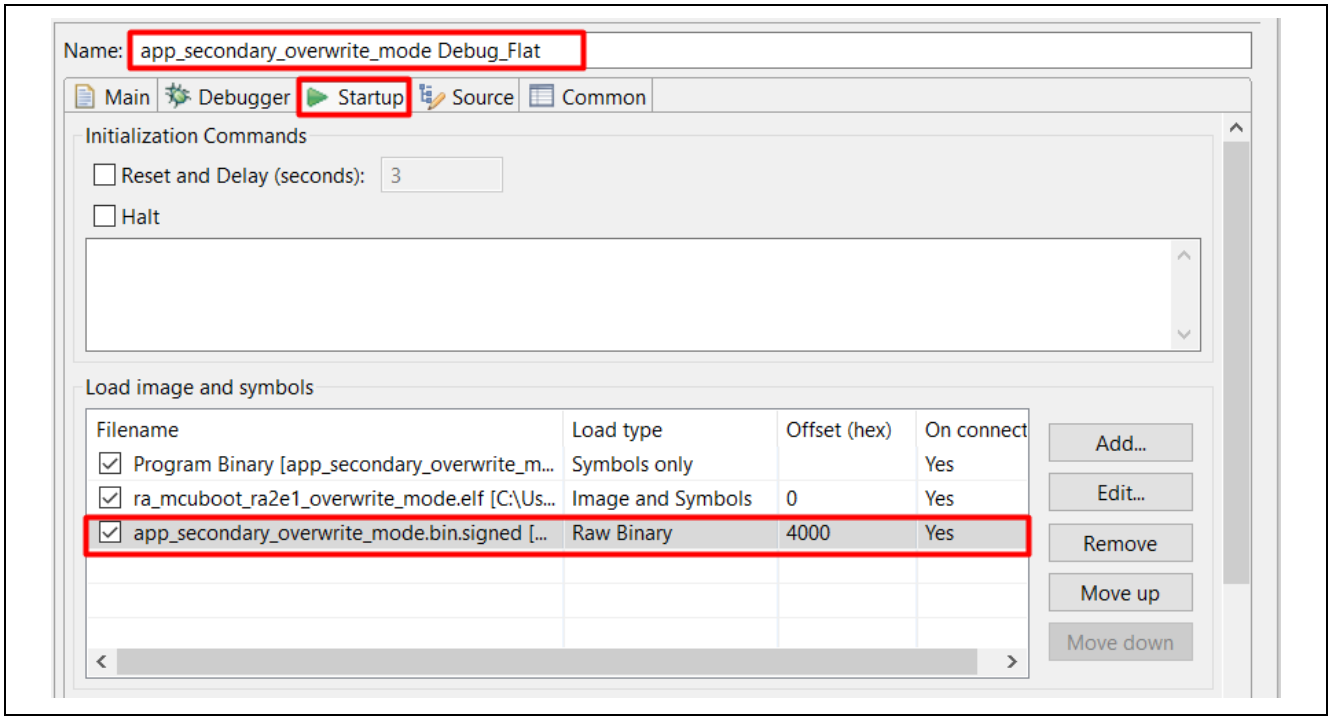

Figure 77. Debug Configuration of app_secondary_overwrite_mode to use for debugging

Note that in order to debug the new image as a primary image, for overwrite and swap mode, we want to set the download address of the signed new image binary to the location of the primary slot.

To create a brand-new application when using the overwrite, swap or Direct XIP upgrade mode without importing the previous application, users can follow section 4 to configure the application to use the bootloader and to sign the application image.

7.2 Configure the Swap Test Mode

Prior to introducing the swap test mode, it helps to introduce the **image_ok** byte as part of the application image trailer. The **image_ok** byte resides in the image trailer area. It is a flag byte that is used in Swap and Direct XIP upgrade modes. This byte is used to determine whether the new image will be swapped or not after the next reset following an image update. Please refer to Figure 18 for the location of the image trailer and the **image_ok** byte.

When using the Swap update mode, after the new image is loaded to the Secondary slot and authenticated successfully, the old image and the new image are swapped. At the next system reset, the system behavior differs depending on whether the **image_ok** byte which resing in the primary slot is **0x01** or **0xFF**.

If the **image_ok** byte is **0x01**, after the next reset, there will be no swapping, and hence the new image still stays in the Primary Slot and will be booted.

If the **image_ok** byte is **0xFF**, after the next reset, the new image and the old image is swapped again and the old image is booted. This is the rollback feature of swap mode.

Setting the image in the Primary slot as Confirmed can be achieved at the new image compile time or runtime.

This is explained in section 7.2.1 and 7.2.2.

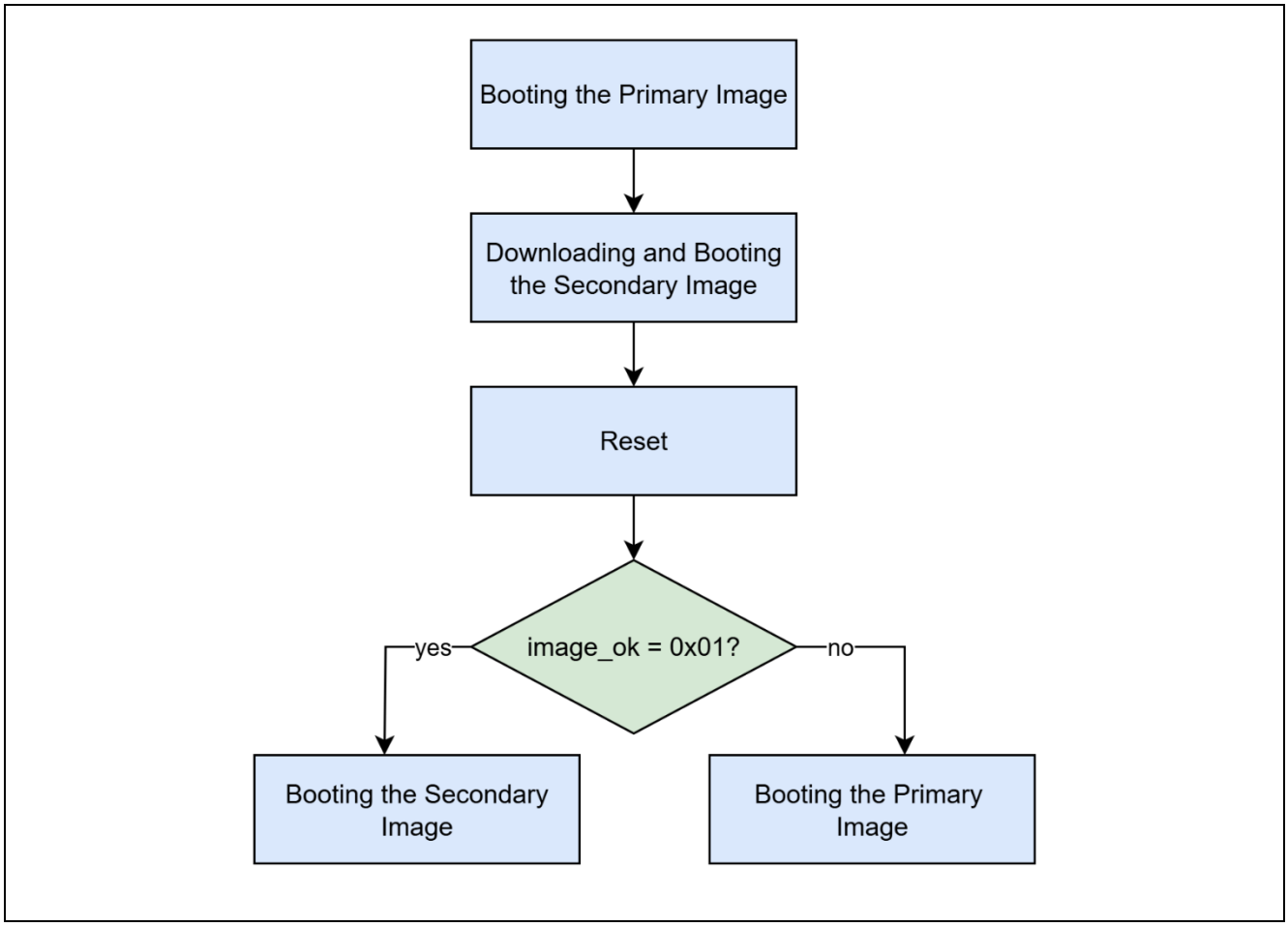

Figure 78. Swap Test Mode

7.2.1 Confirming the New Application at Compile Time

Confirming the new image (which will be loaded to the primary slot) at compile time requires setting the Custom Signing Options to `--confirm`, as shown in Figure 79.

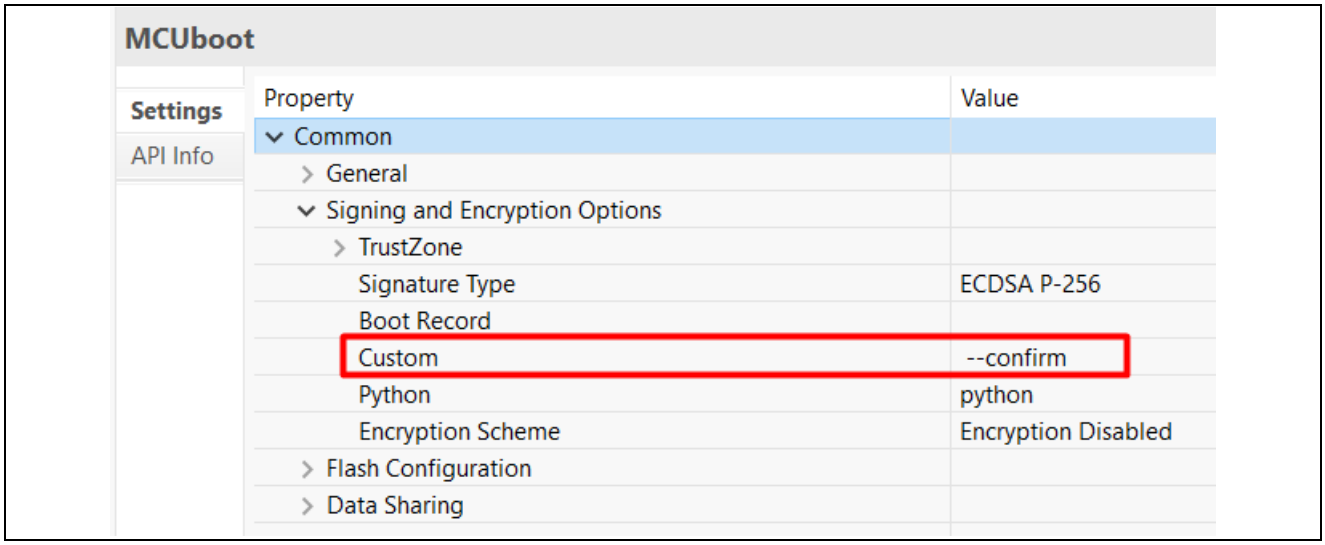

Figure 79. Set the Custom Signing Options to --confirm with swap update mode

7.2.2 Confirming the New Application at Run-time

Confirming the new application at runtime requires the bootloader to use `--pad` for the Custom signing command as shown in Figure 80.

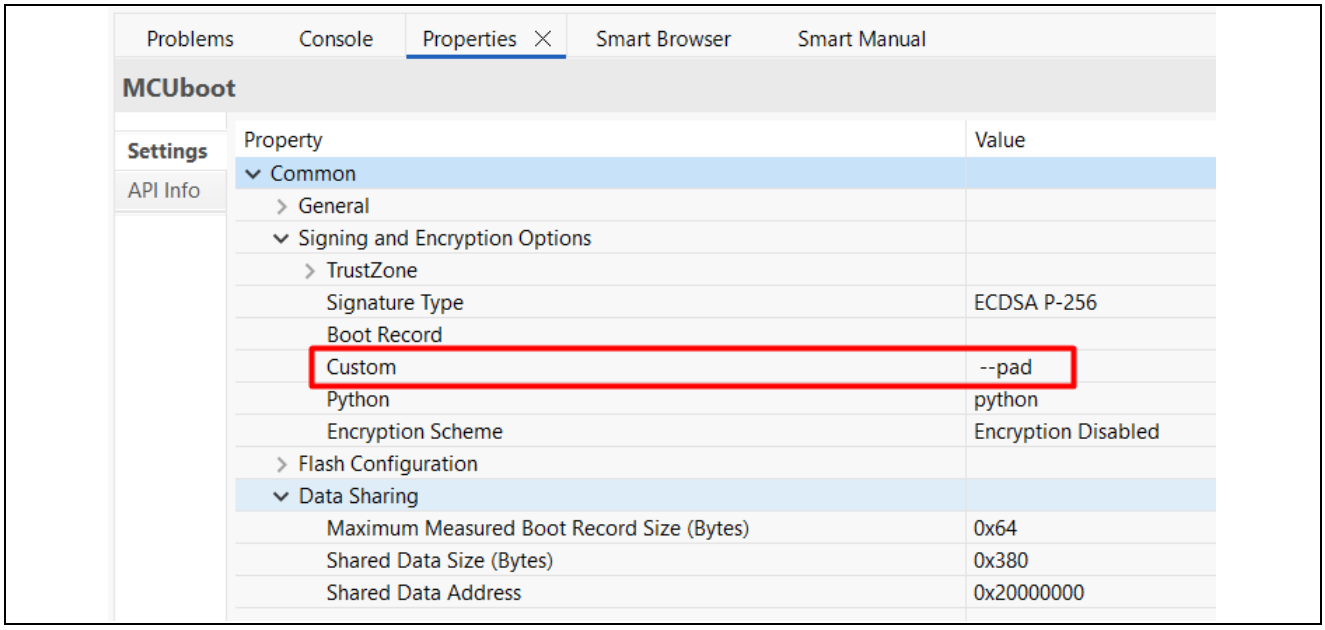

Figure 80. Set the Custom Signing Options to `--pad` with swap update mode

In addition, confirming the new image at runtime requires the **MCUboot Image Utilities** module to be included in the new application image and configure the system to use several files from the bootloader project. The example project demonstrates this feature. This module is included in the example bootloader `ra_mcuboot_ra2e1_swap_mode`.

Open the Secondary Application project `app_secondary_swap_mode` , and navigate to the **Stacks** tab, click **New stack > Bootloader > MCUboot Image Utilities**. Then, configure the properties of **MCUboot Image Utilities** module as shown in Figure 82. Adding this module adds about 2 KB of flash usage in the application.

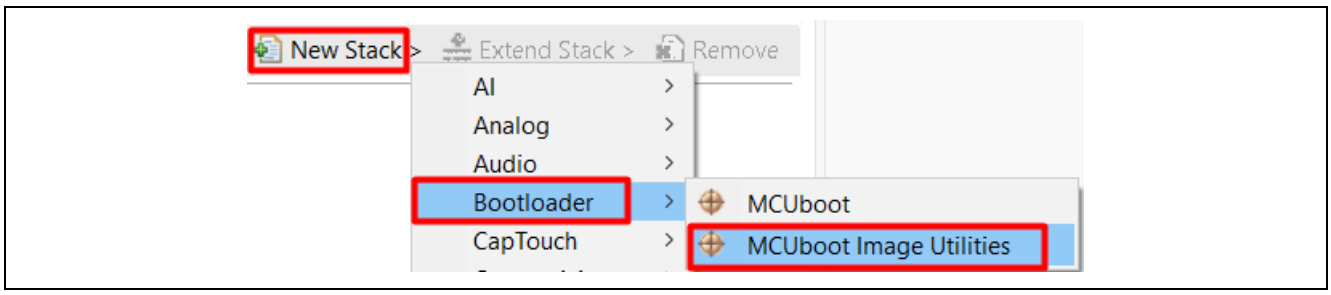

Figure 81. Include the MCUboot Image Utilities Module

Configure the path of the header files needed.

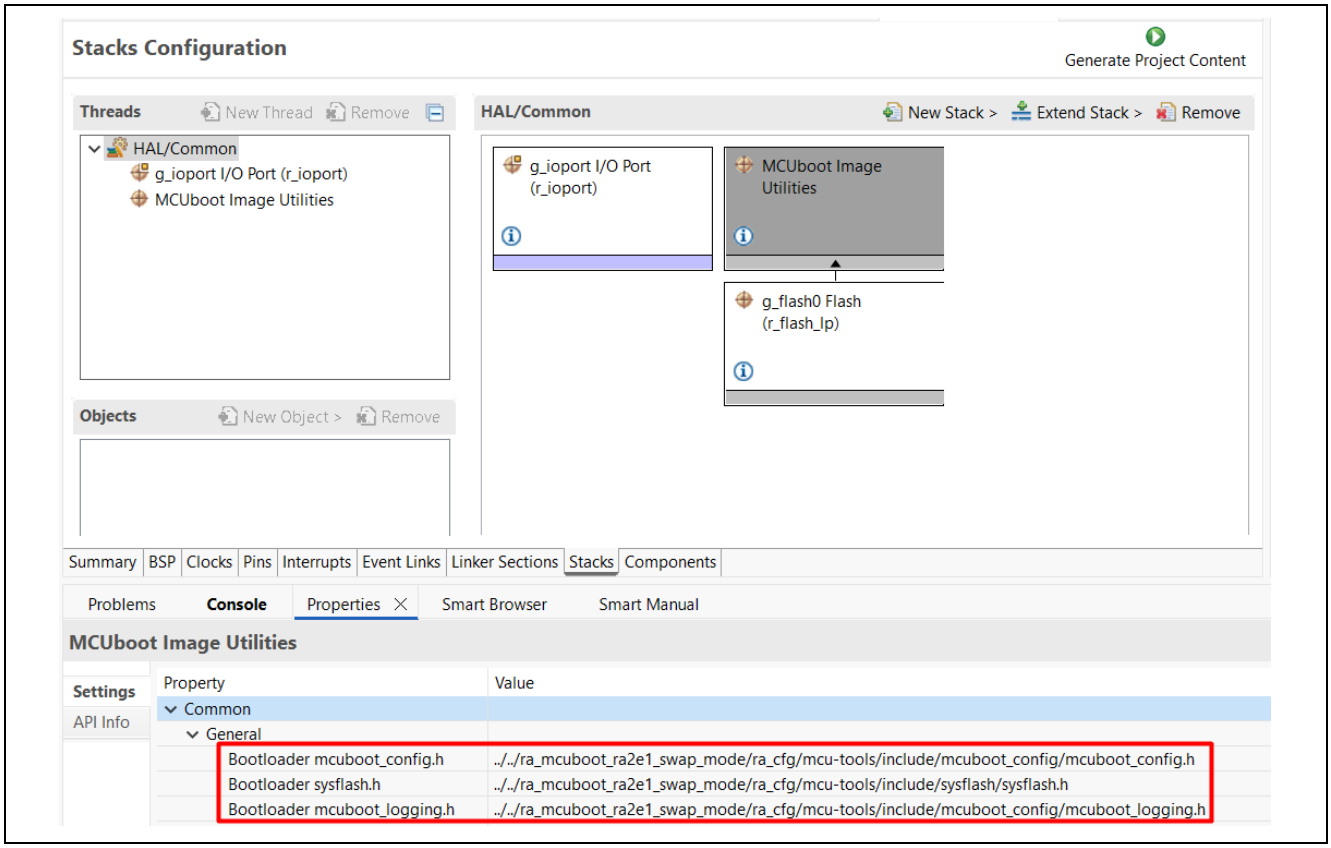

Figure 82. Include the Bootloader Header Files

Next configure the **r_flash_lp** module in the same way as in Figure 24.

In the secondary application project, insert the following function call to activate the confirmation of the application image. This function call can be added at a user chosen location after the desired testing of the application project is finished.

In the included example project, this function is demonstrated in the `hal_entry()` function located in `\ra2e1_swap_update_mode\app_secondary_swap_mode\hal_entry.c`.

```
/* Confirm the image in the primary slot.
* This is required after a test update in swap mode.
* This makes the swap permanent, and prevents MCUboot from reverting to the previous image at the next reset.
*/
assert(0 == boot_set_confirmed());
```

Figure 83. Confirm the Update Image

7.3 Downloading and Booting the New Application

Assume the Primary application blinky is now up and running and the three LEDs are blinking.

For testing purpose, user can click **Pause** and use the **Ancillary Download** button (which is available under the e2studio Debug view) to load the compiled Secondary Application `app_secondary_overwrite_mode.bin.signed`.

Select the new application image and set the download address. The download address depends on the bootloader flash memory allocation.

The download address should be the sum of Bootloader Flash Area Size + Image 1 Flash Area Size based on update mode shown in Figure 84.

For example, for the overwrite update mode, `ra_mcuboot_ra2e1_overwrite_mode`, the download address should be **0x12000**.

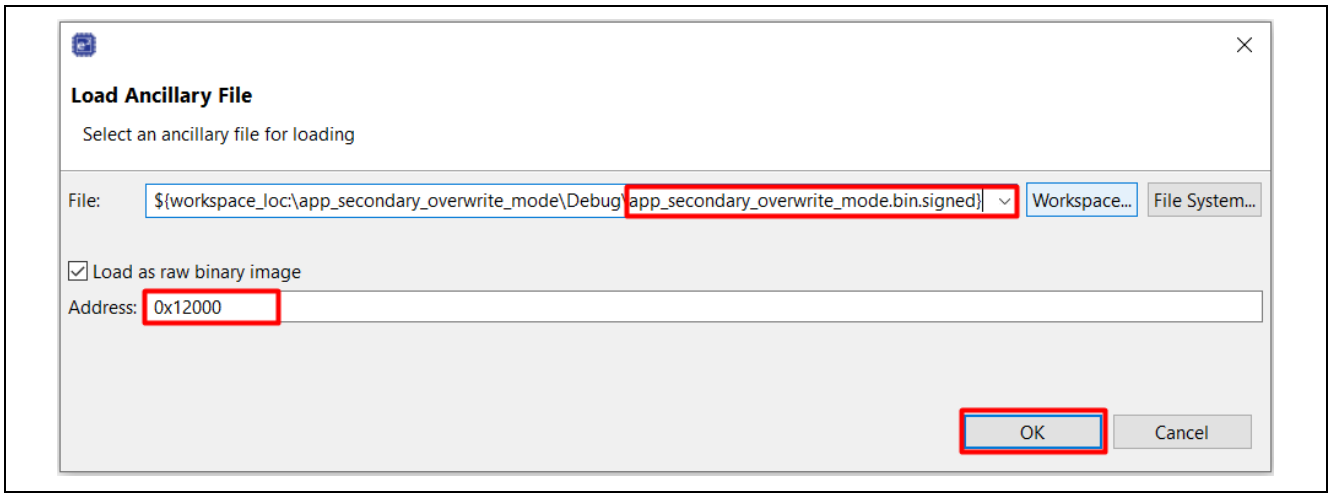

Figure 84. Download the Secondary Application Image in Overwrite Mode

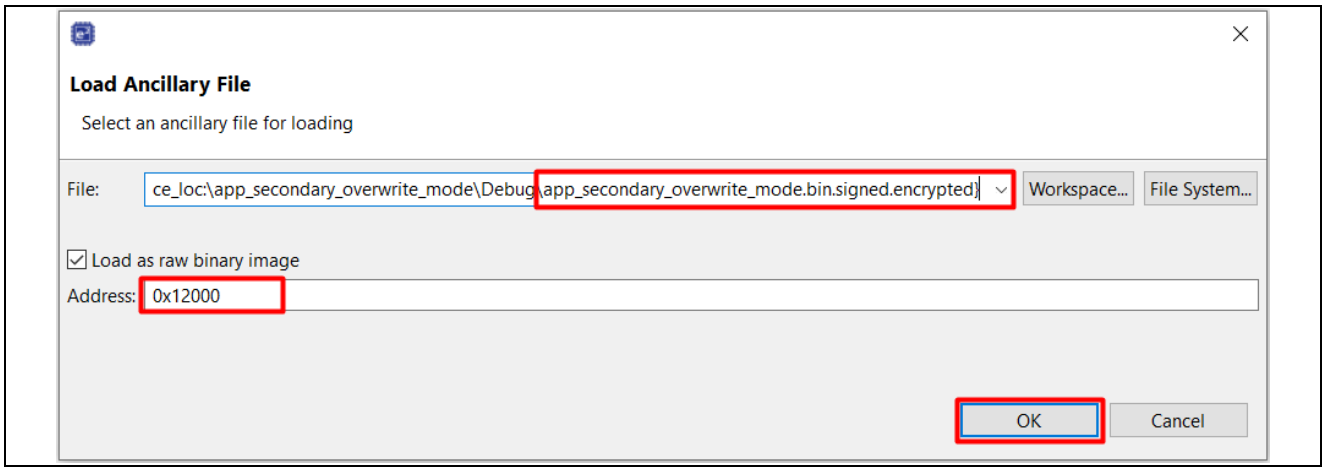

Figure 85. Download the Encrypted Secondary Application Image in Overwrite Mode

Note that for user-created customized applications, the download address needs to be adjusted by referencing the specific flash layout. Users can refer to Figure 56 to learn how to come up the download address.

Notes on using the Load Ancillary File Download

When we use the **Load Ancillary File** to download a new image during a debug session, the GDB server reconnects with the target, downloads the image, and restarts the debug session as shown in the following Console window output.

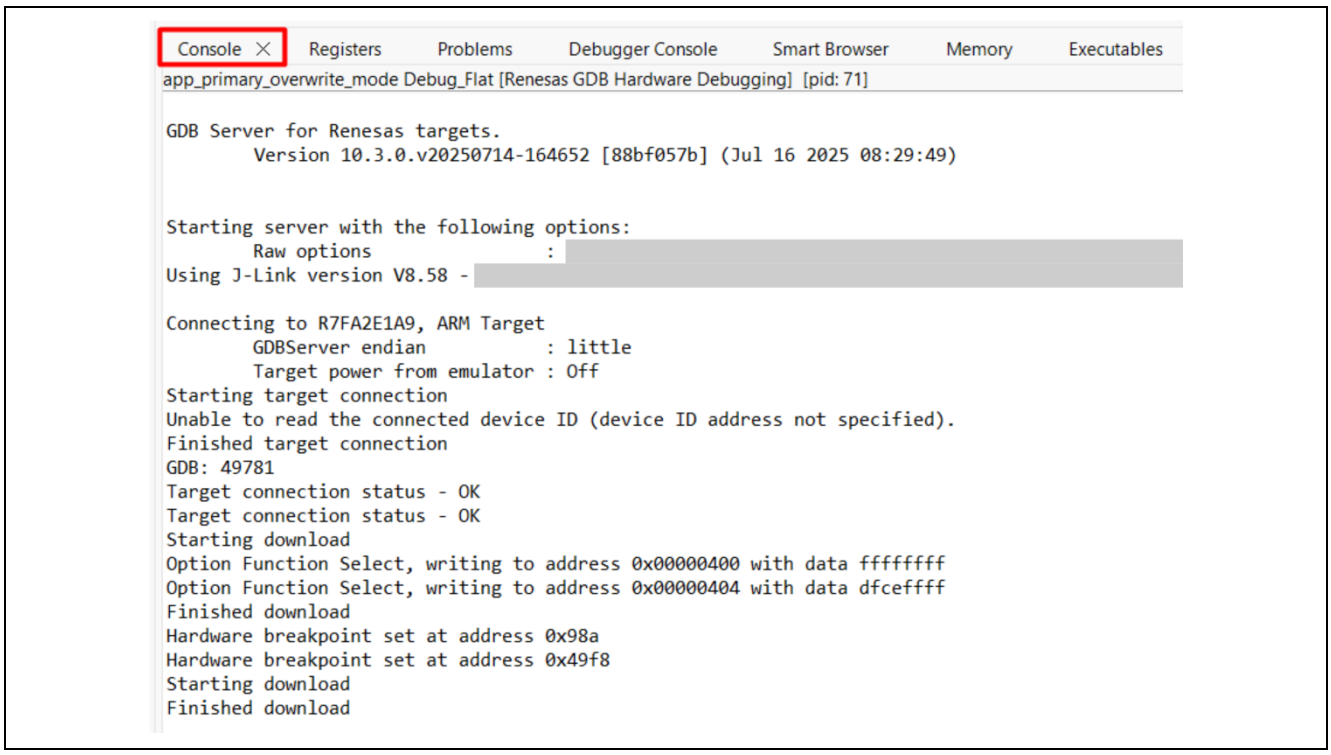

Figure 86. GDB Actions when using the Load Ancillary File Button

After the new image is downloaded and the GDB debug session is restarted, user can click **Resume** to allow the system to perform image overwrite and the new image will be booted. Only the blue LED should be blinking now, which indicates the new image is flashed to the Primary slot of the application area.

On the RTT Viewer, information on the secondary application execution is displayed. Below is an example when `app_secondary_overwrite_mode` is booted.

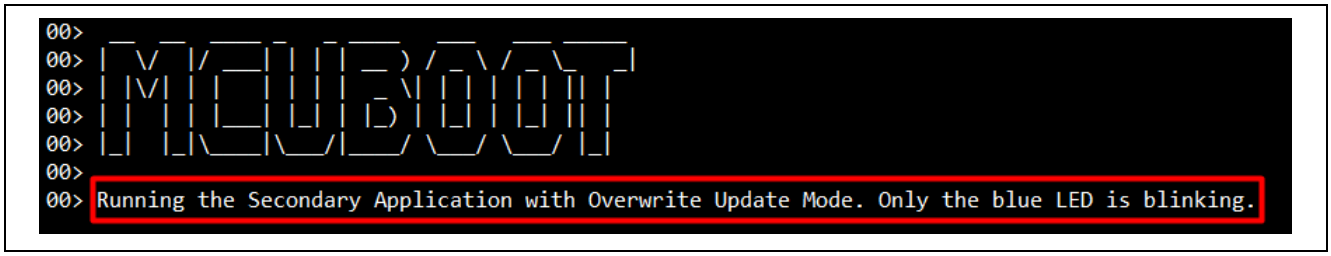

Figure 87. RTT Viewer Output from the New Application

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

Unzip `RA2_Secure_Bootloader_using_Ocrypto.zip` to access the included bootloader and example application projects.

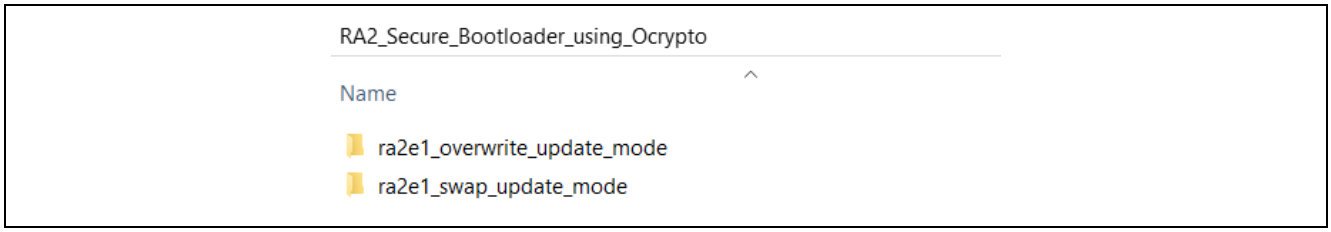

Figure 88. Example Projects Included

8.1 Running the EK-RA2E1 Overwrite Update Mode

Follow the steps below to run the example projects under folder `\ra2e1_overwrite_update_mode`:

1. Import projects to a workspace.
2. Right-click on the **Solution Project**: `overwrite_mode_solution`. Select **Build Project**. This command will build all projects within the **Solution Project**.
3. Select **Build Project** for the Secondary Application: `app_secondary_overwrite_mode`.
4. Make sure that `app_primary_overwrite_mode.bin.signed` and `app_secondary_overwrite_mode.bin.signed` files are generated in the `Debug` folder.
5. Initialize the RA2E1 MCU (see section 6.2 for reference).
6. Debug the application from project `app_primary_overwrite_mode`.
7. Resume the program execution twice. All LEDs should be blinking.
8. Pause the execution.
9. Download the `app_secondary_overwrite_mode.bin.signed` using **Load Ancillary File** to address **0x12000**.
10. Resume the program execution. The blue LED should be blinking.

8.2 Running the EK-RA2E1 Swap Update Mode

Follow the steps below to run the example projects under folder `\ra2e1_swap_update_mode`:

1. Import projects to a workspace.
2. Right-click on the **Solution Project**: `swap_mode_solution`. Select **Build Project**. This command will build all projects within the **Solution Project**.
3. Select **Build Project** for the Secondary Application: `app_secondary_swap_mode`.
4. Make sure that `app_primary_swap_mode.bin.signed` and `app_secondary_swap_mode.bin.signed` files are generated in the `Debug` folder.
5. Initialize the RA2E1 MCU (see section 6.2 for reference).
6. Debug the application from project `app_primary_swap_mode`.
7. Resume the program execution twice. All LEDs should be blinking.
8. Pause the execution.
9. Download the `app_secondary_swap_mode.bin.signed` using **Load Ancillary File** to address **0x12000**.
10. Resume the program execution. The blue LED should be blinking.

9. Production Support Considerations

9.1 Using Custom Signing Key and Encryption Key

In this section, users will generate two sets of **ECDSA SECP256R1** keys using the `imgtool.py` tool included with MCUboot.

The **MCUboot Example Keys** stack imports the example keys included in the MCUboot public port to use in the image signing/verifying. The custom keys generated in this section replace these example keys.

The `root_pub_der` array is the public key for image verification which is located in `\{bootloader project}\ra\mcu-tools\MCUboot\sim\mcuboot-sys\csupport\keys.c`.

For **ECDSA P-256**, the public key for **image verification** is shown as the following (from `keys.c`).

```
const unsigned char root_pub_der[] = {
    0x30, 0x59, 0x30, 0x13, 0x06, 0x07, 0x2a, 0x86,
    0x48, 0xce, 0x3d, 0x02, 0x01, 0x06, 0x08, 0x2a,
    0x86, 0x48, 0xce, 0x3d, 0x03, 0x01, 0x07, 0x03,
    0x42, 0x00, 0x04, 0x2a, 0xcb, 0x40, 0x3c, 0xe8,
    0xfe, 0xed, 0x5b, 0xa4, 0x49, 0x95, 0xa1, 0xa9,
    0x1d, 0xae, 0xe8, 0xdb, 0xbe, 0x19, 0x37, 0xcd,
    0x14, 0xfb, 0x2f, 0x24, 0x57, 0x37, 0xe5, 0x95,
    0x39, 0x88, 0xd9, 0x94, 0xb9, 0xd6, 0x5a, 0xeb,
    0xd7, 0xcd, 0xd5, 0x30, 0x8a, 0xd6, 0xfe, 0x48,
    0xb2, 0x4a, 0x6a, 0x81, 0x0e, 0xe5, 0xf0, 0x7d,
    0x8b, 0x68, 0x34, 0xcc, 0x3a, 0x6a, 0xfc, 0x53,
    0x8e, 0xfa, 0xc1, };
const unsigned int root_pub_der_len = 91;
```

Figure 89. Public Key used for Image Verification (from keys.c)

The private key for **image decryption** is shown as the following (from `keys.c`).

```
unsigned char enc_key[] = {
  0x30, 0x81, 0x43, 0x02, 0x01, 0x00, 0x30, 0x13, 0x06, 0x07, 0x2a, 0x86,
  0x48, 0xce, 0x3d, 0x02, 0x01, 0x06, 0x08, 0x2a, 0x86, 0x48, 0xce, 0x3d,
  0x03, 0x01, 0x07, 0x04, 0x29, 0x30, 0x27, 0x02, 0x01, 0x01, 0x04, 0x20,
  0xf6, 0x1e, 0x51, 0x9d, 0xf8, 0xfa, 0xdd, 0xa1, 0xb7, 0xd9, 0xa9, 0x64,
  0x64, 0x3b, 0x54, 0xd0, 0x3d, 0xd0, 0x1f, 0xe5, 0x78, 0xd9, 0x17, 0x98,
  0xa5, 0x28, 0xca, 0xcc, 0x6b, 0x67, 0x9e, 0x06, 0xa1, 0x44,

};
static unsigned int enc_key_len = 70;
```

Figure 90. Private Key used for Image Decryption (from keys.c)

The matching private key for the public key `root_pub_der` is `root-ec-p256.pem`. We will generate a custom private key `ecc_sign_private.pem` to replace the usage of `root-ec-p256.pem` which is used in the image signing process.

Similarly, the matching public key for the private key `enc_key` is `enc-ec256-pub.pem`. We will generate a custom public key `ecc_enc_public.pem` to replace `enc-ec256-pub.pem` which is used in the image encryption process.

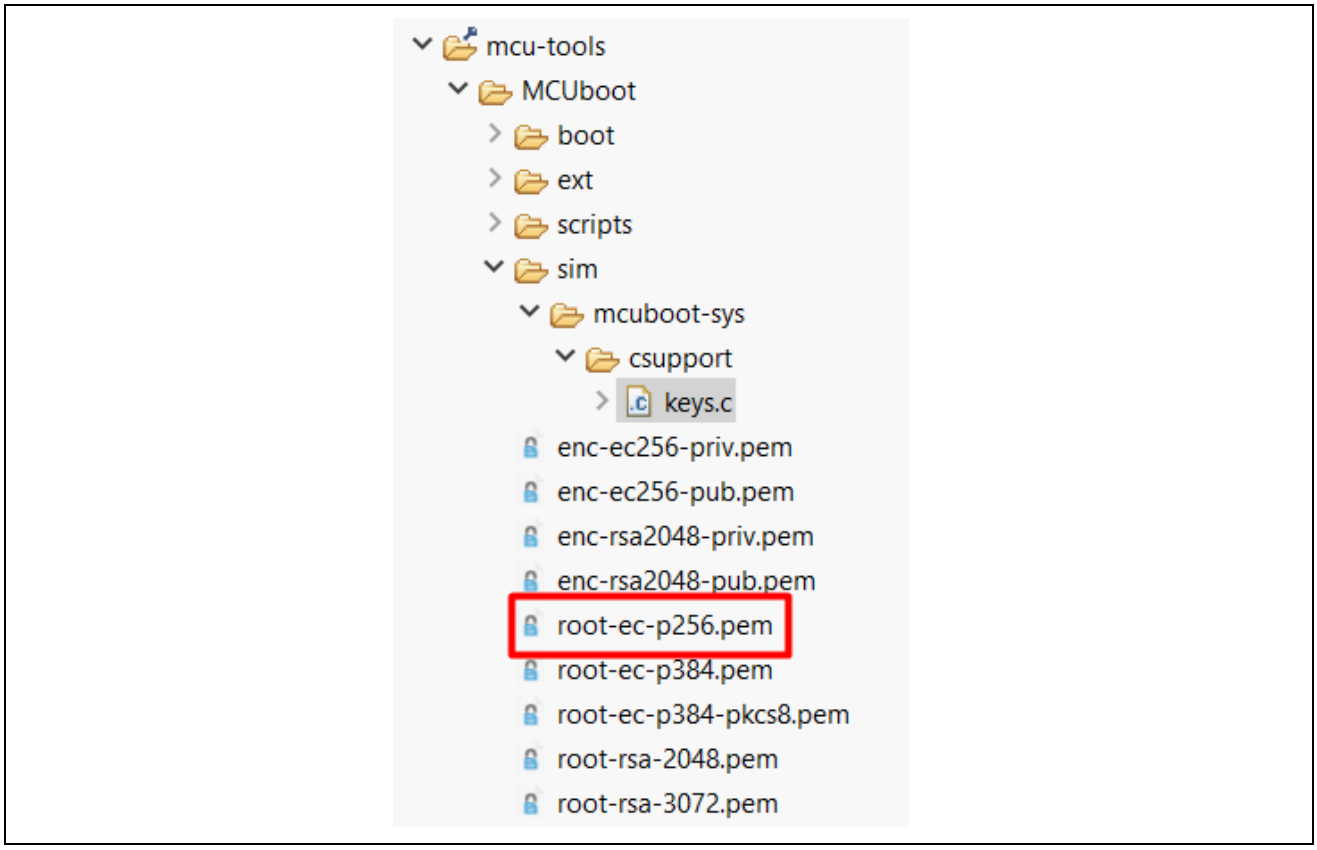

Figure 91. Example Image Signing Private Key

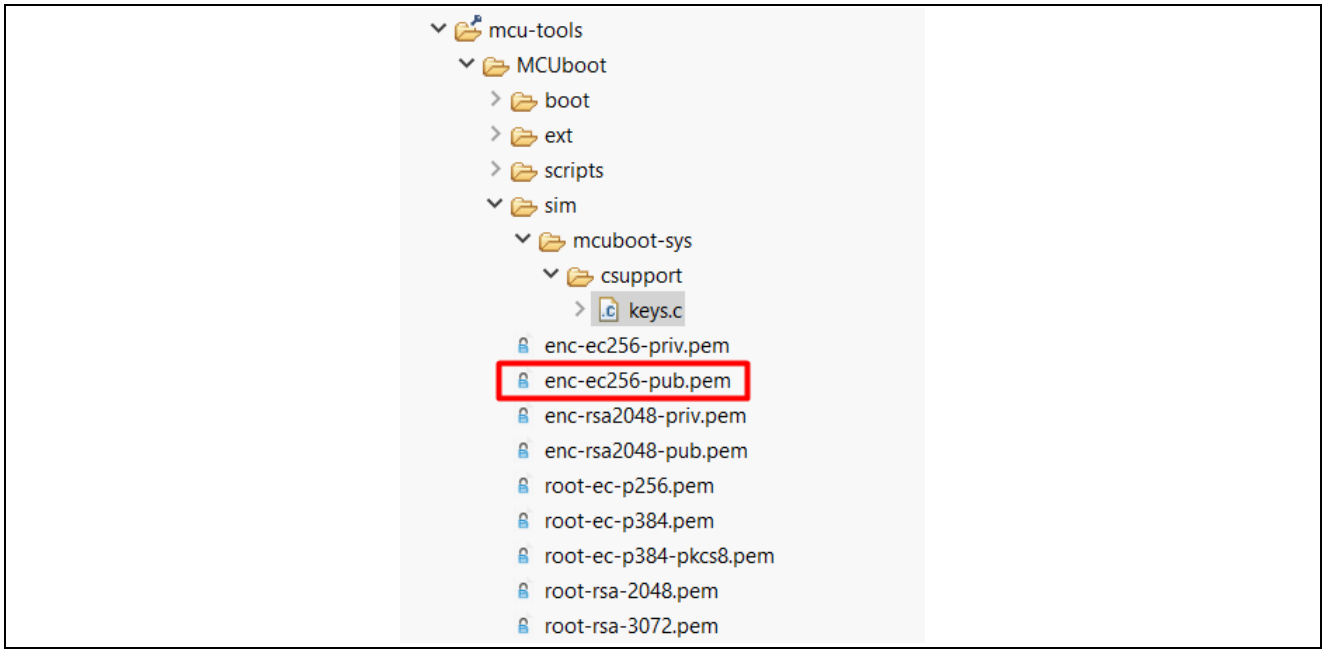

Figure 92. Example Image Encryption Public Key

Use the following steps to create and replace example keys generated by the MCUboot stack:

1. In the bootloader project, copy `keys.c` from the MCUboot folder to the `\src` folder of the bootloader project: `ra_mcuboot_ra2e1_overwrite_mode`.

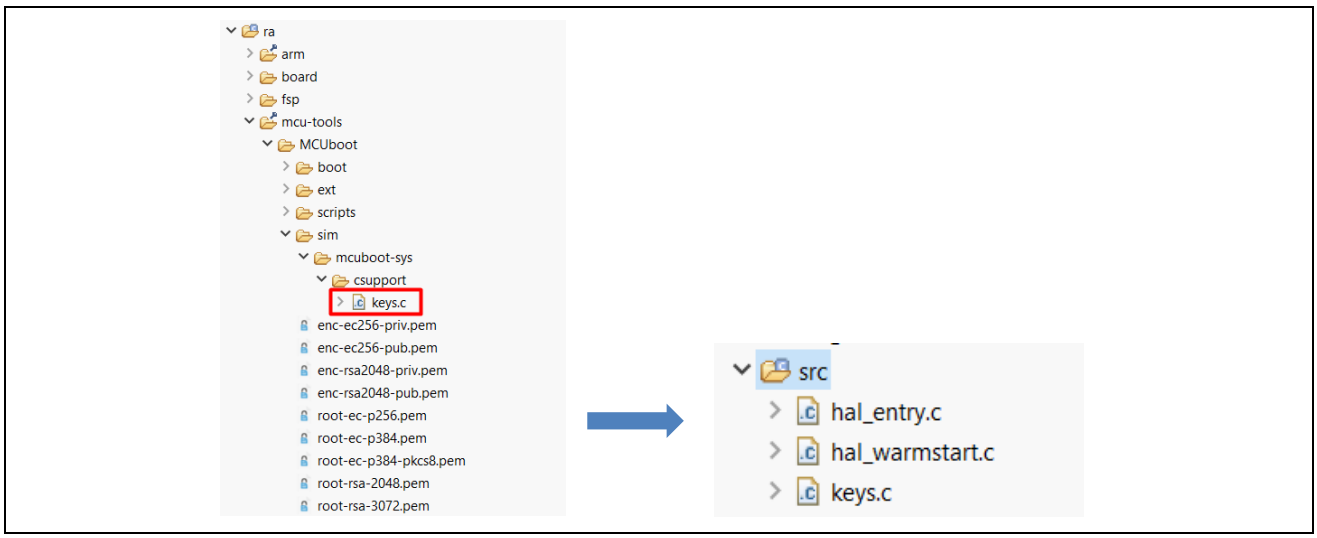

Figure 93. Copy the example keys.c

2. Open the configurator for the bootloader project, right click on **MCUboot Example Keys** stack and select **Delete**, and then click **Generate Project Content** to apply the updated configuration settings.

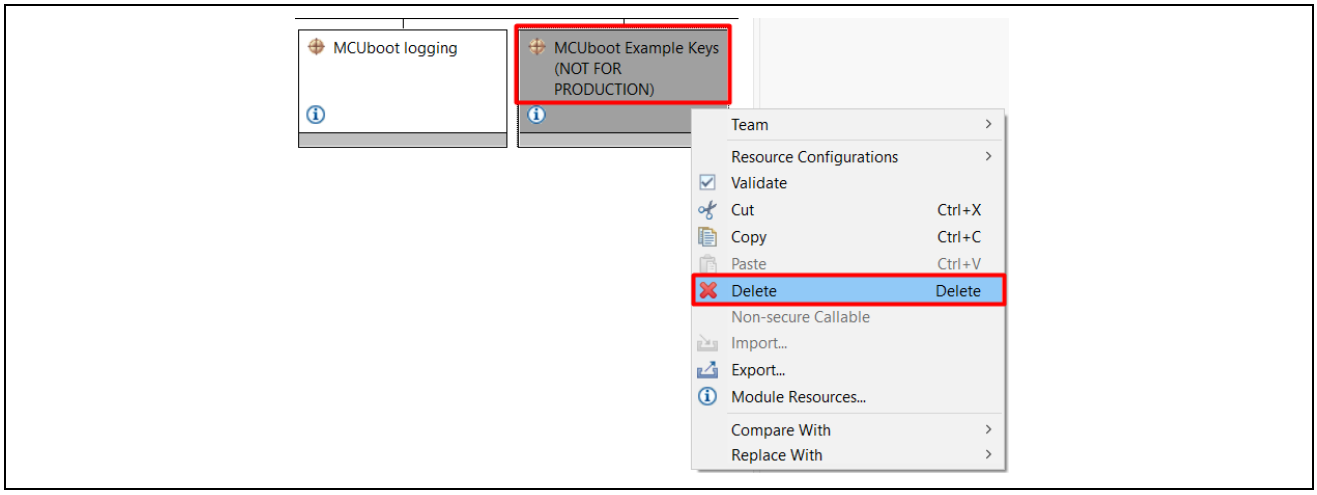

Figure 94. Delete the MCUboot Example Keys Stack

3. Extend the booloader project and navigate to foler `\ra\mcu-tools\MCUboot\scripts\`. Right click on this folder and select **Command Prompt**.

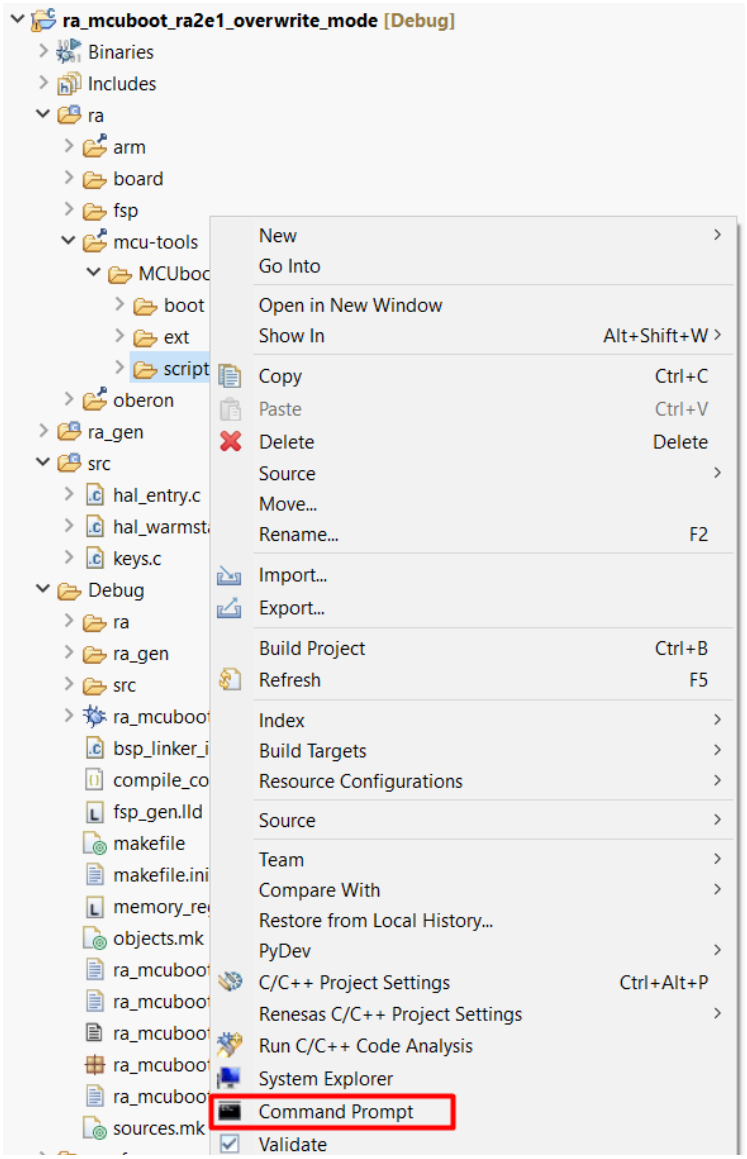

Figure 95. Start Command Prompt under the \MCUboot\scripts Folder

4. Under the command window, execute command:

```
python imgtool.py keygen -k ecc_sign_private.pem -t ecdsa-p256
```

5. Copy the generated `ecc_sign_private.pem` to folder `\ra_mcuboot_ra2e1_overwrite_mode\src`. This is new image signing key.

6. Extract the public key from `ecc_sign_private.pem`. Execute command:

```
python imgtool.py getpub -k ecc_sign_private.pem
```

```
te_mode\ra\mcu-tools\MCUboot\scripts>python imgtool.py getpub -k ecc_sign_private.pem
/* Autogenerated by imgtool.py, do not edit. */
const unsigned char ecdsa_pub_key[] = {
    0x30, 0x59, 0x30, 0x13, 0x06, 0x07, 0x2a, 0x86,
    0x48, 0xce, 0x3d, 0x02, 0x01, 0x06, 0x08, 0x2a,
    0x86, 0x48, 0xce, 0x3d, 0x03, 0x01, 0x07, 0x03,
    0x42, 0x00, 0x04, 0x68, 0xb3, 0xcf, 0xd9, 0x31,
    0xe5, 0x0a, 0xc9, 0xb2, 0xc5, 0xae, 0xc4, 0x63,
    0x70, 0xf9, 0x37, 0x37, 0x29, 0xa0, 0x59, 0xde,
    0xe6, 0x14, 0x23, 0x4c, 0xaa, 0xea, 0x61, 0x2b,
    0xbf, 0x66, 0x44, 0x2d, 0x12, 0x8a, 0x56, 0xdc,
    0xc5, 0xf4, 0xdc, 0xc9, 0x93, 0x64, 0x5d, 0xaf,
    0x3c, 0x80, 0x99, 0x91, 0x49, 0x26, 0xeb, 0x29,
    0x97, 0x8a, 0xe2, 0x8f, 0x27, 0x39, 0xb8, 0xb7,
    0x65, 0x6f, 0x5c,
};
const unsigned int ecdsa_pub_key_len = 91;
```

Figure 96. Generate ECDSA Public Key

7. Replace the `root_pub_der` array in `keys.c` (copied in step 1) with the content of `ecdsa_pub_key`.
8. Execute the following command to generate the ecc private key to be used in the application image encryption process:

   `python imgtool.py keygen -k ecc_enc_private.pem -t ecdsa-p256`
9. Copy the generated `ecc_enc_private.pem` to folder `\ra_mcuboot_ra2e1_overwrite_mode\src`. This is new image encryption key.
10. Extract the private key to include in the bootloader. Execute command:

    `python imgtool.py getpriv --minimal -k ecc_enc_private.pem`
    Remove superfluous fields from the ASN1 by passing it `--minimal`.

```
\MCUboot\scripts>python imgtool.py getpriv --minimal -k ecc_enc_private.pem
/* Autogenerated by imgtool.py, do not edit. */
const unsigned char enc_priv_key[] = {
    0x30, 0x41, 0x02, 0x01, 0x00, 0x30, 0x13, 0x06,
    0x07, 0x2a, 0x86, 0x48, 0xce, 0x3d, 0x02, 0x01,
    0x06, 0x08, 0x2a, 0x86, 0x48, 0xce, 0x3d, 0x03,
    0x01, 0x07, 0x04, 0x27, 0x30, 0x25, 0x02, 0x01,
    0x01, 0x04, 0x20, 0xc9, 0x1e, 0x9c, 0x37, 0xd4,
    0x36, 0x30, 0x9a, 0x95, 0x91, 0x91, 0x9c, 0xf9,
    0xc5, 0x31, 0x26, 0x9d, 0x94, 0x2a, 0xff, 0x8a,
    0xbd, 0x2d, 0xd1, 0x75, 0x52, 0x96, 0xd3, 0x86,
    0xe1, 0x35, 0xd9,
};
const unsigned int enc_priv_key_len = 67;
```

Figure 97. Generate the Private Key used for Image Encryption

11. Replace the `enc_key` array in `keys.c` (copied in step 1) with the content of `enc_priv_key`.

Note that users need to update the `enc_key_len` in `\src\keys.c` to match `enc_priv_key_len`, as shown in Figure 97.

12. Users will derive the encryption public key in pem format using the private key. Execute command:

    `python imgtool.py getpub -k ecc_enc_private.pem -e pem > ecc_enc_public.pem`

```
.\ra\mcu-tools\MCUboot\scripts>python imgtool.py getpub -k ecc_enc_private.pem -e pem > ecc_enc_public.pem
```

Figure 98. Generate the Public using the Private Key

13. Copy the generated `ecc_enc_public.pem` to the folder `\ra_mcuboot_ra2e1_overwrite_mode\src.`
14. To use the new image signing key, users need to update the signing key configuration of the application projects.

Choose **MCUBOOT_IMAGE_SIGNING_KEY** Variable, click **Edit** and define the **Value** as:
`${workspace_loc:ra_mcuboot_ra2e1_overwrite_mode}/src/ecc_sign_private.pem`

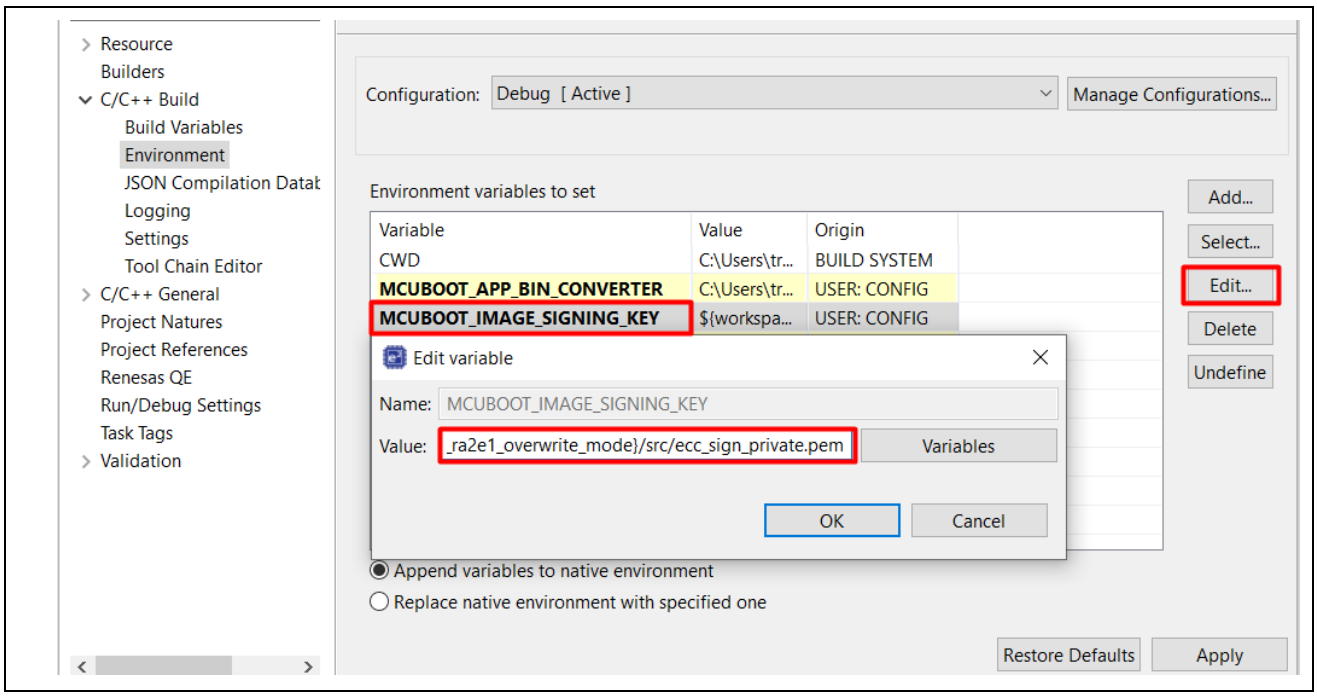

Figure 99. Configure the Application Project to use the Custom Image Signing

15. To use the new image encryption key, users need to update the encryption key configuration of the application projects.

Choose **MCUBOOT_IMAGE_ENC_KEY** Variable, click **Edit** and define the **Value** as:
`${workspace_loc:ra_mcuboot_ra2e1_overwrite_mode}/src/ecc_enc_public.pem`

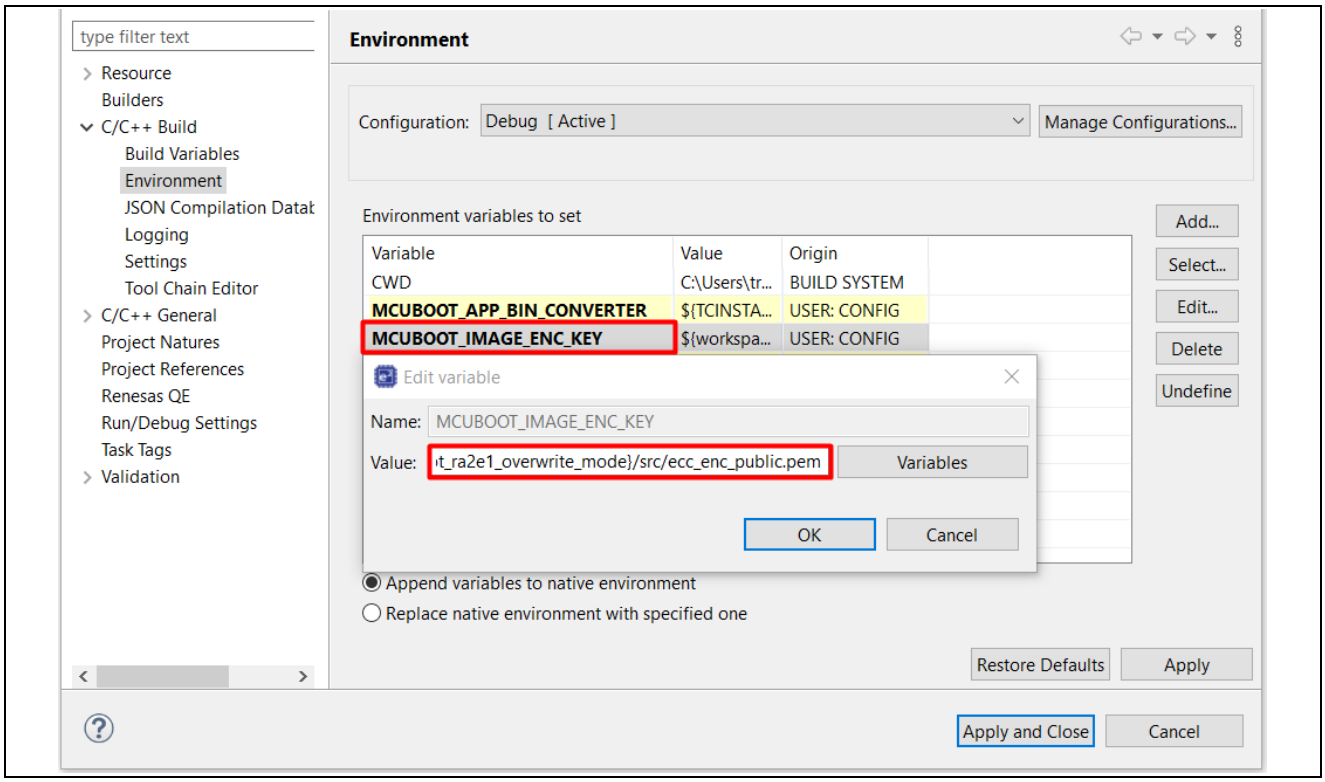

Figure 100. Configure the Application Project to use the Custom Key for the Image Encryption

16. Users need to navigate to the **Solution Project**: `overwrite_mode_solution`, and right-click it, and select **Build Project**. This action builds both the **Bootloader Project** and the **Primary Project**.
17. Select **Build Project** for the Secondary Application: `app_secondary_overwrite_mode`.

9.2 Provision the Bootloader and the Initial Application to MCU

In this section, the **Renesas Flash Programmer (RFP)** is used to provision the bootloader and the initial application onto the MCU, supporting users during the production phase.

Open the **Renesas Flash Programmer (RFP)** and set up the following configurations. Then click **Connect**.

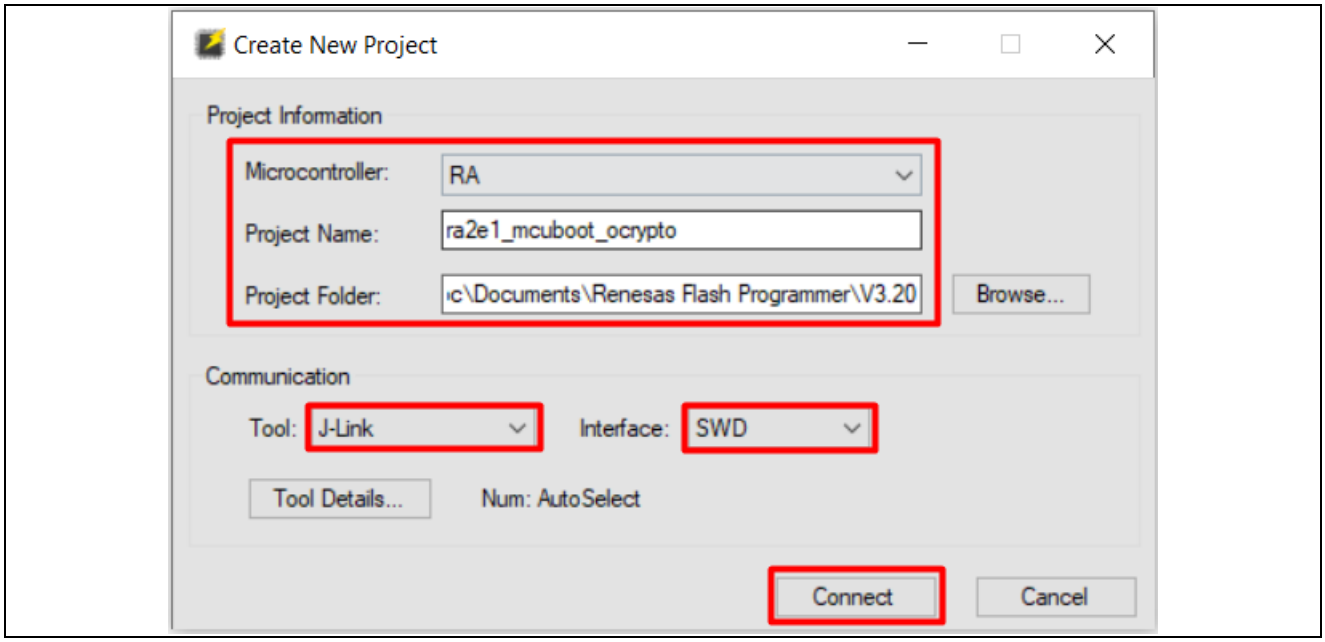

Figure 101. Lauch the RFP and configure the New Project

Select a program file, as shown in Figure 102.

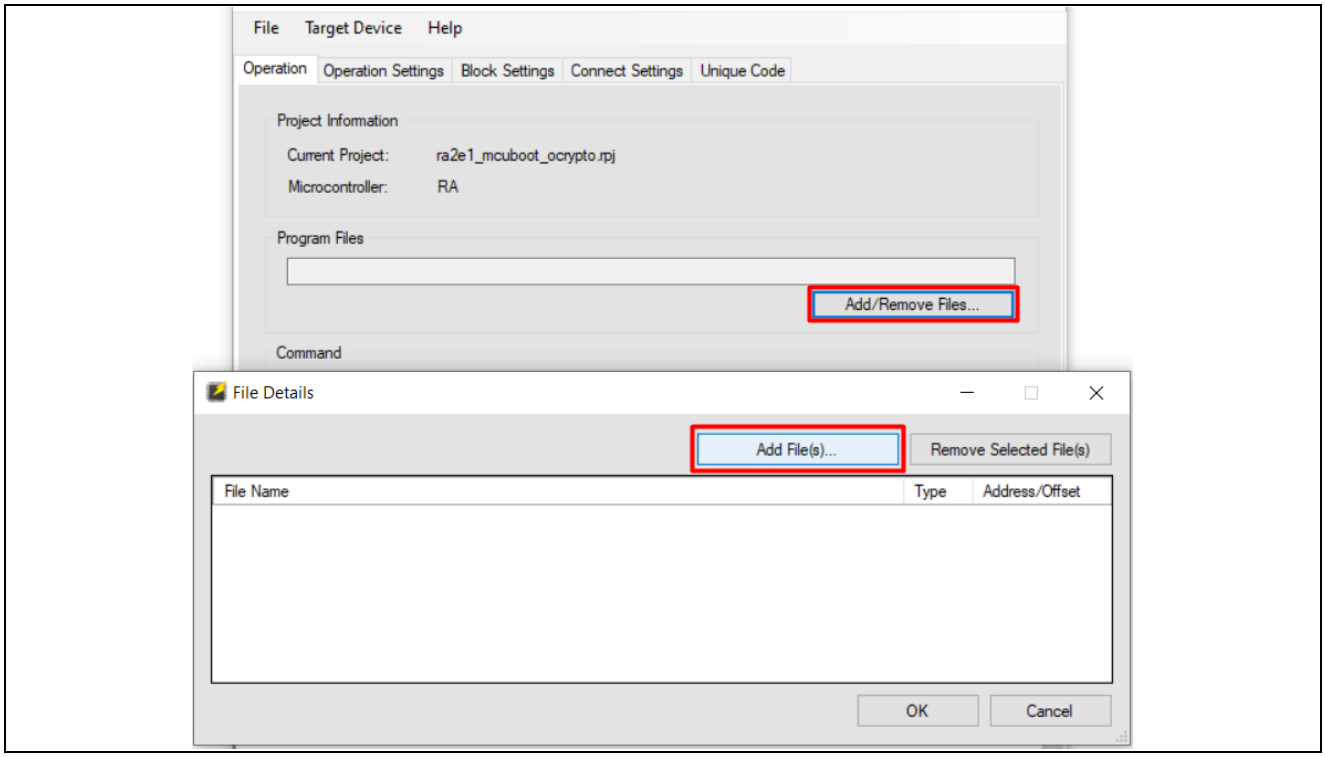

Figure 102. Select a program file to execute command

Next, select the `app_primary_overwrite_mode.bin.signed` file and assign the start address for the primary image.

For the bootloader image, users need to choose the `ra_mcuboot_ra2e1_overwrite_mode.srec`.

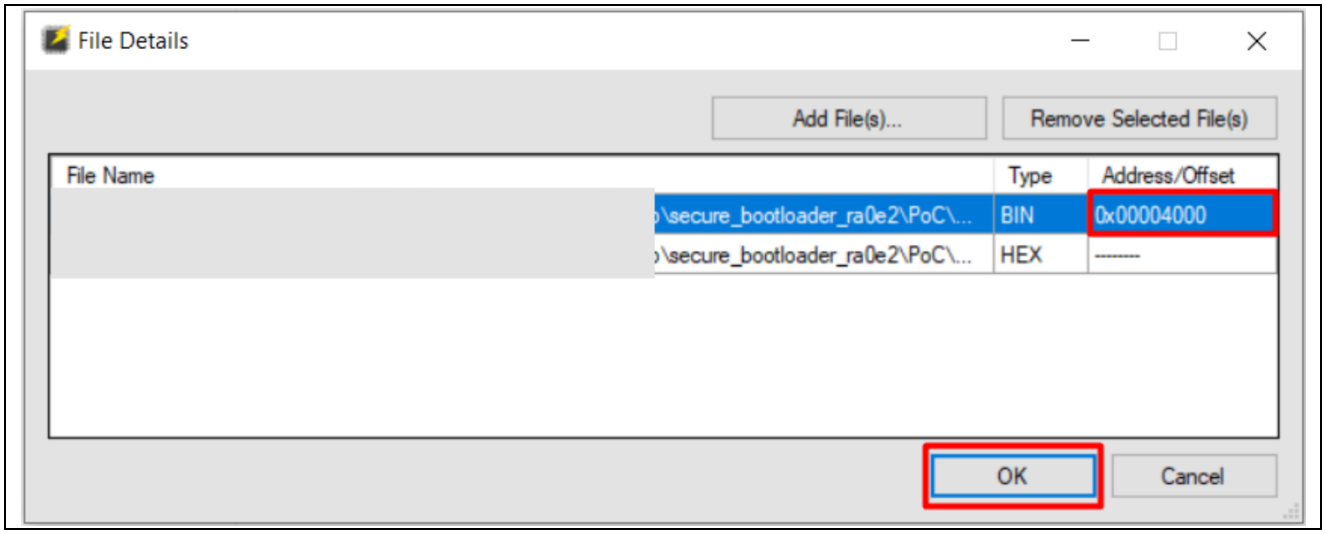

Figure 103. Selecting the Primary and Bootloader Images

Finally, execute the command to load both the bootloader and application images into the MCU.

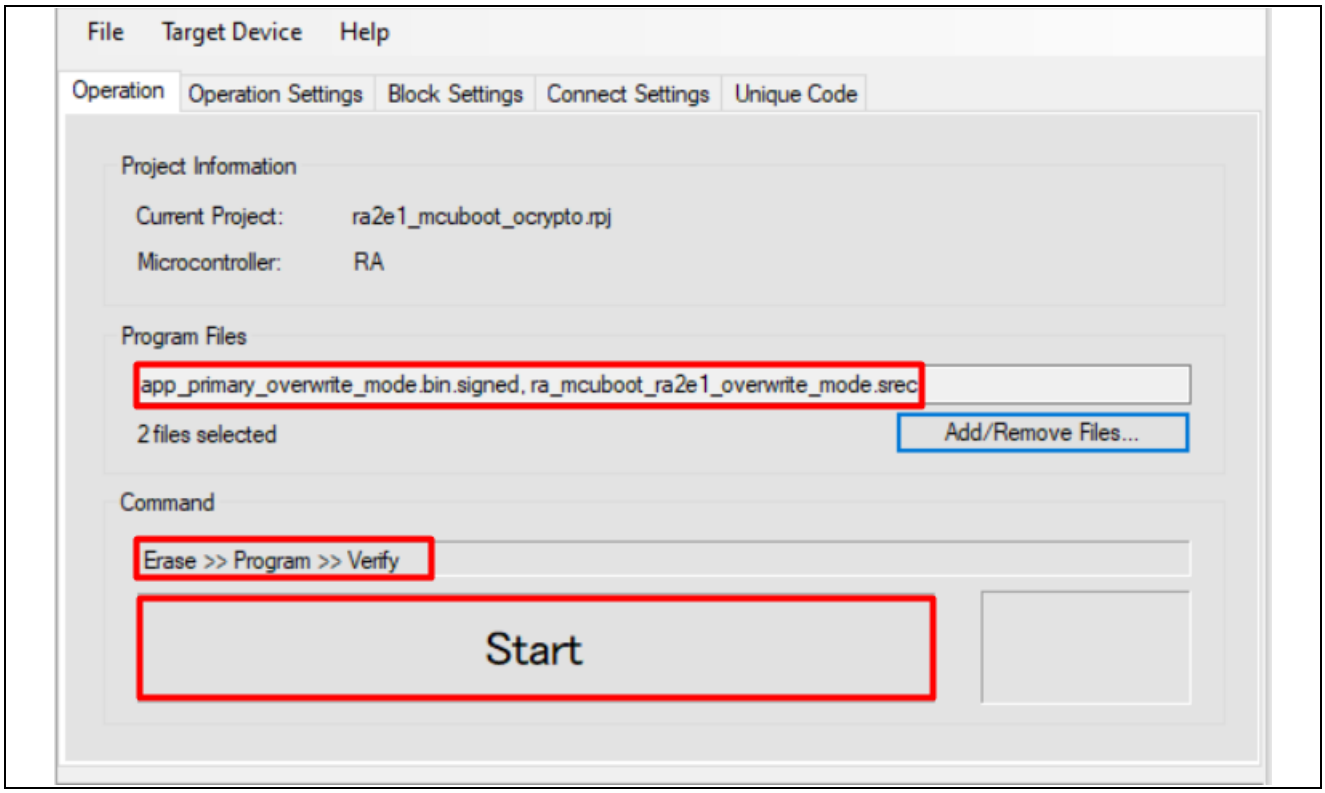

Figure 104. Execute the command to load images into the MCU

If the download images is successful, a message in Figure 105 will be displayed in the status window.

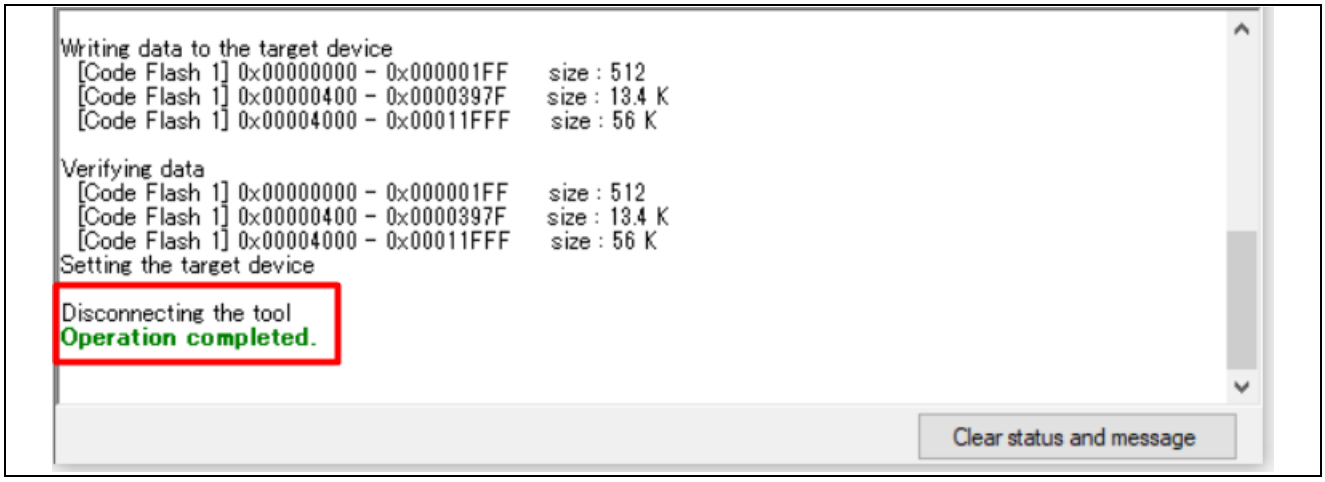

Figure 105. Images Download Succeeded

At this point, the primary application will be booted successfully. All LEDs should be blinking.

In addition, to update the new application, the image downloader will be implemented within the application project, refer to section 9.3.

9.3 Using the Image Downloader for updating the New Application

Prior to deployment, a system with a bootloader solution would typically need to include an image downloader and programmer in the application (primary and secondary applications), so a new application can be downloaded in the field. Users can refer to the *RA2 MCU Advanced Secure Bootloader Design using MCUboot Internal Code Flash and Memory Mirror Function Application Project (R01AN7766)* for more details.

9.4 Optimizing SRAM Utilization

Maximizing SRAM usage on the MCU is essential for large-scale projects. We will guide you on how to achieve this as follows.

Users should navigate to the **Solution Project**: `overwrite_mode_solution` to configure the maximum SRAM in the **Memory Partition** for the application project and then press **Ctrl + S** to save the settings.

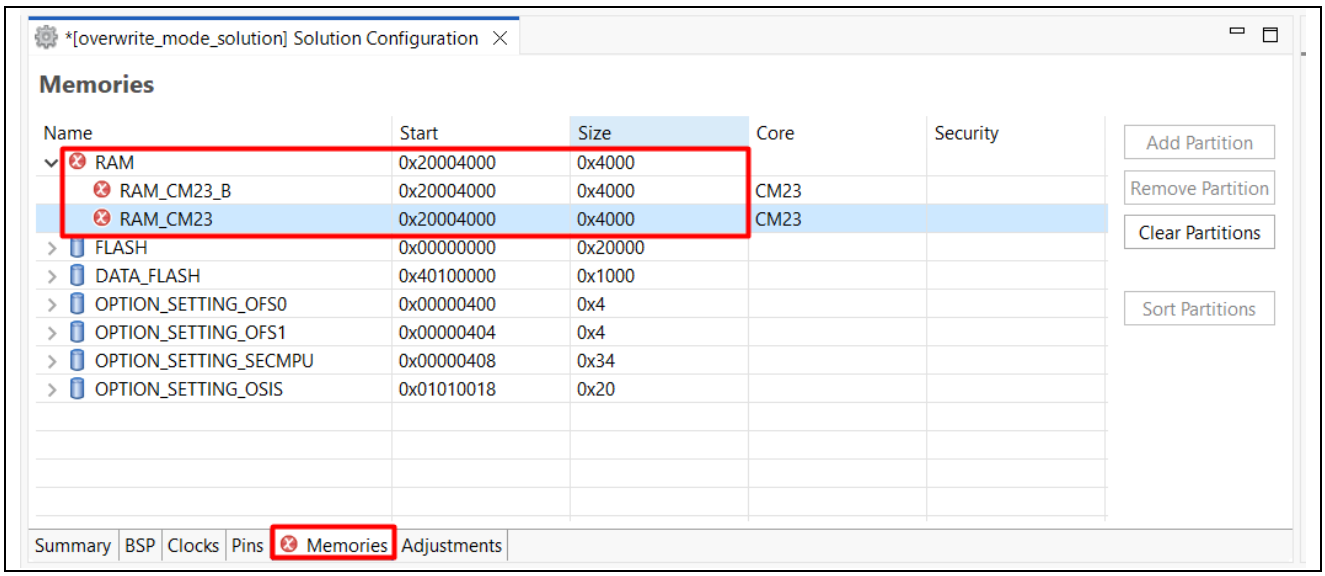

Figure 106. Adjust the maximum SRAM for the application

The partitioning shown in Figure 106 will cause a **Partition Overlapping** error, as shown in Figure 107.

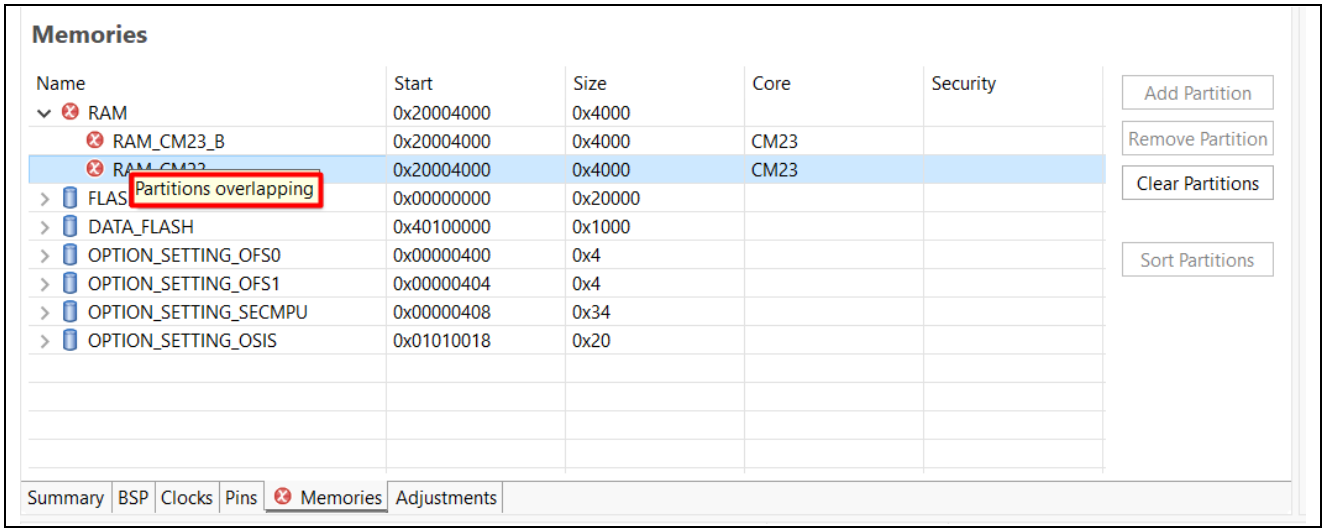

Figure 107. Partition Overlapping Error

The bootloader project needs to reserve a certain amount of SRAM, but once it has booted the application project, that SRAM becomes available for the application project.

This issue has no impact on the actual SRAM allocation for the bootloader or the application. Users can ignore this error message and proceed to the compilation process. The system functions as expectedly regardless of this error message.

9.5 Making the Bootloader Immutable

Refer to the *Renesas RA MCU Family Securing Data at Rest Utilizing the Renesas Security MPU* application project section Permanent Locking of the FAW Region to understand how to make the bootloader immutable. The PC Application to Permanently Lock the FAW section in the same application note describes how to handle flash locking in production mode.

9.6 Disabling the debug and Serial Programming Interface Prior to Deployment

Once the bootloader development is finished, you may want to set up ID Code protection on the Renesas RA2 MCU to lock down the debugger and the serial programming interface.

Refer to the *Securing Data at Rest Utilizing the Renesas Security MPU Application Project* section Setting up the Security Control for Debugging for the desired settings to control the device lifecycle management of the RA2 MCUs using the ID Code protection method.

10. References

1. Renesas RA Family MCU Securing Data at Rest using Security MPU Application Project (R11AN0416)
2. RA6 Secure Bootloader Using MCUboot and Internal Code Flash Application Project (R11AN0497)
3. RA6 Booting Encrypted Image using MCUboot and QSPI Application Project (R11AN0567)
4. RA2 MCU Advanced Secure Bootloader Design using MCUboot Internal Code Flash and Memory Mirror Function Application Project (R01AN7766)

11. Website and Support

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

EK-RA2E1 Resources	renesas.com/ra/ek-ra2e1
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	Dec.04.25	-	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.

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/.