RX Family

Renesas Secure IP Compatibility Mode Firmware Integration Technology

Introduction

This document describes how to use the software driver for the Renesas Secure IP (RSIP) module on RX family microcontrollers. This software driver is referred to as the RSIP compatibility mode driver (RSIP CM driver).

The RSIP CM driver is provided as a firmware integration technology (FIT) module. Refer to the Web page at the URL below for an overview of the FIT concept.

https://www.renesas.com/en/software-tool/fit

The RSIP CM driver provides API functions for executing the cryptographic functionality summarized in Table 1.

Target Devices

RX261 Group microcontrollers

Table 1 Cryptographic Algorithms for the RX RSIP CM Driver

Type of Cryptography		Algorithms
Asymmetric key cryptography	Signature generation/verification	ECDSA (secp256r1, brainpoolP256r1, secp256k1): RFC 6979
	Key generation	secp256r1, brainpoolP256r1, secp256k1
Symmetric key cryptography	AES	AES (128-/256-bit) ECB/CBC/CTR: FIPS 197, SP 800-38A
Hashing	SHA	SHA224, SHA256: FIPS 180-4
Authenticated encryption with associated data (AEAD)		GCM/CCM: FIPS 197, SP 800-38C, SP 800-38D
Message authentication		CMAC (AES): FIPS 197, SP 800-38B GMAC: RFC 4543 HMAC (SHA): RFC 2104
Pseudo-random bit generation		SP 800-90A
Random number generation		Tested with SP 800-90B.

Note

RFC 2104: HMAC: Keyed-Hashing for Message Authentication (rfc-editor.org)
RFC 4543: The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH (rfc-editor.org)
RFC 6979 - Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) (ietf.org)
NIST SP 800-38A, Recommendation for Block Cipher Modes of Operation Methods and Techniques
NIST SP 800-38-B Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication (nist.gov)
NIST SP 800-38D, Recommendationfor Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC
NIST SP800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Lograrithm Cryptography (nist.gov)
NIST SP800-56C: Recommendation for Key-Derivation Methods in Key-Establishment Schemes (nist.gov)
NIST SP800-90A: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf
NIST SP800-90B: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf
RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm (rfc-editor.org)
FIPS 180-4: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

Contents

1. Overview

1.1 Terminology

Terms used in this document are defined below.

Table 1-1 Descriptions of Terms

Term	Description
Key injection	Injecting a wrapped key into the device at the factory
User key	An encryption key in plaintext for use by the user. User keys for AES and HMAC are symmetric keys. User keys for ECC are asymmetric keys.
Encrypted key	Key information generated by using a UFPK to append the MAC value to a user key and then encrypting the result. The value of the encrypted key generated from the same user key is common to devices of a given type that have users in common.
Wrapped key	Data generated by using key injection to convert an encrypted key into the format that allows the use of the given key by the RSIP module. The wrapped keys have been wrapped with the use of HUKs, so their values are unique to each device even if they are from the same encrypted key.
UFPK	User factory programming key A user-set keyring used to generate an encrypted key from a user key during key injection. The UFPK is not used on the device.
W-UFPK	Wrapped UFPK Key information generated by using an HRK, which is available from the “Renesas Key Wrap Service” Web site, to wrap a UFPK. This is decrypted by using the HRK within the RSIP to obtain the UFPK, which is then used.
Hardware root key (HRK)	A common encryption key, which only exists in the RSIP and the secure rooms in Renesas.
Hardware unique key (HUK)	A device-specific encryption key that is derived within the RSIP and used to protect key data.
Renesas Key Wrap Service	Web site for use in generating W-UFPKs from UFPKs https://dlm.renesas.com/keywrap/toEnglish

1.2 Overview

The RX RSIP CM driver realizes hardware acceleration by using the RSIP in implementing the PSA Certified Crypto API (hereinafter referred to as PSA Crypto API). The RX RSIP CM driver provides optimized performance and unlimited secure key storage while enabling easy integration with existing systems and third-party software and solutions.

The RX RSIP CM driver is combined with the mbedTLS version 3.6.2 library, which is compliant with the PSA Crypto API 1.0 specifications. For details on the PSA Crypto API specifications, refer to the Arm documentation on the Web page at the following URL.

https://armmbed.github.io/mbed-crypto/psa/#applicationprogramming-interface

- Hashing

  SHA224 calculation

  SHA256 calculation

- MAC

  HMAC

  CMAC

- AES

  Key bits: 128, 256

  Generation of a plaintext key

  Generation of a wrapped key

  Encryption and decryption without padding or with PKCS7-type padding

  CBC, CTR, CCM, or GCM mode

- ECC

  Elliptic curve

    secp256r1

    secp256k1

    brainpoolP256r1

  Generation of a wrapped key

  Signature and verification

- Generation of random numbers

The following describes the features of the RX RSIP CM driver.

- Plaintext keys are usable.
  This ensures compatibility with legacy systems and simplifies software development. It may also be required for integration with existing software and infrastructure. Many existing programming systems support the installation of plaintext keys, which are saved securely on chips by using application code.
- Support for wrapped keys
  The use of wrapped keys for secure key storage is possible, but not required. The generation of wrapped keys is also supported.

Also note the following point.

- If a function for protection against simple or differential power analysis is required, consider using the RX RSIP protected mode driver.
- Using plaintext keys may create a risk of user keys being leaked.
  Using plaintext keys creates a risk of externally exposing the keys. Users should fully evaluate this risk and take appropriate measures.

- Restrictions on updating secure keys
  Updating secure keys is restricted due to the potential exposure of keys external to the RSIP.

1.3 Structure of Files in the Product

Table 1-2 below lists the files included in the product.

Table 1-2 Structure of Files in the Product

File or Directory (in Bold Type) Name	Description
r01an7445jj0100-rx-rsip-cm-security.pdf	RSIP CM driver application note (in Japanese)
r01an7445ej0100-rx-rsip-cm-security.pdf	RSIP CM driver application note (in English)
reference_documents	Contains documents on topics such as how to use the FIT module in various integrated development environments.
ja	Contains documents on topics such as how to use the FIT module in various integrated development environments. (In Japanese)
r01an1826jj0110-rx.pdf	Describes how to add firmware integration technology (FIT) modules to CS+ projects. (In Japanese)
r01an1723ju0121-rx.pdf	Describes how to add firmware integration technology (FIT) modules to e^2 studio projects. (In Japanese)
r20an0451js0140-e2studio-sc.pdf	Smart configurator user's guide (in Japanese)
en	Contains documents on topics such as how to use the FIT module in various integrated development environments. (In English)
r01an1826ej0110-rx.pdf	Describes how to add firmware integration technology (FIT) modules to CS+ projects. (In English)
r01an1723eu0121-rx.pdf	Describes how to add firmware integration technology (FIT) modules to e^2 studio projects. (In English)
r20an0451es0140-e2studio-sc.pdf	Smart configurator user's guide (in English)
FITModules	FIT module folder
r_rsip_cm_rx_v1.00.zip	RSIP CM driver FIT module
r_rsip_cm_rx_v1.00.xml	XML file for the RSIP CM driver FIT module as an e^2 studio FIT plug-in
r_rsip_cm_rx_v1.00_extend.mdf	Configuration settings file for the RSIP CM driver FIT module in use with the smart configurator
FITDemos	Demo project folder
rx261_ek_rsip_cm_sample	Demo project for key injection and cryptographic usage

Table 1-3 lists the files and folders contained in the folder produced by unzipping r_rsip_cm_rx_v.1.00.zip.

Table 1-3 File Structure

File or Directory (in Bold Type) Name	Description
r_config	RSIP CM driver config file folder
r_rsip_cm_rx_config.h	RSIP CM driver config file (with default settings)
r_rsip_cm_rx	
src	Source code folder
mbedtls	Open-source software (OSS) Mbed TLS folder
rm_psa_crypto	Hardware acceleration of PSA Crypto API implementation
r_rsip_cm_rx	RSIP CM driver FIT module folder
src	Source code folder
adaptors	PSA crypto adaptor folder
r_sce_adapt.c	PSA crypto adaptor source code
primitive	Source code folder for use in RSIP access
rx261	Folder containing the MCU type-dependent program code
DomainParams.c	DomainParam data information file
r_rsip_rx_finctionxxx.c	Source code for use in RSIP access “xxx” in the file name represents a numeric value.
r_rsip_rx_pxx.c	Source code for use in RSIP access “xx” in the file name represents a numeric value.
s_flash.c	Key information file
private/inc	Folder containing the program code for the internal functions of the RSIP CM driver
hw_sce_rx_private.h	Internal function header file
r_rsip_rx261_iodefine.h	Header file for use in RSIP access
public/inc	Folder containing the program for the API functions of the RSIP CM driver
hw_sce_rx_public.h	Header file for external functions
commom	Common source code folder
hw_sce_common.h	Common header file
hw_sce_private.h	Header file for the RSIP CM driver
hw_sce_aes_private.h	Header file for the AES API functions of the RSIP CM driver
hw_sce_ecc_private.h	Header file for the ECC API functions of the RSIP CM driver
hw_sce_hash_private.h	Header file for the hash API functions of the RSIP CM driver
hw_sce_trng_private.h	Header file for the random number generation API functions of the RSIP CM driver
hw_sce_rsa_private.h	Header file for the RSA API functions of the RSIP CM driver
r_rsip_cm_key_injection	Folder for key injection by the RSIP CM driver
r_rsip_cm_key_injection.h	Header file for the key injection API functions
r_rsip_cm_key_injection.c	Source code for the key injection API functions
doc	Folder containing the documentation
ja	Folder containing the documentation (in Japanese)
r01an7445jj0100-rx-rsip-security.pdf	RSIP CM driver application note (in Japanese)
en	Folder containing the documentation (in English)
r01an7445ej0100-rx-rsip-security.pdf	RSIP CM driver application note (in English)
r_rsip_cm_rx_if.h	Header file for the RSIP CM driver
readme.txt	A “readme” file

1.4 Documents for Reference

Table 1-4 lists documents related to this document.

Table 1-4 Documents for Reference

Document Name	Link to the Document
Mbed TLS 3.6.2	https://github.com/Mbed-TLS/mbedtls/releases/tag/mbedtls-3.6.2
Mbed TLS documentation hub	https://mbed-tls.readthedocs.io/en/latest/
PSA Crypto API 1.0	https://arm-software.github.io/psa-api/crypto/1.0/
Mbed TLS (Renesas)	https://github.com/renesas/mbedtls

Note: Mbed TLS 3.6.2 uses the specifications of PSA Crypto API 1.0.

The RX RSIP CM driver uses Mbed TLS (Renesas), which includes modifications to the official Mbed TLS source code.

1.5 Development Environment

The RSIP CM driver was developed by using the development environment described below. When developing your own applications, use the versions of software indicated below, or newer versions.

(1) Integrated development environment

Refer to the “Integrated development environments” item in section 7.1, Environments for Confirming Operation.

(2) C compiler

Refer to the “C compiler” item in section 7.1, Environments for Confirming Operation.

(3) Emulator/debugger

E2 Lite

(4) Evaluation board

Refer to the “Board used” item in section 7.1, Environments for Confirming Operation. Be sure to confirm the product part number at the time of purchase.

The e^2 studio and CC-RX were used in combination for evaluation and to create the demo project.

The project conversion function can be used to convert projects from the e^2 studio to CS+. If you encounter errors such as compiler errors, contact a Renesas sales office or representative.

1.6 Code Size

The table below lists the ROM and RAM sizes and the maximum stack usage for this module.

The configuration options listed in section 2.6, Configuration, during the build process determine the actual ROM (code and constants) and RAM (global data) sizes.

The values listed in the table below have been confirmed under the following conditions.

Module revision: r_rsip_cm_rx rev1.00

Compiler version: Renesas Electronics C/C++ Compiler Package for RX Family V3.07.00

(optimization level 2 with the “-lang = c99” option added to the default setting)

GCC for Renesas RX 8.3.0.202411

(Size-focused optimization with the “-std = gnu99” option added to the default setting)

IAR C/C++ Compiler for Renesas RX version 5.10.01

(optimization level high (or balanced))

Configuration options:

Renesas Electronics C/C++ Compiler Package for RX Family: -isa = rxv3, optimization level 2

GCC for Renesas RX: RXv3, optimization level -Os

IAR C/C++ Compiler for Renesas RX: --core rxv3 -Oh, optimization level high (or balanced)

Code Sizes of ROM, RAM, and Stack

Category	Memory Used		
	Renesas Compiler	GCC	IAR Compiler
ROM	**156,990 bytes**	**185,376 bytes**	**134,280 bytes**
RAM	**16,731 bytes**	**19,836 bytes**	**22,876 bytes**
Stack	**3,592 bytes**	**2,108 bytes**	**3,968 bytes**

1.7 Performance

Performance is measured in cycles of ICLK, the core clock. The RSIP CM operating clock (PCLKB) is set to a value that ensures the condition ICLK:PCLKB = 2:1. The driver is built by using CC-RX with optimization level 2. See section 7.1, Environments for Confirming Operation, for the version information. The configuration options are left at their default settings.

1.7.1 RX261

Table 1-5 Performance of Common API Functions

API Function	Performance (Unit: Cycles)
mbedtls_platform_setup	460,000
mbedtls_platform_teardown	600

Table 1-6 Performance of API Functions for Managing Key Data

API Function	Key Type	Performance (Unit: Cycles)
R_RSIP_CM_AES128_InitialKeyWrap	Encrypted	10,000
	Plaintext	7,000
R_RSIP_CM_AES256_InitialKeyWrap	Encrypted	10,000
	Plaintext	7,000
R_RSIP_CM_ECC_secp256r1_InitialPrivateKeyWrap	Encrypted	10,000
	Plaintext	7,000
R_RSIP_CM_ECC_secp256k1_InitialPrivateKeyWrap	Encrypted	10,000
	Plaintext	7,000
R_RSIP_CM_ECC_brainpoolP256r1_InitialPrivateKeyWrap	Encrypted	10,000
	Plaintext	7,000

Table 1-7 Performance of the API Function for Generating Random Numbers

API Function	Performance (Unit: Cycles)
psa_generate_random	17,000

Table 1-8 Performance of AES API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
ECB mode encryption	psa_cipher_encrypt_setup	2,600	2,600	2,600
	psa_cipher_update	22,000	29,000	36,000
	psa_cipher_finish	900	900	900
ECB mode decryption	psa_cipher_decrypt_setup	2,600	2,600	2,600
	psa_cipher_update	22,000	29,000	36,000
	psa_cipher_finish	1,300	1,300	1,300
CBC mode encryption	psa_cipher_encrypt_setup	2,800	2,800	2,800
	psa_cipher_update	8,000	8,200	8,300
	psa_cipher_finish	1,400	1,400	1,400
CBC mode decryption	psa_cipher_decrypt_setup	2,800	2,800	2,800
	psa_cipher_update	8,100	8,300	8,500
	psa_cipher_finish	1,900	1,900	1,900
CTR mode encryption	psa_cipher_encrypt_setup	2,900	2,900	2,900
	psa_cipher_update	8,100	8,200	8,400
	psa_cipher_finish	1,400	1,400	1,400
CTR mode decryption	psa_cipher_decrypt_setup	3,000	3,000	3,000
	psa_cipher_update	8,100	8,200	8,400
	psa_cipher_finish	1,900	1,900	1,900

Table 1-9 Performance of AES AEAD API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
GCM mode encryption	psa_aead_encrypt_setup	15,000	15,000	15,000
	psa_aead_update_ad	840	840	840
	psa_aead_update	75,000	95,000	100,000
	psa_aead_finish	16,000	16,000	16,000
GCM mode decryption	psa_aead_decrypt_setup	15,000	15,000	15,000
	psa_aead_update_ad	840	840	840
	psa_aead_update	75,000	95,000	100,000
	psa_aead_verify	16,000	16,000	16,000
CCM mode encryption	psa_aead_encrypt_setup	2,700	2,700	2,700
	psa_aead_update_ad	7,600	7,600	7,600
	psa_aead_update	44,000	59,000	73,000
	psa_aead_finish	9,400	9,400	9,400
	psa_aead_decrypt_setup	2,700	2,700	2,700
CCM mode decryption	psa_aead_update_ad	7,600	7,600	7,600
	psa_aead_update	44,000	59,000	73,000
	psa_aead_verify	9,300	9,300	9,400
	psa_aead_encrypt_setup	15,000	15,000	15,000
	psa_aead_update_ad	840	840	840

The GCM performance was measured with parameters fixed as follows: ivec = 128 bits, additional authentication data = 56 bits, and authentication tag = 128 bits.
The CCM performance was measured with parameters fixed as follows: nonce = 56 bits, additional authentication data = 56 bits, and authentication tag = 128 bits.

Table 1-10 Performance of AES MAC API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
CMAC generation	psa_mac_sign_setup	8,000	8,000	8,000
	psa_mac_update	1,700	1,800	1,900
	psa_mac_sign_finish	2,600	2,600	2,600
CMAC verification	psa_mac_verify_setup	8,000	8,000	8,000
	psa_mac_update	1,700	1,800	1,900
	psa_mac_verify_finish	2,800	2,800	2,800

Table 1-11 Performance of ECC API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
NIST secp256r1	psa_sign_hash	4,900,000	4,900,000	4,900,000
	psa_verify_hash	9,700,000	9,600,000	9,600,000
Koblitz secp256k1	psa_sign_hash	4,900,000	4,900,000	4,900,000
	psa_verify_hash	9,700,000	9,700,000	9,700,000
Brainpool p256r1	psa_sign_hash	4,900,000	4,900,000	4,900,000
	psa_verify_hash	9,600,000	9,600,000	9,600,000

Table 1-12 Performance of Hash API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
SHA224 generation	psa_hash_setup	390	390	390
	psa_hash_update	700	2,800	2,900
	psa_hash_finish	3,400	3,400	3,400
SHA224 verification	psa_hash_setup	390	390	390
	psa_hash_update	700	2,800	2,900
	psa_hash_verify	4,300	4,300	4,300
SHA256 generation	psa_hash_setup	380	380	380
	psa_hash_update	700	2,800	2,900
	psa_hash_finish	3,400	3,400	3,400
SHA256 verification	psa_hash_setup	380	380	380
	psa_hash_update	700	2,800	2,900
	psa_hash_verify	4,400	4,400	4,400

Table 1-13 Performance of HMAC API Functions

Algorithm	API Function	Performance (Unit: Cycles)		
		48-Byte Processing	64-Byte Processing	80-Byte Processing
HMAC-SHA224 generation	psa_mac_sign_setup	390	390	390
	psa_mac_update	700	2,800	2,900
	psa_mac_sign_finish	3,400	3,400	3,400
HMAC-SHA224 verification	psa_mac_verify_setup	390	390	390
	psa_mac_update	700	2,800	2,900
	psa_mac_verify_finish	4,300	4,300	4,300
HMAC-SHA256 generation	psa_mac_sign_setup	380	380	380
	psa_mac_update	700	2,800	2,900
	psa_mac_sign_finish	3,400	3,400	3,400
HMAC-SHA256 verification	psa_mac_verify_setup	380	380	380
	psa_mac_update	700	2,800	2,900
	psa_mac_verify_finish	4,400	4,400	4,400

2. API Information

2.1 Hardware Requirements

The RX RSIP CM driver can only be used with devices that incorporate an RSIP module. Check the part number of the device to ensure that the one you intend to use is suitable.

2.2 Software Requirements

The RSIP CM driver is dependent on the following module.

- r_bsp Use v7.52 or a later version. “BSP” stands for “board support package”.

Change the value of the following macro in r_bsp_config.h in the r_config folder to 0xB.

```
/* Chip version.
   Character(s) = Value for macro =
   A          = 0xA          = Chip version A
                            = Encryption module not included, USB included,
CAN FD included (only CAN 2.0 protocol supported)
   B          = 0xB          = Chip version B
                            = Encryption module and USB included, CAN FD
included
*/
#define BSP_CFG_MCU_PART_VERSION       (0xB)
```

2.3 Supported Toolchains

The operation of the RSIP CM driver has been confirmed with the toolchains indicated in section 7.1, Environments for Confirming Operation.

2.4 Header File

All API calls and their supported interface definitions are contained in r_rsip_cm_rx_if.h.

2.5 Integer Types

The RCIP CM driver uses ANSI C99 integer types as defined in stdint.h.

2.6 Configuration

The Mbed TLS configuration options are set in mbedtls_config.h. The option names and settings are listed in Table 2-1, Definitions in mbedtls_config.h, below.

Table 2-1 Definitions in mbedtls_config.h

Option Name	Default Value
MBEDTLS_PLATFORM_SETUP_TEARDOWN_ALT	Defined
MBEDTLS_AES_ALT	Defined
MBEDTLS_CCM_ALT	Defined
MBEDTLS_GCM_ALT	Defined
MBEDTLS_SHA256_ALT	Defined
MBEDTLS_SHA256_PROCESS_ALT	Defined
MBEDTLS_AES_SETKEY_ENC_ALT	Defined
MBEDTLS_AES_SETKEY_DEC_ALT	Defined
MBEDTLS_AES_ENCRYPT_ALT	Defined
MBEDTLS_AES_DECRYPT_ALT	Defined
MBEDTLS_ECDSA_VERIFY_ALT	Defined
MBEDTLS_ECDSA_SIGN_ALT	Defined
MBEDTLS_ENTROPY_HARDWARE_ALT	Defined
MBEDTLS_CIPHER_MODE_CBC	Defined
MBEDTLS_CIPHER_MODE_CTR	Defined
MBEDTLS_CIPHER_PADDING_PKCS7	Defined
MBEDTLS_CIPHER_PADDING_ONE_AND_ZEROS	Defined
MBEDTLS_CIPHER_PADDING_ZEROS_AND_LEN	Defined
MBEDTLS_CIPHER_PADDING_ZEROS	Defined
MBEDTLS_ECP_DP_SECP256R1_ENABLED	Defined
MBEDTLS_ECP_DP_SECP256K1_ENABLED	Defined
MBEDTLS_ECP_DP_BP256R1_ENABLED	Defined
MBEDTLS_ERROR_STRERROR_DUMMY	Defined
MBEDTLS_GENPRIME	Defined
MBEDTLS_FS_IO	Defined
MBEDTLS_NO_PLATFORM_ENTROPY	Defined
MBEDTLS_PKCS1_V15	Defined
MBEDTLS_PKCS1_V21	Defined
MBEDTLS_VERSION_FEATURES	Defined
MBEDTLS_AES_C	Defined
MBEDTLS_ASN1_PARSE_C	Defined
MBEDTLS_ASN1_WRITE_C	Defined
MBEDTLS_BASE64_C	Defined
MBEDTLS_BIGNUM_C	Defined
MBEDTLS_CCM_C	Defined
MBEDTLS_CIPHER_C	Defined
MBEDTLS_CMAC_C	Defined
MBEDTLS_CTR_DRBG_C	Defined
MBEDTLS_CTR_DRBG_C_ALT	Defined
MBEDTLS_ECDSA_C	Defined
MBEDTLS_ECP_C	Defined
MBEDTLS_ENTROPY_C	Defined
MBEDTLS_ERROR_C	Defined
MBEDTLS_GCM_C	Defined
MBEDTLS_HKDF_C	Defined

Option Name	Default Value
MBEDTLS_LMS_C	Defined
MBEDTLS_MD_C	Defined
MBEDTLS_MD5_C	Defined
MBEDTLS_OID_C	Defined
MBEDTLS_PEM_PARSE_C	Defined
MBEDTLS_PEM_WRITE_C	Defined
MBEDTLS_PKCS5_C	Defined
MBEDTLS_PKCS12_C	Defined
MBEDTLS_PLATFORM_C	Defined
MBEDTLS_PSA_CRYPTO_C	Defined
MBEDTLS_PSA_CRYPTO_ACCEL_DRV_C	Defined
MBEDTLS_PSA_CRYPTO_STORAGE_C	Defined
MBEDTLS_PSA_ITS_FILE_C	Defined
MBEDTLS_RSA_C	Defined
MBEDTLS_SHA224_C	Defined
MBEDTLS_SHA256_C	Defined
MBEDTLS_VERSION_C	Defined
MBEDTLS_MPI_WINDOW_SIZE	6
MBEDTLS_MPI_MAX_SIZE	1024
MBEDTLS_ECP_WINDOW_SIZE	6
MBEDTLS_ECP_FIXED_POINT_OPTIM	1
MBEDTLS_CHECK_RETURN	Enabled

2.6.1 Setting of the Platform

To realize hardware acceleration with the RSIP CM driver, define the MBEDTLS_PLATFORM_SETUP_TEARDOWN_ALT macro in the mbedtls_config.h config file. This enables the code for initializing the RSIP.

2.6.2 Setting for Generating Random Numbers

The MBEDTLS_ENTROPY_HARDWARE_ALT macro must be defined in the mbedtls_config.h config file. This allows use of the TRNG as an entropy source for the RSIP. Without this functionality, other encryption operations will not work, even in software-only mode.

2.6.3 Setting for Generating a Wrapped Key

To use the RSIP to generate a wrapped key, specify PSA_KEY_TYPE_AES_WRAPPED or PSA_KEY_TYPE_ECC_KEY_PAIR_WRAPPED(curve) as the key type attribute.

2.6.4 Setting of AES

To enable hardware acceleration of AES-128 or AES-256 operations, the MBEDTLS_AES_SETKEY_ENC_ALT, MBEDTLS_AES_SETKEY_DEC_ALT, MBEDTLS_AES_ENCRYPT_ALT, and MBEDTLS_AES_DECRYPT_ALT macros must be defined in the mbedtls_config.h config file. AES is enabled with the default setting.

2.6.5 Setting of ECC

To enable hardware acceleration of ECC-based key generation, the MBEDTLS_ECP_ALT macro must be defined in the configuration file. For the ECDSA, the MBEDTLS_ECDSA_SIGN_ALT and MBEDTLS_ECDSA_VERIFY_ALT macros must be defined. ECC and the ECDSA are enabled with the default settings. To disable ECC, delete the definitions of the MBEDTLS_ECP_C, MBEDTLS_ECDSA_C, and MBEDTLS_ECDH_C macros in the mbedtls_config.h config file.

2.6.6 Setting of SHA224 and SHA256

To enable hardware acceleration of SHA224 and SHA256 calculations, the MBEDTLS_SHA256_ALT and MBEDTLS_SHA256_PROCESS_ALT macros must be defined in the configuration file. SHA256 is enabled with the default setting.

2.7 Structures

The table below describes the definitions of the structures used in key injection by the RSIP CM driver.

Table 2-2 Definitions of the Structures Used in Key Injection by the RSIP CM Driver

Definition	Description
rsip_key_injection_type_t	Key injection type structure
rsip_aes_wrapped_key_t	AES wrapped key structure
rsip_ecc_private_wrapped_key_t	ECC private wrapped key structure

2.8 Return Values

The table below describes the return values used by the key injection API functions for use with the RSIP CM driver. The enumerated type of the return values is defined as fsp_err_t in /r_bsp/mcu/all/fsp_common_api.h.

Table 2-3 enum fsp_err_t Return Values

Enumerator	Value	Description
FSP_SUCCESS	0x00000	Successful completion
FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT	0x10001	The hardware resource required for this process is in use by another process so resource contention between the processes prevented the key injection processing.
FSP_ERR_CRYPTO_SCE_FAIL	0x10002	An input parameter was incorrect.

2.9 Including the FIT Module in Your Project

This module must be included in each project where it is to be used. Renesas recommends using the Smart Configurator in the way described in either (1) or (3) below. However, the Smart Configurator only supports this for some RX devices. If the RX device in use is not supported, use the method described in either (2) or (4).

(1) Including the FIT module in your project by using the Smart Configurator in the e^2 studio
Using the Smart Configurator in the e^2 studio allows the automatic inclusion of the FIT module in your project. Refer to the application note *Renesas e^2 studio Smart Configurator User's Guide* (R20AN0451) for details.

(2) Including the FIT module in your project by using the FIT Configurator in the e^2 studio
Using the FIT Configurator in the e^2 studio allows the automatic inclusion of the FIT module in your project. Refer to the application note *RX Family: Adding Firmware Integration Technology Modules to Projects* (R01AN1723) for details.

(3) Including the FIT module in your project by using the Smart Configurator in CS+
Using the standalone version of the Smart Configurator in CS+ allows the automatic inclusion of the FIT module in your project. Refer to the application note *Renesas e^2 studio Smart Configurator User's Guide* (R20AN0451) for details.

(4) Including the FIT module in your project in CS+
Manually include the FIT module in your project in CS+. Refer to the application note *RX Family: Adding Firmware Integration Technology Modules to CS+ Projects* (R01AN1826) for details.

3. Using the RSIP CM Driver

The RSIP CM driver for the RX family provides the following functions. Detailed descriptions of the API functions are given in section 4.

- Random number generation
- Secure key management
- Unauthorized access monitoring
- Acceleration of cryptographic operations

The RSIP CM driver implements secure key management by using a hardware unique key (HUK) to wrap keys. This realizes key confidentiality and the detection of tampering external to the RSIP. See section 3.5.1 for the flow of key injection. Detailed descriptions of how to inject keys are given in section 5.

Unauthorized access monitoring by the RSIP covers all cryptographic processing performed by the driver and is always enabled during cryptographic operations. If tampering with cryptographic operations is detected while the driver is in use, the driver stops operating.

The RSIP CM driver uses the open-source PSA Crypto API to proceed with cryptographic operations. The rm_psa_crypto program and RSIP CM, the primary modules of the RX RSIP CM driver, work with the PSA Crypto API to handle cipher-related operations. The RSIP key injection module of the RX RSIP CM driver supports key injection. The configuration of the RSIP CM driver is shown in Figure 3-1, Schematic Diagram of the RSIP CM Driver.

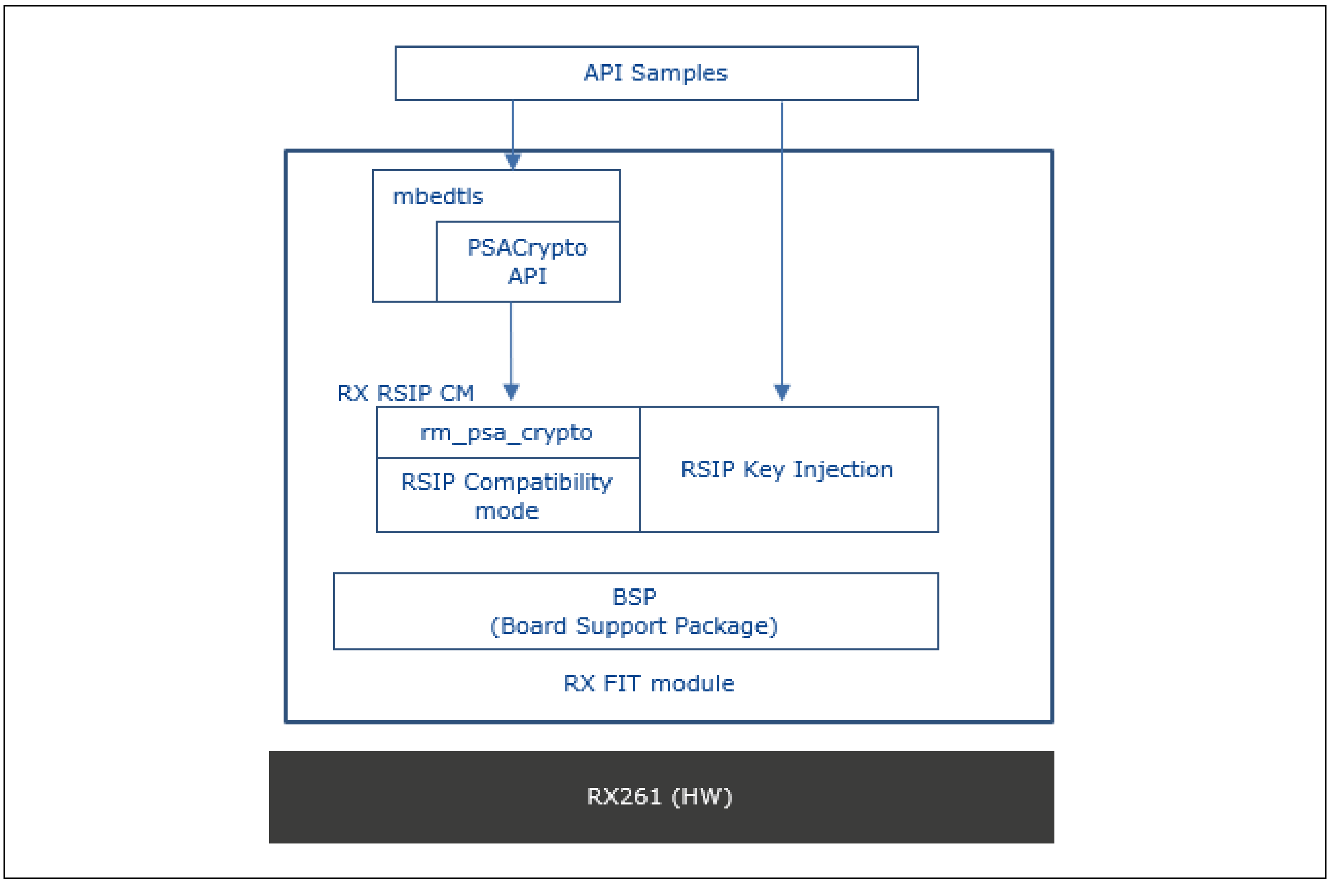


Figure 3-1 Schematic Diagram of the RSIP CM Driver

3.1 Initializing the RSIP

In initializing the RSIP, mbedtls_platform_setup() must be called before use of the PSA Crypto API.

No.	PSA Crypto API	Description
1	mbedtls_platform_setup()	Processing for initialization

3.2 Memory Usage

Depending on the PSA Crypto features used, the heap size settings are required. The total amount of heap memory to be allocated is the sum of heap areas required for the individual algorithms that will be run.

Algorithm	Heap Size in Bytes
SHA224 and SHA256	0x500
AES	0x2000
ECC	0x2000

Using the module also requires a stack area of at least 0x2000 bytes.

3.3 Restrictions

3.3.1 Endian for Operation

Only little endian is supported.

3.3.2 Definition of MBEDTLS_PLATFORM_SETBUF_MACRO

In the RX RSIP CM driver, the dummy_setbuf() dummy function is defined in MBEDTLS_PLATFORM_SETBUF_MACRO to prevent errors during building. Defining a user-defined function in MBEDTLS_PLATFORM_SETBUF_MACRO, which is defined in the mbedtls_config.h file, allows replacing the dummy function with the user-defined function.

3.4 Single-Part and Multi-Part Operations

The PSA Crypto API has two types of API functions: those that provide cryptographic operations with a single API function and those that provide the operations through multiple API functions. In this document, the former are referred to as single-part operations and the latter as multi-part operations.
APIs for single-part and multi-part operations are provided for symmetric key cryptography and hashes (message digest generation and HMAC functions), and APIs for single-part operations are provided for other cryptographic operations.
Multi-part operations are parts of the API in which a single cryptographic operation is split into a sequence of separate steps (Allocate-Initialize-Setup-Update-Finish). This enables fine control over the configuration of the cryptographic operation and allows message data to be processed in fragments instead of all at once.
For details on the single-part operations, refer to section 3.3.1, Single-part Functions, in the PSA Crypto API 1.0 documentation. For details on the multi-part operations, refer to section 3.3.2, Multi-part operations, in the PSA Crypto API 1.0 documentation.

3.5 Key Management

No.	PSA Crypto API	RSIP CM API Functions	Description

1	-	R_RSIP_CM_AES128_InitialKeyWrap()
		R_RSIP_CM_AES256_InitialKeyWrap()
		R_RSIP_CM_ECC_secp256r1_InitialPrivateKeyWrap()
		R_RSIP_CM_ECC_secp256k1_InitialPrivateKeyWrap()
		R_RSIP_CM_ECC_brainpoolP256r1_InitialPrivateKeyWrap()
			Key injection

2	psa_generate_key()	-	Key generation

Figure 3-2 shows how keys are handled during cryptographic operations by the RSIP CM driver.

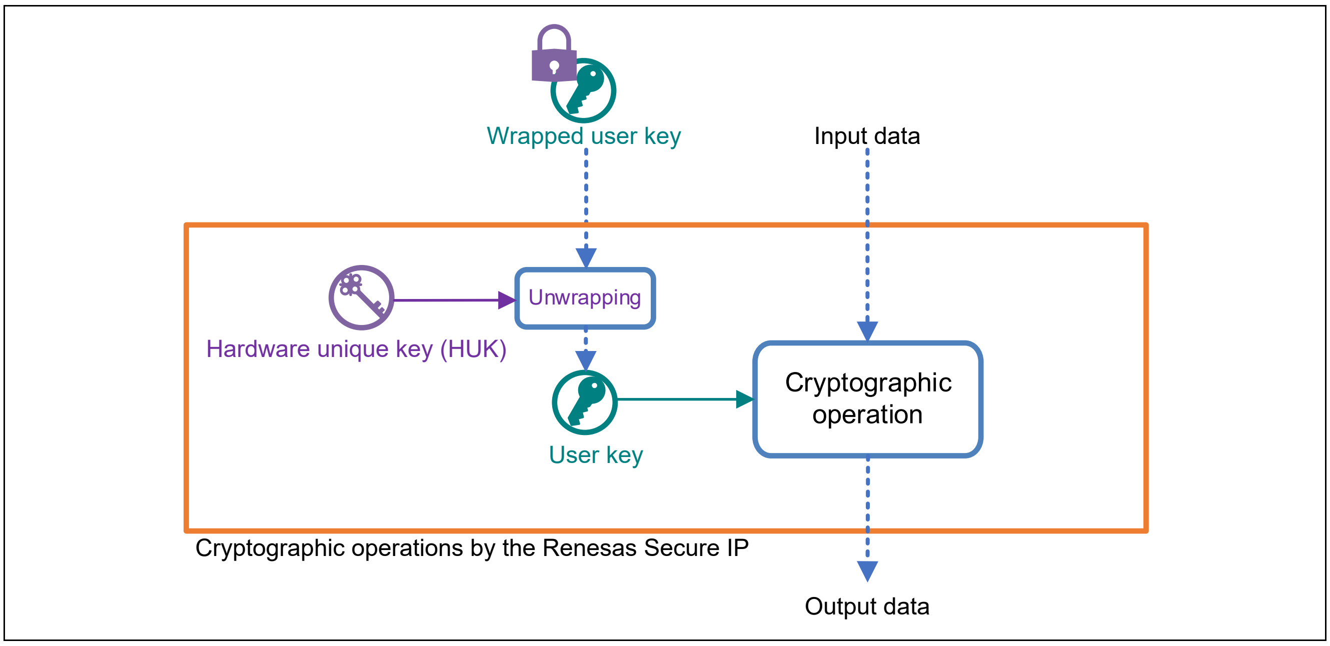


Figure 3-2 How Keys are Handled during Cryptographic Operations by the RSIP CM Driver

The keys handled in cryptographic operations by the RSIP CM driver (input and output keys) are opaque keys wrapped by using a device-specific key called an HUK, which is only accessible by the RSIP. In the case of the RSIP CM driver, this type of opaque key is called a wrapped key. Note that a public key for use in asymmetric key cryptography is used without change from the plaintext key data.

The RSIP CM driver implements secure key management by using a device-specific key to wrap user keys. This realizes key confidentiality and the detection of tampering external to the RSIP. A wrapped key can only be unwrapped by the RSIP, and the unwrapped key only exists during the cryptographic processing within the RSIP. Since the wrapped key has been wrapped by using a device-specific key, it cannot be unwrapped by using a different device-specific key, even if the wrapped key is copied from the nonvolatile memory of one device to another.

3.5.1 Key Injection

The RX RSIP CM driver only handles wrapped keys in key injection. Direct input of a plaintext key to the API functions for cryptographic operations is not possible. When a plaintext key is to be used, it requires conversion to a wrapped key through the key injection API function before use. Note that a public key for use in asymmetric key cryptography can be used directly by the API functions for cryptographic operations without converting the plaintext key to a wrapped key.

3.5.1.1 Key Injection by Using Encrypted Keys

Key injection provides a mechanism enabling the secure delivery of user keys by converting them into wrapped keys, which have been wrapped by using the HUK. Figure 3-3 shows the flow of injecting encrypted keys, including use of the Renesas Key Wrap Service.

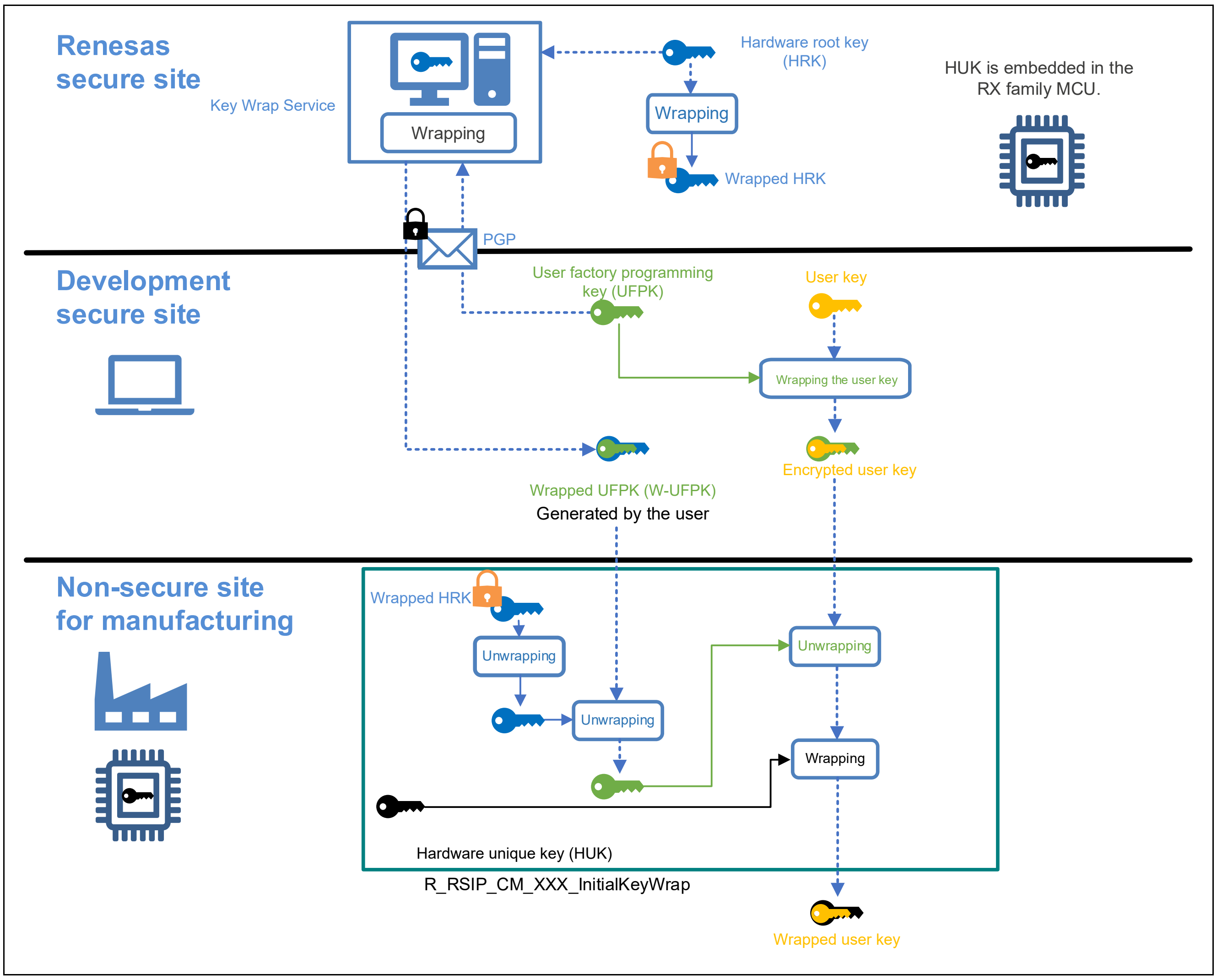


Figure 3-3 Key Injection by Using Encrypted Keys

Generate a user key and UFPK at your secure site (“Development secure site” in Figure 3-3). Next, generate an encrypted user key by using the UFPK to wrap the user key, which is to be used for your application. Also use the Renesas Key Wrap Service to generate a W-UFPK from the UFPK that was used in wrapping. For key injection, input the encrypted user key to the key injection API function (R_RSIP_CM_XXX_InitialKeyWrap) to generate a wrapped user key.

Figure 3-4 shows the method of generating an encrypted user key by using a UFPK to wrap a user key.

Encrypt a user key in AES-128 CBC mode with the first 128 bits of a UFPK for use in wrapping as the key, then use AES-128 CBC-MAC to calculate the MAC of the user key with the trailing 128 bits of the UFPK for use in wrapping as the key. Concatenate the MAC of the user key to the user key and encrypt the result of concatenation to generate an encrypted user key.

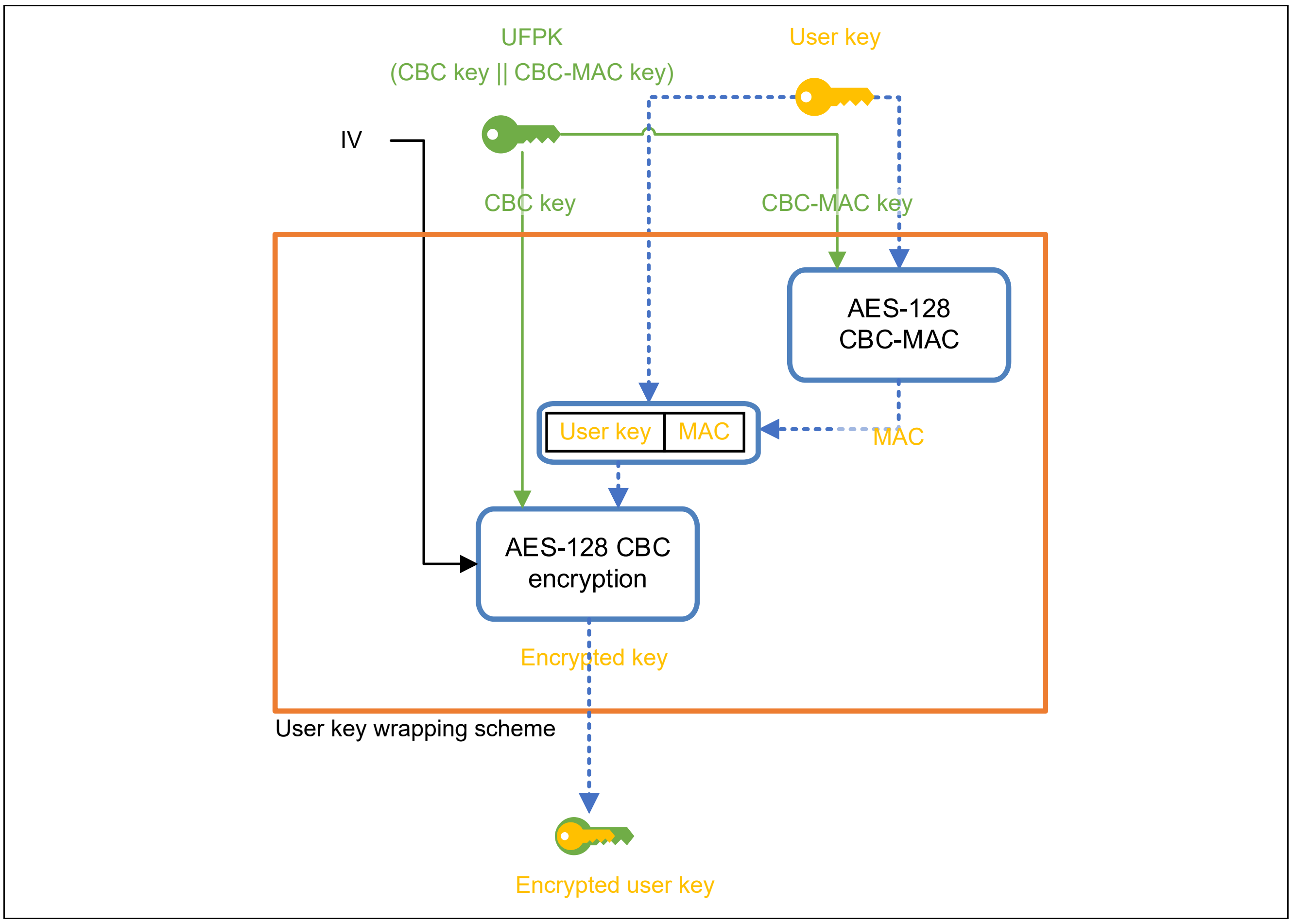


Figure 3-4 User Key Wrapping Scheme during Key Injection

The specific listing for generating an encrypted user key is given below.

```
--------------------------------------------------------------------------------
uint32_t user_key[len];
uint32_t MAC[4] = 0;
uint32_t iv[4] = IV;
for (i = 0; i < len; i += 4)
{
        MAC   = AES_128_ENCRYPT(CBCMACkey[0: 3], xor_16byte(user_key[i: i+3], MAC[0: 3]));
        encrypted_key[i: i+3] = AES_128_ENCRYPT(CBCkey[0: 3], xor_16byte(user_key[i: i+3], iv[0: 3]));
        iv[0: 3] = encrypted_key [i: i+3];
}
encrypted_key[i: i+3] = AES_128_ENCRYPT(CBCkey[0: 3], xor_16byte(MAC[0: 3], iv[0: 3]));
--------------------------------------------------------------------------------
```

The functions used in the listing above handle the following processing.

- AES_128_ENCRYPT(Key, Data): Encryption of data in AES-128 ECB mode with the use of an encryption key
- xor_16byte(data1, data2): Taking the XOR of 16-byte values data1 and data2

The elements of arrays CBCkey[], CBCMACkey[], MAC[], iv[], user_key[], and encrypted_key[] each have a capacity of 4 bytes.
For details on the data formats for user keys and encrypted user keys, see section 7.3, User Key Formats.

3.5.1.2 Key Injection with the Use of a Plaintext Key

To inject a plaintext key, the output of a key injection API function is used within a PSA Crypto key setting API function (psa_xxx_setup). The key injection API function uses an HUK to convert a user key to a wrapped key. Figure 3-5 shows the flow of injecting a plaintext key.

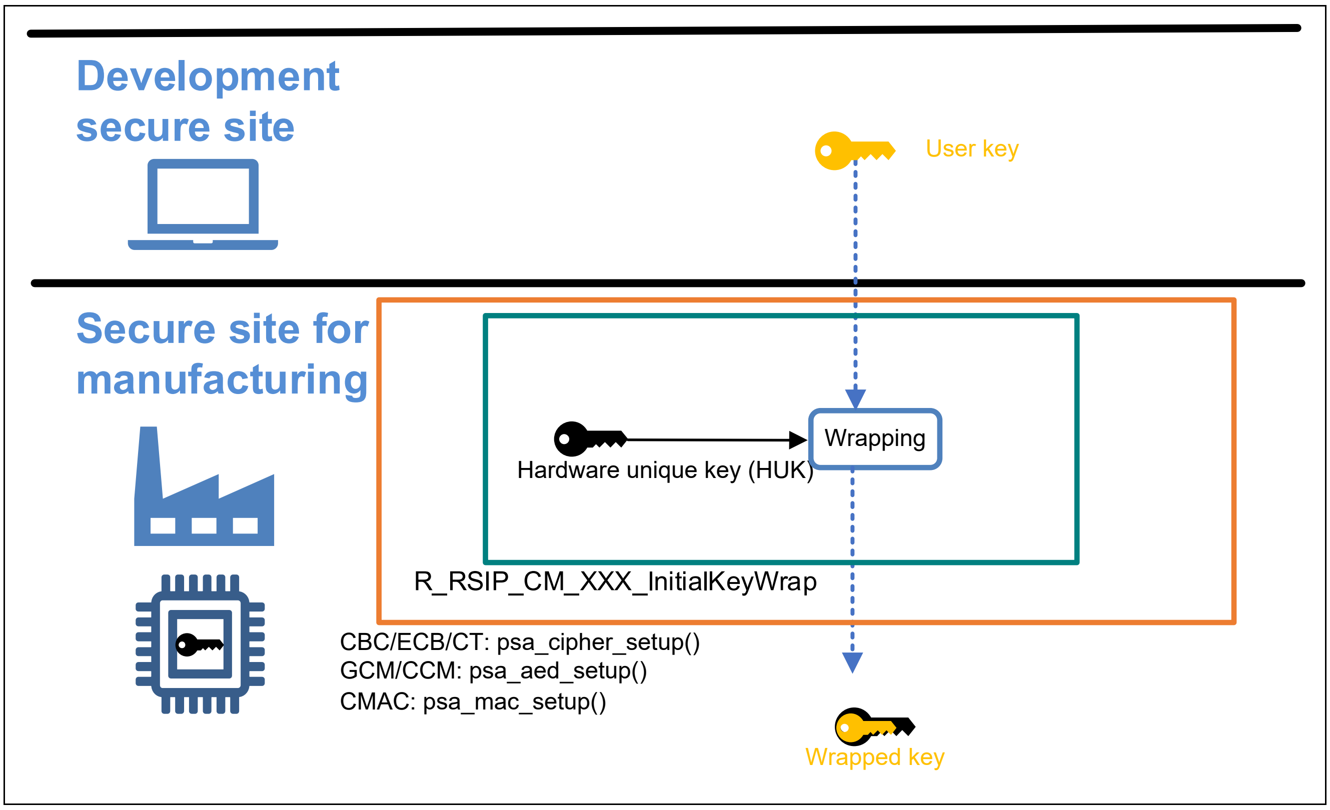

Development secure site
User key
Secure site for manufacturing
Wrapping
Hardware unique key (HUK)
R_RSIP_CM_XXX_InitialKeyWrap
CBC/ECB/CT: psa_cipher_setup()
GCM/CCM: psa_aed_setup()
CMAC: psa_mac_setup()
Wrapped key

Figure 3-5 Key Injection with the Use of a Plaintext Key

3.5.2 Key Generation

In key generation, the key generation functionality of the PSA Crypto API is used to generate random number keys. When a wrapped key is specified as the key type, the random number generation functionality generates a new plaintext key and outputs it in the wrapped key format. Figure 3-6 shows the flow of generating a wrapped key. When a plaintext key is specified as the key type, the random number generation functionality generates a new plaintext key and outputs it without change. Figure 3-7 shows the flow of generating a plaintext key. Generating an ECC plaintext key is not supported.
In generating keys, persistent keys with a lifetime specified as a key attribute are not supported. Only volatile keys are specifiable.

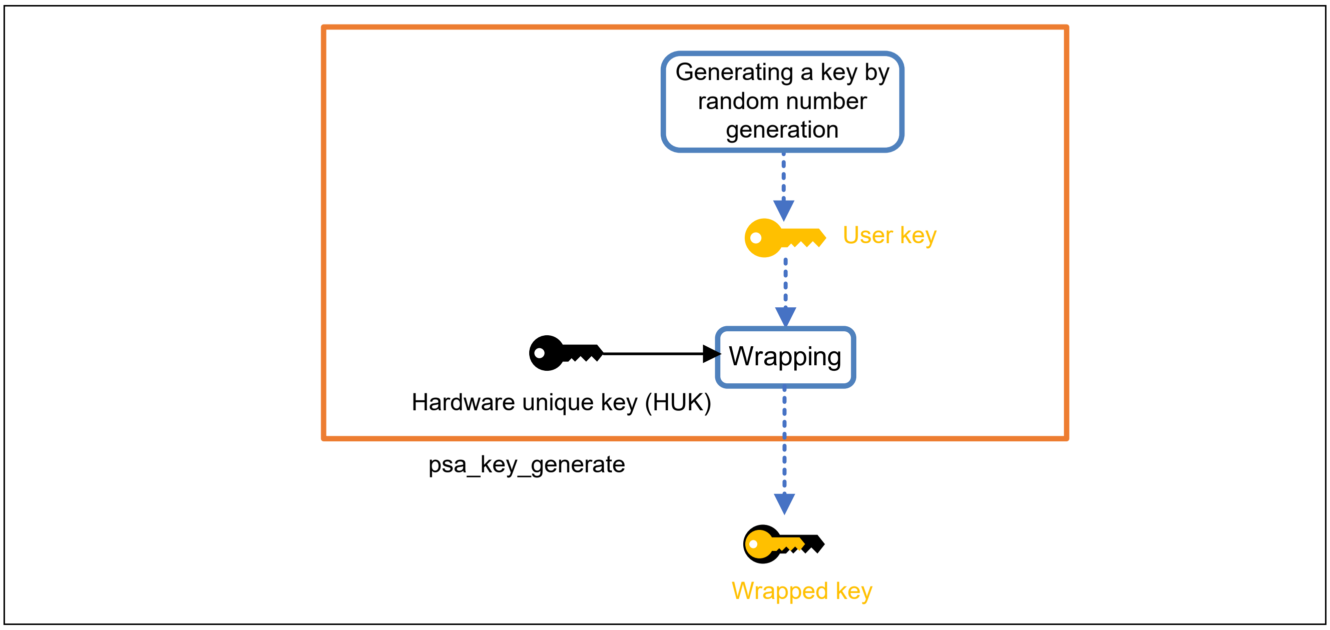

Generating a key by random number generation
User key
Hardware unique key (HUK)
Wrapping
psa_key_generate
Wrapped key

Figure 3-6 Flow of Generating a Wrapped Key

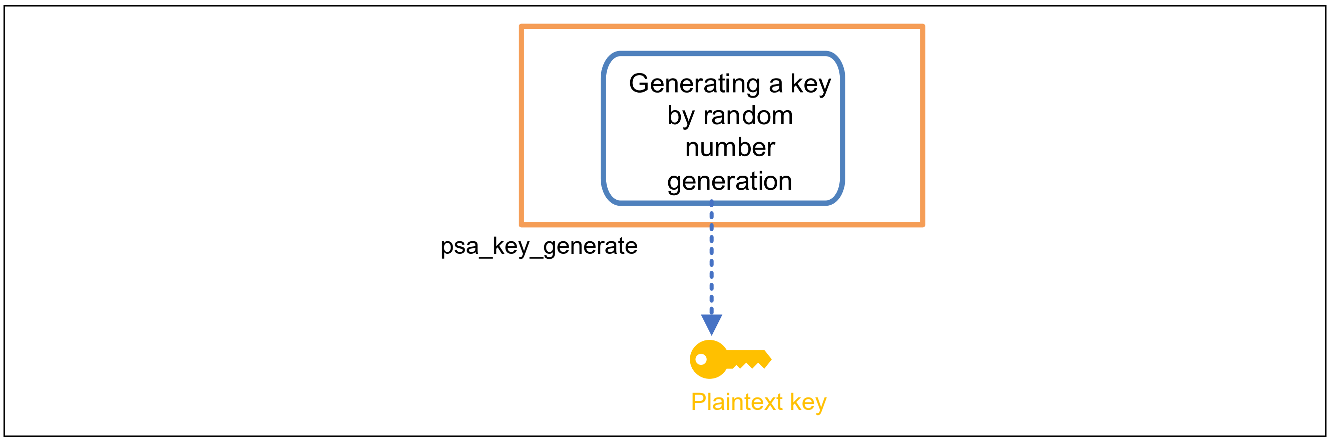


Figure 3-7 Key Injection with the Use of a Plaintext Key

3.6 Random Number Generation

No.	PSA Crypto API	Description
1	psa_generate_random	Generates random numbers.

3.7 Symmetric Key Cryptography

No.	PSA Crypto API	Description
1	**AES-ECB/CBC/CTR** psa_cipher_encrypt psa_cipher_decrypt psa_cipher_encrypt_setup psa_cipher_decrypt_setup psa_cipher_update psa_cipher_finish	AES-ECB/CBC/CTR encryption and decryption
2	**AES-CCM/GCM** psa_aead_encrypt psa_aead_decrypt psa_aead_encrypt_setup psa_aead_decrypt_setup psa_aead_set_nonce psa_aead_update_ad psa_aead_update psa_aead_finish psa_aead_verify	AES-CCM/GCM encryption and decryption
3	**AES-CMAC** psa_mac_compute psa_mac_verify psa_mac_sign_setup psa_mac_verify_setup psa_mac_update psa_mac_sign_finish psa_mac_verify_finish	AES-CMAC signature and verification

3.8 Asymmetric Key Cryptography

No.	PSA Crypto API	Description
1	psa_sign_message psa_verify_message psa_sign_hash psa_verify_hash mbedtls_ecp_mul	ECC signature, verification, and scalar multiplication

3.9 Hash Functions

No.	PSA Crypto API	Description
1	psa_hash_compute psa_hash_setup psa_hash_update psa_hash_finish psa_hash_verify	Hash calculation and verification

4. API Functions

4.1 List and Details of API Functions

The tables below and on the following pages list the API functions for use with the RX RSIP CM driver.

The initialization API function uses the platform setting function of the Mbed TLS library to open the RX RSIP CM driver. For details on the API function, refer to “Mbed TLS documentation hub” at the URL stated in section 1.4, Documents for Reference.

The key management, random number generation, AES, ECC, and hash API functions support the PSA Crypto API of the Mbed TLS library. For details on the API functions, refer to “PSA Crypto API 1.0” at the URL stated in section 1.4, Documents for Reference.

The API functions that are provided for version information and key injection are original to the RX RSIP CM driver.

Table 4-1 Initialization API Function

API Function	Description	Details
mbedtls_platform_setup	Releases the RSIP module from the module stop state and opens the RSIP CM driver.	Refer to the following section at the “Mbed TLS documentation hub”. Platform setup

Table 4-2 Key Management API Functions

API Function	Description	Details
psa_key_attributes_init	Gets the initial values of the key attribute objects.	Refer to the following section at the “PSA Crypto API 1.0”. 9. Key management reference
psa_get_key_attributes	Gets the key attributes.	
psa_reset_key_attributes	Initializes the states of the key attribute objects.	
psa_set_key_type	Sets the key type.	
psa_get_key_type	Gets the key type.	
psa_get_key_bits	Gets the key length in bits.	
psa_set_key_bits	Sets the key length in bits.	
psa_set_key_lifetime	Sets the lifetime of a persistent key.	
psa_get_key_lifetime	Gets the key lifetime.	
psa_set_key_id	Sets the key ID.	
psa_get_key_id	Gets the key ID.	
psa_set_key_algorithm	Sets the algorithm for key encryption.	
psa_get_key_algorithm	Gets the algorithm for key encryption.	
psa_set_key_usage_flag	Sets the intended use of a key.	
psa_get_key_usage_flag	Gets the intended use of a key.	
psa_import_key	Imports a key.	
psa_generate_key	Generates a key or key pair.	
psa_copy_key	Copies a key.	
psa_destoroy_key	Destroys a key.	
psa_purge_key	Purges unnecessary copies of a key.	
psa_export_key	Exports a key.	
psa_export_public_key	Exports a public key or the public key of a key pair.	

Table 4-3 Random Number Generation API Function

API Function	Description	Details
psa_generate_random	Generates random numbers.	Refer to the following section at the "PSA Crypto API 1.0". 10.10.1. Random number generation

Table 4-4 AES-ECB/CBC/CTR API Functions

API Function	Description	Details
psa_cipher_encrypt	Executes an AES encryption operation.	Refer to the following section at the "PSA Crypto API 1.0". 10.4. Unauthenticated ciphers
psa_cipher_decrypt	Executes an AES decryption operation.	
psa_cipher_operation_init	Prepares for execution of an AES encryption operation.	
psa_cipher_encrypt_setup	Sets a key for use in an AES encryption operation.	
psa_cipher_decrypt_setup	Sets a key for use in an AES decryption operation.	
psa_cipher_set_iv	Sets an initialization vector (IV) for use in an AES encryption operation.	
psa_cipher_generate_iv	Generates an initialization vector (IV) for use in an AES encryption operation.	
psa_cipher_update	Executes an AES encryption operation.	
psa_cipher_finish	Finishes an AES encryption operation.	

Table 4-5 AES-CCM/GCM API Functions

API Function	Description	Details
psa_aead_encrypt	Executes an AEAD encryption operation.	Refer to the following section at the "PSA Crypto API 1.0". 10.5. Authenticated encryption with associated data (AEAD)
psa_aead_decrypt	Executes an AEAD decryption operation.	
psa_aead_operation_init	Returns the initial value of an AEAD operation object.	
psa_aead_encrypt_setup	Sets a key for use in an AEAD encryption operation.	
psa_aead_decrypt_setup	Sets a key for use in an AEAD decryption operation.	
psa_aead_set_lengths	Sets the size of additional authenticated data for an AEAD operation.	
psa_aead_generate_nonce	Generates a nonce for use with an AEAD operation.	
psa_aead_set_nonce	Sets a nonce for use with an AEAD operation.	
psa_aead_update_ad	Passes additional authenticated data for use with an AEAD operation.	
psa_aead_update	Executes an AEAD operation.	
psa_aead_finish	Finishes an AEAD encryption operation.	
psa_aead_verify	Handles termination and verification of an AEAD decryption operation.	
psa_aead_abort	Aborts an AEAD operation.	

Table 4-6 AES-CMAC/HMAC-SHA API Functions

API Function	Description	Details
psa_mac_compute	Calculates the MAC of a message.	Refer to the following section at the “PSA Crypto API 1.0”. 10.3. Message authentication codes (MAC)
psa_mac_verify	Calculates the MAC of a message and compares the result with a reference value.	
psa_mac_operation_init	Returns the initial value of a MAC calculation object.	
psa_mac_sign_setup	Sets a key for use in a MAC signature operation.	
psa_mac_verify_setup	Sets a key for use in a MAC verification operation.	
psa_mac_update	Executes a MAC calculation.	
psa_mac_sign_finish	Finishes a MAC signature operation.	
psa_mac_verify_finish	Finishes a MAC verification operation.	
psa_mac_abort	Aborts a MAC calculation.	

Table 4-7 ECC API Functions

API Function	Description	Details
psa_sign_message	Signs a message with a private key.	Refer to the following section at the “PSA Crypto API 1.0”. 10.7. Asymmetric signature
psa_verify_message	Verifies a message with a public key.	
psa_sign_hash	Signs a hash with a private key.	
psa_verify_hash	Verifies a hash with a private key.	
mbedtls_ecp_mul	Executes a scalar multiplication.	Refer to the following section at the “Mbed TLS documentation hub”. Scalar multiplication API function

Table 4-8 SHA-224/256 API Functions

API Function	Description	Details
psa_hash_compute	Calculates the hash of a message.	Refer to the following section at the “PSA Crypto API 1.0”. 10.2. Message digests (Hashes)
psa_hash_compare	Calculates the hash of a message and compares the result with a reference value.	
psa_hash_operation_init	Returns the initial value of a hash calculation object.	
psa_hash_setup	Makes the initial setting for a hash calculation object.	
psa_hash_update	Executes a hash calculation.	
psa_hash_finish	Finishes a hash calculation.	
psa_hash_verify	Handles processing to finish hash verification.	
psa_hash_abort	Aborts a hash calculation.	
psa_hash_clone	Clones a hash calculation.	

Table 4-9 Version Information API Function

API Function	Description
R_RSIP_CM_GetVersion	Gets the version information of the RX RSIP CM driver.

Table 4-10 Key Injection API Functions

API Function	Description
R_RSIP_CM_AES128_InitialKeyWrap	Generates a 128-bit AES wrapped key.
R_RSIP_CM_AES256_InitialKeyWrap	Generates a 256-bit AES wrapped key.
R_RSIP_CM_ECC_secp256r1_InitialPrivateKeyWrap	Generates a wrapped key from an ECC-secp256r1 private key.
R_RSIP_CM_ECC_secp256k1_InitialPrivateKeyWrap	Generates a wrapped key from an ECC-secp256k1 private key.
R_RSIP_CM_ECC_brainpoolP256r1_InitialPrivateKeyWrap	Generates a wrapped key from an ECC-brainpoolP256r1 private key.

4.1.1 Version Information

4.1.1.1 R_RSIP_CM_GetVersion

Format

(1) uint32_t R_RSIP_CM_GetVersion (void)

Parameters

None

Return Values

Higher-order 2 bytes:	Major version (in decimal)
Lower-order 2 bytes:	Minor version (in decimal)

Description

R_RSIP_CM_GetVersion outputs the driver version.

4.1.2 Key Injection

4.1.2.1 R_RSIP_CM_AESxxx_InitialKeyWrap

Format

```
(1)  fsp_err_t R_RSIP_CM_AES128_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)
(2)  fsp_err_t R_RSIP_CM_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)
```

Parameters

key_injection_type	Input	Type of the key to be input to p_user_key (1) RSIP_KEY_INJECTION_TYPE_ENCRYPTED(0): Encrypted key (2) RSIP_KEY_INJECTION_TYPE_PLAIN (1): Plaintext key
p_wrapped_user_factory_programming_key	Input	W-UFPK This setting is not required when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type.
p_initial_vector	Input	Initialization vector This setting is not required when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type.
p_user_key	Input	User key Encrypted user key for input when RSIP_KEY_INJECTION_TYPE_ENCRYPTED is specified as key_injection_type Plaintext user key for input when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type
p_wrapped_key	Output	AES wrapped key

Return Values: fsp_err_t

FSP_SUCCESS	Successful completion
FSP_ERR_CRYPTO_SCE_FAIL	An input parameter was incorrect.
FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT	The hardware resource required for this process is in use by another process so resource contention between the processes prevented the key injection processing.

Description

The output of R_RSIP_CM_AES128_InitialKeyWrap API function is a 128-bit AES wrapped key.
The output of R_RSIP_CM_AES256_InitialKeyWrap API function is a 256-bit AES wrapped key.
When RSIP_KEY_INJECTION_TYPE_ENCRYPTED (0) is specified as key_injection_type, specify the W-UFPK generated from the UFPK that was used to wrap the user key as p_wrapped_user_factory_programming_key, the initialization vector that was used to wrap the user key as p_initial_vector, and the encrypted user key as p_user_key.

When RSIP_KEY_INJECTION_TYPE_PLAIN (1) is specified as key_injection_type, the settings for p_wrapped_user_factory_programming_key and p_initial_vector are not required. Specify the plaintext user key as p_user_key.
When specifying a user key as p_user_key, input data as shown in section 7.3.1, AES in section 7.3, User Key Formats.

4.1.2.2 R_RSIP_CM_ECC_xxx_InitialPrivateKeyWrap

Format

```
(1)  fsp_err_t R_RSIP_CM_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_aes_wrapped_key_t * const    p_wrapped_key)
(2)  fsp_err_t R_RSIP_CM_ECC_secp256k1_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_aes_wrapped_key_t * const    p_wrapped_key)
(3)  fsp_err_t R_RSIP_CM_ECC_brainpoolP256r1_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_aes_wrapped_key_t * const    p_wrapped_key)
```

Parameters

key_injection_type	Input	Type of the key to be input to p_user_key (1) RSIP_KEY_INJECTION_TYPE_ENCRYPTED(0): Encrypted key (2) RSIP_KEY_INJECTION_TYPE_PLAIN (1): Plaintext key
p_wrapped_user_factory_programming_key	Input	W-UFPK This setting is not required when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type.
p_initial_vector	Input	Initialization vector This setting is not required when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type.
p_user_key	Input	User key Encrypted user key for input when RSIP_KEY_INJECTION_TYPE_ENCRYPTED is specified as key_injection_type Plaintext user key for input when RSIP_KEY_INJECTION_TYPE_PLAIN is specified as key_injection_type
p_wrapped_key	Output	Wrapped key of a 256-bit ECC private key

Return Values: fsp_err_t

FSP_SUCCESS	Successful completion
FSP_ERR_CRYPTO_SCE_FAIL	An input parameter was incorrect.
FSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT	The hardware resource required for this process is in use by another process so resource contention between the processes prevented the key injection processing.

Description

The output of R_RSIP_CM_ECC_secp256r1_InitialPrivateKeyWrap API function is a wrapped key of a 256-bit ECC-secp256r1 private key.
The output of R_RSIP_CM_ECC_secp256k1_InitialPrivateKeyWrap API function is a wrapped key of a 256-bit ECC-secp256k1 private key.
The output of R_RSIP_CM_ECC_brainpoolP256r1_InitialPrivateKeyWrap API function is a wrapped key of a 256-bit ECC-brainpoolP256r1 private key.
When RSIP_KEY_INJECTION_TYPE_ENCRYPTED (0) is specified as key_injection_type, specify the W-UFPK generated from the UFPK that was used to wrap the user key as p_wrapped_user_factory_programming_key, the initialization vector that was used to wrap the user key as p_initial_vector, and the encrypted user key as p_user_key.
When RSIP_KEY_INJECTION_TYPE_PLAIN (1) is specified as key_injection_type, the settings for p_wrapped_user_factory_programming_key and p_initial_vector are not required.
When specifying a user key as p_user_key, input data as shown in section 7.3.2, ECC in section 7.3, User Key Formats.

5. Key Injection

This section describes how to program encryption keys handled by the RSIP CM driver in nonvolatile memory such as flash memory. Wrapped keys are used in key injection by default. The current version of the RSIP CM driver only supports the use of plaintext keys as ECC public keys.

5.1 Key Injection

The procedure for safely injecting keys into your products as part of the manufacturing process is given below.

See section 3.5.1, Key Injection for details on the mechanism for key injection by the RSIP CM driver.

The Renesas Key Wrap Service provided by Renesas and a key injection program running on an RX family MCU are required to inject a user key. Supplementary tools such as the Security Key Management Tool are also available for use to simplify the process.

The demo project which comes with this application note includes a sample key injection program, which can be used for reference.

The procedure for implementing user key injection is as follows.

1. Preparing the key data required for injecting a user key

   Use a desired tool to prepare a 256-bit UFPK and 128-bit IV. These are to be used in wrapping the user key to be injected. OpenSSL is used to generate the UFPK and IV in the example below.

   ```
   > openssl rand 32 > ufpk.bin
   > openssl rand 16 > iv.bin
   ```

   Use the Renesas Key Wrap Service (https://dlm.renesas.com/keywrap/toEnglish) to generate a W-UFPK by using an HRK to wrap ufpk.bin. For detailed information, refer to the Renesas Key Wrap Service FAQ.

   Follow the procedure for use of the key wrapping algorithms described in section 3.5.1 to generate an encrypted key by using the UFPK (ufpk.bin) to wrap the user key. For details on the formats of user keys, see section 7.3.

2. Creating a user key injection program

   Input the encrypted key, ufpk.bin, and iv.bin generated in step 1 to the corresponding key injection API function from among those provided for each type of cryptographic algorithm to generate a wrapped key from the user key, and create a program to write the result to nonvolatile memory.

   For the API function to be used, see Figure 3-4, User Key Wrapping Scheme during Key Injection.

3. Injecting the key

   Run the user key injection program on a RX family MCU to inject the user key into the flash memory. We recommend erasing the data for use with key injection in the user key injection program on completion of the key injection.

The Secure Key Management Tool is available as a supplementary tool for performing steps 1 and 2. See section 5.2 for details of this tool.

5.2 Using the Security Key Management Tool to Generate an Encrypted Key

The Security Key Management Tool can be used to generate an encrypted user key.

A command line interface (CLI) version of the Security Key Management Tool is also available. This eases its use in production processes such as those at a factory.

Security Key Management Tool

https://www.renesas.com/en/software-tool/security-key-management-tool

For details on how to use the Security Key Management Tool, see the user's manual.

5.2.1 Key Injection Procedure

The procedure for generating a key file to be used in key injection is described below.

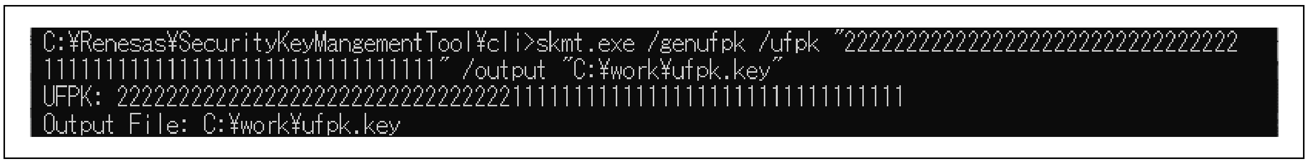
In the following example, C source code for AES is generated.

5.2.1.1 Procedure for Using the CLI Version

1. Generating a UFPK

   Use the terminal software to execute the genufpk command.

```
> skmt.exe /genufpk
            /ufpk "2222222222222222222222222222222211111111111111111111111111111111"
            /output "C:¥work¥ufpk.key"
```



```
C:¥Renesas¥SecurityKeyMangementTool¥cli>skmt.exe /genufpk /ufpk "22222222222222222222222222222222
11111111111111111111111111111111" /output "C:¥work¥ufpk.key"
UFPK: 2222222222222222222222222222222211111111111111111111111111111111
Output File: C:¥work¥ufpk.key
```

Figure 5-1 Result of Executing the genufpk Command

2. Obtaining a W-UFPK

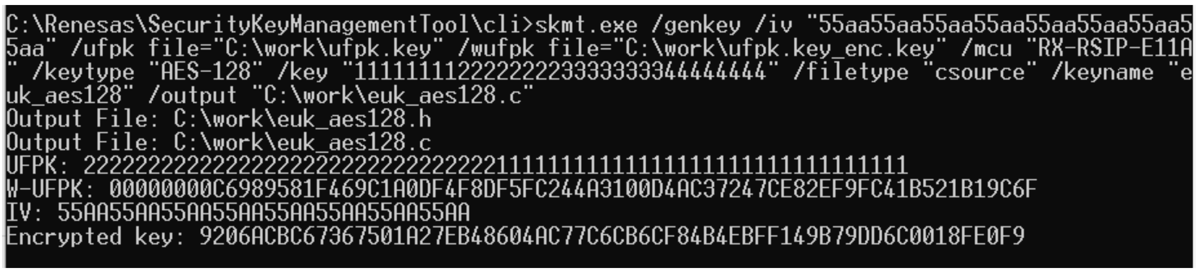
   Send the ufpk.key file generated in step 1 to the Renesas Key Wrap Service (https://dlm.renesas.com/keywrap/toEnglish) to obtain a W-UFPK.
   For detailed information on obtaining the file, refer to the Renesas Key Wrap Service FAQ.

3. Generating an AES-128 key file as a C source file

   Use the terminal software to execute the genkey command.

```
> skmt.exe /genkey /iv "55aa55aa55aa55aa55aa55aa55aa55aa" /ufpk file="C:¥work¥ufpk.key"
            /wufpk file="C:¥work¥ufpk.key_enc.key" /mcu "RX-RSIP-E11A" /keytype "AES-128"
            /key "11111111222222223333333344444444" /filetype "csource" /keyname
            "euk_aes128" /output "C:¥work¥euk_aes128.c"
```

Use the UFPK file generated in step 1 and the W-UFPK file generated in step 2.

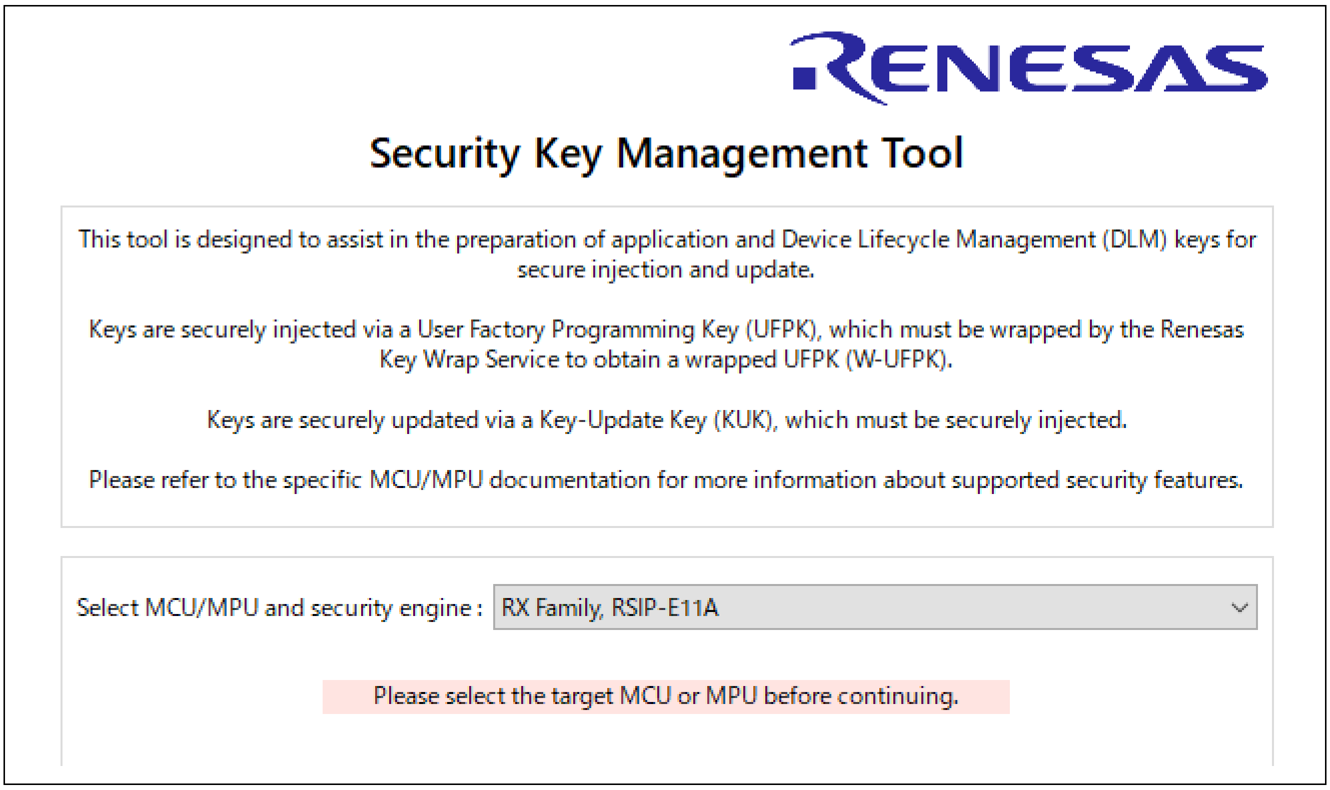


```
C:\Renesas\SecurityKeyManagementTool\cli>skmt.exe /genkey /iv "55aa55aa55aa55aa55aa55aa55aa5
5aa" /ufpk file="C:\work\ufpk.key" /wufpk file="C:\work\ufpk.key_enc.key" /mcu "RX-RSIP-E11A
" /keytype "AES-128" /key "11111111222222223333333344444444" /filetype "csource" /keyname "e
uk_aes128" /output "C:\work\euk_aes128.c"
Output File: C:\work\euk_aes128.h
Output File: C:\work\euk_aes128.c
UFPK: 2222222222222222222222222222222211111111111111111111111111111111
W-UFPK: 00000000C6989581F469C1A0DF4F8DF5FC244A3100D4AC37247CE82EF9FC41B521B19C6F
IV: 55AA55AA55AA55AA55AA55AA55AA55AA
Encrypted key: 9206ACBC67367501A27EB48604AC77C6CB6CF84B4EBFF149B79DD6C0018FE0F9
```

Figure 5-2 Example of the Display Produced by Executing the genkey Command

5.2.1.2 Procedure for Using the GUI Version

1. Selecting the MCU or MPU and an encryption engine
   On the [**Overview**] tabbed page, select the MCU or MPU and an encryption engine.



Security Key Management Tool

This tool is designed to assist in the preparation of application and Device Lifecycle Management (DLM) keys for secure injection and update.

Keys are securely injected via a User Factory Programming Key (UFPK), which must be wrapped by the Renesas Key Wrap Service to obtain a wrapped UFPK (W-UFPK).

Keys are securely updated via a Key-Update Key (KUK), which must be securely injected.

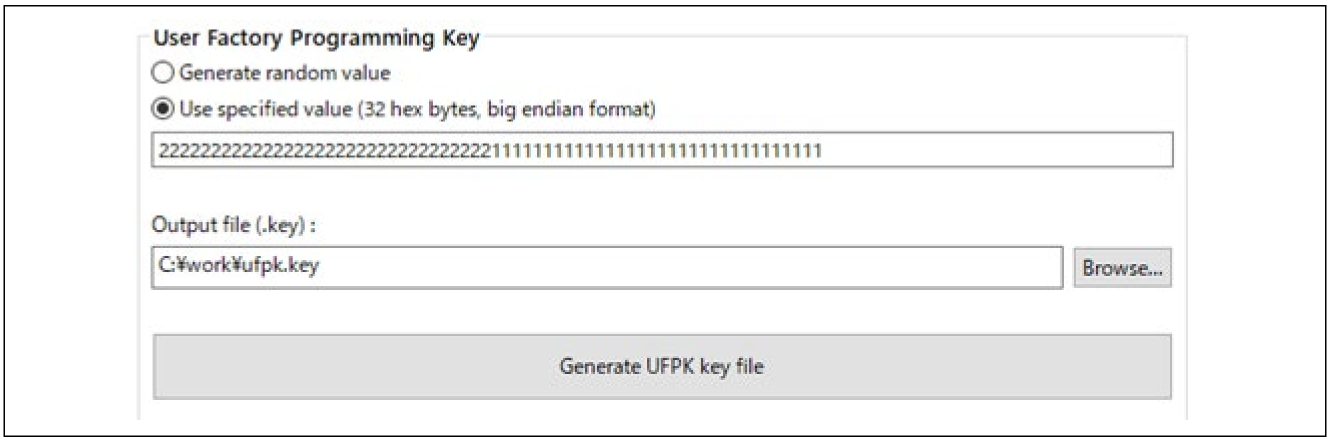
Please refer to the specific MCU/MPU documentation for more information about supported security features.

Select MCU/MPU and security engine : RX Family, RSIP-E11A

Please select the target MCU or MPU before continuing.

Figure 5-3 [**Overview**] Tabbed Page

2. Generating a UFPK
   To generate a UFPK file, enter a UFPK value and a file name with the .key filename extension on the [**Generate UFPK**] tabbed page. The filename is ufpk.key in this example.



User Factory Programming Key

○ Generate random value

● Use specified value (32 hex bytes, big endian format)

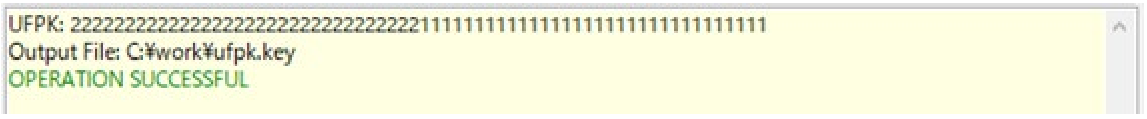
2222222222222222222222222222222211111111111111111111111111111111

Output file (.key) :

C:¥work¥ufpk.key

Figure 5-4 Example of UFPK Generation by Specifying Values on the [**Generate UFPK**] Tabbed Page

Clicking on the [**Generate UFPK key file**] button generates the UFPK file. The result of execution shown on the following page is output when the file has been successfully generated.

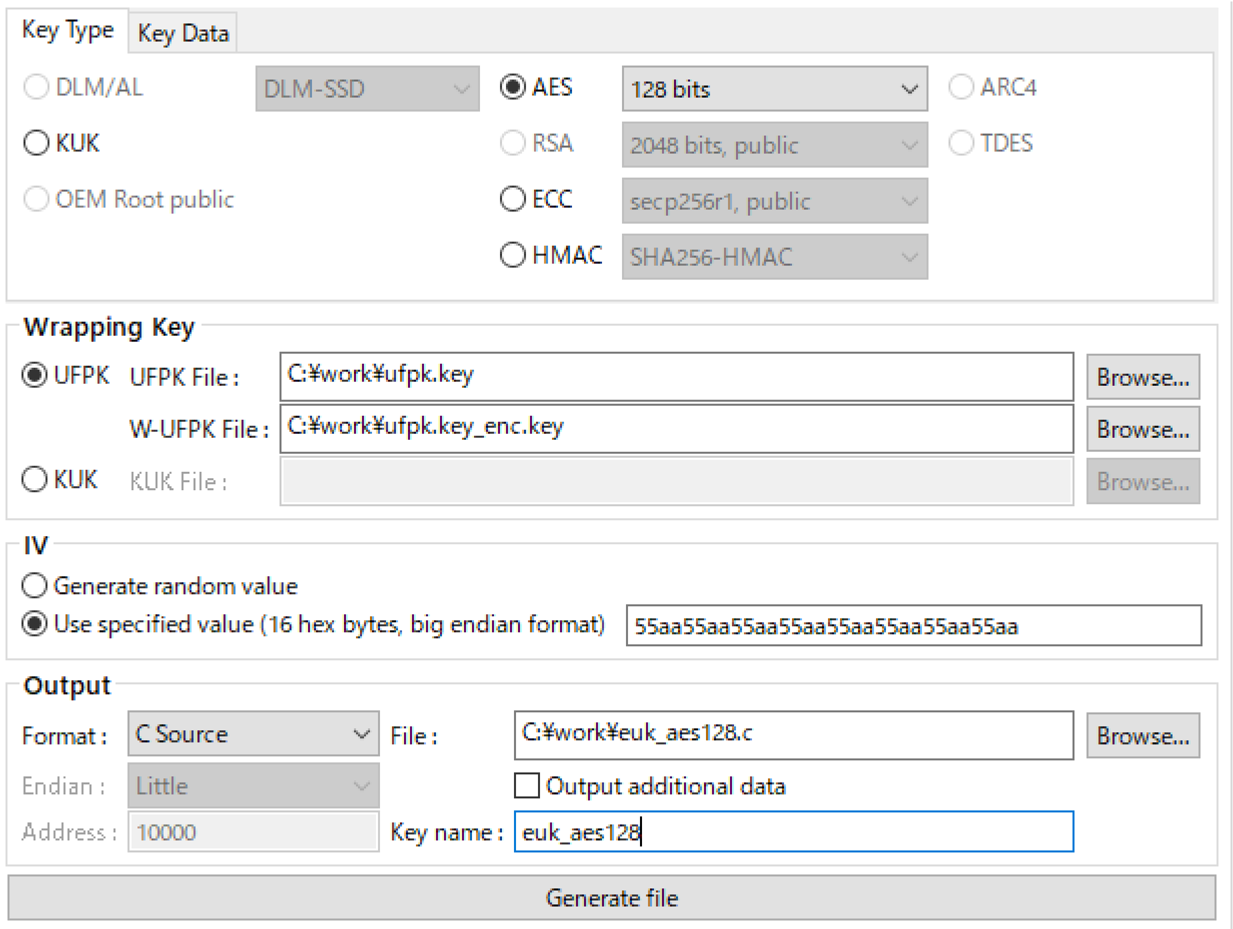


```
UFPK: 2222222222222222222222222222222211111111111111111111111111111111
Output File: C:¥work¥ufpk.key
OPERATION SUCCESSFUL
```

Figure 5-5 Result of Execution on the [**Generate UFPK**] Tabbed Page

3. Obtaining a W-UFPK
   Send the ufpk.key file generated in step 2 to the Renesas Key Wrap Service (https://dlm.renesas.com/keywrap/toEnglish) to obtain a W-UFPK.
   For detailed information on obtaining the file, refer to the Renesas Key Wrap Service FAQ.

4. Generating an AES-128 key file as a C source file
   Generate an AES-128 key file on the [**Wrap Key**] tabbed page.
   Select "AES" and "128 bits" on the [**Key Type**] tabbed page, and then enter the value of the AES-128 key on the [**Key Data**] tabbed page.
   Next, on the [**Key Type**] tabbed page, enter the UFPK file generated in step 2 and the W-UFPK file obtained in step 3 in the [**Wrapping Key**] section and specify "C Source" as "**Format**" in the [**Output**] section.



Key Type | Key Data

DLM/AL | DLM-SSD | [✓ AES] | 128 bits | ARC4
KUK | RSA | 2048 bits, public | TDES
OEM Root public | ECC | secp256r1, public
HMAC | SHA256-HMAC

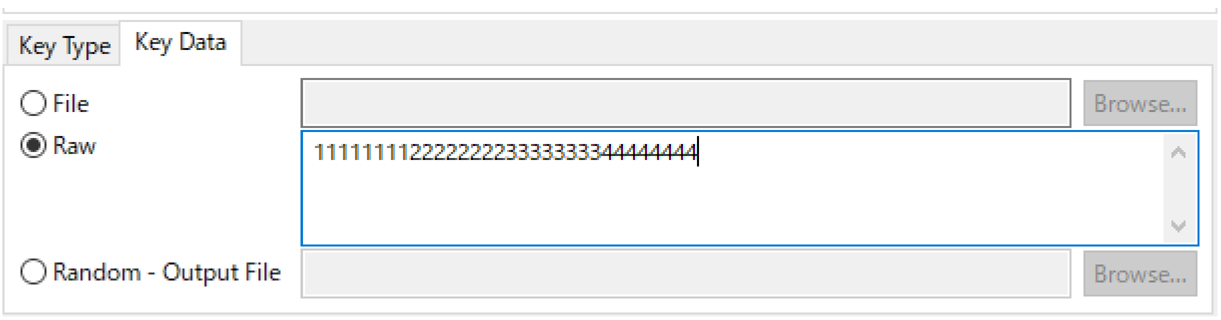
Wrapping Key
[✓ UFPK] UFPK File : C:¥work¥ufpk.key Browse...
W-UFPK File : C:¥work¥ufpk.key_enc.key Browse...
KUK KUK File : Browse...

IV
Generate random value
[✓ Use specified value (16 hex bytes, big endian format)] 55aa55aa55aa55aa55aa55aa55aa55aa

Output
Format : C Source File : C:¥work¥euk_aes128.c Browse...
Endian : Little Output additional data
Address : 10000 Key name : euk_aes128

Generate file

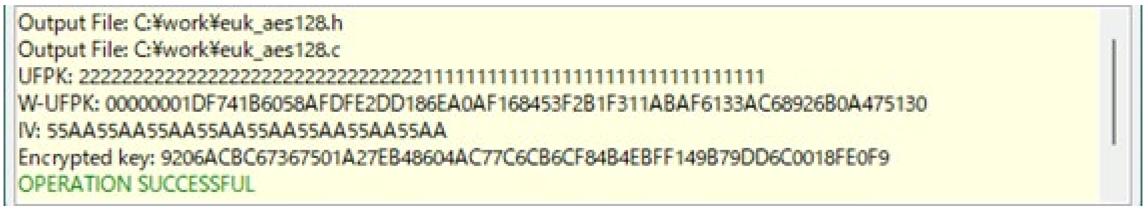
Figure 5-6 Example of Settings for Output of an AES-128 Key File as a C Source File on Tabbed Page [**Key Type**] within [**Wrap Key**]



Raw: 11111111222222223333333344444444
Random - Output File

Figure 5-7 Example of Settings for Output of an AES-128 Key File as a C Source File on Tabbed Page [**Key Data**] within [**Wrap Key**]

The result of execution shown below is output on successful completion of these operations.



```
Output File: C:¥work¥euk_aes128.h
Output File: C:¥work¥euk_aes128.c
UFPK: 2222222222222222222222222222222211111111111111111111111111111111
W-UFPK: 00000001DF741B6058AFDFE2DD186EA0AF168453F2B1F311ABAF6133AC68926B0A475130
IV: 55AA55AA55AA55AA55AA55AA55AA55AA
Encrypted key: 9206ACBC67367501A27EB48604AC77C6CB6CF84B4EBFF149B79DD6C0018FE0F9
OPERATION SUCCESSFUL
```

Figure 5-8 Result of Executing Operations on the **[Wrap Key**] Tabbed Page

Inject data in the output C source file in the same way as was described in section 5.2.1.1.

6. Sample Program

6.1 Key Injection and Cipher Usage

A demo project shown in Table 6-1 can be used to confirm the usage of API functions provided by the RSIP driver for cryptographic operations, random number generation, and key injection.

Table 6-1 Demo Project for Key Injection and Cipher Usage

MCU	Demo Project
RX261	rx261_ek_rsip_cm_sample

Results of executing the demo project are output through a UART. Connect a PC on which terminal software has been installed to the board on which the demo project is to run. In descriptions in this section, Tera Term is used as the terminal software on the PC.

The demo project handles data as little endian.

6.1.1 Setting up the Demo Project

Connections between the board and PC are shown below.

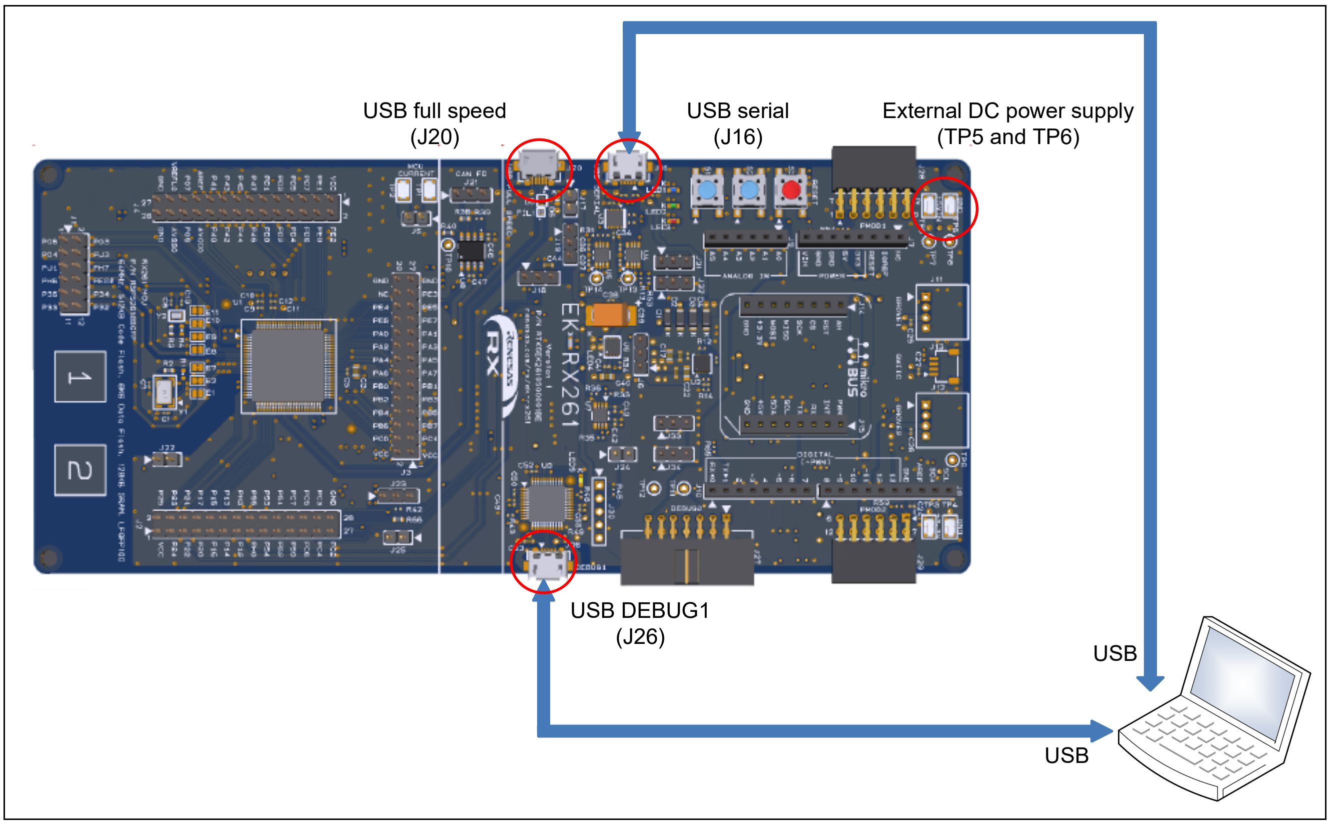


Figure 6-1 Connections between the EK-RX261 and PC

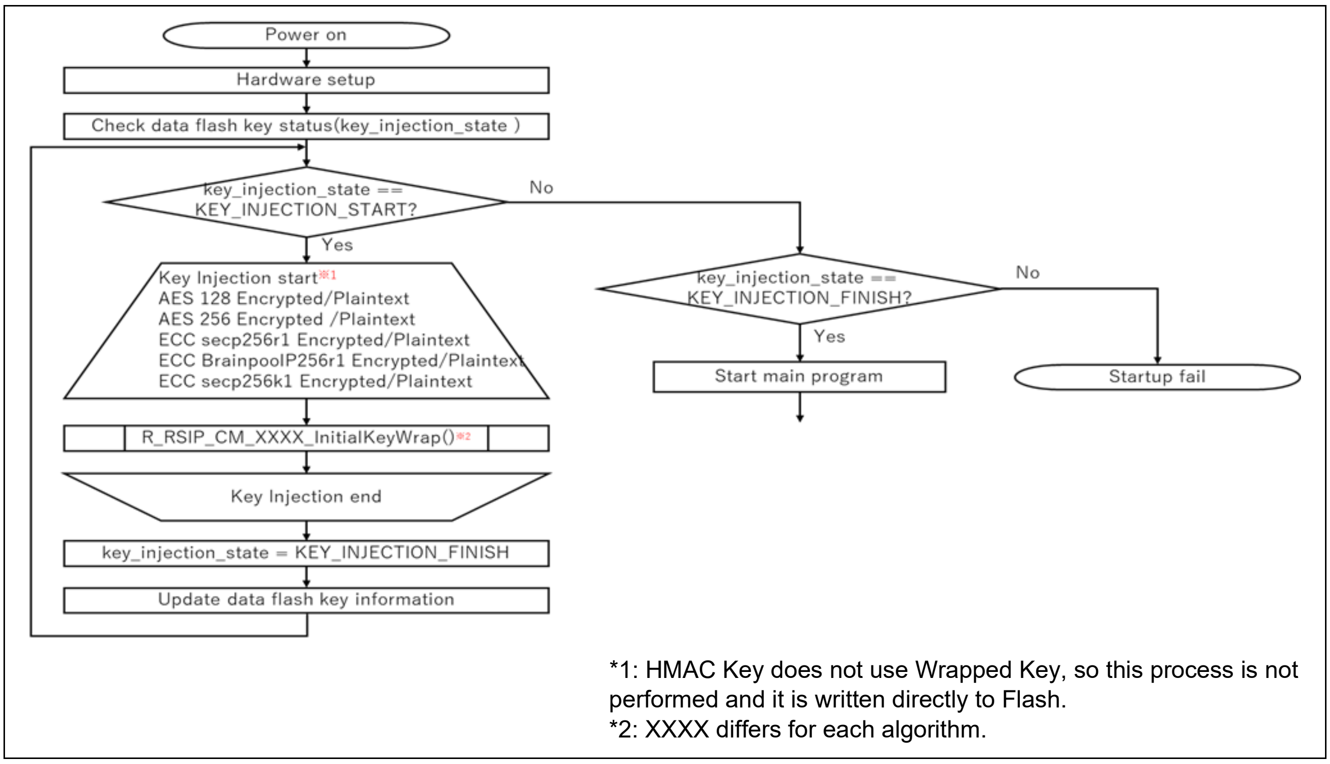
Power to the EK-RX261 board can be supplied from the pins of the USB DEBUG1, USB full speed, or USB serial port, or through those for an external DC power supply. Use one from among these to supply power to the EK-RX261 board.

The following lists the serial port and terminal settings of Tera Term.

- Bit rate: 115200 bps
- Data length: 8 bits
- Parity bits: None
- Stop bit: 1
- Newline code (for transmission): Carriage return (CR)

6.1.2 Overview of the Demo Project

Figure 6-2 shows the flow of operation of the demo project.



Power on
Hardware setup
Check data flash key status(key_injection_state)
key_injection_state == KEY_INJECTION_START?
No
Yes
Key Injection start※1
AES 128 Encrypted/Plaintext
AES 256 Encrypted /Plaintext
ECC secp256r1 Encrypted/Plaintext
ECC BrainpoolP256r1 Encrypted/Plaintext
ECC secp256k1 Encrypted/Plaintext
R_RSIP_CM_XXXX_InitialKeyWrap()※2
Key Injection end
key_injection_state = KEY_INJECTION_FINISH
Update data flash key information
key_injection_state == KEY_INJECTION_FINISH?
No
Yes
Start main program
Startup fail

*1: HMAC Key does not use Wrapped Key, so this process is not performed and it is written directly to Flash.
*2: XXXX differs for each algorithm.

Figure 6-2 Flow of Operation of the Demo Project for Key Injection and Cipher Usage

The demo project status and the keys used are managed in the data flash (key_block_data). As a measure against power outages during key injection or data updates, data is stored in two areas, the main area and the mirror area, in the data flash for data management.

Table 6-2 Key Data Structures of the Demo Project

Type	Name	Description
st_key_block_ data_t	(key_block_data structure)	The key data structure stored in data flash
uint32_t	key_injection_status	Indicates the following STATE KEY_INJECTION_START KEY_INJECTION_FINISH See the following table for details
(struct)	key_data	Key Data Storage Structure
rsip_aes_wrapped_key_t	user_aes128_key_index_encrypted	Encrypted key of AES 128
rsip_aes_wrapped_key_t	user_aes128_key_index_plaintext	Plaintext Key of AES 128
rsip_aes_wrapped_key_t	user_aes256_key_index_encrypted	Encrypted Key of AES 256
rsip_aes_wrapped_key_t	user_aes256_key_index_plaintext	Plaintext Key of AES 256
uint8_t[]	user_sha224hmac_key_index_plaintext	Plaintext Key of HMAC SHA-224
uint8_t[]	user_sha256ac_key_index_plaintext	Plaintext Key of HMAC SHA-256
rsip_ecc_private_wrapped_ key_t	user_ecc_secp256r1_private_key_index_enc rypted	Encrypted Key of ECDSA secp256r1
rsip_ecc_private_wrapped_ key_t	user_ecc_secp256r1_private_key_index_plai ntext	Plaintext Key of HMAC SHA-256 of ECDSA secp256r1
rsip_ecc_private_wrapped_ key_t	user_ecc_brainpoolp256r1_private_key_inde x_encrypted	Encrypted Key of ECDSA Brainpoolp256r1

Type	Name	Description
rsip_ecc_private_wrapped_key_t	user_ecc_brainpoolp256r1_private_key_index_plaintext	Plaintext Key of ECDSA Brainpoolp256r1
rsip_ecc_private_wrapped_key_t	user_ecc_secp256k1_private_key_index_encrypted	Encrypted Key of ECDSA secp256k1
rsip_ecc_private_wrapped_key_t	user_ecc_secp256k1_private_key_index_plaintext	Plaintext Key of ECDSA secp256k1

The behavior of the demo project is managed with the state transition and the state is managed with the flag in the data flash.

Table 6-3 State of the Demo Project

State	Operation content
KEY_INJECTION_START	Inject the wrapped keys to the data flash. In the demo project, the area to store the keys I the data flash is dvided to main area and mirror area. The key data can be recovered with the structure if the power shutoff has occurred while writing the keys. After injecting the keys, transit the state to KEY_INJECTION_FINISH. The key injection is executed only in the first time of the start operation. After the time, the state begins with the state KEY_INJECTION_FINISH.
KEY_INJECTION_FINISH	Confirm the key data in the data flash with checking its hash value. When the validity is confirmed, the project begins accepting the commands.

The demo project incorporates the following commands.

Table 6-4 List of Commands of the Demo Project

Command	Operation
display	Displays the generated key type in terms of whether the key is wrapped or plaintext.
encdemo-encrypted [Arg1]	Encrypts the value of Arg1 in AES-128 ECB mode (by using an encrypted key).
encdemo-plaintext [Arg1]	Encrypts the value of Arg1 in AES-128 ECB mode (by using a plaintext key).
function	Executes the various types of processing listed below and confirms operation of the relevant API functions. • AES-128 or AES-256 ECB, CBC, CTR, CCM, or GCM encryption and decryption • AES-128 or AES-256 CMAC generation and verification • SHA224 or SHA256 HMAC generation and verification • ECDSA P256 signature generation and verification
random	Generates a pseudo-random number.

The default values of keys are as follows and IV is “55aa55aa55aa55aa55aa55aa55aa55aa” in the demo project.

Table 6-5 List of Commands of the Demo Project

key type	value
UFPK	2222222222222222222222222222222211111111111111111111111111111111
AES-128bit	11111111222222223333333344444444
AES-256bit	ffffffffeeeeeeeeddddddddccccccccbbbbbbbbaaaaaaaa0000000099999999
HMAC SHA-224	000102030405060708090a0b0c0d0e0f101112131415161718191a1b
HMAC SHA-256	000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
ECC secp256r1 private	519b423d715f8b581f4fa8ee59f4771a5b44c8130b4e3eacca54a56dda72b464
ECC secp256r1 public	1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83 ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9
ECC BrainpoolP256 r1 private	01e98cf5a934e4f829f94a3fa4ec18ab421fbaa40e0da1f7a3cad1d7ac38cf4f
ECC BrainpoolP256 r1 public	5c568a2b6d3d9828734f9793507ce15629d37d357586e789faf07aeb69c50207 9a61f3fad6defedcb410a709055ca1a033422a9db29f554548438d6f5138d375
ECC secp256k1 private	ef871c07ce463c95384f1f388ef0107b655998623365eab0debdf9a94a7d2303
ECC secp256k1 public	1f9dfb391adade2c8fcbfef52e3115982e19c79ff0571c988a06aec223fb58a6 13898b7f7123287906f43ba6bfbf920aa49377e0ad400d3b8bd94dbb7245c025

The input values used by the function command are shown below.

Table 6-6 List of Commands of the Demo Project

data type	value
AES ECB/CBC/CTR/ CCM plaintext	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb ccccccccccccccccccccccccccccccccdddddddddddddddddddddddddddddddd
AES CBC IV	12345678fedcba0955555555aaaaaaaa
AES CTR Counter	12345678fedcba0955555555aaaaaaaa
AES CCM Nonce	101112131415161718191a1b
AES CCM AAD	000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f 404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f 606162636465666768696a6b
AES GCM plaintext	aaaaaaaabbbbbbbbccccccccdddddddd
AES GCM IV	b3d8cc017cbb89b39e0f67e2
AES GCM AAD	24825602bd12a984e0092d3e448eda5f
HMAC SHA-224/SHA-256 plaintext	53616D706C65206D65737361676520666F72206B65796C656E3C626C6F636 B6C656E
SHA-224/ SHA-256 plaintext	6162636462636465636465666465666765666768666768696768696a68696a6b 696a6b6c6a6b6c6d6b6c6d6e6c6d6e6f6d6e6f706e6f7071
ECC secp256r1/BrainpoolP256r1/secp256k1 plaintext	5905238877c77421f73e43ee3da6f2d9e2ccad5fc942dcec0cbd25482935faaf 416983fe165b1a045ee2bcd2e6dca3bdf46c4310a7461f9a37960ca672d3feb5 473e253605fb1ddfd28065b53cb5858a8ad28175bf9bd386a5e471ea7a65c17c c934a9d791e91491eb3754d03799790fe2d308d16146d5c9b0d0debd97d79ce8

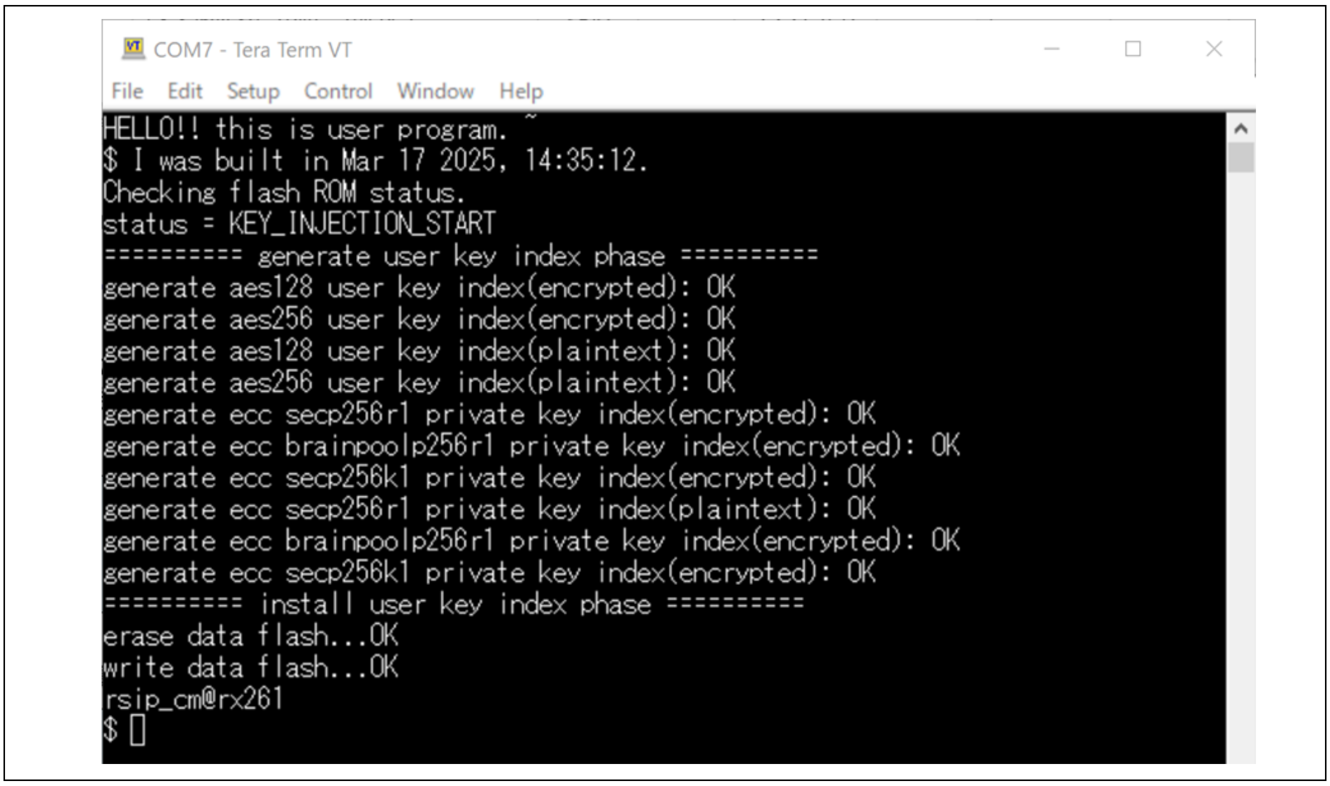
6.1.2.1 Confirmation of Keys and the Demo Project

The demo project generates a wrapped key from a user key. The wrapped key is generated with the use of an HUK, so copying a wrapped key generated for a chip and using the result with another chip is not possible. This means that the use of dead copies of the user key is prevented. You can confirm that attempting to use a wrapped key generated for another device to run the demo project produces an error in the RSIP CM driver.

In addition, wrapped keys include their own random numbers, which makes guessing the original user key from a wrapped key infeasible. Specifically, even if wrapped keys are generated for the same device and with the use of the same user key, the results will each have different values. You can use the demo project to confirm this feature of wrapped keys. Every time you download the demo project and execute the display command, you can confirm the change in the value of the wrapped key.

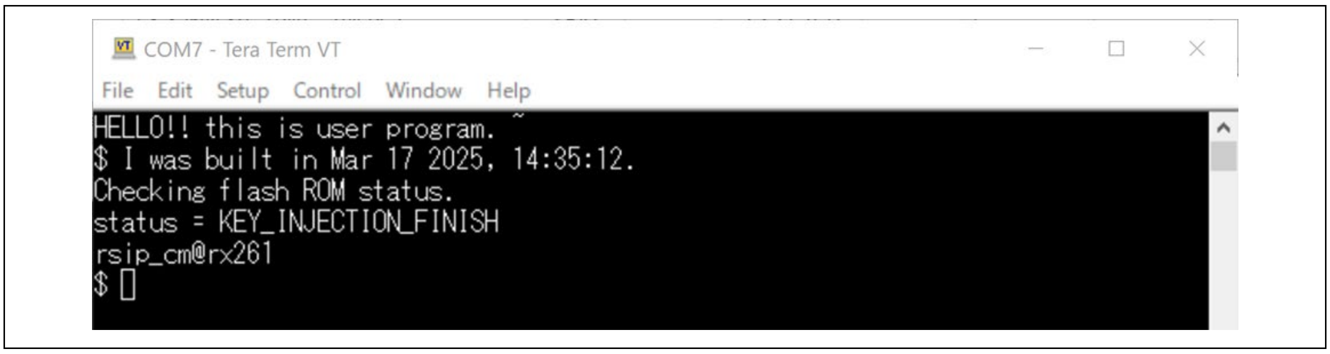
6.1.3 Example of Executing the Demo Project

Figure 6-3 shows the display produced when key injection is successfully completed during start-up after the program has been downloaded to the MCU. After the listing below is displayed, the program is ready to accept the commands listed in Table 6-4.



```
HELLO!! this is user program. ~
$ I was built in Mar 17 2025, 14:35:12.
Checking flash ROM status.
status = KEY_INJECTION_START
========== generate user key index phase ==========
generate aes128 user key index(encrypted): OK
generate aes256 user key index(encrypted): OK
generate aes128 user key index(plaintext): OK
generate aes256 user key index(plaintext): OK
generate ecc secp256r1 private key index(encrypted): OK
generate ecc brainpoolp256r1 private key index(encrypted): OK
generate ecc secp256k1 private key index(encrypted): OK
generate ecc secp256r1 private key index(plaintext): OK
generate ecc brainpoolp256r1 private key index(encrypted): OK
generate ecc secp256k1 private key index(encrypted): OK
========== install user key index phase ==========
erase data flash...OK
write data flash...OK
rsip_cm@rx261
$ ▯
```

Figure 6-3 Display in the Tera Term Window (at the Time of Start-Up)



```
HELLO!! this is user program. ~
$ I was built in Mar 17 2025, 14:35:12.
Checking flash ROM status.
status = KEY_INJECTION_FINISH
rsip_cm@rx261
$ ▯
```

Figure 6-4 Display in the Tera Term Window (2nd and Subsequent Executions)

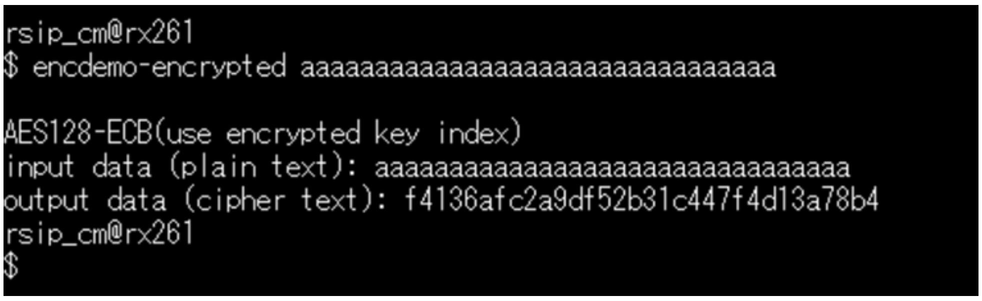
An example of using the encdemo-encrypted and encdemo-plaintext command is given below as an example of command usage.

The encdemo-encrypted command uses a encrypted wrapped 128-bit AES key that has been injected to encrypt the value input as the argument of the command in AES ECB mode.

The encdemo-plaintext command uses a plaintext wrapped 128-bit AES key that has been injected to encrypt the value input as the argument of the command in AES ECB mode.

The key value “11111111222222223333333344444444” has been injected as the 128-bit AES key into the sample program beforehand.

Figure 6-5 shows an example of executing the encdemo-encrypted command with the argument “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”.

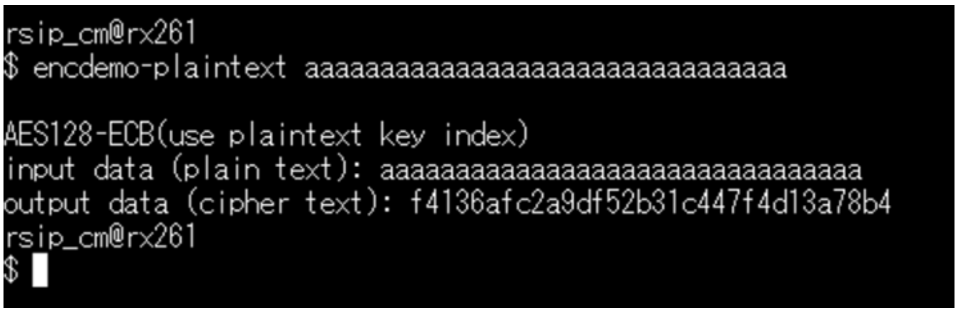


```
rsip_cm@rx261
$ encdemo-encrypted aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

AES128-ECB(use encrypted key index)
input data (plain text): aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
output data (cipher text): f4136afc2a9df52b31c447f4d13a78b4
rsip_cm@rx261
$
```

Figure 6-5 Display in the Tera Term Window (at the Time of encdemo-encrypted Command Execution)

Figure 6-6 shows an example of executing the encdemo-plaintext command with the argument “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”.



```
rsip_cm@rx261
$ encdemo-plaintext aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

AES128-ECB(use plaintext key index)
input data (plain text): aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
output data (cipher text): f4136afc2a9df52b31c447f4d13a78b4
rsip_cm@rx261
$
```

Figure 6-6 Display in the Tera Term Window (at the Time of encdemo-plaintext Command Execution)

7. Appendix

7.1 Environments for Confirming Operation

The table below lists the environments used in confirming operation of this driver.

Table 7-1 Environments for Confirming Operation

Item	Description
Integrated development environments	e^2 studio 2025-01 from Renesas Electronics IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	C/C++ Compiler for RX Family (CC-RX) V3.07.00 from Renesas Electronics Compilation options: The following option was added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202411 Compilation options: The following option was added to the default settings of the integrated development environment. -std = gnu99
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compilation options: Default settings of the integrated development environment
Endian for operation	Little
Module version	Ver.1.00
Board used	EK-RX261 (Part number: RTK5EK2610SxxxxxBJ)

7.2 Troubleshooting

(1) Q: I added the FIT module to my project, but when I attempted to build it, an error message, “Could not open source file ‘platform.h’,” appeared.

A: The FIT module may not have been correctly added to your project. Refer to the appropriate document of the two listed below to confirm how to add the FIT module to a project.

- When CS+ is in use
  Application note *RX Family: Adding Firmware Integration Technology Modules to CS+ Projects* (R01AN1826)
- When the e^2 studio is in use
  Application note *RX Family: Adding Firmware Integration Technology Modules to Projects* (R01AN1723)

When the FIT module is to be used, the board support package FIT module (BSP module) must also be added to the project. For details on how to add the BSP module, refer to the application note *RX Family: Board Support Package Module Using Firmware Integration Technology* (R01AN1685).

(2) Q: I want to use CS+ to run the sample project for use with the e^2 studio from the FITDemos directory.

A: Refer to the Web page at the URL below.
Porting from the e^2 studio to CS+
> Convert an Existing Project to Create a New Project With CS+
https://www.renesas.com/en/software-tool/migration-e2studio-to-csplus

Note: In step 5, the **[Q0268002]** dialog box may appear if the [Backup the project composition files after conversion] checkbox is checked. Clicking on the **[Yes]** button in the **[Q0268002]** dialog box requires re-setting the include paths for the compiler.

7.3 User Key Formats

This section describes the data formats for user keys (plaintext user keys) and for encrypted user keys.

7.3.1 AES

128-bit AES key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 15	128-bit AES key			

Input (Encrypted User Key)

Bytes	16			
	4	4	4	4
0 to 15	encrypted_user_key (128-bit AES key)			
16 to 31	MAC			

256-bit AES key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 31	256-bit AES key			

Input (Encrypted User Key)

Bytes	16			
	4	4	4	4
0 to 31	encrypted_user_key (256-bit AES key)			
32 to 47	MAC			

7.3.2 ECC

ECC secp256r1/secp256k1/brainpoolP256r1 public key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 31	ECC 256-bit public key Qx			
32 to 63	ECC 256-bit public key Qy			

ECC secp256r1/secp256k1/brainpoolP256r1 private key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 31	ECC 256-bit private key d			

Input (Encrypted User Key)

Bytes	16			
	4	4	4	4
0 to 31	encrypted_user_key (ECC P 256-bit private key d)			
32 to 47	MAC			

7.3.3 HMAC

HMAC-SHA224 key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 31	HMAC-SHA224 key			
				0 padding

HMAC-SHA256 key

Input (Plaintext User Key)

Bytes	16			
	4	4	4	4
0 to 31	HMAC-SHA256 key			

8. Reference Documents

User’s Manual: Hardware
(The latest version can be downloaded from the Renesas Electronics Web site.)

Technical Updates/Technical News
(The latest information can be downloaded from the Renesas Electronics Web site.)

User’s Manual: Development Environment
RX Family CC-RX Compiler User’s Manual (R20UT3248)
(The latest version can be downloaded from the Renesas Electronics Web site.)

Web Site and Support Desk

Renesas Electronics Web site

https://www.renesas.com/en

Contact information

https://www.renesas.com/en/contact-us

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Apr.25, 2025	-	First edition issued.

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.