

RX Family

Implementing TLS Using TSIP Driver

Introduction

The Trusted Secure IP (TSIP) driver supports APIs for SSL/TLS (referred to below as TLS) communication. This document describes the TLS APIs of the TSIP driver and how to implement them in user programs. A sample project based on FreeRTOS is appended to this document. The sample project incorporates the TSIP driver as well as FreeRTOS, including Mbed TLS, and can be used to test MQTT communication with Amazon Web Services (AWS).

Devices on Which Operation Confirmed

The operation of the sample program appended to this document has been confirmed on the following devices.

- RX72N: R5F572NDHDFB

Operating Environment

The operation of the sample program appended to this document has been confirmed on the following environment.

IDE	e ² studio 2021-04
Toolchain	CCRX compiler v3.0.3 GCC for Renesas 8.3.0.202002-GNURX
Target board	RX72N Envision Kit (product No.: RTK5RX72N0C00000BJ)
Debugger	E2 Lite emulator (RX72N Envision Kit onboard debugger)
TSIP driver	Version 1.11
Tera Term	Version 4.105
OpenSSL	1.1.1f

Related Documents

- RX Family TSIP (Trusted Secure IP) Module Firmware Integration Technology (R20AN0371)
- RX Family How to implement FreeRTOS OTA by using Amazon Web Services on RX65N (R01AN5549)

Contents

1. Overview	4
1.1 Advantages of TLS Communication Using TSIP	4
1.2 Cipher Suites Supported by TSIP Driver	4
1.3 TLS APIs of TSIP Driver	5
1.4 Definitions of Terms	6
2. Implementing TLS Communication Using TSIP	7
2.1 Preparation Beforehand	8
2.1.1 Preparation of Root CA Certificate	8
2.1.2 Preparation of Client Certificate	9
2.2 Verifying Root CA Certificate	10
2.3 Handshake Protocol	12
2.3.1 Certificate	12
2.3.2 Server Key Exchange and Client Key Exchange	14
2.3.2.1 ECDHE Key Exchange	14
2.3.2.2 RSA Key Exchange	15
2.3.3 Certificate Verify	16
2.3.4 Finished	17
2.4 Application Data Protocol	19
3. Sample Project	20
3.1 Folder Structure	22
3.2 Key and Certificate Preparation	22
3.2.1 Obtaining Root CA Certificate	23
3.2.2 Generating RSA Keys and Client Certificate	24
3.2.3 Generating ECDSA Client Certificate and Key Pair	25
3.2.3.1 Generating ECC Key Pair	25
3.2.3.2 Registering Keys on AWS	26
3.2.4 Root CA Certificate Signature Generation and Certificate File Format Conversion	32
3.2.4.1 RSA Certificate	32
3.2.4.2 ECDSA Certificate	35
3.2.5 Key Wrapping	36
3.3 Settings for Communication with AWS	38
3.3.1 AWS IoT Settings	38
3.3.2 IP Settings	39
3.3.3 Client Certificate Format Selection	39
4. Building and Running the Project	40
4.1 Importing the Project	40
4.2 Building the Project	41
4.3 Connecting to AWS IoT	41

5. Using Renesas Secure Flash Programmer	44
5.1 Generating a Provisioning Key File	44
5.2 Generating Encrypted Key Files.....	44
6. Appendix.....	47
6.1 TLS Communication Performance Using TSIP Driver.....	47
6.2 Flowchart of TLS Negotiation and Calls to TSIP Driver.....	47
Revision History	50

Notes:

- AWS™ is a trademark of Amazon.com, Inc. or its affiliates.
(<https://aws.amazon.com/trademark-guidelines>)
- FreeRTOS™ is a trademark of Amazon Web Services, Inc. (<https://freertos.org/copyright.html>)
- Git® is a trademark of Software Freedom Conservancy, Inc. (<https://www.git-scm.com/about/trademark>)
- GitHub® is a trademark of GitHub, Inc. (<https://github.com/logos>)
- Arm® is a trademark of Arm Limited or its subsidiaries.
(<https://www.arm.com/company/policies/trademarks/guidelines-trademarks>)
- Mbed™ is a trademark of Arm Limited or its subsidiaries.
(<https://www.arm.com/company/policies/trademarks/guidelines-trademarks>)
- OpenSSL™ is a trademark of OpenSSL Software Foundation.
(<https://www.openssl.org/policies/trademark.html>)

1. Overview

1.1 Advantages of TLS Communication Using TSIP

The TSIP driver supports APIs for TLS (client side only). These APIs provide the following two advantages.

- No keying information is handled as plaintext during TLS protocol processing, thereby reducing the risk that customer keying information stored on the device may leak.
- Hardware acceleration speeds up encryption processing.

1.2 Cipher Suites Supported by TSIP Driver

The TSIP driver supports the following cipher suites conforming to TLS 1.2.

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

1.3 TLS APIs of TSIP Driver

Table 1.1 lists the TSIP driver APIs used for TLS communication. For details of each API, refer to section 2, Implementing TLS Communication Using TSIP, and the application note RX Family TSIP (Trusted Secure IP) Module Firmware Integration Technology (R20AN0371).

Table 1.1 API Functions Used for TLS Communication

Where Used	API Function
Certificate installation	R_TSIP_GenerateTlsRsaPublicKeyIndex R_TSIP_Close R_TSIP_Open R_TSIP_TlsRootCertificateVerification
Certificate	R_TSIP_TlsCertificateVerification
Server Key Exchange Client Key Exchange (ECDHE key exchange algorithm)	R_TSIP_TlsServersEphemeralEcdhPublicKeyRetrieves R_TSIP_GenerateTlsP256EccKeyIndex R_TSIP_TlsGeneratePreMasterSecretWithEccP256Key
Client Key Exchange (RSA key exchange algorithm)	R_TSIP_TlsGeneratePreMasterSecret R_TSIP_TlsEncryptPreMasterSecretWithRsa2048PublicKey
Certificate Verify	R_TSIP_RsassaPkcs1024/2048SignatureGenerate R_TSIP_RsassaPkcs1024/2048SignatureVerification R_TSIP_EcdsaP192/224/256/384SignatureGenerate R_TSIP_EcdsaP192/224/256/384SignatureVerification
Finished	R_TSIP_TlsGenerateMasterSecret R_TSIP_TlsGenerateVerifyData R_TSIP_TlsGenerateSessionKey R_TSIP_Sha1HmacGenerateInit/Update/Final R_TSIP_Sha1HmacVerifyInit/Update/Final R_TSIP_Sha256HmacGenerateInit/Update/Final R_TSIP_Sha256HmacVerifyInit/Update/Final R_TSIP_Aes128CbcEncryptInit/Update/Final R_TSIP_Aes128CbcDecryptInit/Update/Final R_TSIP_Aes256CbcEncryptInit/Update/Final R_TSIP_Aes256CbcDecryptInit/Update/Final R_TSIP_Aes128GcmEncryptInit/Update/Final R_TSIP_Aes128GcmDecryptInit/Update/Final
Application Data	R_TSIP_TlsGenerateSessionKey R_TSIP_Sha1HmacGenerateInit/Update/Final R_TSIP_Sha1HmacVerifyInit/Update/Final R_TSIP_Sha256HmacGenerateInit/Update/Final R_TSIP_Sha256HmacVerifyInit/Update/Final R_TSIP_Aes128CbcEncryptInit/Update/Final R_TSIP_Aes128CbcDecryptInit/Update/Final R_TSIP_Aes256CbcEncryptInit/Update/Final R_TSIP_Aes256CbcDecryptInit/Update/Final R_TSIP_Aes128GcmEncryptInit/Update/Final R_TSIP_Aes128GcmDecryptInit/Update/Final

1.4 Definitions of Terms

Terms used in this document are defined below.

Table 1.2 Terms

Terms	Description
User key	The key employed by the user when inputting data to the encryption function of the device. Generated by the user.
Encrypted key	Keying information generated by using a provisioning key to encrypt the user key with AES-128 and appending a MAC value. Generated by Renesas Secure Flash Programmer.
Key index	User key or other data converted to a format usable by the TSIP driver. Generated by the TSIP.
Provisioning key	A key necessary for generating an encrypted key from a user key. Generated by the user.
Encrypted provisioning key	Keying information used by the TSIP to decrypt an encrypted key and convert it into a key index. Generated by the DLM server.
Hidden Root Key (HRK)	A key that exists only inside the TSIP and in a secure room (the DLM server, etc.) at Renesas.
DLM server (https://dlm.renesas.com/)	The key administration server at Renesas. "DLM server" is short for "device lifecycle management server." Used to perform key wrapping (encrypting) of provisioning keys.

2. Implementing TLS Communication Using TSIP

Figure 2.1 shows an outline flowchart of TLS 1.2 communication and the processing performed using the TSIP driver. The processing enclosed in white boxes in the figure must be implemented using the TSIP driver. In order to make use of the TLS APIs of the TSIP driver, it is first necessary to use the TSIP driver to verify the integrity of the root CA certificate stored on the device. To do this it is necessary to append the signature to be used by the TSIP for verification to the root CA certificate beforehand.

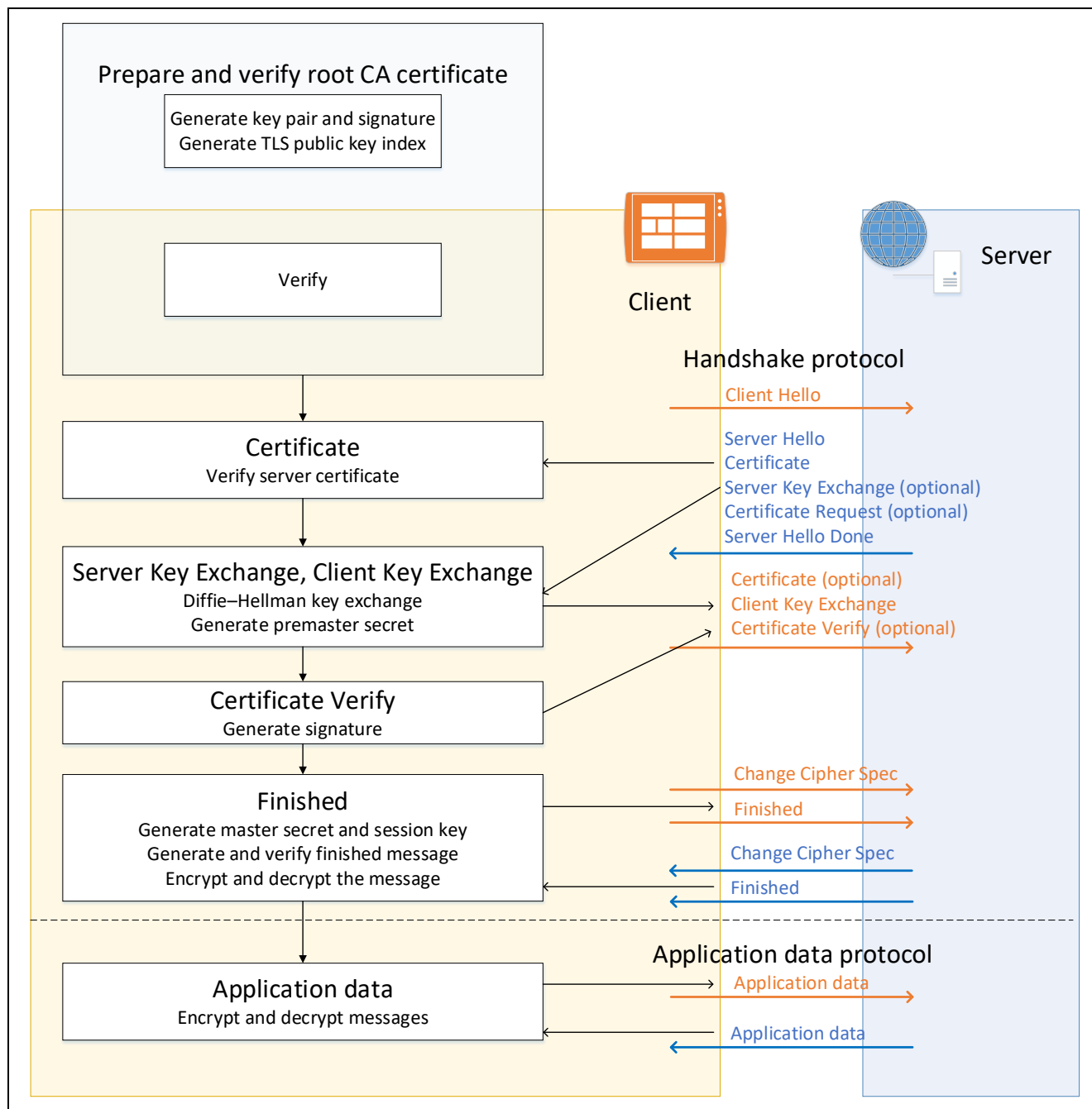


Figure 2.1 Outline of TLS Communication and Processing Performed Using TSIP Driver

For details of the parts of Figure 2.1 involving use of the TSIP driver, refer to Figure 6.1 and Figure 6.2 in section 6.2, Flowchart of TLS Negotiation and Calls to TSIP Driver.

Section 2.1 below describes preparation of the root CA certificate and client certificate as well as verification using the TSIP. Sections 2.3 and 2.4 describe the implementation of TLS protocols using the TSIP driver.

2.1 Preparation Beforehand

2.1.1 Preparation of Root CA Certificate

Before using the TLS APIs of the TSIP driver to extract the public key from the root CA certificate, it is necessary to verify the integrity of the root CA certificate.

Follow the steps below to obtain the root CA certificate and generate the signature to be verified by the TSIP driver. Refer to Figure 2.2 for the preparation sequence.

1. Obtain the root CA certificate.
2. Convert the root CA certificate to DER format.
3. Generate the signature of the root CA certificate and generate an RSA 2048-bit key pair to be used for signature verification.
4. Use the private key from the generated key pair to generate the signature corresponding to the root CA certificate. The signature format is "RSA2048 PSS with SHA256."

The TSIP driver will not except input of user keys in plaintext, so the RSA 2048-bit public key used for signature verification must be converted to a format that will be accepted by the TSIP driver and embedded into a program. The procedure for "wrapping" user keys for use by the TSIP driver is described in section 3.2.5.

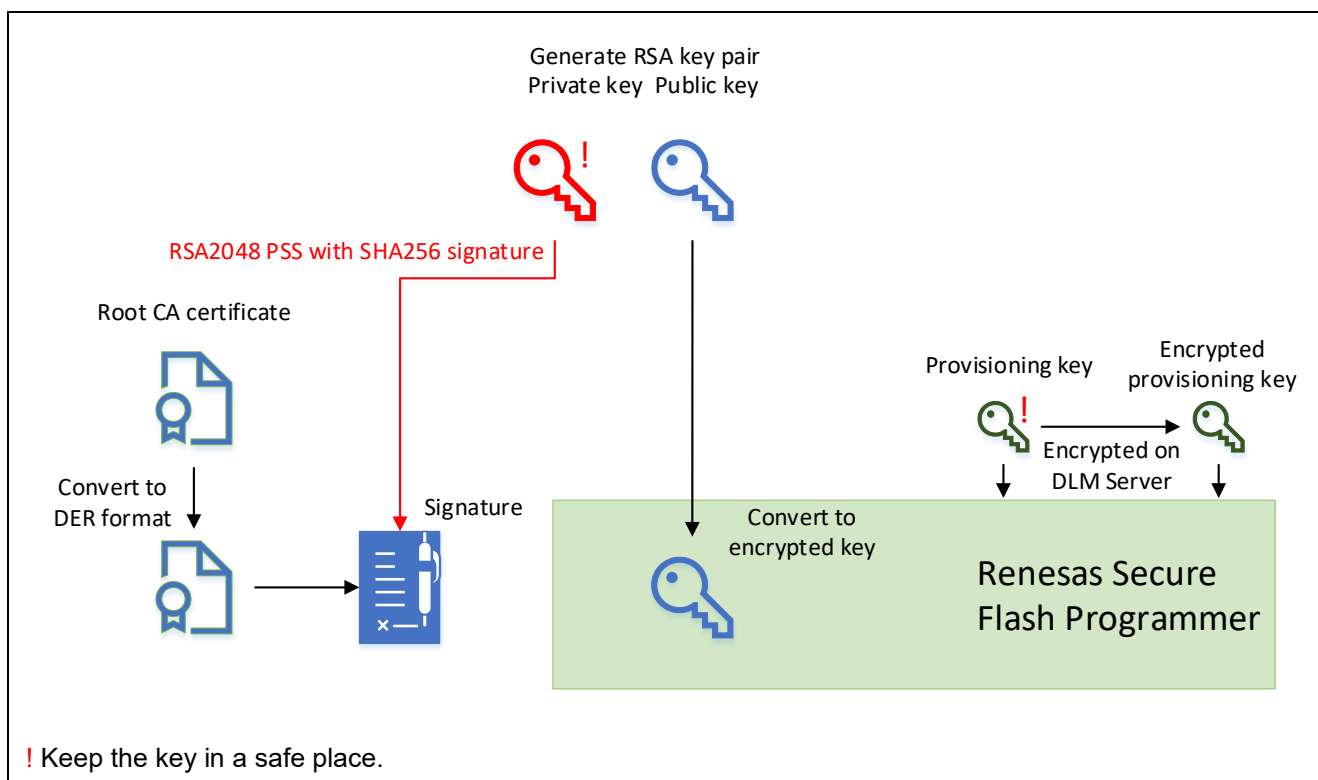


Figure 2.2 Root CA Certificate Preparation Sequence

2.1.2 Preparation of Client Certificate

This procedure involves generating the client key pair and preparing the client certificate.

Follow the steps below to generate the key pair and accept issuance of the client certificate. Refer to Figure 2.3 for the preparation sequence.

1. Generate an RSA and ECC key pair for use by the client.
2. Generate a certificate signing request (CSR) for the generated key pair.
3. Submit the CSR to the certificate authority (CA).
4. Obtain the client certificate issued by the CA based on the CSR.

The TSIP driver will not except input of user keys in plaintext, so the key pair used for signature generation and verification by the client must be converted to a format that will be accepted by the TSIP driver and embedded into a program. The procedure for “wrapping” user keys for use by the TSIP driver is described in section 3.2.5.

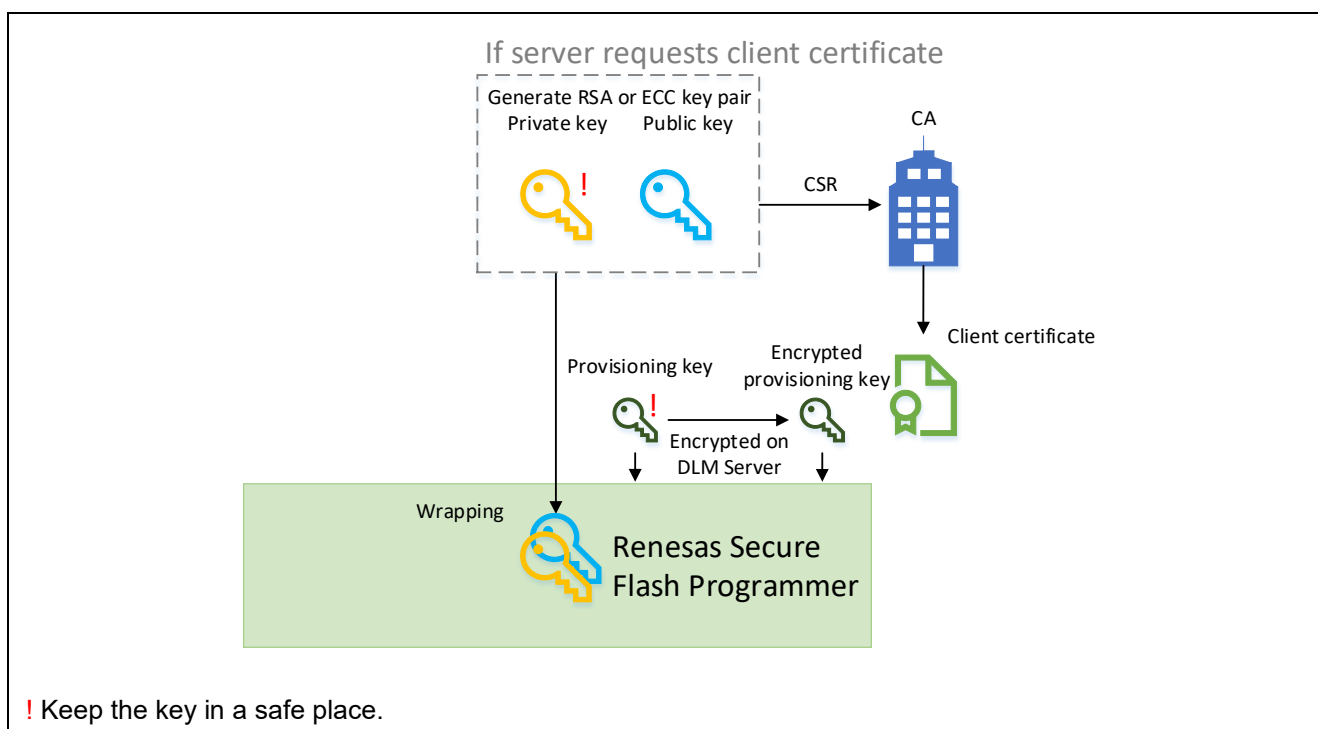


Figure 2.3 Key Pair and Client Certificate Generation Sequence

2.2 Verifying Root CA Certificate

Follow the steps below to verify the root CA certificate, using the DER format certificate, signature, and encrypted key created as described in section 2.1.1. Note that there can be a division of programs between steps 3 and 4. However, the TLS public key index must have been generated on the same device. Refer to Figure 2.4 for the processing sequence and to Table 2.1 for details of the TSIP driver APIs used.

1. Use the `R_TSIP_Open()` function to validate the TSIP.
2. Use the `R_TSIP_GenerateTlsRsaPublicKeyIndex()` function to generate the TLS public key index.
3. Use the `R_TSIP_Close()` function to halt operation of the TSIP.
4. Use the `R_TSIP_Open()` function to validate the TSIP once again and read in the TLS public key index.
5. Use the `R_TSIP_TlsRootCertificateVerification()` function to verify the root CA certificate.

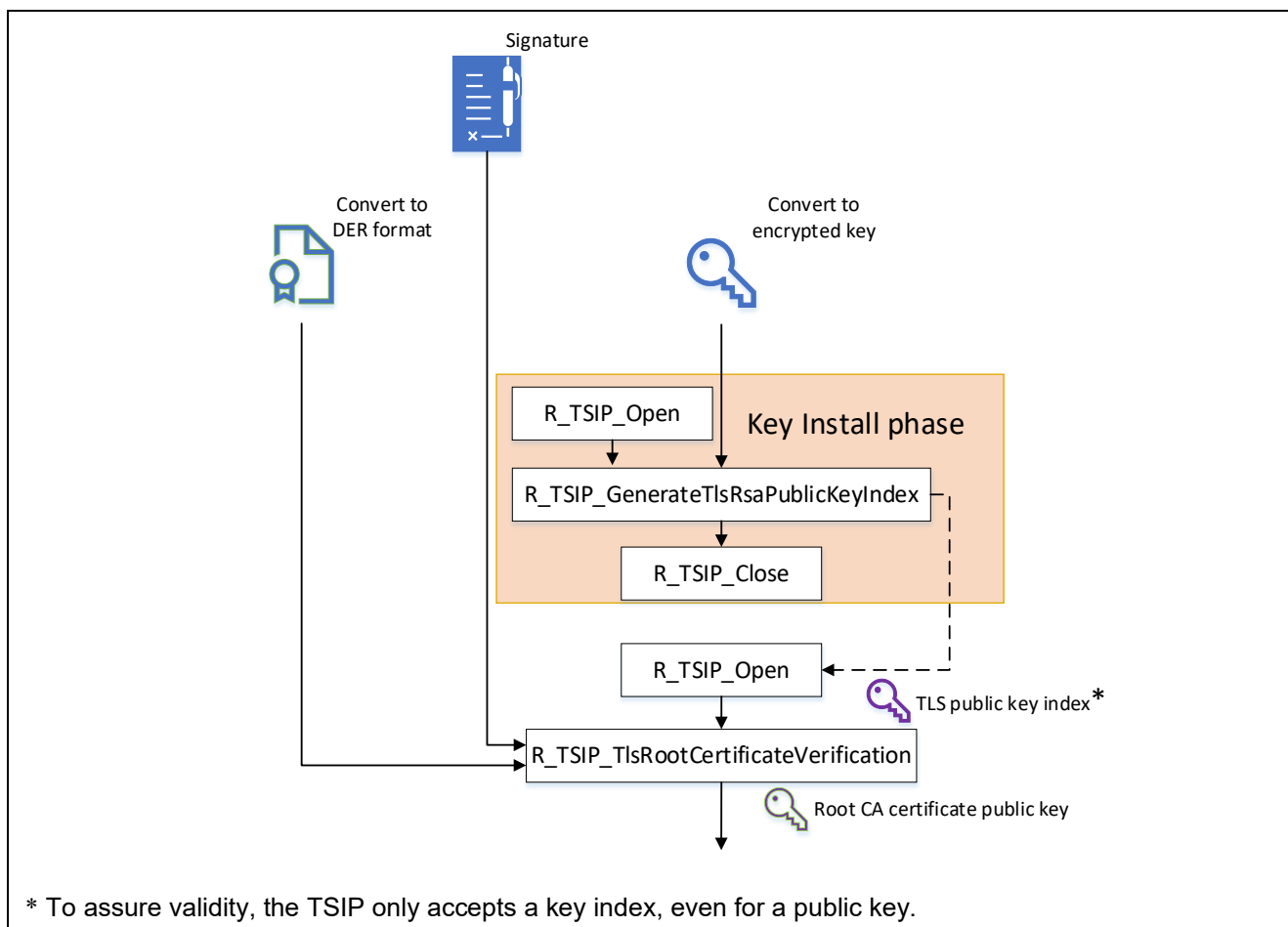


Figure 2.4 Root CA Certificate Verification Sequence

Table 2.1 API Functions Used for Root CA Certificate Preparation and Installation

API Function	Description
<pre>e_tsip_err_t R_TSIP_Open (tsip_tls_ca_certification_public_key_index_t *key_index_1, tsip_update_key_ring_t *key_index_2)</pre>	<p>Validates the TSIP.</p> <p>To validate the TLS APIs of the TSIP driver, it is necessary to input the key_index parameter of R_TSIP_GenerateTlsRsaPublicKeyIndex().</p> <p>Parameters</p> <p>For key_index_1, input a null pointer the first time the function is called, and input the key_index value output by R_TSIP_GenerateTlsRsaPublicKeyIndex() the second time the function is called. If the key update function is not used, input a null pointer for key_index_2.</p>
<pre>e_tsip_err_t R_TSIP_GenerateTlsRsaPublicKeyIndex(uint8_t * encrypted_provisioning_key, uint8_t *iv, uint8_t *encrypted_key, tsip_tls_ca_certification_public_key_index_t *key_index)</pre>	<p>Outputs the RSA public key key_index value used by R_TSIP_TlsRootCertificateVerification().</p> <p>Parameters</p> <p>For encrypted_provisioning_key, iv, and encrypted_key, input the corresponding variables in key_data.c, output by Renesas Secure Flash Programmer.</p>
<pre>e_tsip_err_t R_TSIP_Close (void)</pre>	<p>Halts operation of the TSIP.</p>
<pre>e_tsip_err_t R_TSIP_TlsRootCertificateVerification (uint32_t public_key_type uint8_t *certificate, uint32_t certificate_length, uint32_t public_key_n_start_position, uint32_t public_key_n_end_position, uint32_t public_key_e_start_position, uint32_t public_key_e_end_position, uint8_t *signature, uint32_t *encrypted_root_public_key);)</pre>	<p>Verifies the root CA certificate prepared beforehand.</p> <p>Parameters</p> <p>For public_key_type, input the type of the public key contained in the certificate. For certificate, input the certificate in DER format, and for certificate_length, input the length of the certificate. For public_key_*_position, input the addresses corresponding to the start and end points of the public keying information of the root CA certificate obtained by decrypting the certificate. For signature, input the signature data corresponding to the certificate. For encrypted_root_public_key, keying information is output that is used to verify the server certificate in the next procedure.</p>

2.3 Handshake Protocol

The handshake protocol processing requiring implementation of the TSIP driver is described below.

2.3.1 Certificate

Follow the steps below to verify the certificate chain. Refer to Figure 2.5 for the processing sequence and to Table 2.2 for details of the TSIP driver API used.

1. Prepare the public key (the **encrypted_root_public_key** parameter of the `R_TSIP_TlsRootCertificateVerification()` function) extracted from the root CA certificate.
2. Use the `R_TSIP_TlsCertificateVerification()` function to verify the next certificate after the last certificate is verified.
3. If there is an unverified certificate remaining, the certificate verified in step 2 is an intermediate certificate. Prepare the public key (the **encrypted_output_public_key** parameter) contained in the verified certificate and return to step 2. If there are no unverified certificates remaining, the certificate verified in step 2 is the server certificate. Prepare the public key contained in the verified certificate and proceed to the next processing procedure.

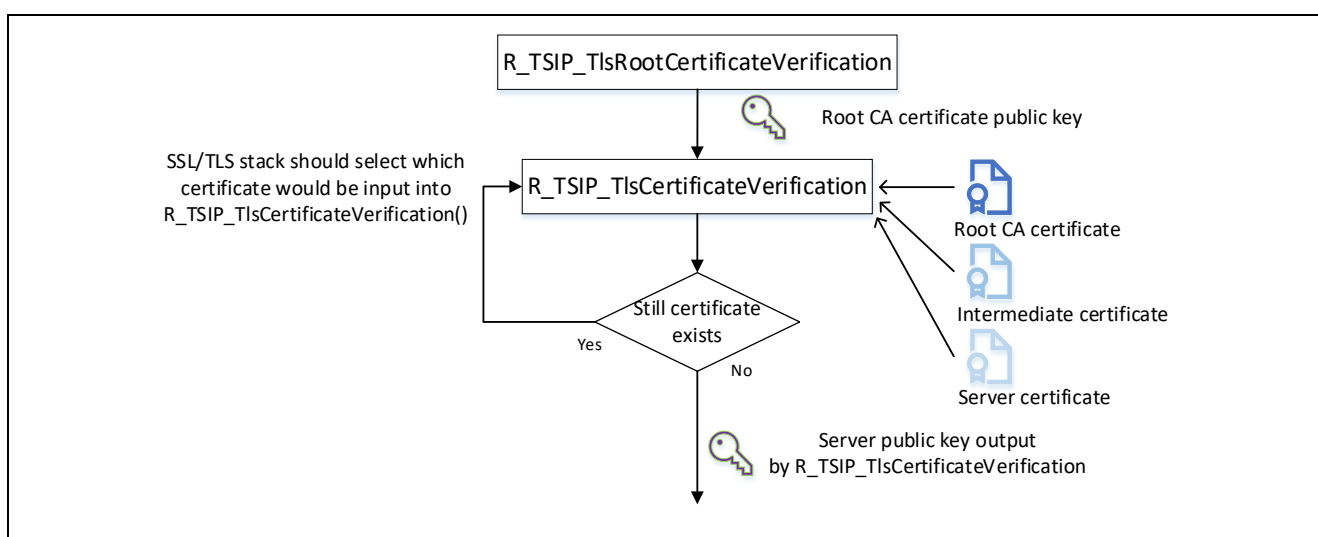


Figure 2.5 Server Certificate Verification Sequence

Table 2.2 API Function Used for Server Certificate Verification

API Function	Description
<pre> e_tsip_err_t R_TSIP_TlsCertificateVerification (uint32_t public_key_type, uint32_t *encrypted_input_public_key uint8_t *certificate, uint32_t certificate_length, uint8_t *signature, uint32_t public_key_n_start_position, uint32_t public_key_n_end_position, uint32_t public_key_e_start_position, uint32_t public_key_e_end_position, uint32_t *encrypted_output_public_key) </pre>	<p>Verifies the certificate chain. Call this function repeatedly to perform the necessary processing, starting with verification of the intermediate certificate after the root CA certificate and ending with verification of the server certificate.</p> <p>Parameters</p> <p>For public_key_type, input the type of the public key contained in the certificate. For certificate, input the certificate in DER format, and for certificate_length, input the length of the certificate. For signature, input the signature data from the certificate to be verified. For public_key_*_position, input the addresses corresponding to the public keying information to be verified. For encrypted_input_public_key, input the public key of the certificate verified immediately previously. The parameter encrypted_output_public_key, output when the server certificate is verified, is used when <code>R_TSIP_TlsEncryptPreMasterSecretWithRsa2048PublicKey()</code> or <code>R_TSIP_TlsServersEphemeralEcdhPublicKeyRetrieves()</code> is called to perform key exchange.</p>

2.3.2 Server Key Exchange and Client Key Exchange

2.3.2.1 ECDHE Key Exchange

Follow the steps below to perform key exchange. Refer to Figure 2.6 for the processing sequence and to Table 2.3 for details of the TSIP driver APIs used.

1. Use the `R_TSIP_TlsServersEphemeralEcdhPublicKeyRetrieves()` function to verify the server ephemeral ECDH public key received in the Server Key Exchange message.
2. Use the `R_TSIP_GenerateTlsP256EccKeyIndex()` function to generate the client ephemeral ECDH key pair. Send the client ephemeral ECDH public key to the server in the Client Key Exchange message.
3. Use the `R_TSIP_TlsGeneratePreMasterSecretWithEccP256Key()` function to generate the premaster secret from the server ephemeral ECDH public key and client ephemeral ECDH private key.

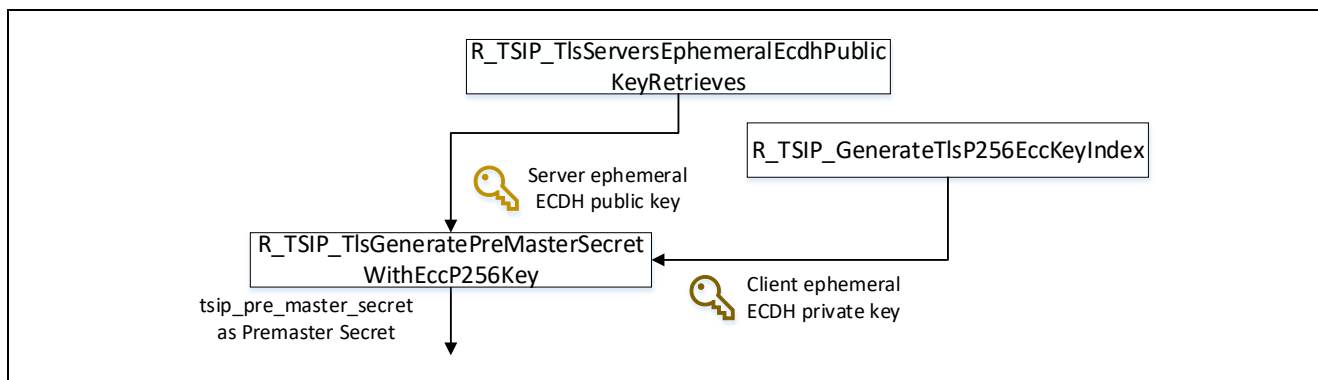


Figure 2.6 ECDHE Key Exchange Sequence

Table 2.3 API Functions Used for ECDHE Key Exchange

API Function	Description
<code>e_tsip_err_t</code> <code>R_TSIP_TlsServersEphemeralEcdhPublicKeyRetrieves (</code> <code>uint32_t public_key_type,</code> <code>uint8_t *client_random,</code> <code>uint8_t *server_random,</code> <code>uint8_t *server_ephemeral_ecdh_public_key,</code> <code>uint8_t *server_key_exchange_signature,</code> <code>uint32_t *encrypted_public_key,</code> <code>uint32_t *encrypted_ephemeral_ecdh_public_key</code> <code>)</code>	Verifies the Server Key Exchange signature based on the server public key. The output is an encrypted ephemeral ECDH public key used by <code>R_TSIP_TlsGeneratePreMasterSecretWithEccP256Key()</code> .
<code>e_tsip_err_t</code> <code>R_TSIP_GenerateTlsP256EccKeyIndex (</code> <code>tsip_tls_p256_ecc_key_index_t</code> <code>*tls_p256_ecc_key_index,</code> <code>uint8_t *ephemeral_ecdh_public_key</code> <code>)</code>	Generates a key pair using the ECDH algorithm. The output is the keying information used by <code>R_TSIP_TlsGeneratePreMasterSecretWithEccP256Key()</code> and the ephemeral ECDH public key sent to the server in the Client Key Exchange message.
<code>e_tsip_err_t</code> <code>R_TSIP_TlsGeneratePreMasterSecretWithEccP256Key (</code> <code>uint32_t *encrypted_public_key,</code> <code>tsip_tls_p256_ecc_key_index_t</code> <code>*tls_p256_ecc_key_index,</code> <code>uint32_t *tsip_pre_master_secret</code> <code>)</code>	Outputs the premaster secret based on the values input from <code>R_TSIP_TlsServersEphemeralEcdhPublicKeyRetrieves()</code> and <code>R_TSIP_GenerateTlsP256EccKeyIndex()</code> .

2.3.2.2 RSA Key Exchange

Follow the steps below to perform key exchange. Refer to Figure 2.7 for the processing sequence and to Table 2.4 for details of the TSIP driver APIs used.

1. Use the `R_TSIP_TlsGeneratePreMasterSecret()` function to generate the premaster secret.
2. Use the `R_TSIP_TlsEncryptPreMasterSecretWithRsa2048PublicKey()` function to encrypt the premaster secret. Send the encrypted premaster secret to the server in the Client Key Exchange message.

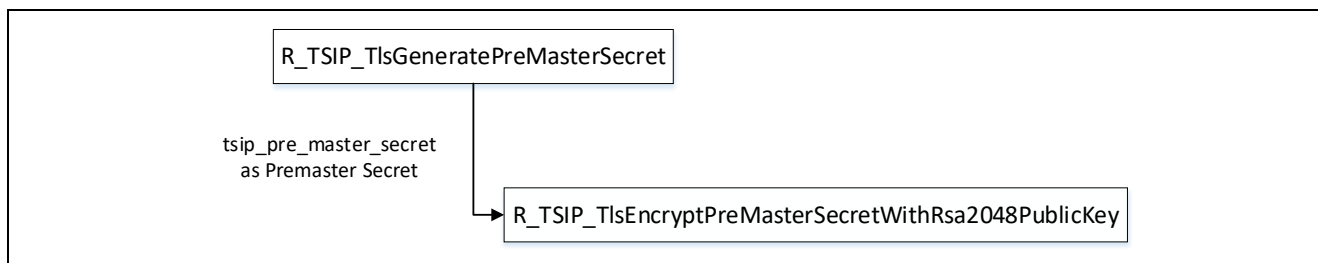


Figure 2.7 RSA Key Exchange Sequence

Table 2.4 API Functions Used for RSA Key Exchange

API Function	Description
<code>e_tsip_err_t R_TSIP_TlsGeneratePreMasterSecret (uint32_t *tsip_pre_master_secret)</code>	Generates the premaster secret.
<code>e_tsip_err_t R_TSIP_TlsEncryptPreMasterSecretwithRsaPublicKey (uint32_t *encrypted_public_key, uint32_t *tsip_pre_master_secret, uint8_t *encrypted_pre_master_secret)</code>	Outputs the premaster secret to be sent to the server in the Client Key Exchange message as data encrypted with the RSA-2048 public key.

2.3.3 Certificate Verify

This procedure involves generating a signature used to verify the client certificate on the server side.

The API functions used differ according to the type of public key contained in the client certificate. Refer to Figure 2.8 for the processing sequence and to Table 2.5, API Functions Used for Client Certificate Verification, for details of the TSIP driver APIs used.

If the public key type is RSA, the following sequence is used to generate the signature.

1. Use the `R_TSIP_RsassaPkcs1024/2048SignatureGenerate()` function to generate a signature for the message.
2. If necessary, generate a public key index from the public key with `R_TSIP_GenerateRsa1024/2048PublicKeyIndex()` function, and use the `R_TSIP_RsassaPkcs1024/2048SignatureVerification()` function to self-verify the generated signature.

If the public key type is ECC, the following sequence is used to generate the signature.

1. Use the `R_TSIP_EcdsaP192/224/256/384SignatureGenerate()` function to generate a signature for the message.
2. If necessary, generate a public key index from the public key with `R_TSIP_GenerateEccP192/224/256/384PublicKeyIndex()` function, and use the `R_TSIP_EcdsaP192/224/256/384SignatureVerification()` function to self-verify the generated signature.

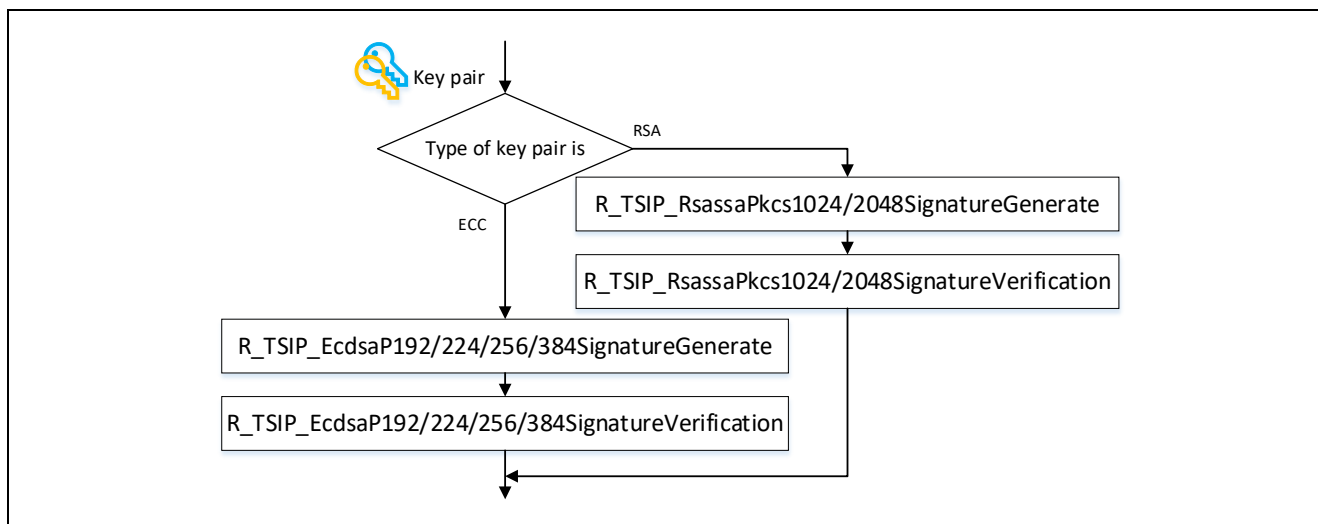


Figure 2.8 Signature Generation Sequence for Client Certificate Verification

Table 2.5 API Functions Used for Client Certificate Verification

API Function	Description
<code>R_TSIP_RsassaPkcs1024/2048SignatureGenerate</code>	Uses an RSA private key to generate an RSASSA-PKCS1-v1_5 signature.
<code>R_TSIP_RsassaPkcs1024/2048SignatureVerification</code>	Uses an RSA public key to verify an RSASSA-PKCS1-v1_5 signature.
<code>R_TSIP_EcdsaP192/224/256/384SignatureGenerate</code>	Uses an ECC private key to generate an ECDSA signature.
<code>R_TSIP_EcdsaP192/224/256/384SignatureVerification</code>	Uses an ECC public key to verify an ECDSA signature.

2.3.4 Finished

Follow the steps below to create and verify the Finished message. Refer to Figure 2.9 for the processing sequence and to Table 2.6, Table 2.7, Table 2.8, Table 2.9, and Table 2.10 for details of the TSIP driver APIs used.

1. Use the `R_TSIP_TlsGenerateMasterSecret()` function to generate the master secret from the premaster secret.
2. Use the `R_TSIP_TlsGenerateSessionKey()` function to generate four session keys (client write MAC key, server write MAC key, client write encryption key, and server write encryption key) and two IVs (client write IV and server write IV) from the master secret.
3. Use the `R_TSIP_TlsGenerateVerifyData()` function to generate verify data from the content of the Finished message sent from the client.
4. Use the hash function and AES function supported by the cipher suite to generate and encrypt the signature of the Finished message.
5. Send the Finished message from the client to the server.
6. Receive the Finished message from the server.
7. Use the hash function and AES function supported by the cipher suite to decrypt and verify the signature of the Finished message.
8. Use the `R_TSIP_TlsGenerateVerifyData()` function to verify the verify data. This concludes the handshake protocol.

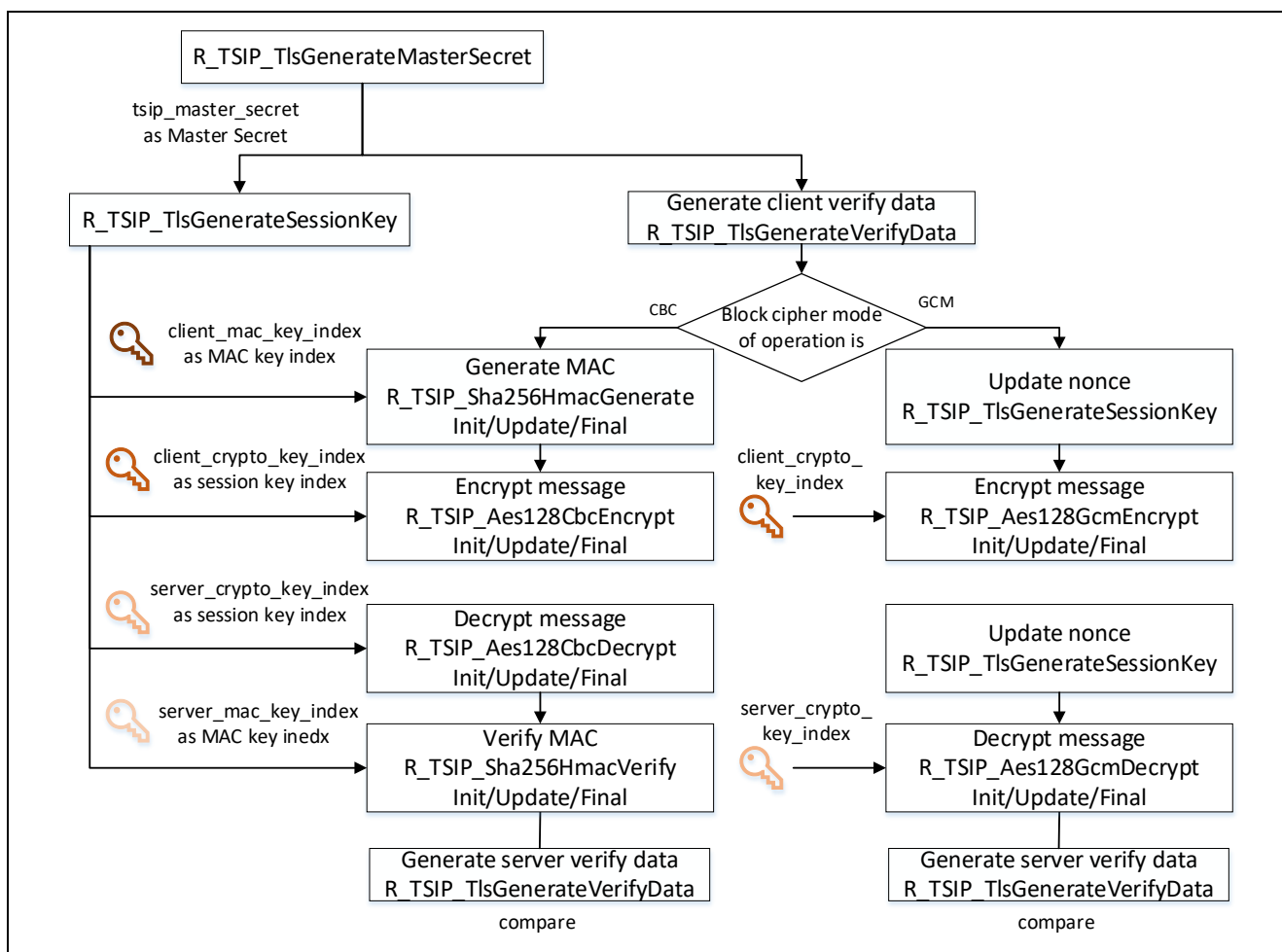


Figure 2.9 Finished Message Generation and Verification Sequence

Table 2.6 API Functions Used for Master Secret Generation and Finished Message Generation and Verification

API Function	Description
e_tsip_err_t R_TSIP_TlsGenerateMasterSecret (uint32_t select_cipher_suite, uint32_t *tsip_pre_master_secret, uint8_t *client_random, uint8_t *server_random,uint32_t *tsip_master_secret)	Generates the master secret based on the premaster secret.
e_tsip_err_t R_TSIP_TlsGenerateSessionKey (uint32_t select_cipher_suite, uint32_t *tsip_master_secret, uint8_t *client_random, uint8_t *server_random, uint8_t *nonce_explicit, tsip_hmac_sha_key_index_t *client_mac_key_index, tsip_hmac_sha_key_index_t *server_mac_key_index, tsip_aes_key_index_t *client_crypto_key_index, tsip_aes_key_index_t *server_crypto_key_index, uint8_t *client_iv, uint8_t *server_iv)	Outputs the session key key_index (client_mac_key_index , server_mac_key_index , client_crypto_key_index , and server_crypto_key_index) based on the master secret. The IVs are contained in client_crypto_key_index and server_crypto_key_index . When using CBC mode, input NULL for nonce_explicit . When using GCM mode, input a nonce value.
e_tsip_err_t R_TSIP_TlsGenerateVerifyData (uint32_t select_verify_data, uint32_t *tsip_master_secret, uint8_t *hand_shake_hash, uint8_t *verify_data)	Generates verify data for the Finished message.

Table 2.7 API Functions Used for Encryption in CBC Mode

API Function	Description
R_TSIP_Sha1HmacGenerateInit/Update/Final R_TSIP_Sha256HmacGenerateInit/Update/Final	Uses client_mac_key_index to generate the MAC value of the data to be sent to the server.
R_TSIP_Aes128CbcEncryptInit/Update/Final R_TSIP_Aes256CbcEncryptInit/Update/Final	Uses client_crypto_key_index to encrypt the data to be sent to the server.

Table 2.8 API Functions Used for Decryption in CBC Mode

API Function	Description
R_TSIP_Aes128CbcDecryptInit/Update/Final R_TSIP_Aes256CbcDecryptInit/Update/Final	Uses server_crypto_key_index to decrypt cipher text received from the server.
R_TSIP_Sha1HmacVerifyInit/Update/Final R_TSIP_Sha256HmacVerifyInit/Update/Final	Uses server_mac_key_index to verify the MAC value of decrypted data received from the server.

Table 2.9 API Functions Used for Encryption in GCM Mode

API Function	Description
R_TSIP_TlsGenerateSessionKey	Updates the nonce in the TSIP. For nonce_explicit , input a different nonce for each packet.
R_TSIP_Aes128GcmEncryptInit/Update/Final	Uses client_crypto_key_index to encrypt and generate a certification tag for data to be sent to the server. Input NULL for lvec and 0 for lvec_len .

Table 2.10 API Functions Used for Decryption in GCM Mode

API Function	Description
R_TSIP_TlsGenerateSessionKey	Updates the nonce in the TSIP. For nonce_explicit , input the nonce contained in the packet.
R_TSIP_Aes128GcmDecryptInit/Update/Final	Uses server_crypto_key_index to decrypt and verify the certification tag of data received from the server.

2.4 Application Data Protocol

Like the Finished message of the handshake protocol, the application data protocol uses TSIP driver APIs to carry out encryption and decryption processing and to perform encrypted communication. Refer to Table 2.7 and Table 2.8 for the APIs used in CBC mode and to Table 2.9 and Table 2.10 for the APIs used in GCM mode.

3. Sample Project

The sample project is a demo program that uses the RX72N Envision Kit to establish a TLS connection to AWS and performs MQTT communication.

Information on the connections used when running the sample project on the RX72N Envision Kit is shown below. Connect the RX72N Envision Kit to a PC using two USB cables for debugging and serial communication. To connect to the internet, connect the RX72N Envision Kit to a router using an Ethernet cable.

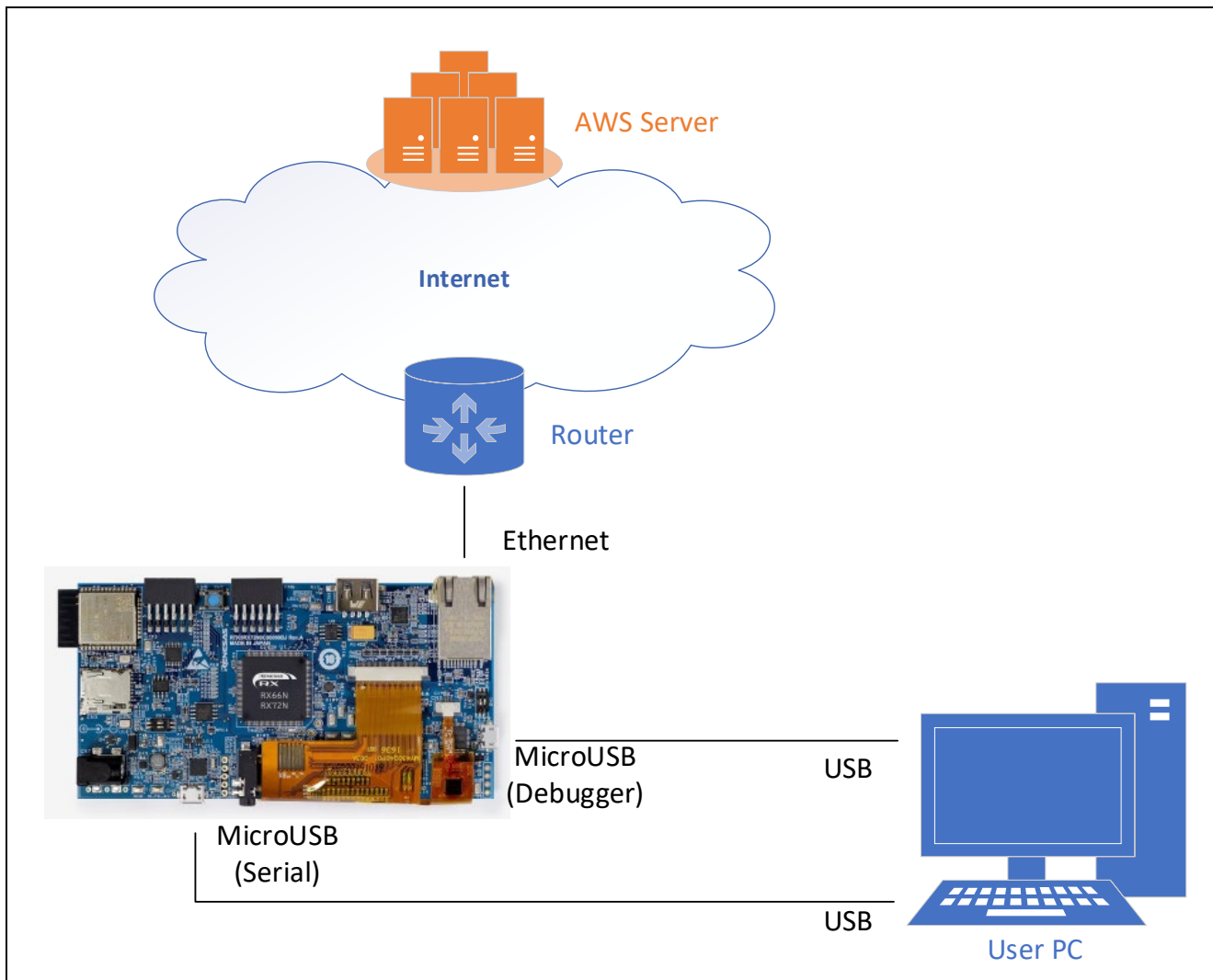


Figure 3.1 Connections for Sample Project

The sample project is based on an FreeRTOS project. FreeRTOS provides IoT libraries containing source code necessary for applications involving IoT devices. The sample project makes use of one of these, the open source Mbed TLS, as its encryption library. In the project, some of the processing of the Mbed TLS library is reassigned to the TLS APIs of the TSIP driver. The software configuration of the sample project is shown below.

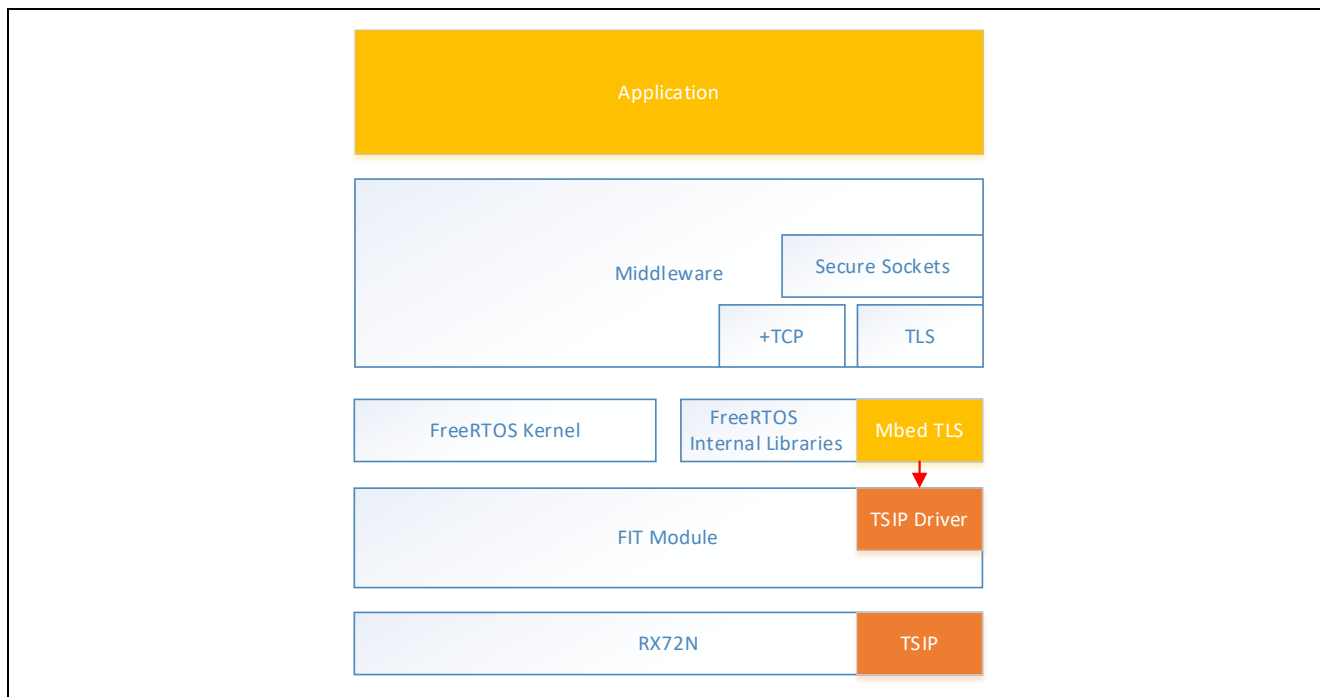


Figure 3.2 Software Configuration of Sample Project

The sample project is based on the FreeRTOS project for RX located in the following repository.

<https://github.com/renesas/amazon-freertos/releases/tag/v202002.00-rx-1.0.5>

The following tools are used in the operations described in this section, so you will need to obtain them before starting.

- Shell script (bash) execution environment
- OpenSSL
- Renesas Secure Flash Programmer

3.1 Folder Structure

The folder structure of the sample project is shown below. **Bold text** indicates the locations of files that have been modified from the base project. Only the top folder is shown in cases where a folder contains many subfolders. To check for differences in detail, use a utility such as the diff tool.

```
amazon-freertos
|--demos
|   |--dev_mode_key_provisioning/src/aws_dev_mode_key_provisioning.c
|--doc
|--freertos_kernel
|--libraries
|   |--3rdparty/mbedtls
|   |--abstractions/pkcs11/mbedtls/iot_pkcs11_mbedtls.c
|   |--freertos_plus/standard/tls
|--projects
|--tests
|--tools
|--vendors/renesas
|   |--boards/rx_mcu_boards/(board name)/aws_demos/src/smc_gen
|   |--general
|   |--r_config/r_tsip_rx_config.h
|--rx_driver_package/v125/r_tsip_rx
```

In the sample project the following modifications have been made to the base project to enable use of the TSIP driver.

- The TSIP driver FIT module has been added to the project.
- Some of the processing of FreeRTOS and Mbed TLS has been reassigned to the TSIP driver.
- Exclusive control has been added to prevent conflicts when accessing the TSIP in a multitasking environment.
- New setting files have been added containing descriptors of certificates and their signatures.

3.2 Key and Certificate Preparation

The methods for obtaining keys and certificates for use with the sample project and utilizing them with the TSIP driver are described below. Table 3.1 summarizes the methods for obtaining keys and certificates for use with the sample project. The procedure described in section 3.2.2 can be omitted if no RSA client certificate will be used. The procedure described in section 3.2.3 can be omitted if no ECDSA client certificate will be used. In those cases, skip ahead to the procedure described in sections 3.2.4 and 3.2.5.

Table 3.1 Methods for Obtaining Keys and Certificates for Use with Sample Project

Key/Certificate	How to Obtain	Section
RSA root CA certificate	Download from AWS.	3.2.1
ECDSA root CA certificate	Download from AWS.	3.2.1
RSA key pair	Download automatically generated certificate from AWS.	3.2.2
RSA client certificate	Download from AWS.	3.2.2
ECC key pair	Create using OpenSSL tools or equivalent.	3.2.3.1
ECDSA client certificate	Create certificate signing request (CSR) using OpenSSL tools or equivalent, and upload it to AWS. Then download the certificate from AWS.	3.2.3.2
Key pair for root CA certificate signature generation and signature verification	Created by user using OpenSSL tools or equivalent.	3.2.4.1 3.2.4.2

3.2.1 Obtaining Root CA Certificate

Obtain a root CA certificate via the following link.

<https://docs.aws.amazon.com/iot/latest/developerguide/server-authentication.html#server-authentication-certs>

To use an RSA certificate, use the **Amazon Root CA 1** download link. To use an ECDSA certificate, use the **Amazon Root CA 3** download link.

CA certificates for server authentication

Depending on which type of data endpoint you are using and which cipher suite you have negotiated, AWS IoT Core server authentication certificates are signed by one of the following root CA certificates:

VeriSign Endpoints (legacy)

- RSA 2048 bit key: [VeriSign Class 3 Public Primary G5 root CA certificate](#)

Amazon Trust Services Endpoints (preferred)

Note

You might need to right click these links and select **Save link as...** to save these certificates as files.

- RSA 2048 bit key: [Amazon Root CA 1](#)
- RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
- ECC 256 bit key: [Amazon Root CA 3](#)
- ECC 384 bit key: Amazon Root CA 4. Reserved for future use.

These certificates are all cross-signed by the [Starfield Root CA Certificate](#). All new AWS IoT Core regions, beginning with the May 9, 2018 launch of AWS IoT Core in the Asia Pacific (Mumbai) Region, serve only ATS certificates.

Copy the downloaded certificate file to the following location in the **key_cert_sig_generator** folder of the sample project.

```
key_cert_sig_generator
|-- ca
|   |-- AmazonRootCA1.pem
|   |-- AmazonRootCA3.pem
|-- ca-sign-keypair-rsa2048
|-- client-ecc256
|-- client-rsa2048
|-- output
|--1_rsa2048_convertCert.sh
|--2_1_ecc256_generateKeyPair.sh
|--2_2_ecc256_convertCert.sh
|--3_showkeyValues.sh
|--convertCert.sh
```

3.2.2 Generating RSA Keys and Client Certificate

You can have an RSA key pair and client certificate generated automatically on the AWS server. Follow the steps in 1.1, Sign in the console, of the application note RX Family: How to implement FreeRTOS OTA by using Amazon Web Services on RX65N (R01AN5549) to register a thing and obtain an RSA-2048 client certificate, public key, and private key.

Copy the downloaded certificate and key files to the following location in the **key_cert_sig_generator** folder of the sample project.

```
key_cert_sig_generator
|-- ca
|   |-- AmazonRootCA1.pem
|   |-- AmazonRootCA3.pem
|-- ca-sign-keypair-rsa2048
|-- client-ecc256
|-- client-rsa2048
|   |-- *-certificate.pem.crt
|   |-- *-private.pem.key
|   |-- *-public.pem.key
|-- output
|--1_rsa2048_convertCrt.sh
|--2_1_ecc256_generateKeyPair.sh
|--2_2_ecc256_convertCrt.sh
|--3_showkeyValues.sh
|--convertCrt.sh
```


3.2.3 Generating ECDSA Client Certificate and Key Pair

To generate an ECDSA client certificate using AWS, you will need to generate an ECC key pair and certificate signing request (CSR) and upload them to AWS.

3.2.3.1 Generating ECC Key Pair

Run **2_1_ecc256_generateKeyPair.sh**, located in the **key_crt_sig_generator** folder.

The contents of **2_1_ecc256_generateKeyPair.sh** are listed below.

```
#!/bin/sh

# Create a key pair and CSR
openssl ecparam -genkey -name prime256v1 -out client-ecc256/prime256v1-private.pem.key
openssl req -new -sha256 -key client-ecc256/prime256v1-private.pem.key -out client-ecc256/prime256v1-csr.pem.csr
echo -e "\nPlease upload \"prime256v1-csr.pem.csr\" to AWS IoT Core and download \"*-certificate.pem.crt\"."
```

After running the script, the ECC key pair and associated CSR are output to the **client-ecc256** folder.

```
key_crt_sig_generator
|-- ca
|   |-- AmazonRootCA1.pem
|   |-- AmazonRootCA3.pem
|-- ca-sign-keypair-rsa2048
|-- client-ecc256
|   |-- prime256v1-csr.pem.csr
|   |-- prime256v1-private.pem.key
|-- client-rsa2048
|-- output
|--1_rsa2048_convertCrt.sh
|--2_1_ecc256_generateKeyPair.sh
|--2_2_ecc256_convertCrt.sh
|--3_showkeyValues.sh
|--convertCrt.sh
```

3.2.3.2 Registering Keys on AWS

Upload the output CSR to AWS IoT.

If you skipped the procedure described in 3.2.2, Generating RSA Keys and Client Certificate, you will need to start by creating a thing on AWS. Follow the steps in 1.1, Sign in the console, of the application note RX Family: How to implement FreeRTOS OTA by using Amazon Web Services on RX65N (R01AN5549) to register a thing. On the **Add a certificate for your thing** page shown below, click **Create thing without certificate**.

AWS IoT > Things > Create things > Add your device to the thing registry > Add certificate

CREATE A THING

Add a certificate for your thing STEP 2/3

A certificate is used to authenticate your device's connection to AWS IoT.

One-click certificate creation (recommended)
This will generate a certificate, public key, and private key using AWS IoT's certificate authority. Create certificate

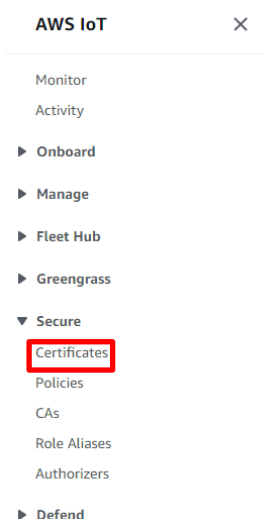
Create with CSR
Upload your own certificate signing request (CSR) based on a private key you own. Create with CSR

Use my certificate
Register your CA certificate and use your own certificates for one or many devices. Get started

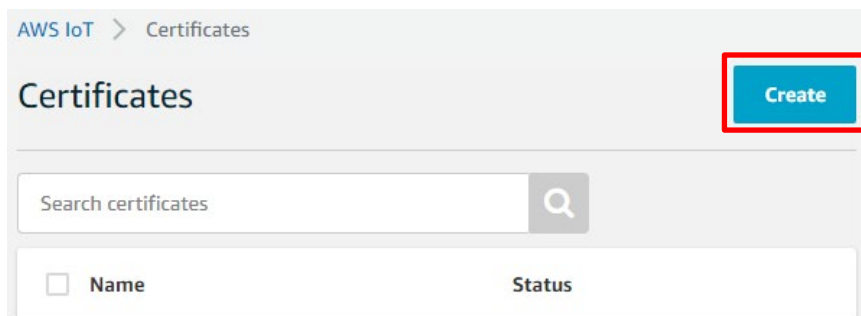
Skip certificate and create thing
You will need to add a certificate to your thing later before your device can connect to AWS IoT. Create thing without certificate

Sign in to the AWS Management Console (<https://aws.amazon.com/console/>) and select **All Services** → **Internet of Things** → **IoT Core**.

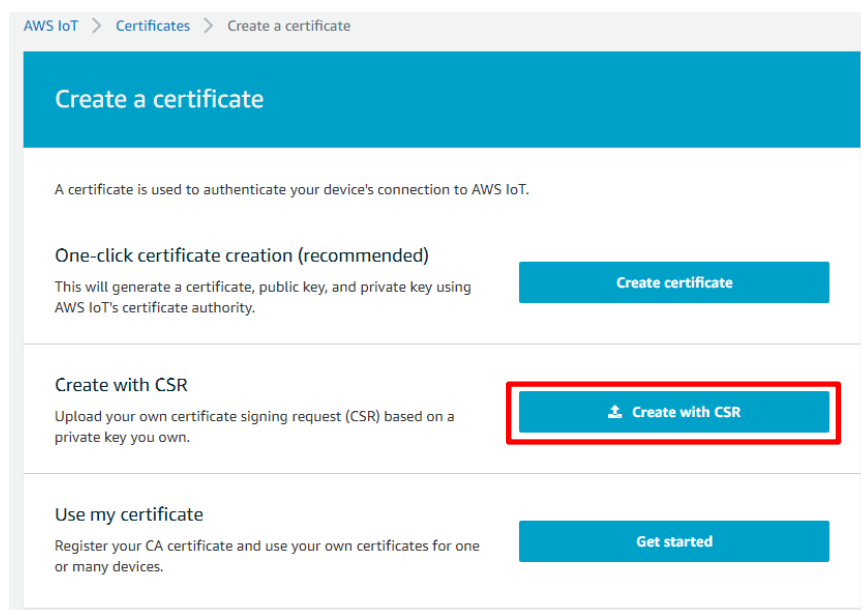
From the menu on the left, select **Secure** → **Certificates**.



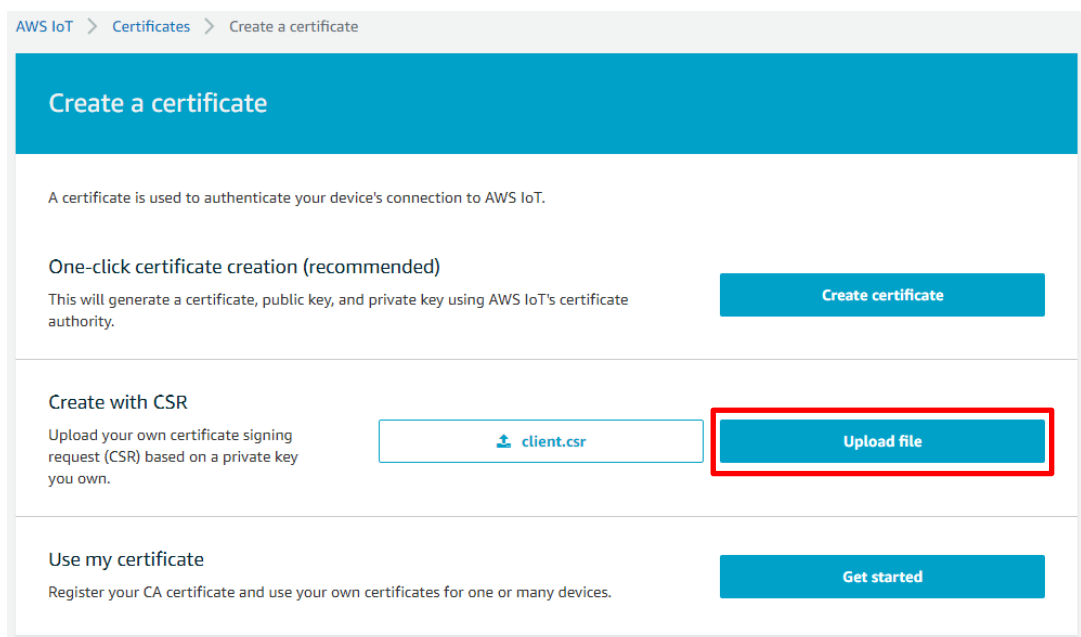
Click the **Create** button at the upper right.



Click **Create with CSR**.



When the **Open File** dialog box appears, select the **prime256v1-csr.pem.csr** file created as described in section 3.2.3.1. Next, click the **Upload file** button.



AWS IoT issues a certificate for the ECC key pair. Click **Download**, then **Activate**, then **Attach a policy**.

Certificate created!

Download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys cannot be retrieved after you close this page.

In order to connect a device, you need to download the following:

A certificate for this thing [redacted] cert.pem **Download**

You also need to download a root CA for AWS IoT:
A root CA for AWS IoT [Download](#)

Activate

[Cancel](#) [Done](#) **Attach a policy**

Select the policy that was created when you registered the thing, then click **Done**.

AWS IoT > Certificates > Create a certificate > Add authorization to certificate

Add authorization to certificate

You are attaching a policy to the following certificate:
[redacted]

Select a policy to attach to this certificate:

Search policies

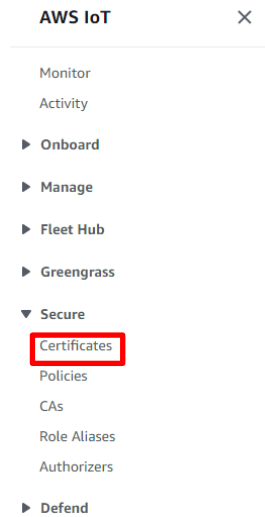
<input checked="" type="checkbox"/>	rx72n_ek	View
<input type="checkbox"/>	[redacted]	View
<input type="checkbox"/>	[redacted]	View
<input type="checkbox"/>	[redacted]	View
<input type="checkbox"/>	[redacted]	View

[Create new policy](#)

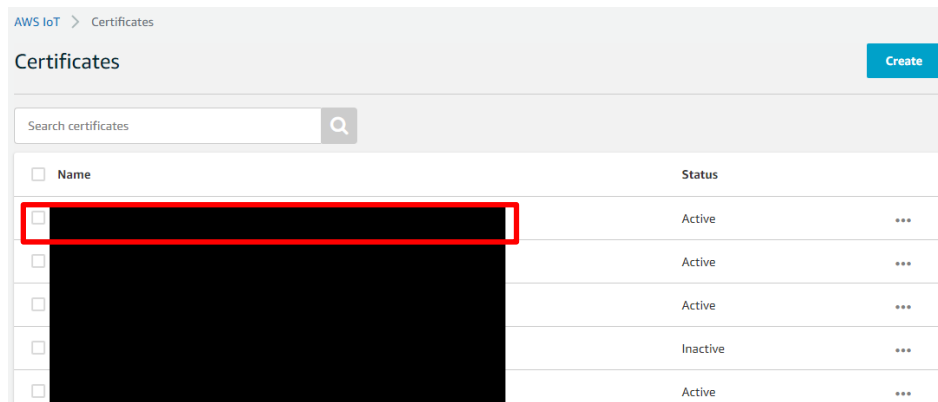
0 policies selected

Done

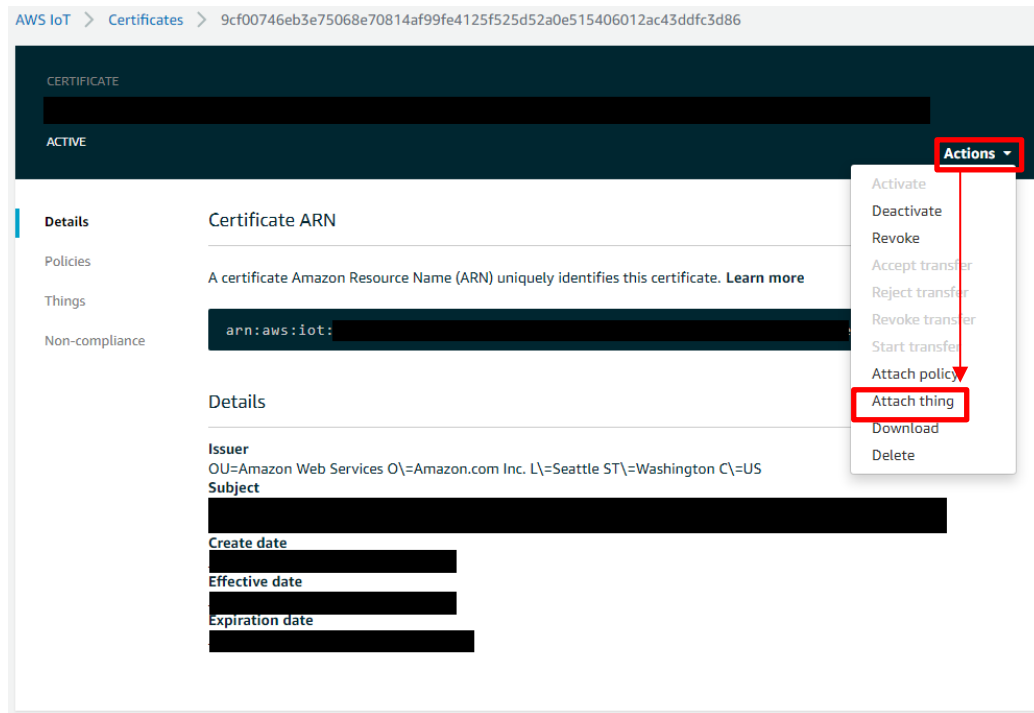
From the menu on the left, select **Secure** → **Certificates**.



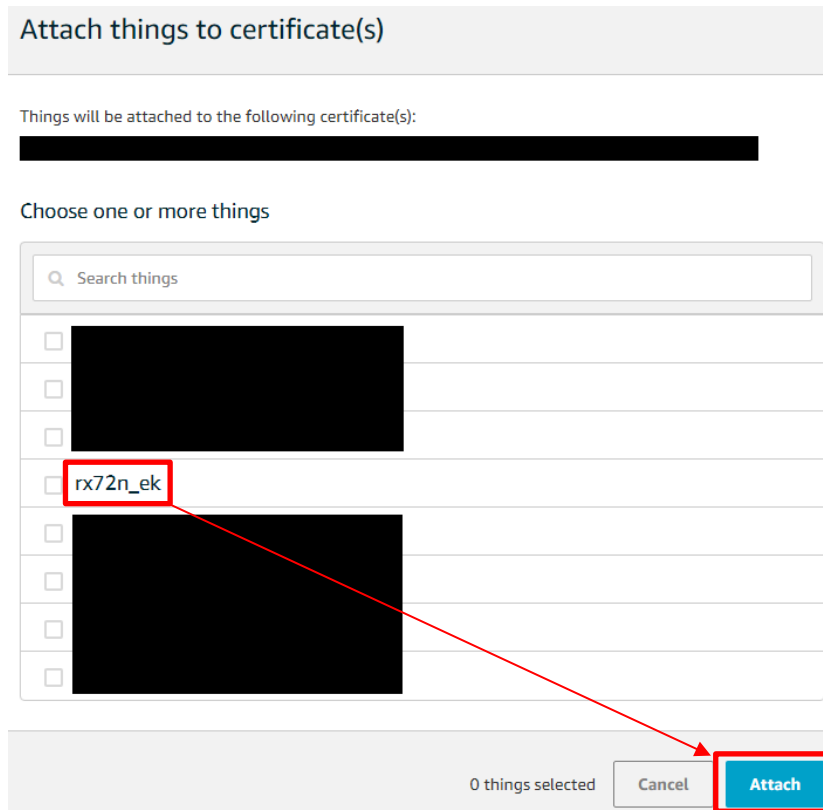
Select the previously created certificate from the list.



Under **Actions**, select **Attach thing**.



Select the previously created thing and click the **Attach** button.



This completes the procedure for attaching the certificate to the thing.

Copy the downloaded certificate and key files to the following location in the **key_cert_sig_generator** folder of the sample project.

```
key_cert_sig_generator
|-- ca
|   |-- AmazonRootCA1.pem
|   |-- AmazonRootCA3.pem
|-- ca-sign-keypair-rsa2048
|-- client-ecc256
|   |-- *-certificate.pem.crt
|   |-- prime256v1-csr.pem.csr
|   |-- prime256v1-private.pem.key
|-- client-rsa2048
|-- output
|--1_rsa2048_convertCert.sh
|--2_1_ecc256_generateKeyPair.sh
|--2_2_ecc256_convertCert.sh
|--3_showkeyValues.sh
|--convertCert.sh
```

3.2.4 Root CA Certificate Signature Generation and Certificate File Format Conversion

Certificates used for TLS are generally provided in PEM format. To use certificates with the TSIP driver, you must first convert them from PEM format to DER format. Convert just the RSA or ECDSA certificate, or convert both types of certificates, to match the cipher suite you are using. Also, append a signature to the root CA certificate.

3.2.4.1 RSA Certificate

Follow the steps below to convert the RSA root CA certificate (**AmazonRootCA1.pem**) and client certificate to DER format. In addition, we will generate an RSA 2048-bit key pair for generating and verifying the root CA certificate signature and then use the private key from the generated key pair to generate a signature for the root CA certificate.

Run **1_rsa2048_convertCrt.sh**, located in the **key_crt_sig_generator** folder.

The contents of **1_rsa2048_convertCrt.sh** are listed below.

```
#!/bin/sh

# If the format of client certificate is RSA, this project utilize RSA root
CA certificate.

INPUT_CRT_CLIENT_PEM="./client-rsa2048/*-certificate.pem.crt"
OUTPUT_CRT_CLIENT_DER="./output/client_rsa2048_crt.der"
OUTPUT_CRT_CLIENT_TXT="./output/client_rsa2048_crt_array.txt"

INPUT_CRT_ROOT_PEM="./ca/AmazonRootCA1.pem"
OUTPUT_CRT_ROOT_DER="./output/AmazonRootCA1_crt.der"
OUTPUT_CRT_ROOT_TXT="./output/AmazonRootCA1_crt_array.txt"

INPUT_KEY_PRIVATE_PEM="./ca-sign-keypair-rsa2048/rsa2048-private.pem"
INPUT_KEY_PUBLIC_PEM="./ca-sign-keypair-rsa2048/rsa2048-public.pem"

OUTPUT_CRT_ROOT_SIG_BIN="./output/AmazonRootCA1_sig.sig"
OUTPUT_CRT_ROOT_SIG_TXT="./output/AmazonRootCA1_sig_array.txt"

. ./convertCrt.sh
```


The contents of **convertCrt.sh**, which is called by **1_rsa2048_convertCrt.sh**, are listed below.

```
#!/bin/sh

# 1.Convert PEM format client certiccate to DER
if [ -e $INPUT_CERT_CLIENT_PEM ]
then
    openssl x509 -in $INPUT_CERT_CLIENT_PEM -out $OUTPUT_CERT_CLIENT_DER -
outform der
    hexdump -v -e '/1 "0x%02x, "' $OUTPUT_CERT_CLIENT_DER >
$OUTPUT_CERT_CLIENT_TXT
else
    echo "Client certificate is not found"
    exit 1
fi

# 2.Convert PEM format root CA certiccate to DER
if [ -e $INPUT_CERT_ROOT_PEM ]
then
    openssl x509 -in $INPUT_CERT_ROOT_PEM -out $OUTPUT_CERT_ROOT_DER -outform
der
    hexdump -v -e '/1 "0x%02x, "' $OUTPUT_CERT_ROOT_DER >
$OUTPUT_CERT_ROOT_TXT
else
    echo "Root CA certificate is not found"
    exit 1
fi

# 3.Generate RSA-2048 key pair for signature if it is not exist
if [ ! -e $INPUT_KEY_PRIVATE_PEM ] || [ ! -e $INPUT_KEY_PUBLIC_PEM ]
then
    openssl genrsa -out $INPUT_KEY_PRIVATE_PEM 2048
    openssl rsa -in $INPUT_KEY_PRIVATE_PEM -pubout -out
$INPUT_KEY_PUBLIC_PEM
fi

# 4.Create a signature of the Root CA certificate
if [ -e $INPUT_KEY_PRIVATE_PEM ] && [ -e $OUTPUT_CERT_ROOT_DER ]
then
    openssl dgst -sha256 -sigopt rsa_padding_mode:pss -sigopt
rsa_pss_saltlen:-1 -sign $INPUT_KEY_PRIVATE_PEM -out
$OUTPUT_CERT_ROOT_SIG_BIN $OUTPUT_CERT_ROOT_DER
    openssl dgst -sha256 -sigopt rsa_padding_mode:pss -verify
$INPUT_KEY_PUBLIC_PEM -signature $OUTPUT_CERT_ROOT_SIG_BIN
$OUTPUT_CERT_ROOT_DER
    hexdump -v -e '/1 "0x%02x, "' $OUTPUT_CERT_ROOT_SIG_BIN >
$OUTPUT_CERT_ROOT_SIG_TXT
fi
```

After running the script, the six files listed below are generated in the **output** folder. Of these, the files listed in **red** are used by the sample project. These files contain generated binary data in C language uint8_t array format. Copy these files to the **amazon-freertos/vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/src/smc_gen/general** folder of the sample project, overwriting the files of the same names.

- AmazonRootCA1_crt_array.txt
- AmazonRootCA1_sig_array.txt
- client_rsa2048_crt_array.txt
- AmazonRootCA1_crt.der
- AmazonRootCA1_sig.sig
- client_rsa2048_crt.der

3.2.4.2 ECDSA Certificate

Follow the steps below to convert the ECDSA root CA certificate (**AmazonRootCA3.pem**) and client certificate to DER format. In addition, we will generate an RSA 2048-bit key pair for generating and verifying the root CA certificate signature and then use the private key from the generated key pair to generate a signature for the root CA certificate.

Run **2_2_ecc256_convertCrt.sh**.

The contents of **2_2_ecc256_convertCrt.sh** are listed below.

```
#!/bin/sh

# If the format of client certificate is ECDSA, this project utilize ECDSA
root CA certificate.

INPUT_CERT_CLIENT_PEM="./client-ecc256/*-certificate.pem.crt"
OUTPUT_CERT_CLIENT_DER="./output/client_ecc256_crt.der"
OUTPUT_CERT_CLIENT_TXT="./output/client_ecc256_crt_array.txt"

INPUT_CERT_ROOT_PEM="./ca/AmazonRootCA3.pem"
OUTPUT_CERT_ROOT_DER="./output/AmazonRootCA3_crt.der"
OUTPUT_CERT_ROOT_TXT="./output/AmazonRootCA3_crt_array.txt"

INPUT_KEY_PRIVATE_PEM="./ca-sign-keypair-rsa2048/rsa2048-private.pem"
INPUT_KEY_PUBLIC_PEM="./ca-sign-keypair-rsa2048/rsa2048-public.pem"

OUTPUT_CERT_ROOT_SIG_BIN="./output/AmazonRootCA3_sig.sig"
OUTPUT_CERT_ROOT_SIG_TXT="./output/AmazonRootCA3_sig_array.txt"

. ./convertCrt.sh
```

After running the script, the six files listed below are generated in the **output** folder. Of these, the files listed in **red** are used by the sample project. These files contain generated binary data in C language uint8_t array format. Copy these files to the **amazon-freertos/vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/src/smc_gen/general** folder of the sample project, overwriting the files of the same names.

- **AmazonRootCA3_crt_array.txt**
- **AmazonRootCA3_sig_array.txt**
- **client_ecc256_crt_array.txt**
- AmazonRootCA3_crt.der
- AmazonRootCA3_sig.sig
- client_ecc256_crt.der

3.2.5 Key Wrapping

The TSIP driver will not accept input of user keys in plaintext, so the keys must be “wrapped” to convert them to a format that will be accepted by the TSIP driver. The procedure for converting the keys used for TLS to the format the TSIP driver accepts, and writing them to the device, is described below.

As with certificates, keys used for TLS are generally provided in PEM format. For use by the TSIP driver, the keying data must be extracted from the PEM format key file. After this, follow the steps below to wrap it using the Renesas DLM server (<https://dlm.renesas.com/>) and Renesas Secure Flash Programmer.

1. Extract the keying data for root CA certificate signature verification and the client certificate keying data from the PEM format key files.

Run **3_showkeyValues.sh**.

```
#!/bin/sh

KEY_ROOT_SIGNATURE_RSA2048="./ca-sign-keypair-rsa2048/rsa2048-private.pem"
KEY_CLIENT_RSA2048="./client-rsa2048/*-private.pem.key"
KEY_CLIENT_ECC256="./client-ecc256/prime256v1-private.pem.key"

echo Root CA signature RSA-2048bit Public:
if [ -e $KEY_ROOT_SIGNATURE_RSA2048 ]
then
    openssl asn1parse -in $KEY_ROOT_SIGNATURE_RSA2048 | awk -F:
'NR==3{print $4} NR==4{printf("%08d\n", $4)}' | tr -d "\n"
else
    echo "Not found"
fi

echo -e "\n\nClient RSA-2048bit All:"
if [ -e $KEY_CLIENT_RSA2048 ]
then
    openssl asn1parse -in $KEY_CLIENT_RSA2048 | awk -F: 'NR==3{print $4}
NR==4{printf("%08d\n", $4)} NR==5{print $4}' | tr -d "\n"
else
    echo "Not found"
fi

echo -e "\n\nClient ECC-256bit All: "
if [ -e $KEY_CLIENT_ECC256 ]
then
    TEMP_PRIVATE=`openssl ec -in $KEY_CLIENT_ECC256 -text -noout | sed -n
3,5p`
    TEMP_PUBLIC=`openssl ec -in $KEY_CLIENT_ECC256 -text -noout | sed -n
7,11p`
    echo $TEMP_PUBLIC $TEMP_PRIVATE | sed "s/^04//" | tr -d " :"
else
    echo "Not found"
fi
```

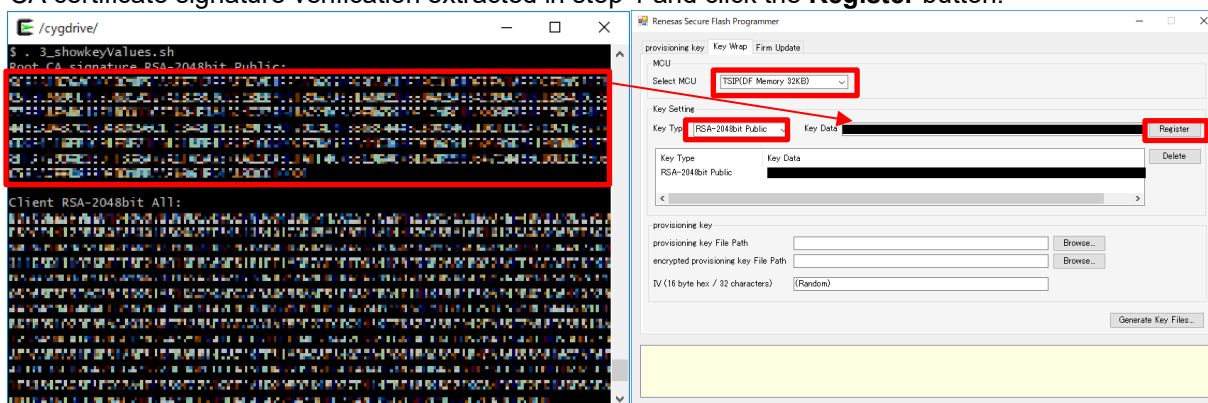
When you run the script, the keying data for root CA certificate signature verification and the client certificate keying data are output to the console in a format that can be input to the **Key Data** field on the **Key Wrap** tab of Renesas Secure Flash Programmer.

2. Generate a random provisioning key and then upload it to the DLM server to generate an encrypted provisioning key. The provisioning key is used to wrap the public key used for signature verification. You can use Renesas Secure Flash Programmer to create a provisioning key file in a format that will be accepted by the DLM server. Refer to section 5.1 for instructions for creating a provisioning key file. For information on using the DLM server, refer to the KeyWrap Service Operation Manual that is accessible via the FAQ link on the DLM server top page.
3. Next, enter the encrypted provisioning key and plaintext provisioning key, and the keying data for root CA certificate signature verification and client certificate keying data extracted in step 1, in Renesas Secure Flash Programmer and generate encrypted key files (**key_data.c** and **key_data.h**). The key for root CA certificate signature verification and client certificate keys are output to the encrypted key files as keys (encrypted keys) wrapped using the encrypted provisioning key and provisioning key. Refer to section 5.2 for a detailed description of the steps involved. Follow the steps below to enter the key for root CA certificate signature verification and client certificate key pair in Renesas Secure Flash Programmer.

3-1. Entering Key for Root CA Certificate Signature Verification

For **Key Type** on the **Key Wrap** tab, specify **RSA-2048bit Public** as the type of the key pair to be registered.

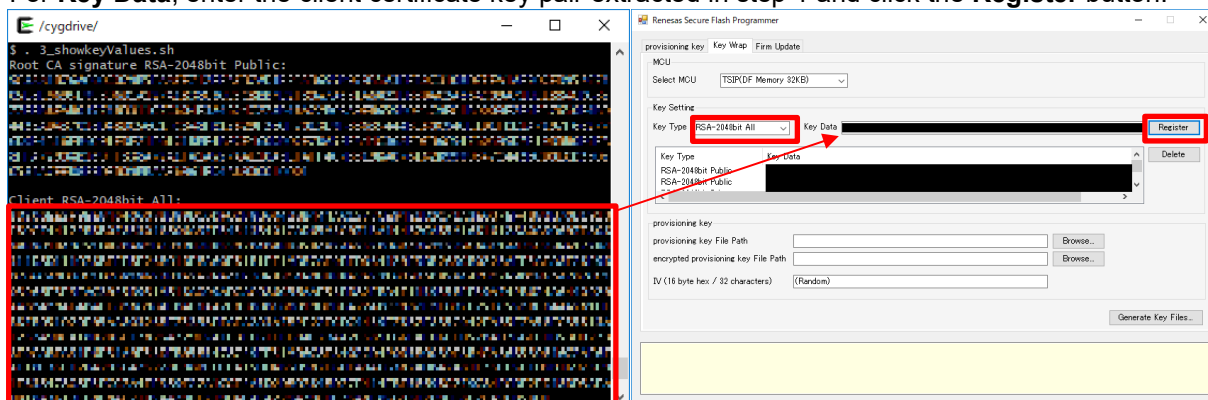
For **Key Data** on the **Key Wrap** tab of Renesas Secure Flash Programmer, enter the keying data for root CA certificate signature verification extracted in step 1 and click the **Register** button.



3-2. Entering Client Certificate Key Pair

For **Key Type** on the **Key Wrap** tab, specify the type of the key pair to be registered. Specify **RSA-2048bit All** when using an RSA client certificate and **ECC-256bit All** when using an ECDSA client certificate.

For **Key Data**, enter the client certificate key pair extracted in step 1 and click the **Register** button.



Copy the generated encrypted key files (**key_data.c** and **key_data.h**) to the **amazon-freertos/vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/src/smc_gen/general** folder of the sample project, overwriting the files of the same names.

3.3 Settings for Communication with AWS

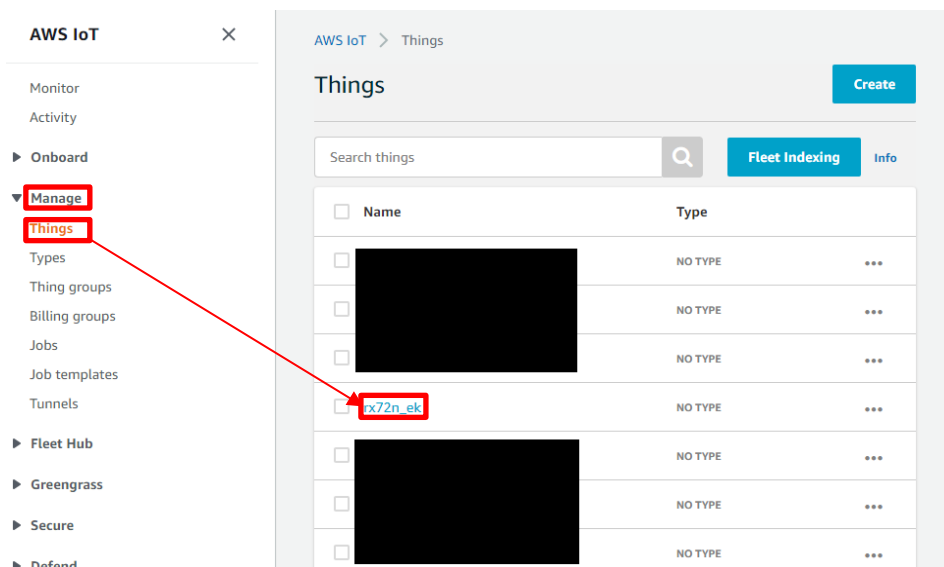
The procedure for making settings in the setting file supplied with FreeRTOS and the setting file of the sample program is described below.

3.3.1 AWS IoT Settings

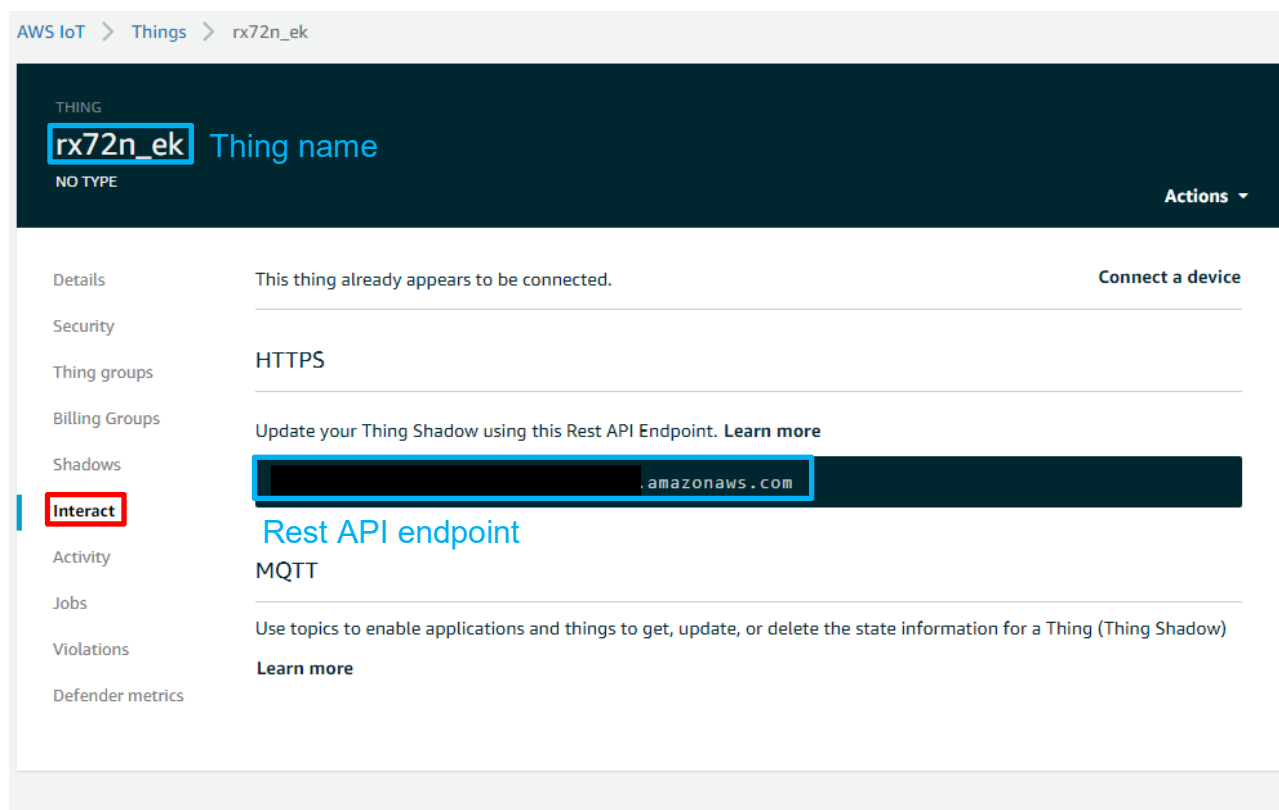
Open the file `demos/include/aws_clientcredential.h` in the **amazon-freertos** folder and enter the following settings.

- For `#define clientcredentialMQTT_BROKER_ENDPOINT`, enter the Rest API endpoint.
- For `#define clientcredentialIOT_THING_NAME`, enter the name of the thing.

To confirm the Rest API endpoint and thing name, sign in to the AWS Management Console (<https://aws.amazon.com/console/>) and select **All Services** → **Internet of Things** → **IoT Core**. Under **Manage** click **Things**, and select the thing created as described in section 3.2.



Click **Interact**, and on the page that is displayed the name of the thing is shown at the top left, and the Rest API endpoint is shown under **HTTPS**.



3.3.2 IP Settings

The default setting is to use DHCP. If DHCP has been disabled on the router connected to the target board, make settings as follows.

Open the file **vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/config_files/FreeRTOSIPConfig.h** in the **amazon-freertos** folder, and set **#define ipconfigUSE_DHCP** to **0**.

Open the file **vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/config_files/FreeRTOSConfig.h** in the **amazon-freertos** folder, and enter settings for IP address, default gateway, DNS server address, and subnet mask.

3.3.3 Client Certificate Format Selection

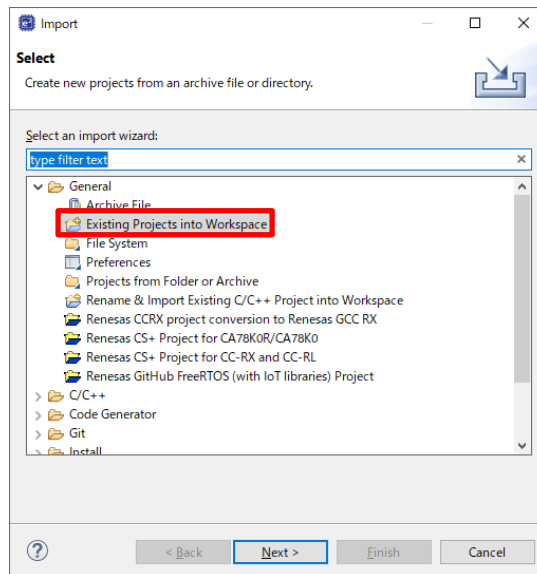
By default, the RSA certificate type is enabled. To switch to the ECDSA certificate type, open the file **amazon-freertos/vendors/renesas/boards/rx_mcu_boards/(board_name)/aws_demos/src/smc_gen/general/r_trust_certificate_data.h**, and edit it as shown below.

```
// RSA or ECDSA
// #define TSIP_CLIENT_CERTIFICATE_TYPE R_TSIP_TLS_PUBLIC_KEY_TYPE_RSA2048
// RSA
#define TSIP_CLIENT_CERTIFICATE_TYPE R_TSIP_TLS_PUBLIC_KEY_TYPE_ECDSA_P256
// ECDSA
```

4. Building and Running the Project

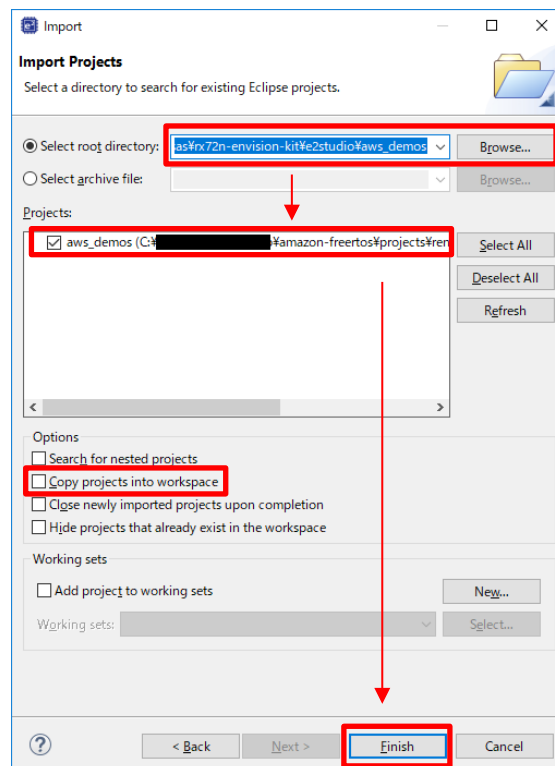
4.1 Importing the Project

Launch e² studio, go to a workspace of your choice, and select **File** → **Import**. Next, select **Existing Projects into Workspace** under **General**, as shown below.



Click the **Browse...** button next to the **Select root directory** item and select **amazon-freertos/projects/renesas/(board_name)/e2studio/aws_demos**. After confirming that a project called **aws_demos** is now selectable, click the **Finish** button.

At this time, check to make sure that **Copy projects into workspace** is unchecked.



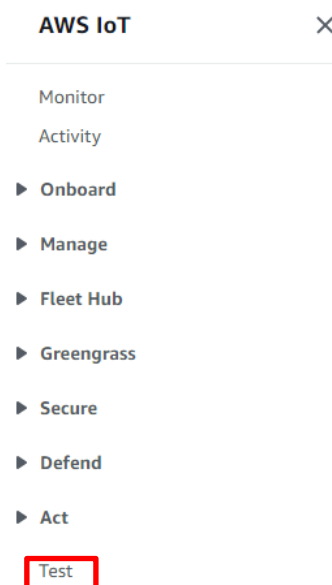
4.2 Building the Project

Select **Project** → **Build All** to build the project. Note that a warning message appears at this time, but this does not indicate a problem.

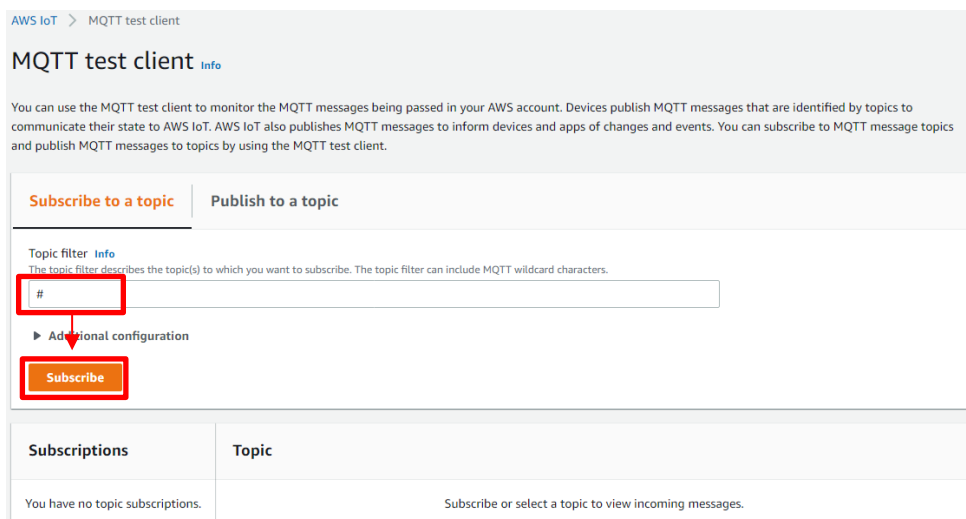
After the build finishes, connect the target board to the PC and router as shown in Figure 3.1. Then select **Run** → **Debug** to start debugging.

4.3 Connecting to AWS IoT

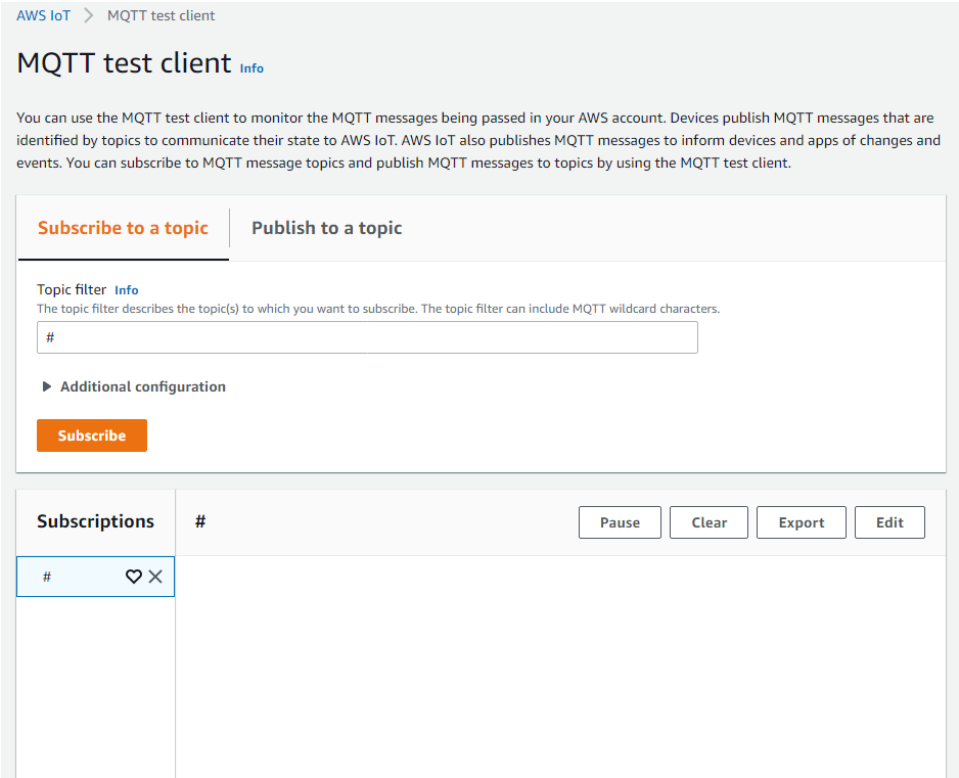
Sign in to the AWS Management Console (<https://aws.amazon.com/console/>) and select **All Services** → **Internet of Things** → **IoT Core**. From the menu on the left, select **Test** to launch the MQTT test client.



In the **Topic filter** field enter the wildcard character #, then click **Subscribe**.

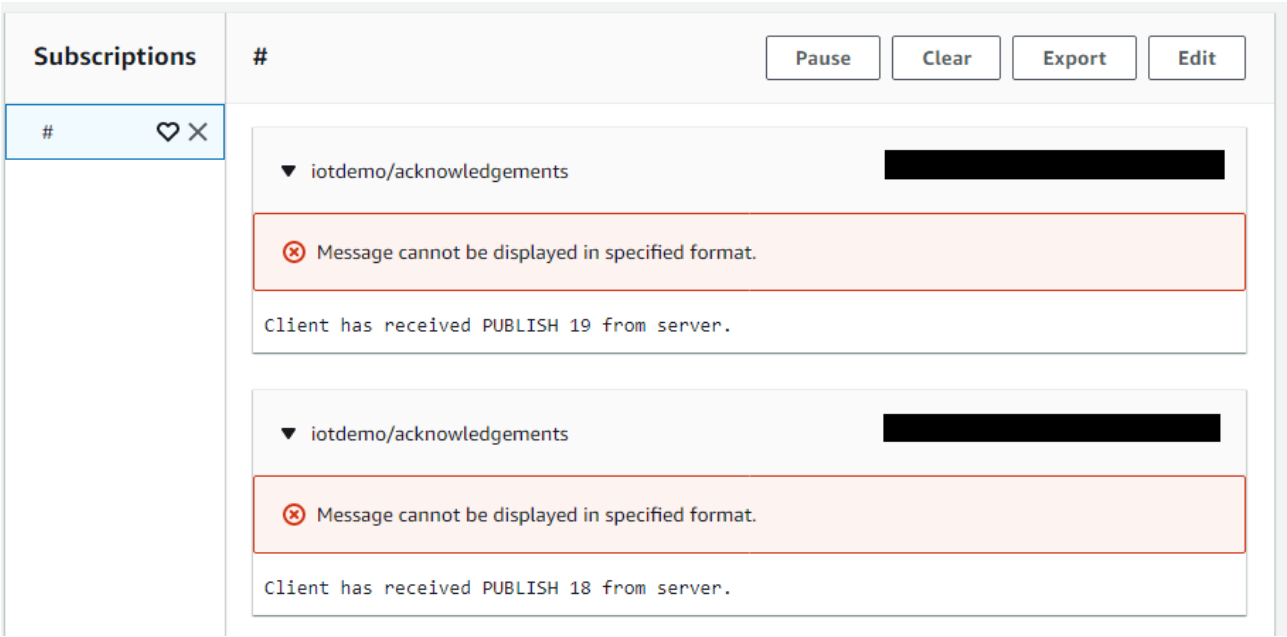


Confirm that an empty console is displayed at the bottom of the page, as shown below.

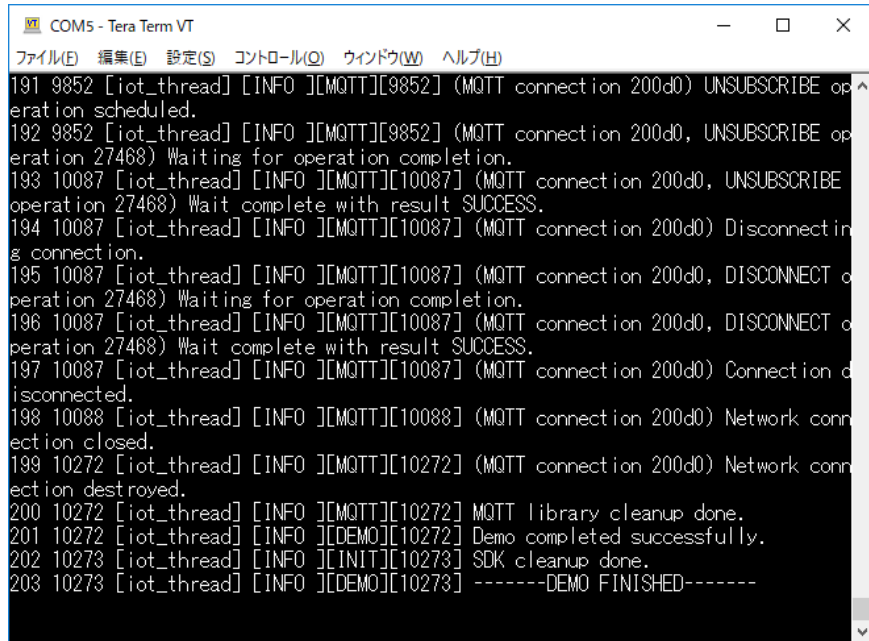


Launch the terminal emulator of your choice, such as Tera Term, and establish a serial communication link with the target board. Next, in e² studio, select **Run** and then click the **Resume** button to run the program, which will connect to AWS.

When a connection to AWS is successfully established, a communication log is output on the MQTT client.



In addition, a task log is displayed on the terminal emulator.



```
COM5 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)
191 9852 [iot_thread] [INFO ][MQTT][9852] (MQTT connection 200d0) UNSUBSCRIBE op
eration scheduled.
192 9852 [iot_thread] [INFO ][MQTT][9852] (MQTT connection 200d0, UNSUBSCRIBE op
eration 27468) Waiting for operation completion.
193 10087 [iot_thread] [INFO ][MQTT][10087] (MQTT connection 200d0, UNSUBSCRIBE
operation 27468) Wait complete with result SUCCESS.
194 10087 [iot_thread] [INFO ][MQTT][10087] (MQTT connection 200d0) Disconnectin
g connection.
195 10087 [iot_thread] [INFO ][MQTT][10087] (MQTT connection 200d0, DISCONNECT o
peration 27468) Waiting for operation completion.
196 10087 [iot_thread] [INFO ][MQTT][10087] (MQTT connection 200d0, DISCONNECT o
peration 27468) Wait complete with result SUCCESS.
197 10087 [iot_thread] [INFO ][MQTT][10087] (MQTT connection 200d0) Connection d
isconnected.
198 10088 [iot_thread] [INFO ][MQTT][10088] (MQTT connection 200d0) Network conn
ection closed.
199 10272 [iot_thread] [INFO ][MQTT][10272] (MQTT connection 200d0) Network conn
ection destroyed.
200 10272 [iot_thread] [INFO ][MQTT][10272] MQTT library cleanup done.
201 10272 [iot_thread] [INFO ][DEMO][10272] Demo completed successfully.
202 10273 [iot_thread] [INFO ][INIT][10273] SDK cleanup done.
203 10273 [iot_thread] [INFO ][DEMO][10273] -----DEMO FINISHED-----
```

5. Using Renesas Secure Flash Programmer

5.1 Generating a Provisioning Key File

Figure 5.1 shows the **provisioning key** tab in Renesas Secure Flash Programmer. For **provisioning key Value**, enter the value of the provisioning key in 32-byte hexadecimal format, then click the **format to DLM server file...** button to create a provisioning key file that can be sent to the DLM server.

By clicking the **format to DLM server file...** button when **(Random)** is displayed you can generate a provisioning key file using a random number as the basis, but this method should not be used for production products because the random number used lacks sufficient precision.

Figure 5.1 Provisioning Key Tab in Renesas Secure Flash Programmer

5.2 Generating Encrypted Key Files

Figure 5.2 shows the **Key Wrap** tab in Renesas Secure Flash Programmer, and Table 5.1 contains descriptions of the **Key Wrap** tab setting values. Enter setting values based on the descriptions in Table 5.1, then click the **Generate Key Files...** button to generate the encrypted key files (**key_data.c** and **key_data.h**). Refer to Table 5.2 for descriptions of the various buttons.

Figure 5.2 Key Wrap Tab in Renesas Secure Flash Programmer

Table 5.1 Key Wrap Tab Setting Values

Parameter	Setting Value	Description
Select MCU	<ul style="list-style-type: none"> • TSIP-Lite(DF Memory 8KB) • TSIP-Lite(DF Memory 32KB) • TSIP(DF Memory 8KB) • TSIP(DF Memory 32KB) 	Selects the MCU (TSIP type and data flash memory size) to be used. For the sample program, specify TSIP(DF Memory 32KB) .
Key Type	<ul style="list-style-type: none"> • AES-128bit • AES-256bit • DES • 2Key-TDES • Triple-DES • ARC4-2048bit • SHA1-HMAC • SHA256-HMAC • RSA-1024bit Public/Private/All • RSA-2048bit Public/Private/All • RSA-3072bit Public • RSA-4096bit Public • ECC-192bit Public/Private/All • ECC-224bit Public/Private/All • ECC-256bit Public/Private/All • ECC-384bit Public/Private/All • Update Key Ring 	<p>Specifies the type of the user key to be generated. For the sample program, specify the following:</p> <ul style="list-style-type: none"> • RSA-2048bit Public Where Key Data is keying data used to verify the signature of the root CA certificate. • RSA-2048bit All Where Key Data is keying data for an RSA client certificate. • ECC-256bit All Where Key Data is keying data for an ECDSA client certificate.
Key Data	User key Refer to section 7 of RX Family: TSIP (Trusted Secure IP) Module Firmware Integration Technology (R20AN0371) for the key format.	Specify the user keying data to be generated. For the sample program, the key specified by Key Type above is used.
provisioning key File Path	File path of provisioning key file	Specify the file path of the plaintext provisioning key file to be used when encrypting the user key. Refer to section 5.1 for instructions for creating a provisioning key file.
encrypted provisioning key File Path	File path of wrapped provisioning key file	Specify the file path of the wrapped provisioning key file to be output as a C language file. Use the DLM server to create the wrapped provisioning key file.
IV (16 byte hex / 32 characters)	IV value	Enter a 16-byte IV value. The user key input to the TSIP must have an encrypted MAC value appended, and the IV value is used as the initialization vector when calculating the MAC value.
Generate Key Files...	Button for generating C language files	Outputs C language encrypted key files.

Table 5.2 Descriptions of Buttons on Key Wrap Tab

Button	Description
Register	Registers the user keying data specified in the Key Data field. Enter user keying data matching Key Type in the Key Data field and click this button to register it.
Delete	Deletes registered user keying data. First select in the window the user keying data to be deleted, then click this button to delete it.
Browse...	Click these buttons to use Explorer to specify the file paths to the provisioning key file and the wrapped provisioning key file. You can also enter file paths directly.
Generate Key Files...	Generates encrypted key files (key_data.c and key_data.h). Click this button after entering appropriate values into the various fields. After the button is clicked a dialog box appears for specifying the output folder for the encrypted key files, and the files are output.

6. Appendix

6.1 TLS Communication Performance Using TSIP Driver

Table 6.1 lists examples of application data transfer speeds for TLS communication using the RX72N Envision Kit. The MCU's internal timer was used to measure transfer times for 1 MB of data, five transfers were performed, and the average time was calculated. In these examples, using the TSIP driver boosted the transfer rate from 3 to 7 Mbps to 20 to 30 Mbps.

Table 6.1 Examples of TLS Communication Speeds Using TSIP Driver

Cipher Suite	Block Cipher	Mbed TLS* ¹	Mbed TLS with TSIP* ²
TLS_RSA_WITH_AES_128_CBC_SHA	128-bit AES-CBC	Up: 6.4 Mbps Down: 6.6 Mbps	Up: 25.0 Mbps Down: 28.3 Mbps
TLS_RSA_WITH_AES_256_CBC_SHA	256-bit AES-CBC	Up: 5.5 Mbps Down: 5.6 Mbps	Up: 24.2 Mbps Down: 27.2 Mbps
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	128-bit AES-GCM	Up: 3.7 Mbps Down: 3.8 Mbps	Up: 22.4 Mbps Down: 29.5 Mbps

Notes: System clock (ICLK): 240 MHz

TSIP operating clock (PCLKB): 60 MHz

1. Mbed TLS: Software processing

2. Mbed TLS with TSIP: Using TLS APIs of TSIP driver

6.2 Flowchart of TLS Negotiation and Calls to TSIP Driver

Below are flowcharts of TLS negotiation overall and associated calls to the TSIP driver. Figure 6.1 applies when the key exchange algorithm is RSA and Figure 6.2 when it is ECDHE.

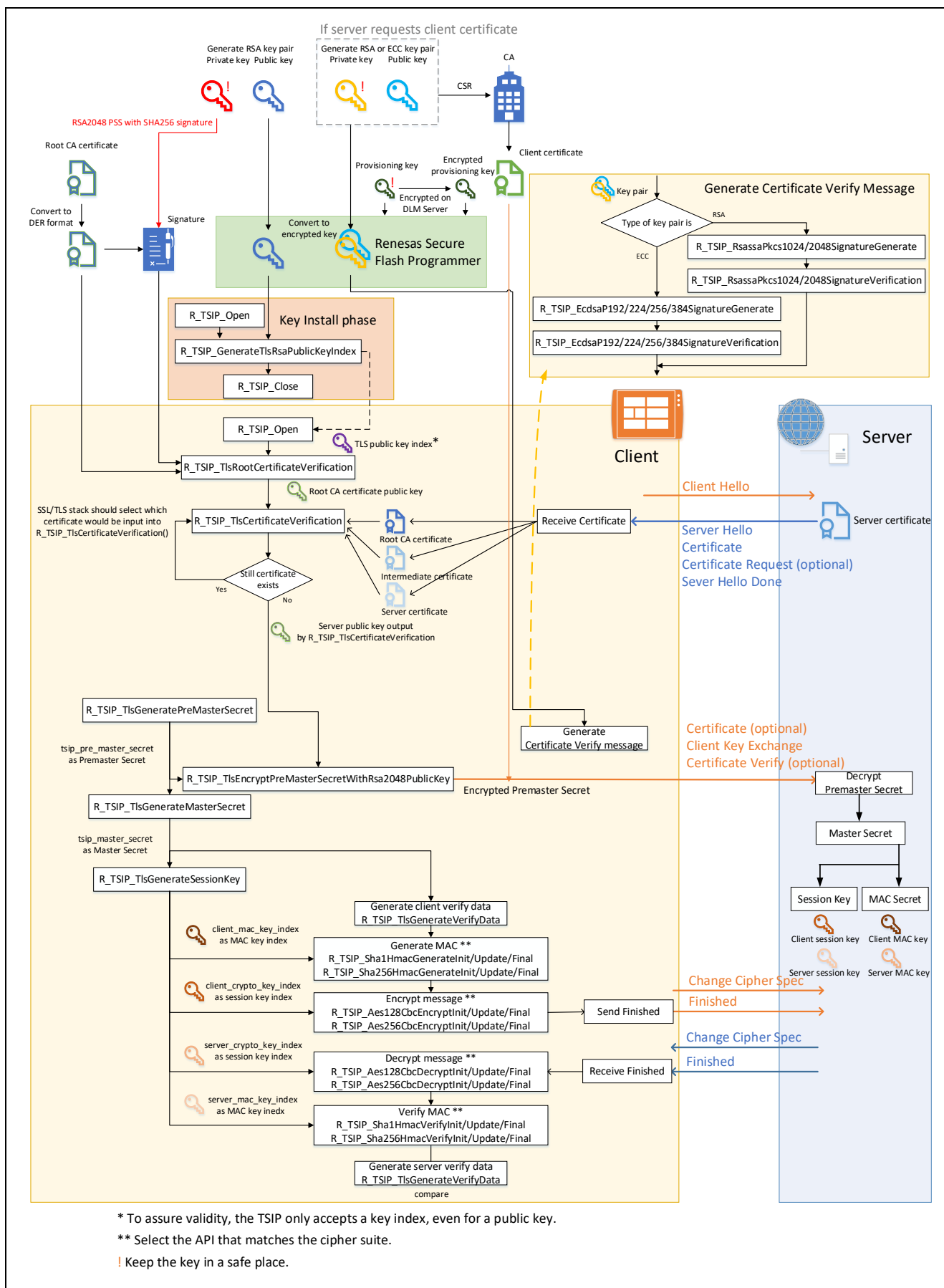


Figure 6.1 Flowchart of TLS Negotiation and Associated Calls to TSIP Driver (RSA Key Exchange Algorithm)

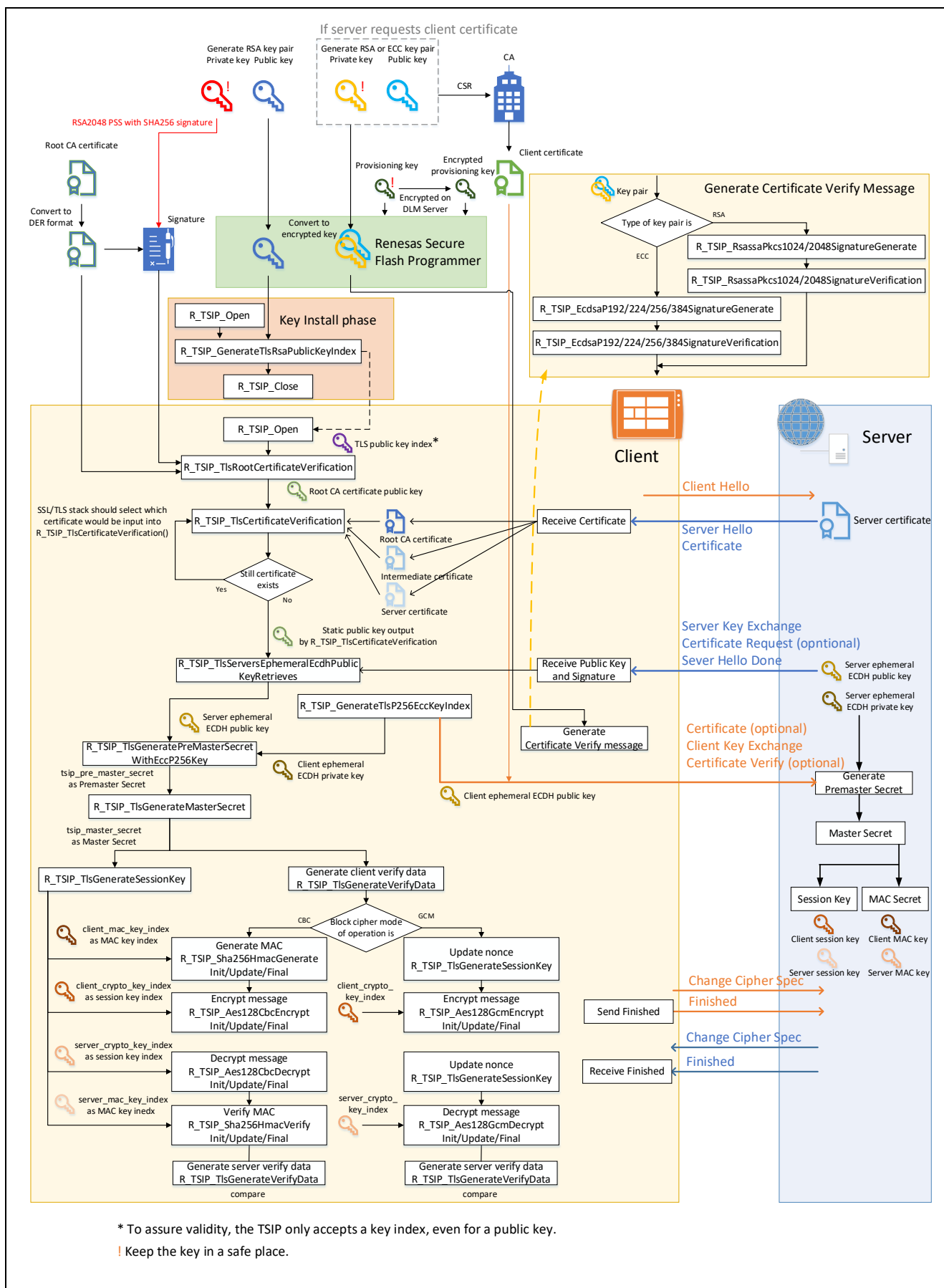


Figure 6.2 Flowchart of TLS Negotiation and Associated Calls to TSIP Driver (ECDHE Key Exchange Algorithm)

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jun. 30, 2021	—	First edition issued
1.01	Mar. 31, 2022	—	Error correction
1.02	Sep. 15, 2022	16	2.3.3 Added explanation about generating Certificate Verify

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.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

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

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

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

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

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/.