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. Currently 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 will refer to the above-mentioned documents wherever possible.

The Renesas Flexible Software Package (FSP) integrates an MCUboot port across the entire RA MCU Families starting from FSP v3.0.0. Renesas RA2 MCU series are based on Arm® Cortex®-M23 core and have 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 guide.

This application note walks the user through the secure bootloader creation using the MCUboot Module with TinyCrypt for enhanced security on Renesas EK-RA2E1 kit. In addition, examples of how to configure application project to use the bootloader are provided. In this first release, the application image is checked for integrity but not authenticity. This application project will be updated to add application image signature checking when future FSP release supports this functionality.

For Renesas RA6 and RA4 MCU Series, Renesas provides an application project *Using MCUboot with Renesas RA MCU Application Project* which walks users through using MCUboot with RA6 and RA4 MCU groups with Mbed Crypto module. See the References section for information on this Application Project.

Required Resources

Development tools and software

- e^2 studio ISDE v2021-04 or greater
- Renesas Flexible Software Package (FSP) v3.1.0 or later
- 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.

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 you have some experience with the Renesas e^2 studio IDE development. Users are required to 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, or end users who are involved with designing application systems involving usage of a secure bootloader with Renesas RA2 MCU family.

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 start-up and the firmware switch part of the firmware update process. The operation of switching of the firmware from the original image to a new image depends on the image upgrade methods. 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. Below is an example of the single image MCUboot memory map. For more information on the MCUboot memory layout, refer to the Flash Map section of the reference MCUboot website.

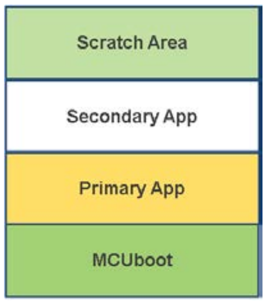


Figure 1. Single Image MCUboot Memory Flash Map

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

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

1. The bootloader verifies the integrity and authenticity of the Secondary image.
2. Upon successful authentication, the bootloader will switch to the new image based on the update method selected. Available update methods are introduced in section 1.2.2.
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. The firmware image can be authenticated by hash (SHA-256) and digital signature validation. For this release of the RA2 MCU bootloader, only integrity of the bootloader is checked based on hash (SHA-256). Image signature generation is not supported with current FSP release.

1.1.2 Applications Update Strategies

The following are the update strategies supported by MCUboot. The Renesas FSP MCUboot Module in FSP v3.1.0 does not yet support all of the MCUboot update strategies. The analysis of pros and cons is based on the MCUboot functionality, but not the FSP v3.1.0 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 supported yet.
- **Swap**
  In the Swap image upgrade mode, the active image is also stored in the Primary slot and it will always be 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 will be swapped. The new image will then start 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 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 yet. Runtime image testing is not supported at the time of the release of this application project.
- **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 not currently supported by the Renesas RA FSP, but it is planned for a future release.
- **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 MCU environment. Please refer to the RAM Loading section in the MCUboot page for more information on this update strategy. This image update mode does not support encrypted images (see MCUboot documentation on encrypted image operation).
  RAM loading update mode is not supported by the Renesas RA FSP.

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

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 recommended 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. FSP v3.1.0 release uses TinyCrypt v0.2.8. For the RA2 bootloader design, SHA256 from TinyCrypt is used to ensure the application image integrity.

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. For FSP, a template for signing the new firmware image based on the properties specified in the MCUboot Module is output in a comment at the top of `ra_cfg/mcu-tools/include/mcuboot_config/mcuboot_config.h`.
- 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 is support with FSP v3.1.0, image signature authentication will be supported in future releases.

Customize the RA2 Bootloader

Customizing the Bootloader involves 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. Details on the operational flow of these optimization are described in section 3.
  - 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)
- Refer to section 1 to understand the available features and section 3 for where and how to update the bootloader features.

2.4 Production Recommendations for RA2 MCU

2.4.1 Make 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 Disable 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 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 walks the user through the creation process of the RA2 bootloader provided in this application note.

The example bootloader which user will create by following this section is provided in the `RA2_secure_bootloader.zip`. User can follow section 7 to exercise the example bootloader and application projects without going through the creation process in this section.

3.1 Include the MCUboot Module in the Bootloader Project

1. Launch e^2 studio 2021-04 and start to establish a new C/C++ Project. Click **File** > **New** > **C/C++ Project.**

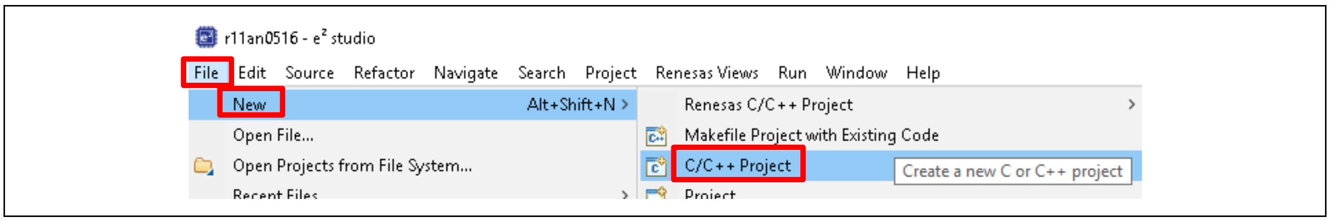

r11an0516 - e² studio
File Edit Source Refactor Navigate Search Project Renesas Views Run Window Help
New Alt+Shift+N >
Open File...
Open Projects from File System...
Recent Files
Renesas C/C++ Project
Makefile Project with Existing Code
C/C++ Project
Create a new C or C++ project
Project

Figure 2. Start a New Project

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

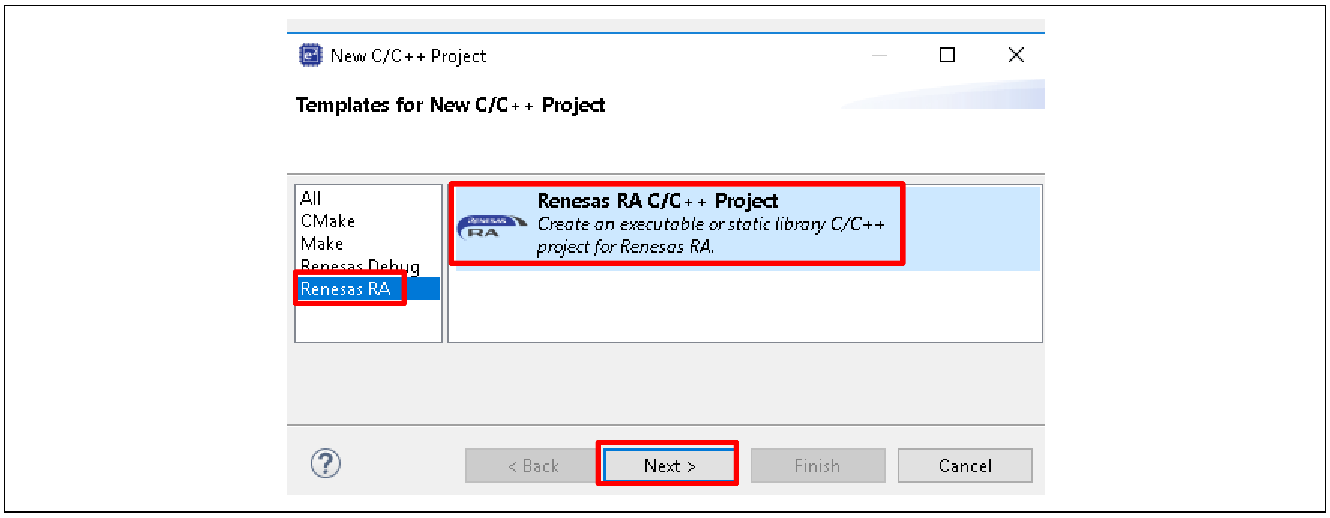
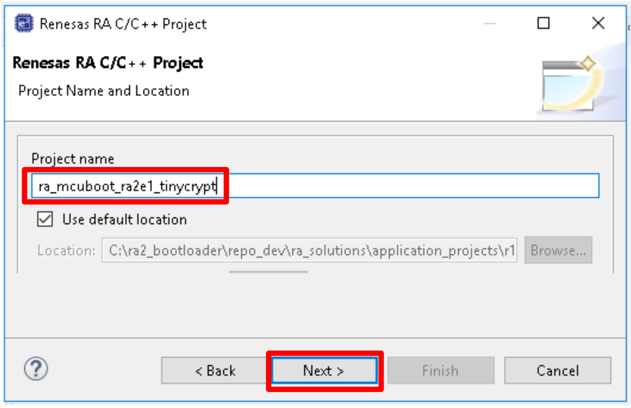


Figure 3. Choose Renesas RA C/C++ Project

3. Provide a project name such as **ra_mcuboot_ra2e1_tinycrypt**. Click **Next**. You can choose other names for the bootloader. If a different name is chosen, you need to update the corresponding instructions in this application note to the project name used.


Renesas RA C/C++ Project

Project Name and Location

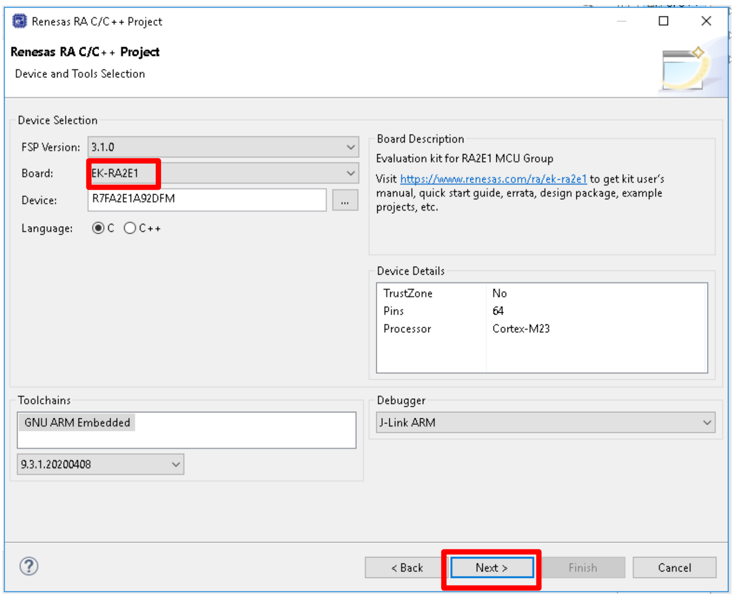
Project name: ra_mcuboot_ra2e1_tinycrypt

☑ Use default location

Location: C:\ra2_bootloader\repo_dev\ra_solutions\application_projects\r1

Figure 4. Name the Bootloader Project

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


Renesas RA C/C++ Project

Device and Tools Selection

Device Selection

FSP Version: 3.1.0

Board: EK-RA2E1

Device: R7FA2E1A92DFM

Language: [✓ C] C++

Board Description

Evaluation kit for RA2E1 MCU Group

Visit https://www.renesas.com/ra/ek-ra2e1 to get kit user's manual, quick start guide, errata, design package, example projects, etc.

Device Details

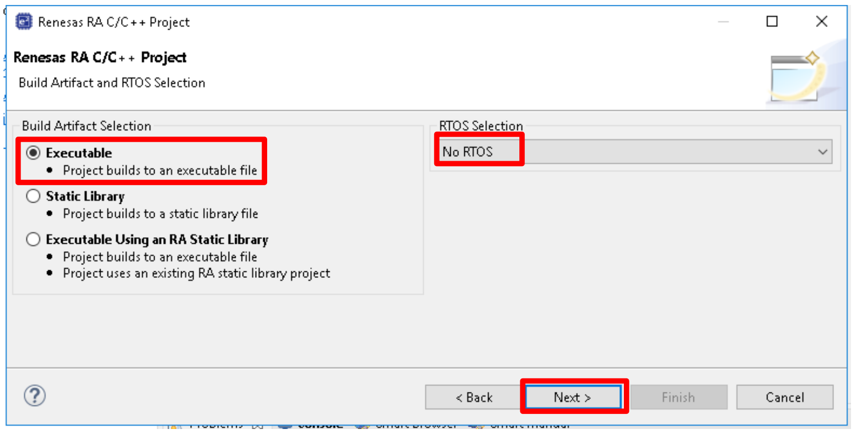
TrustZone	No
Pins	64
Processor	Cortex-M23

Toolchains: GNU ARM Embedded, 9.3.1.20200408

Debugger: J-Link ARM

Figure 5. Select the Board

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


Renesas RA C/C++ Project

Build Artifact and RTOS Selection

Build Artifact Selection

[✓ Executable]
- Project builds to an executable file

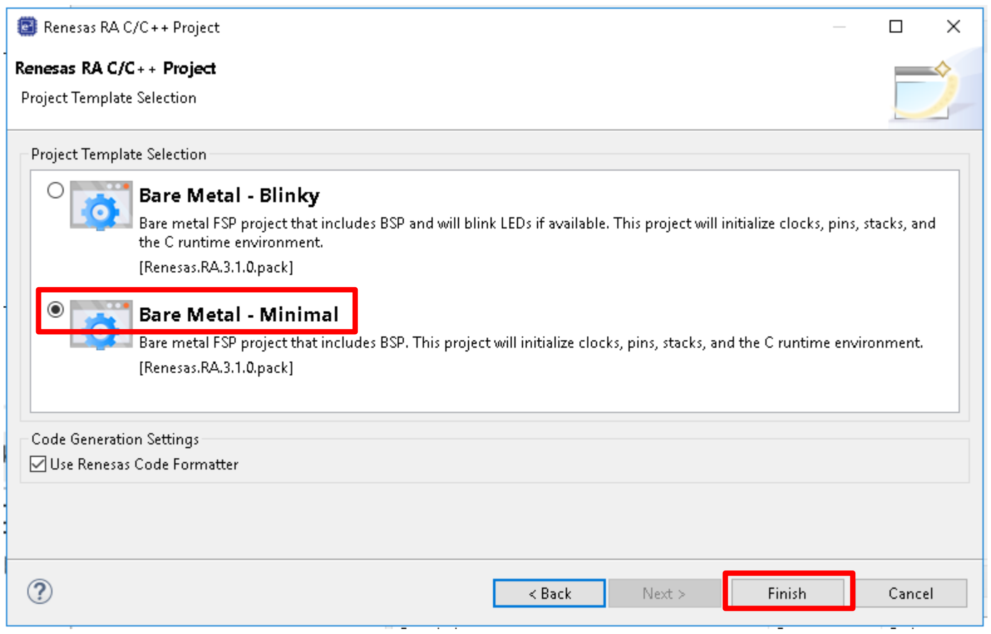
Static Library
- Project builds to a static library file

Executable Using an RA Static Library
- Project builds to an executable file
- Project uses an existing RA static library project

RTOS Selection: No RTOS

Figure 6. Choose to Build Executable and No RTOS

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



Renesas RA C/C++ Project

Renesas RA C/C++ Project

Project Template Selection

Project Template Selection

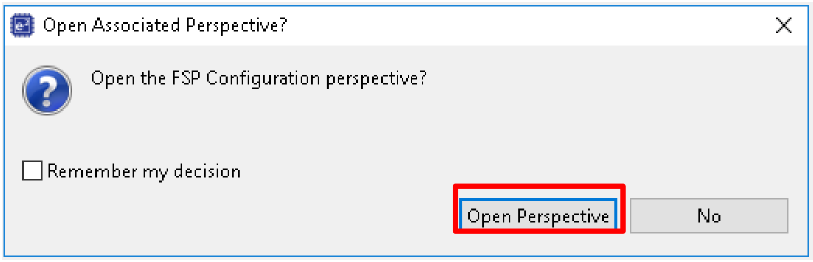
- ○ **Bare Metal - Blinky**
  Bare metal FSP project that includes BSP and will blink LEDs if available. This project will initialize clocks, pins, stacks, and the C runtime environment.
  [Renesas.RA.3.1.0.pack]
- ● **Bare Metal - Minimal**
  Bare metal FSP project that includes BSP. This project will initialize clocks, pins, stacks, and the C runtime environment.
  [Renesas.RA.3.1.0.pack]

Code Generation Settings

☑ Use Renesas Code Formatter

Figure 7. Choose the Project Template

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



Open Associated Perspective?

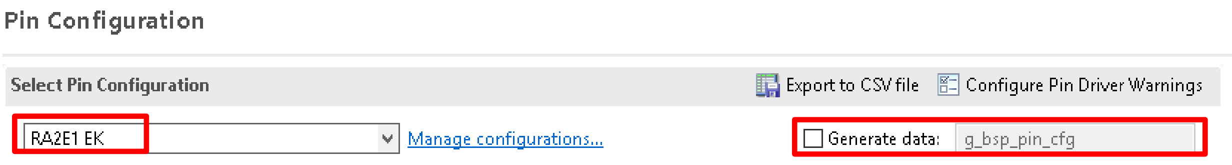
Open the FSP Configuration perspective?

☐ Remember my decision

Figure 8. Choose Open the FSP Configuration perspective

The project will now be created, and the bootloader project configuration will be displayed.

1. Select the **Pins** tab and uncheck **Generate data** for **RA2E1 EK.**



Pin Configuration

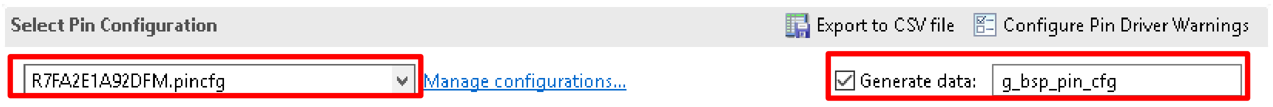
Select Pin Configuration

RA2E1 EK

☐ Generate data: g_bsp_pin_cfg

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.



Select Pin Configuration

R7FA2E1A92DFM.pincfg

☑ Generate data: g_bsp_pin_cfg

Figure 10. Select R7FA2E1A92DFM.pincfg and Generate data g_bsp_pin_cfg

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

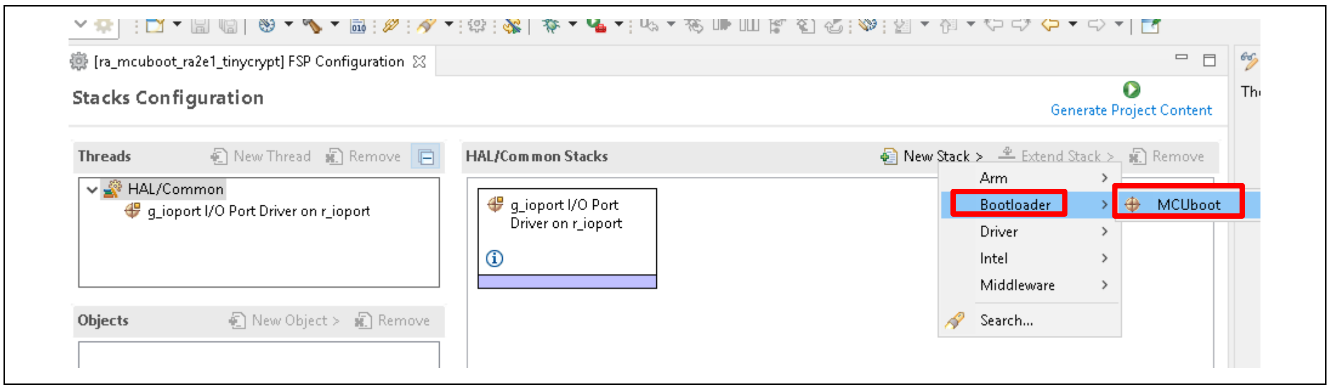
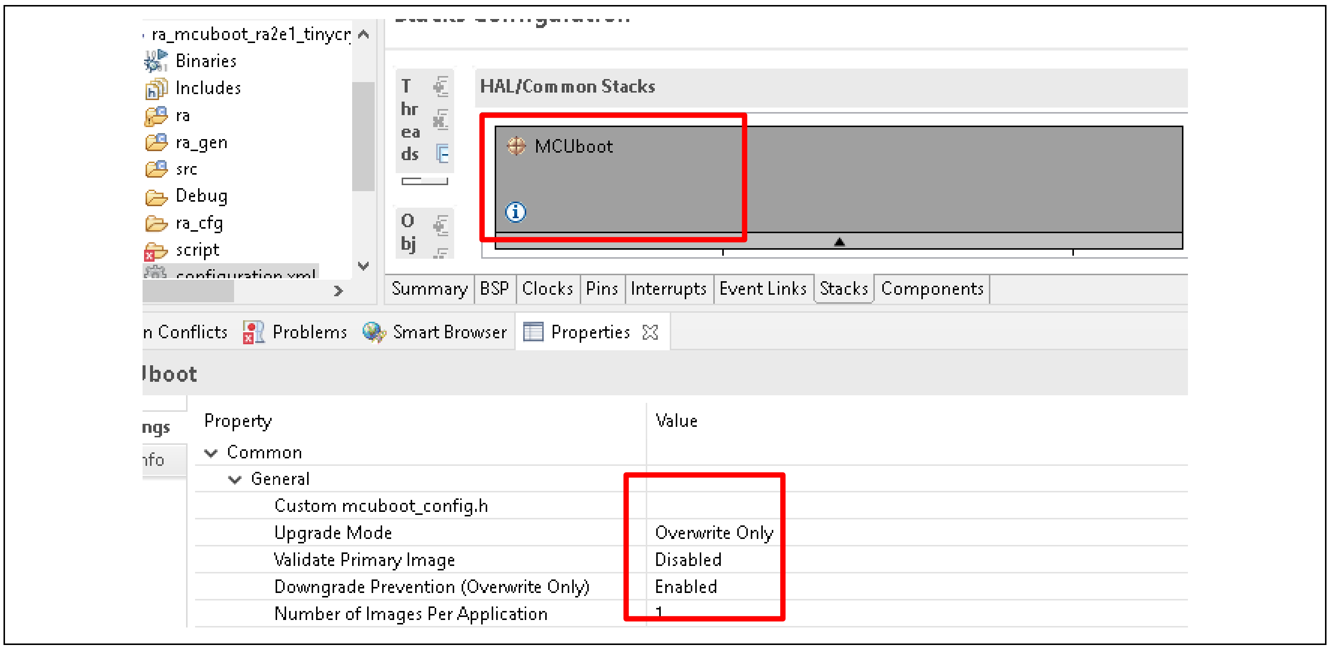


Figure 11. Add the MCUboot Port

3. Once the MCUboot module is added to the project, configure **General** properties of MCUboot as shown below.



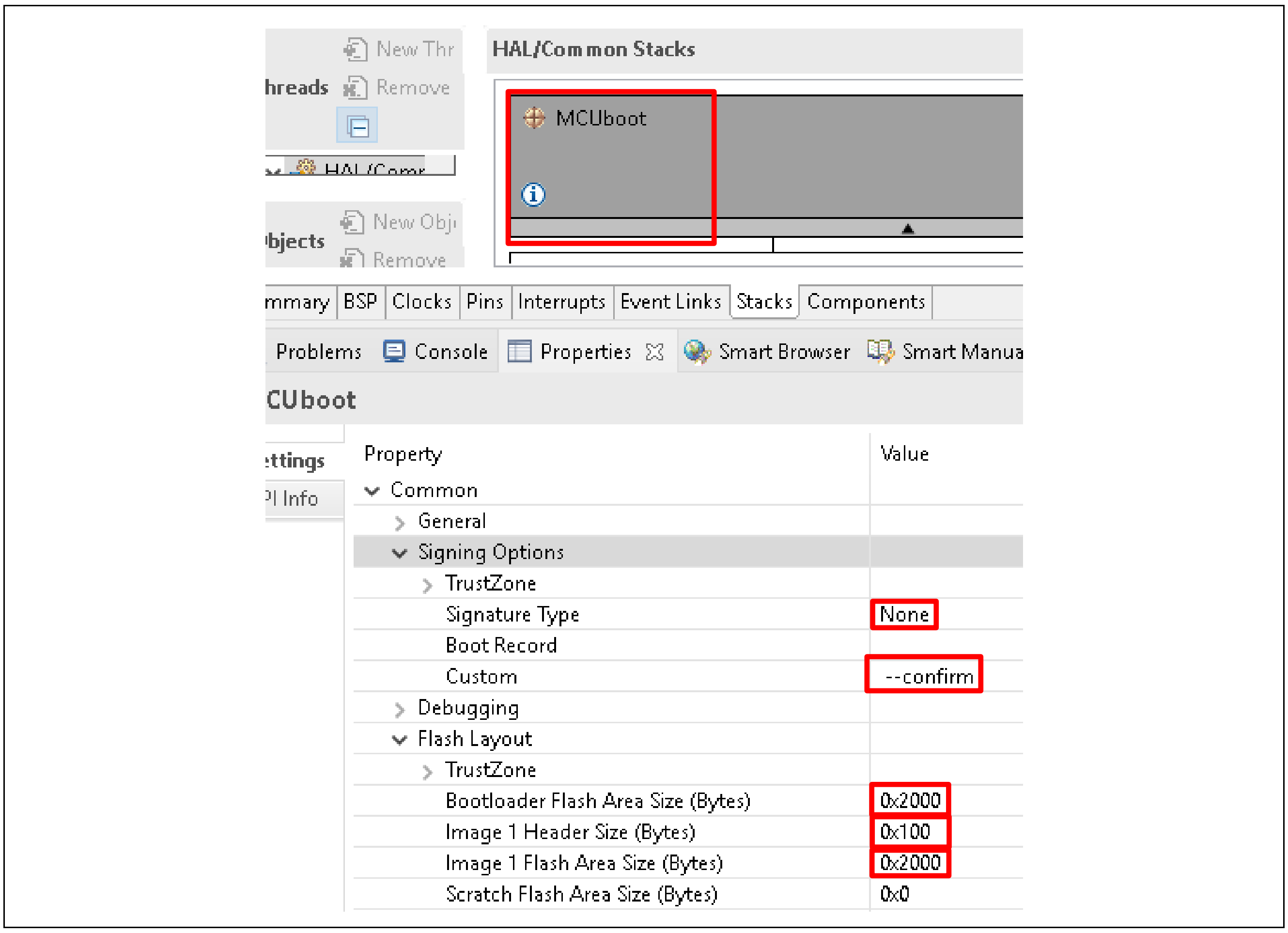
Property	Value
Common	
General	
Custom mcuboot_config.h	
Upgrade Mode	Overwrite Only
Validate Primary Image	Disabled
Downgrade Prevention (Overwrite Only)	Enabled
Number of Images Per Application	1

Figure 12. FSP RA2 MCUboot Module General Configuration

The following explains the various properties configured:

- **Custom mcuboot_config.h**: The default `mcuboot_config.h` file contains the MCUboot Module configuration that the user selected from the RA configurator. The user can create a custom version of this file to achieve additional bootloader functionalities available in MCUboot.
- **Upgrade Mode**: This property configures the application image upgrade method. The available options are Overwrite Only, Overwrite Only Fast, and Swap. Choose **Overwrite Only** for this bootloader project.
- **Validate Primary Image**: If this property is enabled, the bootloader will always check the signature of the image in the primary slot before booting, even if no upgrade was performed. This version of the FSP does not support signature generation with TinyCrypt, so this function is **Disabled**.
- **Number of Images Per Application:** This property allows user 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):** When this property is **Enabled**, a new firmware with a lower version number will not overwrite the existing application. For how to set the version number of an image, refer to section 3.5, step 6) .

4. Next configure the following properties to remove the warnings for the **MCUboot** module.



Property	Value
Common	
General	
Signing Options	
TrustZone	
Signature Type	None
Boot Record	
Custom	--confirm
Debugging	
Flash Layout	
TrustZone	
Bootloader Flash Area Size (Bytes)	0x2000
Image 1 Header Size (Bytes)	0x100
Image 1 Flash Area Size (Bytes)	0x2000
Scratch Flash Area Size (Bytes)	0x0

Figure 13. Update Configurations for the MCUboot Module

For both single-image and two-image configurations, the following four 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 both RA2 MCUs, the scratch area should be set up to 0x800 when Swap mode is used.

The properties under **TrustZone** are not used for RA2 MCUs since they do not have TrustZone.

The bootloader we are creating in this section will have a size of 0x2000 when all the configuration updates are followed through in this section. Otherwise, the bootloader may result in different flash usage size and the **Bootloader Flash Area Size** property needs to be updated accordingly.

Image 1 Flash Area Size is the application code flash area usage, which should be larger than the application code flash usage and on a 0x800 boundary for RA2 MCUs.

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. FSP v3.1.0 does not support signature generation with TinyCrypt, the **Signature Type** is set to **None**, so only hash is used for the image integrity checking.

Notes on setting the **Custom** property:

- By default, FSP sets **--confirm** for the Custom property.
- When **--confirm** is set:
  - For Overwrite upgrade mode, the new image will always overwrite the original application image.
  - For Swap upgrade mode, if the new image passed the integrity check, it will swap with the original image and will be booted. On the next MCU reset, there will be no Swap.

5. Next add the **TinyCrypt** module. The **TinyCrypt Hardware** includes Hardware accelerated AES functionality which is not used in the bootloader, this module is not selected. The **MbedTLS (Crypto Only)** module has a larger memory footprint compared with **TinyCrypt** and is not used in this bootloader design.

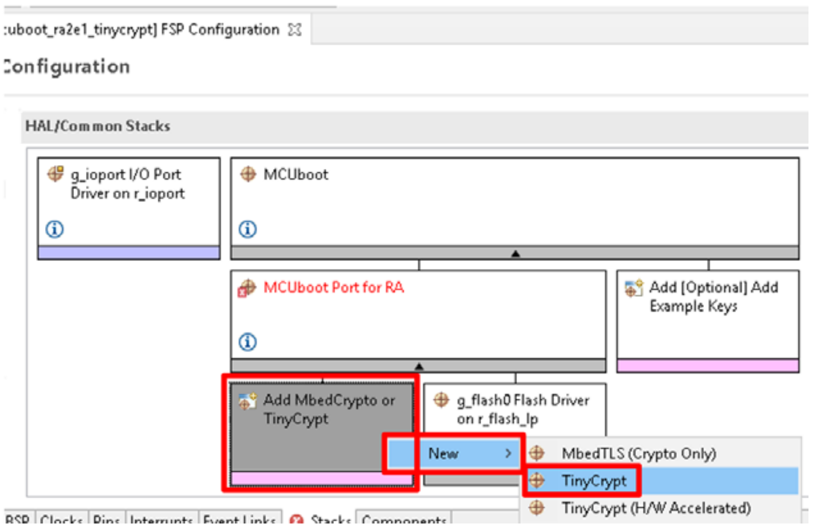
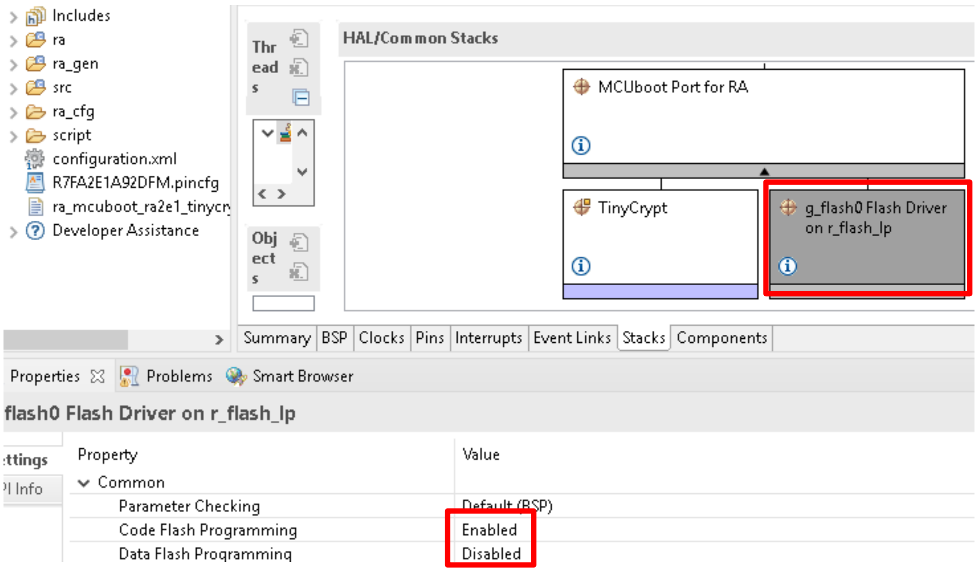

:uboot_ra2e1_tinycrypt] FSP Configuration
Configuration
HAL/Common Stacks
g_ioport I/O Port Driver on r_ioport
MCUboot
MCUboot Port for RA
Add [Optional] Add Example Keys
Add MbedCrypto or TinyCrypt
g_flash0 Flash Driver on r_flash_lp
New
MbedTLS (Crypto Only)
TinyCrypt
TinyCrypt (H/W Accelerated)
BSP Clocks Pins Interrupts Event Links Stacks Components

Figure 14. Select TinyCrypt Module

6. Now click on the **Flash Driver** block and set the **Code Flash Programming** to **Enabled**. As **Data Flash Programming** is not used in the bootloader, select **Disabled** for the **Data Flash Programming** to reduce the bootloader memory footprint.



Property	Value
Common	
Parameter Checking	Default (BSP)
Code Flash Programming	Enabled
Data Flash Programming	Disabled

Figure 15. Enable Code Flash programming

7. In this step, save the `Configruation.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();`

8. Notice that by default the **I/O Port** Driver is brought in to 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.

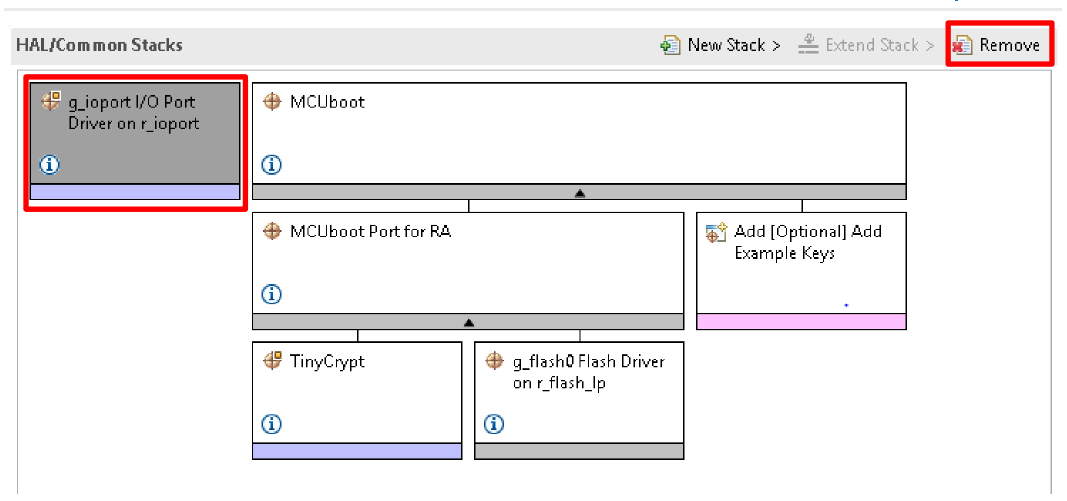
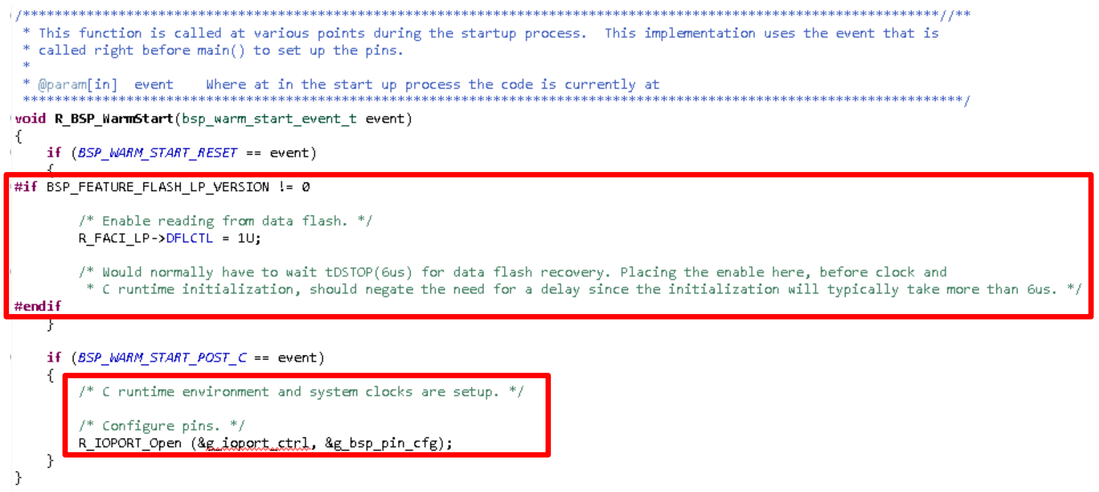

HAL/Common Stacks
New Stack >
Extend Stack >
Remove
g_ioport I/O Port Driver on r_ioport
MCUboot
MCUboot Port for RA
Add [Optional] Add Example Keys
TinyCrypt
g_flash0 Flash Driver on r_flash_lp

Figure 16. Remove the I/O Port Stack

After the I/O Port is deleted, remove all sections of code referencing the I/O Port API. E.g., remove below two sections of the code in the red boxes in the function `R_BSP_WarmStart` in `hal_entry.c` as shown in Figure 17.



```
/*******************************************************************************************************************//**
 * 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 17. Remove Unused Code in hal_entry.c

3.2 Optimize the Bootloader Project for Size

To further optimize the bootloader project for size, you can put some application code in the first 2k flash sector. There is about 1k of used flash space between the interrupt vector and the RA2E1 ROM registers. You can create a section (`.code_in_gap`) in the linker script to store some application code in this section.

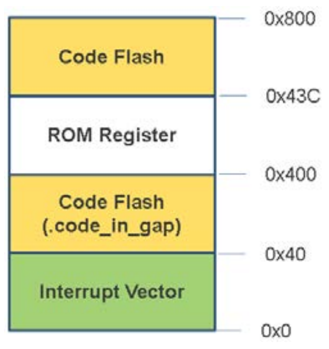
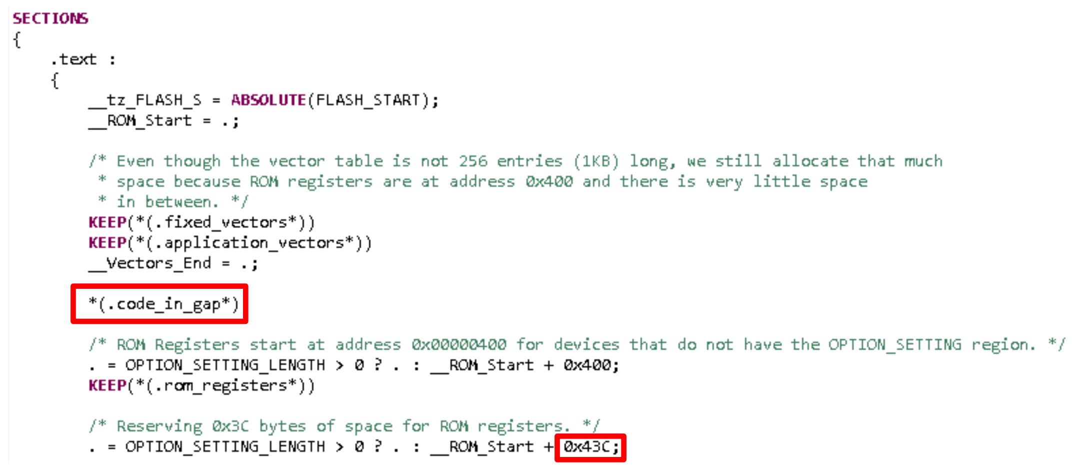


Figure 18. First Flash Sector

First, update the default linker script to include section `.code_in_gap` between the interrupt vector and the ROM register as shown in Figure 19. In addition, the application code after the ROM register can start at 0x43C instead of 0x500 as used in the default linker script.



```
SECTIONS
{
    .text :
    {
        __tz_FLASH_S = ABSOLUTE(FLASH_START);
        __ROM_Start = .;

        /* Even though the vector table is not 256 entries (1KB) long, we still allocate that much
         * space because ROM registers are at address 0x400 and there is very little space
         * in between. */
        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 0x3C bytes of space for ROM registers. */
        . = OPTION_SETTING_LENGTH > 0 ? . : __ROM_Start + 0x43C;
```

Figure 19. Linker Script Update

Next, you can choose some functions to put in the `.code_in_gap` section in order to reduce the flash usage after 0x43C. What functions to put in the `.code_in_gap` section is the user's choice. This example bootloader has chosen some functions to put in the `.code_in_gap` section as explained in this section.

In `\src\hal_entry.c`, put these two function prototypes at the top of `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*");
```

Figure 20. hal_entry.c functions in .code_in_gap

In `\ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\bootutil.h`, put below function in the `.code_in_gap` section:

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

Figure 21. bootutil.h function in .code_in_gap

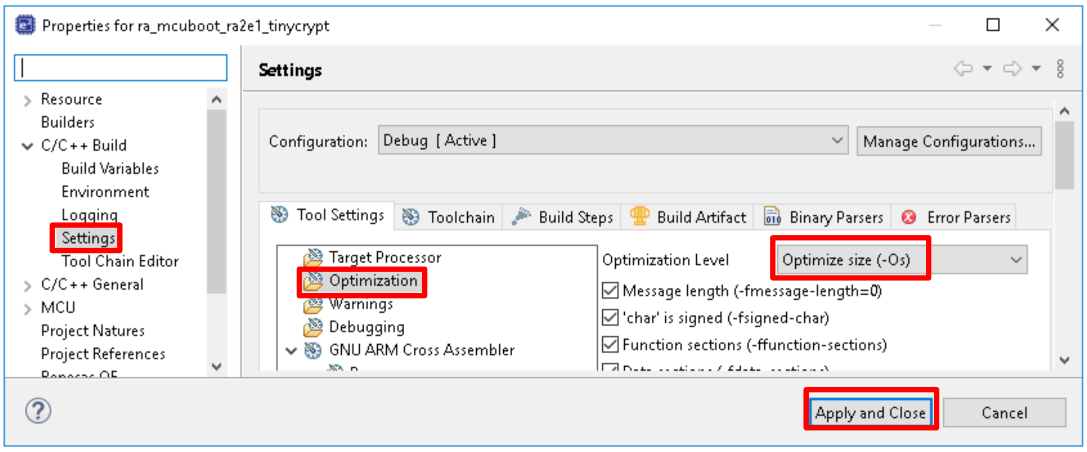
In `\ra\mcu-tools\MCUboot\boot\bootutil\include\bootutil\image.h`, put below function in the `.code_in_gap` section:

```
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 22. image.h function in .code_in_gap

3.3 Compile the Bootloader Project

Once all the above updates are done, change the compiling optimization to **Optimize size** and compile the project.



Configuration: Debug [Active]

Optimization Level: Optimize size (-Os)

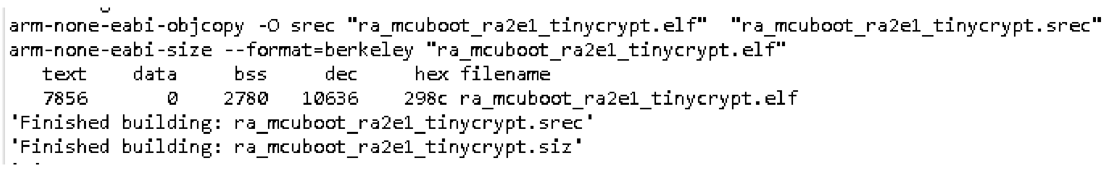
☑ Message length (-fmessage-length=0)

☑ 'char' is signed (-fsigned-char)

☑ Function sections (-ffunction-sections)

Figure 23. Optimize Bootloader Size

After the above update, compile the project. The size of the flash usage should be around 0x7800.



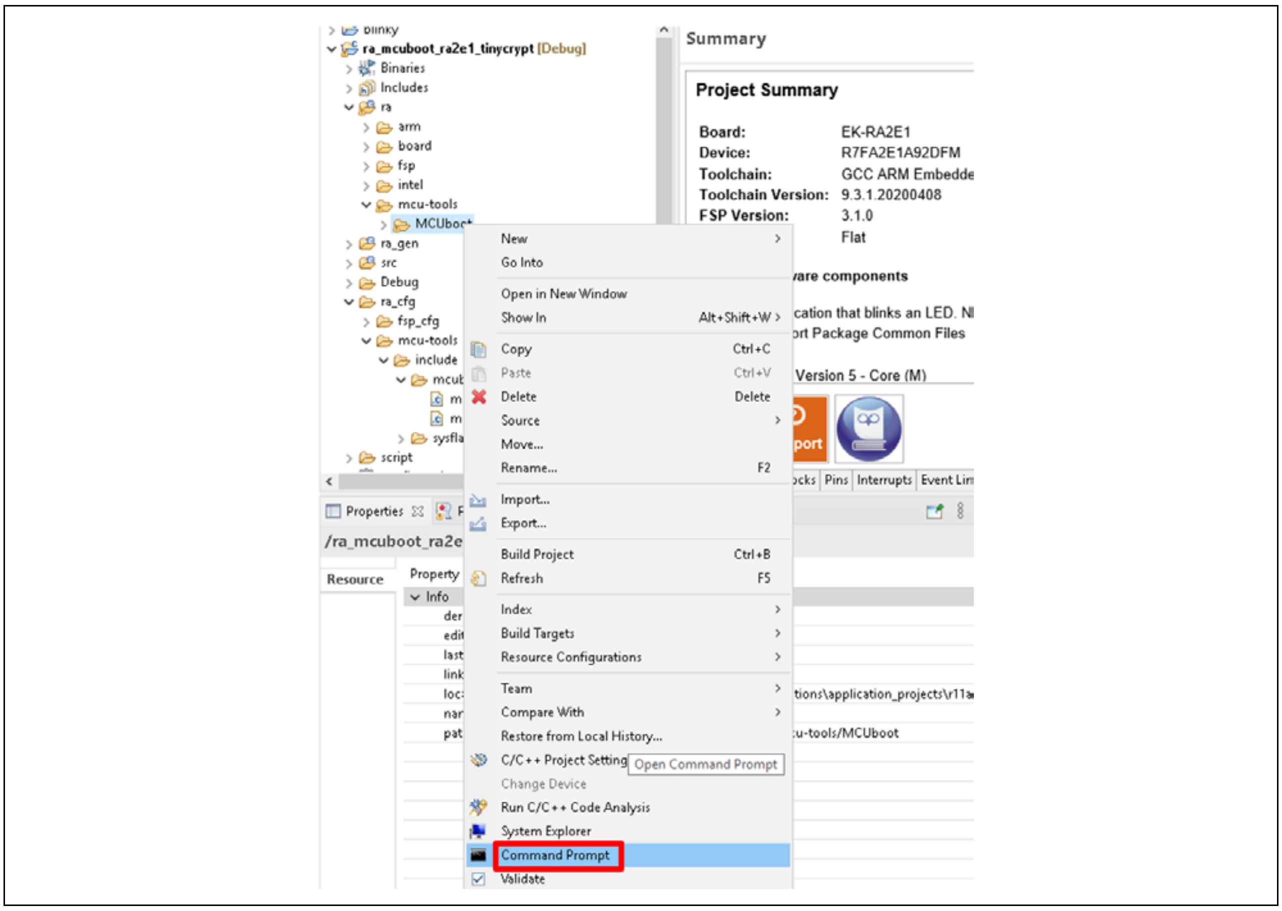
```
arm-none-eabi-objcopy -O srec "ra_mcuboot_ra2e1_tinycrypt.elf"  "ra_mcuboot_ra2e1_tinycrypt.srec"
arm-none-eabi-size --format=berkeley "ra_mcuboot_ra2e1_tinycrypt.elf"
   text    data     bss     dec     hex filename
   7856       0    2780   10636    298c ra_mcuboot_ra2e1_tinycrypt.elf
'Finished building: ra_mcuboot_ra2e1_tinycrypt.srec'
'Finished building: ra_mcuboot_ra2e1_tinycrypt.siz'
```

Figure 24. Compile the Bootloader

3.4 Configure the Python Signing Environment

Signing the application image can be done using a post-build step in e^2 studio using the image signing tool `Imgtool.py` is included with MCUboot. This tool is integrated as a post-build tool in e^2 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 **ra_mcuboot_ra2e1>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.



Project Summary

Board:	EK-RA2E1
Device:	R7FA2E1A92DFM
Toolchain:	GCC ARM Embedde
Toolchain Version:	9.3.1.20200408
FSP Version:	3.1.0
	Flat

Figure 25. Open the Command Prompt

It is recommended to upgrade 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
```

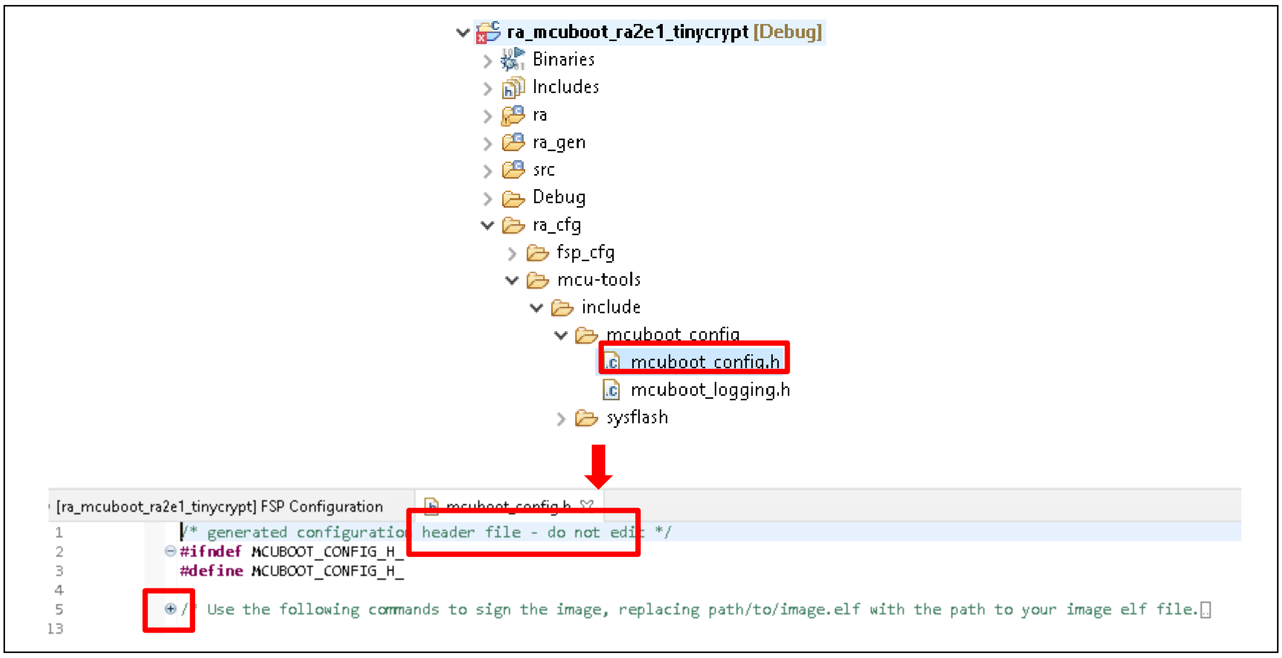
This will verify and install any dependencies that are required.

3.5 Create the Signing Command

This section provides instructions for creating the signing command using the bootloader. Even though a signature is not generated, hashing is generated using the following procedure.

e^2 studio collects user input from the RA configurator and generates the bootloader configuration in `mcuboot_config.h`. The template for generating the signing command is also generated at the top of `mcuboot_config.h`. This template is generated based on user selections from the various configuration properties in the RA Configurator.

1. In the **Project Explorer**, navigate to the **ra_mcuboot_ra2e1_tinycrypt>ra_cfg>mcu-tools>include>mcuboot_config** folder and open the `mcuboot_config.h` file.

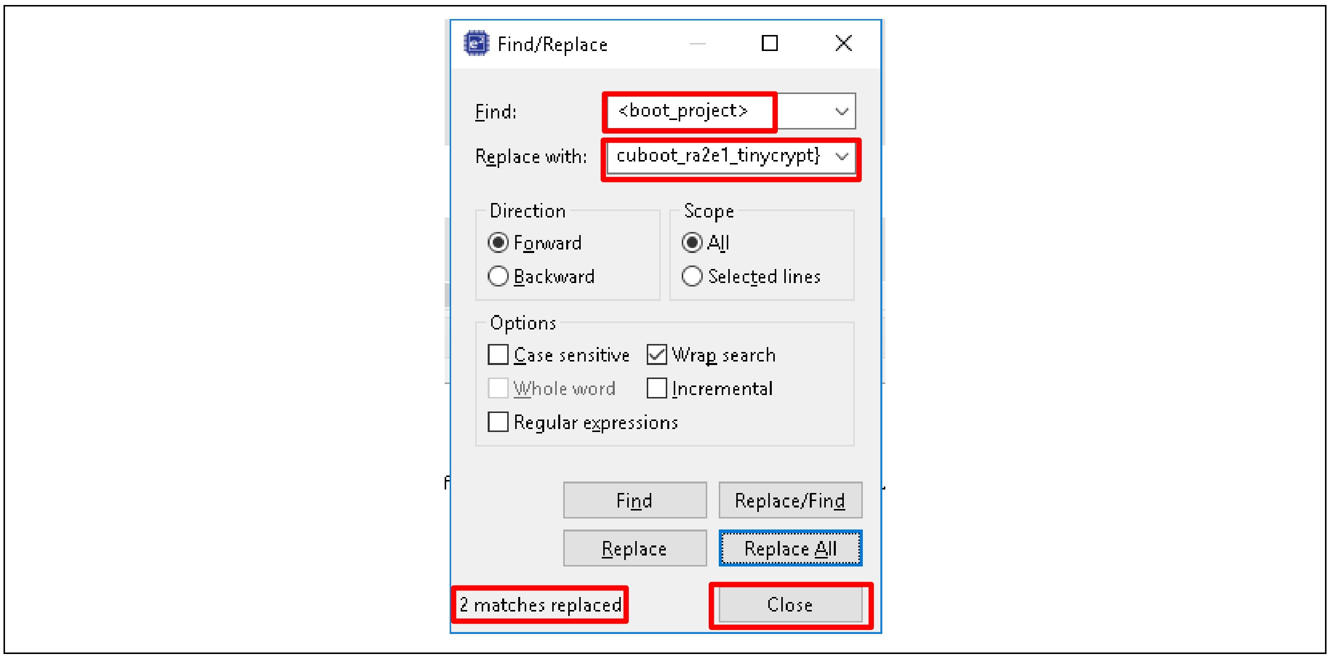


```
/* generated configuration header file - do not edit */
#ifndef MCUBOOT_CONFIG_H_
#define MCUBOOT_CONFIG_H_

/* Use the following commands to sign the image, replacing path/to/image.elf with the path to your image elf file.
```

Figure 26. Open mcuboot_config.h

2. Expand the comment block in red in Figure 26. Type **ctrl+F** to open the **Find/Replace** dialogue box and input the key words in the quotation marks (please do not include the quotation mark). Input **Find <boot_project>** and set **Replace All** with **${workspace_loc:ra_mcuboot_ra2e1_tinycrypt}**. Do not add any spaces before or after the key words in bold when inputting into the Find/Replace dialog.



Find: <boot_project>
Replace with: cuboot_ra2e1_tinycrypt}

Direction: [✓ Forward] Backward
Scope: [✓ All] Selected lines

Options:
- ☐ Case sensitive
- ☑ Wrap search
- ☐ Whole word
- ☐ Incremental
- ☐ Regular expressions

2 matches replaced

Figure 27. Configure the Bootloader Project Path

There should be two matches replaced. If the number of matches is not two, check this operation again.

3. Change the **Find** setting to **path/to/image** and the **Replace with** setting to **${ProjName}**. Do not add any spaces before or after the key words in bold when inputting into the Find/Replace dialog. Click **Replace All**.

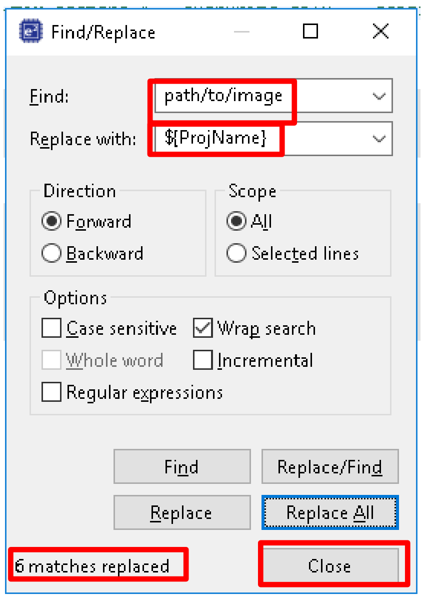
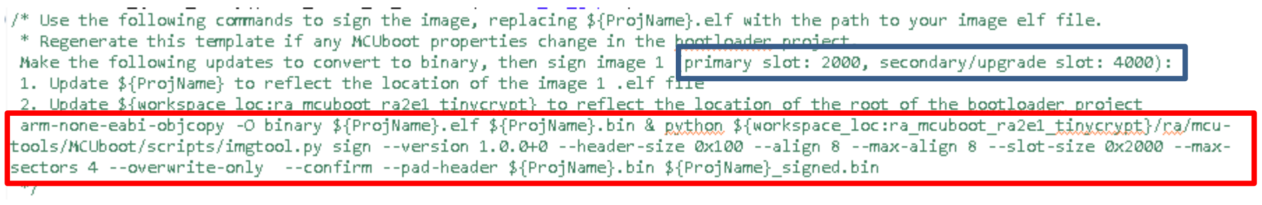


Figure 28. Configure the Application Project Path

There should be six matches replaced. If the number of matches is not six, it is recommended to check this operation again. Then **Close** the dialogue box.

4. Add an & to the end of the following line so it looks like this:
`arm-none-eabi-objcopy -O binary ${ProjName}.elf ${ProjName}.bin &`
5. Copy the comment block from `mcuboot_config.h` and paste it into the `hal_entry.c` file before the comment block "Quick setup for MCUboot" in the **ra_mcuboot_ra2e1_tinycrypt** project to save the edits. Since the `mcuboot_config.h` file will be overwritten when the project is built, otherwise your edits will be lost.
6. Navigate to the **Window>Editor>Toggle Word Wrap** to make the lines easier to read. Delete the carriage return after the & to join the two command lines.



```
/* Use the following commands to sign the image, replacing ${ProjName}.elf with the path to your image elf file.
 * Regenerate this template if any MCUboot properties change in the bootloader project
 Make the following updates to convert to binary, then sign image 1 primary slot: 2000, secondary/upgrade slot: 4000):
 1. Update ${ProjName} to reflect the location of the image 1 .elf file
 2. Update ${workspace_loc:ra_mcuboot_ra2e1_tinycrypt} to reflect the location of the root of the bootloader project
 arm-none-eabi-objcopy -O binary ${ProjName}.elf ${ProjName}.bin & python ${workspace_loc:ra_mcuboot_ra2e1_tinycrypt}/ra/mcu-tools/MCUboot/scripts/imgtool.py sign --version 1.0.0+0 --header-size 0x100 --align 8 --max-align 8 --slot-size 0x2000 --max-sectors 4 --overwrite-only  --confirm --pad-header ${ProjName}.bin ${ProjName}_signed.bin
 */
```

Figure 29. Application Image Signing Command

Note that the application image version number is defined in the signing command leading by `--version` option. By default, FSP generates a signing command with version number 1.0.0+0. You can manually update this information to define different version numbers. When **Downgrade Prevention** property is **Enabled**, a new image with lower version numbers, such as 0.9.0+0, will not overwrite an image with version 1.0.0+0 which resides in the Primary slot.

Note that the application image load addresses are indicated in this comment area (in the blue boxes). These addresses will be used in Figure 41 and Figure 48

7. Save `\ra_mcuboot_ra2e1_tinycrypt\hal_entry.c`.

The command lines for the primary image in red box as shown in Figure 29 is the signing command for the Primary image. We will use this command in the next section when signing the Application image.

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 below 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.

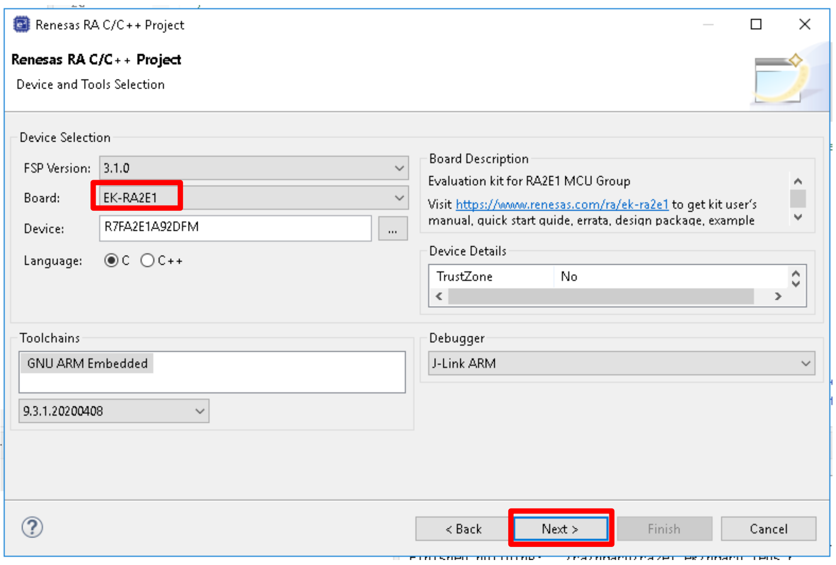
This section uses a simple blinky project to demonstrate how to use the bootloader with the blinky application. After the initial blinky project is established, we need to configure this blinky project to the use the bootloader project generated in the previous section. We also need to sign the blinky project using the signing command generated in Figure 29. Detailed instructions are provided in this section.

Note that user 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 for their specific application.

4.1 Generate the Initial Application Project

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

1. Assign the name **blinky** to this new project.
2. Click **Next** and choose **EK-RA2E1** as the **Board** from the drop-down menu. Then click **Next**.


Renesas RA C/C++ Project

Renesas RA C/C++ Project

Device and Tools Selection

Device Selection

FSP Version: 3.1.0

Board: EK-RA2E1

Device: R7FA2E1A92DFM

Language: [✓ C] C++

Board Description

Evaluation kit for RA2E1 MCU Group

Visit https://www.renesas.com/ra/ek-ra2e1 to get kit user's manual, quick start guide, errata, design package, example

Device Details

TrustZone No

Toolchains

GNU ARM Embedded

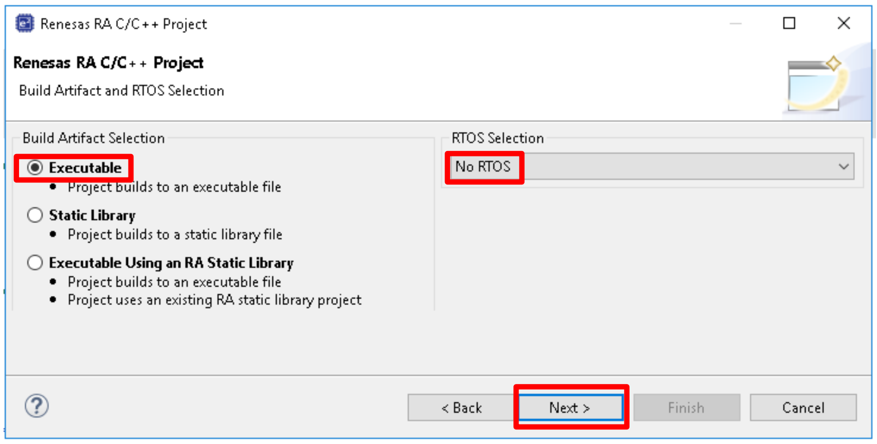
9.3.1.20200408

Debugger

J-Link ARM

Figure 30. Choose EK-RA2E1 for the Blinky Project

3. In the next screen, select **Executable** as the **Build Artifact** and **No RTOS** for the **RTOS Selection**.


Renesas RA C/C++ Project

Renesas RA C/C++ Project

Build Artifact and RTOS Selection

Build Artifact Selection

(●) Executable
- Project builds to an executable file

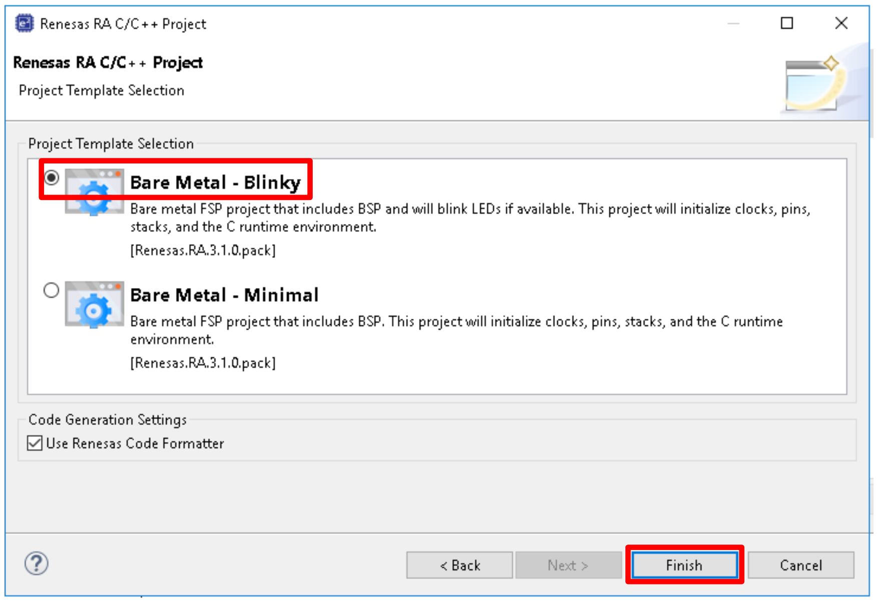
( ) Static Library
- Project builds to a static library file

( ) Executable Using an RA Static Library
- Project builds to an executable file
- Project uses an existing RA static library project

RTOS Selection: No RTOS

Figure 31. Choose to Build Executable with No RTOS

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


Renesas RA C/C++ Project

Renesas RA C/C++ Project

Project Template Selection

Project Template Selection

(●) Bare Metal - Blinky
Bare metal FSP project that includes BSP and will blink LEDs if available. This project will initialize clocks, pins, stacks, and the C runtime environment.
[Renesas.RA.3.1.0.pack]

( ) Bare Metal - Minimal
Bare metal FSP project that includes BSP. This project will initialize clocks, pins, stacks, and the C runtime environment.
[Renesas.RA.3.1.0.pack]

Code Generation Settings

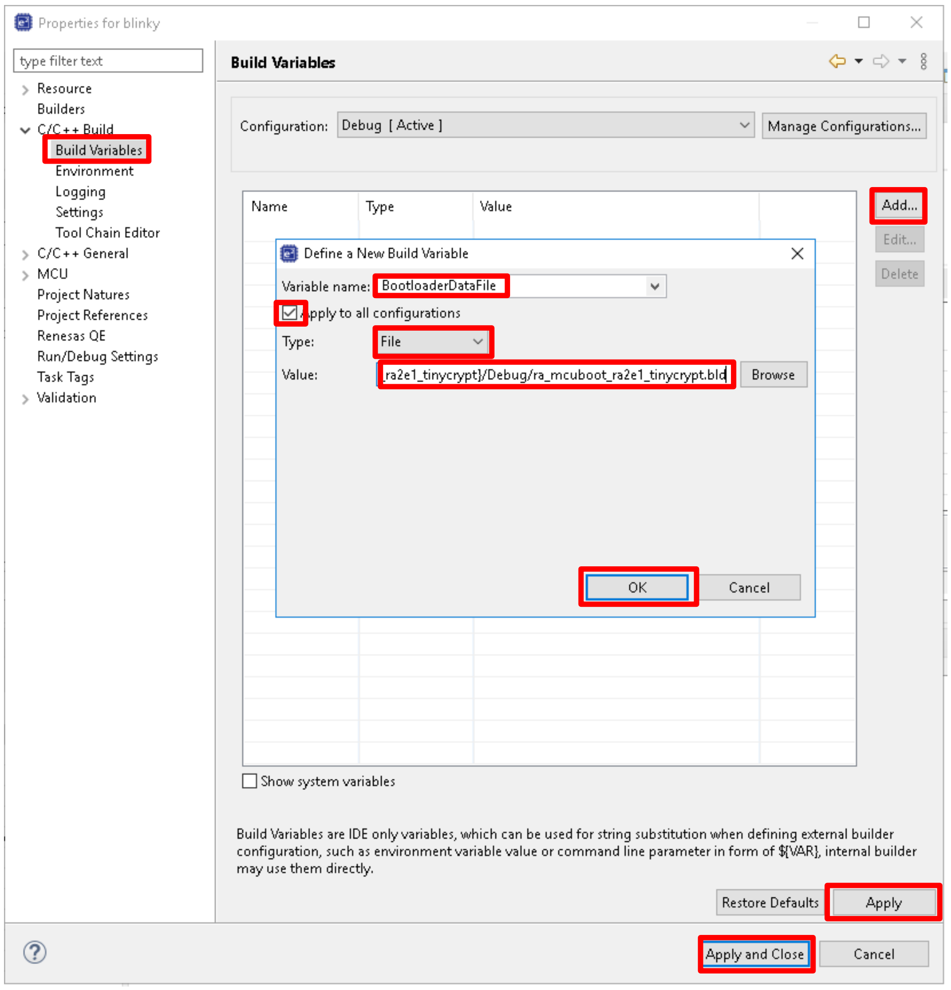
[✓] Use Renesas Code Formatter

Figure 32. Choose Bare Metal – Blinky as Project Template

4.2 Configure the Existing Application to Use the Bootloader Project

Note that 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 in consideration of the size the application project. When using the bootloader with a different application project, the Image 1 Flash Area Size property in Figure 13 should be adjusted accordingly. User also needs to regenerate the signing command to the corresponding application by following the steps in this section.

Right click on the `blinky` 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 **${workspace_loc:ra_mcuboot_ra2e1_tinycrypt}/Debug/ra_mcuboot_ra2e1_tinycrypt.bld** for the value. Click **OK** to save the changes.



Configuration: Debug [Active]

Variable name: BootloaderDataFile

☑ Apply to all configurations

Type: File

Value: ra2e1_tinycrypt}/Debug/ra_mcuboot_ra2e1_tinycrypt.bld

Build Variables are IDE only variables, which can be used for string substitution when defining external builder configuration, such as environment variable value or command line parameter in form of ${VAR}, internal builder may use them directly.

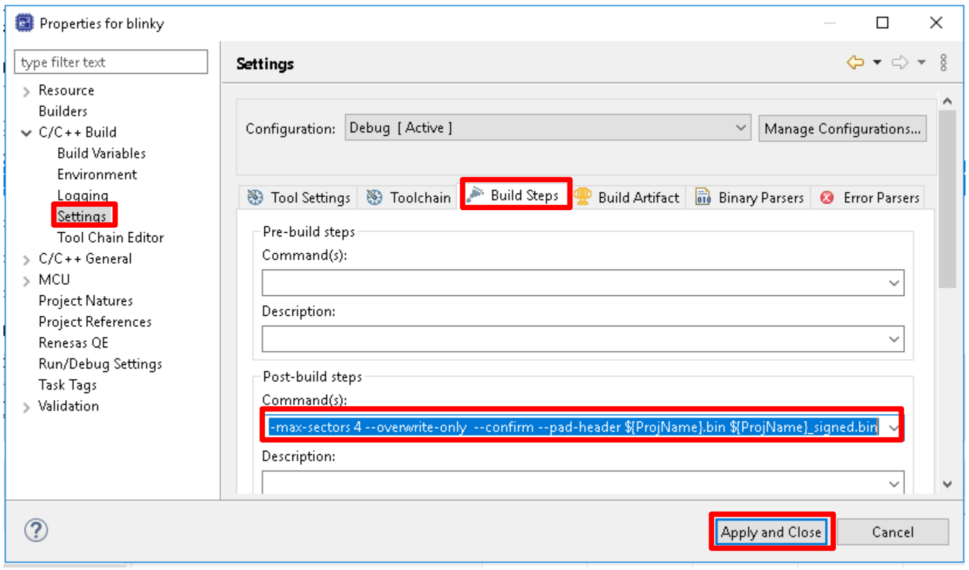
Figure 33. Configure the Build Variable to Use the Bootloader

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

4.3 Signing the Application Image

In this section, we will use the signing command generated in Figure 29 to sign the newly created application. Copy the command line generated from Figure 29.

Edit the project **Properties** for the **blinky** project and navigate to the **C/C++ Build > Settings**. Click on the **Build Steps** tab and paste the command line that just copied into the **Post-build steps > commands** and press **Apply and Close**.



Configuration: Debug [Active]

Pre-build steps
Command(s):
Description:

Post-build steps
Command(s):

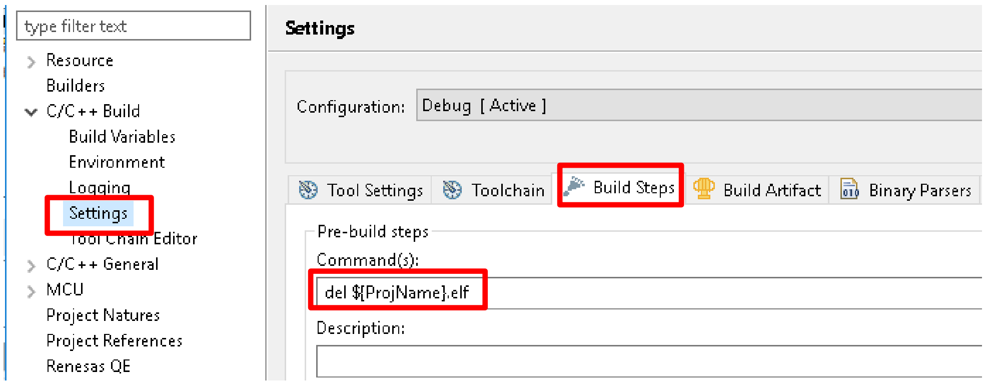
```
-max-sectors 4 --overwrite-only  --confirm --pad-header ${ProjName}.bin ${ProjName}_signed.bin
```

Description:

Figure 34. Configure the Post Build Command

If you do rebuild the bootloader project after changing any of the signing and signature **Properties** of the MCUboot module, you will have to repeat all steps in section 3.4 to resign the application project.

Since the signing command is after the build step, if the application project has already been built without adding the signing command and there are no updates in the application project, the `\debug\*.elf` file needs to be removed for the image signing option to proceed. Optionally, user can add the following **Pre-build Steps > Command(s)** to allow the signed image to be always regenerated regardless of whether there are updates from the application project.



Configuration: Debug [Active]

Pre-build steps
Command(s):

```
del ${ProjName}.elf
```

Description:

Figure 35. Configure the Pre-build Command

Next, a user can add the RTT Viewer usage related application code to the blinky project. Unzip `RA2_secure_bootloader.zip`, open the `RA2_secure_bootlader\blinky\src` folder and copy all files under `\src` to the newly established blinky project `\src` folder.

At this point, user can click **Generate Project Content** and compile the blinky project and ensure `\debug\blinky_signed.bin` is generated.

5. Booting the Initial Application Project

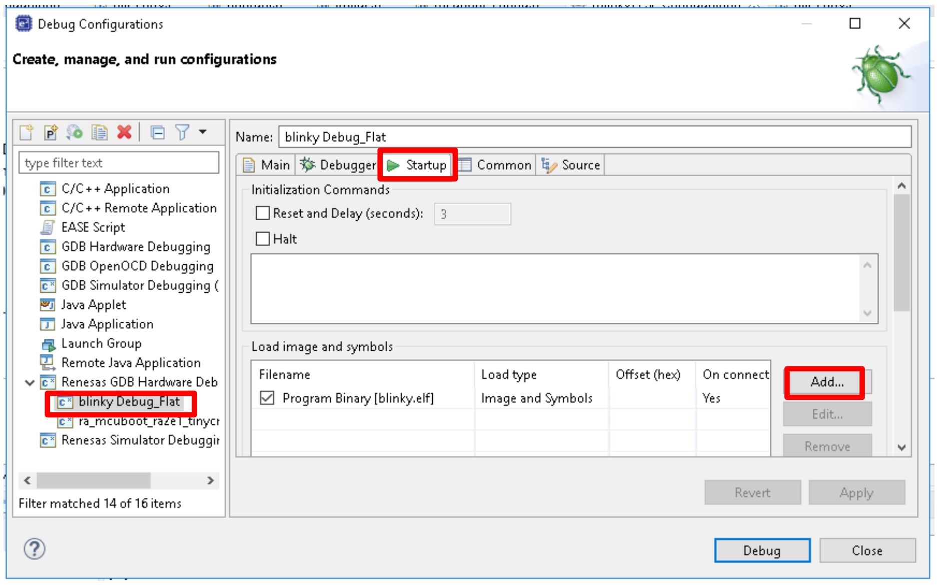
5.1 Set Up the Hardware

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**

Make sure the **blinky Debug_Flat** is selected and select the **Startup** tab.


Debug Configurations

Create, manage, and run configurations

Name: blinky Debug_Flat

Main | Debugger | Startup | Common | Source

Initialization Commands

☐ Reset and Delay (seconds): 3

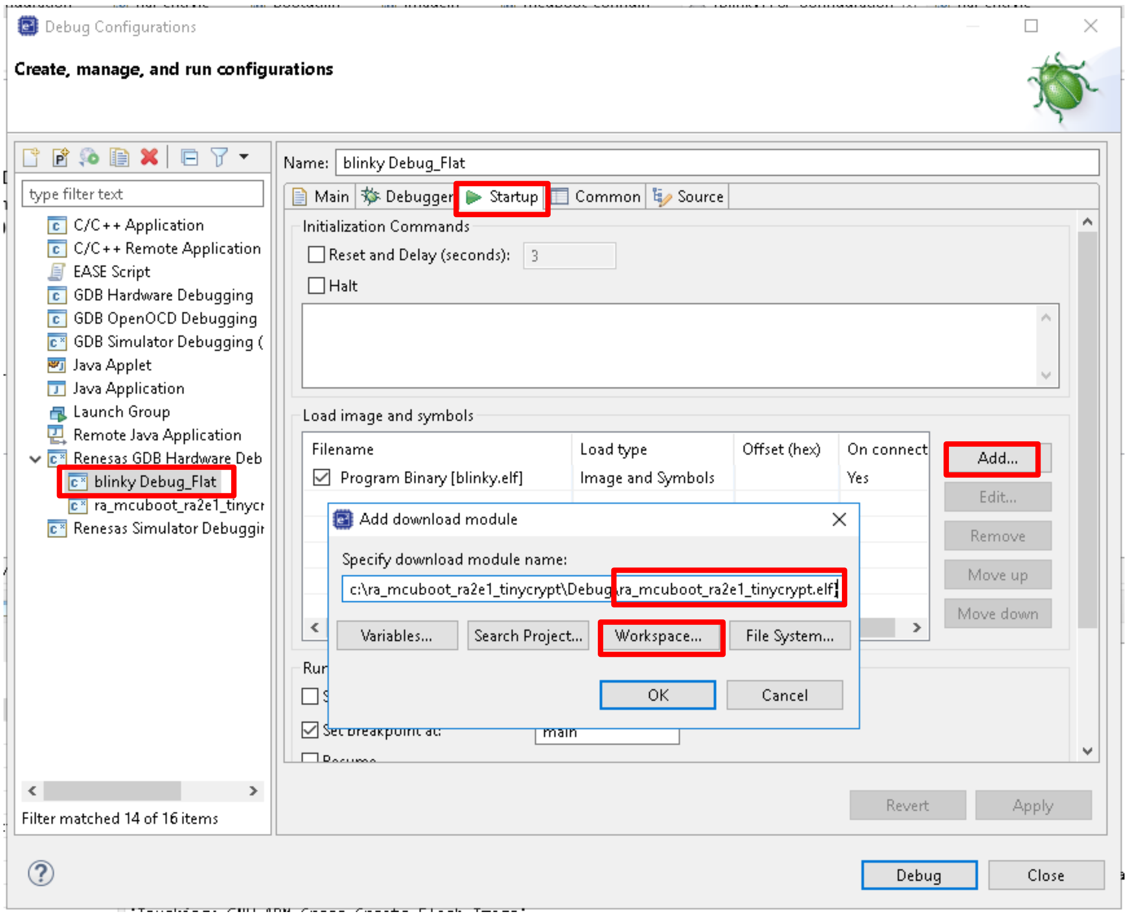
☐ Halt

Load image and symbols

Filename	Load type	Offset (hex)	On connect
☑ Program Binary [blinky.elf]	Image and Symbols		Yes

Figure 36. Configure the Primary Project Debug Startup

Click **Add…** and then **Workspace** and navigate to the **ra_mcuboot_ra2e1_tinycrypt** project and select the **ra_mcuboot_ra2e1_tinycrypt.elf** file from the debug folder. Click **OK**.


Debug Configurations

Create, manage, and run configurations

Name: blinky Debug_Flat

Main | Debugger | Startup | Common | Source

Initialization Commands

☐ Reset and Delay (seconds): 3

☐ Halt

Load image and symbols

Filename	Load type	Offset (hex)	On connect
☑ Program Binary [blinky.elf]	Image and Symbols		Yes

Add download module

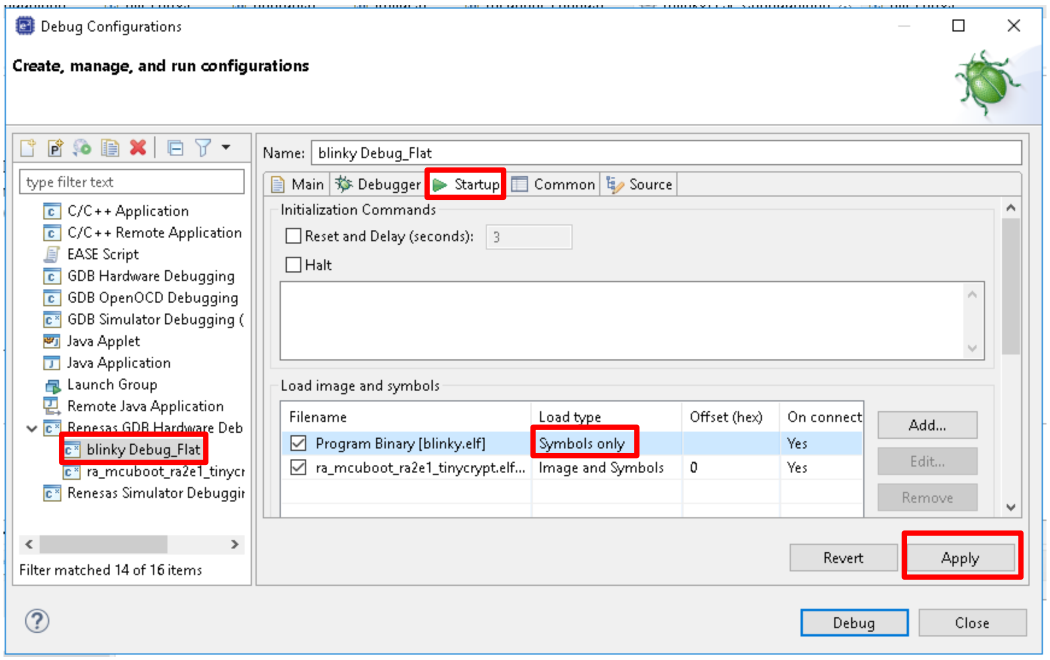
Specify download module name:

c:\ra_mcuboot_ra2e1_tinycrypt\Debug\ra_mcuboot_ra2e1_tinycrypt.elf

☑ Set breakpoint at: main

Figure 37. Add the Bootloader Project to Debug Configuration

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



Name: blinky Debug_Flat

Initialization Commands

☐ Reset and Delay (seconds): 3

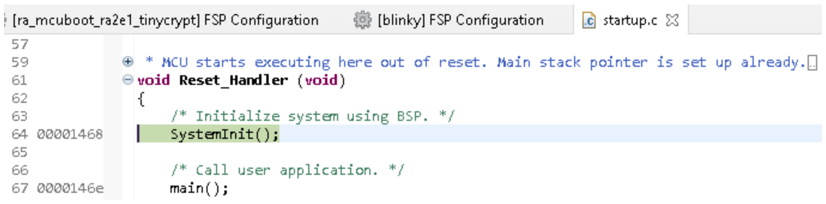
☐ Halt

Load image and symbols

Filename	Load type	Offset (hex)	On connect
☑ Program Binary [blinky.elf]	Symbols only		Yes
☑ ra_mcuboot_ra2e1_tinycrypt.elf...	Image and Symbols	0	Yes

Figure 38. Select to load Symbols only for the Application Project

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


[ra_mcuboot_ra2e1_tinycrypt] FSP Configuration
[blinky] FSP Configuration
startup.c

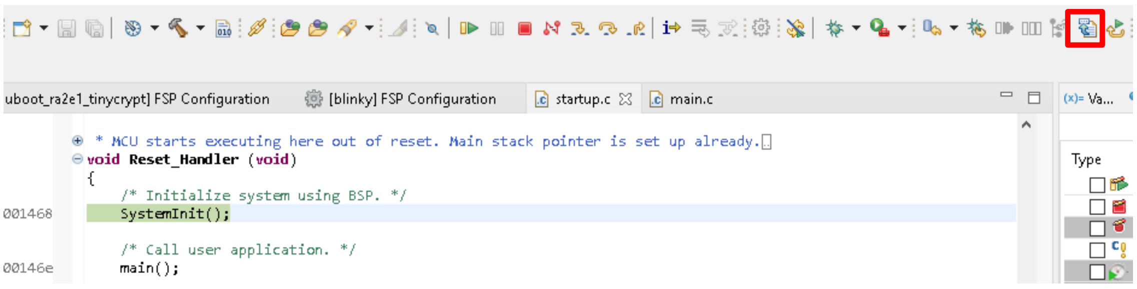
```
       * MCU starts executing here out of reset. Main stack pointer is set up already.
      void Reset_Handler (void)
      {
          /* Initialize system using BSP. */
00001468  SystemInit();

          /* Call user application. */
0000146e  main();
```

Figure 39. Start the Application Execution

5.3 Download the Primary Application

At this point, only the bootloader image has been downloaded to flash. Now download the Application Image using the **Load Ancillary File** button. On the top of the e^2 studio toolbar, click the icon, then browse to the signed image `\Debug\blinky_signed.bin` file.

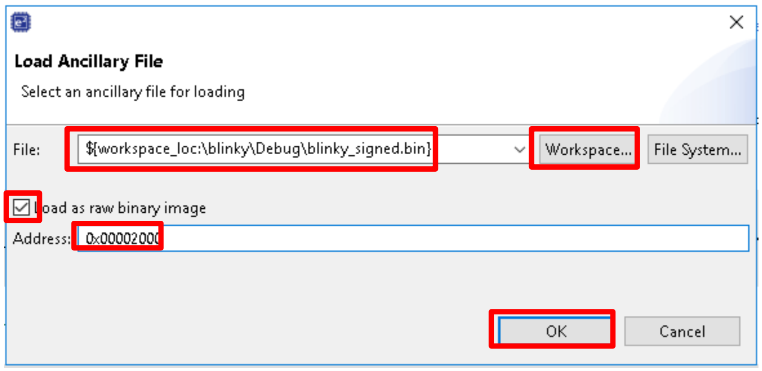

uboot_ra2e1_tinycrypt] FSP Configuration
[blinky] FSP Configuration
startup.c
main.c
(x)= Va...
Type

```
       * MCU starts executing here out of reset. Main stack pointer is set up already.
      void Reset_Handler (void)
      {
          /* Initialize system using BSP. */
001468    SystemInit();

          /* Call user application. */
00146e    main();
```

Figure 40. Load Ancillary File

Check the **Load as raw binary image** and set the address to **0x2000**. Press **OK** to download the image.


Load Ancillary File

Select an ancillary file for loading

File: ${workspace_loc:\blinky\Debug\blinky_signed.bin}

[✓] Load as raw binary image

Address: 0x00002000

Figure 41. Configure the Download Address of the Application Image

5.4 Booting the Primary Application

Click **Resume** to run the project.

The program should now be paused in main at the `hal_entry()` call in the bootloader.

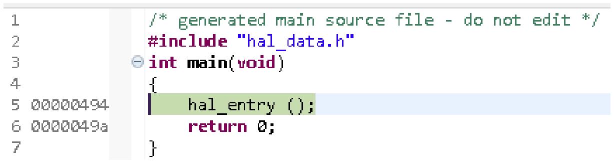
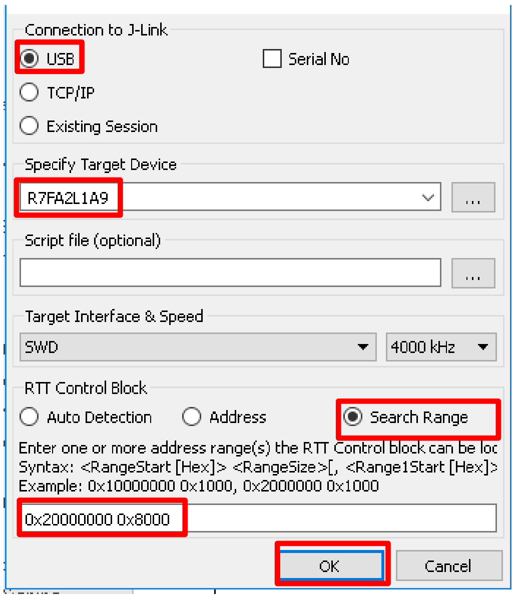

```
          /* generated main source file - do not edit */
          #include "hal_data.h"
          int main(void)
          {
00000494      hal_entry ();
0000049a      return 0;
          }
```

Figure 42. Start the Application Execution

Click to run again. The Red, Blue, and Green LEDs on the EK-RA2E1 should now be blinking while the blinky application is running.

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

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


Connection to J-Link

[✓ USB] Serial No

TCP/IP

Existing Session

Specify Target Device: R7FA2L1A9

Script file (optional):

Target Interface & Speed: SWD 4000 kHz

RTT Control Block

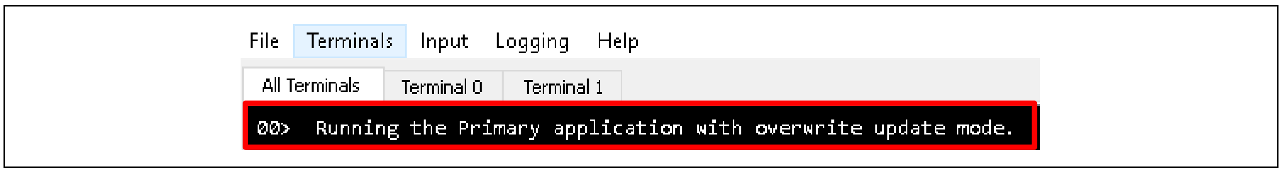
Auto Detection Address [✓ Search Range]

Enter one or more address range(s) the RTT Control block can be loc
Syntax: <RangeStart [Hex]> <RangeSize>[, <Range1Start [Hex]>
Example: 0x10000000 0x1000, 0x2000000 0x1000

0x20000000 0x8000

Figure 43. Configure the RTT Viewer

Click **OK** and observe the followinf output on the RTT Viewer. This output shows the Primary application is being executed and all three LEDs are blinking.



```
00>  Running the Primary application with overwrite update mode.
```

Figure 44. 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, example Primary application (blinky project) as well the example Secondary application (blinky_new, which this section will establish) are provided in the `RA2_secure_bootloader.zip`. The user can also follow section 7 to exercise these project 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 **blinky** project to the same workspace and rename the new project to **blinky_new**.

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

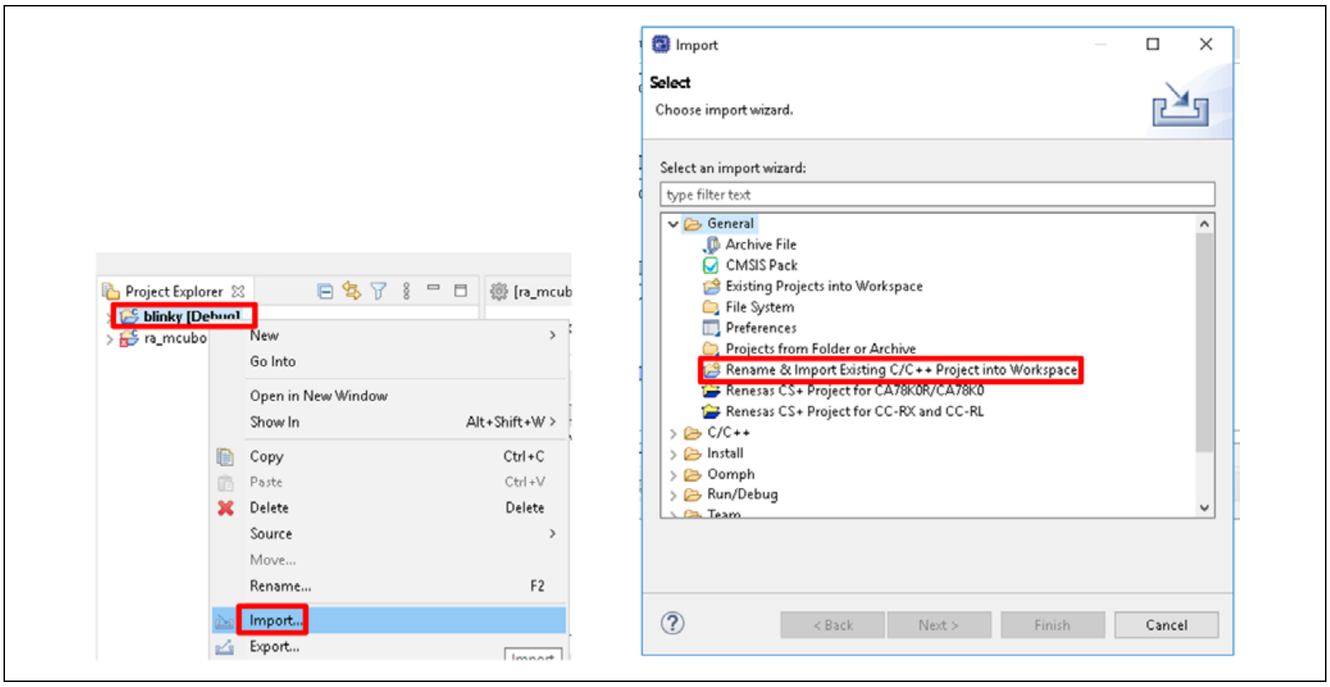


Figure 45. Select Rename and Import the Primary Application

Once the Import window opens, name the project and click **Browse** for **Select root directory** as shown in Figure 46.

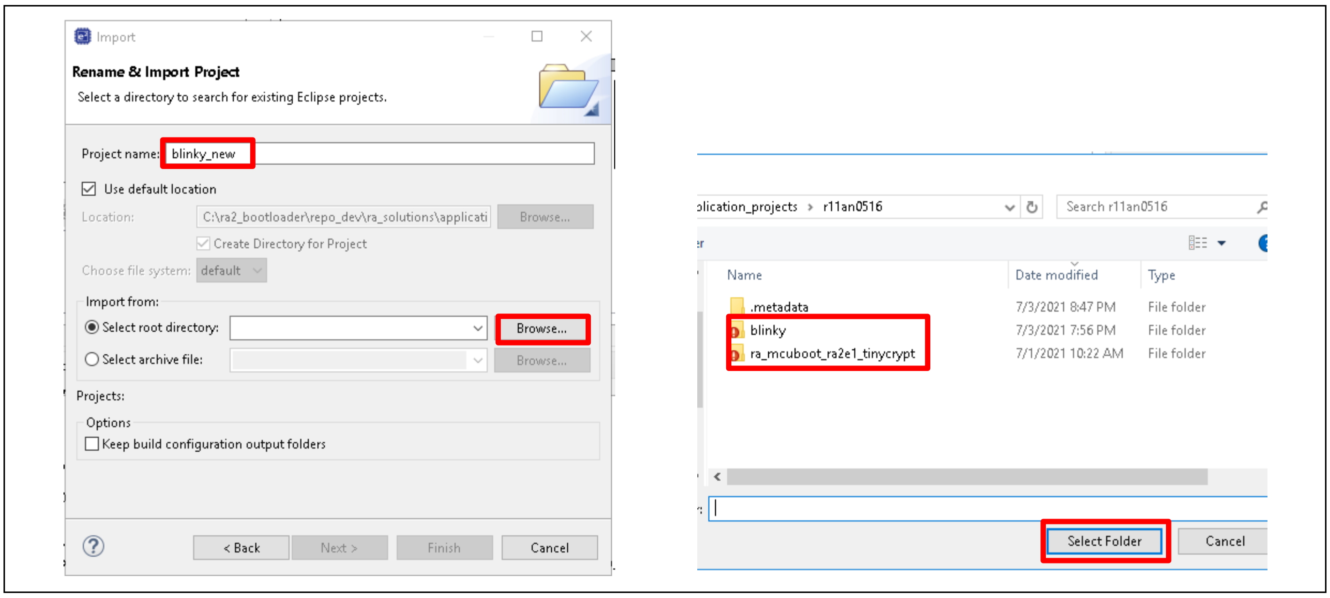

Import
Rename & Import Project
Select a directory to search for existing Eclipse projects.
Project name: blinky_new
☑ Use default location
Location: C:\ra2_bootloader\repo_dev\ra_solutions\applicati
Browse...
☑ Create Directory for Project
Choose file system: default
Import from:
◉ Select root directory:
Browse...
○ Select archive file:
Browse...
Projects:
Options
☐ Keep build configuration output folders
< Back
Next >
Finish
Cancel
plication_projects › r11an0516
Search r11an0516
Name
Date modified
Type
.metadata 7/3/2021 8:47 PM File folder
blinky 7/3/2021 7:56 PM File folder
ra_mcuboot_ra2e1_tinycrypt 7/1/2021 10:22 AM File folder
Select Folder
Cancel

Figure 46. Rename the Project blinky_new

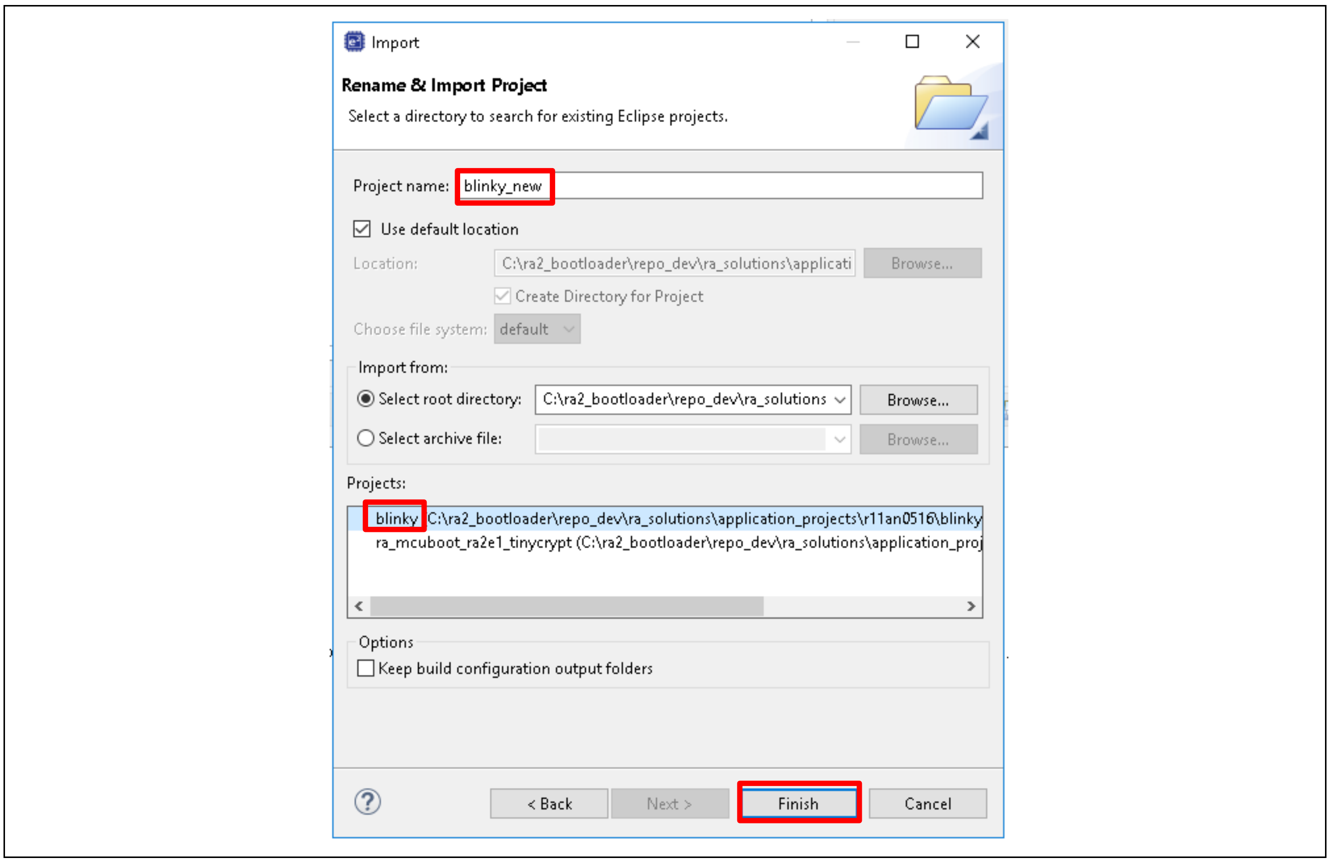

Import
Rename & Import Project
Select a directory to search for existing Eclipse projects.
Project name: blinky_new
☑ Use default location
Location: C:\ra2_bootloader\repo_dev\ra_solutions\applicati
Browse...
☑ Create Directory for Project
Choose file system: default
Import from:
◉ Select root directory: C:\ra2_bootloader\repo_dev\ra_solutions
Browse...
○ Select archive file:
Browse...
Projects:
blinky C:\ra2_bootloader\repo_dev\ra_solutions\application_projects\r11an0516\blinky
ra_mcuboot_ra2e1_tinycrypt (C:\ra2_bootloader\repo_dev\ra_solutions\application_proj
Options
☐ Keep build configuration output folders
< Back
Next >
Finish
Cancel

Figure 47. Import blinky Project as blinky_new

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

Update Existing Application to a New Application

The user can perform the following simple update to the existing blinky project to a new application:

- Open `\blinky_new\src\app_definitions.h` and change
  ```
  #define MENUSTATUS1       " Running the Primary application with overwrite update mode. \r\n"
  ```
  to
  ```
  #define MENUSTATUS1       " Running the Secondary application with overwrite update mode. \r\n"
  ```
- Open `\blinky_new\src\hal_entry.h` and change
  ```
  /* Update all board LEDs */
          for (uint32_t i = 0; i < leds.led_count; i++)
          {
              /* Get pin to toggle */
              uint32_t pin = leds.p_leds[i];

              /* Write to this pin */
              R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
          }
  ```
  to
  ```
          /* update the blue led */
          R_BSP_PinWrite(leds.p_leds[0], pin_level);
  ```

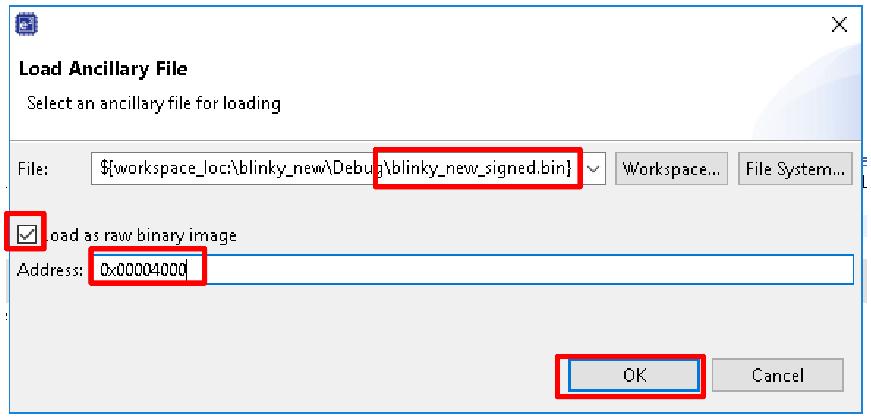
Note that since the new application is renamed and imported from the Primary application, the **Build Variable** configure in Figure 33, the **Post-build steps > Command(s)** in Figure 34 and the **Pre-build steps > Command(s)** in Figure 35 are already configured for blinky_new.

To create a brand new application without importing the previous application, follow section 4.2 to configure the application to use the bootloader and section 4.3 to sign the application image.

Click **Generate Project Content** and compile the blinky_new project.

6.2 Downloading and Booting the New Application

Assume the Primary application blinky is now up and running and the three LEDs are blinking. Click **Pause** and use the **Ancillary Download** file button load the compiled Secondary Application `blinky_new_signed.bin.` Select the new application image and set the download address to **0x4000**.


Load Ancillary File

Select an ancillary file for loading

File: ${workspace_loc:\blinky_new\Debug\blinky_new_signed.bin}

[✓] Load as raw binary image

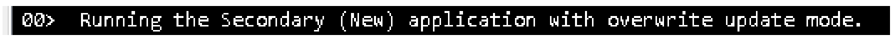
Address: 0x00004000

Figure 48. Download the Secondary Application Image

Note that for user-created customized applications, the download address needs to be adjusted by referencing the specific signing command generated in Figure 29.

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, the following new line will be printed, indicating the new image is loaded to the Primary slot and is booted.



```
00>  Running the Secondary (New) application with overwrite update mode.
```

Figure 49. RTT Viewer Output from the New Application

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. Import all three projects to a Workspace and follow below steps to compile the projects.

1. **Compile the RA2 Example Bootloader Project**
   Open the `Configuration.xml` file from project `ra_mcuboot_ra2e1_tinycrypt`. Click **Generate Project Content** and compile the project.
2. **Compile the Application Projects**
   Open the `Configuration.xml` file from example Primary application project blinky. Click **Generate Project Content** and compile the project.
   Open the `Configuration.xml` file from example Secondary project blinky_new. Click **Generate Project Content** and compile the project.
3. **Boot the Primary Application**

Follow section 5.1 to set up the EK-RA2E1 hardware connection and section 5.3 and 5.4 to boot the blinky project. Verify that all three LEDs are blinking and the RTT Viewer output shown in Figure 44 is observed.

4. **Boot the Secondary Application**
   Follow section 6.2 to download and boot the Secondary Application (blinky_new). Verify that the blue LED is blinking and the RTT Viewer output in Figure 49 is observed.

8. References

1. Renesas RA Family MCU Securing Data at Rest using Security MPU Application Project (R11AN0416)
2. Using MCUboot with RA Family MCUs 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.26.2021	-	First release document

Corporate Headquarters

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

Trademarks

ÿ

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/.