Renesas RA Family

# Secure Bootloader for RA2 MCU Series

## 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 from the operating system it works with. MCUboot is maintained by Linaro in the GitHub mcu-tools page https://github.com/mcu-tools/mcuboot. There is a `/docs` folder that holds the documentation for MCUboot in `.md` file format. This application note refers to 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 RA2 MCU series is based on the Arm® Cortex®-M23 core and has limited flash and RAM memory. This application project is created to address the unique challenges and provide guidelines on the optimization of the RA2 MCU bootloader memory size. For the MCUboot cryptographic support for RA2 MCU groups, TinyCrypt (https://github.com/intel/tinycrypt/) is integrated with the FSP MCUboot module to provide a smaller memory footprint compared with Mbed Crypto. Refer to the GitHub folder `/tinycrypt/documentation/` for details on the TinyCrypt cryptographic algorithm usage.

This application note guides you through secure bootloader creation using the MCUboot Module with TinyCrypt 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. The Overwrite, Swap and Direct XIP upgrade modes are discussed and example projects are provided to support these upgrade modes.

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 with Mbed Crypto module. See the References section for information on that application project.

## Required Resources

### Development Tools and Software

- e² studio IDE v2023-04
- Renesas Flexible Software Package (FSP) v4.5.0
- SEGGER J-link® USB driver v7.88d

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 and Tera Term console, or similar application
- 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 the 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 3 to Section 6 are the walk-throughs of how to create bootloader projects using Overwrite, Swap, and Direct XIP upgrade modes, how to configure the application projects to use the bootloader, and how to boot the primary and secondary images.

Section 7 provides instructions on how to directly run the included example projects without going through Sections 3 to 6. For a quick evaluation of the included example projects, you can go directly to Section 7.

## Contents

# 1. Overview of MCUboot

MCUboot is an open source project hosted at mcu-tools github project. It is currently managed by the Linaro Community Project.

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 1.1.2. 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.

### 1.1.1 Overview of Application Booting Process

For applications using MCUboot, the MCU memory is separated into MCUboot, Primary App, Secondary App, and the Scratch Area. Following is an example of the single-image MCUboot memory map.

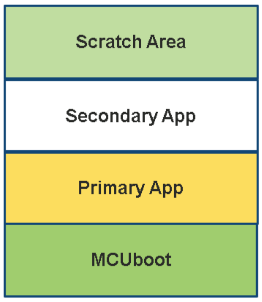


**Figure 1. Single Image MCUboot Memory Flash Map**

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.

### 1.1.2 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 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. However, encrypted image support using external flash is not currently supported.

- **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. However, encrypted image using external flash is not supported. 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 TinyCrypt is used in the RA2 bootloader design. We recommend reviewing the MCUboot Port section the *FSP User's Manual* to understand the build time configurations for MCUboot.

### 2.1 Secure Booting with TinyCrypt

TinyCrypt is a small-footprint cryptography library 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 RA2 bootloader design, SHA256 and ECDSA from TinyCrypt are used to ensure the application image integrity and authenticity. TinyCrypt does not support RSA.

The FSP TinyCrypt port module does not provide any interfaces to the user. Consult the documentation at https://github.com/intel/tinycrypt/blob/master/documentation/tinycrypt.rst for further information on use of the TinyCrypt port. The software only module is available in FSP on all RA devices. Hardware acceleration for AES-128 through FSP TinyCrypt port is provided for the RA2 family.

### 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:

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

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

### 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.
  - Update the linker script to optimize memory usage.
  - Disable unused FSP components to reduce code size.
  - Compile the bootloader with Optimization for Size (-Os).
  - Use pin configurations that initialize fewer peripheral and IO pins.

  Details on the operational flow of these optimization are described in section 3.
- Details on the RA2 bootloader memory optimization are introduced in later sections.

### 2.3.2 Time Usage in an Application Image Update

There are several major factors that can influence how much time an application image update takes. This section will discuss some of the major factors that can influence the time used in an application image update.

First, during an image update, if image verification using ECC or RSA is used, the larger the application image size, the longer it takes to verify the image for a given cryptographic algorithms.

Secondly, the larger the size of the application image, the longer it takes to erase and program the flash during the image upgrade process (for Overwrite and Swap upgrade mode where flash erase and programming are involved). User can reference the MCU Hardware User's manual section Electrical Characteristics to calculate the flash erase and programming time based on the table for code flash characteristics located in the sub section Flash Memory Characteristics.

Thirdly, the upgrade mode itself influences the time used to upgrade an application image. Assuming a new image is already downloaded and programmed to the update slot, the following erase and program events will happen after the MCU comes out from a reset.

For overwrite upgrade mode, the upgrade process involves:

- 2 x erase time (both primary and secondary slot)
- 1 x programming time (primary slot only)

For swap upgrade mode, the upgrade mode involves:

- 2 x erase time (both primary and secondary slot)
- The erase and program time used for erasing and programming the scratch area multiple times (with a total flash area equals the size of the application image on a scratch area size boundary)
- 2 x programming time (both primary and secondary slot)

For Direct XIP mode, the upgrade process does not involve any flash erasing or programming

- Since the image update in the Direct XIP mode does not involve any flash erasing and programming operation, this is the best upgrade mode in terms reducing the system downtime.

The fourth factor is related with the usage of different signature algorithms. RSA typically takes longer verification times compared with ECC for the same image size. Currently, only ECC is supported for RA2 signature verification.

## 2.4 Production Recommendations for RA2 MCU

### 2.4.1 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.

### 2.4.2 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.

# 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.zip`. You can follow section 7 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 e[2] studio and start to establish a new C/C++ Project. Click **File > New > C/C++ Project.**

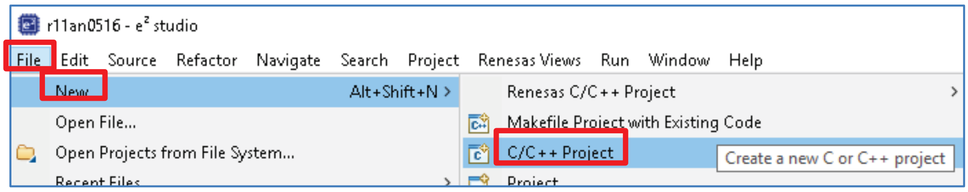


**Figure 2. Start a New Project**

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

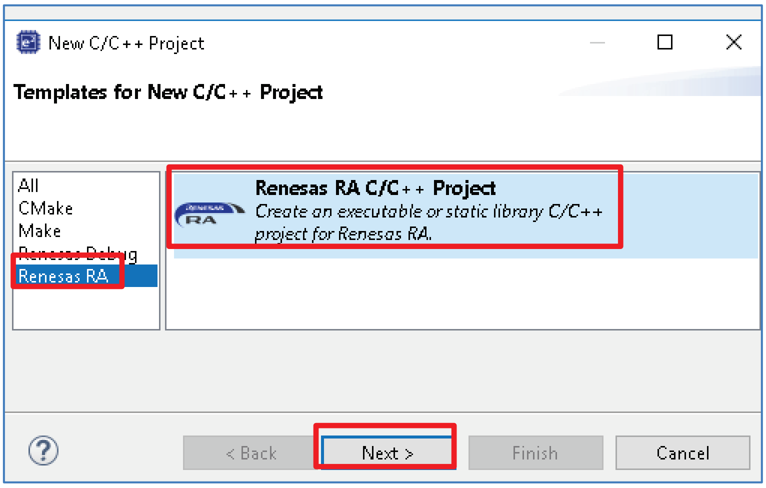


**Figure 3. Choose Renesas RA C/C++ Project**

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 update mode with no signature verification. |
| `ra_mcuboot_ra2e1_overwrite_with_signature` | Overwrite update mode with signature verification. |
| `ra_mcuboot_ra2e1_swap` | Swap update mode with no signature verification. Swap test prior to confirm is not supported. |
| `ra_mcuboot_ra2e1_swap_with_signature` | Swap update mode with signature verification, Swap test prior to confirm is supported. |
| `ra_mcuboot_ra2e1_dxip` | Direct XIP update mode with signature verification. |

Figure 4 is an example of setting the project name to `ra_mcuboot_ra2e1`.

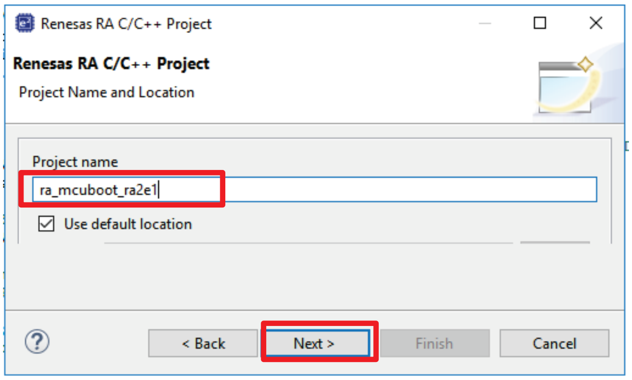


**Figure 4. Name the Bootloader Project**

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

4. In the next screen, choose **EK-RA2E1** for **Board** and click **Next**.

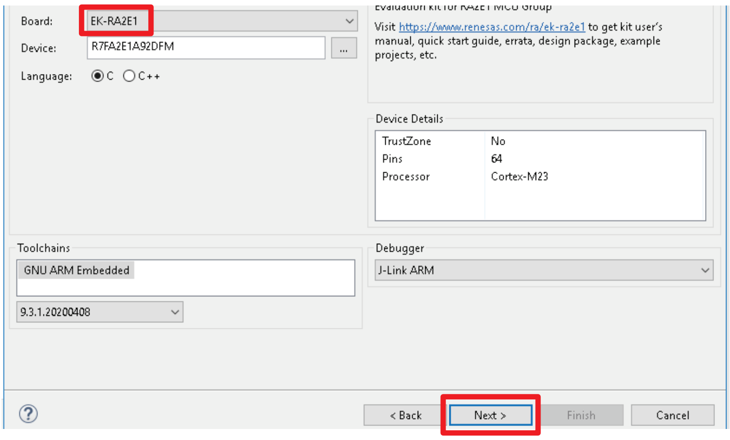


**Figure 5. Select the Board**

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

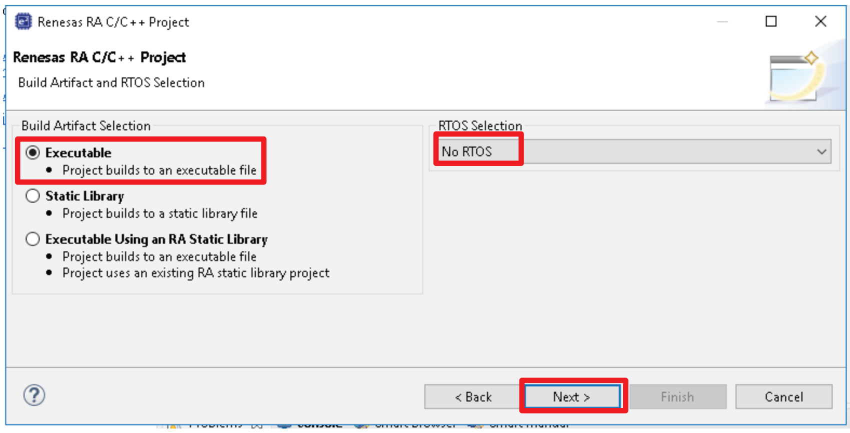


**Figure 6. Choose to Build Executable and No RTOS**

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

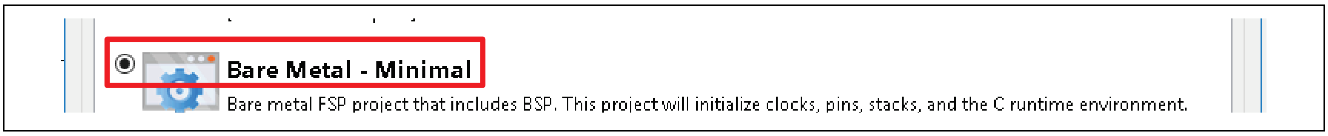


**Figure 7. Choose the Project Template**

7. When following prompt opens, click **Open Perspective**.

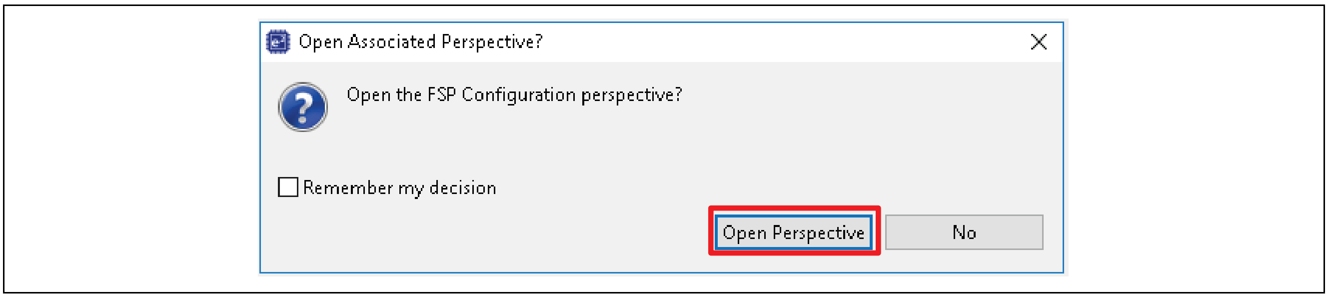


**Figure 8. Choose Open the FSP Configuration Perspective**

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

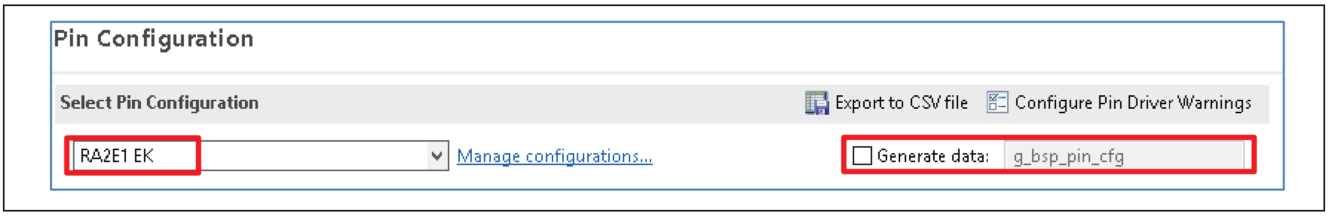


**Figure 9. Uncheck Generate data for RA2E1 EK Pin Configuration**

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 10. Select R7FA2E1A92DFM.pincfg and Generate data g_bsp_pin_cfg**

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

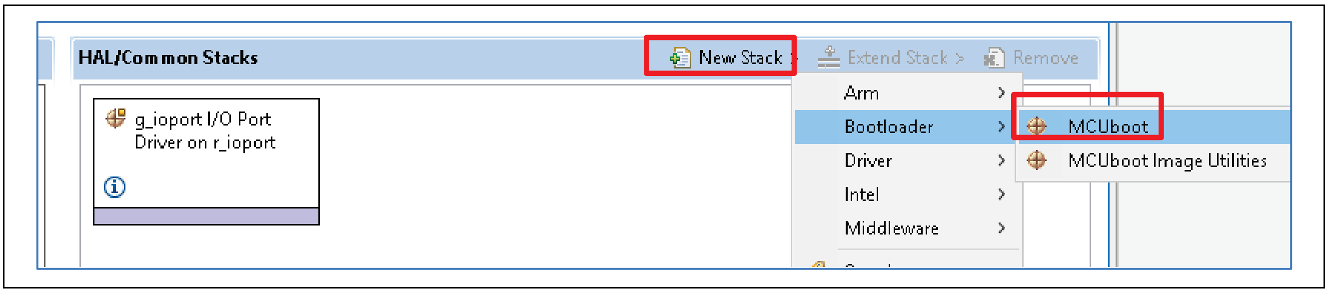


**Figure 11. Add the MCUboot Port**

10. Next, configure the **General** properties of **MCUboot**.
    - For project `ra_mcuboot_ra2e1` and `ra_mcuboot_ra2e1_overwrite_with_signature`, use the settings in Figure 12.
    - For project `ra_mcuboot_ra2e1_swap` and `ra_mcuboot_ra2e1_swap_with_signature`, update the following properties in Figure 12:
        - Change the **Upgrade Mode** to **Swap**.
        - Set the Downgrade Prevention (Overwrite Only) to Disabled.

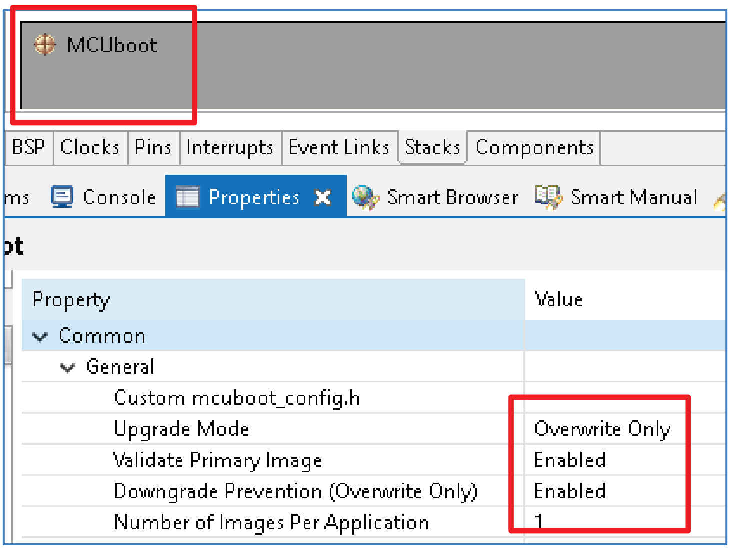


**Figure 12. General Properties for MCUboot**

Figure 13 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 6.2 for more details).
- The trailer magic number is written after the image upgrade is finished.

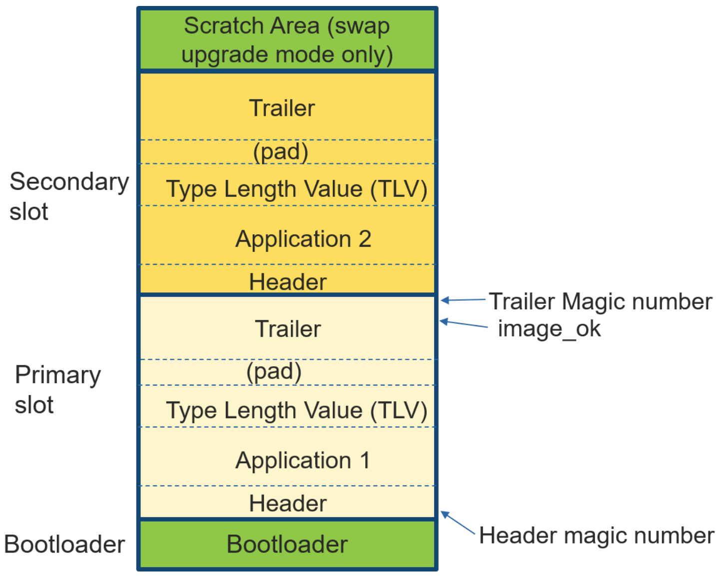


**Figure 13. General Configuration for MCUboot Module**

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 will perform a hash or signature verification, depending on the verification method chosen, in addition to the 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 this property is Disabled, only sanity check is performed based on the MCUboot header magic numbers. It is highly recommended to always enable this property. The additional code used when this property is enabled is less than 30 bytes, while it adds critical security handling to the bootloader.
  Note that the image magic number is not part of the image validation, it is a reference value that can be used for sanity check during application upgrade debugging process. This image magic number is written to the flash after a successful image upgrade.
- **Number of Images Per Application:** This property allows you to choose one image for Non-TrustZone-based applications and two images for TrustZone-based applications. RA2 MCU groups do not support TrustZone, so this property is set to **1**.
- **Downgrade Prevention (Overwrite Only):** This property applies to Overwrite upgrade mode only. When this property is Enabled, a new firmware with a lower version number will not overwrite the existing application. To see how to set the version number of an image, refer to Figure 51.

11. Configure the Signing Options and Flash Layout of the MCUboot module based on Table 2.

**Table 2. Bootloader Configurations**

| Bootloader Project Name | Screenshots for Detailed Configuration |
|---|---|
| `ra_mcuboot_ra2e1` | Figure 14 |
| `ra_mcuboot_ra2e1_overwrite_with_signature` | Figure 15 |
| `ra_mcuboot_ra2e1_swap` | Figure 16 |
| `ra_mcuboot_ra2e1_swap_with_signature` | Figure 17 |
| `Ra_mcuboot_ra2e1_dxip` | Figure 18 |

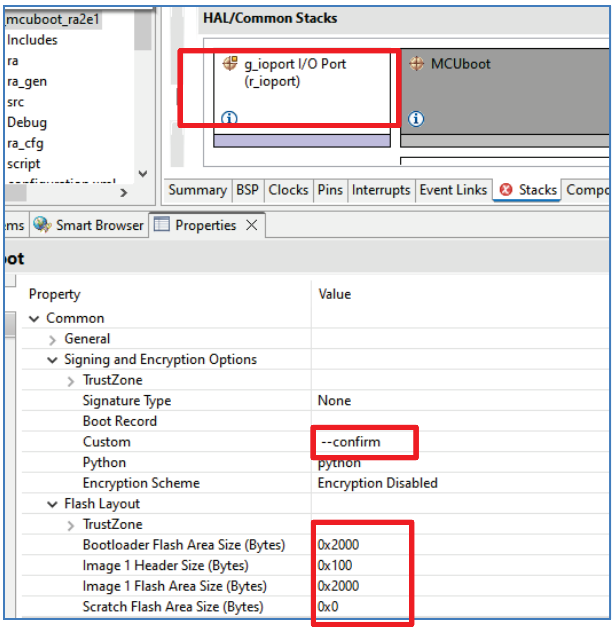


**Figure 14. Update Configurations for Project ra_mcuboot_ra2e1**

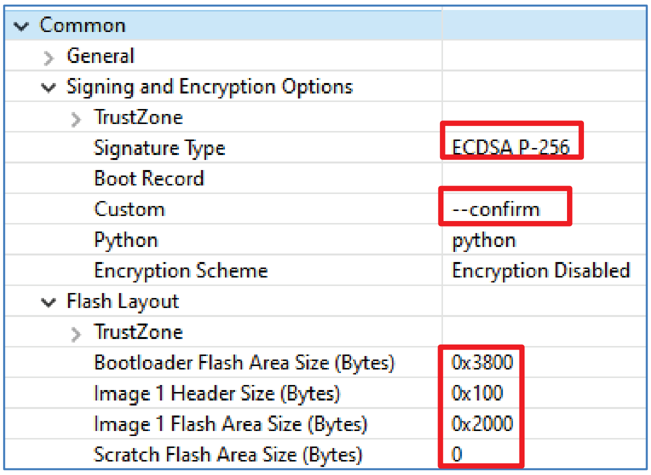


**Figure 15. Update Configurations for Project ra_mcuboot_ra2e1_overwrite_with_signature**

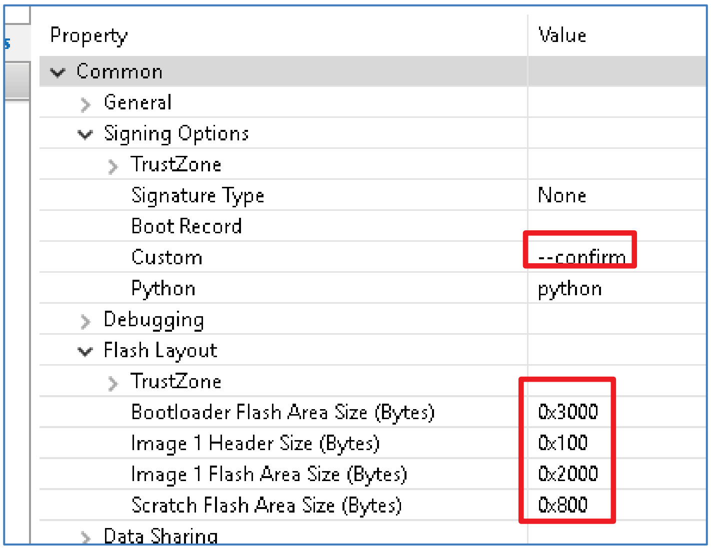


**Figure 16. Update Configurations for ra_mcuboot_ra2e1_swap**

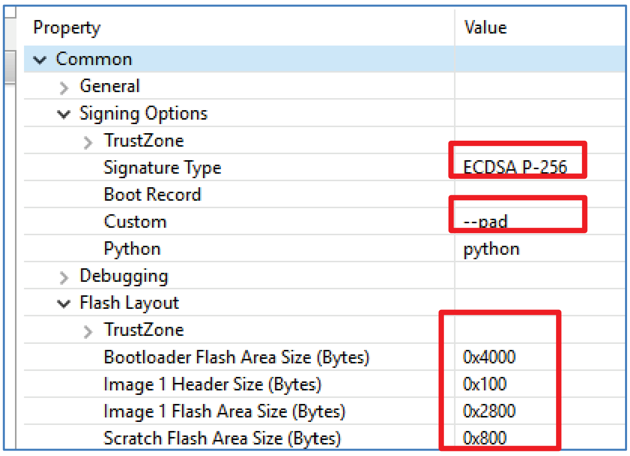


**Figure 17. Update Configurations for ra_mcuboot_ra2e1_swap_with_signature**

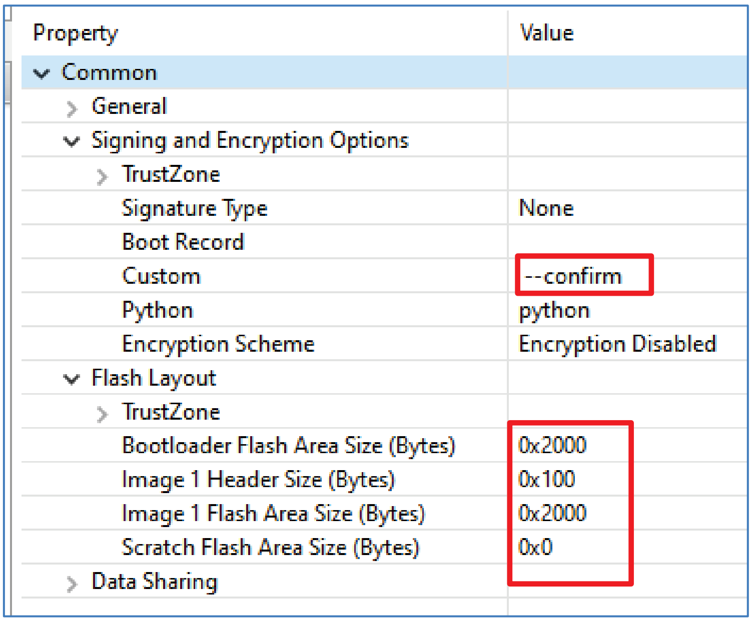


**Figure 18. Update Configurations for ra_mcuboot_ra2e1_dxip**

**Explanation of the Above Configurations**

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

- **Bootloader Flash Area**: Size of the flash area allocated for the bootloader with a boundary of 0x800 since 0x800 is the minimum erase size for code flash.
- **Image 1 Header Size**: Size of the flash area allocated for the application header for single-image configuration. For Arm Cortex-M23 MCUs, this should be set to 0x100.
- **Image 1 Flash Area Size**: Size of the flash area allocated for the application image for single-image configuration. This area needs to be equal or larger than the application image with a boundary of 0x800.
- **Scratch Flash Area Size**: This property is only needed for Swap mode. The Scratch Area must be large enough to store the largest sector that is going to be swapped. For all RA2 MCUs, the Scratch Area should be set up to 0x800 when Swap mode is used.
- **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 TinyCrypt 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 as shown in Figure 14 and Figure 16.
  - **ECDSA P-256**: This option is selected for the example bootloaders that support signature verification included in this application project as shown in Figure 15 and Figure 17.
  - **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 6.2.2.

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

**Properties that vary based on the Upgrade Mode Selection**
See Table 3 for the configuration used in the various bootloader projects introduced in this application project:

- Different authentication methods and different Image Upgrade mode use different amounts of flash memory. Select the most suitable configurations based on your specific application project requirement.
- The **Image 1 Flash Area size** is based on the simple blinky project. Adjust this memory configuration based on the specific application project you want to use with the bootloader.
- The Swap upgrade application project uses a larger flash area because the swap test mode is configured in the example project. For details on the swap test mode, refer to section 6.2.2.
- Note that there is no difference in the bootloader flash memory usage whether `--confirm` or `--pad` is defined for the **Custom** property. However, the new image which includes the MCUboot Image Utilities modules will need to allocate about 2kB flash for the added functionality.

**Table 3. Configurations for Different Upgrade Modes**

| Properties | ra_mcuboot_ra2e1 | ra_mcuboot_ra2e1_overwrite_with_signature | ra_mcuboot_ra2e1_swap | ra_mcuboot_ra2e1_swap_with_signature | ra_mcuboot_ra2e1_dxip |
|---|---|---|---|---|---|
| Bootloader Flash Area Size | 0x2000 | 0x3800 | 0x3000 | 0x4000 | 0x2000 |
| Image 1 Flash Area Size | 0x2000 | 0x2000 | 0x2000 | 0x2800 | 0x2000 |
| Signature Type | NONE | ECDSA P256 | NONE | ECDSA P256 | NONE |
| Custom | `--confirm` | `--confirm` | `--confirm` | `--pad` | `--confirm` |

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.

12. Next, add the **TinyCrypt** module under **MCUboot Port for RA**. **TinyCrypt (H/W Accelerated)** includes hardware accelerated AES functionality, which is not used in the bootloader, so **TinyCrypt (S/W Only)** is used. The **MbedTLS (Crypto Only)** module has a larger memory footprint compared with **TinyCrypt** and is not used in this bootloader design.

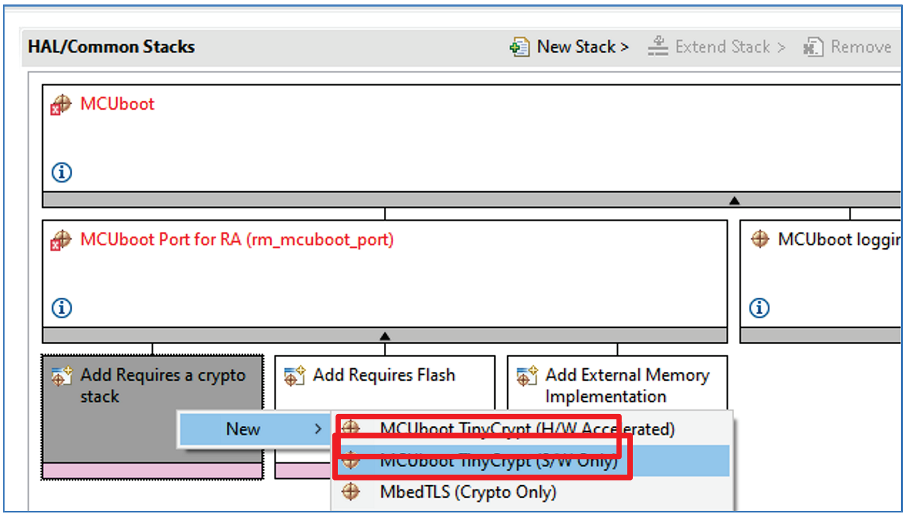


**Figure 19. Select TinyCrypt Module**

13. 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.

- ra_mcuboot_overwrite_with_signature
- ra_mcuboot_ra2e1_swap_with_signature

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

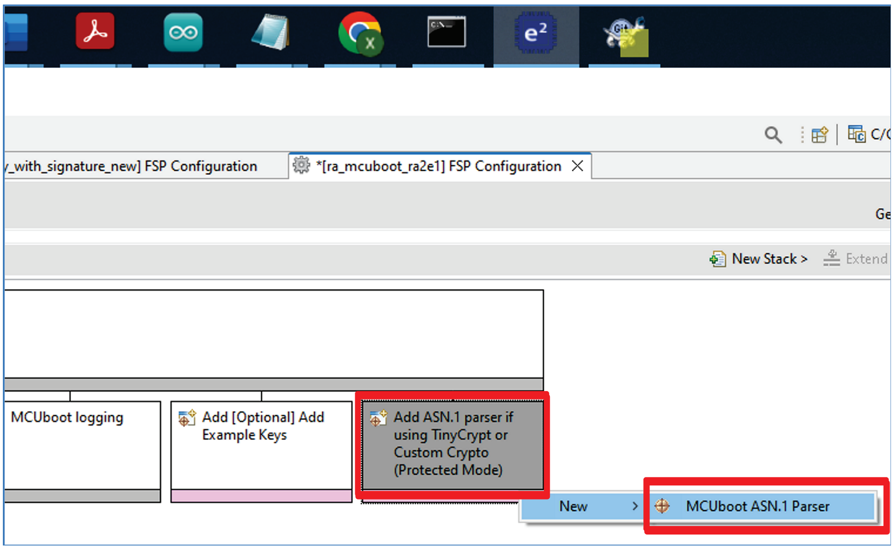


**Figure 20. Add the ASN.1 Parser**

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

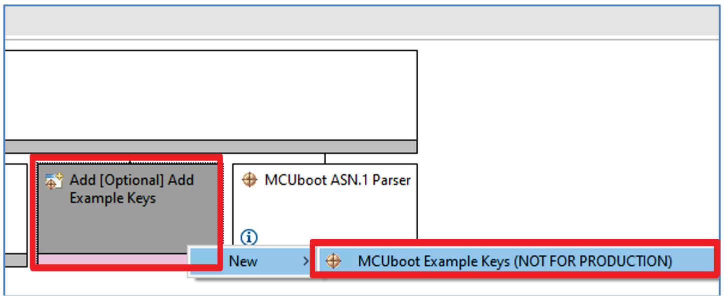


**Figure 21. Add the Example Image Signing Key**

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 3.6.1 to create and use customized signing key.

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

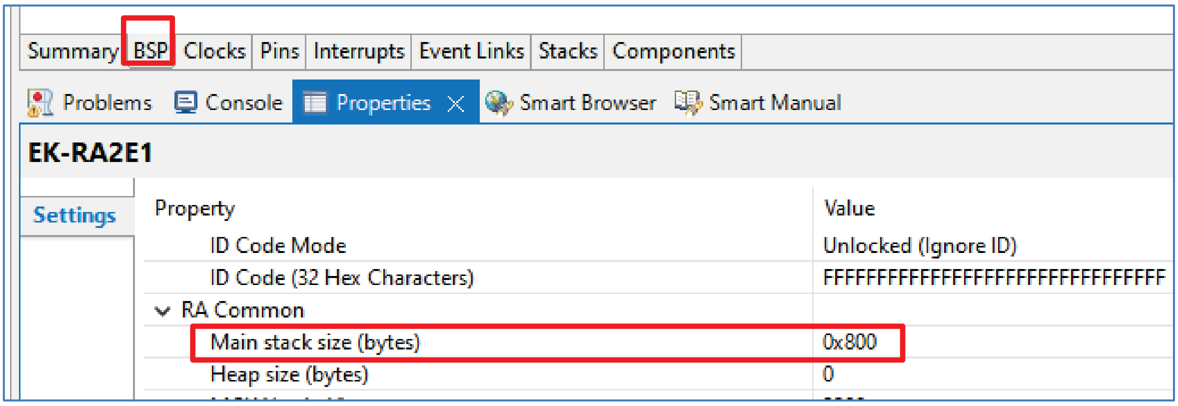


**Figure 22. Update the BSP Main Stack Size**

15. Click on **Add Required Flash** stack and add **Flash (r_flash_lp)**.
16. Click on the **Flash Driver** block and set the **Code Flash Programming** to **Enabled**. As **Data Flash Programming** and **Data Flash Background Operation** are not used in the bootloader, select **Disabled** for these two properties to reduce the bootloader memory footprint.

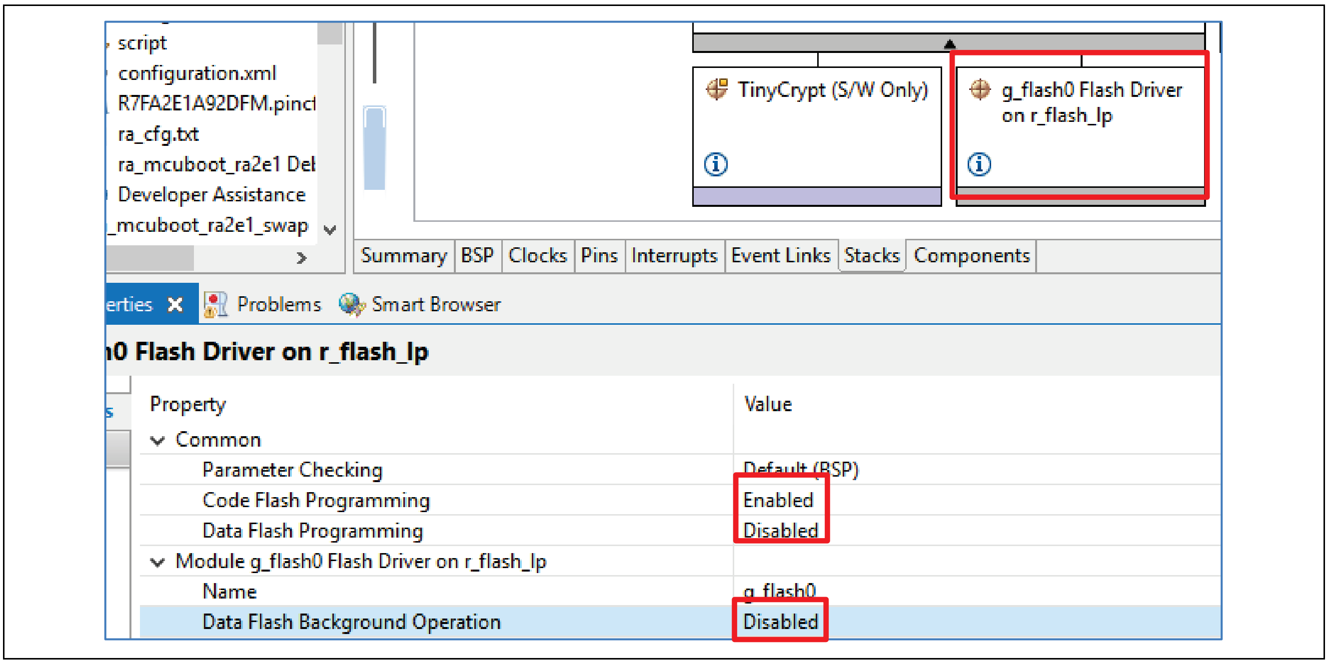


**Figure 23. Enable Code Flash programming**

17. 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();`
18. 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**.

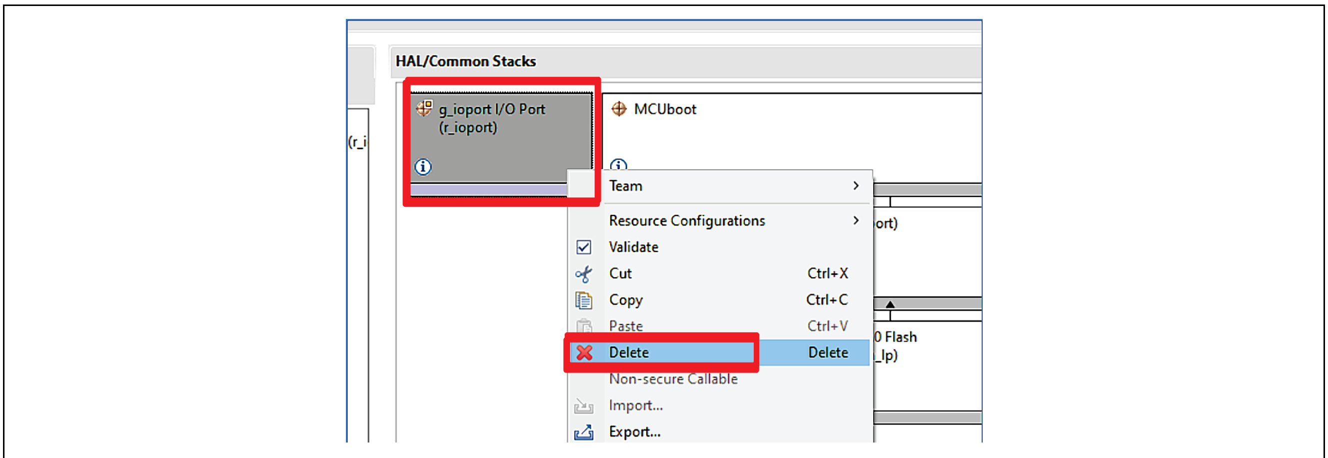


**Figure 24. Remove the I/O Port Stack**

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_entry.c` as shown in Figure 25.

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

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

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

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

        /* Configure pins. */
        R_IOPORT_Open (&g_ioport_ctrl, &g_bsp_pin_cfg);
    }
}
```

**Figure 25. Remove Unused Code in hal_entry.c**

## 3.2 Further Optimizing for the Bootloader Project Size

To further optimize the bootloader project for size, you can put some application code in the gap area between the interrupt vector and the RA2E1 ROM registers. We can create a section (`.code_in_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. You can use the methods described in this section to save code space for any RA2 application.

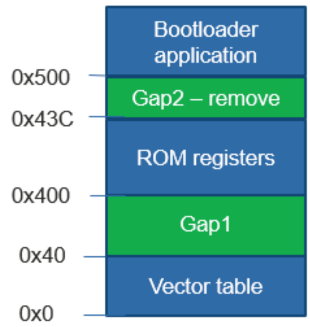


**Figure 26. First Flash Sector**

First, update the default linker script to include section `.code_in_gap` between the interrupt vector and the ROM register as shown in Figure 27. In addition, the application code after the ROM register can start at 0x43C instead of 0x500 as used in the default linker script.

Note that there is a section for `.mcuboot_sce9_key`, which is not used for RA2 MCUs. We can safely comment this section out.

```
KEEP(*(.fixed_vectors*))
KEEP(*(.application_vectors*))
__Vectors_End = .;

*(.code_in_gap*)

/* ROM Registers start at address 0x00000400 for devices that do not have the OPTION_SETTING region. */
. = OPTION_SETTING_LENGTH > 0 ? . : __ROM_Start + 0x400;
KEEP(*(.rom_registers*))

/* Reserving 0x100 bytes of space for ROM registers. */
. = OPTION_SETTING_LENGTH > 0 ? . : __ROM_Start + 0x43C;

/* Allocate flash write-boundary-aligned
 * space for sce9 wrapped public keys for mcuboot if the module is used.
 */
  = ALIGN(128); */
  KEEP(*(.mcuboot_sce9_key*)) */

*(.text*)
```

**Figure 27. Linker Script Update**

Next, you can choose some functions to put in the `.code_in_gap` section in order to reduce the flash usage. What functions to put in the `.code_in_gap` section is your choice.

For all five bootloaders introduced in this application project, the following two functions are put in the gap area. For the `ra_mcuboot_ra2e1_dxip` bootloader, there is no need to add more functions to the gap area.

- Update function prototype **R_BSP_WarmStart** shown in Figure 28.
- Add function prototype definition for **`mcuboot_quick_setup`** as shown in Figure 28 right before this function's implementation (refer to the sample code for an example).

In **`\src\hal_entry.c:`**

```
void R_BSP_WarmStart(bsp_warm_start_event_t event) BSP_PLACE_IN_SECTION(".code_in_gap*");
void mcuboot_quick_setup() BSP_PLACE_IN_SECTION(".code_in_gap*");
```

In **`\ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\bootutil.h:`**

```
fih_int context_boot_go(struct boot_loader_state *state, struct boot_rsp *rsp)
BSP_PLACE_IN_SECTION(".code_in_gap*");
```

**Figure 28. Common Functions to Put in the .code_in_gap Section**

Figure 29 shows the additional function in `image.h` that is put in the gap area for bootloader `ra_mcuboot_ra2e1` in addition to the common functions mentioned in Figure 28.

In **`\ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\image.h:`**

```
fih_int bootutil_img_validate(struct enc_key_data *enc_state, int image_index,
                                                    struct image_header *hdr,
                                                  const struct flash_area *fap,
                                              uint8_t *tmp_buf, uint32_t tmp_buf_sz,
                              uint8_t *seed, int seed_len, uint8_t *out_hash)
BSP_PLACE_IN_SECTION(".code_in_gap*");
```

**Figure 29. Functions to Put in the .code_in_gap Section for ra_mcuboot_ra2e1**

Figure 30 shows the two additional functions in `image.h` that are put in the gap area for bootloader `ra_mcuboot_ra2e1_overwrite_with_signature` in addition to the common functions mentioned in Figure 28.

**in \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\image.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(".code_in_gap*");
int bootutil_tlv_iter_next(struct image_tlv_iter *it, uint32_t *off,
                           uint16_t *len, uint16_t *type) BSP_PLACE_IN_SECTION(".code_in_gap*");
```

**Figure 30. Functions to Put in the .code_in_gap Section for ra_mcuboot_ra2e1_overwrite_with_signature**

Figure 31 shows the addition function in `image.h` that is put in the gap area for `ra_mcuboot_ra2e1_swap` and `ra_mcuboot_ra2e1_swap_with_signature` in addition to the common functions mentioned in Figure 28.

**in \ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\image.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(".code_in_gap*");
```

**Figure 31. Functions to Put in the .code_in_gap Section for the Swap Update Mode**

## 3.3 Compiling the Bootloader Project

When all the above updates are done, change the compiling optimization to **Optimize size (-Os)** and compile the project.

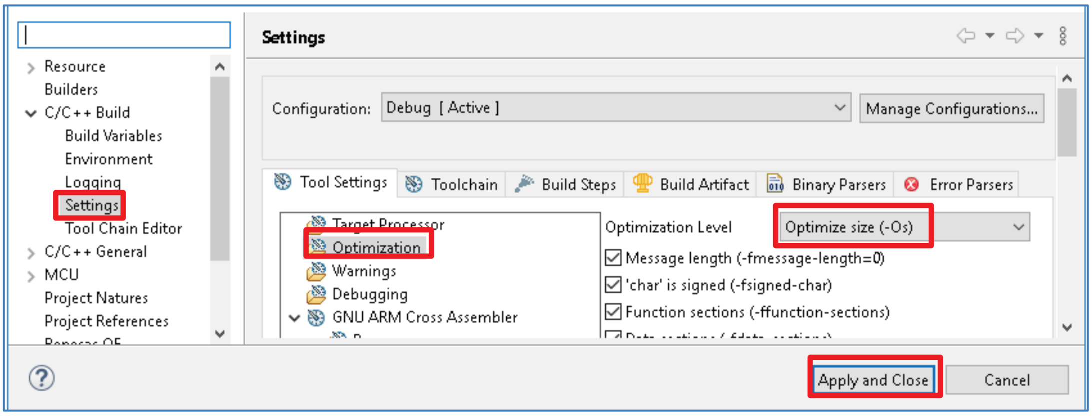


**Figure 32. Optimize Bootloader Size**

Depending on which upgrade mode you have selected, Figure 33-Figure 36 show the compilation results. If you have migrated the projects to a later FSP version, the size may have some minor difference.

```
Building target: ra_mcuboot_ra2e1.elf
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1.elf"  "ra_mcuboot_ra2e1.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1.elf"
   text    data     bss     dec     hex filename
   7976       0    3804   11780    2e04 ra_mcuboot_ra2e1.elf

15:17:25 Build Finished. 0 errors, 0 warnings. (took 1s.318ms)
```

**Figure 33. Compile the Bootloader ra_mcuboot_ra2e1**

```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1_overwrite_with_signature.elf"  "ra_mcuboot_ra2e1_overwrite_with_signature.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1_overwrite_with_signature.elf"
   text    data     bss     dec     hex filename
  12840       0    3808   16648    4108 ra_mcuboot_ra2e1_overwrite_with_signature.elf

23:17:40 Build Finished. 0 errors, 0 warnings. (took 4s.842ms)
```

**Figure 34. Compile the Bootloader ra_mcuboot_ra2e1_overwrite_with_signature**

```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1_swap.elf"  "ra_mcuboot_ra2e1_swap.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1_swap.elf"
   text    data     bss     dec     hex filename
  11556       0    3864   15420    3c3c ra_mcuboot_ra2e1_swap.elf

10:32:28 Build Finished. 0 errors, 0 warnings. (took 5s.519ms)
```

**Figure 35. Compile the Bootloader ra_mcuboot_ra2e1_swap**

```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1_swap_with_signature.elf"  "ra_mcuboot_ra2e1_swap_with_signature.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1_swap_with_signature.elf"
   text    data     bss     dec     hex filename
  16412       0    3884   20296    4f48 ra_mcuboot_ra2e1_swap_with_signature.elf

23:32:18 Build Finished. 0 errors, 125 warnings. (took 5s.89ms)
```

**Figure 36. Compile the Bootloader ra_mcuboot_ra2e1_swap_with_signature**

```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1_dxip.elf"  "ra_mcuboot_ra2e1_dxip.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1_dxip.elf"
   text    data     bss     dec     hex filename
   7384       0    2732   10116    2784 ra_mcuboot_ra2e1_dxip.elf

11:23:29 Build Finished. 0 errors, 0 warnings. (took 4s.1ms)
```

**Figure 37. Compile the Bootloader ra_mcuboot_ra2e1_dxip**

## 3.4 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 3.5.

If this is the first time you are using the Python script signing tool on your system, you will need to install the dependencies required for the script to work. Navigate to the `<boot_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.

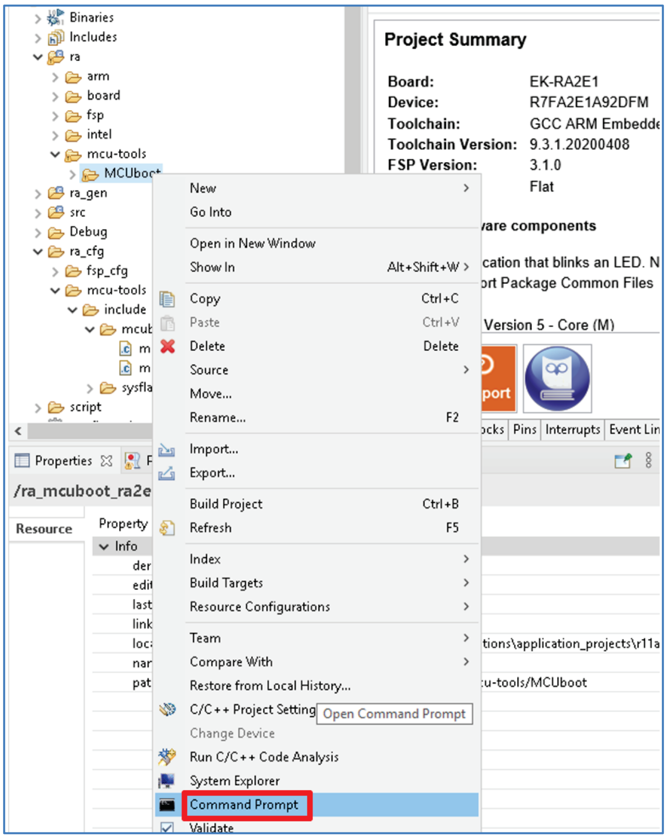


Figure 38. Open the Command Prompt

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.

## 3.5 Review the Signing Command

The signing command for the application image is automatically generated when the bootloader is compiled. In the **Project Explorer**, navigate to the `<boot_project>\Debug\<boot_project >.bld` file and open this `.bld` file. The signing command is under the section <image >. For RA2 MCU groups, the entry immediately after <images> is the signing command for the application image.

The application image uses a Build Variable to link with the `.bld` file. This process is explained in detail in the next section. The signing command is automatically executed when the application image is compiled.

```
<images>
        <image path="${BuildArtifactFileBaseName}.bin.signed">python
${workspace_loc:ra_mcuboot_ra2e1}/ra/fsp/src/rm_mcuboot_port/rm_mcuboot_port_sign.py sign --
header-size 0x100 --align 8 --max-align 8 --slot-size 0x2000 --max-sectors 4 --overwrite-only  -
-confirm --pad-header ${BuildArtifactFileName} ${BuildArtifactFileBaseName}.bin.signed</image>
        <image path="${BuildArtifactFileBaseName}.bin.signed" security="n">python
${workspace_loc:ra_mcuboot_ra2e1}/ra/fsp/src/rm_mcuboot_port/rm_mcuboot_port_sign.py sign --
header-size 0x100 --align 8 --max-align 8 --slot-size 0x0 --max-sectors 4 --overwrite-only  --
confirm --pad-header ${BuildArtifactFileName} ${BuildArtifactFileBaseName}.bin.signed</image>
    </images>
```

Figure 39. Signing Command (in bold) in the .bld File

## 3.6 Usage Notes

### 3.6.1 Using Customized Image Signing Key

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

The stack 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 40. Public Key used for Image Verification (from keys.c)**

The matching private key for the public key `root_pub_der is root-ec-p256.pem`. This example is used in the image signing process in the example bootloaders created in this section.

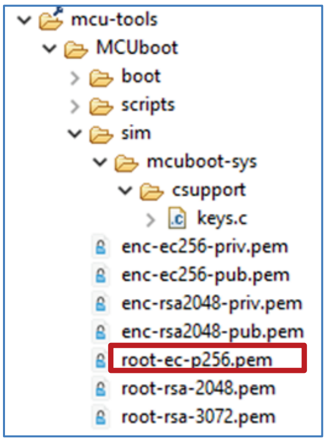


**Figure 41. Example Image Signing Private Key**

We will generate a custom private key `ecc_sign_private.pem` to replace the usage of `root-ec-p256.pem` following the below steps using any of the bootloader example using signature:

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

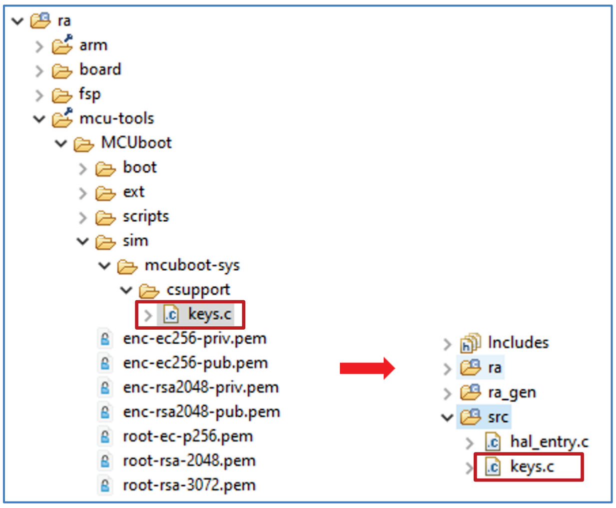


**Figure 42. Copy the example Keys.c**

2. Open the configurator for the bootloader project, right click on MCUboot Example Keys stack and select **Delete**.

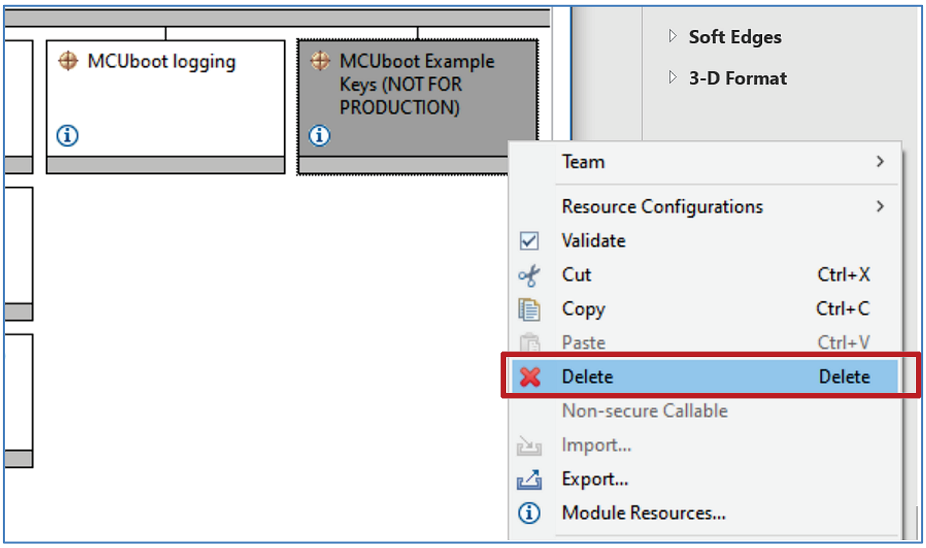


**Figure 43. Delete the MCUboot Example Keys Stack**

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

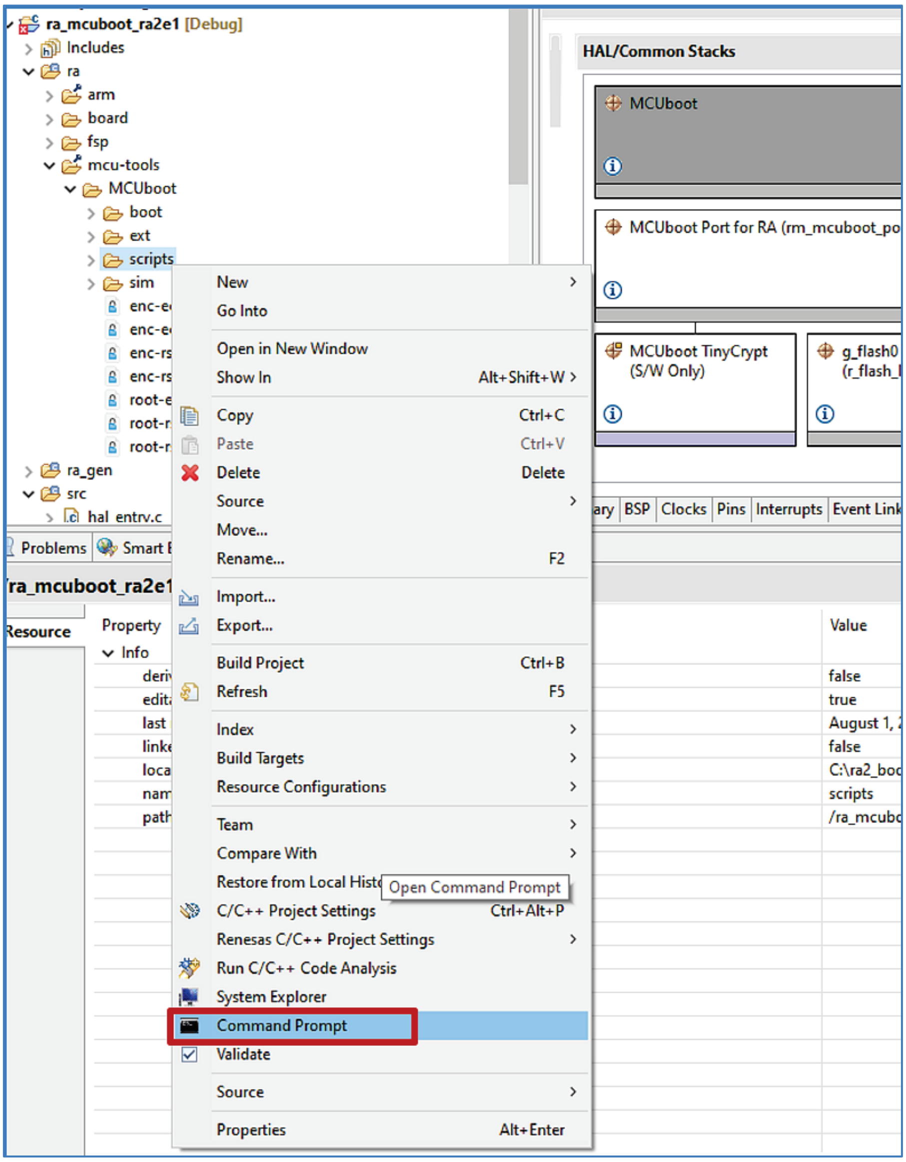


**Figure 44. Start Command Prompt under the \MCUboot\scripts Folder**

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\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`


**Figure 45. Start Command Prompt under the \MCUboot\scripts Folder**

7. Copy the content of `ecdsa_pub_key to keys.c` to `replarray root_pub_der keys.c`. Replace the original `root_pub_der` content.
8. Click **Generate Project Content** and compile the bootloader project.
9. To use the new image signing key, user needs to update the signing key configuration of the application projects.

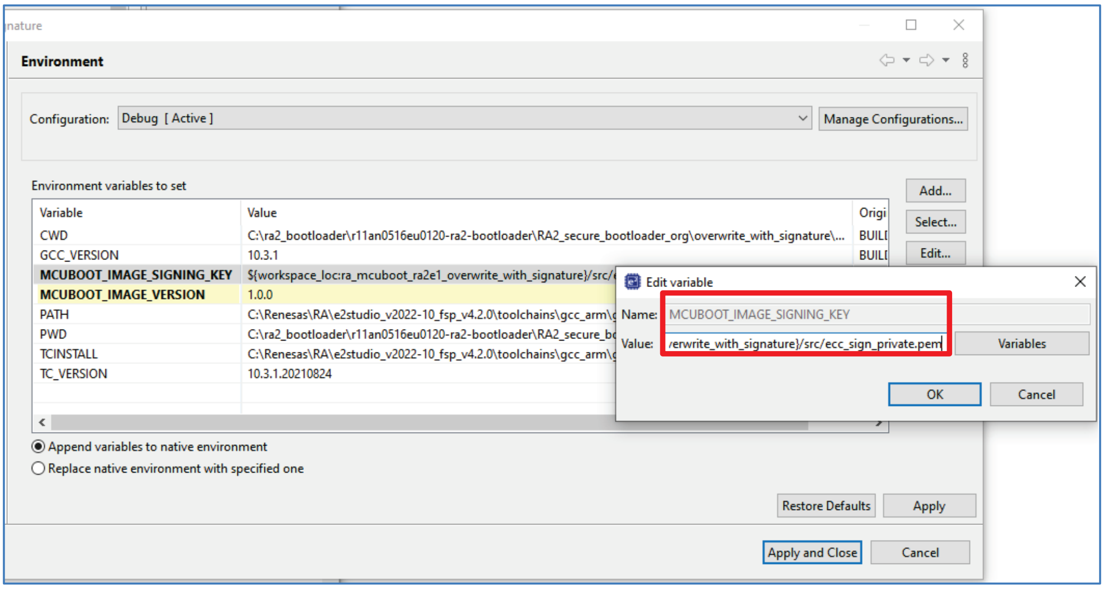


**Figure 46. Configure the Application Project to use the Custom Image Signing**

Recompile the application projects and following the instructions in section 7 to exercise the system.

### 3.6.2 Migrating the Bootloader to other FSP versions

When migrating the bootloader project to a new FSP version, the updated contents in the `\ra` folder will be overwritten by the extracted new FSP content. Hence, the functions that are put in the gap area need to be reconfigured by updating the corresponding header files described in section 3.2.

The linker script is not automatically updated when user performs Generate Project Content, for applications using MCUboot as bootloader, in most cases, user may not need to update the linker script to boot the new application projects. However, there may be other linker script updates that are related with other application areas or new features related with MCUboot. Therefore, user is recommended to extract the new linker script by deleting the included linker script and apply similar updates to the linker script accordingly section 3.2.

Note that the instructions included in this release only applies to FSP v4.5.0, migration to other versions may need more customization. User needs to review the FSP release note on other potential updates needed.

### 3.6.3 Migrating from One Upgrade Mode to Another Upgrade Mode

As shown in section 3.2, a different set of functions need to be put in the gap area. The configurations selected in this application can be used as reference.

### 3.6.4 Use the Memory Usage Window to Select Functions to Put in the Gap Area

After compiling the bootloader project, you can open the **Memory Usage** view to select the functions of suitable size to put in the gap area.

Open the **Memory Usage** view from the e² studio top menu **Windows** tab: **Window > Show View > Other > C/C++ > Memory Usage > Symbol**.

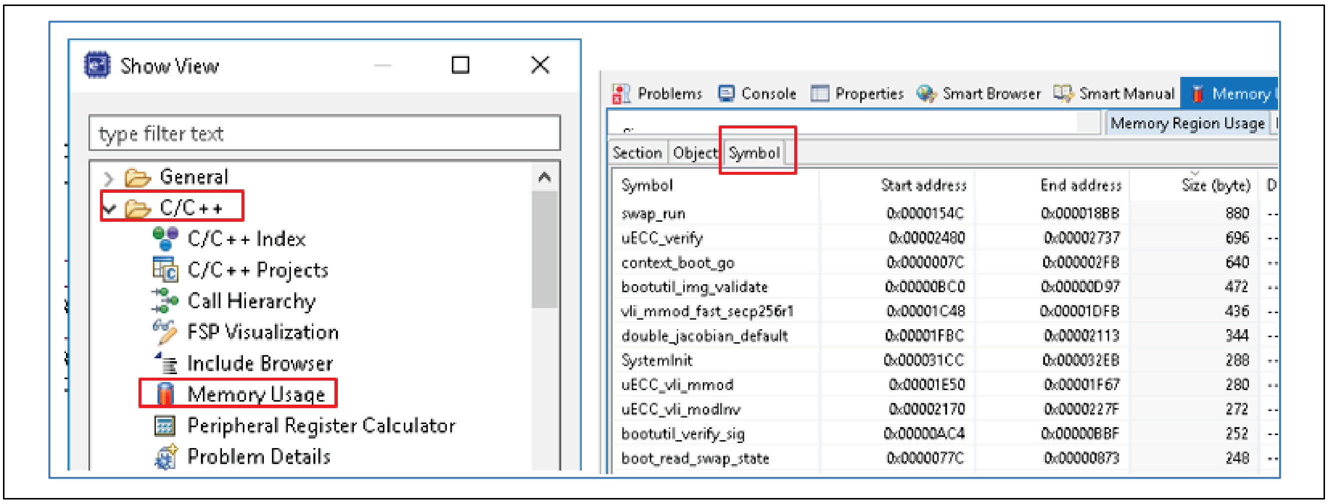


**Figure 47. Memory Usage View**

## 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 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:** You can also follow section 7 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 reference 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 e² studio and open a Workspace, click **File** > **New** > **C/C++ Project** and select **Renesas RA** and **Renesas RA C/C++** Project.

1. Assign the project name based on Table 4.

**Table 4. Name the Initial Application Project**

| Bootloader project name | Initial application project name |
|---|---|
| ra_mcuboot_ra2e1 | blinky |
| ra_mcuboot_ra2e1_overwrite_with_signature | blinky_with_signature |
| ra_mcuboot_ra2e1_swap | blinky_swap |
| ra_mcuboot_ra2e1_swap_with_signature | blinky_swap_with_signature |
| Ra_mcuboot_ra2e1_dxip | blinky_primary |

2. Click **Next** and choose **EK-RA2E1** as the **Board** from the drop-down menu. Then click **Next**.
3. In the next screen, select **Executable** as the **Build Artifact** and **No RTOS** for the **RTOS Selection**. Then click **Next**.

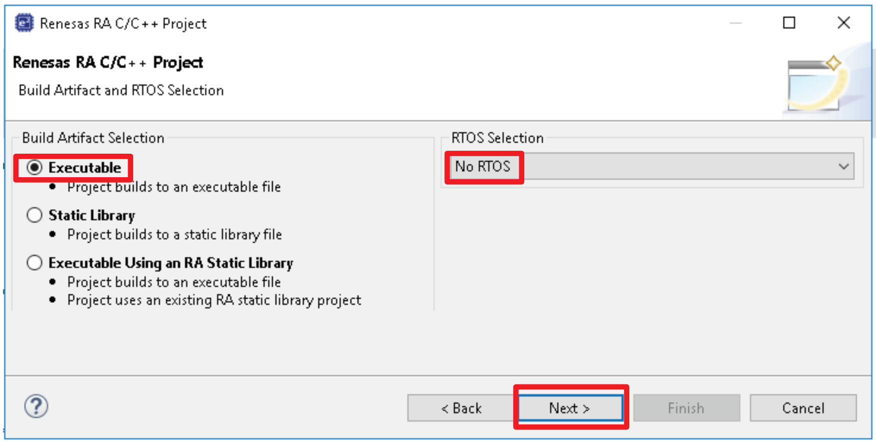


**Figure 48. Choose to Build Executable with No RTOS**

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

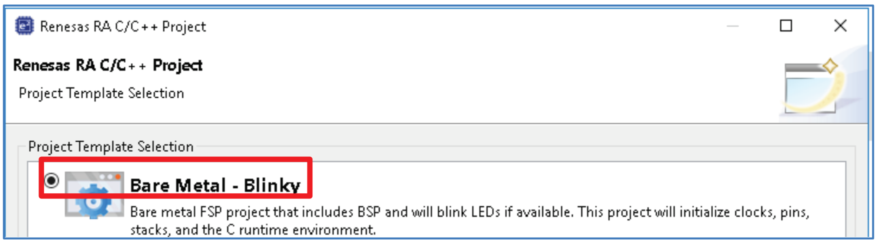


**Figure 49. Choose Bare Metal – Blinky as Project Template**

## 4.2 Configure the Existing Application to Use the Bootloader Project

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

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

- Set **`${workspace_loc:<boot_project_name>}/Debug/<boot_project_name>.bld`** for **Value**.
- For example, for bootloader project ra_mcuboot_ra2e1 (see Figure 50), **Value** will be:
  **`${workspace_loc:ra_mcuboot_ra2e1}/Debug/ra_mcuboot_ra2e1.bld`**

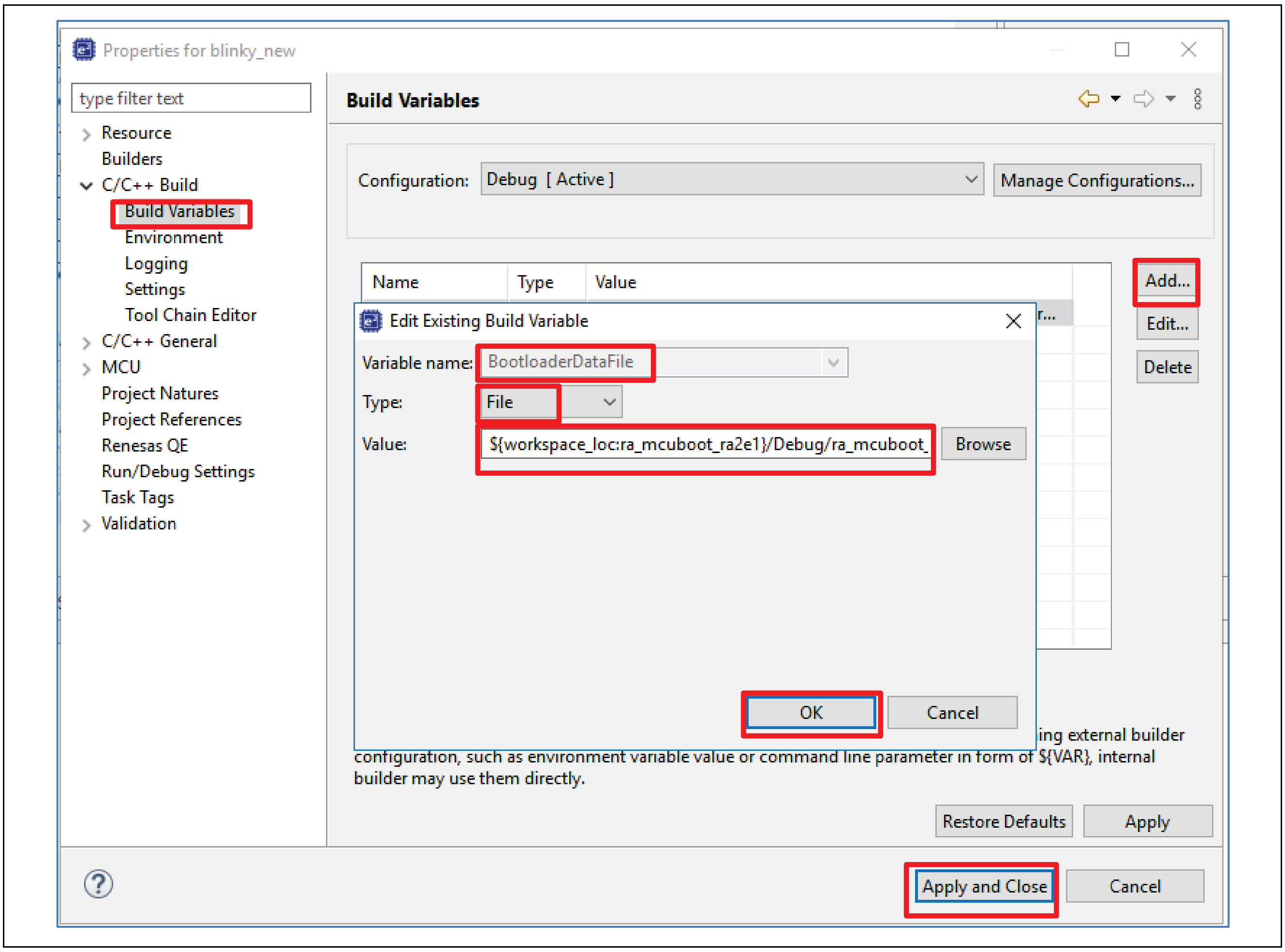


**Figure 50. Configure the Build Variable to Use the Bootloader**

Click **Apply** and then **Apply and Close**.

## 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 will need to select **Generate Project Content** again to bring in the updated `.bld` file.

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.

For applications that support signature verification, meaning for the applications that will work with bootloader `ra_mcuboot_ra2e1_overwrite_with_signature` and `ra_mcuboot_ra2e1_swap_with_signature`, the signing key can be configured using another Environment Variable: MCUBOOT_IMAGE_SIGNING_KEY.

Figure 51 is an example of setting the above two mentioned Environment Variables for the application project used with bootloader `ra_mcuboot_ra2e1_overwrite_with_signature` and `ra_mcuboot_ra2e1_swap_with_signature`.

In this example, the Value of MCUBOOT_IMAGE_SIGNING_KEY is configured to:

```
${workspace_loc:ra_mcuboot_ra2e1_overwrite_with_signature}/ra/mcu-
tools/MCUboot/root-ec-p256.pem
```

If there is no signature verification, then it is not necessary to set the Environment Variable: MCUBOOT_IMAGE_SIGNING_KEY as are the cases for `ra_mcuboot_ra2e1` and `ra_mcuboot_ra21_swap`.

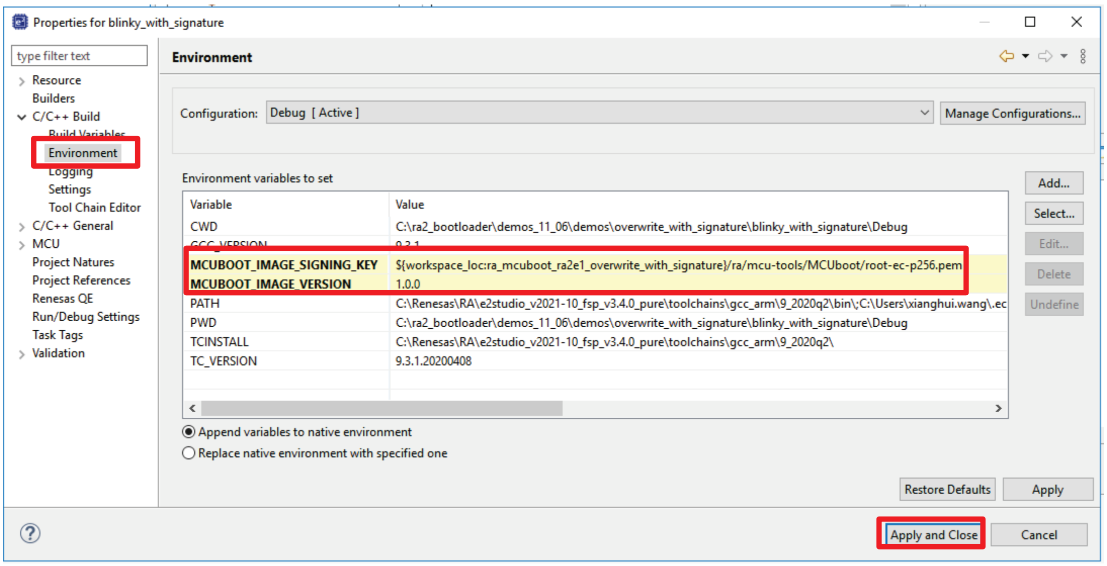


**Figure 51. Configure the Application Image Version Number and Signing Key**

To be able to always recompile the project when the Environment Variables or the linker script are updated, it is recommended add a **Pre-build** step to always delete the `.elf` file as shown in Figure 52.

```
rm -f ${ProjName}.elf
```

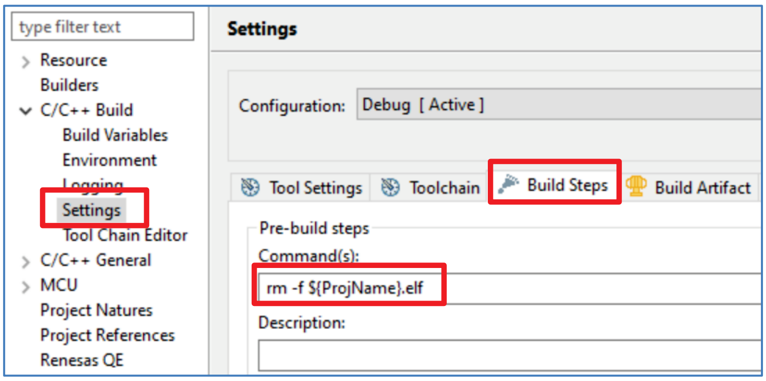


**Figure 52. Configure the Pre-build Command**

Next, you can add the RTT Viewer usage related application code to the primary application project. Unzip RA2_secure_bootloader.zip, open the RA_secure_bootloader\<boot_project_name>\<Initial application project name>\src folder and copy all files under \src to the \src folder for the newly established project.

At this point, you can click **Generate Project Content** and compile the newly created application project and ensure \debug\<Initial application project name>.bin.signed is generated.

# 5. Booting the Initial Application Project

## 5.1 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.

## 5.2 Configure the Debugger

Open the Debug Configurations: **blinky > Debug As > Debug Configurations**

Optional Step: Set Allow caching of flash contents to No, as shown in Figure 53. Otherwise, the debugging bootloader applications memory window information may show wrong information.

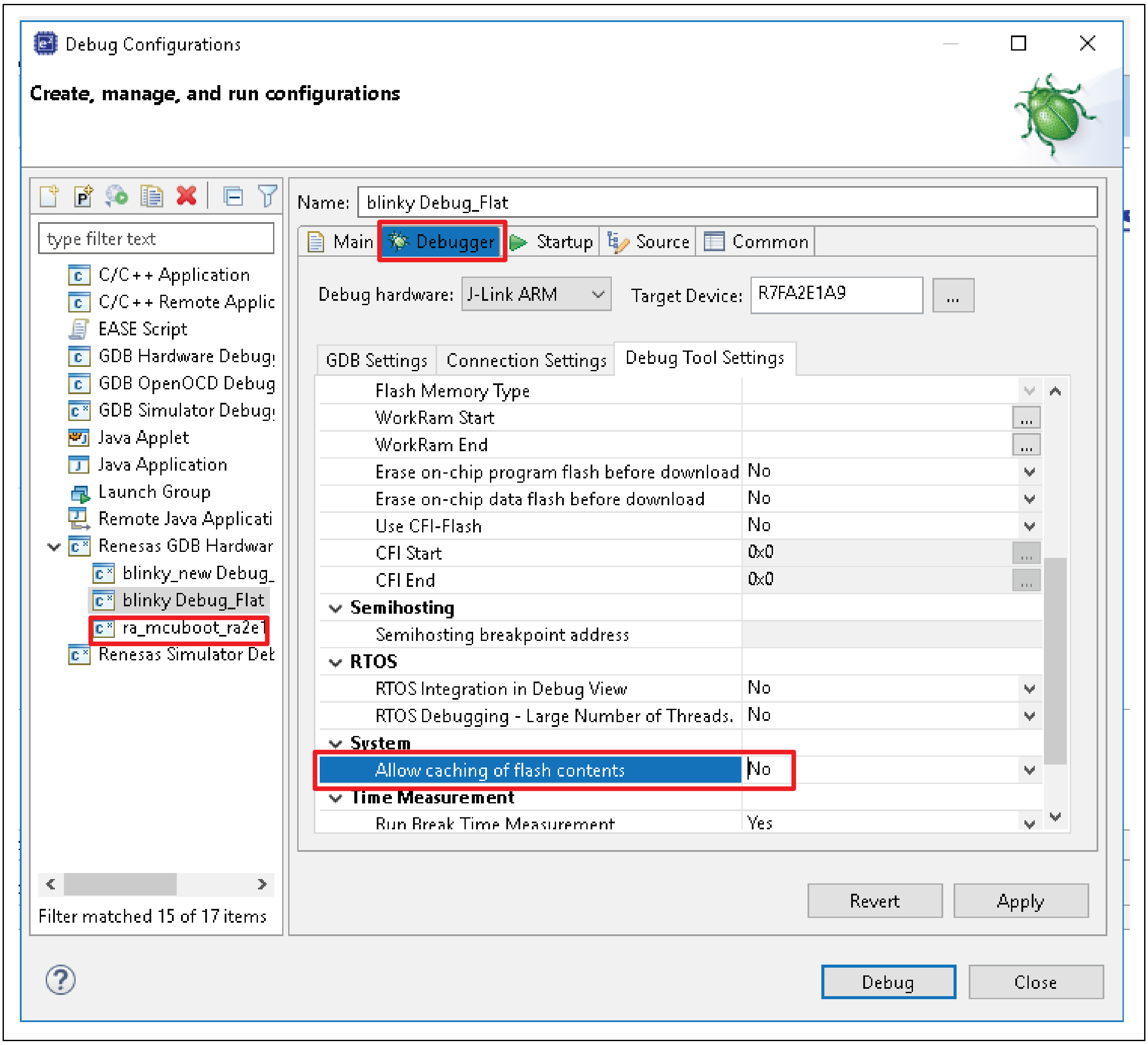


**Figure 53. Disable Flash Content Caching**

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

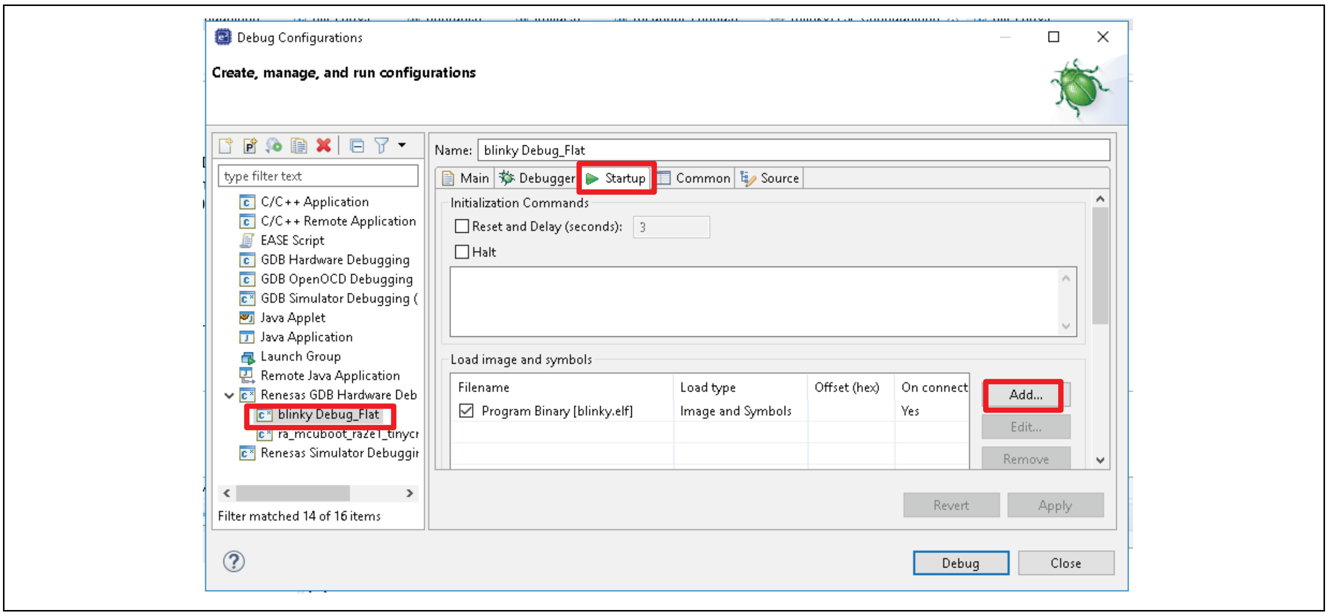

**Figure 54. Configure the Primary Project Debug Startup**

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

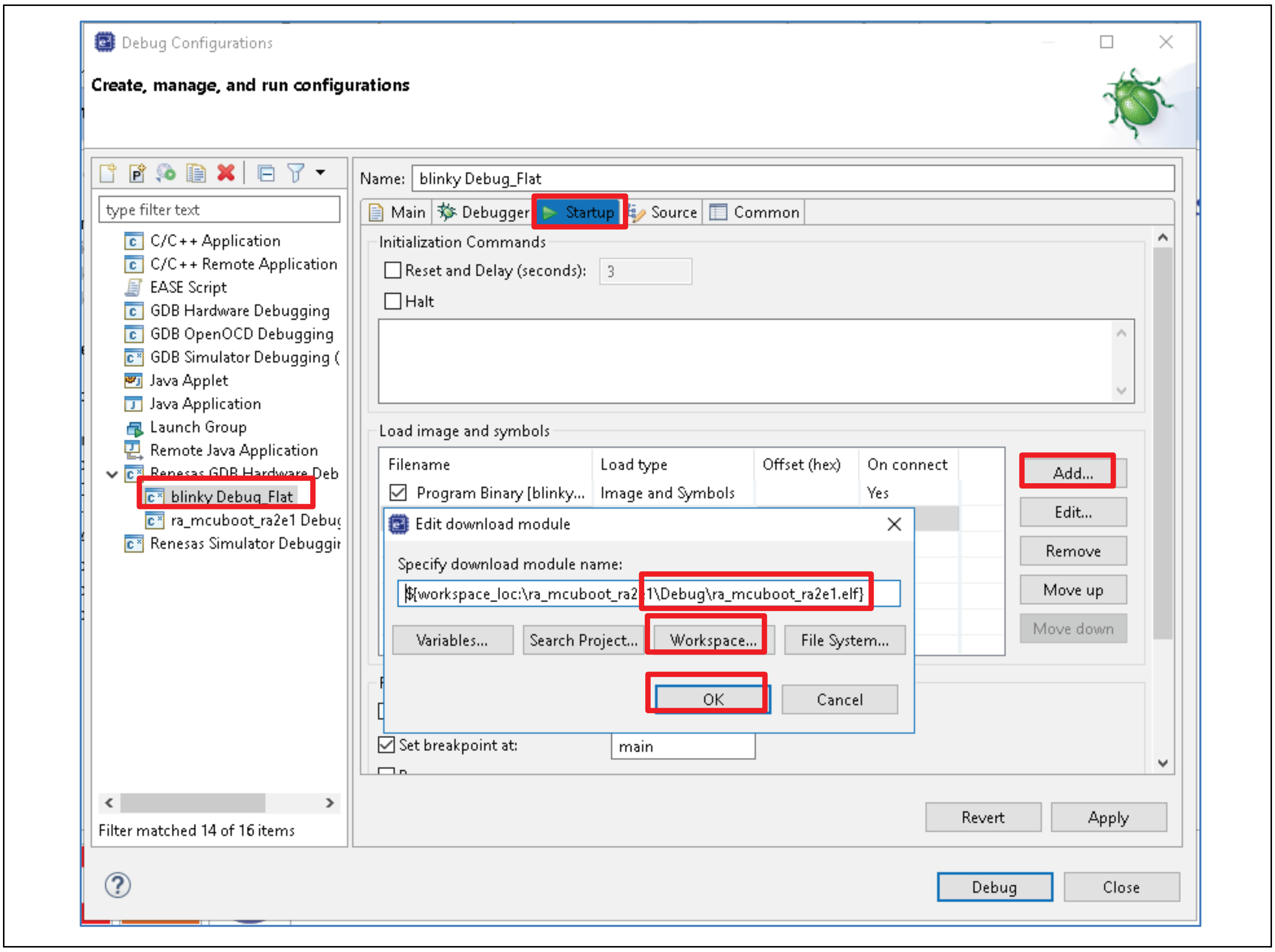

**Figure 55. Add the Bootloader Project to Debug Configuration**

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.

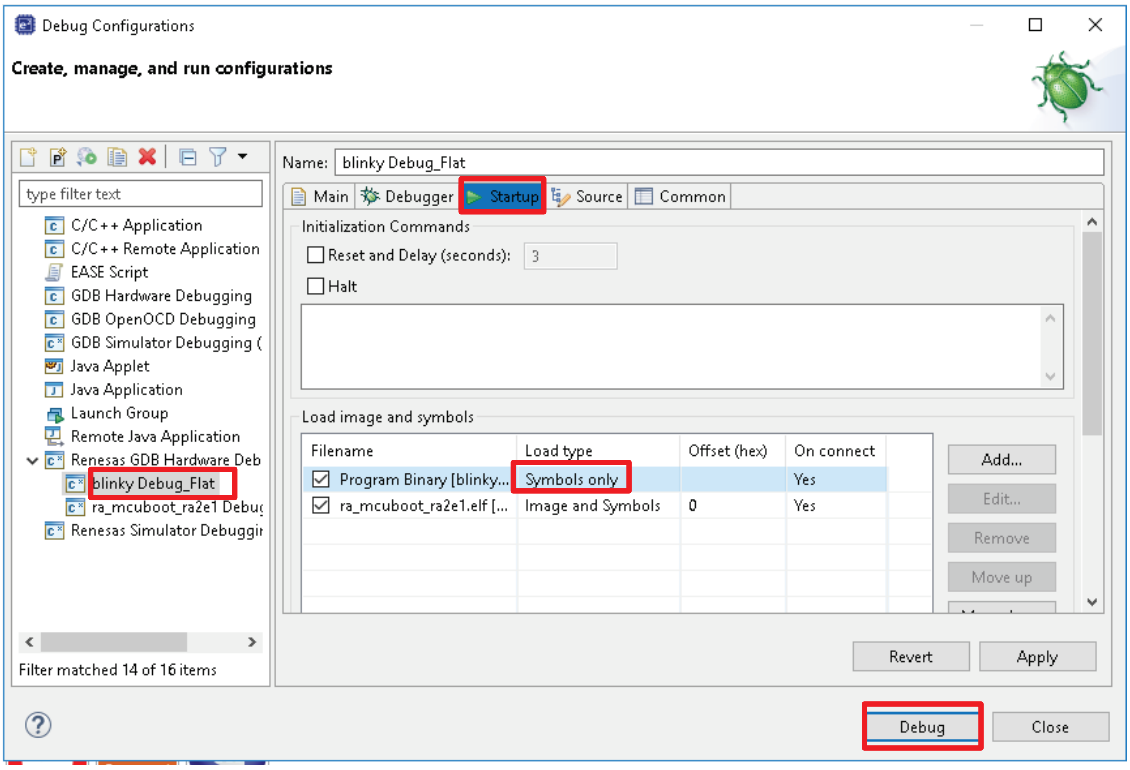


**Figure 56. Select to Load Symbols Only for the Application Project**

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 **<boot_project_name>** and select the **<boot_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 Table 3). Figure 57 is an example of downloading the signed primary image for the overwrite without signature project.

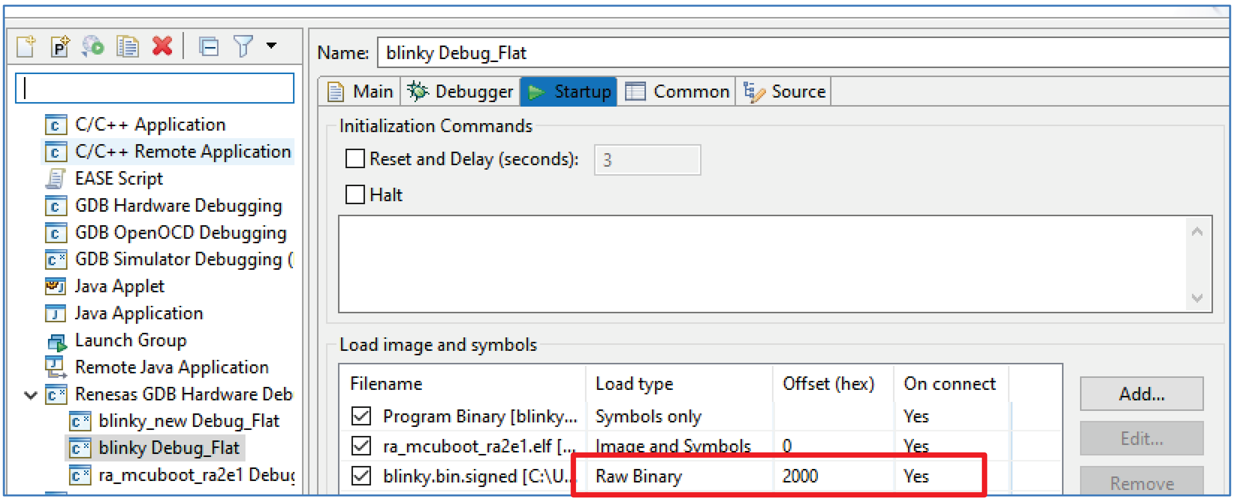


**Figure 57. Include the Raw Binary of the signed image in the download**

Click **Debug**. The debugger should hit the reset handler in the bootloader.

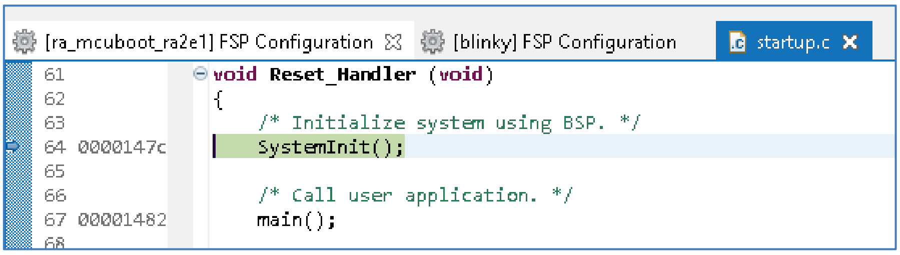


**Figure 58. Start the Application Execution**

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.

Open the JLink RTT Viewer and set up the following configurations.

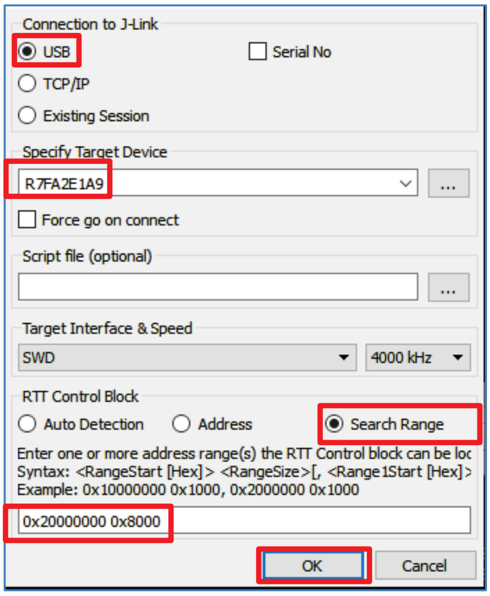


**Figure 59. Configure the RTT Viewer**

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.

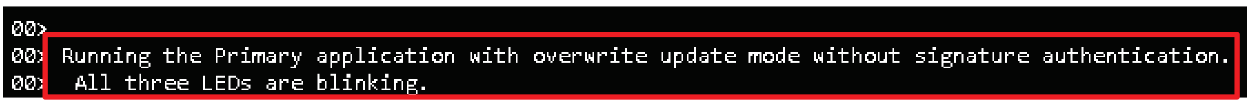


**Figure 60. RTT Viewer Output from the Primary Application**

## 6. 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.zip`. You can also follow section 7 to exercise these projects without going through the new application creation and mastering process described in this section if desired.

### 6.1 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.

Right-click in the white space in the **Project Explorer** area and select **Import**.

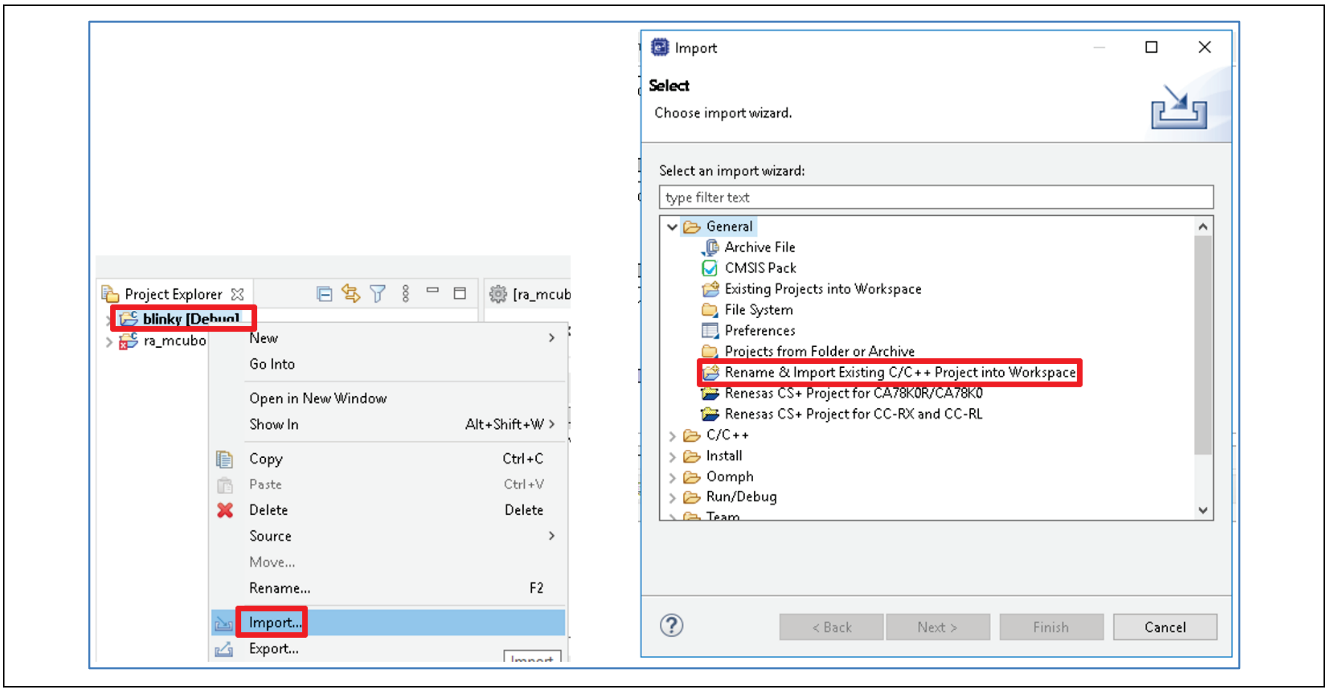


**Figure 61. Select Rename and Import the Primary Application**

Once the **Import** window opens, name the project and click **Browse** for **Select root directory** as shown in Figure 62.

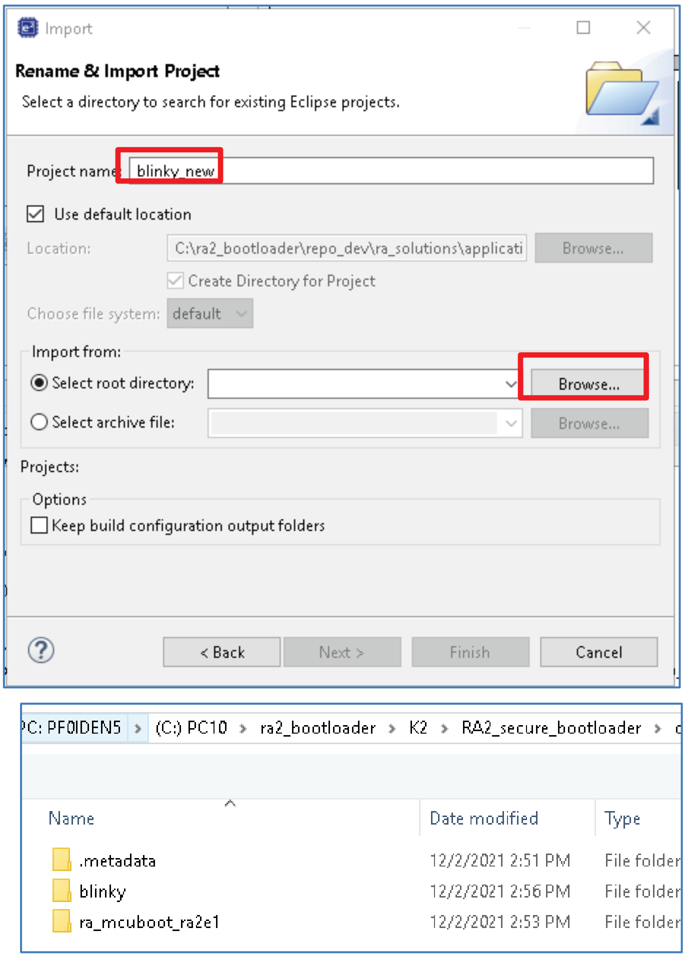


**Figure 62. Rename the Project**

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 | blinky | blinky_new |
| ra_mcuboot_ra2e1_overwrite_with_signature | blinky_with_signature | blinky_with_signature_new |
| ra_mcuboot_ra2e1_swap | blinky_swap | blinky_swap_new |
| ra_mcuboot_ra2e1_swap_with_signature | blinky_swap_with_signature | blinky_swap_with_signature_new |
| ra_mcuboot_ra2e1_dxip | blinky_primary | blinky_secondary |

Figure 63 is an example screenshot when importing the blinky project as **blinky_new**.

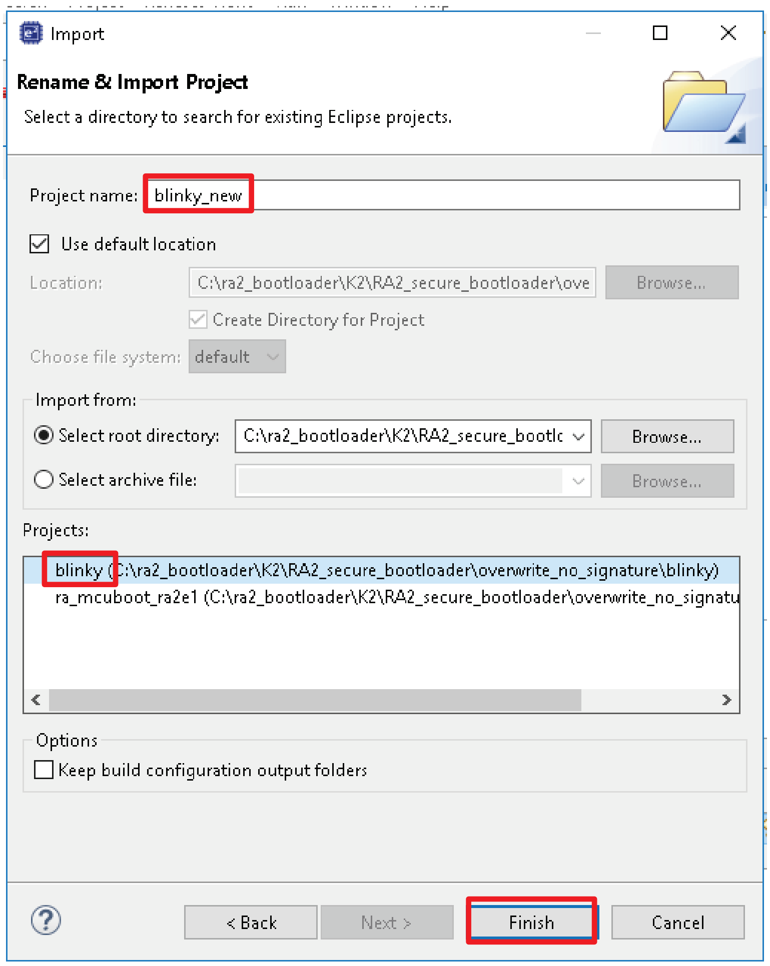


**Figure 63. Import blinky Project as blinky_new**

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, user can unzip `RA2_secure_bootloader.zip`, open the `\<boot_project_name>\<new application project name>\src` folder and copy all files under `\src` to the newly established project `\src` folder.

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, `blinky_new.bin.signed` will be created.

For cleanness of the project, user 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 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 `blinky_new` application rather than the signed binary of the old blinky application.

For example, when using the application projects for the `ra_mcuboot_ra2e1`, we want to change the debug configuration of the `blinky_new` project from the imported result shown in Figure 65:

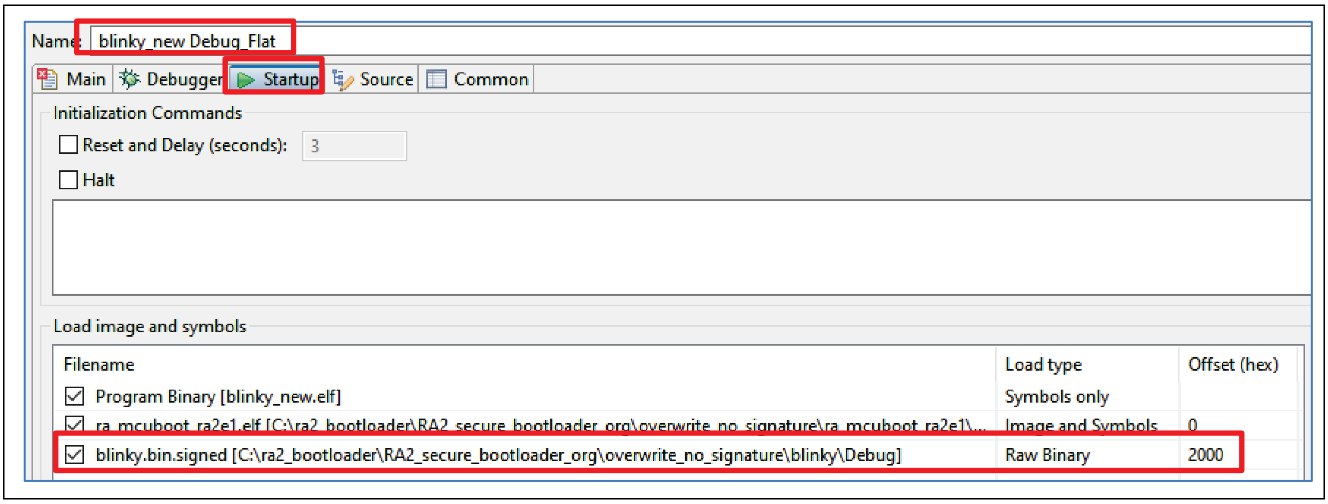


**Figure 64. Debug Configuration of blinky_new initially imported**

In the imported configuration, the signed binary of the blinky project is used. We need to change that to the signed binary of `blinky_new` as shown in Figure 65.

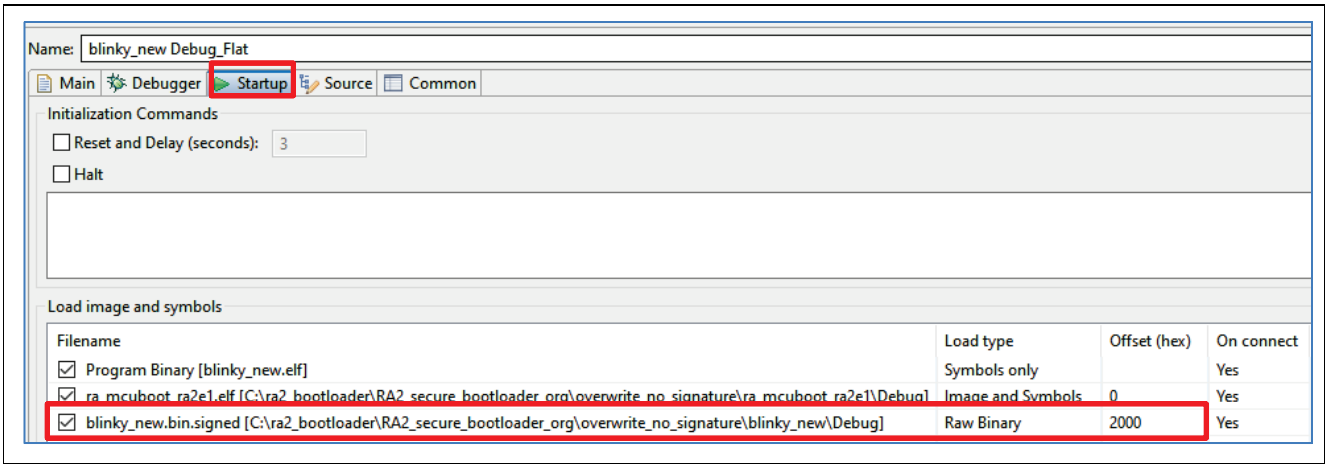


**Figure 65. Debug Configuration of blinky_new to use for debugging**

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. For Direct XIP, we can set the download address of the signed new image binary to the location of the intended slot.

To create a brand-new application when using the overwrite, swap or Direct XIP upgrade mode without importing the previous application, you can follow section 4.2 to configure the application to use the bootloader and section 4.3 to sign the application image.

## 6.2 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 13 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 based on whether the image_ok byte which resides 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 again swapped 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 6.2.1 and 6.2.2.

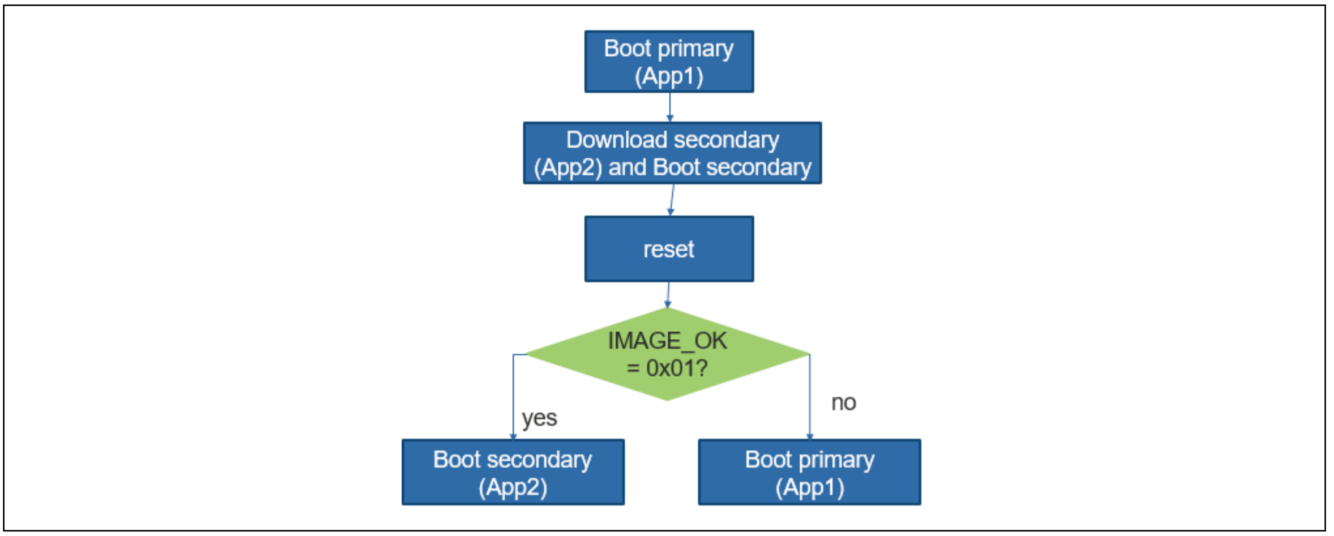


**Figure 66. Swap Test Mode**

### 6.2.1 Confirming the New Application at Compile Time

FSP 3.4.0 or earlier only supports confirming the new image at compile time. FSP 3.5.0 or later supports runtime image confirmation of flat projects.

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 16. This usage is demonstrated in the example bootloader `blinky_swap_new`.

### 6.2.2 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 17. 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 projects demonstrate this feature. This module is included in the example bootloader `blinky_swap_with_signature_new`.

Open the Secondary application project blinky_swap_with_signature_new, 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 68. Adding this module adds about 2 kB of flash usage in the application.

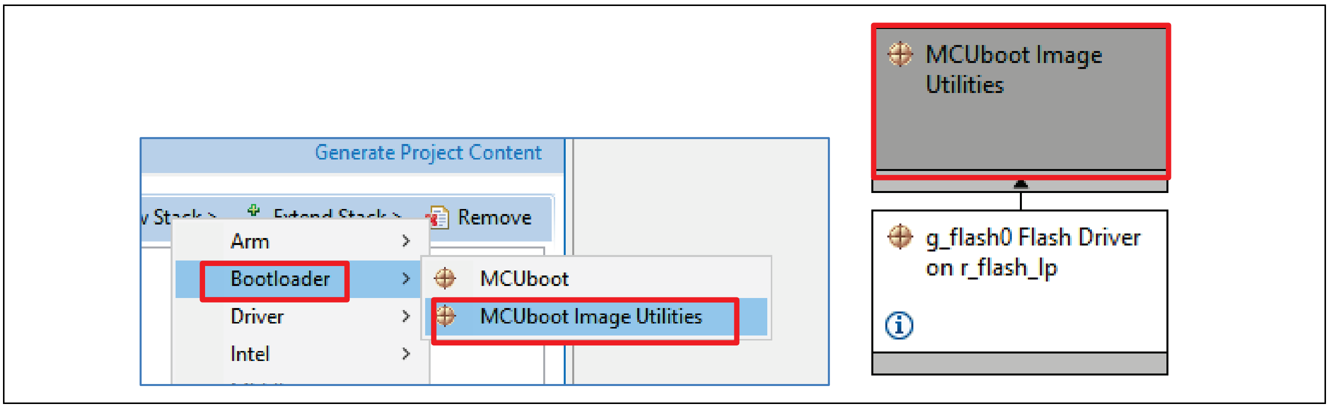


**Figure 67. Include the MCUboot Image Utilties Module**

Configure the path of the header files needed.

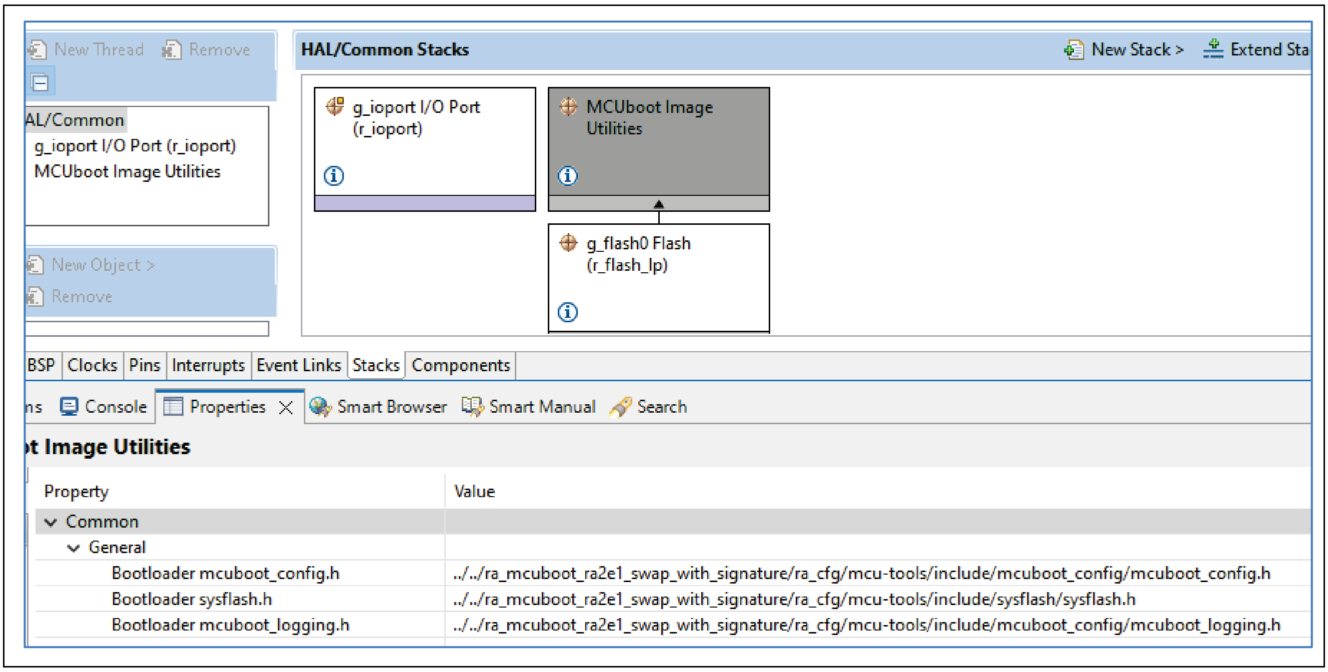


**Figure 68. Include the Bootloader Header Files**

Next configure the **r_flash_lp** module in the same way as in Figure 23.

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 `\swap_with_signature\blinky_swap_with_signature_new\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 69. Confirm the Update Image**

## 6.3 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 `blinky_new_signed.bin`. 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 Table 3. For example, for the overwrite only without signature bootloader `ra_mcuboot_ra2e1`, the download address should be **0x4000**.

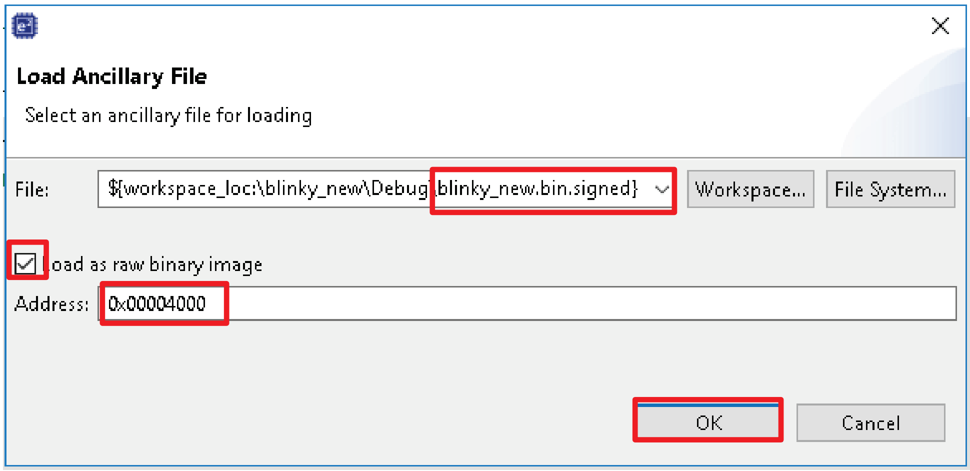


**Figure 70. Download the Secondary Application Image**

Note that for user-created customized applications, the download address needs to be adjusted by referencing the specific flash layout. User can reference Table 3 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.

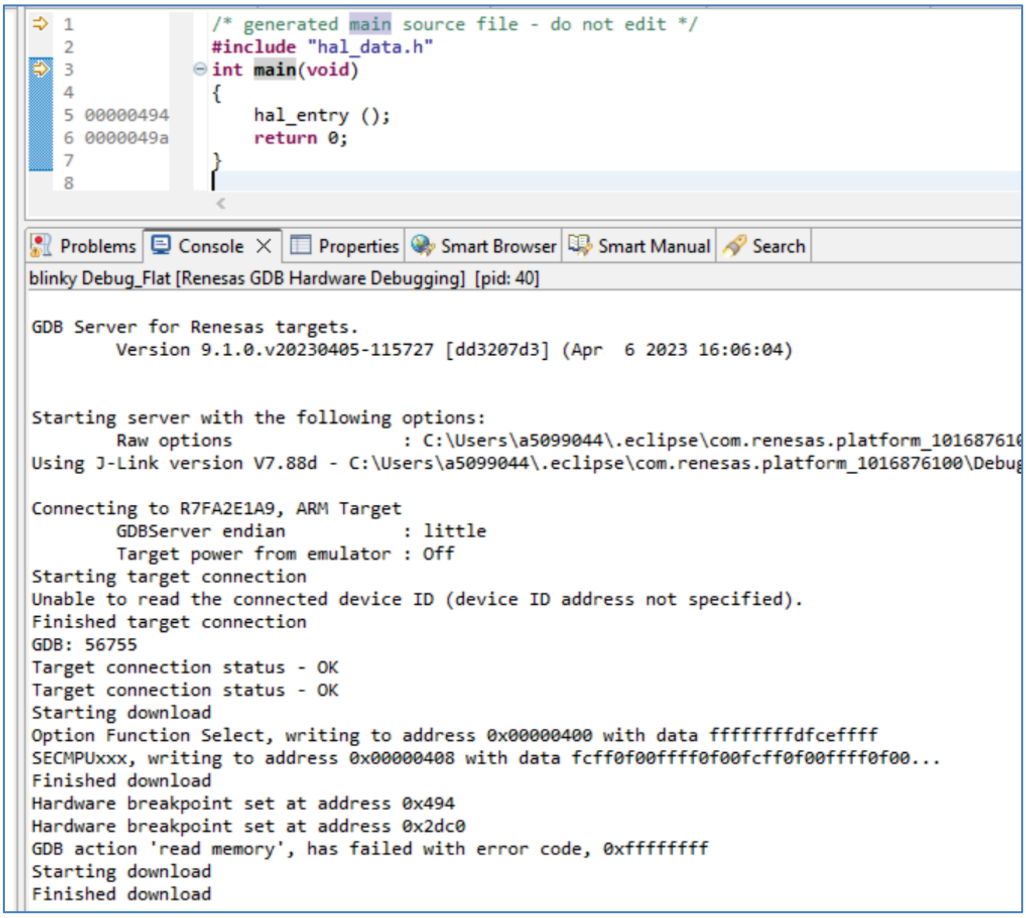


**Figure 71. GDB Actions when using the Load Ancillary File Button**

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 including the upgrade mode, whether signature authentication is supported as well as what LEDs are blinking. Below is an example when `blinky_overwrite_with_signature_new` is booted.

```
00> Running the Secondary (New) application with overwrite update mode without signature authentication.
00>  Only the blue LED is blinking.
```

**Figure 72. RTT Viewer Output from the New Application**

Prior to deployment, a system with bootloader solution would typically need to include a image downloader and programmer in the application (primary and secondary applications), so a new application can be downloaded in the field.

Application project *RA6 Secure Firmware Update using MCUboot and Flash Dual Bank (R11AN0570)* includes an image downloader using XModem over UART interface. User can reference that to create an image downloader.

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

Unzip `RA2_secure_bootloader.zip` to access the included bootloader and example application projects.

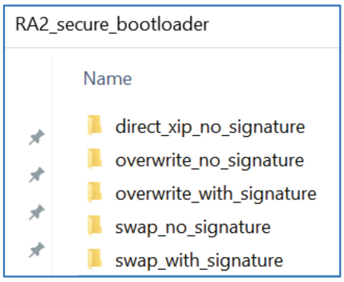


**Figure 73. Example Projects Included**

## 7.1 Running the Example Projects with Overwrite Upgrade Mode

### 7.1.1 Without Signature Verification

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

1. Import projects to a workspace.
2. Open the `configuration.xml` file from project `ra_mcuboot_ra2e1`.
3. Click **Generate Project Content**.
4. Compile the project `ra_mcuboot_ra2e1`.
5. Open the `configuration.xml` file from project `blinky`.
6. Click **Generate Project Content**.
7. Compile the `blinky` project.
8. Open the `configuration.xml` file from project `blinky_new`.
9. Click **Generate Project Content**.
10. Compile the `blinky_new` project.
11. Debug the application from project `blinky`.
12. Resume the program execution twice. All LEDs should be blinking.
13. Pause the execution.
14. Download the `blinky_new.bin.signed` using Load Ancillary File to address 0x4000.
15. Resume the program execution. All LEDs should be blinking.
16. Update the Environment variable of `blinky_new`: MCUBOOT_IMAGE_VERSION from 0.9. 0 to 1.0.0.
17. Recompile project `blinky_new`.
18. Pause the debug session.
19. Download the `blinky_new.bin.signed` using Load Ancillary File to address 0x4000.
20. Resume the program execution. The blue LED should be blinking.

### 7.1.2 With Signature Verification

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

1. Import projects to a workspace.
2. Open the `configuration.xml` file from project `ra_mcuboot_ra2e1_overwrite_with_signature`.
3. Click **Generate Project Content**.
4. Compile the project `ra_mcuboot_ra2e1_overwrite_with_signature`.
5. Open the `configuration.xml file` from project `blinky_with_signature`.
6. Click **Generate Project Content**.
7. Compile the `blinky_with_signature` project.
8. Open the `configuration.xml` file from project `blinky_with_signature_new`.
9. Click **Generate Project Content**.
10. Compile the `blinky_with_signature_new` project.
11. Debug the application from project `blinky_with_signature`.
12. Resume the program execution twice. All LEDs should be blinking.
13. Pause the execution.
14. Download the `blinky_with_signature_new.bin.signed` to address 0x5800.
15. Resume the program execution, the blue LED should be blinking.

## 7.2 Running the Example Projects with Swap Upgrade Mode

### 7.2.1 Without Signature Verification

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

1. Import projects to a workspace.
2. Open the `configuration.xml` file from project `ra_mcuboot_ra2e1_swap`.
3. Click **Generate Project Content**.
4. Compile the project `ra_mcuboot_ra2e1_swap`.
5. Open the `configuration.xml` file from project `blinky_swap`.
6. Click **Generate Project Content**.
7. Compile the `blinky_swap` project.
8. Open the `configuration.xml` file from project `blinky_swap_new`.
9. Click **Generate Project Content**.
10. Compile the `blinky_swap_new` project.
11. Debug the application from project `blinky_swap`.
12. Resume the program execution twice. All LEDs should be blinking.
13. Pause the execution.
14. Download the `blinky_swap_new.bin.signed` using the Load Ancillary File to address 0x5000.
15. Resume the program execution. The blue LED should be blinking.
16. Reset the program execution from e$^2$ studio.
17. Run the application. The blue LED should be blinking.

### 7.2.2 With Signature Verification

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

1. Import projects to a workspace.
2. Open the `configuration.xml` file from project `ra_mcuboot_ra2e1_swap_with_signature`.
3. Click **Generate Project Content**.
4. Compile the project `ra_mcuboot_ra2e1_swap_with_signature`.
5. Open the `configuration.xml` file from project `blinky_swap_with_signature`.
6. Click **Generate Project Content**.
7. Compile the `blinky_swap_with_signature` project.
8. Open the `configuration.xml` file from project `blinky_swap_with_signature_new`.
9. Click **Generate Project Content**.
10. Compile the `blinky_swap_with_signature_new` project.
11. Debug the application from project `blinky_swap_with_signature`.
12. Resume the program execution twice. All LEDs should be blinking.
13. Pause the execution.
14. Download the `blinky_swap_with_signature_new.bin.signed` using Load Ancillary File to address 0x6800.
15. Resume the program execution. The blue LED should be blinking.
16. Reset the program execution from e$^2$ studio.
17. Run the application. The blue LED should be blinking.

## 7.3 Running the Example Project with Direct XIP Upgrade Mode Without Signature

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

1. Import projects to a workspace.
2. Open the `configuration.xml` file from project `ra_mcuboot_ra2e1_dxip`.
3. Click **Generate Project Content**.
4. Compile the project `ra_mcuboot_ra2e1_dxip`.
5. Open the `configuration.xml` file from project `blinky_primary`.
6. Click **Generate Project Content**.
7. Compile the `blinky_primary`.
8. Open the `configuration.xml` file from project `blinky_secondary`.
9. Click **Generate Project Content**.
10. Compile the `blinky_secondary`.
11. Debug the application from project `blinky_primary`.
12. Resume the program execution twice. All LEDs should be blinking.
13. Pause the execution.
14. Download the `blinky_secondary.bin.signed` using Load Ancillary File to address 0x4000.
15. Resume the program execution. The blue LED should be blinking.
16. Reset the program execution from e$^2$ studio.
17. Run the application. The blue LED should be blinking.

# 8. 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)

## 9. 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 | Jul.26.21 | - | First release document |
| 1.10 | Dec.09.21 | - | Update to add swap mode and signature verification support |
| 1.2.0 | Apr.07.23 | - | Update to FSP v4.2.0. Add Direct XIP mode and use new e[2] studio features |
| 1.3.0 | Sep.07.23 | - | Update to usage mode based on FSP v4.5.0. Correct project recreation missing steps. |

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

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

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between $V_{IL}$ (Max.) and $V_{IH}$ (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between $V_{IL}$ (Max.) and $V_{IH}$ (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

## Corporate Headquarters

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


## Trademarks



## 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/.