Renesas RA Family

Injecting Plaintext User Keys

Introduction

Cryptography is important because it provides the tools to implement solutions for authenticity, confidentiality, and integrity, which are vital aspects of any security solution. In modern cryptographic systems, the security of the system no longer depends on the secrecy of the algorithm used but rather on the secrecy of the keys.

There are different types of security engines across the various RA Family MCUs. The MCU's hardware user's manual identifies the security engine that is provided in the MCU.

The security engines can operate in two different modes, called Compatibility Mode and Protected Mode. The application note *Renesas Security Engine Operational Modes (R11AN0498)* explains the definition of the two modes and their use cases. The key injection capabilities, in brief, are:

- Compatibility Mode – both plaintext and secure key injection are supported. All security engines used in RA Family MCUs support this mode.
- Protected Mode – only secure key injection is supported. As such, the Protected Mode does not support the capabilities described in this application project. The current list of security engines that support Protected Mode comprises the Secure Crypto Engine 9 (SCE9) and the Renesas Secure IP security engines.

With this release, this application project demonstrates the following plaintext key injection processes:

- RSIP-E50D Compatibility Mode AES-256 and ECC secp256r1 plaintext key injection using RA8P1 MCU Dual Core.
- RSIP-E51A Compatibility Mode AES-256 plaintext key injection using RA8M1 MCU.
- SCE9 Compatibility Mode AES-256 plaintext key injection using RA6M4 MCU.
- SCE7 Compatibility Mode AES-128 plaintext key injection using RA6M3 MCU. Compatibility Mode plaintext key injection for SCE5 and SCE5_B uses APIs identical to those of SCE7.

Required Resources

Target MCUs and Security Engines

MCUs with RSIP-E51A: RA8M1, RA8D1, RA8T1

MCUs with RSIP-E50D: RA8P1, RA8T2(*), RA8D2 (**), RA8M2(**)

MCUs with RSIP-E11A: RA4L1

MCUs with SCE9: RA4M2, RA4M3, RA6M4, RA6M5

MCUs with SCE7: RA6M1, RA6M2, RA6M3, RA6T1

MCUs with SCE5: RA4M1, RA4W1

MCUs with SCE5_B: RA6T2

(*) These devices will be supported starting from FSP v6.1.0 and later.

(**) These devices will be supported starting from FSP v6.2.0 and later.

Development tools and software

- e^2 studio IDE v2025-04.1
- Renesas Flexible Software Package (FSP) v6.0.0
- SEGGER J-link® USB driver

The above three software components, the FSP, J-Link USB drivers, and e^2 studio are bundled in a downloadable platform installer available on the FSP webpage at [renesas.com/ra/fsp.](renesas.com/ra/fsp)

Hardware

- EK-RA8P1, Evaluation Kit for RA8P1 MCU Group (http://www.renesas.com/ek-ra8p1)
- EK-RA8M1, Evaluation Kit for RA8M1 MCU Group (http://www.renesas.com/ra/ek-ra8m1)
- EK-RA6M4, Evaluation Kit for RA6M4 MCU Group (http://www.renesas.com/ra/ek-ra6m4)
- EK-RA6M3, Evaluation Kit for RA6M3 MCU Group (http://www.renesas.com/ra/ek-ra6m3)
- Workstation running Windows® 10 and Tera Term console or similar application
- One USB device cable (type-A male to micro-B male)
- One USB device cable (type-A male to type-C male) (for the EK-RA8P1 board)

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e² studio IDE and Arm® TrustZone® based development models with e² studio. The application note assumes that you have some knowledge of RA Family MCU security features. In addition, a prerequisite reading is the application note Renesas Security Engine Operational Modes (R11AN0498).

The intended audience includes product developers, product manufacturers, product support, or end users who are involved with any stage of the MCU plaintext key injection of the RA Family MCUs.

Contents

1. Root of Trust and its Protection

1.1 What is the Root of Trust

The roots of trust are highly reliable hardware, firmware, and software components that perform specific, critical security functions. For more information, see https://csrc.nist.gov/projects/hardware-roots-of-trust. In an IoT system, the root of trust typically consists of identity and cryptographic keys rooted in the hardware of a device. It establishes a unique, immutable, and unclonable identity to authorize a device in the IoT network.

- Secure boots are part of the services provided in the Root of Trust in many security systems. Public Key Encryption authenticates the application. The associated keys are part of the system's Root of Trust.
- Device Identity, which consists of Device Private Key and Device Certificate, is part of the Root of Trust for many IoT devices.

1.2 Protecting the Root of Trust

From the above Root of Trust discussion, we can realize that leakage of the cryptographic user keys can bring the secure system into a risky state. Protection of the Root of Trust involves key accessibility within the cryptographic boundary only and unclonable keys. The Root of Trust should be locked from read and write access from unauthorized parties.

The Renesas user key management system can provide all the above desired protection. In addition, Renesas user key injection services provide several options from which users can select injection methods that fit their existing architecture.

1.3 Introduction to the Security Engine and Associated Keys

The security engine (RSIP or SCE) is an isolated subsystem within the MCU. The security engine contains hardware accelerators for symmetric and asymmetric cryptographic algorithms, as well as various hashes and message authentication codes. It also contains a True Random Number Generator (TRNG), providing an entropy source for cryptographic operations. The security engine is protected by an Access Management Circuit, which can shut down the security engine in the event of an illegal external access attempt. Figure 1 shows the conceptual diagram of the security engine.

Refer to Table 1 for a list of cryptographic operations that are supported by each type of security engine.

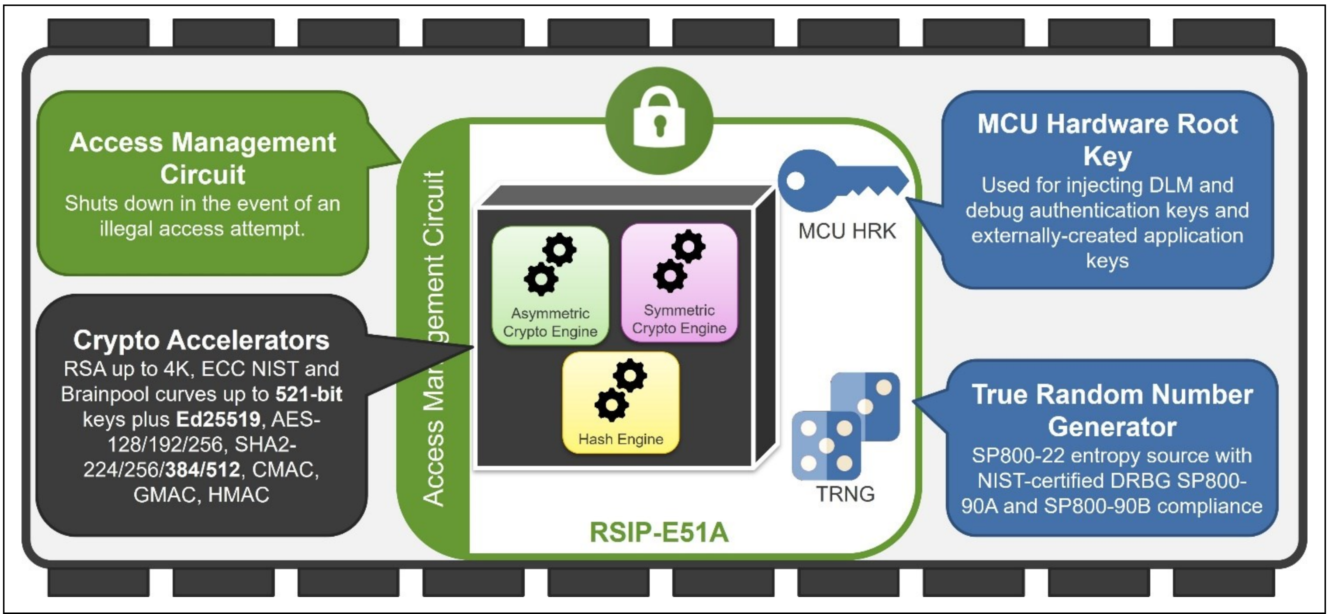

Figure 1. Security Engine RSIP-E51A

The Hardware Root Key (HRK) is not a single key that is physically stored. It is represented here to simplify the description of the concepts. The security engine has its own dedicated internal RAM for operations that deal with sensitive material such as plaintext keys. All crypto operations are physically isolated within the security engine. This RAM is not accessible outside the security engine.

The security engine has its own dedicated internal RAM, enabling all crypto operations to be physically isolated within the security engine. This, combined with advanced key handling capability, means that it is possible to implement applications where there is no plaintext key exposure on any CPU-accessible bus.

Secure key storage and usage is accomplished by storing application keys in wrapped format, encrypted by the MCU's Hardware Unique Key (HUK) and tagged with a Message Authentication Code (MAC). Since wrapped keys can only be unwrapped by the security engine within the specific MCU that wrapped them, the wrapping mechanism provides unclonable secure storage of application keys.

The security engine is packed full of cryptography features that users can leverage in higher-level solutions, providing the option to use hardware acceleration to reduce both execution time and power consumption. There are several different versions of security engine for Renesas RA MCUs. Be sure to check your use-case requirements to find the best-fit security engine for your application.

Table 1 summarizes the different security engines and their associated cryptographic functionalities.

Table 1. Security Engine Cryptographic Capabilities

Functions		RA8P1 RA8D2 RA8M2 RA8T2	RA8D1 RA8M1 RA8T1	RA4C1	RA4L1	RA6M4 RA6M5 RA4M2 RA4M3	RA6M1 RA6M2 RA6M3 RA6T1	RA6T2	RA4M1 RA4W1
Cryptographic Isolation									
Security Engines		RSIP-E50D	RSIP-E51A	RSIP-E31A	RSIP-E11A	SCE9	SCE7	SCE5_B	SCE5
Identity & Key Exchange **(Asymmetric)**									
RSA	Key Gen, Sign/Verify	Up to 4K	Up to 4K	-	-	Up to 4K	Up to 2K	-	-
ECC	Key Gen, ECDSA, ECDH	Up to 521 bits	Up to 521 bits	Up to 384 bits	Up to 256 bits	Up to 512 bits	Up to 384 bits	-	-
Ed25519	EdDSA	Y	Y	Y		-	-	-	-
Privacy **(Symmetric)**									
AES	ECB, CBC, CTR	128/192/256	128/192/256	128/256	128/256	128/192/256	128/192/256	128/256	128/256
	GCTR	128/192/256	128/192/256	128/256	-	128/192/256	128/192/256	-	-
	XTS	128/256	128/256	-	-	128/256	128/256	-	-
	CCM, GCM, CMAC, GMAC	128/192/256	128/192/256	128/256	128/256	128/192/256	128/192/256	128/256	128/256
	ChaCha20-Poly1305	Y	-	-	-	-	-	-	-
Data Integrity									
Hash	GHASH	Y	Y	Y	-	Y	Y	-	-
	HMAC	Y	Y	Y	Y	Y	Y	-	-
	SHA-2 (224/256)	Y	Y	Y	Y	Y	Y	-	-
	SHA-2 (384/512)	Y	Y	Y	-	-	-	-	-
	SHA3(224/256/ 384/512)	Y	-	-	-	-	-	-	-
	SHAKE (128/256)	Y	-	-	-	-	-	-	-
TRNG	HW Entropy, SP800-90B	Y	Y	Y	Y	Y	Y	Y	Y
Key Handling									
Wrapped	Confidentiality, authenticity	Y	Y	Y	Y	Y	Y	Y	Y
Plaintext	Legacy compatibility	Y	Y	Y	Y	Y	Y	Y	Y

RA security engines use a Hardware Unique Key (HUK) to secure the storage of application keys. For RSIP and SCE9, the MCU-unique HUK is a 256-bit random key. For SCE5_B, the HUK is a 128-bit random key. These HUKs are injected at the Renesas factory, and they are never exposed outside the security engine. This key is stored in a wrapped format using an MCU-unique key wrapping mechanism, ensuring that even if an attacker were able to extract the stored key, another MCU would not be able to use it. The MCU-unique HUK for SCE5 and SCE7 is a derived MCU-unique key. The derived HUK for SCE7 and SCE5 are never stored and is never exposed outside the security engine.

This application project uses the RA8M1, RA8P1, RA6M4, and RA6M3 MCUs to demonstrate the plaintext key injection using the FSP Crypto API as well as the PSA Certified™ Crypto API.

2. Plaintext User Key Injection

2.1 Plaintext User Key Injection Features

"Plaintext user key" refers to the fact that the user keys can be provided in plaintext format to the security engine. When the plaintext key is injected, the security engine wraps the plaintext key with the HUK and provides the wrapped key outside the security engine for storage.

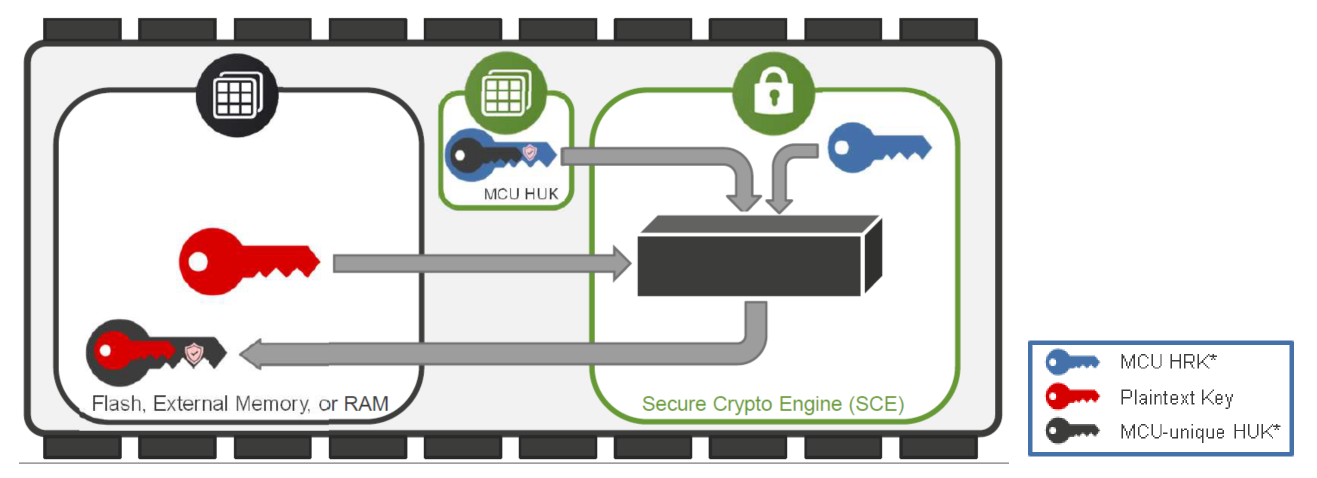

Figure 2. Plaintext Key Injection for SCE

This plaintext key injection process gives all security control of the keys to the product developer, which enables the developer to benefit from any existing secure key provisioning infrastructure. We do not recommend long-term storage of plaintext keys on the MCU; therefore, the RA Family MCUs have the capability to inject and securely store a plaintext key in a wrapped format by wrapping the key with the MCU HUK.

Getting the plaintext user key into the MCU RAM or flash in preparation for injection is out of the scope of this application project. Product developers can use their existing infrastructure to interface with the MCU based on their specific environment.

Note: This plaintext key injection procedure should be performed in a secure environment.

Key wrapping is a process that combines encryption with an integrity check. Wrapping with the MCU HUK provides not only authenticity but also cloning protection.

2.1.1 Advantages of Key Wrapping over Key Encryption

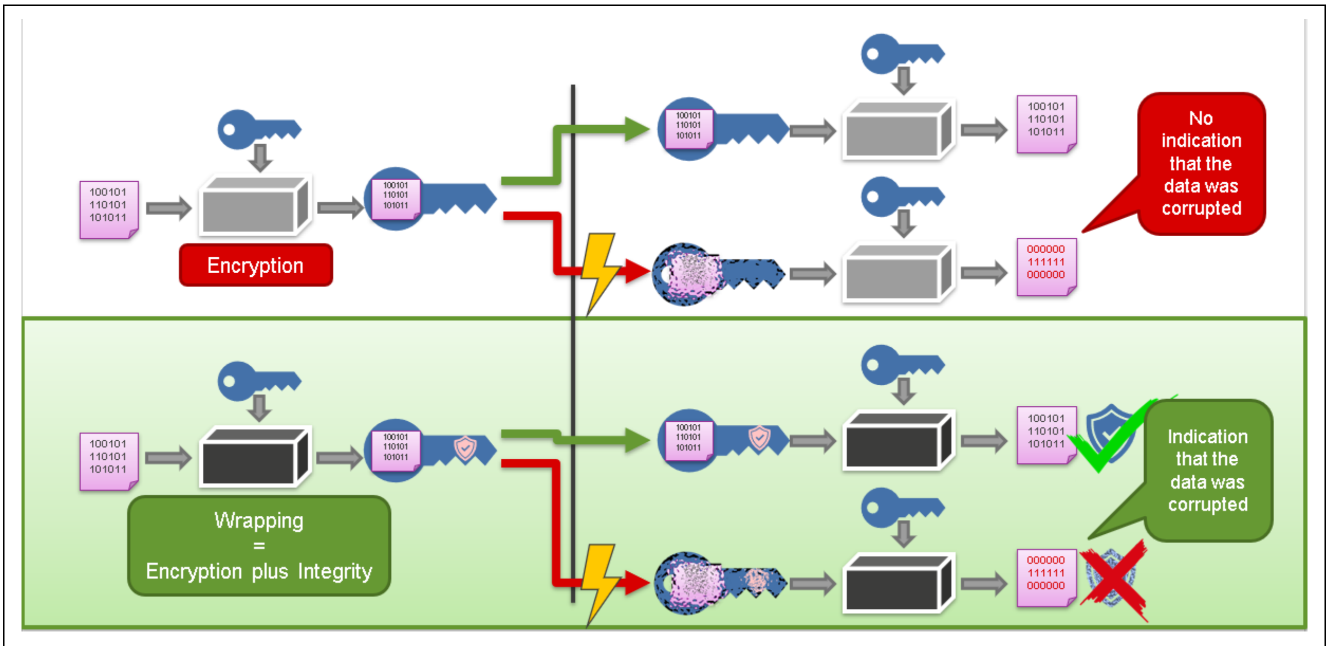

Figure 3. Key Wrapping vs. Key Encryption

It is important to understand the difference between wrapping and encrypting for secure asset storage. We will use symmetric encryption here to demonstrate.

When data is encrypted and sent to another recipient, if that recipient has the same key, they can decrypt the data. This results in a confidential exchange of information. However, what if there was a problem with the transmission of the encrypted data? If the recipient unknowingly receives corrupted information, the decryption algorithm will generate garbage data with no indication that the original data has been corrupted.

Wrapping solves this problem for us by adding an integrity-checking mechanism to the encrypted output.

2.1.2 Advantages of Key Wrapping using MCU HUK

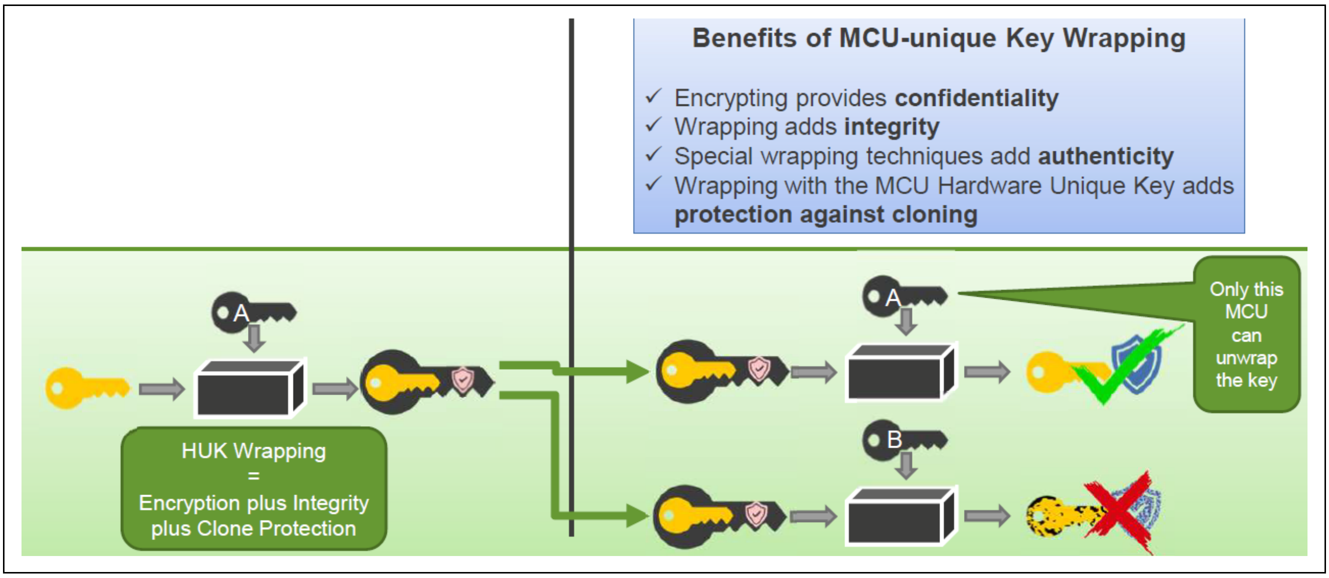

Figure 4. Key Wrapping Using the HUK

Using the MCU Hardware Unique Key to wrap the stored keys adds another protection feature – clone protection.

- If the wrapped key is transmitted or copied to another MCU, then that MCU's HUK will not be able to unwrap or decrypt the information, maintaining the security of the key.
- MCU-wrapped keys can only be unwrapped by the MCU that wrapped them:
  - The MCU's HUK is used as part of the wrapping algorithm.
  - Since the HUK is unique, no other MCU can unwrap the key.

Benefits

- Wrapped keys can be stored in non-secure memory.
- Even if all the MCU contents are copied onto another device, the keys cannot be utilized or exposed.

2.2 Plaintext User Key Injection Use Cases

This section summarizes several common use cases for key injection.

Case 1: Plaintext Key Injection During Production Provisioning/Programming

In this case, user keys are injected into the MCU based on the customer's existing or preferred method. The injected plaintext key is then injected using an MCU application-level code using the Renesas RA Family FSP. This case enables the injection of pre-generated keys, which should be performed in a secure environment. The FSP APIs used are demonstrated in the example projects included in this application project.

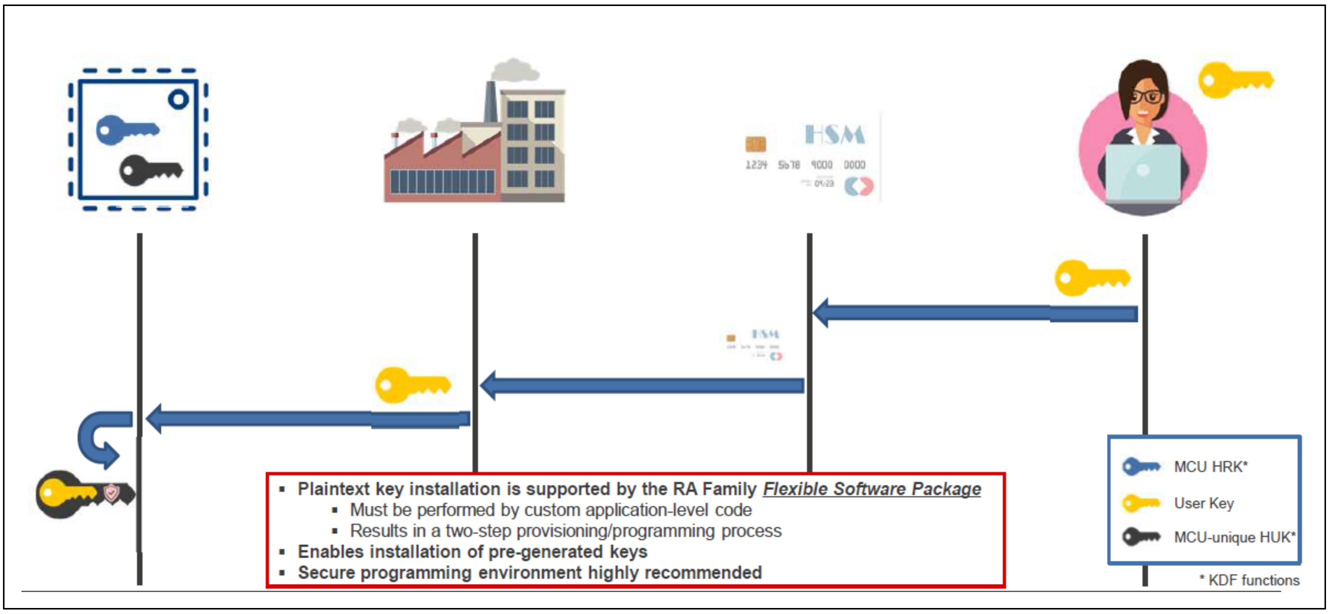

Figure 5. Plaintext Key Injection During Production

Case 2: Plaintext Key Injection Over Secure Communication Path

It is possible to provide a secure communication path for plaintext key injection. In this case, the plaintext key is securely transmitted and injected into the MCU. The MCU secure application software then injects the plaintext key, storing the key in a wrapped format. Solutions to support this use case are dependent on the implementation of the communication path. Customers can leverage the MCU operations provided for Case 1 to implement this solution.

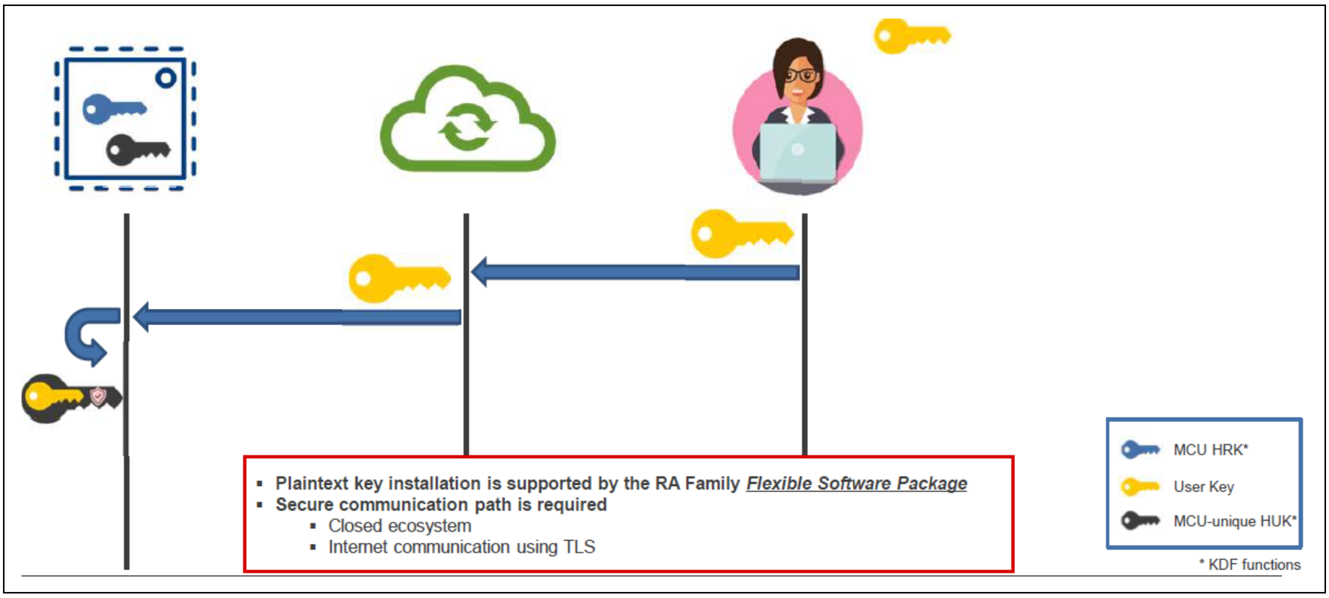

Figure 6. Plaintext Key Injection Over Secure Communication Path

Comparing Key Injection and MCU Key Generation

The following table summarizes the use case comparison between Key Injection and MCU Key Generation:

Table 2. Use Case Comparison with MCU-Generated Keys

Use Case	Plaintext Key Injection	MCU Key Generation (Wrapped Key)
Mass Production	Provides scalability, faster	Provides scalability, slower
Secure Environment	Recommended	Not required
Device Identity	Supported	Supported

3. Example Project for RA6M4 (SCE9) with AES User Key Handling

The hardware features of SCE9 are accessed through the FSP driver r_sce, which can access the key injection APIs. For most application developments, developers can use the middleware PSA Certified Crypto API implementation layer to interface with the SCE9. However, the key injection process does not map to PSA Certified Crypto APIs; therefore, r_sce key injection-related APIs must be used directly.

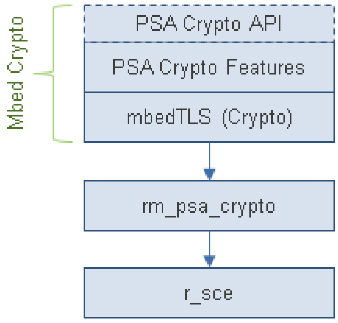

Figure 7. Crypto Stacks

Using PSA Crypto with TrustZone® needs some special handling compared with other drivers. Unlike other FSP drivers, the PSA Crypto module cannot be added as a Non-Secure Callable module. The reason for this is that to achieve the security objective of controlling access to protected keys, both the PSA Crypto code and the keys must be placed in the Secure region. The PSA Certified Crypto API requires access to the keys directly during initialization and later through a key handle. Therefore, the PSA Crypto module should reside in the Secure region.

To provide services to the non-secure region, you need to create application-specific, user-defined Non-Secure Callable (NSC) APIs in the Secure region. Proper security considerations can be implemented in the Non-Secure Callable API to limit access to the NSC APIs.

The need for the non-secure region to access cryptographic service in the Secure region varies from application to application. The Non-Secure Callable API provided in this example project can be adjusted to fit other customer applications. It is not advised to use the example as-is for a real-world secure application.

Figure 8 is the high-level software block diagram of the example project provided in this application project.

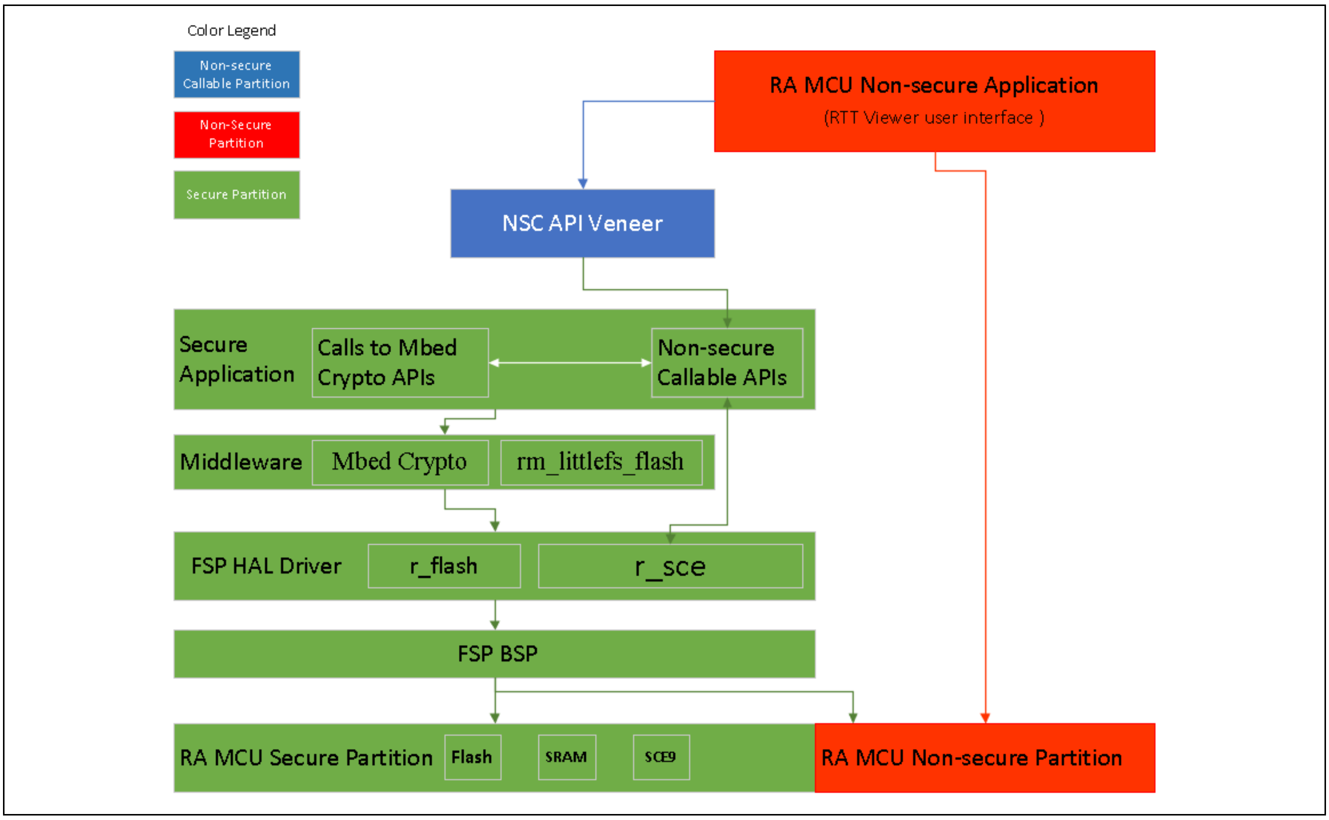

Figure 8. Software Block Diagram

The Non-Secure Callable APIs are defined in `example_AES.h` file. These APIs are explained as follows:

- **`BSP_CMSE_NONSECURE_ENTRY bool init_lfs(void)`**
  Initializes the LittleFS system: formatted and mounted.
- **`BSP_CMSE_NONSECURE_ENTRY bool psacrypto_AES256CBC_example_NIST (void)`**
  Allows the non-secure project to initiate new AES key creation by injecting a 256-bit AES plaintext key (using a set of NIST vectors) as a wrapped key. Once the plaintext user key is injected into the MCU, the SCE9 driver is used to convert the plaintext key into wrapped key format by wrapping the plaintext key using the HUK. The plaintext key will be erased immediately after the conversion. The wrapped AES key is further imported into the PSA key storage system and stored in the data flash for user application usage.
  Then the example project uses this injected key to perform encryption and decryption operation.

3.1 FSP API Used in the Plaintext Key Wrap

The API shown below performs the initial AES256 key wrapping. This API supports both secure key and plaintext key APIs. Notice that some arguments are ignored in plaintext key wrapping.

```
/*******************************************************************************************************************//**
 * This API generates 256-bit AES key within the user routine.
 *
 * @param[in]     key_type                            Selection key type when generating wrapped key
 *                                                    (0: for encrypted key, 1: for plain key)
 * @param[in]     wrapped_user_factory_programming_key Wrapped user factory programming key by the Renesas Key Wrap Service.
 *                                                    When key_type is 1 as plain key, this is not required and
 *                                                    any value can be specified.
 * @param[in]     initial_vector                      Initialization vector when generating encrypted_key.
 *                                                    When key_type is 1 as plain key, this is not required and
 *                                                    any value can be specified.
 * @param[in]     encrypted_key                       Encrypted user key and MAC appended
 * @param[in,out] wrapped_key                         256-bit AES wrapped key
 *
 * @retval FSP_SUCCESS                      Normal termination.
 * @retval FSP_ERR_UNSUPPORTED              API not supported.
 * @return If an error occurs, the return value will be as follows.
 *         * FSP_ERR_CRYPTO_SCE_FAIL Internal I/O buffer is not empty.
 *         * FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT A resource conflict occurred because a hardware resource needed.
 *
 * @note The pre-run state is SCE Enabled State.
 *       After the function runs the state transitions to SCE Enabled State.
 **********************************************************************************************************************/
fsp_err_t R_SCE_AES256_InitialKeyWrap (const uint8_t * const          key_type,
                                       const uint8_t * const          wrapped_user_factory_programming_key,
                                       const uint8_t * const          initial_vector,
                                       const uint8_t * const          encrypted_key,
                                       sce_aes_wrapped_key_t * const  wrapped_key)
{
```

Figure 9. AES256 KeyWrap API

3.2 Import and Compile the Example Project

Follow the FSP User's Manual section *Importing an Existing Project into e² studio* to import the Secure and Non-Secure Projects into the workspace and compile in the order shown below:

1. Expand the secure project `plaintext_key_injection_ek_ra6m4_s` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the secure project. The project should be built with no errors.
   Note that there are third-party software warnings.
2. Expand the non-secure project `plaintext_key_injection_ek_ra6m4_ns` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the non-secure project.

3.3 Setting up the Hardware

Establish the following connections:

- EK-RA6M4 jumper setting: J6 closed, J9 open. For other jumpers, keep the out-of-box setting.
- USB cable connected between J10 and the development PC to provide power and debugging capability using the on-board debugger.

Initialize the MCU using Renesas Device Partition Manager

This step is optional but recommended. Prior to downloading the example application, we recommend initializing the device into the Secure Software Development (SSD) state. Flash content that is not permanently locked down will be erased during this process. This is particularly helpful if the device was previously used in the Non-Secure Software Development (NSECSD) state or if certain flash blocks are locked up temporarily.

Note: Power cycle the board prior to working with the **Renesas Device Partition Manager** after a debug session if using J-Link as the connection interface.

Open the **Renesas Device Partition Manager**. With the e² studio IDE, click the **Run** tab, then select **Renesas Debug Tools > Renesas Device Partition Manager**.

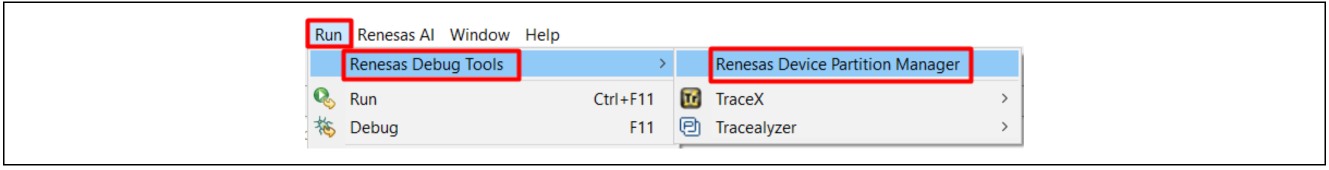

Figure 10. Open Renesas Device Partition Manager

Next, check the **Initialize device**, choose **J-Link** as the connection method, then click **Run**.

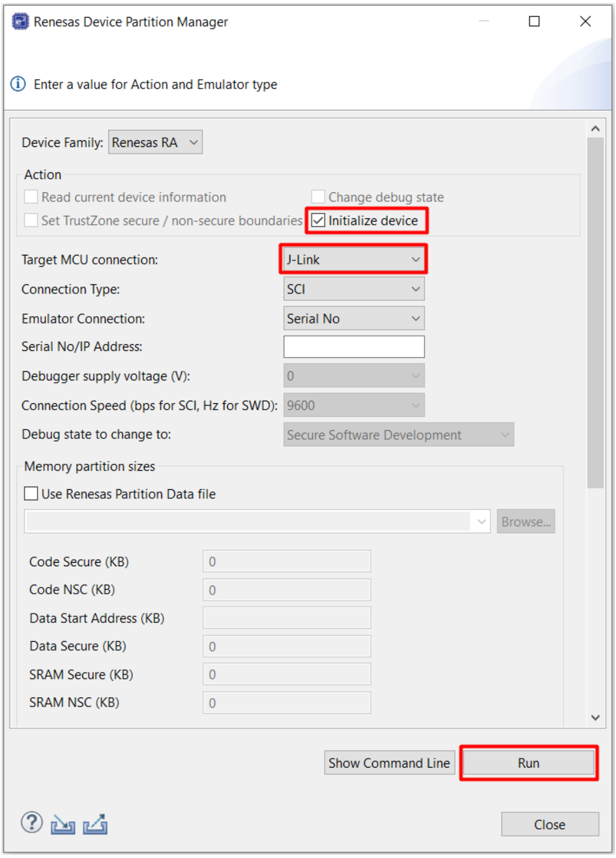

Figure 11. Initialize RA6M4 Using Renesas Device Partition Manager

3.4 Running the Example Project

To run the application, right-click on `plaintext_key_injection_ra6m4_ns` and select **Debug As > Renesas GDB Hardware Debugging**.

Note that prior to the application execution, the Implementation Defined Attribute Unit (IDAU) regions will be set up to assume the values through the debugger interaction with the MCU bootloader.

Both the Secure and Non-Secure projects are now loaded, and the debugger should be paused in the `Reset_Handler()` at the `SystemInit()` call in the Secure project.

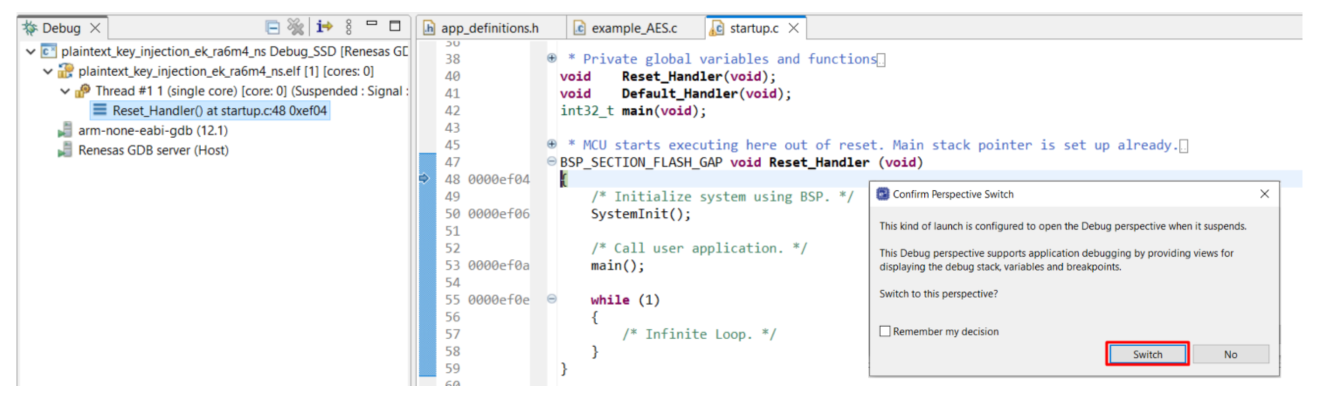

Figure 12. Secure Project Reset Handler

Click **Switch** if the **Confirm Perspective Switch** window pops up. Click twice to run the project.

Next, launch **J-Link RTT Viewer** V8.44.

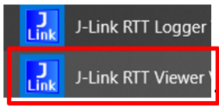

Figure 13. Launch J-Link RTT Viewer

Select **Existing Session** as the connection type. Click on the button and scroll down to **Renesas** to find the correct device, **R7FA6M4AF**. Also, the RTT Control Block for the **Search Range** should be set up. Set the search range to 0x20000000 0x10000, and then click **OK** to start the RTT Viewer.

Note: The Search Size 0x10000 is based on this example application project. If your application uses the RTT Viewer in the Non-Secure region and there is a large secure binary, you need to increase the Search Size to cover the Non-Secure project SRAM regions.

If the host PC has more than one J-Link debugger connected to the PC, set the **Serial No** (by default, **Serial No** is set to 0).

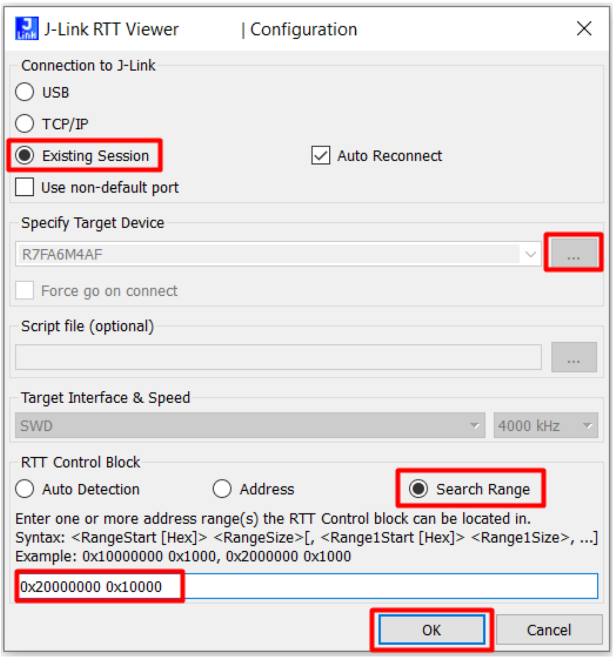

Figure 14. Configure the RTT Viewer for EK-RA6M4

Click **OK** and observe the following output.

```
**********************************************************************
 Utilizing the AES service which resides in secure region of RA6M4
**********************************************************************

 LFS initialization successful

 Plaintext key injected as wrapped AES key successfully.

 Encryption is successful. Encrypted data matches NIST cipher text.

 Decryption is successful. Decrypted data matches NIST plaintext message.
```

Figure 15. RA6M4 Plaintext Key Injection Demonstration

4. Example Project for RA6M3 (SCE7) AES User Key Handling

See Figure 7 for the crypto stack used for this example project. From a high-level understanding, they are identical.

4.1 Import and Compile the Example Project

Follow the FSP User's Manual section *Importing an Existing Project into e² studio* to import the example project `plaintext_key_injection_ek_ra6m3` to a workspace.

Expand the project `plaintext_key_injection_ek_ra6m3` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the project. The project should be built with no errors.

4.2 FSP API Used in the Plaintext Key Wrap

The API shown in this section performs the initial AES128 key wrapping (like the AES256 key wrapping API). This API supports both secure key and plaintext key APIs. Notice that some arguments are ignored in plaintext key wrapping.

```
/*****************************************************************************************************************//**
 * This API generates 128-bit AES key within the user routine.
 *
 * @param[in]     key_type                            Selection key type when generating wrapped key
 *                                                    (0: for encrypted key, 1: for plain key)
 * @param[in]     wrapped_user_factory_programming_key Wrapped user factory programming key by the Renesas Key Wrap Service.
 *                                                    When key_type is 1 as plain key, this is not required and
 *                                                    any value can be specified.
 * @param[in]     initial_vector                      Initialization vector when generating encrypted_key.
 *                                                    When key_type is 1 as plain key, this is not required and
 *                                                    any value can be specified.
 * @param[in]     encrypted_key                       Encrypted user key and MAC appended
 * @param[in,out] wrapped_key                         128-bit AES wrapped key
 *
 * @retval FSP_SUCCESS                          Normal termination.
 * @retval FSP_ERR_UNSUPPORTED                  API not supported.
 * @return If an error occurs, the return value will be as follows.
 *         * FSP_ERR_CRYPTO_SCE_FAIL Internal I/O buffer is not empty.
 *         * FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT A resource conflict occurred because a hardware resource needed.
 *
 * @note The pre-run state is SCE Enabled State.
 *       After the function runs the state transitions to SCE Enabled State.
 ******************************************************************************************************************/
fsp_err_t R_SCE_AES128_InitialKeyWrap (const uint8_t * const         key_type,
                                       const uint8_t * const         wrapped_user_factory_programming_key,
                                       const uint8_t * const         initial_vector,
                                       const uint8_t * const         encrypted_key,
                                       sce_aes_wrapped_key_t * const wrapped_key)
{
```

Figure 16. AES128 KeyWrap API

4.3 Setting up the Hardware

Connect J10 from EK-RA6M3 to the development PC to provide power and debugging capability using the onboard debugger.

4.4 Running the Example Project

To run the application, right-click on `plaintext_key_injection_ek_ra6m3` and select **Debug As > Renesas GDB Hardware Debugging**.

Click **Switch** if the **Confirm Perspective Switch** window pops up. Click  twice to run the project.

Next, launch **J-Link RTT Viewer** V8.44.

Figure 17. Launch J-Link RTT Viewer

Configure the RTT Viewer as shown in Figure 18.

Users need to search for **"_SEGGER_RTT"** in the **Debug > plaintext_key_injection_ek_ra6m3.map** file to obtain the address of the RTT Control Block.

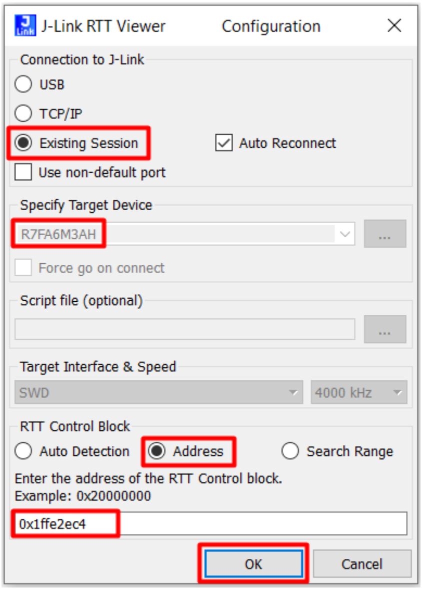

Figure 18. Configure the RTT Viewer for EK-RA6M3

Click OK and observe the RTT Viewer output as shown in Figure 19.

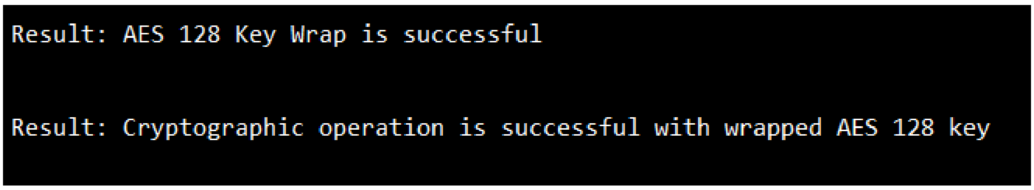

Figure 19. Expected Execution Result of the RA6M3 Example Project

5. Example Project for RA8M1 (RSIP – E51A) AES User Key Injection

5.1 Import and Compile the Example Project

Follow the FSP User's Manual section *Importing an Existing Project into e² studio* to import the example project `plaintext_key_injection_ra8m1` to a workspace.

Expand the project `plaintext_key_injection_ra8m1` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the project. The project should be built with no errors.

5.2 FSP API Used in the Plaintext Key Wrap

The API shown below performs the initial AES256 key wrapping. This API supports both secure key and plaintext key APIs. Notice that some arguments are ignored in plaintext key wrapping.

```
/*******************************************************************************************************************//**
 * This API generates 256-bit AES key within the user routine.
 *
 * @param[in]  key_injection_type                   Selection key injection type when generating wrapped key
 * @param[in]  p_wrapped_user_factory_programming_key  Wrapped user factory programming key by the Renesas Key Wrap Service.
                                                    When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_initial_vector                     Initialization vector when generating encrypted key.
 *                                                  When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_user_key                           User key. If key injection type is not plain, it is encrypted and MAC appended
 * @param[out] p_wrapped_key                        256-bit AES wrapped key
 *
 * @retval FSP_SUCCESS                         Normal termination.
 * @return If an error occurs, the return value will be as follows.
 *          * FSP_ERR_ASSERTION                    A required parameter is NULL.
 *          * FSP_ERR_CRYPTO_SCE_FAIL              MAC anomaly detection.
 *          * FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT Resource conflict.
 *          * FSP_ERR_CRYPTO_UNKNOWN               An unknown error occurred.
 *          * FSP_ERR_INVALID_STATE                Internal state is illegal.
 *
 * @note The pre-run state is RSIP Enabled State.
 *       After the function runs the state transitions to RSIP Enabled State.
 **********************************************************************************************************************/
fsp_err_t R_RSIP_AES256_InitialKeyWrap (rsip_key_injection_type_t const key_injection_type,
                                        uint8_t const * const          p_wrapped_user_factory_programming_key,
                                        uint8_t const * const          p_initial_vector,
                                        uint8_t const * const          p_user_key,
                                        rsip_aes_wrapped_key_t * const p_wrapped_key)
```

Figure 20. RSIP AES256 Key Wrap API

5.3 Setting up the Hardware

Connect J10 from EK-RA8M1 to the development PC to provide power and debugging capability using the onboard debugger.

5.4 Running the Example Project

To run the application, right-click on `plaintext_key_injection_ra8m1` and select **Debug As > Renesas GDB Hardware Debugging**.

Click **Switch** if the **Confirm Perspective Switch** window pops up. Click  twice to run the project.

Next, launch **J-Link RTT Viewer** V8.44.

Figure 21. Launch J-Link RTT Viewer

Configure the RTT Viewer as shown in Figure 22.

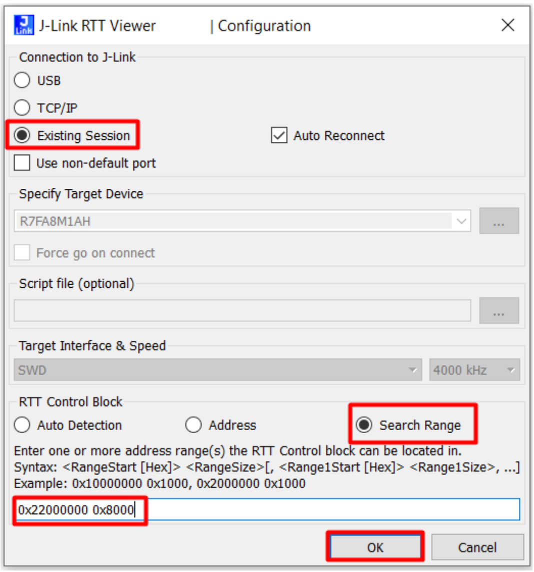

Figure 22. Configure the RTT Viewer for EK-RA8M1

Click OK and observe the RTT Viewer output as shown in Figure 23.

```
LFS initialization successful.
R_RSIP_AES256_InitialKeyWrap is successful.
Encryption operation is successful.
Decryption operation is successful.

RSIP-E51A AES key injection operation is successful.
```

Figure 23. Expected Execution Result of the RA8M1 Example Project

6. Example Project for RA8P1 (RSIP-E50D) AES and ECC User Key Injection

This section includes two example projects demonstrating AES and ECC key injection. CPU0 releases CPU1 from reset after completing its cryptographic operation. The RSIP-E50D is not used concurrently by CPU0 and CPU1.

- The project running on CPU0 demonstrates the injection of an ECC secp256r1 private key.
- The project running on CPU1 demonstrates the injection of an AES256 key.

6.1 Import and Compile the Example Project

Follow the FSP User's Manual section *Importing an Existing Project into e² studio* to import the example project `plaintext_key_injection_ra8p1` into a workspace, and then compile it in the order shown below:

1. Expand the project `ra_plaintext_key_injection_ra8p1_CPU0` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the project. The project should be built with no errors.
2. Expand the project `ra_plaintext_key_injection_ra8p1_CPU1` and double-click `configuration.xml` to launch the configurator. Click **Generate Project Content**, then build the project. The project should be built with no errors.

6.2 FSP API Used in the Plaintext Key Wrap

The APIs shown below perform the initial wrapping process for both ECC secp256r1 private key and AES256 key. These APIs support both secure key and plaintext key operations. Note that some arguments are ignored in plaintext key wrapping.

```
/*******************************************************************************************************************//**
 * This API generates 256-bit ECC private key within the user routine.
 *
 * @param[in]  key_injection_type                   Selection key injection type when generating wrapped key
 * @param[in]  p_wrapped_user_factory_programming_key Wrapped user factory programming key by the Renesas Key Wrap Service.
 *                                                  When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_initial_vector                     Initialization vector when generating encrypted key.
 *                                                  When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_user_key                           User key. If key injection type is not plain, it is encrypted and MAC appended
 * @param[out] p_wrapped_key                        256-bit ECC wrapped private key
 *
 * @retval FSP_SUCCESS                          Normal termination.
 * @return If an error occurs, the return value will be as follows.
 *         * FSP_ERR_ASSERTION                  A required parameter is NULL.
 *         * FSP_ERR_CRYPTO_SCE_FAIL            MAC anomaly detection.
 *         * FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT Resource conflict.
 *         * FSP_ERR_CRYPTO_UNKNOWN             An unknown error occurred.
 *         * FSP_ERR_INVALID_STATE              Internal state is illegal.
 *
 * @note The pre-run state is RSIP Enabled State.
 *       After the function runs the state transitions to RSIP Enabled State.
 **********************************************************************************************************************/
fsp_err_t R_RSIP_ECC_secp256r1_InitialPrivateKeyWrap (rsip_key_injection_type_t const        key_injection_type,
                                                      uint8_t const * const                  p_wrapped_user_factory_programming_key,
                                                      uint8_t const * const                  p_initial_vector,
                                                      uint8_t const * const                  p_user_key,
                                                      rsip_ecc_private_wrapped_key_t * const p_wrapped_key)
```

Figure 24. RSIP ECC secp256r1 Private Key Wrap API

```
/***********************************************************************************************************************//**
 * This API generates 256-bit AES key within the user routine.
 *
 * @param[in]  key_injection_type                    Selection key injection type when generating wrapped key
 * @param[in]  p_wrapped_user_factory_programming_key Wrapped user factory programming key by the Renesas Key Wrap Service.
 *                                                   When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_initial_vector                      Initialization vector when generating encrypted key.
 *                                                   When key injection type is plain, this is not required and any value can be specified.
 * @param[in]  p_user_key                            User key. If key injection type is not plain, it is encrypted and MAC appended
 * @param[out] p_wrapped_key                         256-bit AES wrapped key
 *
 * @retval FSP_SUCCESS                         Normal termination.
 * @return If an error occurs, the return value will be as follows.
 *          * FSP_ERR_ASSERTION                   A required parameter is NULL.
 *          * FSP_ERR_CRYPTO_SCE_FAIL             MAC anomaly detection.
 *          * FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT Resource conflict.
 *          * FSP_ERR_CRYPTO_UNKNOWN              An unknown error occurred.
 *          * FSP_ERR_INVALID_STATE               Internal state is illegal.
 *
 * @note The pre-run state is RSIP Enabled State.
 *       After the function runs the state transitions to RSIP Enabled State.
 **********************************************************************************************************************/
fsp_err_t R_RSIP_AES256_InitialKeyWrap (rsip_key_injection_type_t const key_injection_type,
                                        uint8_t const * const          p_wrapped_user_factory_programming_key,
                                        uint8_t const * const          p_initial_vector,
                                        uint8_t const * const          p_user_key,
                                        rsip_aes_wrapped_key_t * const p_wrapped_key)
```

Figure 25. RSIP AES256 Wrap API

6.3 Setting up the Hardware

Connect J10 from EK-RA8P1 to the development PC to provide power and debugging capability using a type A to type C USB cable.

To create a clean environment for starting the example projects. Users should initialize the MCU using the **Renesas Device Partition Manager** by following these steps:

- Open the Renesas Device Partition Manager. With the e[2] studio IDE, click the **Run** tab, then select **Renesas Debug Tools > Renesas Device Partition Manager**, as shown in Figure 26.

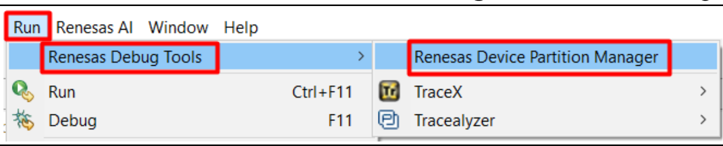

Figure 26. Open Renesas Device Partition Manager

- Next, check the **Initialize device**, choose **J-Link** as the connection method, select **SWD** (or **SCI**) as the connection type, and then click **Run**.

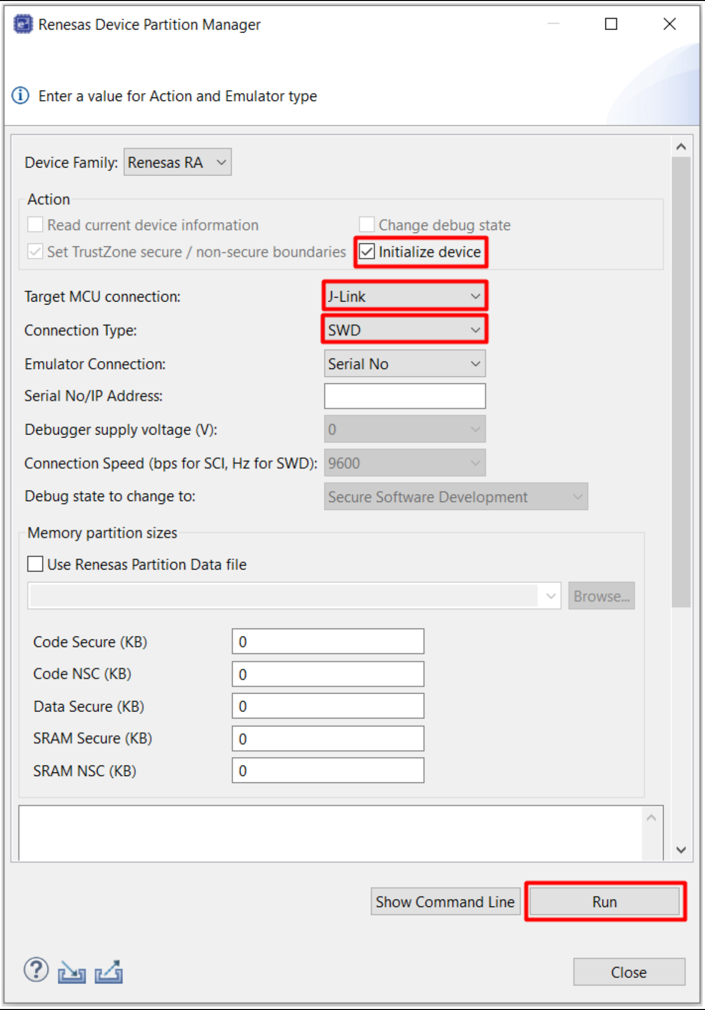

Figure 27. Initialize RA8P1 Using Renesas Device Partition Manager

In addition, this application project on RA8P1 uses the LittleFS file system on external SPI flash. To enable external flash operation, the third switch on SW4 (on the board) must be turned off.

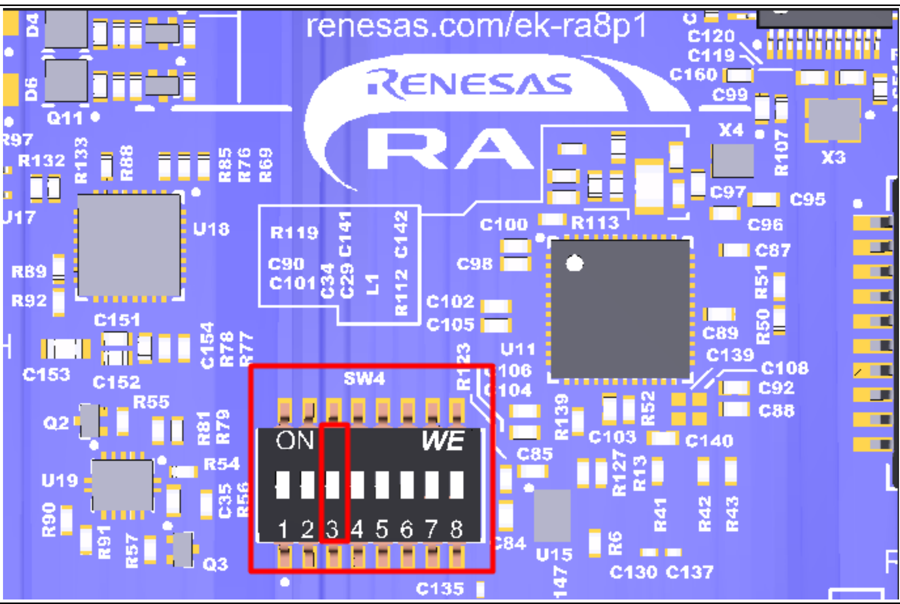

Figure 28. Turn off the third switch on SW4

6.4 Running the Example Project

To run the application, right-click on `ra_plaintext_key_injection_ra8p1_CPU0` and select **Debug As > Debug Configurations**, then select the **Launch Group > ra_plaintext_key_injection_ra8p1_CPU1 Debug_Multicore Launch Group** to debug the dual core, as shown in Figure 29.

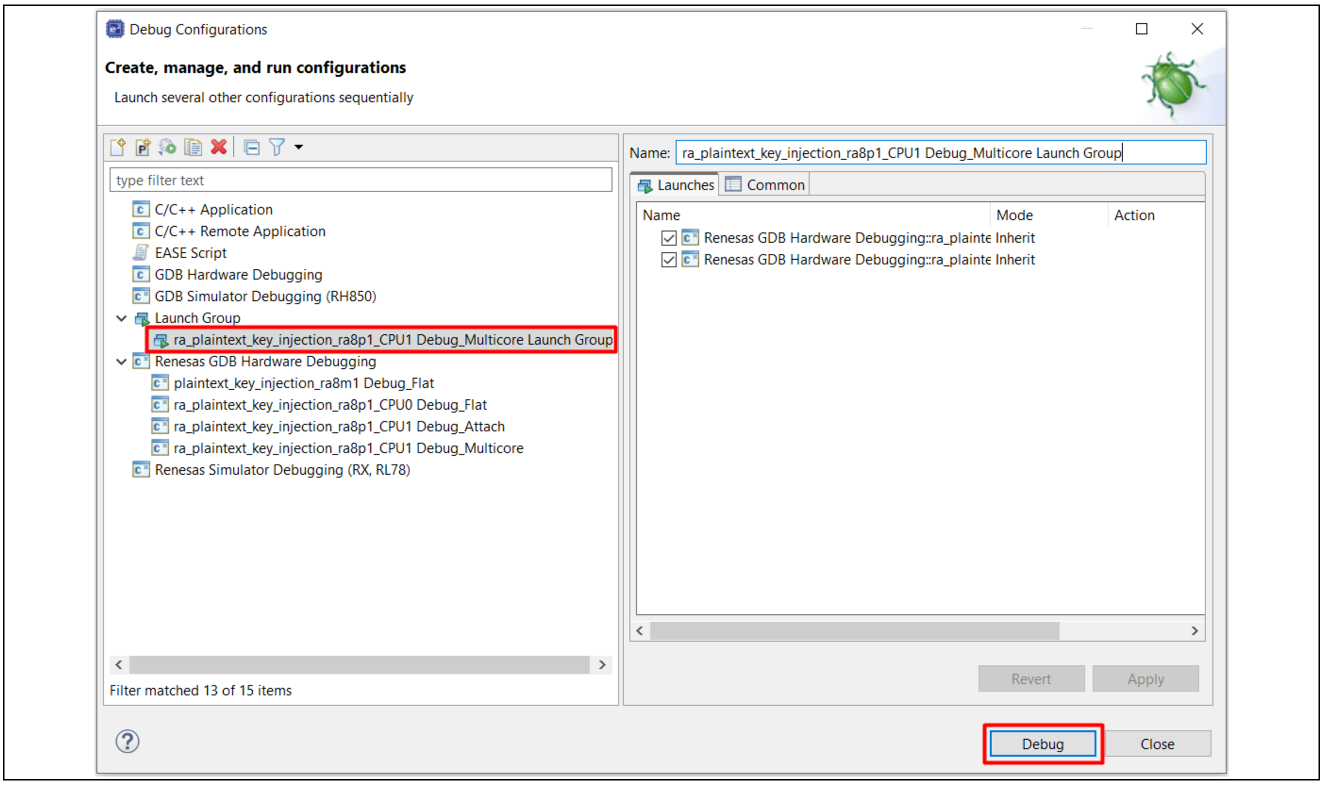

Figure 29. Debug the dual core

Click  three times to run the project. At this point, both the projects on CPU0 and CPU1 are loaded.

Next, launch **J-Link RTT Viewer** V8.44.

Figure 30. Launch J-Link RTT Viewer

Configure the RTT Viewer, as shown in Figure 31.

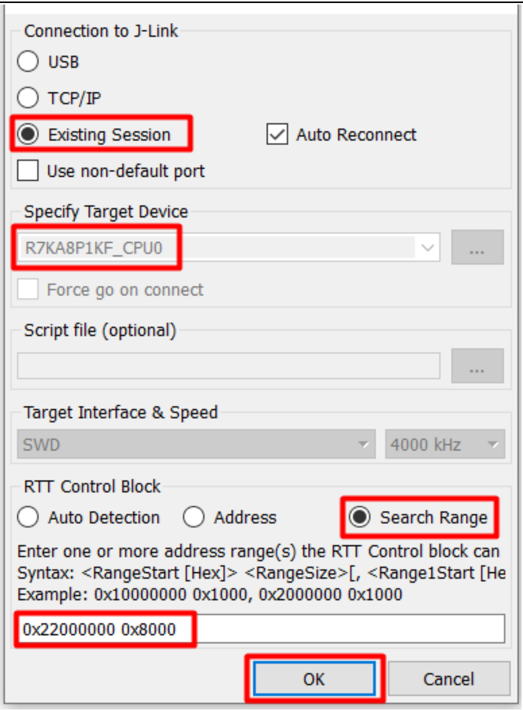

Figure 31. Configure the RTT Viewer for EK-RA8P1 running on CPU0

Click OK and observe the RTT Viewer output as shown in Figure 32.

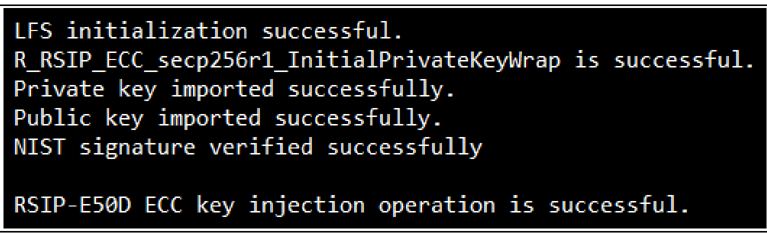

Figure 32. Expected Execution Result of the RA8P1 on CPU0

To observe the output of the example project on CPU1, users need to disconnect the RTT Viewer CPU0. And then, configure the RTT Viewer on CPU1, as shown in Figure 33.

Users need to search for **"_SEGGER_RTT"** in the **Debug > ra_plaintext_key_injection_ra8p1_CPU1.map** file to obtain the address of the RTT Control Block.

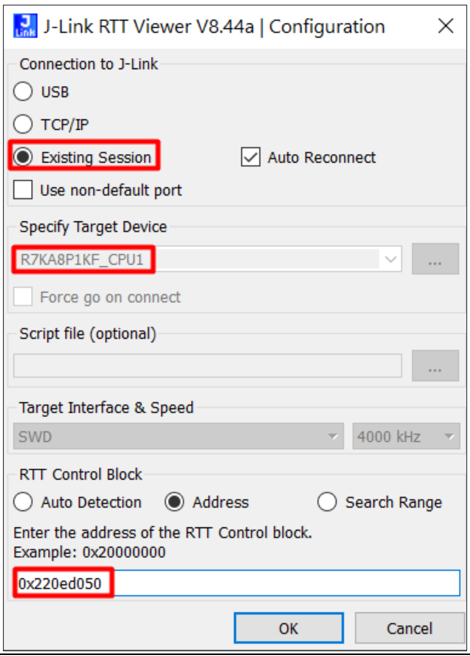

Figure 33. Configure the RTT Viewer for EK-RA8P1 running on CPU1

Click OK and observe the RTT Viewer output as shown in Figure 34.

```
LFS initialization successful.
R_RSIP_AES256_InitialKeyWrap is successful.
Encryption operation is successful.
Decryption operation is successful.

RSIP-E50D AES key injection operation is successful.
```

Figure 34. Expected Execution Result of the RA8P1 on CPU1

7. Glossary

Term	Meaning
HSM	A Hardware Security Module (HSM) is a physical computing device that safeguards and manages digital keys and performs encryption and decryption functions for digital signatures, strong authentication, and other cryptographic functions.
HRK	Hardware Root Key is a secret key residing in the security engine that is common for each MCU.
Unique ID	A Unique Identification value, unique to each individual RA Family MCU, is stored inside the MCU.
MAC	Message Authentication Code is a short piece of information used to authenticate a message to confirm that the message came from the stated sender (its authenticity) and has not been changed. A cryptographic MAC protects both a message's data integrity and its authenticity by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

8. References

1. Renesas RA Family MCU Device Lifecycle Management Key Injection (R11AN0469)
2. Renesas RA Family MCU Secure Key Injection and Update (R11AN0496)
3. Renesas RA Family MCU Security Design with TrustZone® using Cortex-M33 (R11AN0467)
4. Renesas RA Family MCU Security Design with TrustZone® using Cortex-M85 (R11AN0897)
5. Renesas RA Family Security Engine Operational Modes Application Note (R11AN0498)
6. Renesas RA Family Developing with RA8 Dual Core MCU (R01AN7881)

9. Website and Support

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

EK-RA6M4 Resources	renesas.com/ra/ek-ra6m4
EK-RA6M3 Resources	renesas.com/ra/ek-ra6m3
EK-RA8M1 Resources	renesas.com/ra/ek-ra8m1
EK-RA8P1 Resources	renesas.com/ek-ra8p1
RA Product Information	renesas.com/ra
Flexible Software Package (FSP)	renesas.com/ra/fsp
RA Product Support Forum	renesas.com/ra/forum
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Dec.2.20	-	First release document.
1.10	Dec.20.20	-	Added missing graph.
1.20	Nov.11.21	-	Minor updates.
1.30	Dec.07.21	-	Fix Wrap Key API Call bug.
1.40	Nov.11.22	-	Changed the document title from "Installing and Utilizing the Cryptographic User Keys using SCE9" to "Injecting Plaintext User Keys" and added SCE7 support.
2.00	Jan.10.24	-	Updated to FSP v5.1.0.
2.10	Nov.06.24	-	Updated to FSP v5.6.0.
2.20	June.27.25	-	Updated to FSP v6.0.0 and added the RSIP-E50D support on RA8P1 Dual Core.
2.21	Nov.21.25	1	Added RSIP-E50D support on RA8x2

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

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

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

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

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

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

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

Corporate Headquarters

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

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.