Secure Bootloader for RH850 Devices

Rev.3.00 01 Oct.01.21

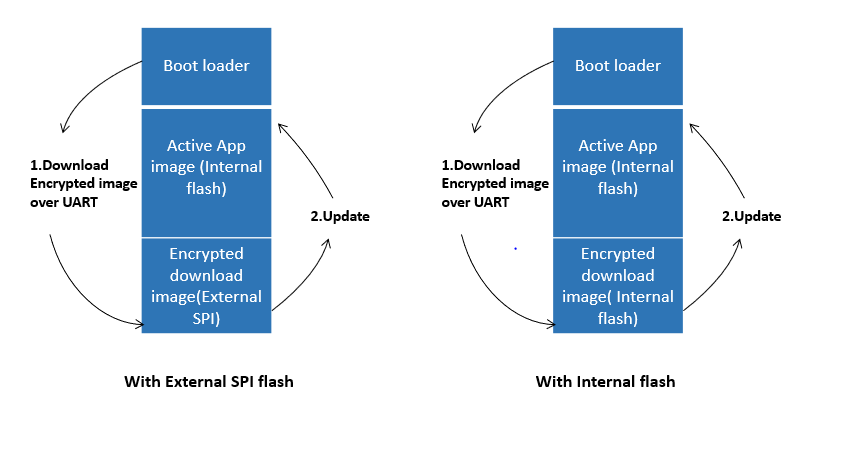
Introduction

In applications that calls for ensuring the integrity of the firmware running on the system, it is essential to take due security considerations in every aspect of the software design such that the root of trust is established at the boot up and continues so till the device is powered off. This application notes and its accompanying software implements a secure bootloader that is robust supporting different use cases that might be needed. The same can be used as such or improved up on based on usage specific requirements.

**RH850 Secure Bootloader Implementation**

This Secure Bootloader, apart from a secure validation of application image, provides features to download the next version of application image over UART, store it locally, validate it and update it only on successful validation. It also leverages the power of ICU-S(RH850\_F1KM\_S1) and ICU-M(RH850\_F1KM\_S4) hardware cryptographic engine to perform some of the process offloading the processor.

Some of the environments that this bootloader can be used are captured below



With the complete source code available, it is possible to customize the same based on the user requirements.

Target Device

RH850/F1K/F1KM

Contents

[1. Secure Bootloader Overview 4](#_Toc83984583)

[1.1 Functional description 4](#_Toc83984584)

[1.2 Secure Bootloader Concepts 5](#_Toc83984585)

[2. Required Environment 6](#_Toc83984586)

[2.1 System Configuration and Required Development Tools 6](#_Toc83984587)

[2.2 Tools Provided 6](#_Toc83984588)

[3. Using the Bootloader 7](#_Toc83984589)

[3.1 Project Organization 7](#_Toc83984590)

[3.2 Boot loader Configuration 9](#_Toc83984591)

[3.3 Cryptography scheme configuration 9](#_Toc83984592)

[3.4 Download location configuration 10](#_Toc83984593)

[3.5 Cryptography Platform configuration 11](#_Toc83984594)

[3.6 Flash Memory Partition 12](#_Toc83984595)

[3.7 Cryptography Setup 14](#_Toc83984596)

[4. Programming the Bootloader 18](#_Toc83984597)

[4.1 Hardware Security module ICUM Programming 18](#_Toc83984598)

[4.2 Built and debug 20](#_Toc83984599)

[4.3 Securing the Bootloader 26](#_Toc83984600)

[5. Firmware Update 30](#_Toc83984601)

[5.1 Application Image Preparation 30](#_Toc83984602)

[5.2 Image Signing 33](#_Toc83984603)

[5.3 Performing Update using UART Xmodem protocol 40](#_Toc83984604)

[5.4 Performing Update using CAN UDS protocol 47](#_Toc83984605)

[6. Notes 51](#_Toc83984606)

[7. ICUM Cryptography 52](#_Toc83984607)

[Revision History <revision history> 63](#_Toc83984608)

[General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products 1](#_Toc83984609)

[Notice 1](#_Toc83984610)

[Corporate Headquarters 2](#_Toc83984611)

[Contact information 2](#_Toc83984612)

[Trademarks 2](#_Toc83984613)

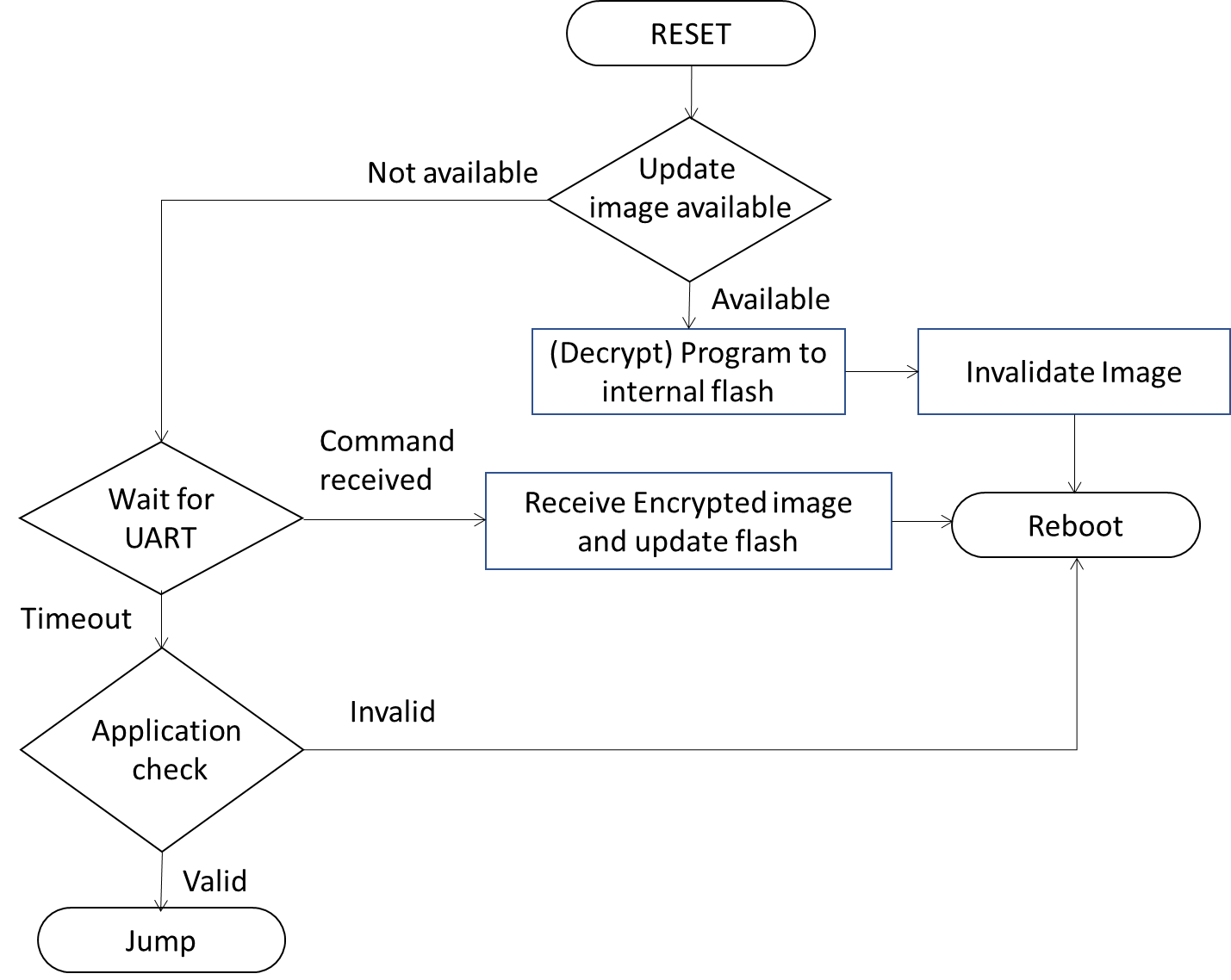
# Secure Bootloader Overview

## Functional description

Bootloader is typically a standalone component that is executed on boot of the MCU. It is responsible for setting up some basic environment for the application image to run. This approach helps in implementing Firmware Upgrade functionality - for example, the bootloader can see if any update image is available, if so program this to the active application area and launch it.

Secure Bootloader, in addition to that, can validate the authenticity of the application image before launching it. So, if the bootloader is ensured that it is not tampered, in effect, it establishes the chain of trust.

This application note implements a boot loader that can perform firmware upgrade securely by downloading the update image to external/internal flash and copying to the active internal flash area upon validation. The below diagram gives a high-level flow of the same.



A basic command line interface is provided in the bootloader that can be used to trigger the download within 5 seconds. XModem protocol is supported over UARTx for its simplicity. Powerful RSA based asymmetric cryptographic algorithm is used to sign the image before update and for validation. All these components are highly customizable.

## Secure Bootloader Concepts

The Secure Boot mechanism relies on asymmetric cryptography that uses public/private key pairs to verify the digital signature of all firmware and software before execution.

After Secure Boot is enabled and configured, only software or firmware signed with approved keys can execute. Conversely, software signed with blacklisted keys are disallowed from executing.

A picture containing diagram

Description automatically generated

Figure 3-1. Asymmetric Cryptography

The basic idea of digital signatures is to generate a pair of keys: A private key to be kept private and secured by the originator. A public key that can be distributed freely.

The mathematical correlation between this public/private key pair allows for checking the digital signature of a message for authenticity. To do the check, only the public key is necessary, and the message can be verified as having been signed by the private key without ever knowing the private key itself.

Remember, a message cannot be signed using the public key. Only the private key can sign the message properly. Basic mechanism digital signature technology is used to verify a message's integrity without compromising the details or contents of the private key.

One other feature of this public/private key pair is that it is impractical to calculate the private key from the contents of the public key. This feature allows for the distribution of the public key without compromising the private key.

As captured in Figure 3-1, rather than signing the entire image, it is less computationally intense to sign only the hash of the entire image, append it with the original message and send it. This “Signed Image” can be split at the receiver, once again hash calculated on the plain text message, and validated with the received signed hash using the public key.

While the above picture, implies plain text message being transmitted along with the signed hash, it is possible to add another level of security by encrypting the image using a symmetric key algorithm.

# Required Environment

## System Configuration and Required Development Tools

The Figure 2-1 and Table 2-1 shows the system configuration and required environment



Figure 2-1. System Configuration

All these components are highly customizable.

Table 2-1. Required Environment

|  |  |  |
| --- | --- | --- |
| Item | Target Device | Detail |
| Emulator | Emulator | Renesas E1/E20./E2 Emulator |
| Integrated Development Environment (version) | RH850/xx | Renesas GHS compiler (MULTI V-7.1.6) |
| Host Machine | Computer/Laptop | Windows 10 64-bit OS |
| Renesas Flash Programmer | RH850/xx | Windows 8/10 64-bit OS |
| PEAK/Vector CAN device | Computer/Laptop | Windows 8/10 64-bit OS |

## Tools Provided

Table 2-2 lists the set of tools/ utilities provided along with this application note.

Table 2-2. Tools provided

|  |  |  |
| --- | --- | --- |
| Item | Description | Application |
| SBL Manager.exe | Integrated Windows application | SBL Manager Tool is used for code signing, public key generation, firmware update via x modem protocol and CAN UDS protocol. |

# Using the Bootloader

## Project Organization

Table 3-1 lists the application notes files up on extraction.

Table 3-1. Project Structure

|  |  |
| --- | --- |
| Item | Description |
| Binaries | Example public, private keys with bootloader and test applications |
| Documents | Documentation for the Application Note |
| secure\_boot\_project/RH850\_F1K | F1K secure Bootloader project |
| secure\_boot\_project/RH850\_F1KM\_S1 | F1KM-S1 secure Boot loader project |
| secure\_boot\_project/RH850\_F1KM\_S4 | F1KM-S4 secure Boot loader project |
| application\_project/RH850\_F1K | F1K Application project |
| application\_project/RH850\_F1KM\_S1 | F1KM-S1 Application project |
| application\_project/RH850\_F1KM\_S4 | F1KM-S4 Application project |
| Tools/ SBL Manager.exe | Secure Bootloader Code signing/Update application |

Secure boot-loader project and application project will be given in separate folder as mentioned in above table like secure boot project and application project. The secure boot loader consists of the following folder when opened as shown in below figure

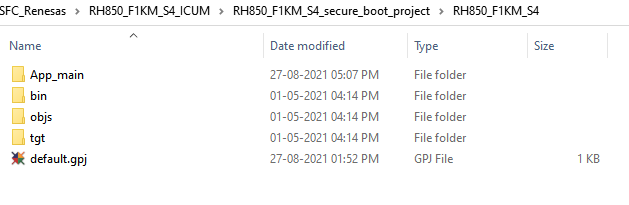


Figure 3-1: Project structure

App\_main is the main folder which consist of the sub folders peripheral, device and secure boot. Peripheral folder consists of following sub folders like clock, UART, CSI, FCL, FDL and so on. Device folder consist of the linker file, ISR file, start-up file and so on. Secure-boot and crypto folder consist of files for secure-boot process.

Open the project by double clicking the default.gpj. Ensure the green Hills IDE is installed and licensed. Once the project is open it looks like the below figure

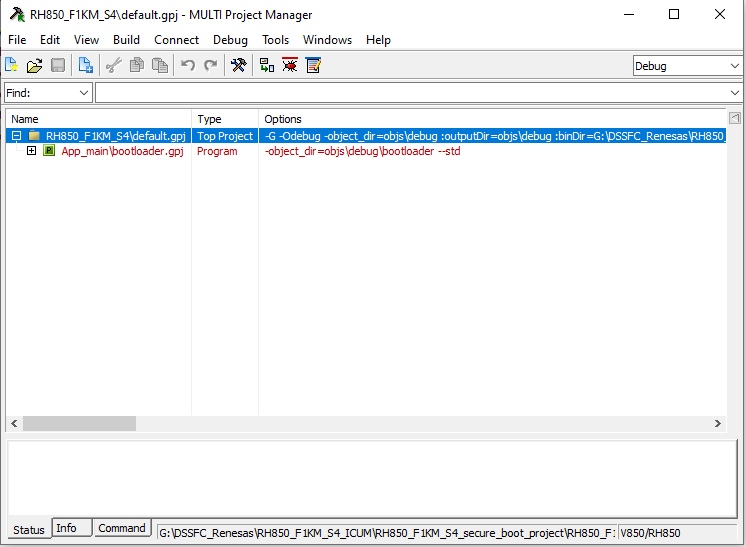


Figure 3-2: Project in Green Hills IDE

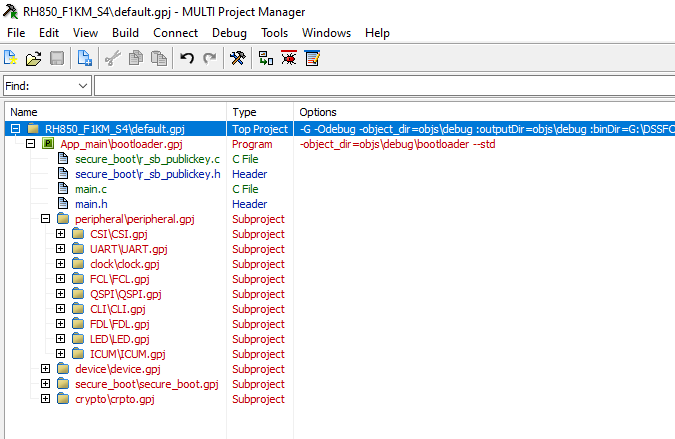


Figure 3-3: Project Folders

## Boot loader Configuration

The reference bootloader is highly configurable to suit needs of the customer specific use cases. The following pre-processor Macros can be used to modify the settings in the r\_sb\_boot\_config.h file.

The secure bootloader offers following settings to select the underlying MCU (for UART setup in common source code), enabling the secure boot feature, selection of security algorithms and the target update image storage type (Internal flash – AB Mode or with external SPI flash)

## Cryptography scheme configuration

Our secure-boot application provides secure-boot with and without encryption to upload the updated application image. The Firmware file to be updated will have a header of size 1Kb, followed by the application image encrypted and non-encrypted. Application image can be updated with or without any encryption but with RSA sign verification, this setting can be configured with the below macros.

Table 3-2. Cryptography Scheme configuration

|  |  |
| --- | --- |
| Macros | Description |
| R\_SECBOOT\_WITHOUT\_SECURITY | No Firmware encryption. Firmware image integrity validated using SHA256 on clear text Firmware (Hash stored in the image header) |
| R\_SECBOOT\_RSA\_WITHOUT\_ENCRYPT\_SHA256 | No firmware encryption, only Authentication and Integrity. Firmware Image integrity validated using SHA256 on clear text Firmware. Authenticating Firmware Metadata using the SHA256 hash signed with RSA. |
| R\_SECBOOT\_RSA\_WITH\_AES128\_CBC\_SHA256 | Authentication, Integrity, Confidentiality are ensured. Authenticate firmware header using SHA256 Hash signed with RSA. Decrypt Firmware Image with AES128-CBC. Firmware image integrity validated using SHA256 Hash stored in the authenticated Header). |
| R\_SECBOOT\_RSA\_WITH\_AES128\_ECB\_SHA256 | Authentication, Integrity, Confidentiality are ensured. Authenticate firmware header using SHA256 Hash signed with RSA. Decrypt Firmware Image with AES128-ECB. Firmware image integrity validated using SHA256 Hash stored in the authenticated Header). |
| R\_SECBOOT\_AES128\_CMAC\_WITH\_CBC\_SHA256 | Authentication, Integrity, Confidentiality are ensured. Authenticate firmware metadata using AES128 tag. Decrypt Firmware image with AES128-CBC. Firmware image integrity check done using CMAC. |
| R\_SECBOOT\_AES128\_CMAC\_WITH\_ECB\_SHA256 | Authentication, Integrity, Confidentiality are ensured. Authenticate firmware metadata using AES128 tag. Decrypt Firmware image with AES128-EBC. Firmware image integrity check done using CMAC. |

These macros should be set to the pre-processor R\_SECBOOT\_CRYPTO\_SCHEME in the configuration file.

One of the above security algorithms can be selected by configuring the R\_SECBOOT\_CRYPTO\_SCHEME macro.

#define R\_SECBOOT\_CRYPTO\_SCHEME R\_SECBOOT\_AES128\_CMAC\_WITH\_CBC\_SHA256 /\*! < Selected Crypto Scheme \*/

In the below figure it is configured as R\_SECBOOT\_AES128\_CMAC\_WITH\_CBC\_SHA256

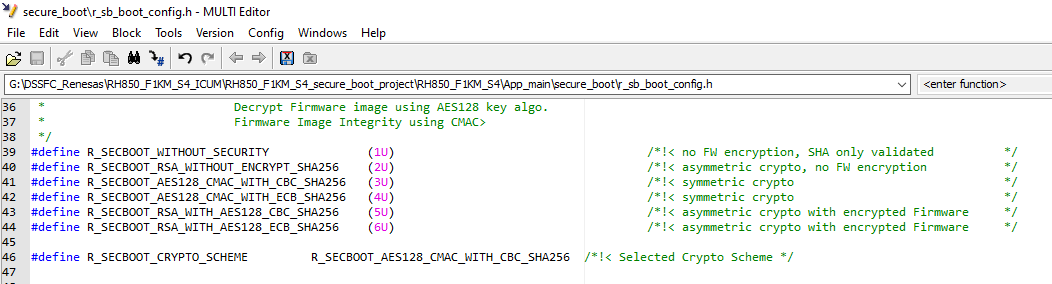


Figure 3-4: Cryptography Scheme Configuration

## Download location configuration

The application image can be downloaded in the internal flash area of the MCU as well external flash which is connected to the MCU. These settings can be configured from the following configuration

R\_DOWNLOAD\_IMAGE\_TO\_EXTERNAL\_FLASH 1

R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH 2

These settings should be set to the R\_DOWNLOAD\_IMAGE\_STORAGE\_TYPE macro.

Table 3-3: Download location configuration

|  |  |
| --- | --- |
| Macro | Description |
| R\_DOWNLOAD\_IMAGE\_TO\_EXTERNAL\_SPI\_FLASH | External SPI FLASH enabled and update image will store in the External flash. |
| R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH | There is no External SPI FLASH. The update image will store in the internal flash Image B area. |

Update image storage type can be selected by configuring the R\_DOWNLOAD\_IMAGE\_STORAGE\_TYPE macro.

#define R\_DOWNLOAD\_IMAGE\_STORAGE\_TYPE R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH /\*< selection of Update image storage type \*/

In the below figure it is configured as R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH

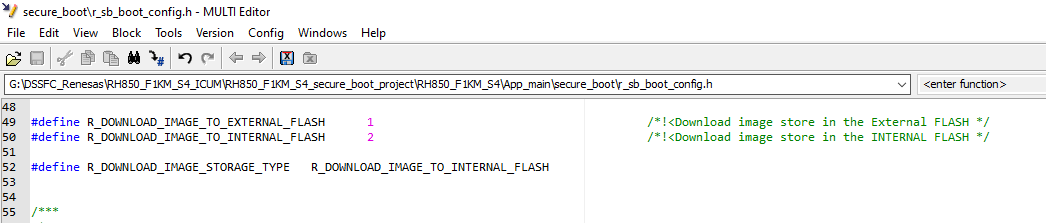


Figure 3-5: Download Location Configuration

## Cryptography Platform configuration

The application image can be decrypt-ed in the board with the help of mbedTLS or ICUM hardware or ICUS hardware. This platform configuration depends upon the MCU in which we are going to do the secure boot configuration

Table 3-4: Cryptography Platform configuration

|  |  |
| --- | --- |
| MCU | Cryptography Platform |
| R\_F1K | mbedTLS stack |
| R\_F1KM\_S1 | ICUS, mbedTLS, ICUS\_with\_DMA |
| R\_F1KM\_S4 | ICUM |

R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE macro in the configuration file is used to set the cryptography platform. As R\_F1K and R\_F1KM\_S4 supports only one cryptography platform this configuration pre-processor is not available in these two boards.

Table 3-4: Cryptography configuration in ICUS

|  |  |
| --- | --- |
| Macro | Description |
| R\_TYPE\_SOFTWARE | Crypto algorithm implementation by using mbedtls stack |
| R\_TYPE\_HARDWARE\_ICUS | Crypto algorithm implementation by using ICU-S with CPU based transfer. |
| R\_TYPE\_HARDWARE\_ICUS\_WITH\_DMA | Crypto algorithm implementation by using Hardware ICU-S plus DMA |

**NOTE:** This pre-processor configuration is available only in R\_F1KM\_S1 series boards

The below figure shows the available configuration for R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE in R\_F1KM\_S1 board in the project configuration file.

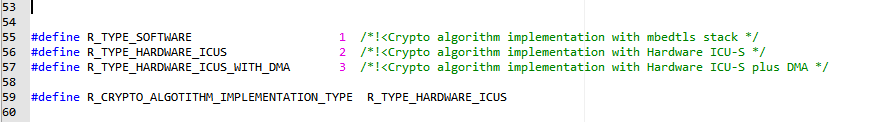


Figure 3-6: Cryptography Implementation algorithm configuration

## Flash Memory Partition

* Boot\_loader Image Address

RH850/F1Kxx series consist of internal flash of their own size. From the starting address of the flash first 128 byte is for secure-boot application code and this area should not come into configuration picture. For example, RH850/F1KM-S4 internal flash is 4MB. In that 4MB last 1MB is allocated for ICUM hardware remaining 3MB is used for the secure-boot process. This address and block number is mentioned in the r\_sb\_boot\_config.h as shown in below figure

R\_SB\_APPLICATION\_ACTIVE\_IMAGE\_ADDR 0x40000

R\_SB\_APPLICATION\_ACTIVE\_IMAGE\_BLOCK 14

**NOTE:** This Macro should not be modified

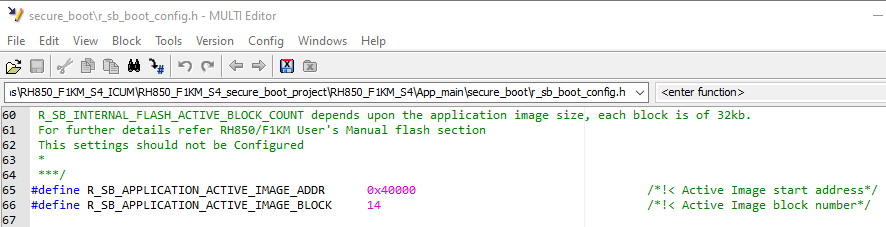


Figure 3-7: Active Image Address

* Application Image Download address

The new image can be download in external or internal flash according to the configuration in the section **3.4 Download location configuration**. If the download locationis set to R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH then the address should be configured after 128KB from the starting address of flash memory. The download address with its respective block number should be mentioned in the configuration file

If download locationis set to R\_DOWNLOAD\_IMAGE\_TO\_EXTERNAL\_FLASH, the download address can be configured from the starting or where ever.

Apart from the download address we need to configure the swap address. This address should be chosen from the internal flash area irrespective of the **Download location configuration**. This area represents the place where decrypted image of downloaded application will store.

**NOTE:** Kindly ensure the download and swap address is not same if Download location configuration is R\_DOWNLOAD\_IMAGE\_TO\_INTERNAL\_FLASH. It will be ok if the difference between download and swap address is 32kb.

Example:

R\_SB\_INTERNAL\_APP\_IMG\_DWLD\_ADDR 0x68000

R\_SB\_FLASH\_SWAP\_ADDR 0x60000

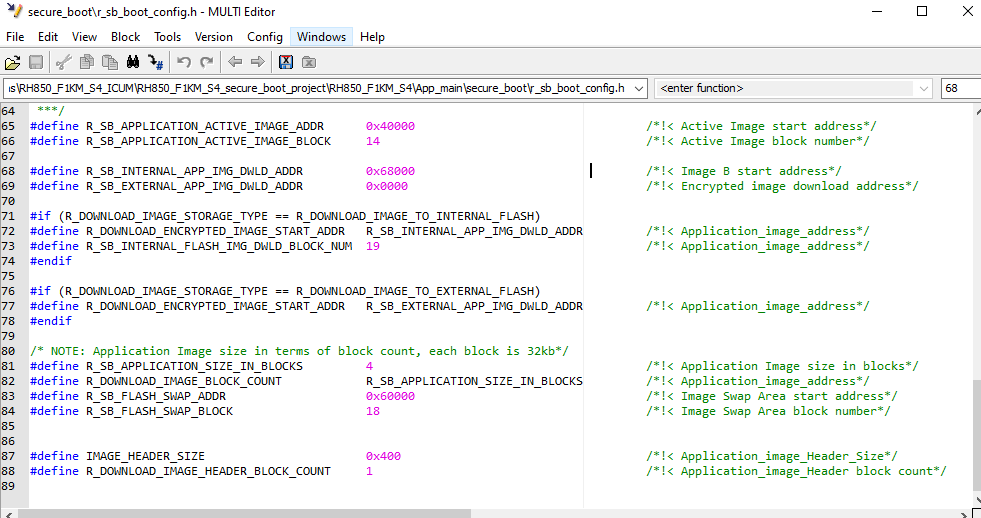


Figure 3-8: Download image Address Configuration

## Cryptography Setup

The bootloader supports RSA algorithm for signing and validating the application image. The below steps detail out the process of generating the public and private key pair and using them in the bootloader. Renesas Secure bootloader windows application helps to generate public and private key for signing the application image. Follow the below steps

**Step 1:** Open Renesas SBL manager windows application

**Step 2:** Under the app signer tab click on the key pair button at the right end.

**Step 3:** For the first time after installation of application this step is skipped, after step 2 step 4 is executed. From next cycle it will ask for the password, provide the password as **“123456”**

**Step 4:** It will ask for the location where to store the public and private key file. Provide the path.

**Step 5:** Once path is selected, the key pair files will be generated in the location automatically. The user is notified with a message **“key pair generated successfully”**.

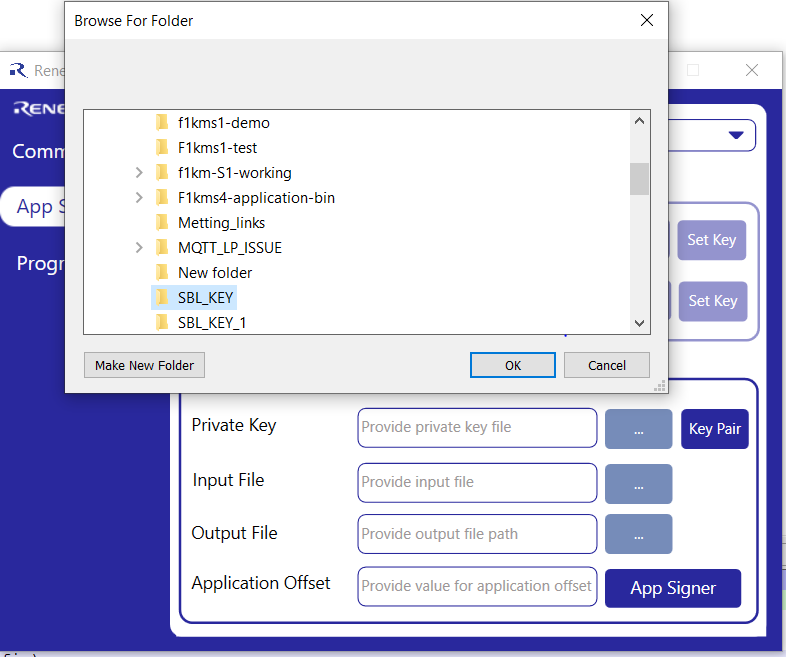


Figure 3-9. Key Generation

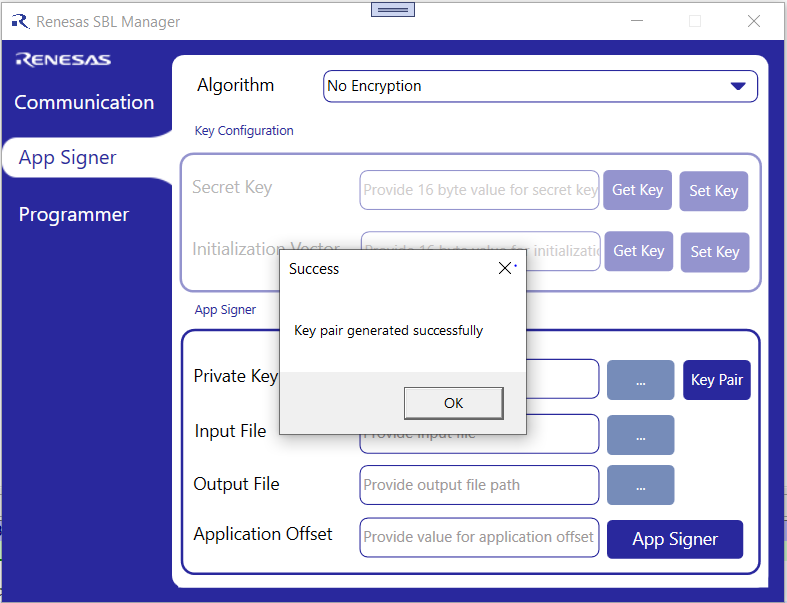


Figure 3-10. Key Generation Status

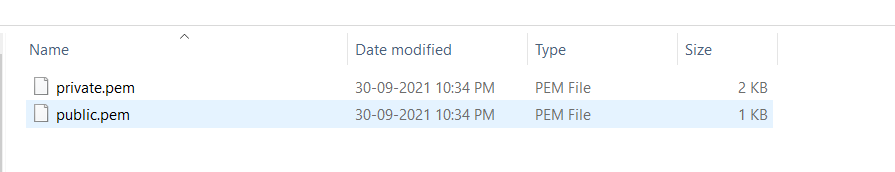


Figure 3-11. Generated Key in the folder

The generated key is used for signing the application image during encryption and decryption process. Private key is used for signing application image during encrypting the application image in Renesas SBL manager windows application, whereas public key is stored in all hardware for verifying the sign during decryption. To load the public key to hardware, follow the below steps

**Step 1:** Connect the hardware through UART with the PC where the Renesas SBL manager application is installed.

**Step 2:** Ensure the hardware is programmed with secure boot loader firmware and key pairs are generated in the connected PC with Renesas SBL application as mentioned in the previous section.

**Step 3:** Power on the hardware. Select programmer tab in the Renesas SBL manager. In communication tab select thecserial communication port and click connect.

**Step 4:** To load public key select load key button and select **public.pem** file which was created in the key pair generation. The key should be loaded within 60 seconds after power on the hardware.

**Step 5:** After successful loading of public key to hardware with Renesas SBL manager windows application it will notify with a pop-up message **“public key loaded successfully”**. In case of failure, it will prompt with the error message.

**Step 6:** Once the public key is loaded it will be available in board and need not be updated for every application image update. The availability of public key in the hardware can be verified through get public key status in the programming tab of the SBL manager application

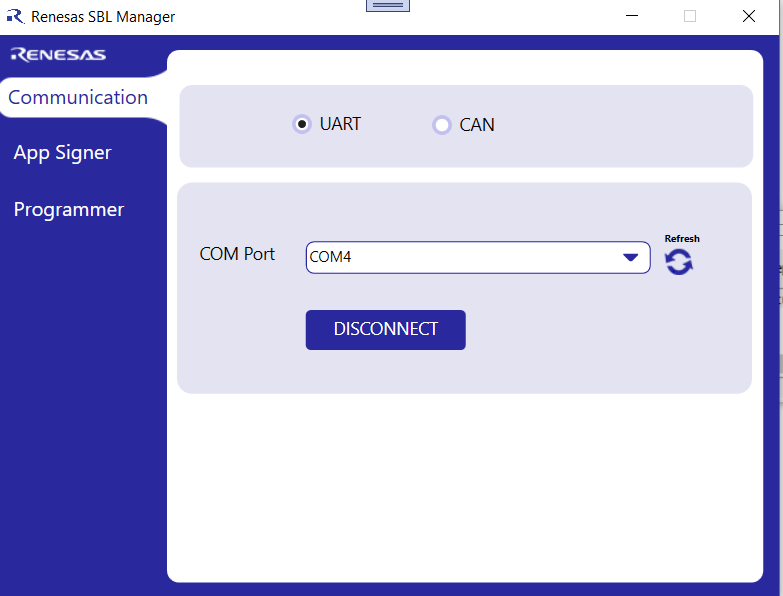


Figure 3-12. Renesas SBL application serial communication selection

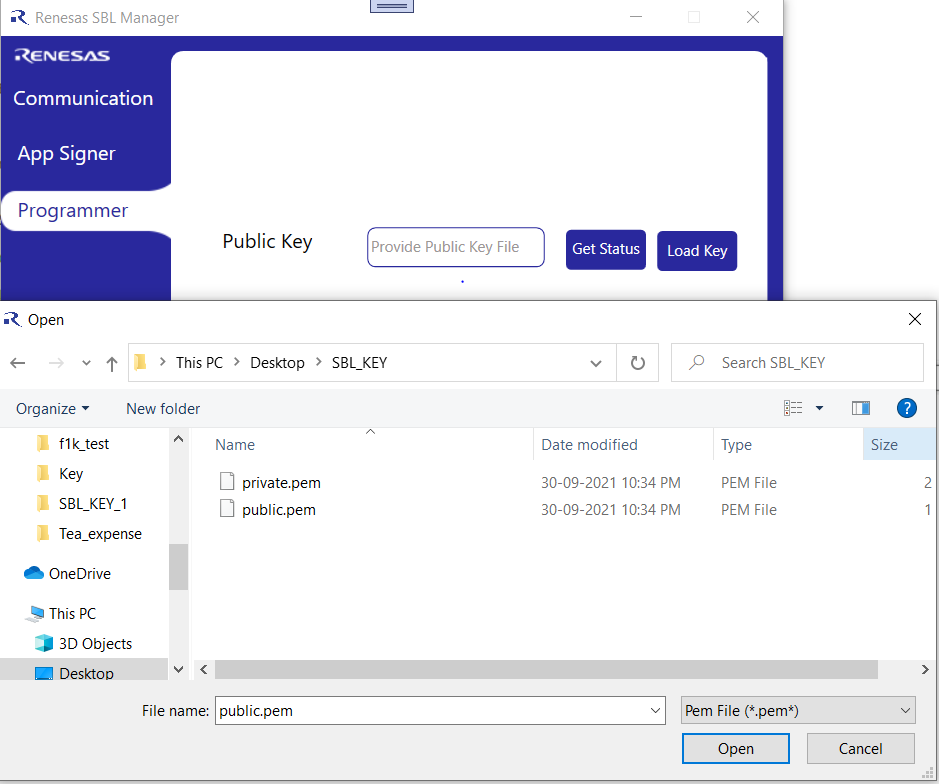


Figure 3-13. Loading Public key from Renesas SBL manager

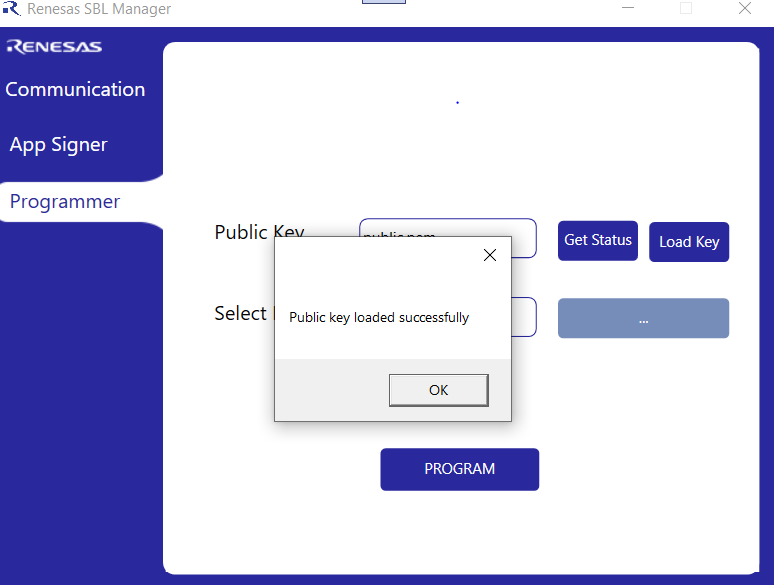


Figure 3-14: Public Key load status

# Programming the Bootloader

The bootloader generated above can be programmed on to the target directly using the GHS IDE.

## Hardware Security module ICUM Programming

The code for ICUM need to be updated, after enabling the ICUM hardware in Renesas Flash programmer. Open ICUM linker file in ICUM project folder and verify the code flash start address is same as OPBT 10. Open App\_main->device->dr7f701649i\_icum.ld file to verify the same

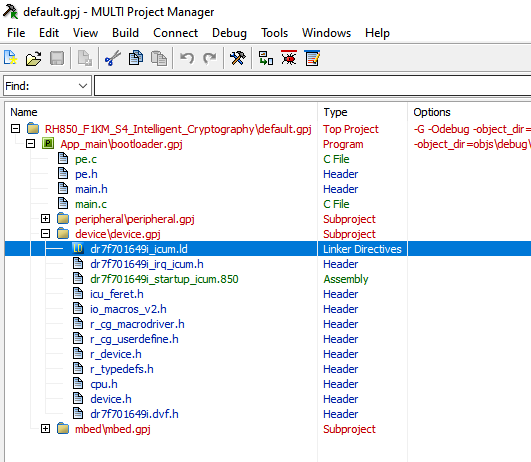


Figure 4-1: ICUM Linker file location

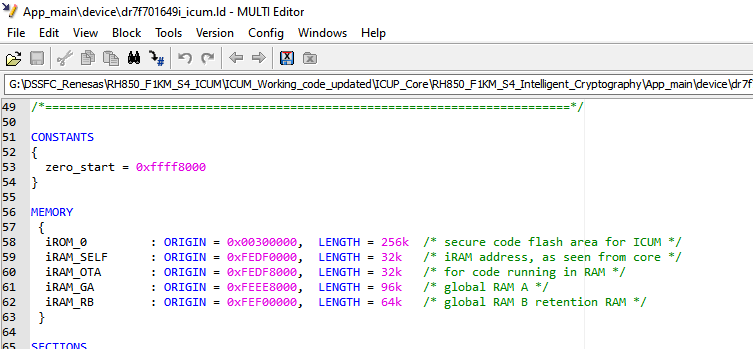


Figure 4-2: ICUM Linker ROM address verification

To flash ICUM code to MCU follow section **4.2** **built and debug** procedure. The device file should be selected from the device folder of ICUM project.

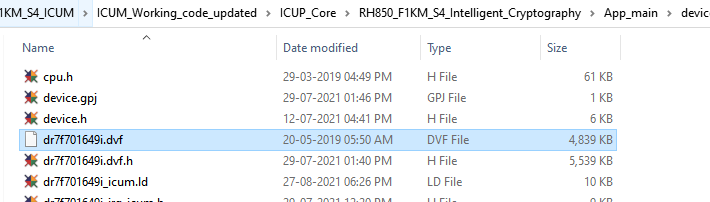


Figure 4-3: ICUM debug configuration file selection

During debug connection it will ask for the core where to flash the code. Select core1 from the dialogue box and click on OK. Rest of the things are same as mentioned in the section **4.2 built and debug**

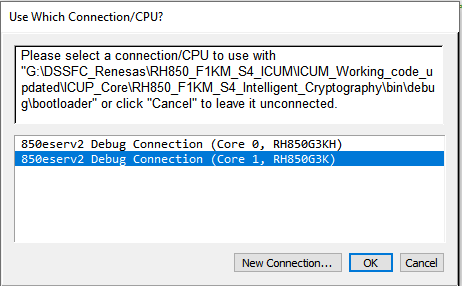


Figure 4-4: ICUM core selection

**NOTE:** For ICUS hardware there is no need to flash any code.

## Built and debug

After successful configuration save the r\_sb\_boot\_config.h configuration file. Follow the below steps to build and flash the code to the board.

**Step 1:** Move to multi project manager screen and click built icon to build the application

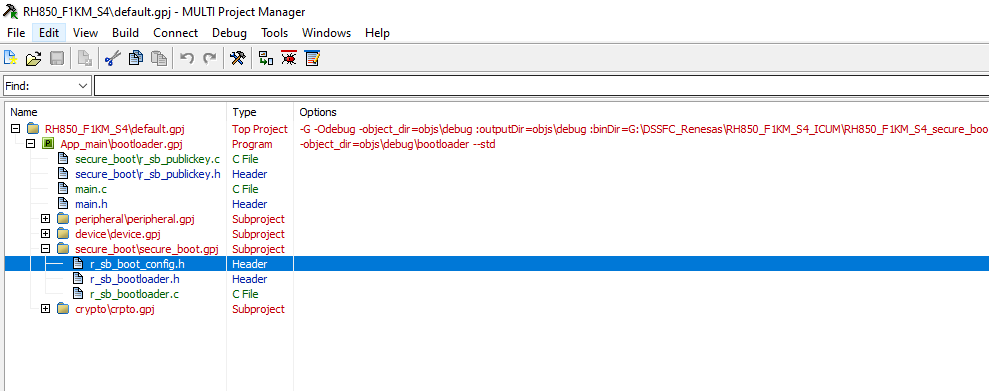


Figure 4-5: Multi project manager screen

**Step 2:** Once built is finished it will be displayed with the status success. Then your boot-loader application is ready.

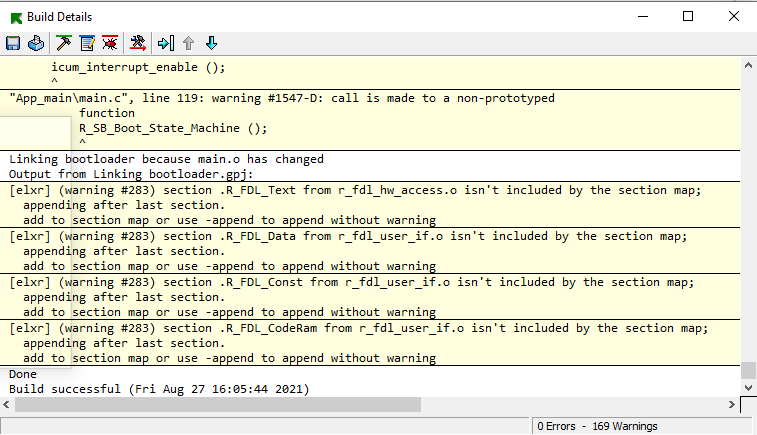


Figure 4-6: Secure-boot built result

**Step 3:** After successful compilation of the application, click the debug icon on the built result or in the Multi project manager. The IDE will enter the debug mode, click download icon in the screen

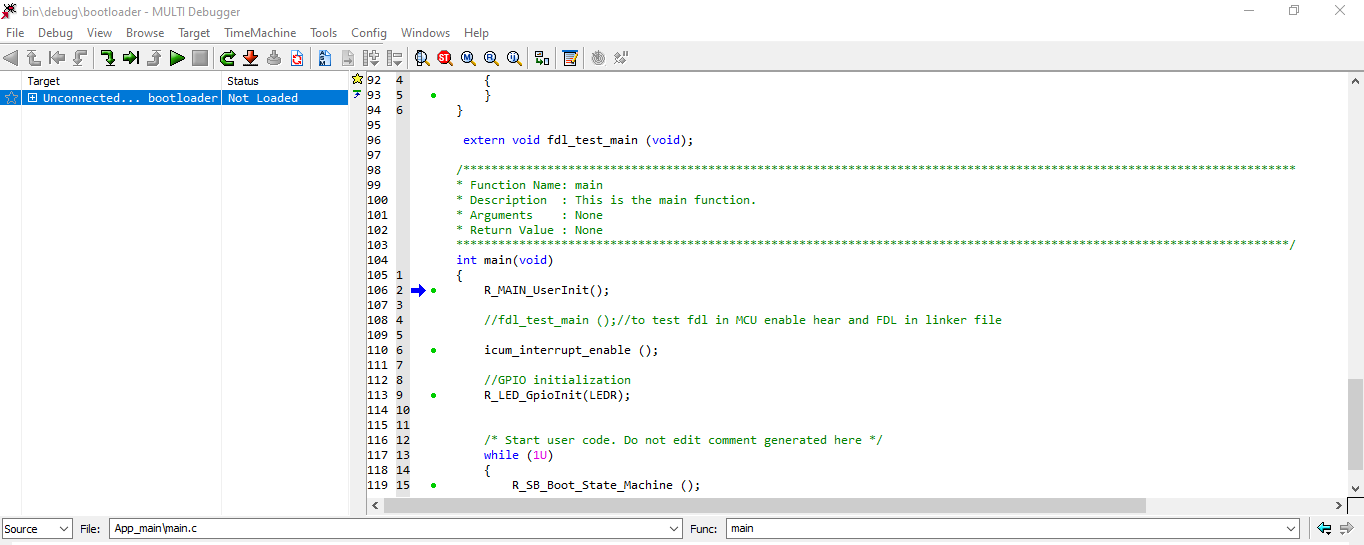


Figure 4-7: Multi IDE debug mode screen

**Step 4:** Once download icon is selected it will prompt a dialogue box for debug configuration

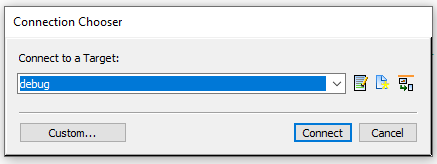


Figure 4-8: Debugger selection dialogue box

**Step 5:** In this connection chooser click on edit icon which in turn opens one dialogue box

Where we need to configure the device file, debugger, power supply for the debugger, clock settings and so on. Set the configuration same as in the below figures

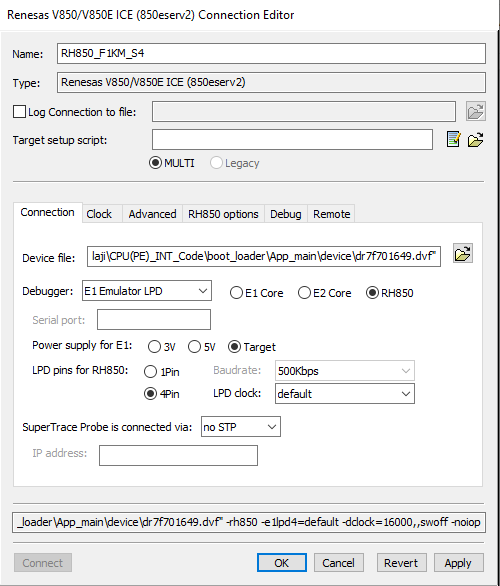


Figure 4-9: Debugger connection settings

The device file should be selected from the project location. App\_main->device->dr7f701649.dvf

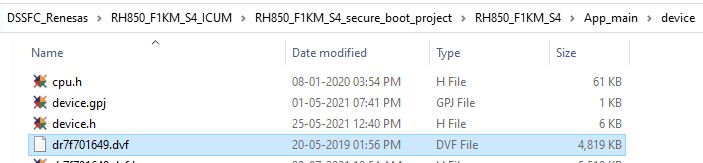


Figure 4-10: Device file location

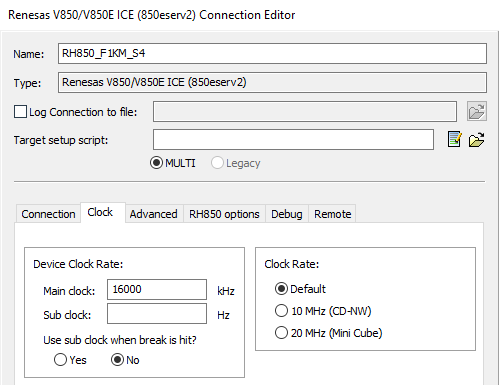


Figure 4-11: Debugger clock configuration settings

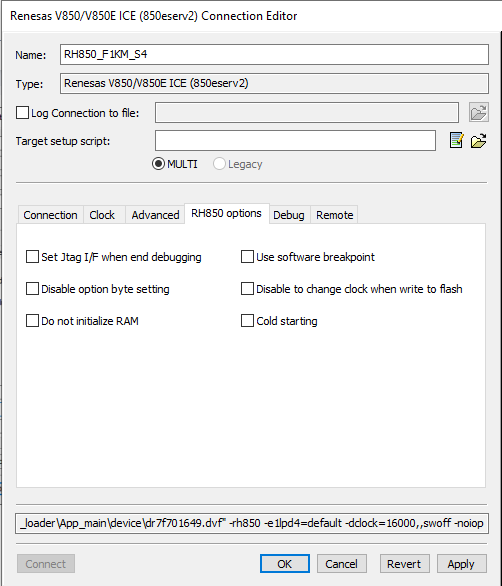
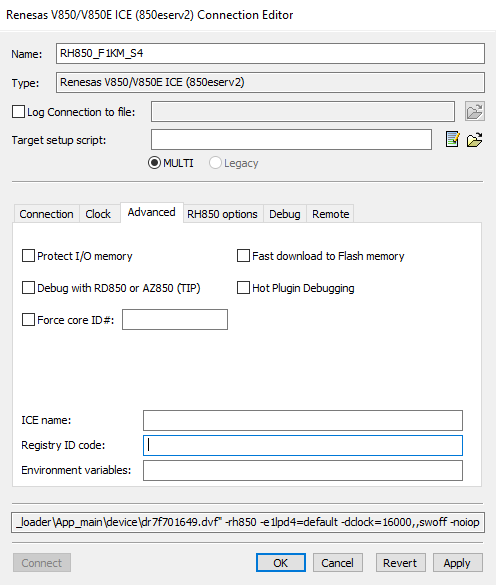


Figure 4-12: Advanced Debugger configuration settings

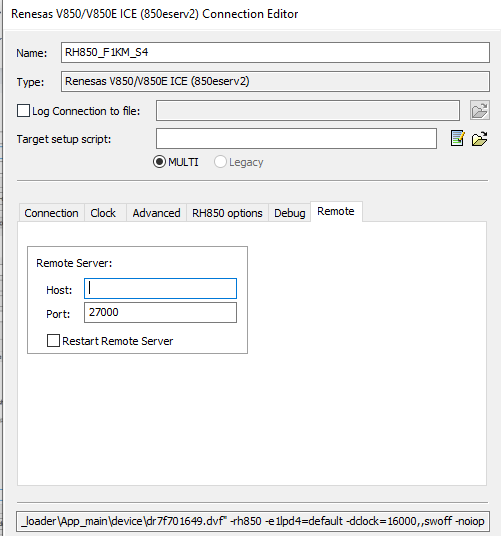
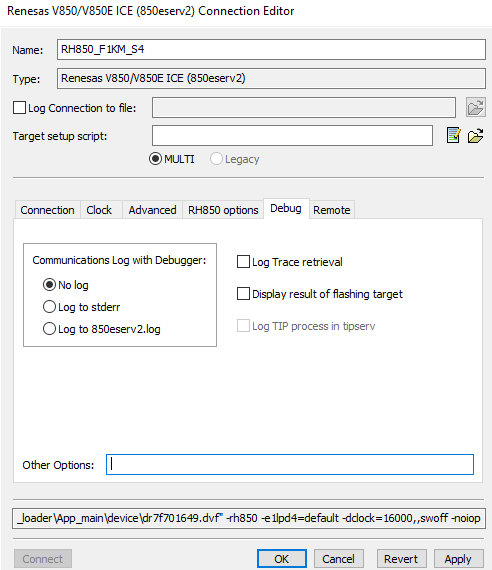


Figure 4-13: Other Debugger configuration settings

After all settings are done click apply and okay, now the debugger configuration is made successfully and ready to debug. Endure the board is power on and connected with debugger properly. In-case of external flash selection ensure the power and connection for that to

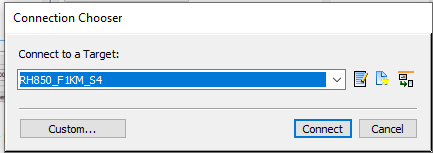


Figure 4-14: Debugger configuration screen

**Step 6:** Once connect is choose in debug configuration screen the code will be flashed into the MCU after successful programming of the code it will wait in the start position as shown in the below figure

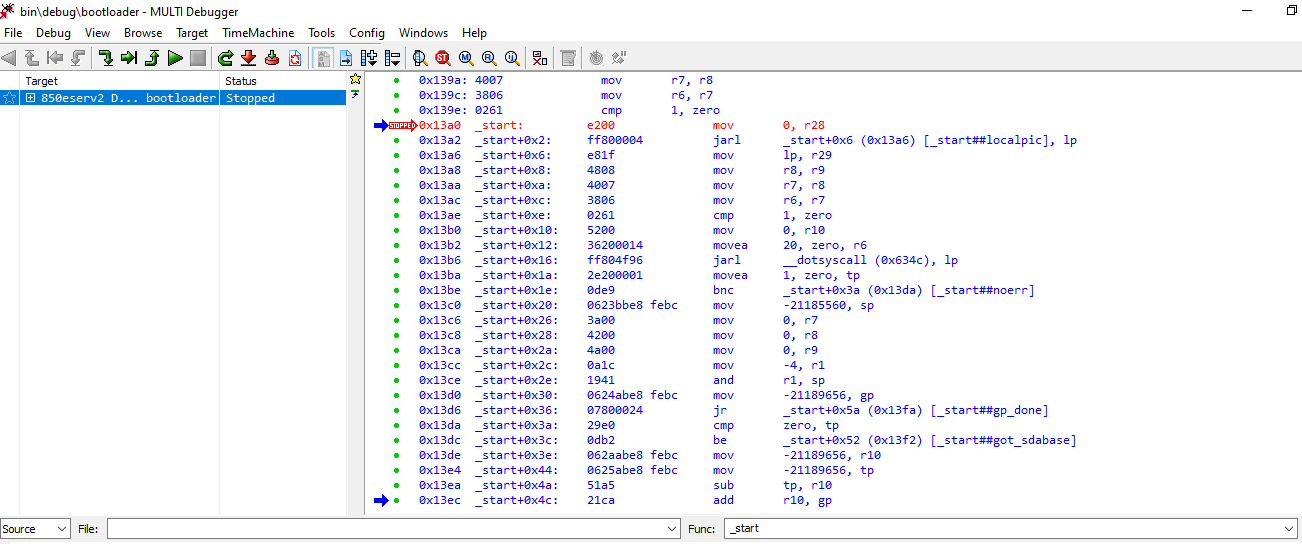


Figure 4.15: Code flash success screen

**Step 7:** click on reset icon right to the download icon, and then click go on selected items icon or press F5 simply to run the application. At the same time open the serial port application like tear term or putty and connect the MCU debug UART(UART-5) with TTL UART and connect to the PC. Open the serial port and set the baud-rate at 9600 baud-rate. The below debug print will appear on the serial port

**NOTE**: This debug print “**FLASH access received from ICUM**” won’t be available in RH850F1K-S1 as there is not ICUM hardware available. Only RH850F1K-S4 we are able to see those debug print

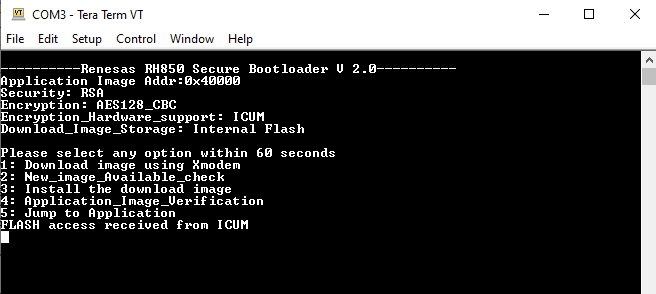


Figure 4-16: Debug print screen

## Securing the Bootloader

**Note:** Please note that this is an irreversible process and once done it is impossible to change the public key and the application image signed only with the corresponding private key will be accepted for updates. So this must be carried out once all the functionality testing of the bootloader are done and finalized.

Now that the bootloader is programmed, as the most important part of the step, it is important to lock the flash sector holding the bootloader from further writes and erase. This can be done as follows:

* First open the Renesas Flash Programmer and set the operation settings as follows.

Graphical user interface, text, application, email

Description automatically generated

Figure 4-14. Operation Settings

* Then enable the OTP configuration by selecting the combo-box option as Set.

Graphical user interface, application

Description automatically generated

Figure 4-15. Enable OTP

* Then protect the necessary sectors to be locked from further updates by selecting the corresponding check boxes. It is recommended to lock out the complete Secure Bootloader area – typically till address 0x1FFFF in the reference implementation.

Graphical user interface, application, table

Description automatically generated

Figure 4-16. Set OTP Regions

* Up on programming, the sectors will be locked out and secure bootloader effectively enabled.

As an added protection mechanism, it is recommended to lock out debugger access when deploying the devices in the field.

As an added protection mechanism, it is recommended to lock out debugger access when deploying the devices in the field.

# Firmware Update

## Application Image Preparation

An example project to blink the LED’s is given along with this application note. Users can use their own project instead of that. The application image is designed to run after 256KB, i.e from 0x40000 address as mentioned in the above section. Kindly review the linker script of the example project for reference.

Up on compiling this project, the output will be available as LED\_Blink\_App.bin file

User can create their own image that can be new one or fixes the bug in the previous code. For example, the code with UART print for continuous interval in debug UART is provided in the application project folder.

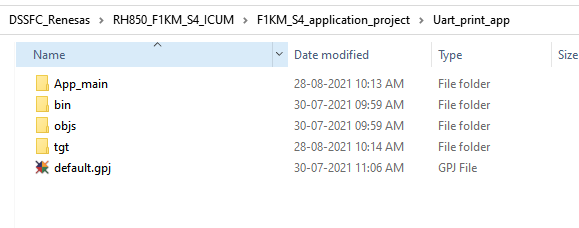


Figure 5-1: Application Folder

Open the application folder and verify the code flash area address of the application folder is equal to the sum of R\_SB\_APPLICATION\_ACTIVE\_IMAGE\_ADDR in the **3.6 Flash Memory Partition** and 1KB. The 1KB is added to the active image region as the boot-loader have header of 1KB which consist of the sign, hash and other information of the application image.

Example:

R\_SB\_APPLICATION\_ACTIVE\_IMAGE\_ADDR is 0x40000, then the code flash address should be 0x40400(R\_SB\_APPLICATION\_ACTIVE\_IMAGE\_ADDR + 1KB). Likewise, the interrupt vector starting address should change accordingly. The below image shows the code flash address and interrupt vector address for RH850\_F1K-S4 board

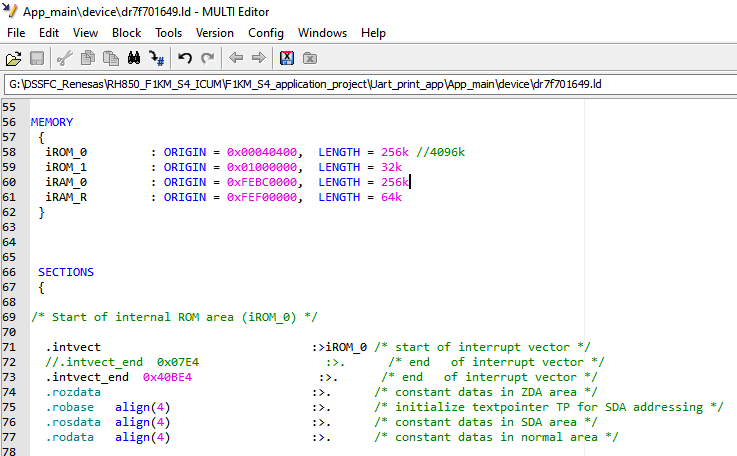


Figure 5-2: Application image code flash and interrupt vector address settings

Once the code flash address and the interrupt vector address are set. Right click on the project name on multi project manager and select set built options

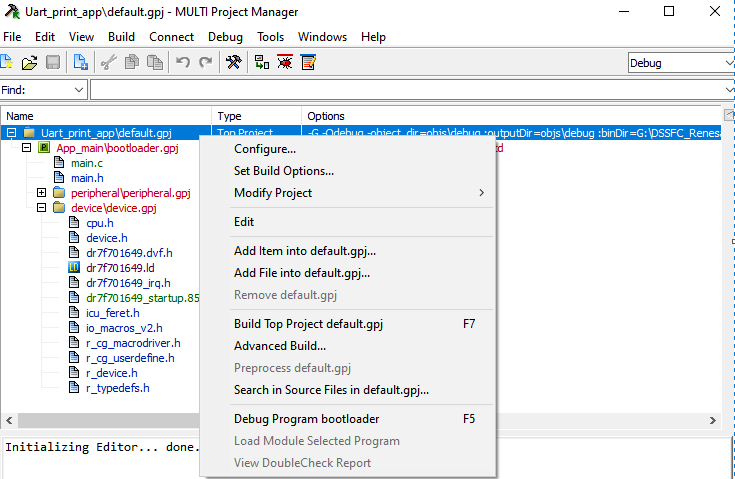


Figure 5-3: Set built option setting

In the build option screen set memory image file in generate additional input file section under linker option as shown in the below figure

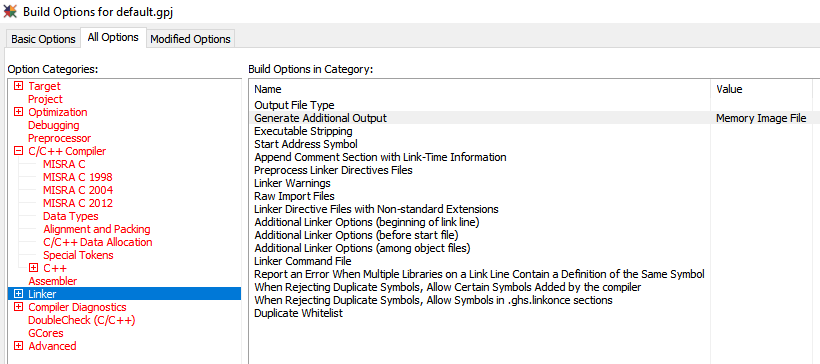


Figure 5-4: Memory Image file Configuration

After setting the memory image file click on done in the right corner and built the application code. After successful built open the Uart\_print\_app->bin->debug folder in the application code project. There we can see a file with extension. mem as shown in below image

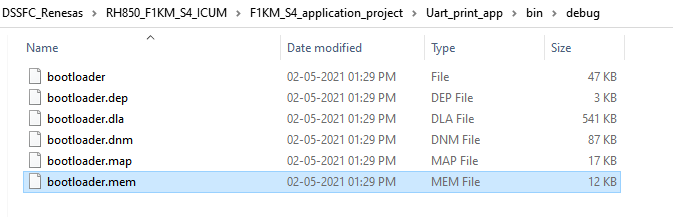


Figure 5-5: Memory Image file location in application project folder

**NOTE:** Kindly rename the “.mem” extension to “.bin”

Application image is created successfully and the next step is to sign and encrypt the application image and update the same to the MCU.

## Image Signing

As mechanism to confirm the authenticity of the image, it is necessary to sign the image with the private key generated earlier. For faster operation on the target side instead of signing the whole image, it is sufficient to sign SHA256 hash of the application image.

Generated application image in section **5.1 Application image preparation** isencrypted with Renesas SBL manager windows application.

Renesas SBL manager provides both encrypted and unencrypted application image, based on the configuration provided for macro R\_SECBOOT\_CRYPTO\_SCHEME in **r\_sb\_boot\_config.h** file.

If R\_SECBOOT\_CRYPTO\_SCHEME is set with any one of the below macro’s

R\_SECBOOT\_WITHOUT\_SECURITY (1U)

R\_SECBOOT\_RSA\_WITHOUT\_ENCRYPT\_SHA256 (2U)

For above security mechanism there is no encryption, and the section key configuration is not required in the Renesas SBL application. Select the private key which is previous generated in section **3.7 crypto setup.** In the input field select the application image need to be updated. Provide the file name and location where signed output of application image need to be store.

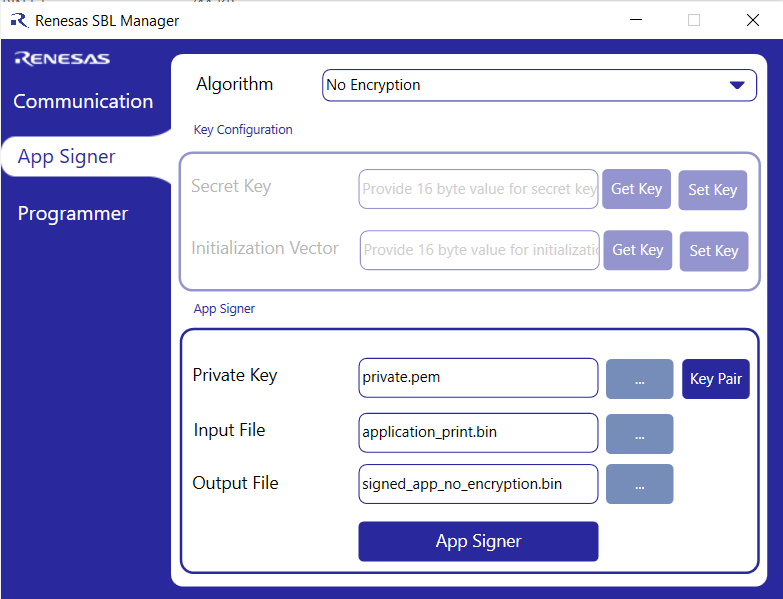


Figure 5-6: Configuration for no encryption application image generation

In-order to do update the application image in a secure way, secure boot provides following cryptography algorithm.

R\_SECBOOT\_AES128\_CMAC\_WITH\_CBC\_SHA256 (3U)

R\_SECBOOT\_AES128\_CMAC\_WITH\_ECB\_SHA256 (4U)

R\_SECBOOT\_RSA\_WITH\_AES128\_CBC\_SHA256 (5U)

R\_SECBOOT\_RSA\_WITH\_AES128\_ECB\_SHA256 (6U)

To do the AES- ECB or CBC encryption algorithm we need to load the encryption key and encryption initialization vector to the board. We can configure the key and get the value of old key and IV which is already configured in the board.

For the first time, there will be no key or IV configuration available. Set only the secret key if the encryption algorithm is AES-ECB and Initialization vector too if encryption algorithm is AES-CBC

Before loading key and IV, the debug UART of the board should connected to the host PC where the SBL manager application is installed and opened. In the communication tab of the SBL manager select UART radio button and choose the communication port. Power on the board and click on connect.

In app signer tab set 16-byte secret key and press set key button. Once the key is configured successfully to the board it will prompt success message. In case of failure, it will give failure message. Similarly, the initialization vector is also can be configured if the algorithm is AES-CBC.

This configured key and IV can be get for next encryption process using get key option in the Renesas SBL manager application, If they wish to use same secret key and IV otherwise it can be changed by setting the another key ad IV to the board.

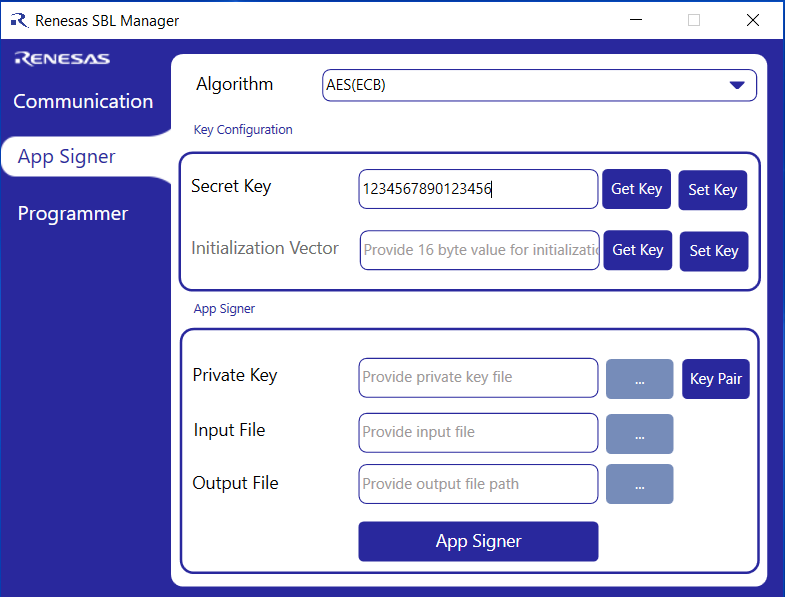


Figure 5-7: secret key Configuration

Once the secret key for AES-ECB algorithm is configured, other field like private key, input file, output file can be configured, and the application image can be generated by selecting app singer button at the bottom. It will prompt with the success message after successful generation of the encrypted application image file.

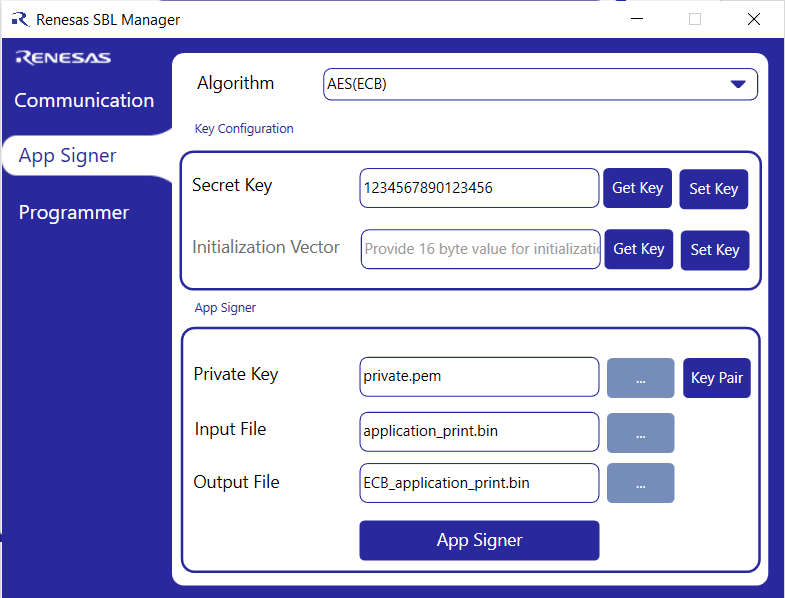


Figure 5-8: SBL key configuration settings for AES-ECB

Similarly for AES-CBC algorithm encryption with secret key we need to configure the Initialization vector and other parameter as same as AES-ECB algorithm.

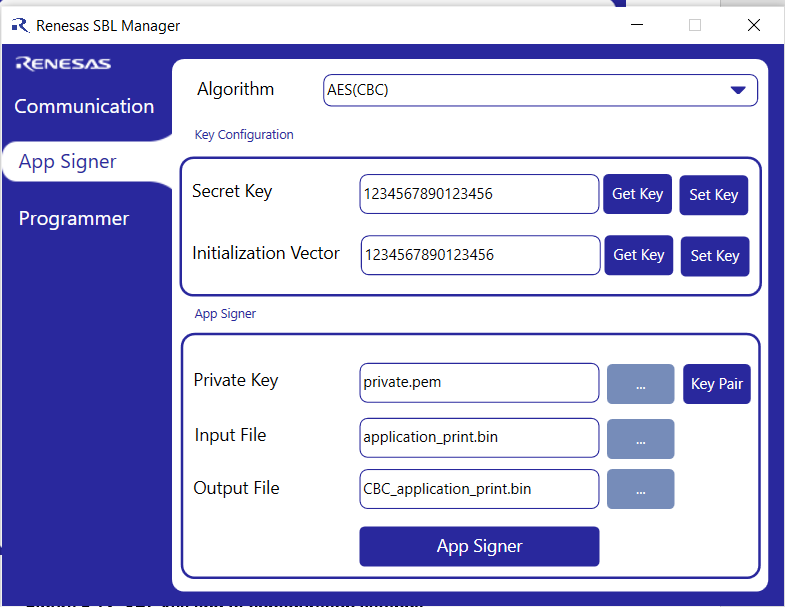


Figure 5-9: SBL key and IV configuration settings for AES-ECB

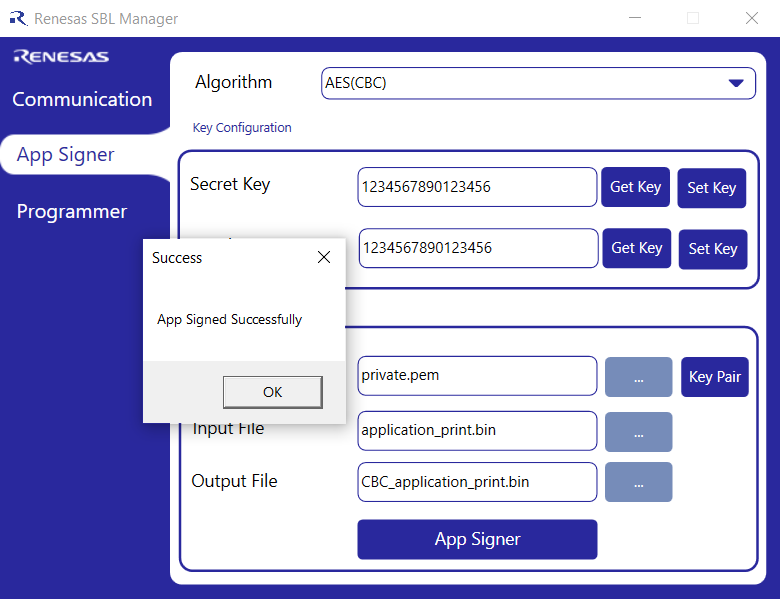


Figure 5-10: Application image creation success

The below figure show all the three application image with and without encryption image generated in the configured file location

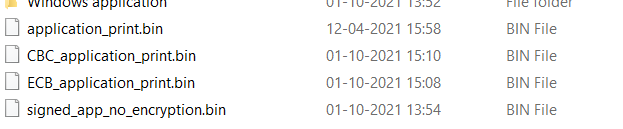


Figure 5-11: Generated encrypted and unencrypted application image in folder

With this step, the final signed application image is ready in the configured directory.

## Performing Update using UART X-modem protocol

Power on the target and it should be in the bootloader mode. With the UART connected to a PC and run the SBL Manager tool. Select the UART radio button and select the respective port number and press the connect button. After successful connection the button state will change as disconnect and Ready to program message will display in the screen.

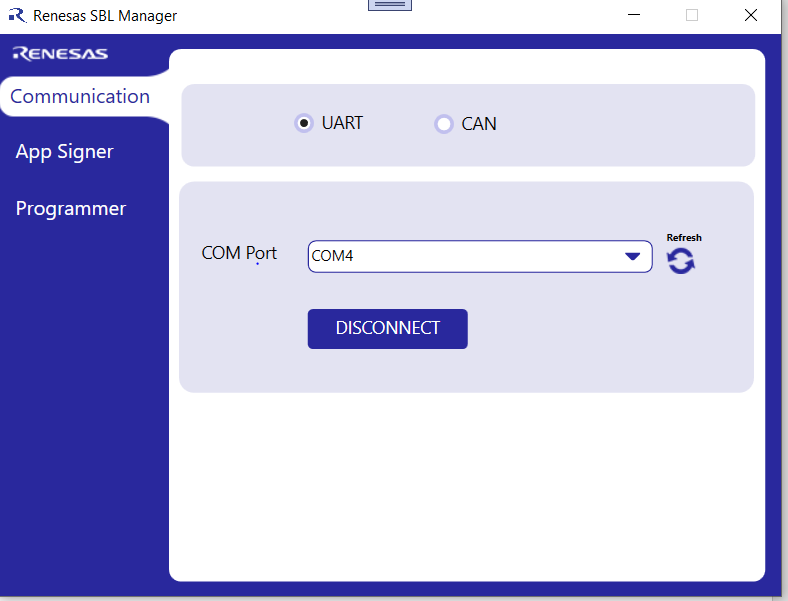


Figure 5-12. UART COM port connection

Go to the programmer tab and select the signed f1kms1\_internal\_Flash\_signed.bin which was signed earlier and available in the SBL manager directory. After that click program and wait for the program to be updated.

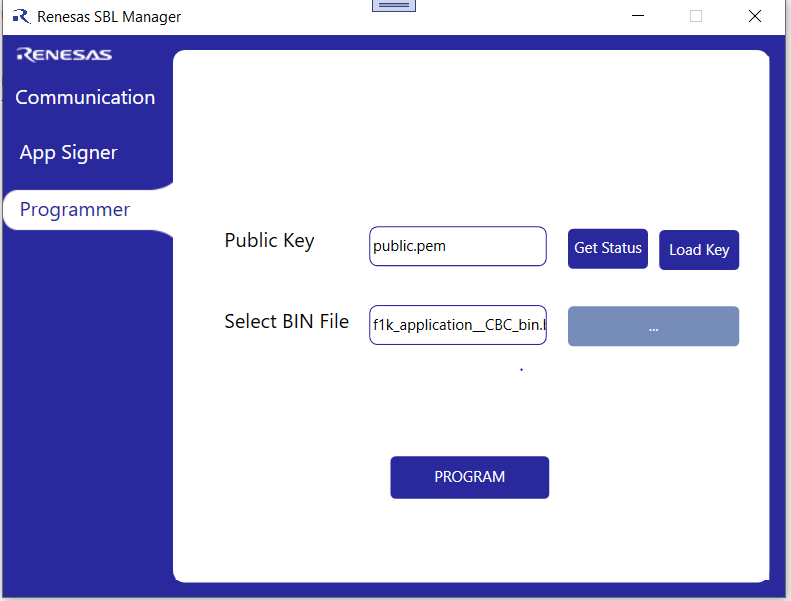


Figure 5-13. Program file selection

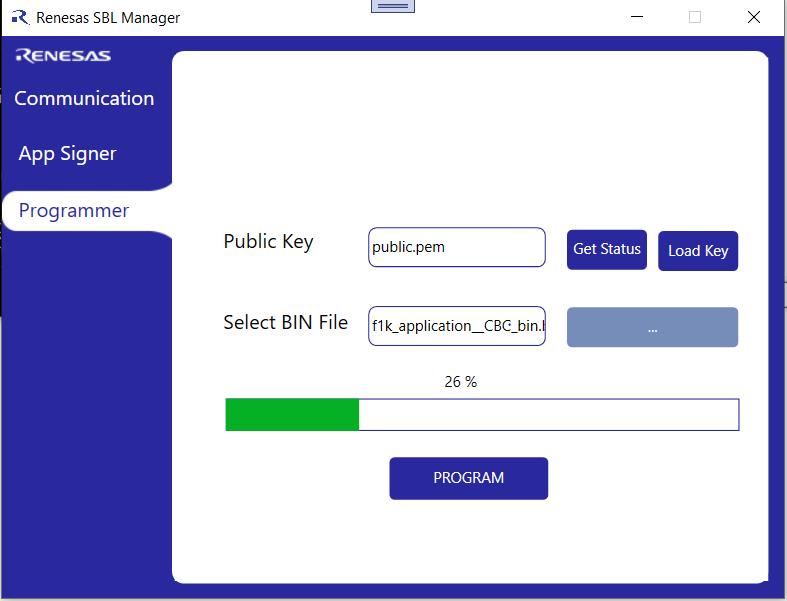


Figure 5-14. Image loading screen

Up on successful download, the following information will be displayed.

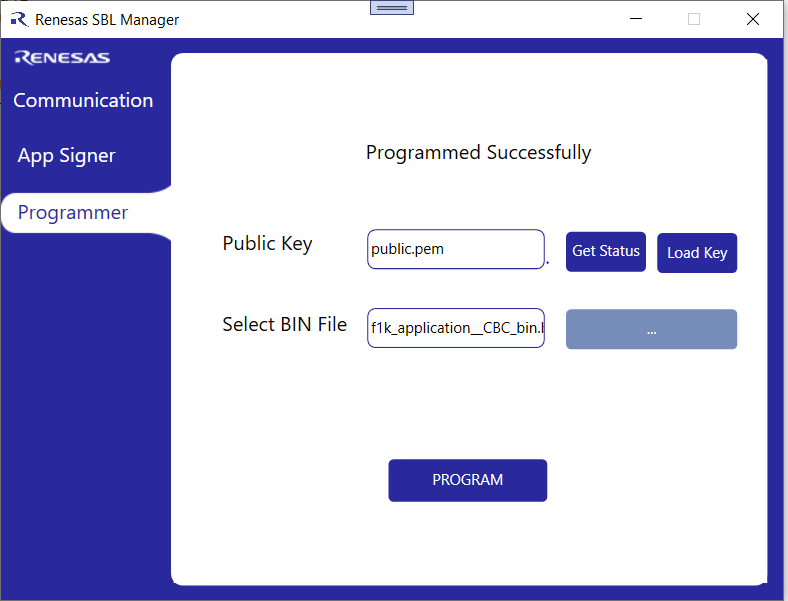


Figure 5-15. Image Download Completion Status

Once application image is loaded successfully, close the application and open the serial port tera term or putty to see the debug message. If all encryption key and IV is correct and application image will be decrypt-ed and stored in the active region and execute the same.

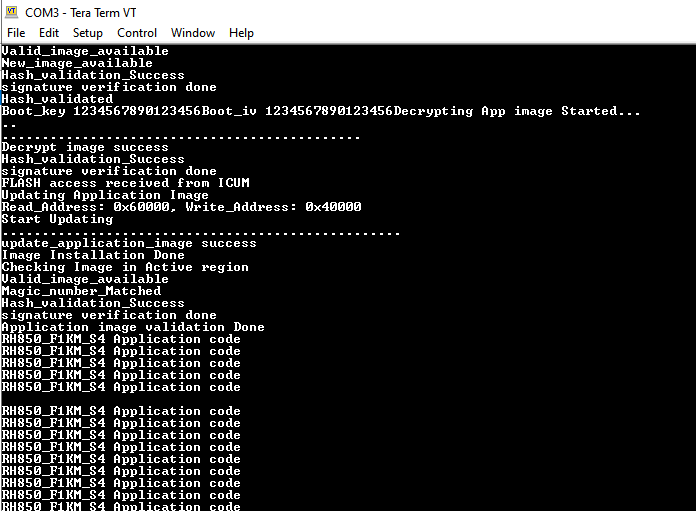


Figure 5-16: Debug print of the board

## Performing Update using CAN UDS protocol

Power on the target and it should be in the bootloader mode. With the PEAK/ CAN Vector tool connected to a PC and run the SBL Manager tool. Select the CAN radio button and select the respective connected CAN tool and press the connect button. After successful connection the button state will change as disconnect and CAN initialization success will display in the screen.

Following CAN UDS services are used for firmware file transmission,

Table 5-1. CAN-UDS Services

|  |  |
| --- | --- |
| **S.No** | **Service** |
| 1 | 0x10 - Diagnostic Session Control |
| 2 | 0x28 – Communication |
| 3 | 0x2E – Write Data by Identifier |
| 4 | 0x34 – Request Control |
| 5 | 0x36 – Transfer Data |
| 6 | 0x37 – Request Transfer Exit |
| 7 | 0x11 – ECU Rest |

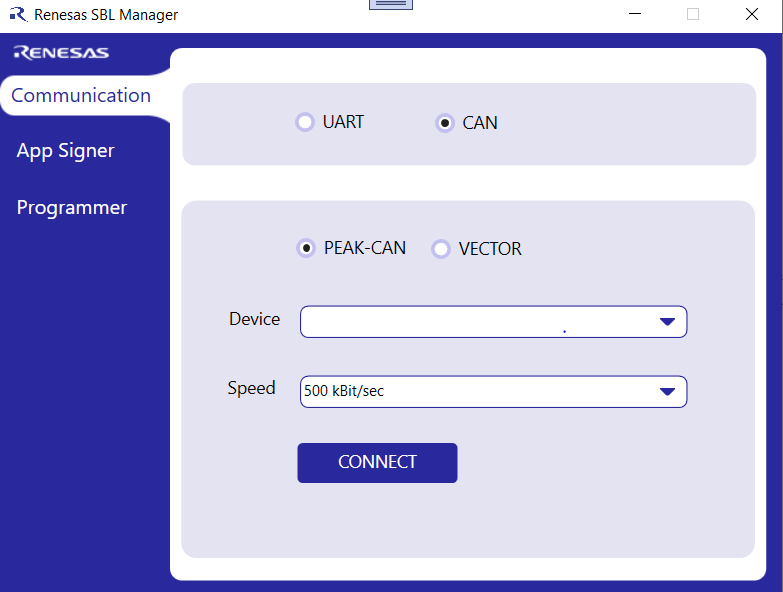


Figure 5-17. CAN Vector connection

Go to the programmer tab and select the signed f1kms1\_internal\_Flash\_signed.bin which was signed earlier and available in the SBL manager directory. After that click program and wait for the program to be updated.

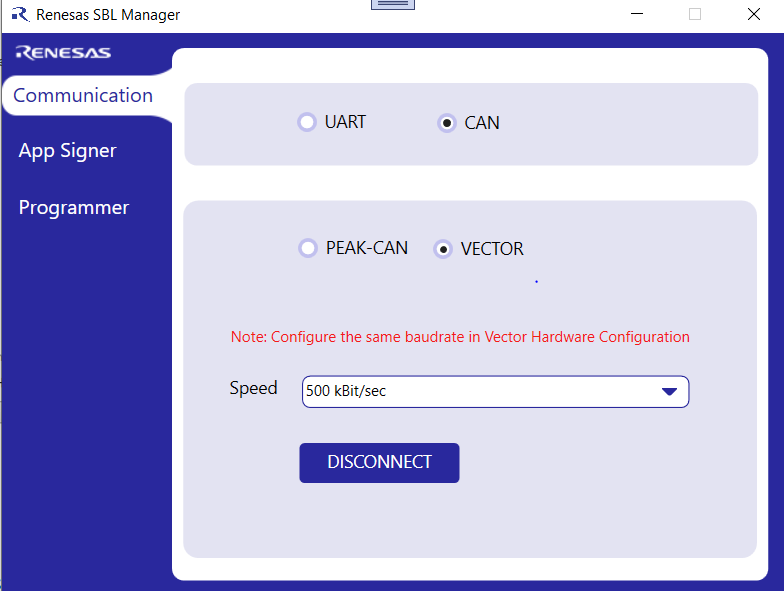


Figure 5-18. Program file selection

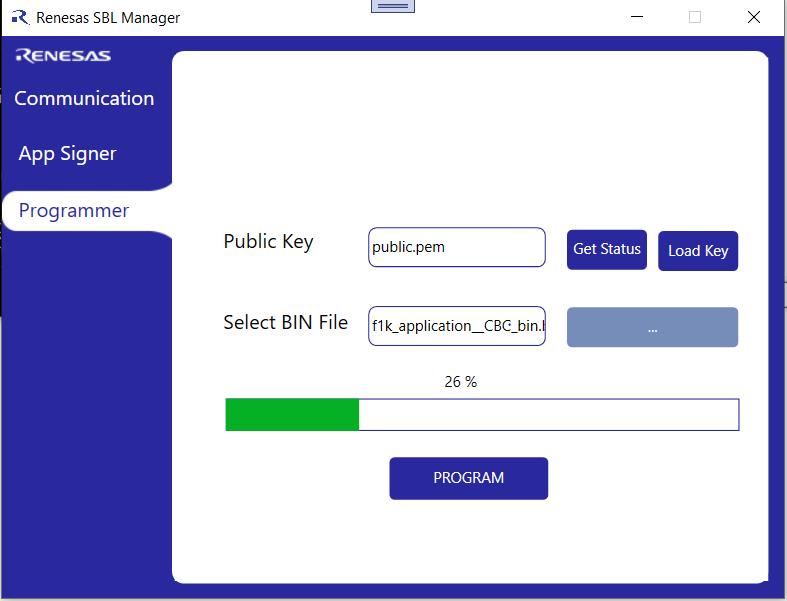


Figure 5-19. Image loading screen

Up on successful download, the following information will be displayed.

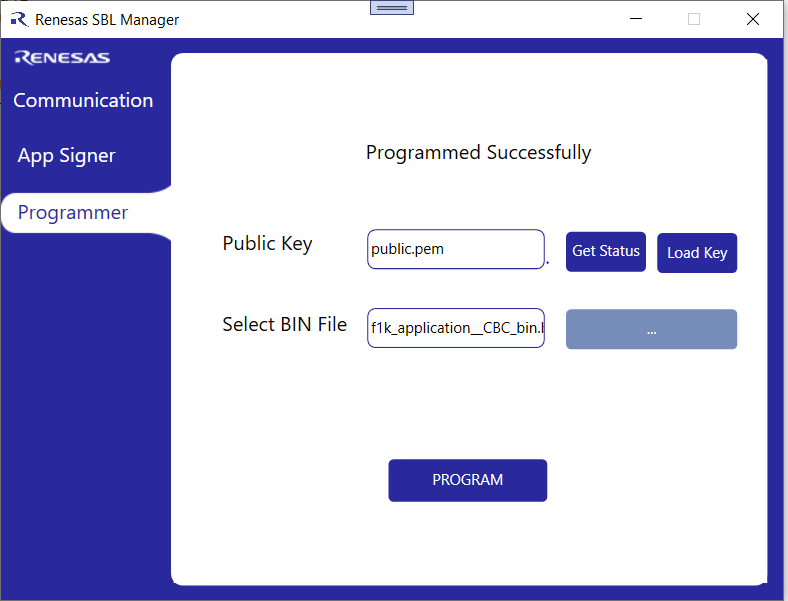


Figure 5-20. Image Download Completion Status

Once application image is loaded successfully, close the application and open the serial port tera term or putty to see the debug message. If all encryption key and IV is correct and application image will be decrypt-ed and stored in the active region and execute the same.

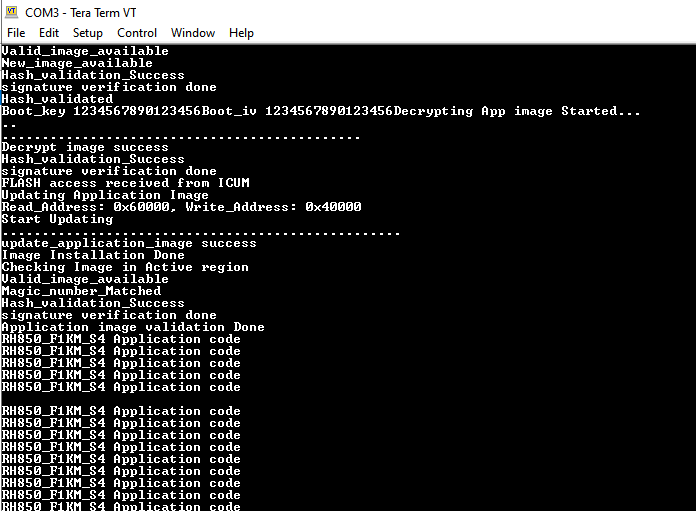


Figure 5-21: Debug print of the board

# Notes

1. Care must be taken to protect the private key. If it is compromised, there is no way to update th public key and hence the devices in the field may have to be recalled.
2. As the locking out of the bootloader flash area, is a one-step process, ensure all the testing are dine before carrying out it.

# ICUM Cryptography

ICUM cryptography is applicable only for RH850\_F1KM\_S4 series. To enable ICUM hardware, kindly go through the document “E1/E20/E2 Emulator additional Document for user's manual: Debugging methods and notes on product equipped with the ICUM”.

Required Option bytes settings are done for enabling the ICUM cryptographic engine. The ICUM cryptographic engine is designed to be executed from 0x300000 and the code flash area is started from 0x300000. These settings are provided by option bytes settings 9 and 10. Once ICUM is enabled, the ICUM source code must be flashed in the ICUM code flash area. Refer to document “RH850/F1KH, RH850/F1KM ICUMD User's Manual: Hardware” for further description of the option bytes.

After successful programming of the cryptographic engine in ICUM (ICUP core) and secure bootloader in CPU (PE core), to invoke the cryptography calls, the input data for the function has to be load in a shared memory area (0xFEFE8000) in accordance to a predefined structure.

This structure is defined in pe.h and is used as the protocol between the ICUM (ICUP core) and CPU (PE core). The structure definition is as follows:

#define IN\_OUT\_DATA\_LEN\_MAX 16

#define SHA256\_DATA\_LEN\_MAX 64

typedef struct

{

ICUM\_CIPHER\_MODES cipher\_mode;

ICUM\_CYPTO\_METHOD crypto\_method;

unsigned char u8\_aes\_input\_data[IN\_OUT\_DATA\_LEN\_MAX];

unsigned char u8\_aes\_output\_data[IN\_OUT\_DATA\_LEN\_MAX];

unsigned char u8\_aes\_key[16];

unsigned char u8\_aes\_init\_vector[16];

unsigned char u8\_sha256\_input\_data [SHA256\_DATA\_LEN\_MAX];

unsigned char u8\_sha256\_output\_data [32];

unsigned char u8\_hash\_data [32];

unsigned char u8\_sign\_data [256];

unsigned char u8\_pub\_key\_data[2048];

uint32\_t u32\_pub\_key\_len;

uint8\_t u8\_sign\_verification\_status;

uint32\_t u32\_block\_count;

uint8\_t u8\_last\_block\_flag;

} ISD;

**ICUM\_CIPHER\_MODES:**

ICUM supports the following cyber mode.

typedef enum

{

*CBC\_ENCRYPT* = 1, /\*AES\_CBC Encryption\*/

*CBC\_DECRYPT*, /\*AES\_CBC Decryption\*/

*ECB\_ENCRYPT*, /\*AES\_ECB Encryption\*/

*ECB\_DECRYPT*, /\*AES\_ECB Decryption\*/

*CMAC\_INIT*, /\*AES\_CMAC Generation initialization\*/

*CMAC\_GENERATION*, /\*AES\_CMAC Generation\*/

*SHA256\_INIT*, /\*SHA256 initialization for mbedTLS\*/

*SHA256\_FREE*, /\*SHA256 free assigned memory for mbedTLS\*/

*SHA256\_UPDATE*, /\*SHA256 update checksum with given input with mbedTLS stack\*/

*SHA256\_FINAL*, /\*SHA256 end and return the final checksum with mbedTLS stack\*/

*RSA\_SIGN\_VERIFICATION* /\*RSA key verification with mbedTLS stack\*/

} ICUM\_CIPHER\_MODES;

**ICUM\_CYPTO\_METHOD:**

ICUM supports two cryptographic methods, one with the ICUM hardware and another with the mbedTLS stack.

typedef enum

{

ICUM\_CRYPTO\_HARDWARE = 1,

ICUM\_CRYPTO\_SOFTWARE

} ICUM\_CYPTO\_METHOD;

**Structure members in shared memory:**

* unsigned char u8\_aes\_input\_data[IN\_OUT\_DATA\_LEN\_MAX]

This input data buffer of length 16 is used to store the input data for AES-CBC and AES-ECB algorithms.

* unsigned char u8\_aes\_output\_data[IN\_OUT\_DATA\_LEN\_MAX]

This output data buffer of length 16 is used to store the output data for AES-CBC and AES-ECB algorithms.

* unsigned char u8\_aes\_key[16]

Key buffer of length 16 is used to store the key for AES-CBC, AES-ECB, AES-CMAC algorithm.

* unsigned char u8\_aes\_init\_vector[16]

Initialize vector buffer of length 16 is used to store the IV for AES-CBC algorithm.

* unsigned char u8\_sha256\_input\_data [SHA256\_DATA\_LEN\_MAX];

sha256\_input\_data buffer of length 64 is used to store the input data to be added to checksum of SHA256 algorithm.

* unsigned char u8\_sha256\_output\_data [32]

sha256\_output\_data buffer of length 32 is used to store the final checksum value of SHA256 algorithm.

* unsigned char u8\_hash\_data [32]

hash\_data buffer of length 32 is used to store the hash input for RSA verification

* unsigned char u8\_sign\_data [256]

sign\_data buffer of length 256 is used to store the sign input for RSA verification

* unsigned char u8\_pub\_key\_data[2048];

pub\_key\_data buffer of length 2kB is used to store the pubilc key input for RSA verification

* uint32\_t u32\_pub\_key\_len

pub\_key\_len is the length of public key.

* uint8\_t u8\_sign\_verification\_status

sign\_verification\_status is used to store the result of the RSA verification done in ICUM. It may be SUCCESS( 0 ) or FAIL( 1 ) according to the input provided.

* uint32\_t u32\_block\_count;

Block\_count is used to store the number of blocks in case of AES\_CBC, AES\_ECB and AES\_CMAC. During SHA256\_UPDATE it hold the length of the input data.

* uint8\_t u8\_last\_block\_flag;

Last\_block\_flag is used in AES\_CMAC algorithm to notify the ICUM that the last block is reached by setting it to 1.

From secure bootloader, an interrupt (INTPES) will be raised to the ICUM after loading respective inputs in the shared memory structure. ICUM, up on reception of this interrupt, will clear the interrupt and do the cryptographic request. After completing the request, ICUM will trigger another interrupt (INTICUPS) to secure booloader. Secure boot loader will clear this interrupt triggered from ICUM and read the response data from the shared memory structure.

**ICUM API’s for Cryptography**

**AES Cryptography Support**

This section explains the AES cryptography support provided by the bootloader.

**AES Cryptography Process**

The secure bootloader running at CPU (PE) core can use the following APIs to perform an AES crypto process in ICUM.

int r\_sb\_aes\_crypto\_init(uint8\_t \*u8\_key, uint8\_t \*u8\_iv);

int r\_sb\_aes\_ecb\_crypto (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_ptr\_output\_data,uint8\_t u8\_flag\_type, uint32\_t u32\_len);//AES-ECB-Decrption/Encryption

int r\_sb\_aes\_cbc\_crypto (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_ptr\_output\_data, uint8\_t \*u8\_ptr\_iv, uint8\_t u8\_flag\_type, uint32\_t len)//AES-CBC-Decrption/Encryption

It is assumed an AES-CBC based decryption is done for 512 bytes of data.

uint8\_t encrypted\_data[512]; //Buffer holding encrypted data

uint8\_t decrypted\_data[512]; //Buffer to hold decrypted data

#define R\_SB\_DECRYPT\_READ\_SIZE 16

uint8\_t iv[16] = R\_SECBOOT\_DECRYPT\_INITIALIZATION\_VECTOR;

uint8\_t key[16] = R\_SECBOOT\_DECRYPT\_KEY;

r\_sb\_aes\_decryption\_init (&key [0], &iv [0]); // For AES-EBC ptr\_iv parameter should be Zero.

for(int data\_index = 0;data\_index < sizeof(encrypted\_data); data\_index += R\_SB\_DECRYPT\_READ\_SIZE)

{

ret = r\_sb\_aes\_ecb\_crypto (&encrypted\_data[0], &decrypted\_data[0], ptr\_iv, 1, R\_SB\_DECRYPT\_READ\_SIZE);

}

The below section explains the implementation of the above API’s inside the ICUM engine.

1. **Initializing IV and key for AES-ECB and AES-CBC**

For loading the initialization vector (IV) for AES-CBC and key for AES-ECB and AES-CBC, the below API is used. A buffer address for key and IV should passed as parameter to the function. For AES-ECB IV, this is not required and the parameter can be zero (0) instead.

int r\_sb\_aes\_crypto\_init(uint8\_t \*u8\_key, uint8\_t \*u8\_iv)

{

int err = -1;

#if ((R\_MCU\_TYPE < R\_F1KM\_S4) && (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE ))

err = mbedtls\_aes\_setkey\_dec( &aes, u8\_key, 16\*8 );

#elif ((R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS) || (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS\_WITH\_DMA))

err = buffer\_load\_plain\_key(u8\_key);

//check if MCU is R\_F1KM\_S4 and respective crypto algorithm

#elif ((R\_MCU\_TYPE == R\_F1KM\_S4) && ((R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUM) || (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE)))

#if ((R\_SECBOOT\_CRYPTO\_SCHEME == R\_SECBOOT\_RSA\_WITH\_AES128\_CBC\_SHA256) || (R\_SECBOOT\_CRYPTO\_SCHEME == R\_SECBOOT\_AES128\_CMAC\_WITH\_CBC\_SHA256))

err = init\_icum\_crypto(u8\_key, u8\_iv); /\*store the key for CBC encryption and decryption\*/

#else

err = init\_icum\_crypto(u8\_key, 0); /\*store the key for ECB encryption and decryption\*/

#endif

#endif

if (err != 0)

return -1;

return err;

}

1. **Process AES-ECB and AES-CBC crypto algorithm API**

The API supports AES-ECB and AES-CBC algorithm with ICUM hardware as well with mbedTLS. To do the encryption or decryption operation, load the starting address of input data and output data as first two parameters.

int r\_sb\_aes\_ecb\_crypto (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_ptr\_output\_data,uint8\_t u8\_flag\_type, uint32\_t u32\_len)

{

long err = -1; //initialize the error value.

int block = u32\_len/BLOCK\_SIZE; //calculate the block for given length. one block has 16 bytes

#if ((R\_MCU\_TYPE < R\_F1KM\_S4) && (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE))

err = mbedtls\_aes\_ecb\_crypto (u8\_ptr\_input\_data, u8\_ptr\_output\_data, u8\_flag\_type,block);

#elif (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS)

err = buffer\_enc\_dec\_ecb(RAM\_KEY, (unsigned char \*)u8\_ptr\_input\_data, (unsigned char \*)u8\_ptr\_output\_data, block, 1);

#elif ((R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUM) || (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE))

uint8\_t u8\_crypto\_type = 0;

if (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE)

u8\_crypto\_type = R\_TYPE\_SOFTWARE;

uint8\_t u8\_cipher\_mode = u8\_flag\_type+ECB\_ENCRYPT; //add ECB algorithm enumeration with flag type

err = process\_icum\_crypto (u8\_ptr\_input\_data , u8\_ptr\_output\_data, block, u8\_crypto\_type, u8\_cipher\_mode);

#elif (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS\_WITH\_DMA)

err = buffer\_dma2\_enc\_dec(RAM\_KEY, 0, (unsigned char \*)u8\_ptr\_input\_data, (unsigned char \*)u8\_ptr\_output\_data, block , 1);

#endif

if (err != 0)

return -1;

return 0;

}

int r\_sb\_aes\_cbc\_crypto (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_ptr\_output\_data, uint8\_t \*u8\_ptr\_iv,uint8\_t u8\_flag\_type, uint32\_t u32\_len)

{

long err = -1; //initialize the error value.

int block = u32\_len/BLOCK\_SIZE; //calculate the block for given length. one block has 16 bytes

#if ((R\_MCU\_TYPE < R\_F1KM\_S4) && (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE))

if(u8\_flag\_type)

err = mbedtls\_aes\_crypt\_cbc(&aes, MBEDTLS\_AES\_DECRYPT, u32\_len, u8\_ptr\_iv, u8\_ptr\_input\_data, u8\_ptr\_output\_data );

else

err = mbedtls\_aes\_crypt\_cbc(&aes, MBEDTLS\_AES\_ENCRYPT, u32\_len, u8\_ptr\_iv, u8\_ptr\_input\_data, u8\_ptr\_output\_data );

#elif (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS)

err =buffer\_enc\_dec\_cbc(RAM\_KEY, u8\_ptr\_iv, (unsigned char \*)u8\_ptr\_input\_data, (unsigned char \*)u8\_ptr\_output\_data, block, 1);

#elif ((R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUM) || (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE))

uint8\_t u8\_crypto\_type = 0;

if (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_SOFTWARE)

u8\_crypto\_type = R\_TYPE\_SOFTWARE;

uint8\_t u8\_cipher\_mode = u8\_flag\_type+CBC\_ENCRYPT; //add ECB algorithm enumeration with flag type

err = process\_icum\_crypto (u8\_ptr\_input\_data , u8\_ptr\_output\_data, block, u8\_crypto\_type, u8\_cipher\_mode);

#elif (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS\_WITH\_DMA)

err = buffer\_dma2\_enc\_dec(RAM\_KEY, u8\_ptr\_iv, (unsigned char \*)u8\_ptr\_input\_data, (unsigned char \*)u8\_ptr\_output\_data, block, 1);

#endif

if (err != 0)

return -1;

return 0;

}

**AES-CMAC Support**

This section explains the AES-CMAC support provided by the bootloader.

**AES CMAC Process**

The secure bootloader running at CPU (PE) core can use the following APIs to perform an CMAC operation.

int r\_sb\_cmac\_generation (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_cmac\_output\_data, uint8\_t \*u8\_key, uint32\_t u32\_len);

It is assumed an CMAC generation is done for 32bytes of data.

uint8\_t cmac\_key[16];//CMAC Key

uint8\_t hash\_data[32];//RSA-256 Hash output data

uint8\_t cmac\_output\_data[32];//Output data buffer

r\_sb\_cmac\_generation (& hash\_data [0], & cmac\_output\_data[0], &cmac\_key [0], 256);

* 1. **AES-CMAC Generation**

To generate AES-CMAC the input data address should pass as first parameter and the output data address as second parameter.

Block count represents the number of blocks in the input data. The block count should be a multiple of 16 bytes. Internal implementation of the function should give a fair idea.

int r\_sb\_cmac\_generation (uint8\_t \*u8\_ptr\_input\_data, uint8\_t \*u8\_cmac\_output\_data, uint8\_t \*u8\_key, uint32\_t u32\_len)

{

uint32\_t err = -1;

#if ((R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS) || (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUS\_WITH\_DMA))

if (ICUS\_init () != 0)

PrintText("ICUS\_init failed\r\n");

err = buffer\_load\_plain\_key(u8\_key);

err = buffer\_dma2\_cmac(RAM\_KEY, (unsigned char \*) u8\_ptr\_input\_data, u32\_len, (unsigned char \*)u8\_cmac\_output\_data, 0);

buffer\_dma2\_set\_callback\_func(0);

while((err == 0) && (!buffer\_dma2\_completed()))

{

\_\_asm("halt");

NOPS(5);

};

#elif (R\_CRYPTO\_ALGOTITHM\_IMPLEMENTATION\_TYPE == R\_TYPE\_HARDWARE\_ICUM)

if (cmac\_init (u8\_key) != 0){

PrintText("ICUM CMAC\_init failed\r\n");

return 1;}

//len is in bits, block should be multiple of 16bytes so divide the u32\_len by 16\*8 = 128

u32\_len /= 128;

err = icum\_cmac\_generation (u8\_ptr\_input\_data , u8\_cmac\_output\_data, u32\_len/\*block param\*/);

if (err == 0)

return 0;

#endif

return err;

}

**SHA256 Support**

This section explains the SHA256 support provided by the bootloader.

To generate a hash data for given input, the following procedure is followed. The input data should be in multiples of 32 (256 bit) bytes as ICUM cryptographic process follows 256 bit hashing algorithm. The following API’s are used for generating hash

r\_sb\_sha256\_init ();

int r\_sb\_sha256\_update (uint8\_t \*u8\_ptr\_input\_data, uint16\_t u16\_len);

int r\_sb\_sha256\_finish (uint8\_t \*u8\_ptr\_output\_data);

int r\_sb\_sha256\_free ();

Let us elaborate with an example,

To generate a hash for 256 bytes of input data, the hashing initialization should done with below function from bootloader.

r\_sb\_sha256\_init ();

Followed by the initialization, address of input data with its corresponding length is passed for hash generation. Maximum of length is 64 bytes, if the input data is more than 64 bytes then it can be done in iteration of 64 bytes. In our example we are doing 256 bytes, so 256/64 which is 4. Then we have to call below function for 4 times.

uint8\_t data [256]; // input data

uint16\_t data\_len = 256; // length of data is 256

r\_sb\_sha256\_update (data, data\_len);

Once all input data is sent for generation of hash then the final hash data of 32 bytes is obtained by the below function call

uint8\_t hash\_output\_data [32]; // hash output data

r\_sb\_sha256\_finish (&hash\_output\_data[0]);

After successful generation of hash, to free the memory below API is called from the bootloader,

r\_sb\_sha256\_free ();

* **SHA256 initialization**

For initializing the SHA256 algorithm the below API is used. SHA256 initialization is done with mbedTLS stack. The function has no parameter.

r\_sb\_sha256\_init ();// initialize the sha256 process.

int r\_sb\_sha256\_init ()

{

#if (R\_MCU\_TYPE < R\_F1KM\_S4)

mbedtls\_sha256\_init (&mbed\_sha256\_ctx);

mbedtls\_sha256\_starts (&mbed\_sha256\_ctx, 0); /\* 0 - SHA-256, 1 - SHA-224 \*/

#elif (R\_MCU\_TYPE == R\_F1KM\_S4)

icum\_sha256\_init ();

#endif

}

* **SHA256 free initialized memory**

This API is used to free the memory which is occupied during processing the hash input data to generate a checksum of 256 bits. The function has no parameter.

int r\_sb\_sha256\_free ()

{

#if (R\_MCU\_TYPE < R\_F1KM\_S4)

mbedtls\_sha256\_free (&mbed\_sha256\_ctx);

#elif (R\_MCU\_TYPE == R\_F1KM\_S4)

icum\_sha256\_free ();

#endif

}

* **SHA256 generation with given input data**

This API is used to generate the checksum of 256 bits with the given input data. The input data buffer address should be the first parameter of the function. Block count is the second parameter and it is number bytes given as input data.

int r\_sb\_sha256\_update (uint8\_t \*u8\_ptr\_input\_data, uint16\_t u16\_len)

{

#if (R\_MCU\_TYPE < R\_F1KM\_S4)

mbedtls\_sha256\_update (&mbed\_sha256\_ctx, u8\_ptr\_input\_data, u16\_len );

#elif (R\_MCU\_TYPE == R\_F1KM\_S4)

icum\_sha256\_update (u8\_ptr\_input\_data, u16\_len);

#endif

}

* **SHA256 output data for given input**

This API is used to get the final checksum for the given input data in “**SHA256 generation with given input data”** function. A pointer to the buffer which will store the checksum of 256 bits is passed as the parameter.

int r\_sb\_sha256\_finish (uint8\_t \*u8\_ptr\_output\_data)

{

#if (R\_MCU\_TYPE < R\_F1KM\_S4)

mbedtls\_sha256\_finish (&mbed\_sha256\_ctx, u8\_ptr\_output\_data);

#elif (R\_MCU\_TYPE == R\_F1KM\_S4)

icum\_sha256\_final (u8\_ptr\_output\_data);

#endif

}

**RSA-Sign verification Support**

This section explains the RSA-Sing verification support provided by the bootloader.

**RSA-Sign verification process**

The secure bootloader running at CPU (PE) core can use the following APIs to perform a sing verification.

int r\_sb\_rsa\_sign\_verification (uint8\_t \*u8\_hash, uint8\_t u8\_hash\_len,uint8\_t \*u8\_pubkey,uint16\_t u16\_pub\_key\_len, uint8\_t \*u8\_sign, uint16\_t u16\_sign\_len);

It is assumed an sign verification is done for hash data by using pubkey.

uint8\_t pub\_key[]; //public Key

uint8\_t hash\_data[32]; //RSA-256 Hash output data

uint8\_t hash\_len = 32; //hash length

uint8\_t cmac\_output\_data[32];//Output data buffer

uint8\_t \*sign\_data\_read\_from\_header;

uint16\_t sign\_len = 256; //sign length

r\_sb\_rsa\_sign\_verification (&hash\_data [0], hash\_len, &pub\_key [0], sizeof(pub\_key ), sign\_data\_read\_from\_header, sign\_len);

int r\_sb\_rsa\_sign\_verification (uint8\_t \*u8\_hash, uint8\_t u8\_hash\_len,uint8\_t \*u8\_pubkey,uint16\_t u16\_pub\_key\_len, uint8\_t \*u8\_sign, uint16\_t u16\_sign\_len)

{

#if (R\_MCU\_TYPE < R\_F1KM\_S4)

return mbedtls\_rsa\_sign\_verification(u8\_hash, u8\_hash\_len,u8\_pubkey,u16\_pub\_key\_len, u8\_sign, u16\_sign\_len);

#elif (R\_MCU\_TYPE == R\_F1KM\_S4)

return icum\_sign\_verification (u8\_hash, u8\_pubkey, u16\_pub\_key\_len, u8\_sign);

#endif

}

Revision History <revision history>

|  |  |  |  |
| --- | --- | --- | --- |
| Rev. | Date | Description | |
| Page | Summary |
|  |  |  |  |
|  |  |  |  |

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

2. Processing at power-on

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

3. Input of signal during power-off state

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

4. Handling of unused pins

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

5. Clock signals

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

6. Voltage application waveform at input pin

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between VIL (Max.) and VIH (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between VIL (Max.) and VIH (Min.).

7. Prohibition of access to reserved addresses

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

8. Differences between products

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

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.

2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.

3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.

4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.

5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.

6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.

9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.

12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.

13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.

14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

|  |  |  |
| --- | --- | --- |
| Corporate Headquarters |  | Contact information |
| TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan  [www.renesas.com](https://www.renesas.com/) |  | 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/](https://www.renesas.com/contact/). |
| Trademarks |  |  |
| Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners. |  |  |