Renesas RA Family

RA6 Booting Encrypted Image using MCUboot and QSPI

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 independent of operating system and hardware and relies on hardware porting layers from the operating system it works with. The Renesas Flexible Software Package (FSP) integrates an MCUboot port starting from FSP v3.0.0. Users can benefit from using the FSP MCUboot Module to create a Root of Trust (RoT) for the system and perform secure booting and fail-safe application updates.

The MCUboot is maintained by Linaro in the GitHub mcu-tools page https://github.com/mcu-tools/mcuboot. There is a `\docs` folder that holds the documentation for MCUboot in `.md` file format. This application note refers to the above-mentioned documents wherever possible and is intended to provide additional information that is related to using the MCUboot module with Renesas RA FSP v3.0.0 or later.

To provide confidentiality of image data while in transport to the device or while residing on an external flash, MCUboot has support for encrypting/decrypting images on-the-fly while upgrading. When upgrading the image from the secondary slot to the primary slot, it is automatically decrypted after validation. Image encryption is supported by FSP v3.8.0 or later.

For using MCUboot module with the internal flash in code flash linear mode without encryption support for the RA6 Family MCUs, user can reference application project (R11AN0497). This application project should be reviewed and followed if users want to create a MCUboot based secure bootloader from scratch.

For the Booting Encrypted Image using MCUboot and QSPI application project, a set of secure bootloader and matching application projects using MCUboot and internal code flash without encryption is included. This application project then walks the user through the updates to the bootloader to add encryption for the QSPI based secondary image storage.

The example projects included in this application project are based on the EK-RA6M4 evaluation kit. The application examples implemented image downloading to the QSPI secondary slot over USB PCDC. MCUboot with encryption also supports internal flash encryption. The operations are very similar to the QSPI usage and are not demonstrated in this application project.

For using MCUboot module with the internal code flash dual bank mode without encryption support for the RA6 Family MCUs, user can reference application project (R11AN0570).

Required Resources

Development tools and software

- The e² studio ISDE v2024-07
- Renesas Flexible Software Package (FSP) v5.5.0
- SEGGER J-link® USB driver

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.

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

Hardware

- EK-RA6M4 Evaluation Kit for RA6M4 MCU Group (http://www.renesas.com/ra/ek-ra6m4)
- Workstation running Windows® 10 and Tera Term console, or similar application
- Two USB device cables (type-A male to micro-B male)

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e^2 studio IDE and Arm® TrustZone® based development models with e^2 studio. Users are required to read the entire FSP User's Manual on the MCUboot Port section and review the RA6 Basic Secure Bootloader Design using MCUboot Application Project (R11AN0497) 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, or end users who are involved with designing application systems involving usage of a secure bootloader.

Using this Application Note

Section 1 covers the general overview of MCUboot and the application upgrade methods supported by the MCUboot. If you have worked with MCUboot module-based bootloader previously, this section can be bypassed.

Section 2 covers the general flow of architecting a system using FSP MCUboot module. If you have previously worked with the MCUboot system using FSP, this section can be bypassed.

Section 3 covers the walk throughs of running the initial example projects which do not include encryption support. These example projects use swap test update mode and internal code flash for both primary and secondary applications. Image downloader using XModem over USB PCDC is implemented in the primary and secondary applications. MCUboot provided example keys are used for image signing and encryption support.

Section 4 covers adding encryption support to the bootloader and applications using internal code flash for both the primary and secondary applications.

Section 5 covers updating the projects created in section 4 to use QSPI for secondary image storage. Note that for the user's convenience, an end solution for this section is provided for the user's reference.

Section 6 covers using custom image signing and image encryption keys in the projects created in Section 5.

Section 7 covers production-related topics.

Contents

1. MCUboot Functionalities Overview

MCUBoot handles the firmware authenticity check after start-up and the firmware switch part of the firmware update process. Downloading the new version of the firmware is out-of-scope for MCUBoot. Typically, downloading the new version of the firmware is functionality that is provided by the application project itself. This application project provides an example of this functionality using XModem transfer protocol over USB PCDC port to download image to the external QSPI secondary image storage area.

1.1 Validate Application before Booting and Updating

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 when using the internal code flash.

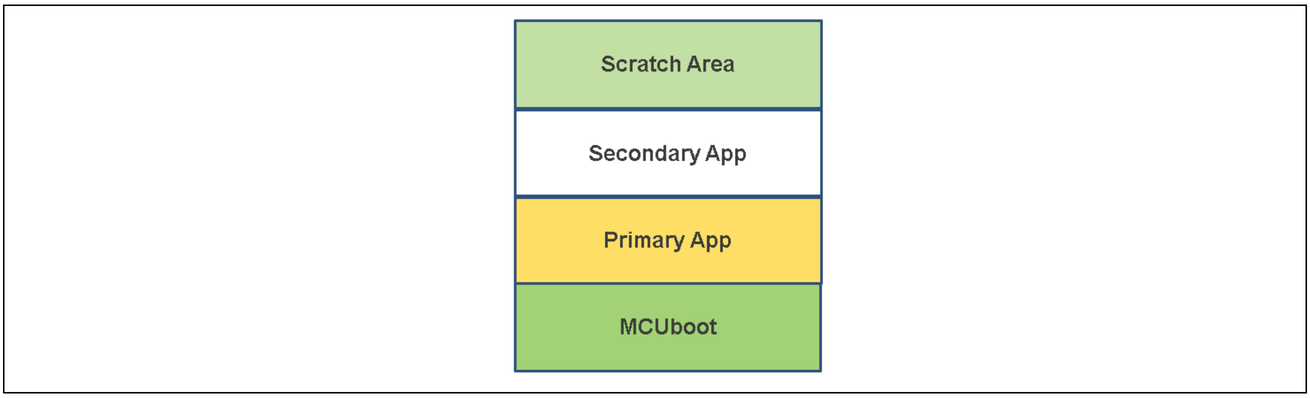

Figure 1. Single Image MCUboot Memory Code Flash Map

The following is an example of the single image MCUboot memory map when using external flash storage as the secondary storage area.

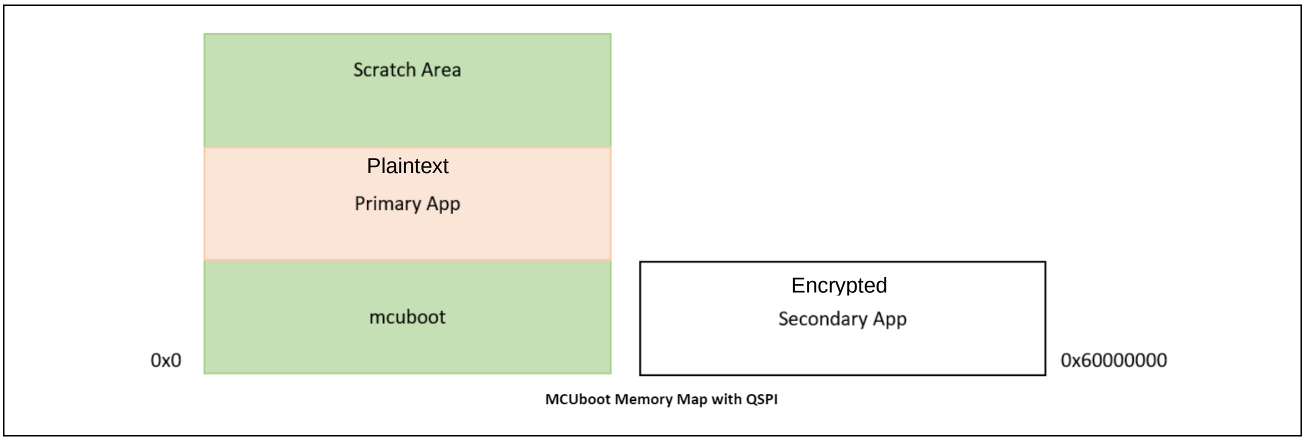

Figure 2. Single Image MCUboot Flash Memory Map with QSPI

For more information on the MCUboot memory layout, refer to the [Flash Map section] of the reference MCUboot website.

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

The bootloader starts when CPU is released from reset. For TrustZone®-based MCUs, MCUboot is designed to run in Secure mode with all access privileges available to it. If there are images in the Secondary App memory marked as to be updated, the bootloader performs the following actions:

1. The bootloader will authenticate the Secondary image.
2. Upon successful authentication, the bootloader will switch to the new image based on the selected update method. Available update methods are introduced in section **1.1.1.**
3. The bootloader will boot the new image.

If there is no new image in the Secondary App memory region, the bootloader will authenticate the Primary applications and boot the Primary image.

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

The image header needs to flag this image as ENCRYPTED (0x04) and a TLV with the key must be present in the image.

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

1.1.1 Encrypted Applications Update

The major use case for encrypted image update is for external flash update image storage. External flash content is prone to theft in many ways. It is critical to secure the external flash secondary image storage area via encryption. Another relatively rare use case is the internal flash update image storage if the image is downloaded via insecure channel.

Encrypted image boot is supported with swap and overwrite upgrade mode on all RA MCUs via FSP. Direct XIP upgrade mode is not supported. The cryptographic operation for RA MCU is supported by MbedCrypto and TinyCrypt. User can reference **Table 1** for the selection of the cryptographic library.

We recommend acquiring more details on the upgrade mode by reviewing the corresponding sections in application project (R11AN0497) as well as the MCUboot design page:

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

If swap upgrades are enabled, the image located in the primary slot, also having the ENCRYPTED flag set and the corresponding Type Length Value (TLV) field present, the primary image is re-encrypted while swapping to the secondary slot.

- The image is encrypted using AES-CTR-128, with a counter that starts from zero (over the payload blocks) and increments by 1 for each 16-byte block. AES-CTR was chosen for speed/simplicity and allowing for any block to be encrypted/decrypted without requiring knowledge of any other block (allowing for simple resume operations on swap interruptions). MCUboot also supports AES-CTR-256, this is not supported from FSP side.

2. Architecting an Application with MCUboot Module using FSP

This section provides an overview of the FSP MCUboot module, which integrates MCUboot as a module into the FSP. The available upgrade modes and memory architecture design are discussed. In addition, signing and mastering new images are discussed.

2.1 MCU Memory Configuration using MCUboot Module with FSP

For the general support information, the user can reference the MCUboot port section of the FSP User's Manual.

It is also highly recommended that the user reviews the MCUboot encrypted image page for background on the encryption scheme.

https://github.com/mcu-tools/mcuboot/blob/main/docs/encrypted_images.md

Users can gain hands on experience in configuring the memory regions using the MCUboot module in the walkthrough section in **section 3**, **section 4** and **section 5**.

2.2 Application Image Format for Encrypted Image

Figure 3 is a more detailed application image format that can be referenced to understand the booting process.

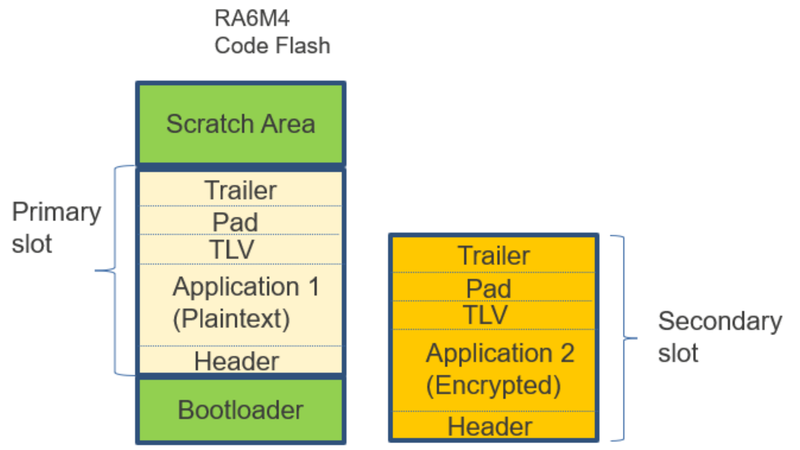

Figure 3. Application Image Format

To signal the bootloader as an encrypted image, the application adds the ENCRYPTED flag in the header area. In addition, the image encryption key is included encrypted in the Trailer area. The key that is used to encrypt the image encryption key is shared between the image encryption process and the image decryption process via ECIES P256 or RSA OAEP 2048.

2.3 Designing Bootloader and the Initial Primary Application Overview

A bootloader is typically designed with an existing initial primary application. The following are the general guidelines for designing the bootloader with 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.
- The bootloader maintains a memory map of all the different images. User needs to perform the memory usage analysis of the application and update the bootloader defined memory map for consistency and adjust as needed.
- When changing the image authentication and image update mode, the bootloader memory allocation may need to be adjusted.

Most of these design aspects are addressed in the walk-through in this application note.

2.4 General Guidelines using the MCUboot Module Across RA Family MCUs

The MCUboot module is supported on all RA Family MCUs. The cryptographic support is provided via MbedTLS Crypto only module and Tiny Crypt module.

Users can reference the following table when choosing the cryptographic module with or without encryption support.

Table 1. Cryptographic Support for RA MCUs

Crypto Stack	RA2 No Encryption	RA2 with Encryption	RA4E1, RA4T1, RA6E1, RA6E2, RA4W1, RA4M1, RA6T2/T3 with or without Encryption *	RA6M1/M2/M3, RA6T1, RA4M2/M3, RA6M4/M5 with or without Encryption
MbedTLS (Crypto Only) HW				x
MbedTLS (Crypto Only) SW			x	

TinyCrypt (HW AES)		x		
TinyCrypt (SW Only)	x			

Note *: some of the MCUs in this group have AES Hardware Support which can be used in the MCUboot based encrypted application booting. Please refer to the Hardware User's Manual to understand if this security feature exists on the MCU of interest.

2.5 Customize the Bootloader

The following are some aspects that need to be considered when customizing the bootloader in a product design.

- Customized method to download the application.
- Adjust the flash memory allocation in the bootloader project for the bootloader as well as the application image.

Porting the EK-RA6M4 example bootloader and application projects to EK-RA6M3 and EK-RA6M5:

- The user is recommended to recreate the project with all the stack components in e^2 studio. In this step, the bootloader size and image size can be adjusted based on the MCU flash memory size and the application image size.
- There is no code update needed when porting the included example projects to RA6M3 and RA6M5. After the configurator stack is created, the user can copy over the application source code under \src folder to the newly created project \src folder.

2.6 Production Support

2.6.1 Key Provisioning

By default, the public key is embedded in the bootloader code and its hash is added to the image manifest as a `KEYHASH TLV` entry. See **section 6** for more details about the public key and private key which are used for testing purposes. For production support, the user needs to follow the example shown in `key.c` to add their public key. A more secure solution is to inject the image verification public key. In addition, the user needs to update the private key for application image signing. This application project provides examples of how to use `imgtool.py` to create custom image signing keys and encryption keys in **section 6**.

As an alternative, the bootloader can be made independent of the included test keys by setting the `MCUBOOT_HW_KEY` option. In this case the hash of the public key must be provisioned to the target device and MCUboot must be able to retrieve the key-hash from there. For this reason, the target must provide a definition for the `boot_retrieve_public_key_hash()` function that is declared in `boot/bootutil/include/bootutil/sign_key.h`. It is also required to use the full option for the `--public-key-format` imgtool argument in order to add the whole public key (`PUBKEY TLV`) to the image manifest instead of its hash (`KEYHASH TLV`).

During boot, the public key is validated before it is used for signature verification. MCUboot calculates the hash of the public key from the TLV area and compares it with the key-hash that was retrieved from the device. This way, MCUboot is independent from the public key(s). The key(s) can be provisioned any time and by different parties.

2.6.2 Make the bootloader immutable for enhanced security

For Cortex-M33 MCU, refer to **section 7.1** to make the bootloader immutable. For Arm® Cortex-M4 MCU, refer to **section 7.2** to make the bootloader immutable.

2.6.3 Advance the device lifecycle states prior to the deploy the product to the field

For Cortex-M33 MCU, user can refer to **section 7.3** for the device lifecycle management of the MCU. For Cortex-M4 MCU, user can refer to **section 7.4** for the device lifecycle management of the MCU.

3. Running the Initial Example Projects

This section provides a walkthrough of running the included initial example projects. The initial projects use internal flash for both primary and secondary applications. To demonstrate the image encryption support, instructions on how to add encryption support to these projects and change the secondary slot from the internal flash to external QSPI are provided in the next section.

To learn how to establish a bootloader using MCUboot module from scratch, user can reference application project R11AN0497.

Prior to signing the application project, the Python package needs to be installed. The instructions on how to install the Python components used for MCUboot is included in **section 3.2.3**.

Unzip `MCUboot_Encryption_Initial_Projects.zip` you can see there are three projects:

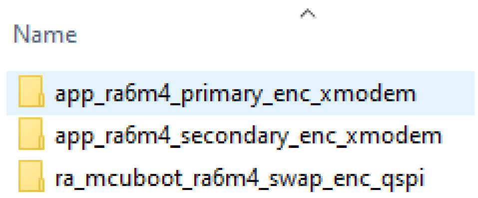

Figure 4. Initial Example Projects

The description for these projects is provided in the following table.

Table 2. Description of the Initial Example Projects

Projects	Description
app_ra6m4_primary_enc_xmodem	Primary application: • Blinky thread blinks three LEDs (red, green, blue) • Downloader thread implemented XModem over USB PCDC support.
app_ra6m4_secondary_enc_xmodem	Secondary application: • Blinky thread blinks blue LED. • Downloader thread implemented XModem over USB PCDC support.
ra_mcuboot_ra6m4_swap_enc_qspi	The bootloader project: • The bootloader is configured with swap upgrade mode. • Swap test mode is enabled in the secondary application. • The maximum application image size is configured. • All application images are plaintext. • Secondary slot is in internal code flash. • Code flash is linear mode.

In this section, we will run the example projects through the following stages.

First, we will erase the MCU. Then we will download the primary application to the internal flash.

In the next stage, we can use the image downloader implemented in the primary application to download the secondary image to the secondary slot. Upon the next reboot, the secondary image will be booted.

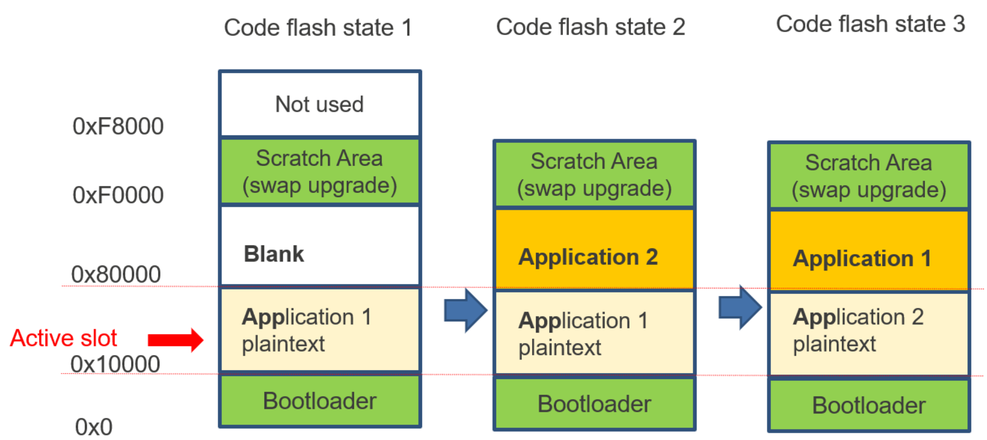

Figure 5. Operational Flow with Swap Update Mode

Note that in the initial application projects, the application image size is defined as 0x70000 which is the maximum application image size based on the example bootloader included when using internal flash for primary and secondary image storage with code flash linear mode.

3.1 Set Up the Python Image Signing Environment

Download and Install Python v3.9 or later.

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

Set up the Python development environment by following **section 3.2, step 3.2.3**. Note that this step only needs to be performed once.

3.2 Running the Initial Example Projects

Use the following steps to run the included initial example projects. The instructions on establishing the initial bootloader are provided in the application project R11AN0497 which is available for download on Renesas website.

3.2.1 Set Up the Hardware

- The default jumper setting of EK-RA6M4 is used for the example projects. In particular, ensure USB FS device mode is set up properly: connect pin 2, 3 on J12, conn ect jumper J15.
- Connect J10 (USB Debug) using a USB micro to B cable from EK-RA6M4 to the development PC to provide power and debug connection using the on-board debugger.
- Connect J11 (USB FS) using a USB micro to B cable from EK-RA6M4 to the development PC to provide USB Device connection.

Once the EK-RA6M4 is powered up, the user needs to initialize the MCU prior to exercising the bootloader project. This will create a clean environment to start the bootloader project verification.

Erase the entire MCU flash using J-Flash Lite.

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

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

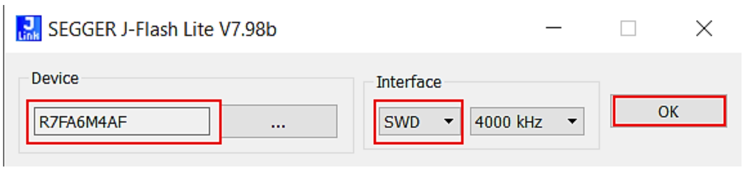

Figure 6. Launch the J-Flash Lite

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

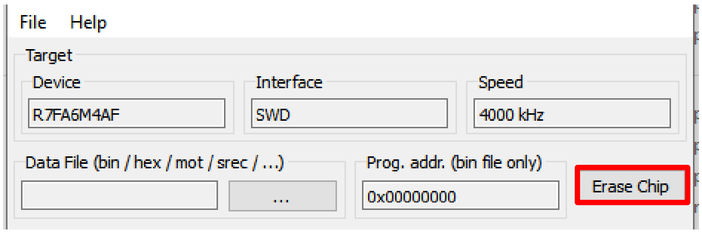

Figure 7. Select Erase Chip

3. Ensure the erase is successful.

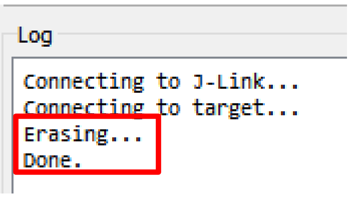

Figure 8. Erase Successful

3.2.2 Import the Projects

For new users, please refer to the FSP User's Manual section on Importing Projects into the IDE for guidelines.

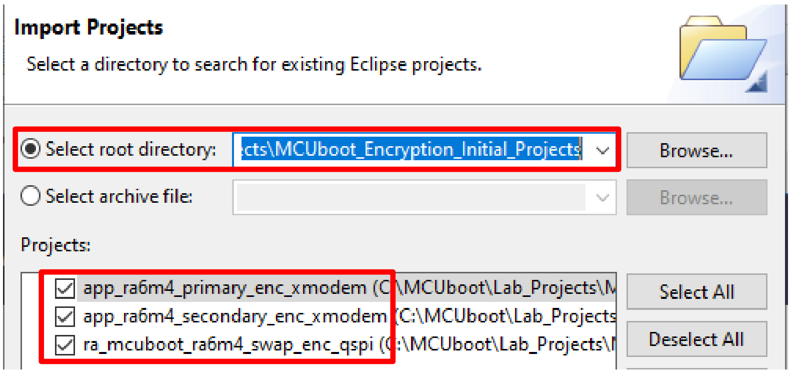

Figure 9. Initial Example Projects

3.2.3 Configure the Python Signing Environment

If this is **NOT** the first time you have used the python script signing tool on your computer, you can skip to **section 3.2.4**.

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.

In the `ra_mcuboot_ra6m4_swap_enc_qspi` project, open the `configuration.xml` file, click **Generate Project Content**. Navigate to the `ra_mcuboot_ra6m4_swap_enc_qspi>ra>mcu-tools>MCUboot` folder in the **Project Explorer** and select **Command Prompt**. This will open a command window with the path set to the `\mcu-tools\MCUboot` folder.

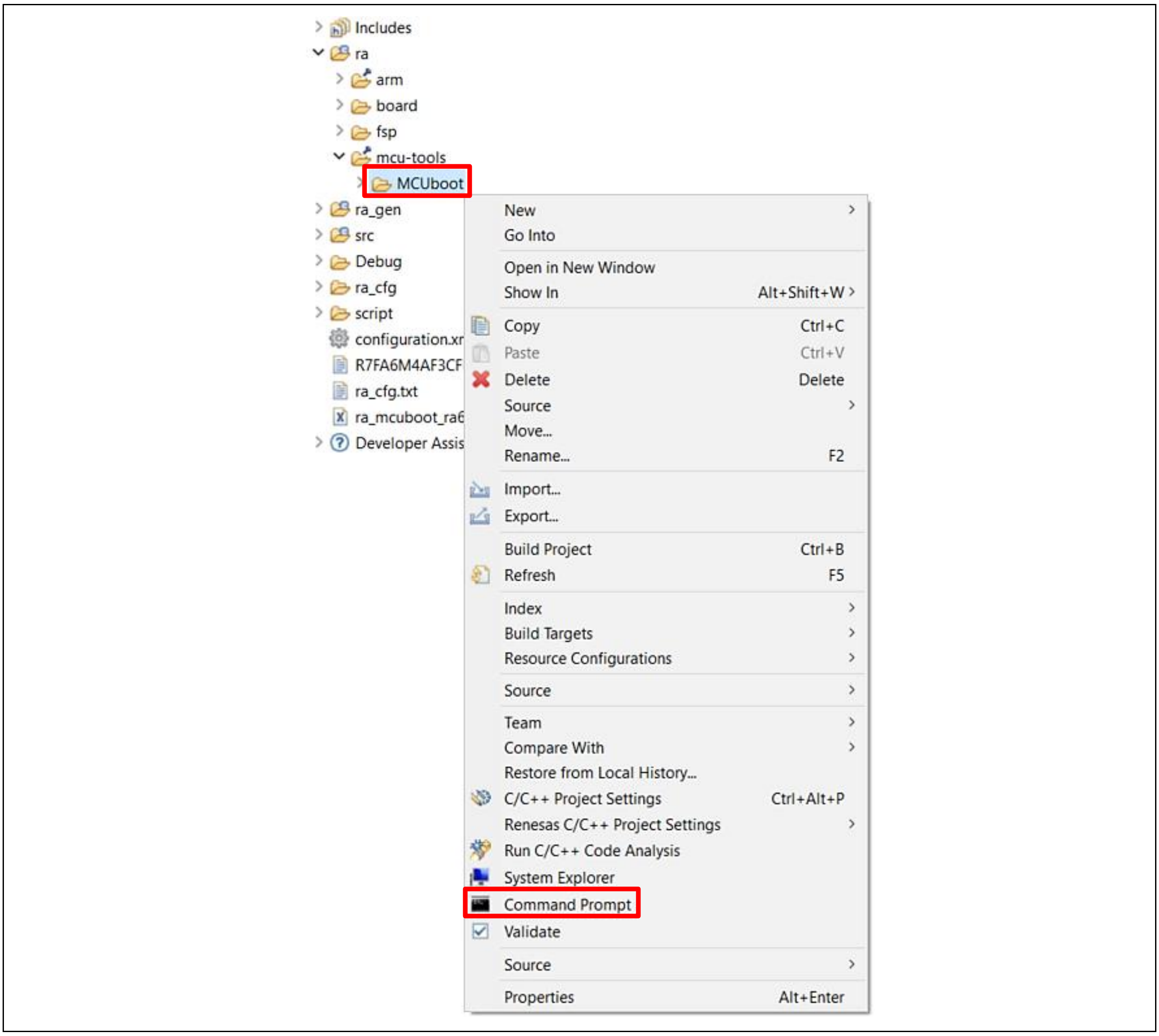

Figure 10. Open the Command Prompt

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.

3.2.4 Compile all the projects

Use the following sequence to build the three projects. For each of these projects, open the `configuration.xml` file, click **Generate Project Content** and then click to build the project.

1. `ra_mcuboot_ra6m4_swap_enc_qspi`
2. `app_ra6m4_primary_enc_xmodem`
3. `app_ra6m4_secondary_enc_xmodem`

The signed image for the application projects is located under the `\Debug` folder:
`/app_ra6m4_primary_enc_xmodem/Debug/app_ra6m4_primary_enc_xmodem.bin.signed`

and

/app_ra6m4_secondary_enc_xmodem/Debug/app_ra6m4_secondary_enc_xmodem.bin.signed

3.2.5 Debug the Applications

Choose to debug from primary application project `app_ra6m4_primary_enc_xmodem`.

Right click on project `app_ra6m4_primary_enc_xmodem` and select **Debug As** > **Debug Configurations**. Select **app_ra6m4_primary_enc_xmodem Debug_Flat** > **Startup** and confirm that the following configuration exists.

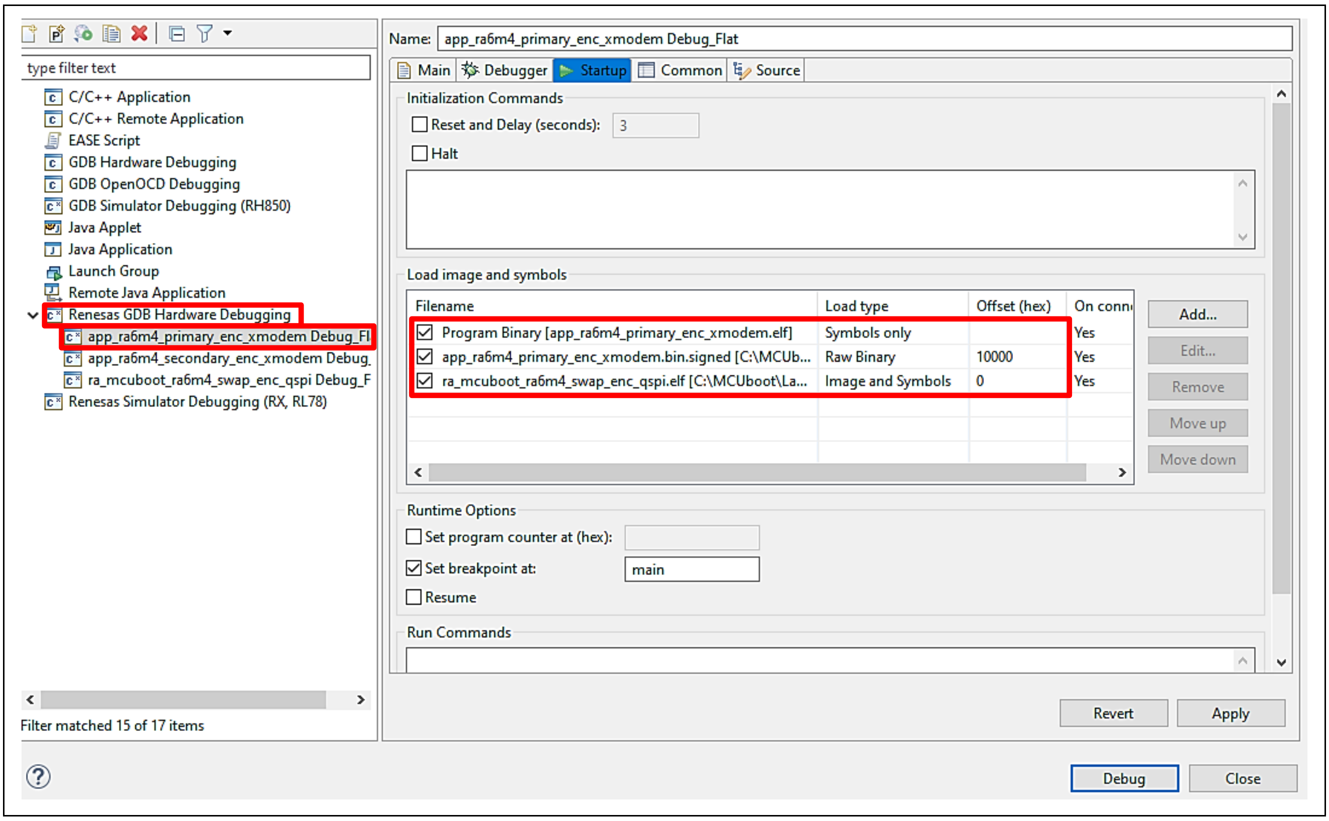

Figure 11. Debug Configurations

- Under the Startup configuration, verify the Load type of `app_ra6m4_primary_enc_xmodem.elf` is Symbols only rather than Image and Symbols.
- The `app_ra6m4_primary_enc_xmodem.bin` signed entry exists with Load type as Raw Binary and the Offset is set to 0x10000 since that is the beginning of the primary application.
- The `ra_mcuboot_ra6m4_swap_enc_qspi`.elf is added with Load type as Image and Symbols with an Offset of 0 since the bootloader starts from 0x0.

Click **Debug**, then **Resume** the execution twice by clicking . The primary application is then booted, and the three LEDs are blinking.

3.2.6 Downloading and Running the Secondary Application

Use the following steps to download and run the secondary application.

1. Launch Tera Term and selected the enumerated COM port "USB Serial Device". Your port number may be different from this. Click OK.

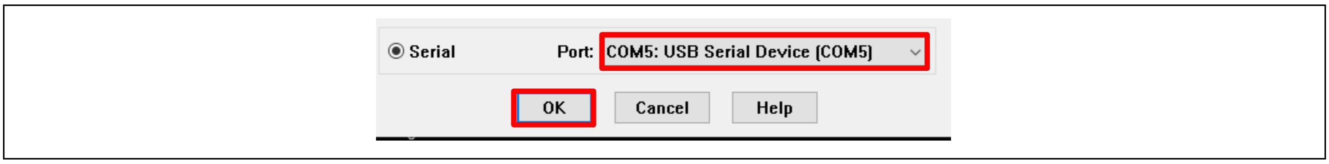

Figure 12. Launch Tera Term

2. Below message will be printed.

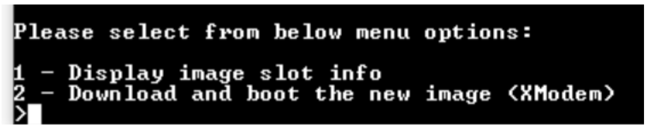

Figure 13. Menu item

3. View option 1 result. We can see Secondary image is empty.

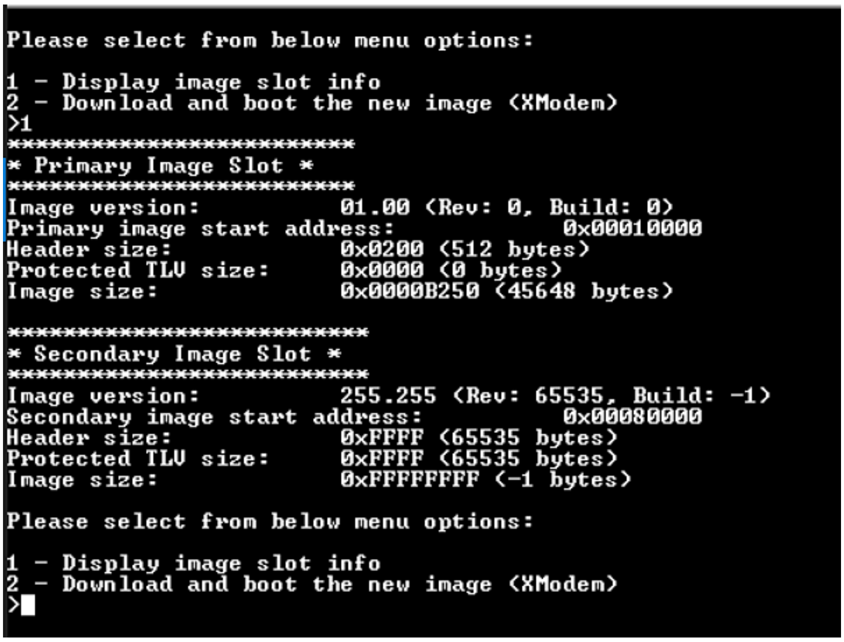

Figure 14. Primary and Secondary Slot Status

4. Now use the image downloader to load the new secondary application image. Choose option 2 to download the secondary image.

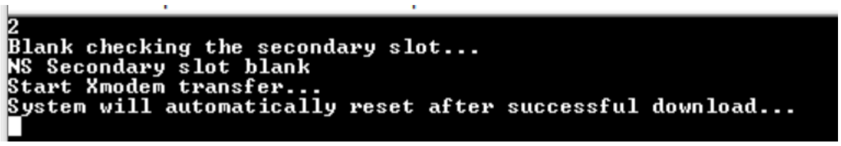

Figure 15. Initiate Secondary Image Download

5. Choose **File** > **Transfer** > **XMODEM** > **Send**

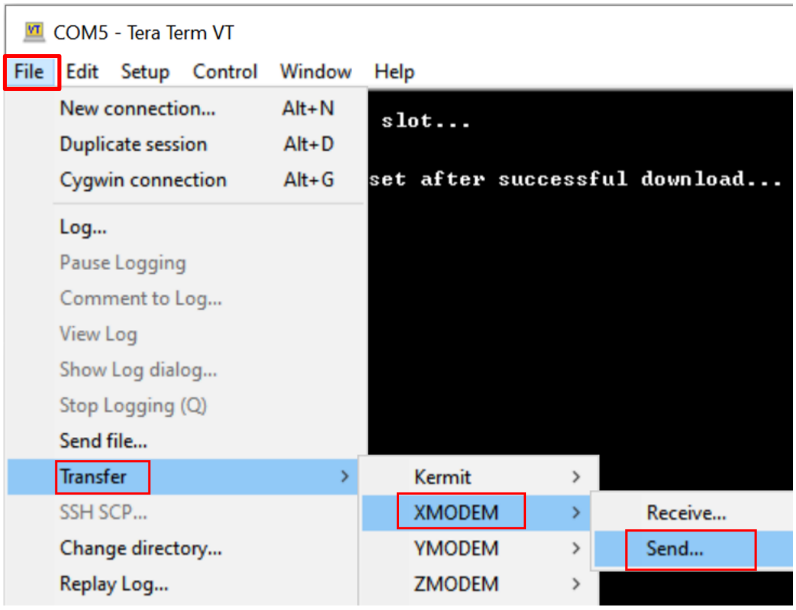

Figure 16. Choose to use XModem

6. Select the signed secondary image binary.

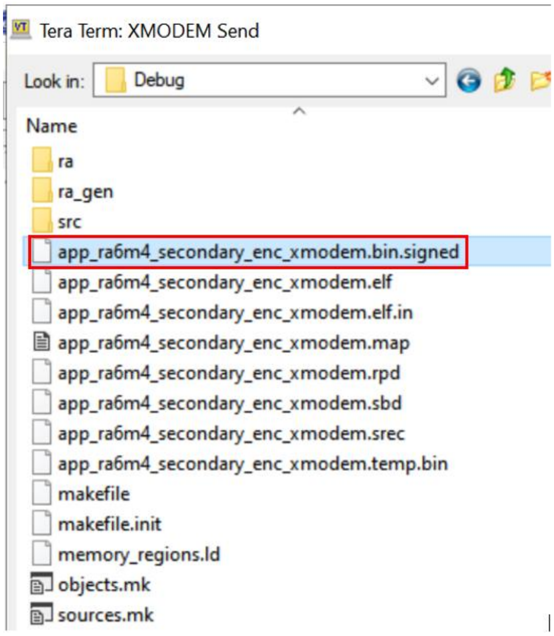

Figure 17. Select the Signed Secondary Image

7. It takes about 25 seconds to download the new image.

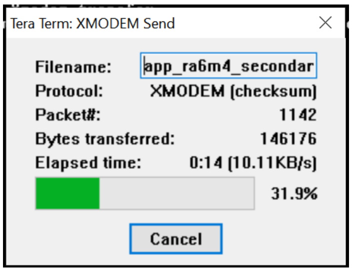

Figure 18. Download the Secondary Image using XModem

8. The primary application will reset the system once the entire secondary application is downloaded. The menu from the secondary application is printed. Wait about two seconds prior to the output of the new menu. The Blue LED should be blinking.

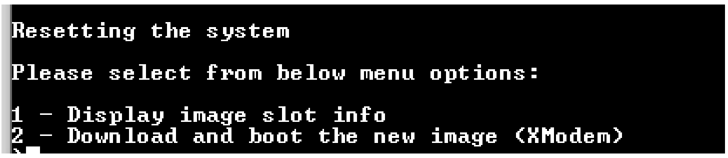

Figure 19. Secondary Image is booted

9. Reset the application from the debugger, the blue LED should still be blinking. There is no revert back to the original Primary application because the swap test mode is implemented with the secondary application.

4. Add Encryption to the Initial Example Project

In this section, we will add encryption to the application image. The bootloader is first updated and then the application projects are configured to use the new bootloader.

The system will go through the following stages. Note that when encryption is enabled, the bootloader image size increases to about 83 kB. With the code flash boundary at 32 kB, the bootloader image is allocated 96 kB.

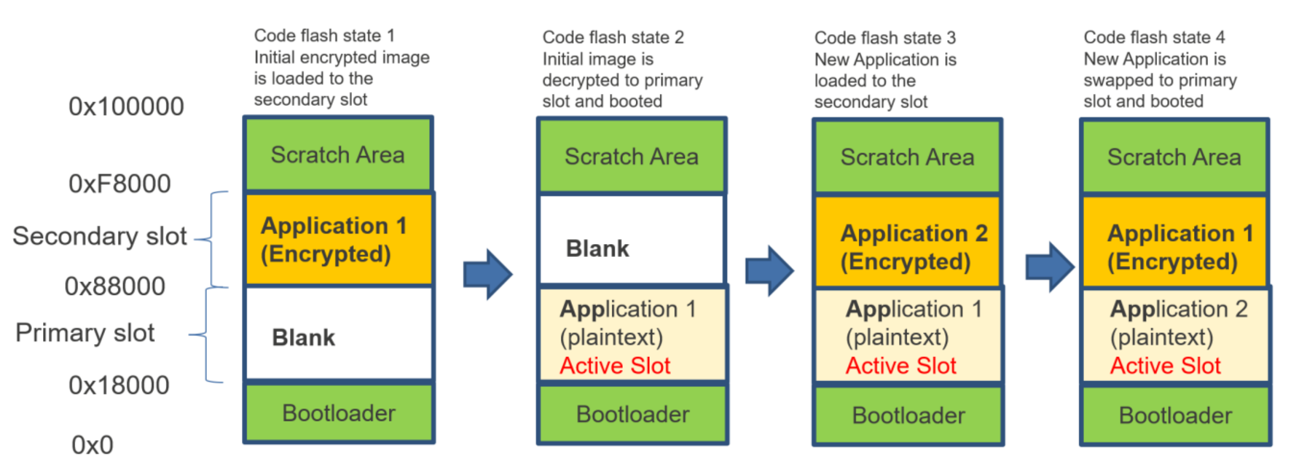

Figure 20. Booting Encrypted Image (Secondary Image Stored in Internal Flash)

Note that the initial application is downloaded to the secondary slot as encrypted rather than downloaded to the primary slot as plaintext image. This allows plaintext image being swapped to the secondary slot as plaintext.

4.1 Configure the Bootloader for Encryption Support

Stay in the same Workspace from the previous section and start to configure the bootloader using the following steps:

1. Double click and open the configuration.xml file from ra_mcuboot_ra6m4_swap_enc_qspi project.
2. Navigate to the **Stacks** tab, select **MCUboot > Settings > Property > Common > Signing and Encryption Options > Encryption Scheme > ECIES-P256.**

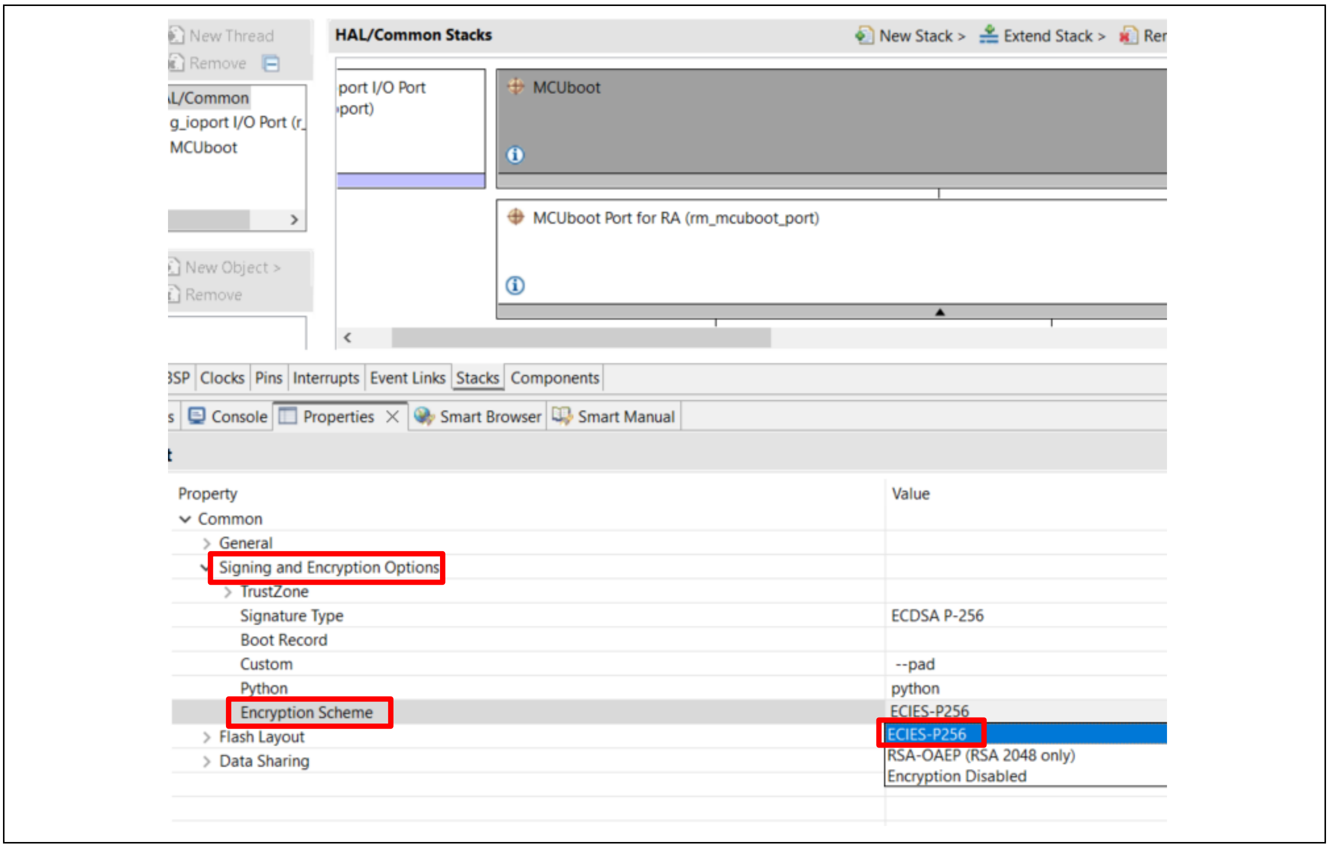

Figure 21. Choose ECIES-P256

3. Update the Bootloader Flash Area Size from 0x10000 to 0x18000.
   MCUboot > Settings > Property > Common > Flash Layout > Bootloader Flash Area Size (Bytes): 0x18000

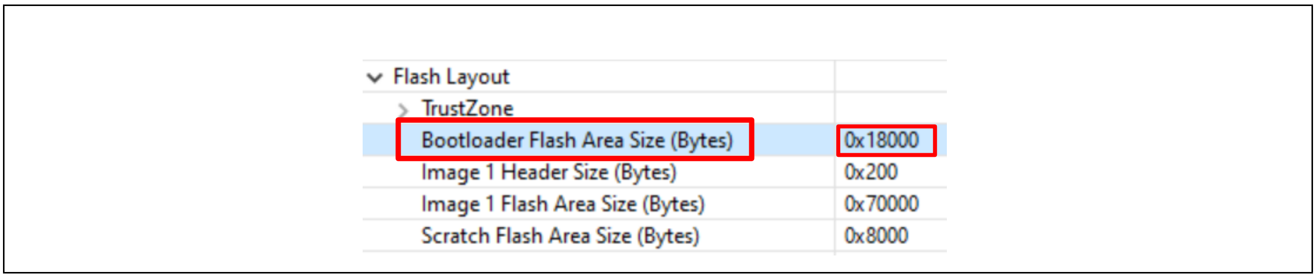

Figure 22. Update the Bootloader Flash Area Size

4. Navigate to the BSP tab and update the BSP heap size from 0x600 to 0x1000. When encryption is used, a minimum of 0x200 heap needs to be added. This increased heap usage came from the added AES algorithm usage.

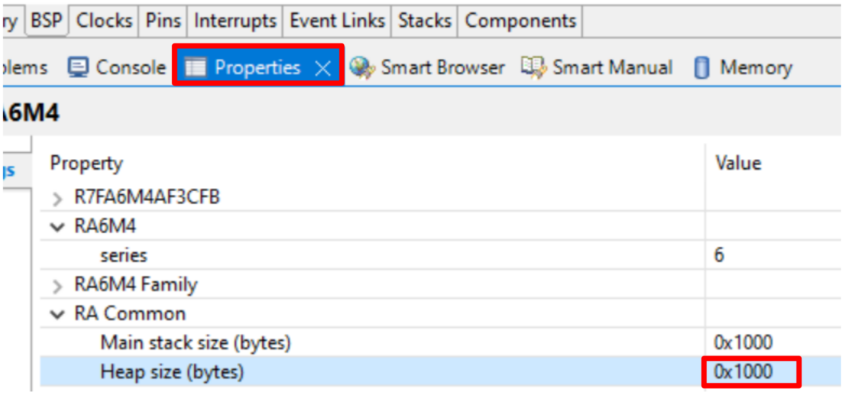

Figure 23. Update the Heap size to 0x1000

5. Right click on the bootloader project and select **Properties** (at the end of the menu tree).

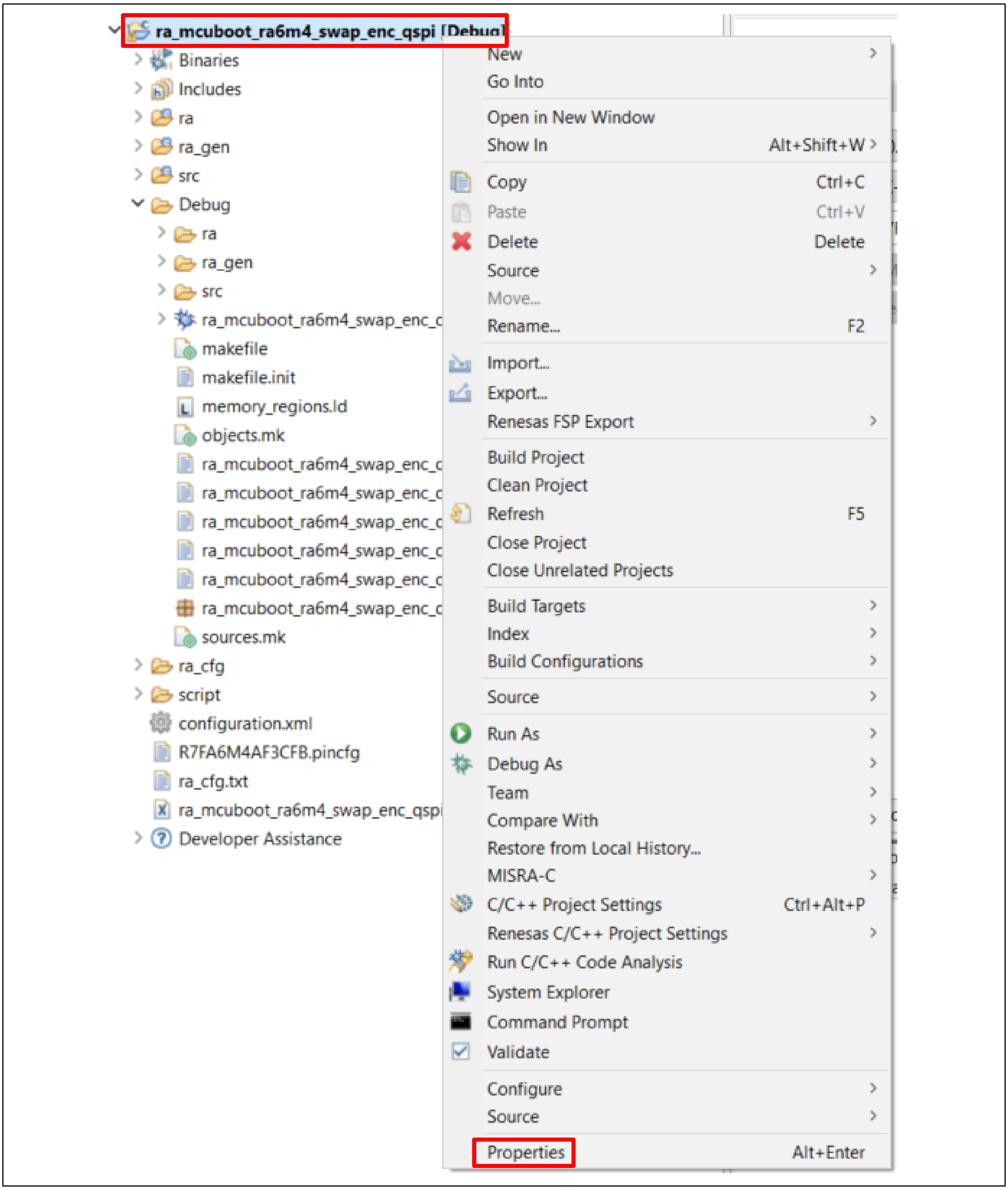

Figure 24. Open the Properties Window

6. Navigate to the **C/C++ Build > Settings > Tool Settings > GNU Arm Cross C Compiler** > **Preprocessor**.

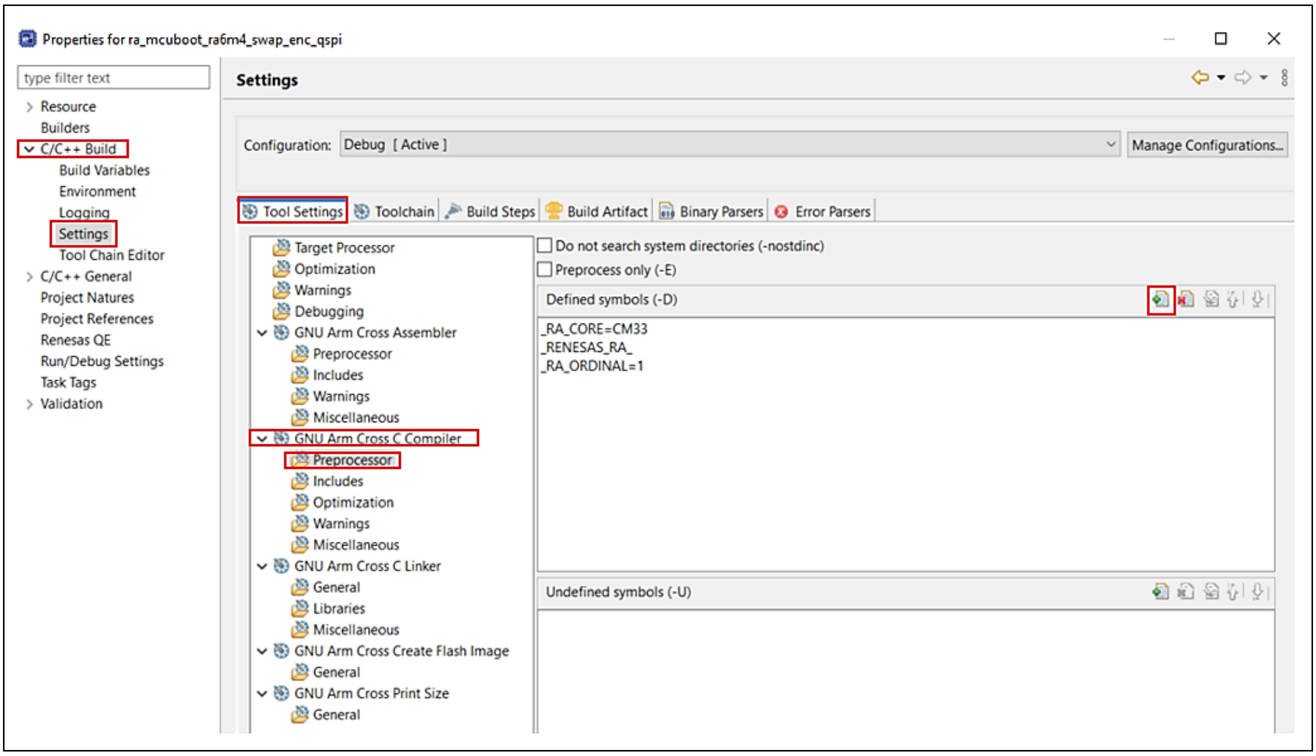

Figure 25. Add Preprocessor setting

7. Click the green '+' sign and add `MCUBOOT_BOOTSTRAP`. This preprocessor enables booting the first encrypted image from the secondary slot when having an empty image from the primary slot. Click **OK**.

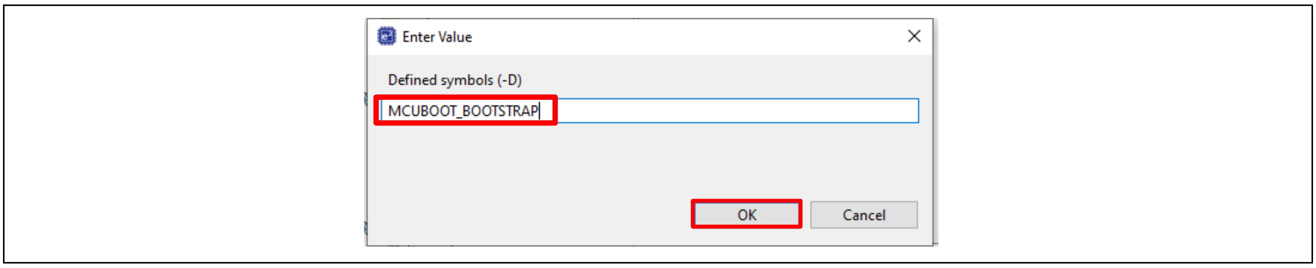

Figure 26. Add Preprocessor MCUBOOT_BOOTSTRAP

8. Click **Apply and Close**.

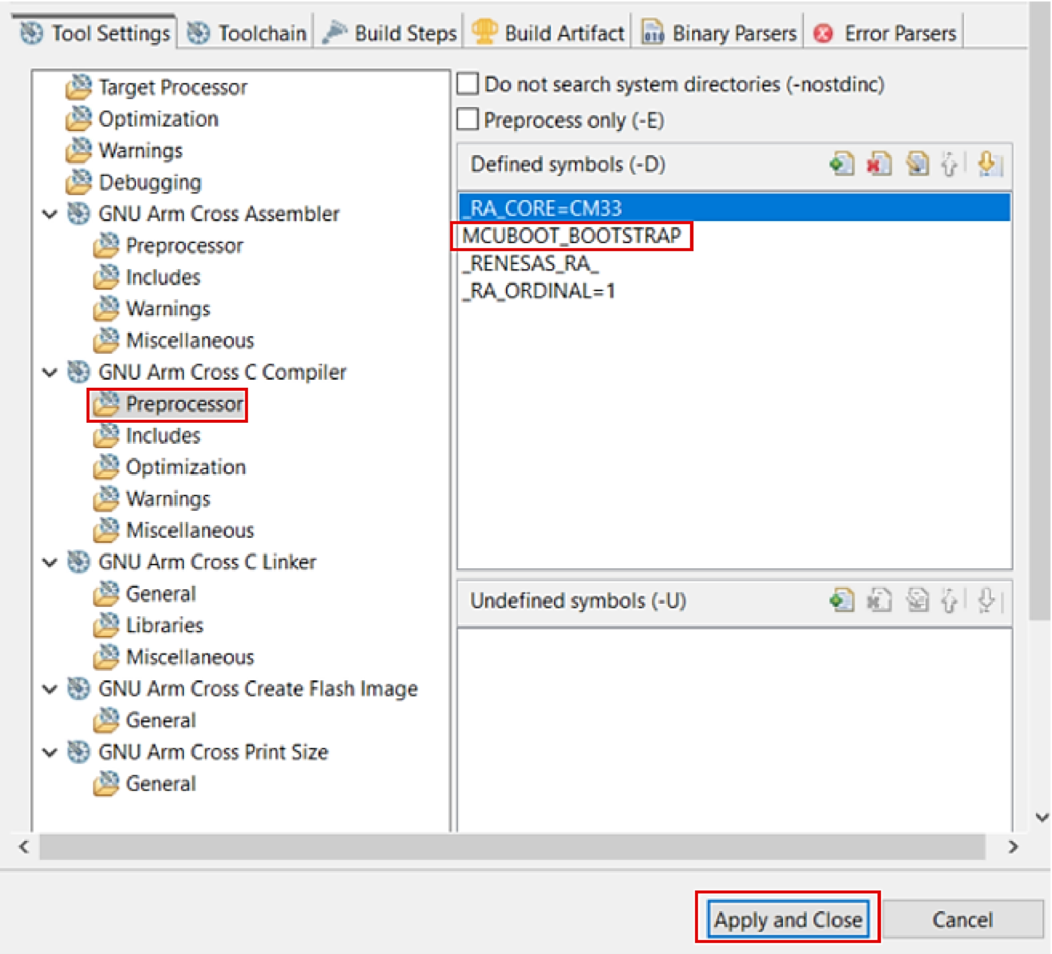

Figure 27. Add Preprocessor MCUBOOT_BOOTSTRAP

9. Check **Remember my decision** and click **Rebuild Index** if below window pops up.

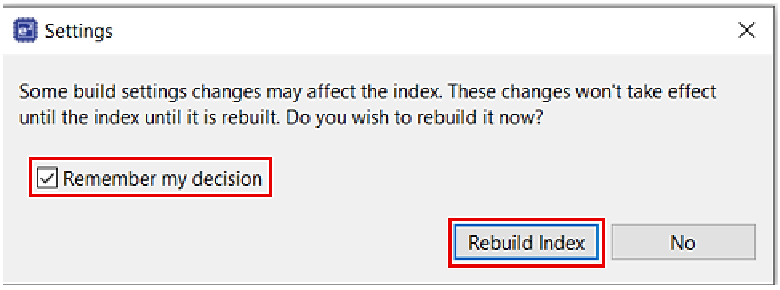

Figure 28. Add Preprocessor MCUBOOT_BOOTSTRAP

10. Click **Generate Project Contents** and then compile the bootloader project. Check **Always save and generate without asking** if this window pops up. Click **Proceed** and compile the updated bootloader.

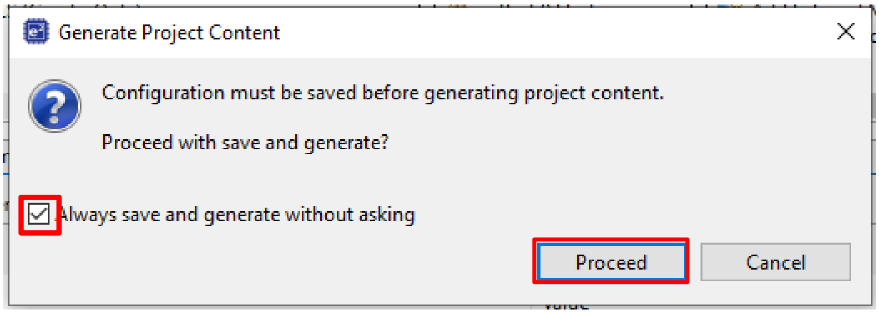

Figure 29. Configure settings for Generate Project Content

4.2 Configure the Application Project for Encryption Support

Follow the steps below to configure the application project to support image encryption.

1. Right click on the Primary Application app_ra6m4_primary_enc_xmodem, select **Properties** > **C/C++ Build** > **Environment**.

   Click **Add** and define the New variable **Name** as:

   MCUBOOT_IMAGE_ENC_KEY

   Define the **Value** as:

   `${workspace_loc:ra_mcuboot_ra6m4_swap_enc_qspi}/ra/mcu-tools/MCUboot/enc-ec256-pub.pem`

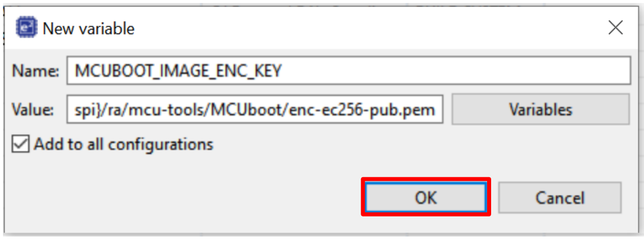

Figure 30. Configure the ECDSA Public Key to be Used in Image Encryption

2. Review the Build Variable Settings and click **Apply and Close**.

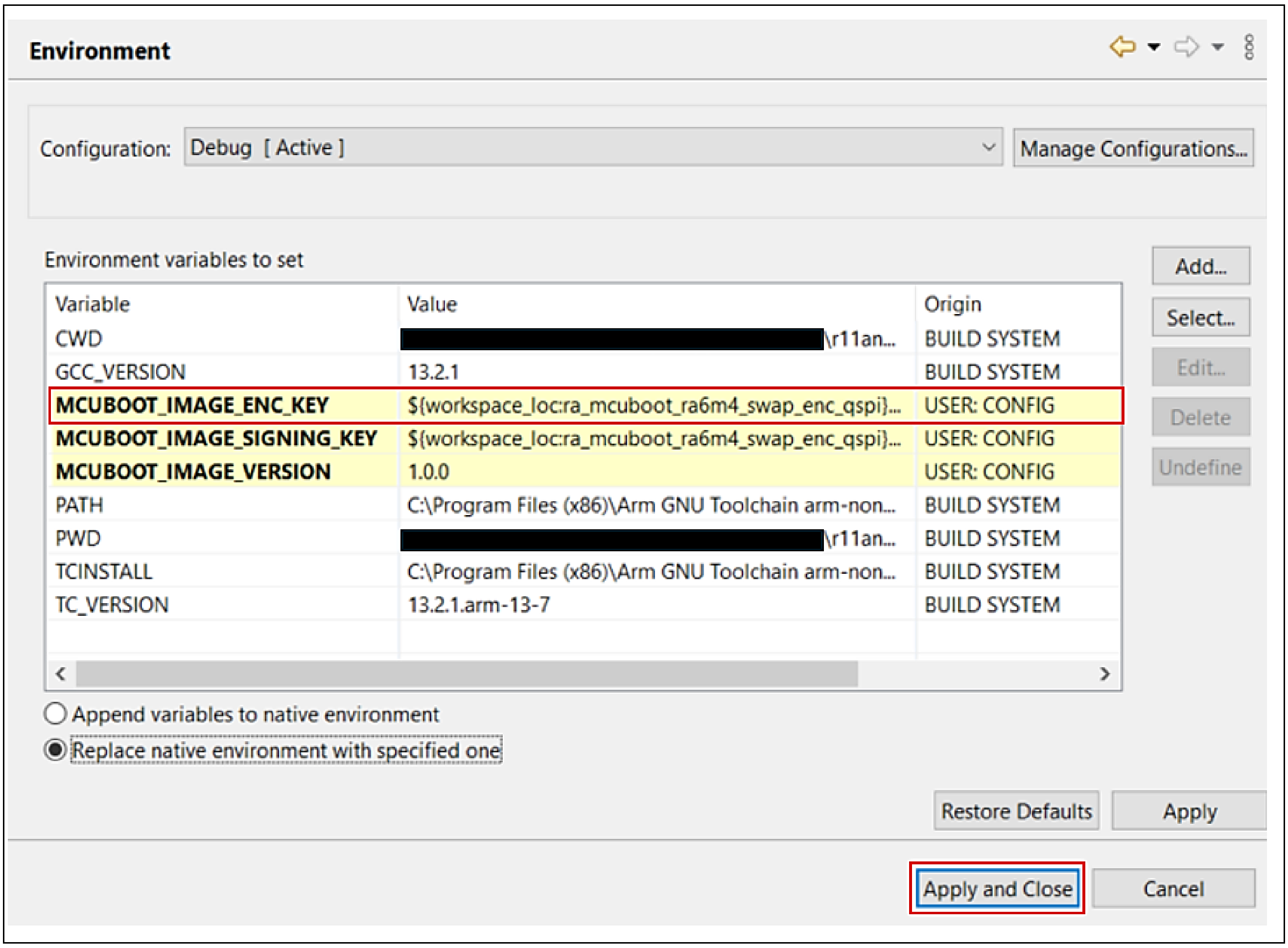

Figure 31. Review the Application Project Encryption Support Setting

3. Update the `\app_ra6m4_primary_enc_xmodem\src\header.h` file. This update takes care of the application image location change due to the change in the bootloader size.

   Update below address configuration from:

```
#define PRIMARY_IMAGE_START_ADDRESS      0x00010000
#define PRIMARY_IMAGE_END_ADDRESS        0x0007FFFF
#define SECONDARY_IMAGE_START_ADDRESS    0x00080000
#define SECONDARY_IMAGE_END_ADDRESS      0x000EFFFF
To:
#define PRIMARY_IMAGE_START_ADDRESS      0x00018000
#define PRIMARY_IMAGE_END_ADDRESS        0x00087FFF
#define SECONDARY_IMAGE_START_ADDRESS    0x00088000
#define SECONDARY_IMAGE_END_ADDRESS      0x000F7FFF
```

4. Double click `configuration.xml` to open the smart configurator, click **Generate Project Content** and compile the Primary application.

   Ensure `\Debug\app_ra6m4_primary_enc_xmodem.bin.signed.encrypted` is generated.

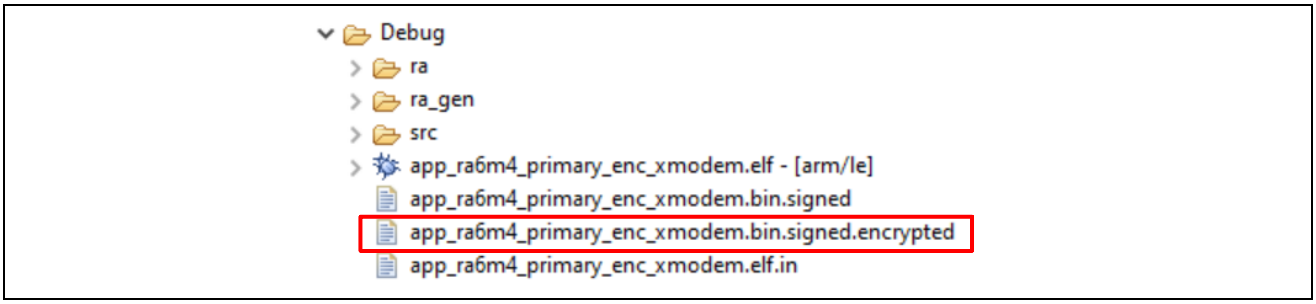

Figure 32. Ensure the Encrypted Binary is Generated

5. Repeat previous **steps 1, 2, 3** and **4** in this section for the secondary project.
6. Follow **step 2, 3** in **section 3.2.1** to Erase the chip.
7. Update the Debug configuration.
   Right click on the Primary application **app_ra6m4_primary_enc_xmodem** > **Debug As** > **Debug Configurations**, make sure the Primary application is selected and navigate to the Startup window. Update the Startup configuration Load image and symbols area as shown below.
   - Remove the entry of `app_ra6m4_primary_enc_xmodem.bin.signed`.
   - Click **Add** > **Workspace** and browse to the file `app_ra6m4_primary_enc_xmodem.bin.signed.encrypted`.

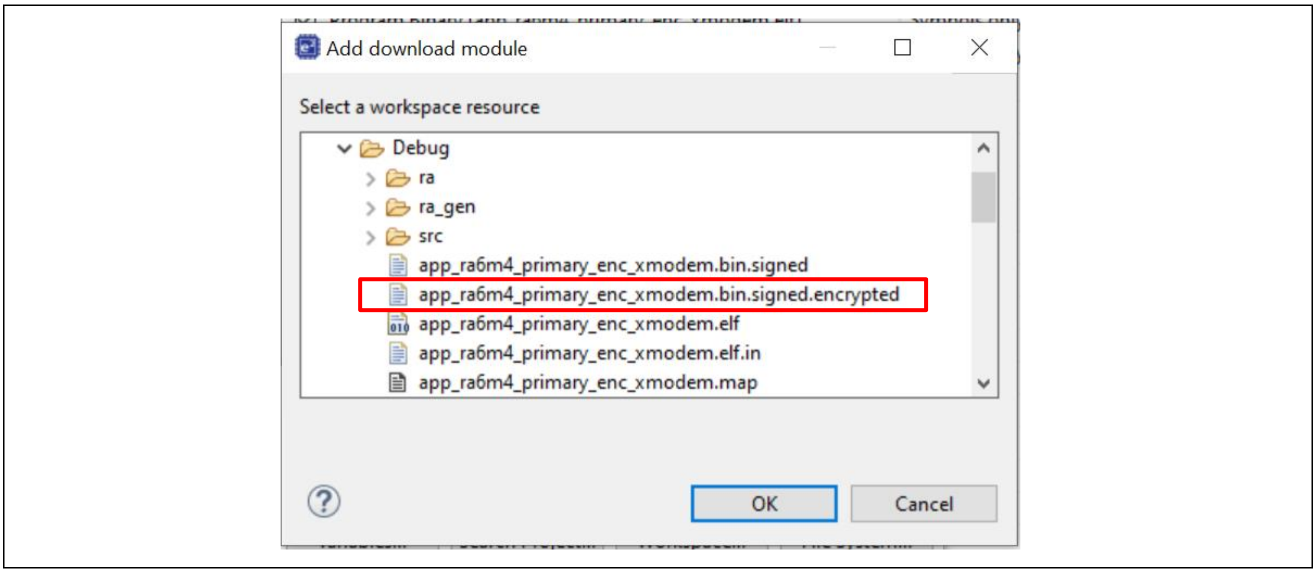

Figure 33. Update the Debug Configuration

Click **OK**.

8. Update the Primary Image download address and Load type.
   Change the Load type to of the `app_ra6m4_primary_enc_xmodem.bin.signed.encrypted` to **Raw Binary**. Update the **Offset** to the **secondary slot address** based on the new bootloader size.

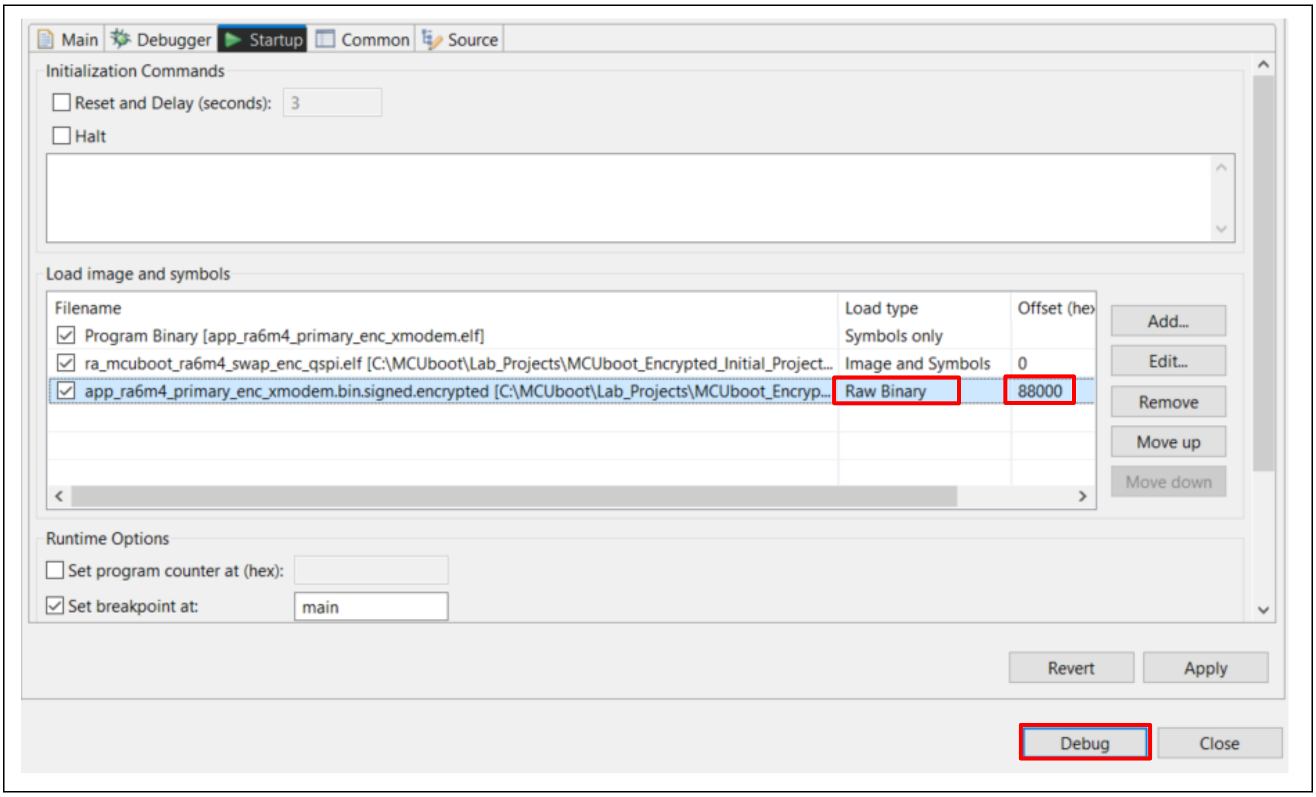

Figure 34. Update the Primary Application Load Address

9. Click **Debug** and resume  the execution twice; the Primary application will be booted, and three LEDs should be blinking.
10. Follow **steps 3** to **8** in section **3.2.6** to use the X Modem downloader to download the secondary application.
11. Make sure to select the encrypted secondary image.

    When downloading the seconday image, make sure to select the encrypted image.

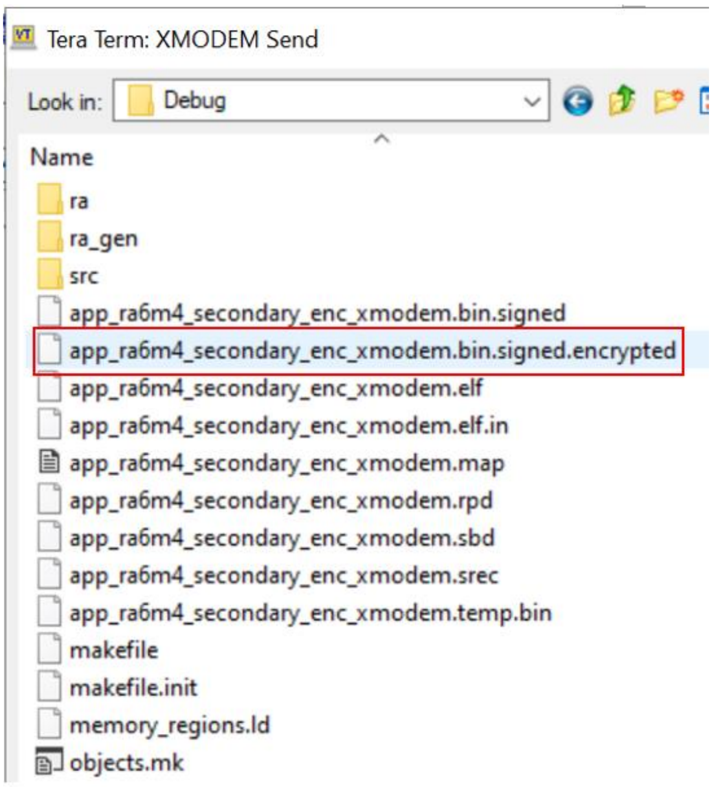

Figure 35. Select the Encrypted Secondary Image

12. After the secondary image is downloaded, it will be booted after the bootloader verified the image. The blue LED should be blinking.

5. Use QSPI as Secondary Storage Area

In this section, we will switch the secondary image storage area from internal flash to QSPI. User can also benefit from this section in terms of learning the key steps in the image downloader design when using XModem. Below is the memory layout of the resulting system.

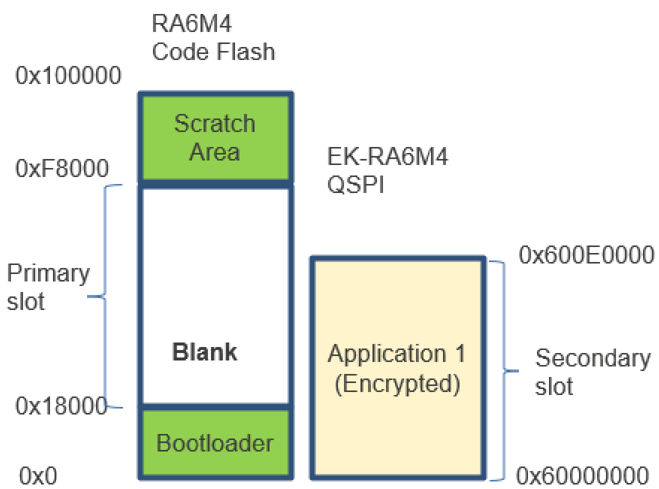

Figure 36. Using QSPI for Secondary Image Storage

Note that the primary and secondary application image sizes are increased to benefit from the usage of the QSPI.

There are four stages the system will go through by following the steps layout described in this section, which is generally similar to the case of using internal flash.

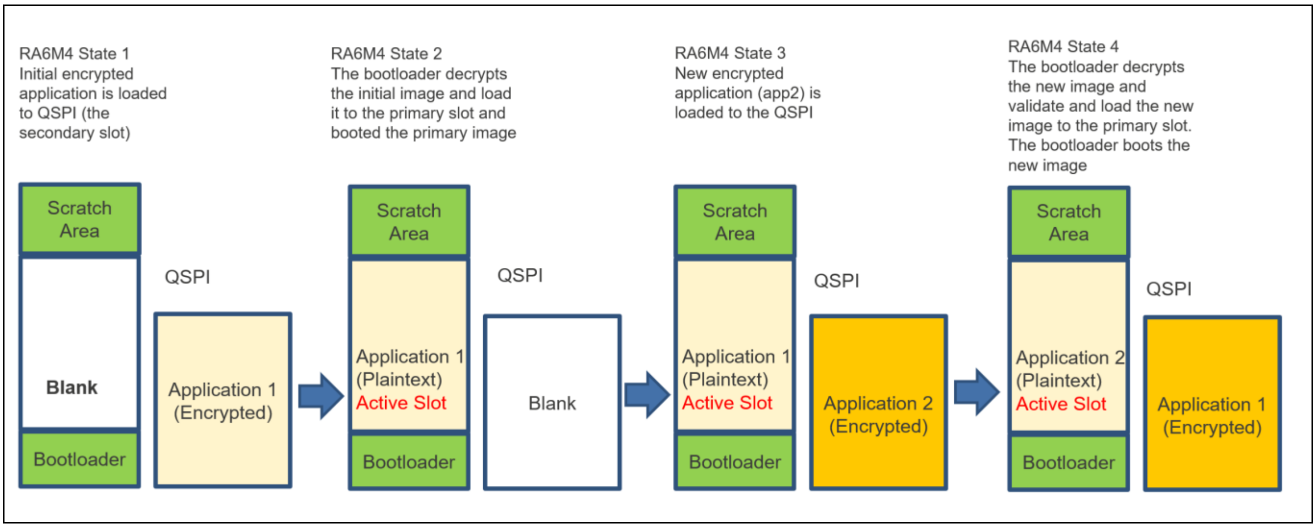

Figure 37. Functional Stages

5.1 Configure the Bootloader to Use QSPI for Secondary Application Storage

Use the following steps to update the secondary storage area to QSPI.

1. Open the `configuration.xml` file from the bootloader project `ra_mcuboot_ra6m4_swap_enc_qspi`.
2. Click on **MCUboot > MCUboot Port for RA (rm_mcuboot_port) > Add External Memory Implementation (Optional)**, select **New > MCUboot External Memory (QSPI)** to add the QSPI stack:

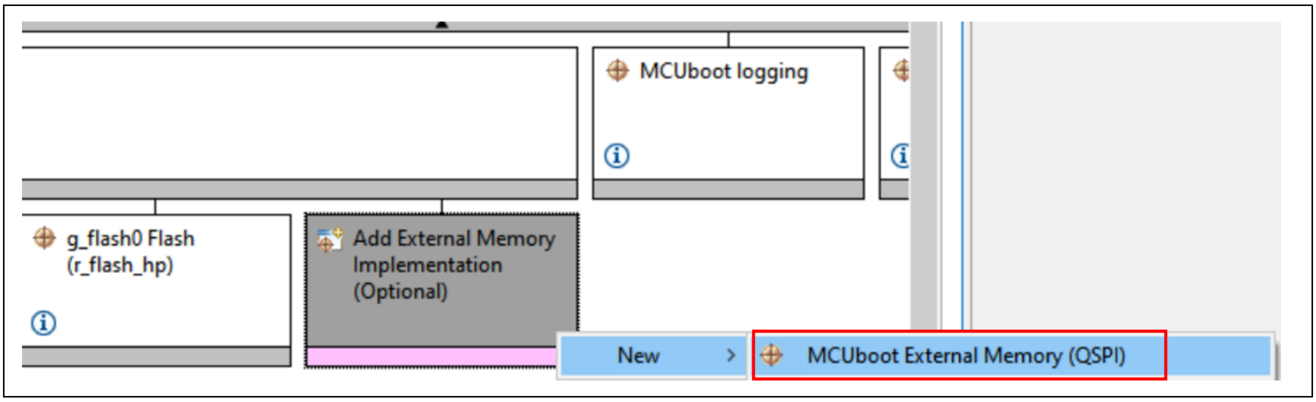

Figure 38. Choose QSPI from the Smart Configurator Stack Tab

3. Navigate to the **Pins** tab **Peripherals** group and select the **Storage:QSPI > QSPI0**. First select **_B only** for the **Pin Group Selection**, then select **Quad** as the **Operation Mode**. The correct **Input/Output** pins will be automatically selected. We need to do this because the bootloader uses a minimal pin configuration rather than the pin configuration for EK-RA6M4.

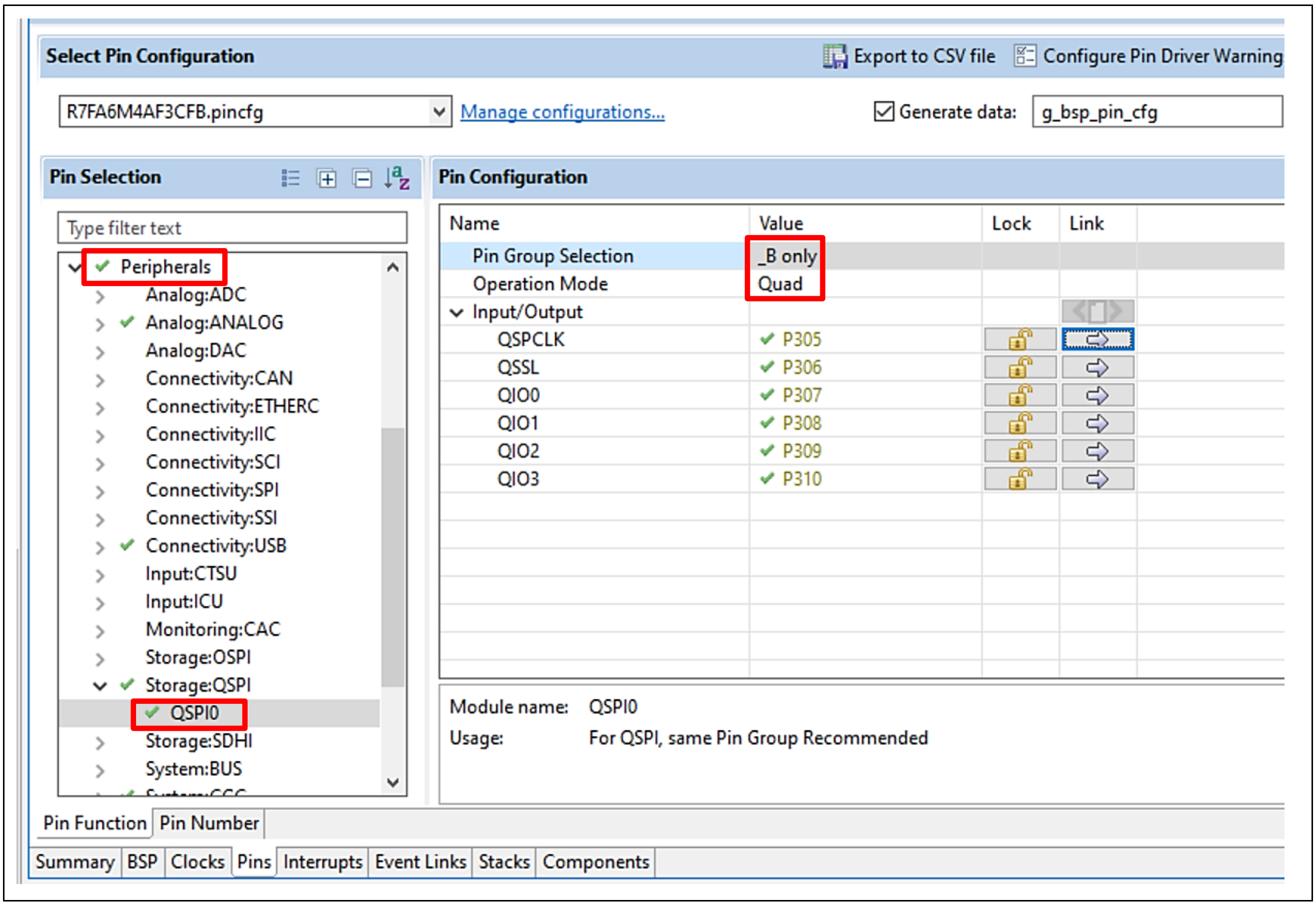

Figure 39. Configure the QSPI Pin and Operation Mode

4. Navigate to the **Stacks** tab, highlight the QSPI stack and update the Bus Timing Minimum QSSL Deselect Cycles to 8 QSPICLK.

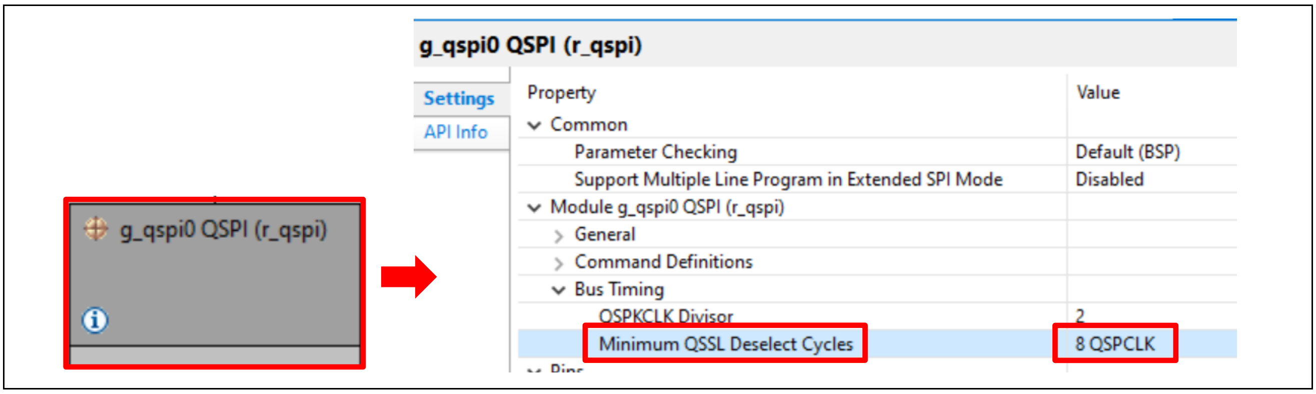

Figure 40. Update the QSPI Bus Timing Minimum QSSL Deselect Property

5. Highlight the MCUboot stack and change the Image 1 Flash Area Size Configuration using the value indicated below. When using QSPI, a much larger image is supported.

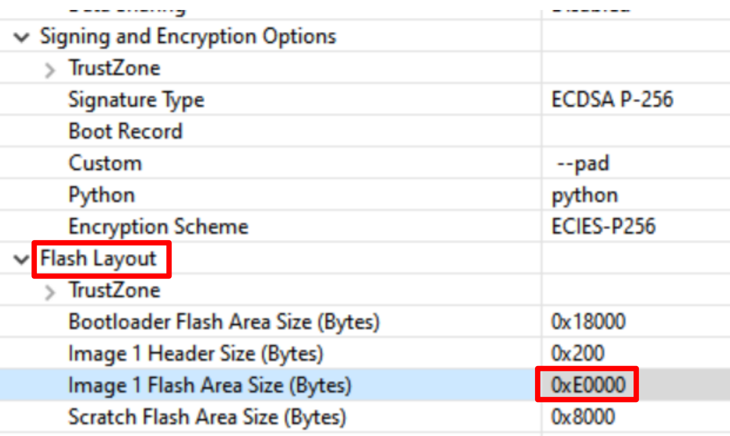

Figure 41. Configure the QSPI Pin and Operation Mode

6. Inside the bootloader project, add these variable definitions to the beginning of `hal_entry.c` file after the `R_BSP_WarmStart` function call:

```
FSP_CPP_HEADER
void R_BSP_WarmStart(bsp_warm_start_event_t event);
FSP_CPP_FOOTER
/* SREG pay-load size */
#define SREG_SIZE                        (0x03)
/* Status register pay-load */
#define STATUS_REG_PAYLOAD               {0x01,0x40,0x00}
uint8_t   data_sreg[SREG_SIZE]                    = STATUS_REG_PAYLOAD;
```

Figure 42. Add QSPI Variable Definition

7. Stay with `hal_entry.c`, add below code to the beginning of hal_entry() function and before the line `mcuboot_quick_setup();`.

```
fsp_err_t err = FSP_SUCCESS;
R_QSPI_Open(&g_qspi0_ctrl, &g_qspi0_cfg);
/* write enable for further operations */
err = R_QSPI_DirectWrite(&g_qspi0_ctrl, &(g_qspi0_cfg.write_enable_command), 1, false);
if(FSP_SUCCESS == err)
{
    err = R_QSPI_DirectWrite(&g_qspi0_ctrl, data_sreg, SREG_SIZE, false);
    if(FSP_SUCCESS != err)
    {
        while(1);
    }
}
```

Figure 43. Set up the QSPI

8. Within the bootloader smart configurator, click **Generate Project Content** and compile the bootloader project.

5.2 Update the Primary Application Project to Support QSPI

1. Within the primary application smart configurator, click **Downloader Thread** > **New Stack** > **Storage** > **QSPI**, add the QSPI stack.

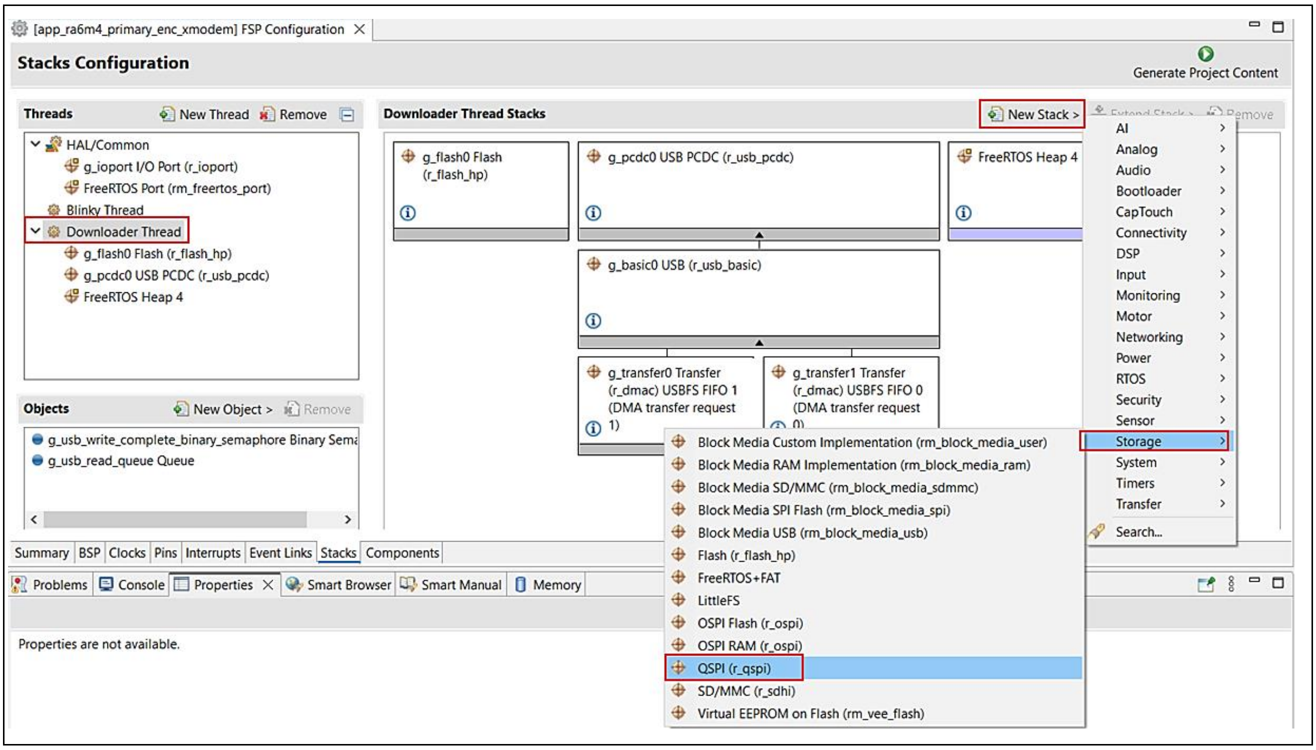

Figure 44. Add the QSPI Stack

2. Highlight the QSPI stack and update the **Bus Timing, Minimum QSSL Deselect Cycles** to **8 QSPCLK**.

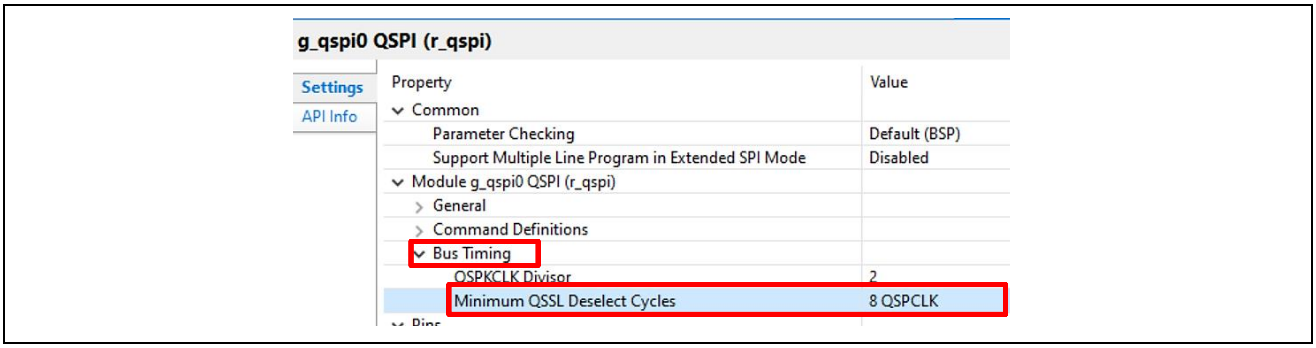

Figure 45. Add the QSPI Stack

3. Copy below files from the `qspi_souce.zip` to overwrite the existing files in the primary application project. The updates related with supporting QSPI usage are explained in the updates performed column.

Table 3. Source File Updates Moving from Internal Flash to QSPI for Secondary Image Storage

Files to overwrite	Updates Performed
`downloader_thread_entry.c`	Remove code flash initialization and add QSPI initialization
`menu.c`	Prior to image download over USB PCDC, the flash area needs to be erased. The update performed is to switch from erasing the code flash to erasing the QSPI.

`xmodem.c`	`xmodem.c` handles downloading the new image and writing to the secondary application storage area. The updates to this file are to change from writing to internal flash to writing to QSPI.
`header.h`	The `header.h` file has definitions on the start and end location of the primary and secondary slot. The update to this file is to change the secondary application starting address as well as the size of the primary and secondary application based on the new bootloader image size configuration and the QSPI address.

4. Copy the highlighted files `qspi_source.zip` to the `\src` folder for the primary project. These are files supporting QSPI operations.

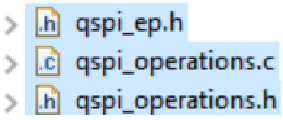

Figure 46. QSPI related Source Files

5. Save all files. Navigate to the smart configurator, click **Generate Project Content** and compile the Primary application.
6. Perform the same update steps from **step 1** to **5** for the secondary application project.
7. Follow **step 2, 3** in **section 3.2.1** to Erase the chip.
8. Update the Debug Configuration of the primary application. Right click on `app_ra6m4_primary_enc_xmodem`, select **Debug As** > **Debug Configurations**. Navigate to the Startup window and update the primary image download Offset to the address of the secondary slot 0x60000000.

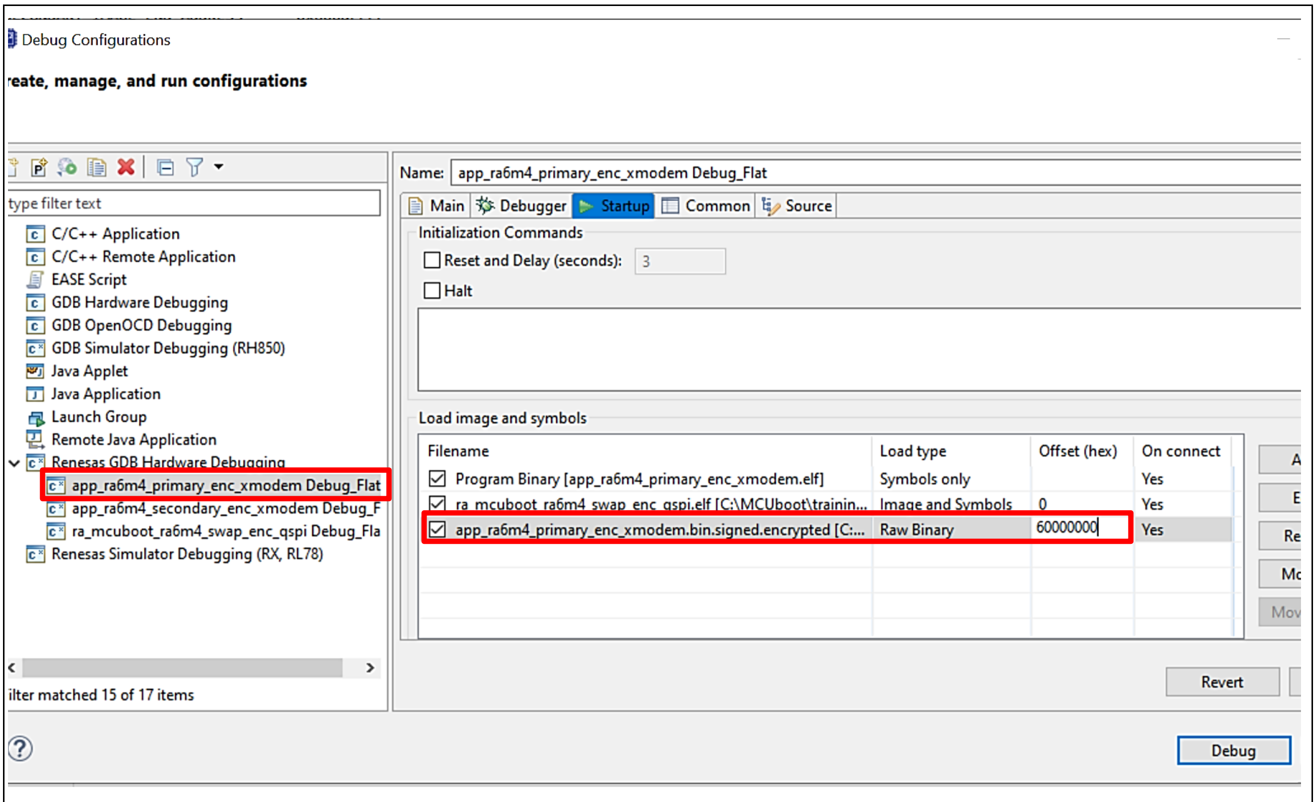

Figure 47. Configure the Debug Configuration

9. Click **Debug** and resume the execution twice to boot the primary application. The three LEDs should be blinking.
10. Follow section **3.2.6** to download and exercise the secondary application.

Note that a solution to this section is provided with this application project as `MCUboot_Encryption_QSPI_Solution.zip` for user's reference.

6. Using Custom Signing Key and Encryption Key

In this section, you will generate two sets of ECDSA SECP256R1 keys using the `imgtool.py` tool included with MCUboot. One set will be used for image signing support, the other pair will be used for image encryption support.

User can also use other key generation method to generate the keys, for example OpenSSL. OpenSSL encodes its keys in SEC1 format, while MCUboot uses PKCS#8. So, if customer uses OpenSSL, a conversion needs to take place. The command used for this conversion is inserted in line in the lab steps for your reference.

The stack MCUboot Example Keys stack generates the example keys used in the image signing/verifying and image encryption/decryption process. The custom keys generated in this section replace these example keys.

These are the two example key structures in the bootloader project `\ra_mcuboot_ra6m4_swap_enc_qspi\ra\mcu-tools\MCUboot\sim\mcuboot-sys\csupport\keys.c` file.

The root_pub_der array is the public key for image verification.

```
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 48. Public Key used for Image Verification

The `enc_key` array is the private key used in the image decryption process.

```
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 49. Private Key used for Image Decryption

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. The matching public key for the private key enc_key is `enc-ec256-pub.pem`. For custom encryption support, 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.

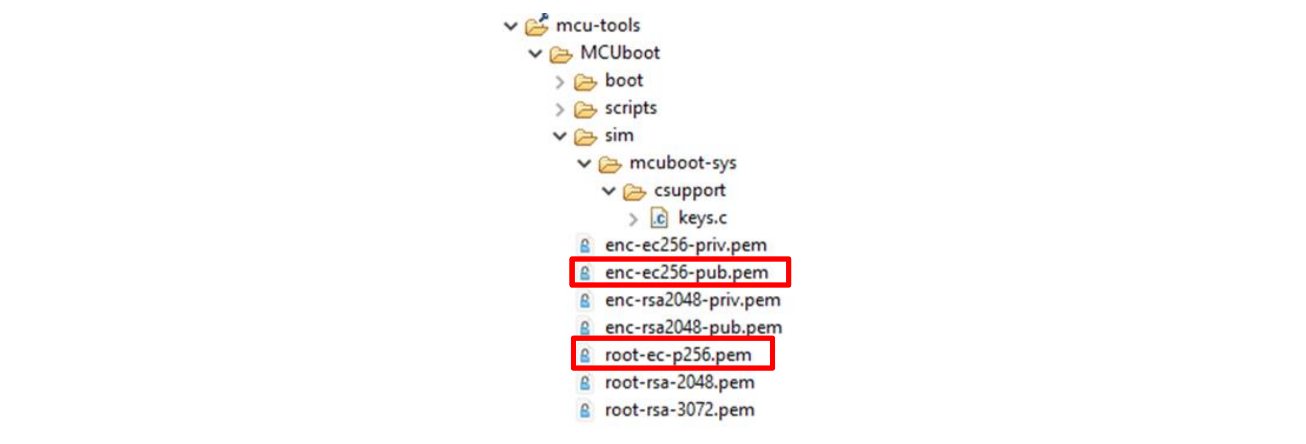

Figure 50. Image Signing Private Key and ECDSA SECP256R1 Public Key used in Image Encryption Process

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.

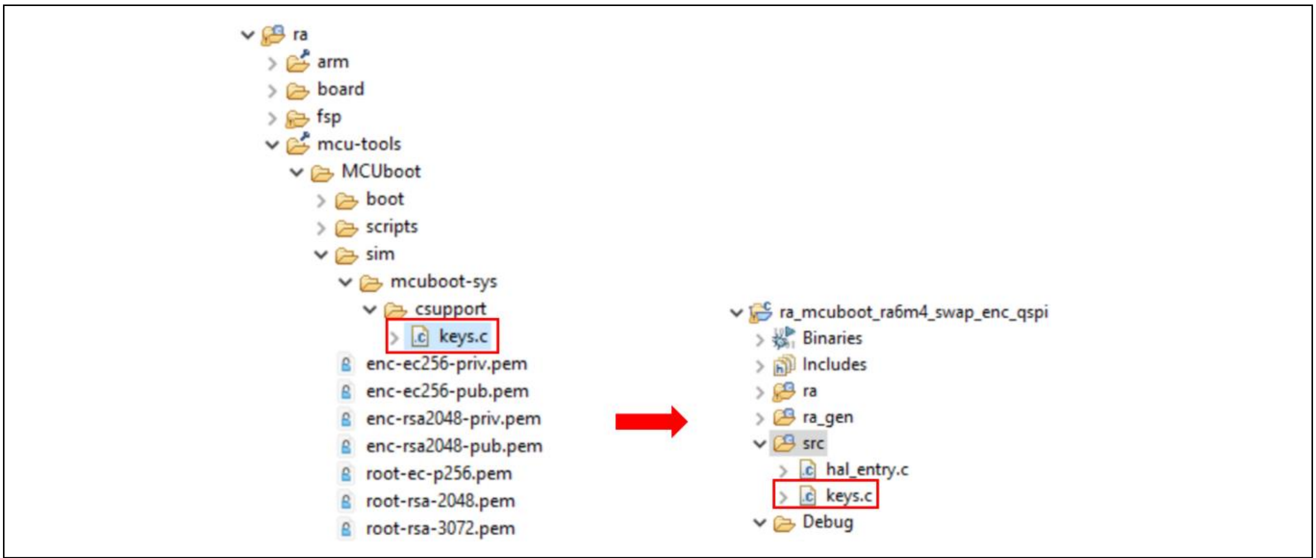

Figure 51. Copy the Example keys.c

2. Open the configurator for `ra_mcuboot_ra6m4_swap_enc_qspi`, right click on `MCUboot Example Keys` and select **Delete**.

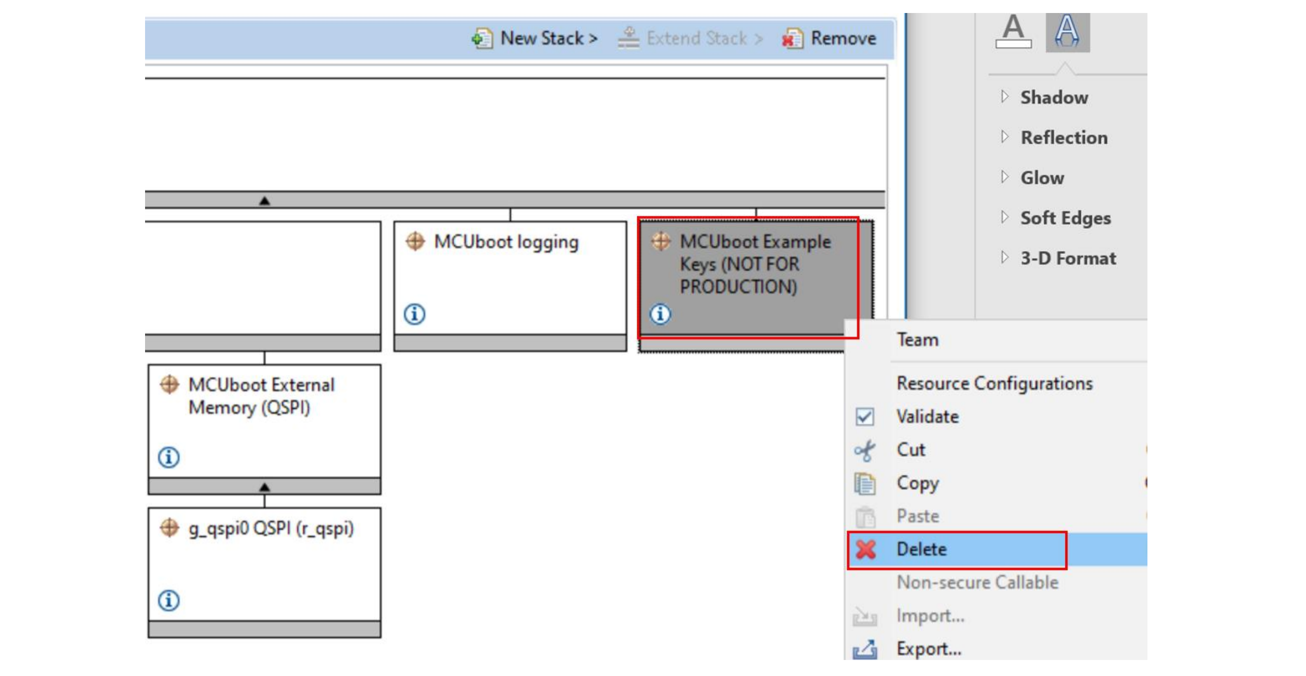

Figure 52. Delete the MCUboot Example Keys Stack

3. Extend `ra_mcuboot_ra6m4_swap_enc_qspi`, right click on folder `\scripts`. Select **Command Prompt** from this folder.

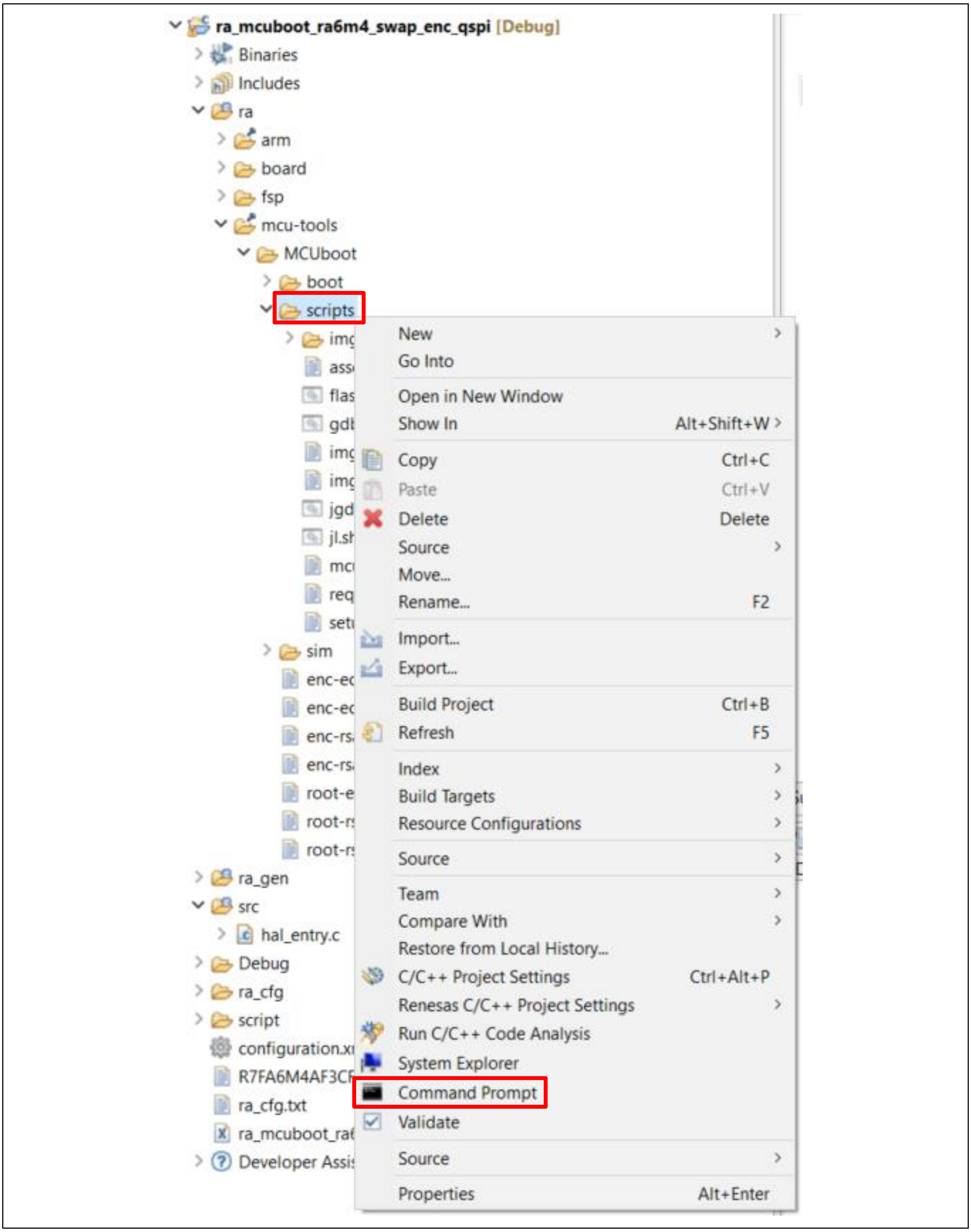

Figure 53. Start Command Prompt under the \MCUboot\scripts Folder

4. Under the command window, execture 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_ra6m4_swap_enc_qspi\src`
6. Extract the public key from ecc_sign_private.pem to use in the bootloader project.

Execute command:

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

Figure 54. Generate ECDSA Public Key

7. Copy the generated content of `ecdsa_pub_key` from **Figure 54** to array `root_pub_der` in `\src\keys.c`. Replace the original `root_pub_der` content.
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_ra6m4_swap_enc_qspi\src`.
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.

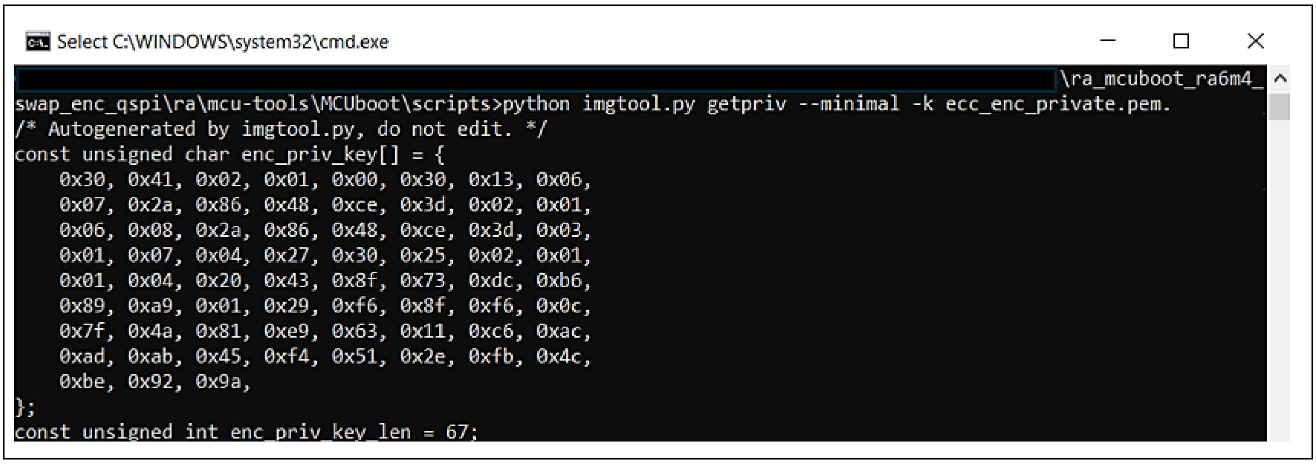

Figure 55. Generate the Private Key used for Image Encryption

11. Copy the content of `enc_priv_key` array generated in **Figure 55** to the array `enc_key` in `\src\keys.c`. Replace the orginal `enc_key` array content.
12. User need to download OpenSSL tool at https://sourceforge.net/projects/openssl-for-windows/files/OpenSSL-1.1.1h_win32.zip/download. Then, unzip `OpenSSL-1.1.1h_win32.zip`. Open another command line window under folder `\OpenSSL-1.1.1h_win32`.
13. Copy `ecc_enc_private.pem` to folder `\OpenSSL-1.1.1h_win32`.

14. We will derive the encryption public key in pem format using the private key using OpenSSL. Execute command:

```
openssl ec -in ecc_enc_private.pem -pubout -out ecc_enc_public.pem
```

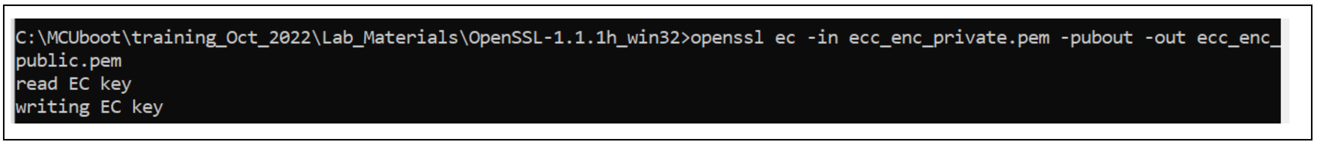

Figure 56. Generate the Public using the Private Key

15. Copy the generated `ecc_enc_public.pem` to the folder `\ra_mcuboot_ra6m4_swap_enc_qspi\src`.
16. Click **Generate Project Content** and compile the bootloader project.
17. Update the signing key configuration of the primary application project

Right click on the Primary Application app_ra6m4_primary_enc_xmodem, select **Properties** > **C/C++ Build** > **Environment**.

Choose "**MCUBOOT_IMAGE_SIGNING_KEY**" Variable, click **Edit** and define the **Value** as:

${workspace_loc:ra_mcuboot_ra6m4_swap_enc_qspi}/src/ecc_sign_private.pem

Click **OK**.

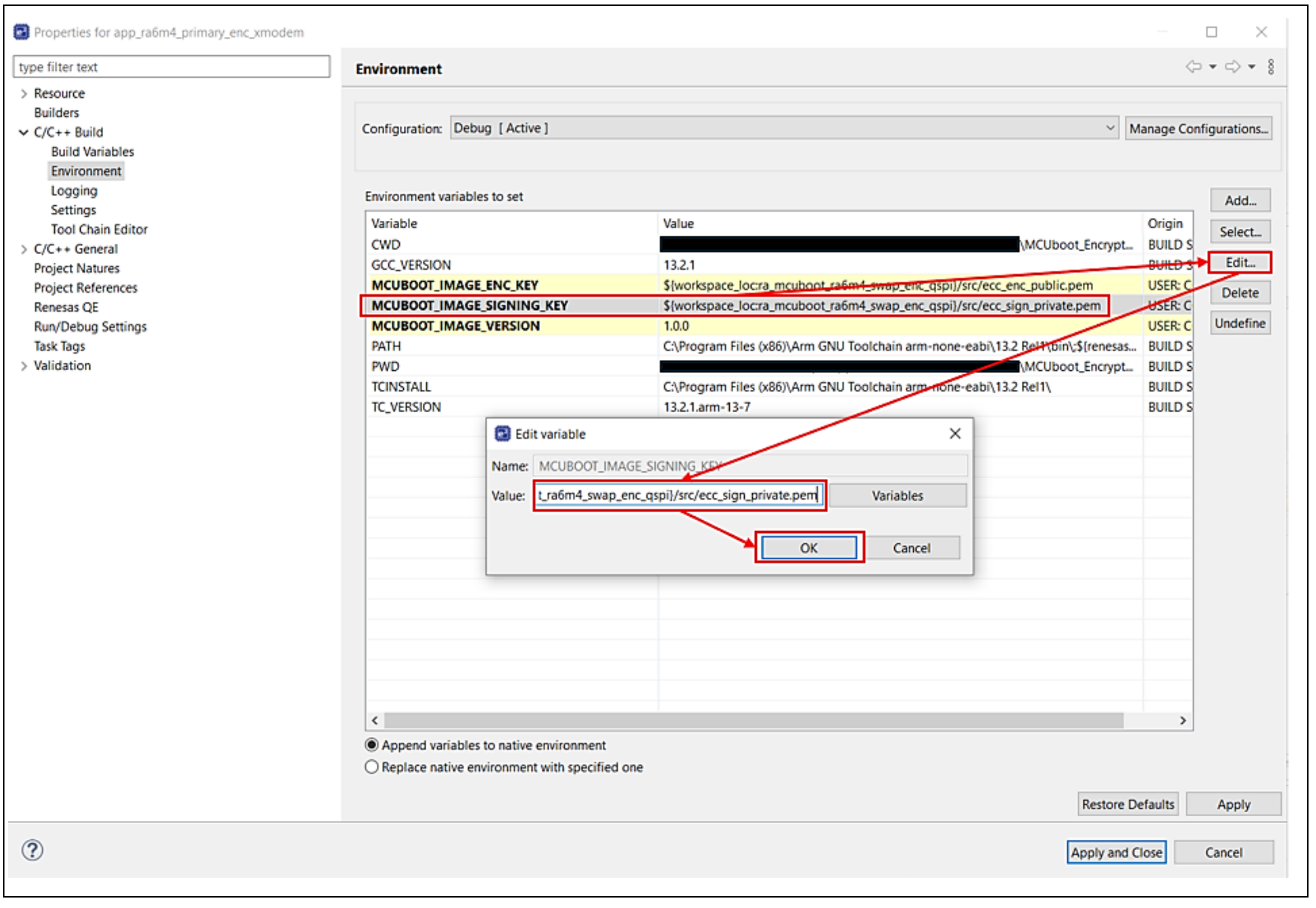

Figure 57. Configure the Application Project to use the Custom Image Signing

18. Update the encryption key configuration of the primary application project.
    Choose "**MCUBOOT_IMAGE_ENC_KEY**" Variable, click **Edit** and define the **Value** as:
    ${workspace_loc:ra_mcuboot_ra6m4_swap_enc_qspi}/src/ecc_enc_public.pem
    Click **OK** > **Apply and Close**.

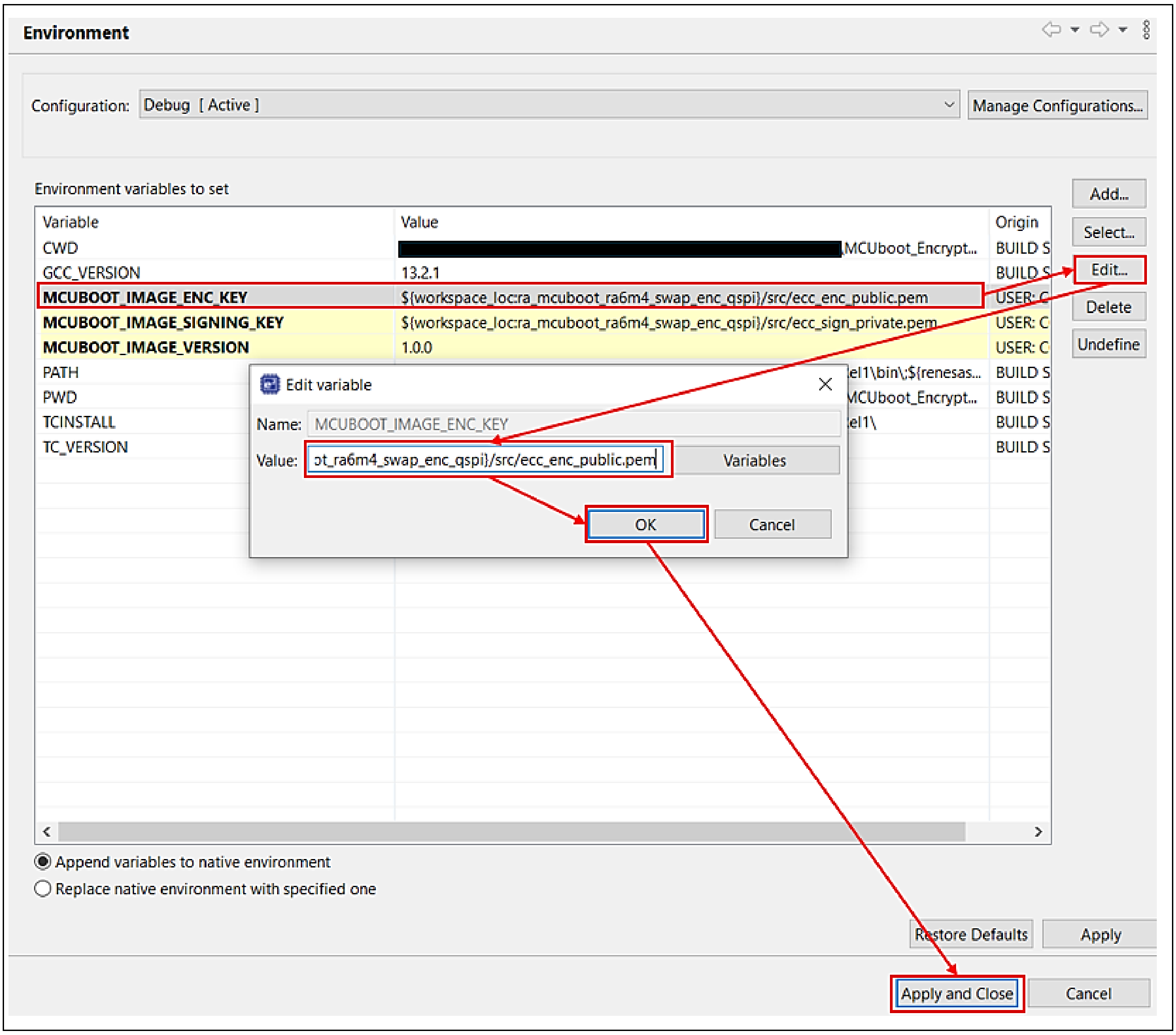

Figure 58. Configure the Application Project to use the Custom Key for the Image Encryption Process

19. For the primary application project, navigate to the smart configurator, click **Generate Project Content** and recompile the application.
20. Repeat **steps 17, 18** and **19** for the secondary application project.
21. Follow steps in **section 3.2.1** to erase the flash.
22. Start the Debug session from the primary application project, resume twice to boot the primary application. The three LEDs should be blinking.
    User can now use the XModem to download and verify the operation fo the secondary application image.

7. Appendix

7.1 Making the Bootloader for Cortex-M33 Immutable

To make the bootloader immutable, the flash blocks containing the bootloader must be locked from being programmed and erased.

The RA6M4 features two sets of registers which facilitate flash block locking. Block Protect Setting (BPS) registers feature bits that map to individual flash blocks. When a bit is set to zero, the corresponding flash block cannot be erased or programmed. The Permanent Block Protect Setting (PBPS) Registers have a similar bit mapping to flash blocks. When a bit is set in one of these registers, the corresponding flash block is permanently locked from being erased and programmed so long as the same bit in the Block Protect Setting Register is also cleared to zero. This process is irreversible. Once a flash block is **permanently** locked, it cannot be unlocked again.

Based on the example bootloaders provided in this application project, the flash blocks used by the bootloader are:

- RA6M4 Overwrite Mode: block 0-7
- RA6M4 Swap Mode: block 0-8
- RA6M3 Overwrite Mode: block 0-7

Users can refer to the *RA Family MCU Securing Data at Rest using Arm TrustZone Application Project* to understand the operational flow of setting up the Flash Block Protection.

Note that ticking the BSP0 and PBPS0 Flash Block settings will permanently lock the flash blocks. This **CANNOT** be reversed. Further details can be found in sections 6.2.6 and 6.2.7 of the RA6M4 Hardware User’s Manual.

7.2 Making the Bootloader for Cortex-M4 Immutable

Customers can 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 for Cortex-M4 Immutable. Section *PC Application to Permanently Lock the FAW* in the same application note describes how to handle Flash locking in production mode.

7.3 Device Lifecycle Management for Renesas RA Cortex-M33 MCUs

Once the bootloader development is finished, the user may want to transition the Device Lifecycle State of the RA Cortex-M33 MCU to lock down the debugger and the serial programming interface.

We recommend referring to the Device Lifecycle State Transitions in the Production Flow section in the *Renesas RA Family MCU Device Lifecycle Management Key Installation Application Note* to understand the device lifecycle management options during production.

The operational overview of how to use Renesas Flash Programmer to perform these transitions is explained in the *Overview of Device Lifecycle State Transitions using Renesas Flash Programmer* section.

7.4 Device Lifecycle Management for Renesas RA Cortex-M4 MCUs

Once the bootloader development is finished, you may want to set up the ID Code protection on Renesas RA Cortex-M4 MCU to lock down the debugger and the serial programming interface.

You can 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 setting to control the device lifecycle management of the RA Cortex-M4 MCUs using the ID Code protection method.

8. References

1. Renesas RA Family MCU Securing Data at Rest using Security MPU Application Project (R11AN0416)
2. Renesas RA Family MCU Securing Data at Rest using Arm TrustZone® Application Project (R11AN0468)
3. Renesas RA Family MCU Device Lifecycle Management Key Injection Application Project (R11AN0469)
4. Renesas RA Family MCU Security Design with TrustZone – IP Protection Application Project (R11AN0467)

9. Website and Support

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

EK-RA6M4 Resources	renesas.com/ra/ek-ra6m4
EK-RA6M3 Resources	renesas.com/ra/ek-ra6m3
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	Oct.28.22	-	First release document
1.10	Nov.02.23	-	Update to FSPv5.0.0
1.20	Jan.24.24	-	Updates throughout the document
1.30	Oct.21.24	-	Update to FSPv5.5.0

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/.