Renesas Synergy™ Platform

SCE HAL Module Guide

R11AN0088EU0101
Rev.1.01
Sep 14, 2017

This module guide will enable you to effectively use a module in your own design. Upon completion of this guide, you will be able to add this module to your own design, configure it correctly for the target application and write code, using the included application project code as a reference and efficient starting point. References to more detailed API descriptions and suggestions of other application projects that illustrate more advanced uses of the module are available in the Renesas Synergy Knowledge Base (as described in the References section at the end of this document), and should be valuable resources for creating more complex designs.

The Secure Cryptographic Engine (SCE) HAL module is a high-level API for random number generation, digest computing (hash), data encryption and decryption, and digital signing and verification. It is implemented on r_sce. The SCE is a dedicated hardware block that can perform cryptography related functions. The functionality provided by the SCE varies across the Synergy MCU Series.

Contents

1. SCE HAL Module Features

The SCE HAL module configures the cryptographic module, which allows the user to build cryptographic protocols for security with the following cryptographic primitives:

- Random-number generation
- Data encryption and decryption using AES or Triple DES (3DES) algorithms
- Signature generation and verification using the RSA or DSA algorithms
- Message-digest computation using HASH algorithms SHA1, SHA224, or SHA256

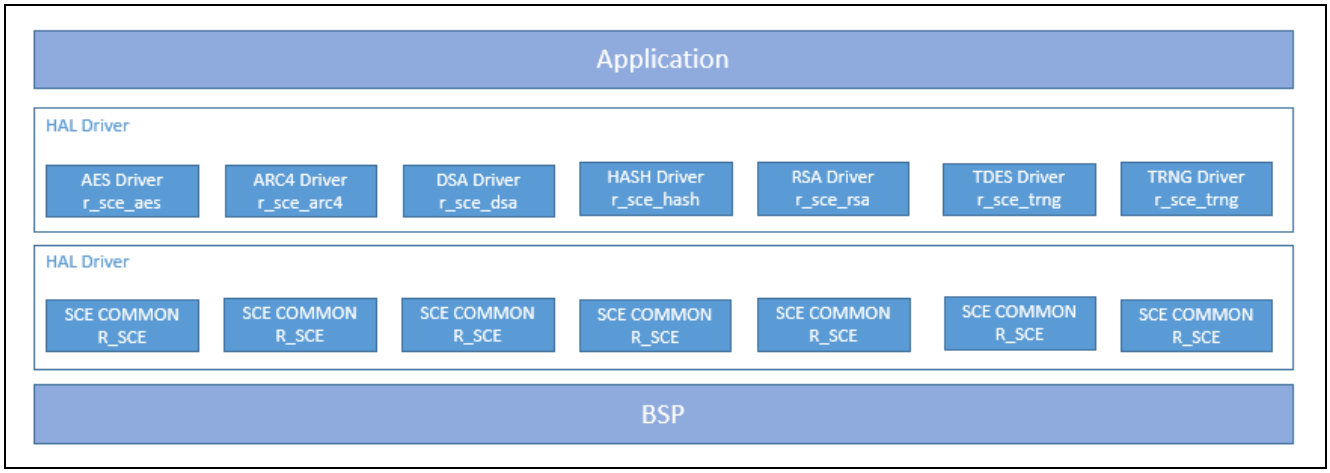

Figure 1 SCE HAL Module Block Diagram

2. SCE APIs Overview

The SCE interface provides a common API for SCE HAL modules. The SCE interface supports multiple operations depending on the chosen module (AES, ARC4, RSA, DSA, HASH, TDES, or TRNG).

The AES interface defines APIs for opening, closing, encrypting and decrypting data using the AES algorithm; it uses a 128-bit, 192-bit or 256-bit key and ECB, CBC, CTR, GCM or XTS chaining-mode options. A complete list of the available APIs, an example API call and a short description of each can be found in the following table. For status return values, refer to the SCE API References section of the SSP User's Manual.

Table 1 AES HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_aes.p_api->open(g_sce_aes.p_ctrl, g_sce_aes.p_cfg);` AES module open function. Must be called before performing any encrypt/decrypt operations.
.createKey	`g_sce_aes.p_api->close(g_sce_aes.p_ctrl, num_words, p_key);` Generate an AES key for encrypt/decrypt operations.
.encrypt	`g_sce_aes.p_api->encrypt(g_sce_aes.p_ctrl, p_key, p_vi, num_words, p_source, p_dest);` AES encryption using the chaining mode and padding mode specified in the `aes.open()` function call.
.addAdditionalAuthenticationData	`g_sce_aes.p_api->addAdditionalAuthenticationData (g_sce_aes.p_ctrl, p_key, p_vi, num_words, p_source);` Add additional authentication data (called before starting an encryption or decryption operation).

.encryptFinal	`g_sce_aes.p_api->encryptFinal(g_sce_aes.p_ctrl, p_key, p_iv, input_num_words, p_source, output_num_words, p_dest);` AES final encryption using the chaining mode and padding mode specified in the `aes.open()` function call.
.decrypt	`g_sce_aes.p_api->decrypt(g_sce_aes.p_ctrl, p_key, p_iv, num_words, p_source, p_dest);` AES decryption.
.setGcmTag	`g_sce_aes.p_api->setGcmTag(g_sce_aes.p_ctrl, words, &source);` Set the GCM tag.
.getGcmTag	`g_sce_aes.p_api->getGcmTag(g_sce_aes.p_ctrl, words, &destination);` Get the GCM tag.
.close	`g_sce_aes.p_api->close(g_sce_aes.p_ctrl);` Close the module.
.zeroPaddingEncrypt	`g_sce_aes.p_api->zeroPaddingEncryption(g_sce_aes.p_ctrl,&key, &iv, bytes, &source, &destination);` Zero Padding encryption.
.zeroPadding Decrypt	`g_sce_aes.p_api->zeroPaddingDecryption(g_sce_aes.p_ctrl, &key, &iv, bytes, &source, &dest);` Zero Padding decryption.
.versionGet	`g_sce_aes.p_api->decrypt(&version);` Get the API version using the version pointer.

The ARC4 interface defines APIs for opening, closing, setting a key, and processing data. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 2 ARC4 HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_arc4.p_api->open(g_sce_arc4.p_ctrl, g_sce_trng.p_cfg);` Open the ARC4 module.
.keySet	`g_sce_arc4.p_api->keySet(g_sce_arc4.p_ctrl, &rngbuf, nbytes);` Set the key to be used by the ARC4 module.
.arc4Process	`g_sce_arc4.p_api-> arc4Process(g_sce_arc4.p_ctrl, nbytes, &source, &destination);` Encrypt or decrypt data using the ARC4 module.
.close	`g_sce_arc4.p_api->close(g_sce_arc4.p_ctrl);` Close the ARC4 module.
.versionGet	`g_sce_arc4.p_api->versionGet (&version);` Retrieve the version using the version pointer.

The DSA interface defines APIs for opening, closing, digital signing and verification. Available options include a 1024-bit public key and a 160-bit private key, a 2048-bit public key and a 224-bit private key, or a 2048-bit public key and a 256-bit private key. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 3 DSA HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_dsa.p_api->open(g_sce_dsa.p_ctrl, g_sce_dsa.p_cfg);` DSA module open function. Must be called before performing any sign/verify operations.
.verify	`g_sce_dsa.p_api->verify(p_key, p_domain, num_words, p_signature, p_paddedHash);` DSA signature verification using given DSA public key. This function is deprecated. The function hashVerify should be used instead.
.hashVerify	`g_sce_dsa.p_api->hashVerify(g_sce_dsa.p_ctrl, p_key, p_domain, num_words, p_signature, p_paddedHash);` DSA signature verification using given DSA public key.
.sign	`g_sce_dsa.p_api->sign(p_key, p_domain, num_words, p_padded_hash, p_dest);` DSA Signature generation using DSA private key. This function is deprecated. The function hashSign should be used instead.
.hashSign	`g_sce_dsa.p_api->hashSign(g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_padded_hash, p_dest);` DSA Signature generation using DSA private key.
.close	`g_sce_dsa.p_api->close(g_sce_dsa.p_ctrl);` Close the DSA module.
.versionGet	`g_sce_dsa.p_api->versionGet(p_version);` Gets version and stores it in provided pointer p_version.

The HASH interface defines APIs for calculating hash values for a given data-set. Available options include SHA1 and SHA256 algorithms. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 4 HASH HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_hash.p_api->open(g_sce_hash.p_ctrl, g_sce_hash.p_cfg);` HASH module open function. Must be called before performing any sign/verify operations.
.updateHash	`g_sce_hash.p_api->updateHash(p_source, num_words, p_dest);` Update hash for the num_words words from source buffer p_source. This function is deprecated. The function hashUpdate should be used instead.
.hashUpdate	`g_sce_hash.p_api->hashUpdate(g_sce_hash.p_ctrl, p_source, num_words, p_dest);` Update hash for the num_words words from source buffer p_source.
.versionGet	`g_sce_hash.p_api->versionGet(p_version);` Gets version and stores it in provided pointer p_version.

The RSA interface defines APIs for opening, closing, encrypting and decrypting data using an RSA algorithm as well as digitally signing and verifying the algorithm. The RSA interface employs a 1024-bit or 2048-bit key. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 5 RSA HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_rsa.p_api->open(g_sce_rsa.p_ctrl, g_sce_rsa.p_cfg);` RSA module open function. Must be called before performing any encrypt/decrypt or sign/verify operations.
.encrypt	`g_sce_rsa.p_api->encrypt(g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_source, p_dest);` Encrypt source data from p_source using an RSA public key from p_key and write the results to destination buffer p_dest.
.decrypt	`g_sce_rsa.p_api->decrypt (g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_source, p_dest);` Decrypt source data from p_source using an RSA private key from p_key and write the results to destination buffer p_dest.
.decryptCrt	`g_sce_rsa.p_api->decryptCrt(g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_source, p_dest);` Decrypt source data from p_source using an RSA private key from p_key and write the results to destination buffer p_dest. RSA private key data is specified in CRT format.
.verify	`g_sce_rsa.p_api->verify(g_sce_rsa.p_ctrl, num_words, p_source);` Verify signature given in buffer p_signature using the RSA public key p_key for the given padded message hash from buffer p_padded_hash.
.sign	`g_sce_rsa.p_api->sign(g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_padded_hash, p_dest);` Generate signature for the given padded hash buffer p_padded_hash using the RSA private key p_key. Write the results to the buffer p_dest.
.signCrt	`g_sce_rsa.p_api->signCrt(g_sce_rsa.p_ctrl, p_key, p_domain, num_words, p_padded_hash, p_dest);` Generate signature for the given padded hash buffer p_padded_hash using the RSA private key p_key. RSA private key p_key is assumed to be in CRT format. Write the results to the buffer p_dest.
.close	`g_sce_rsa.p_api->close(g_sce_rsa.p_ctrl);` Close the RSA module.
.versionGet	`g_sce_rsa.p_api->versionGet(p_version);` Gets version and stores it in provided pointer p_version.

The TDES interface defines APIs for encrypting and decrypting data according to the TDES standard. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 6 TDES HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_tdes.p_api->open(g_sce_tdes.p_ctrl, g_sce_tdes.p_cfg);` Open the TDES module.
.encrypt	`g_sce_tdes.p_api->read(g_sce_tdes.p_ctrl, &key, &iv, nwords, &source, &destination);` Encrypt the data.
.decrypt	`g_sce_trng.p_api->close(g_sce_tdes.p_ctrl, &key, &iv, nwords, &source, &destination);` Decrypt the data.
.close	`g_sce_tdes.p_api->close(g_sce_tdes.p_ctrl);` Close the TDES module.
.versionGet	`g_sce_tdes.p_api->versionGet(p_version);` Gets version and stores it in provided pointer p_version.

The TRNG interface defines APIs for computing the random-number generator. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table:

Table 7 TRNG HAL Module API Summary

Function Name	Example API Call and Description
.open	`g_sce_trng.p_api->open(g_sce_trng.p_ctrl, g_sce_trng.p_cfg);` Open the TRNG driver for reading random data from the hardware TRNG module.
.read	`g_sce_trng.p_api->read(g_sce_trng.p_ctrl, p_rngbuf, nbytes);` Generate nbytes of random number bytes and store them in p_rngbuf buffer.
.close	`g_sce_trng.p_api->close(g_sce_trng.p_ctrl);` Close the TRNG interface driver.
.versionGet	`g_sce_trng.p_api->versionGet(p_version);` Gets version and stores it in provided pointer p_version.

3. SCE HAL Module Operational Overview

Different cryptographic functions are available for different target MCUs. The following table shows what support is available for each individual MCU group:

Table 8 Available Cryptographic Functions

Function	S7G2, S5D9, S5D5	S3A7, S3A6	S124, S128	Notes
TRNG	Generate and read random number	Generate and read random number	Generate and read random number	Generate and read random number
AES	Encryption, decryption	Encryption, decryption	Encryption, decryption	Symmetric Key Encryption based on AES standard
AES Key Size	128-bit, 192-bit, 256-bit	128-bit, 256-bit	128-bit, 256-bit	
AES Chaining Modes	ECB, CBC, CTR, GCM, XTS	ECB, CBC, CTR, GCM	ECB, CBC, CTR	
TDES	Encryption, Decryption	NA	NA	
TDES Key Size	192-bit	NA	NA	
TDES Chaining Modes	ECB, CBC, CTR	NA	NA	
RSA	Signature Generation, Signature Verification, Public-key Encryption, Private-key Decryption	NA	NA	Supports CRT keys and standard keys for private key operations
RSA Key Size	1024-bit, 2048-bit	NA	NA	
DSA	Signature Generation, Signature Verification	NA	NA	
DSA Key Size	(1024, 128)-bit, (2048, 224)-bit, (2048, 256)-bit	NA	NA	
HASH	SHA1, SHA224, SHA256	NA	NA	Message digest algorithms

Configuration Settings for the AES Module

The AES module can be configured for a user-specified key length and chaining modes.

Configuration Settings for the ARC4 Module

The ARC4 module can be configured for a user-specified key length.

Configuration Settings for the DSA Module

The DSA module can be configured for a user-specified key length.

Configuration Settings for the HASH Module

The HASH module can be configured for a user specified HASH algorithm (depending on the target MCU).

Configuration Settings for the RSA Module

The RSA module can be configured for a user-specified key length.

Configuration Settings for the TDES Module

The TDES module can be configured for the chaining mode type.

Configuration Settings for the TRNG Module

Random number generation can be configured for the maximum number of attempts it makes to the underlying hardware to generate a unique 16-byte random number that differs from the previously-generated random number. On reaching the maximum number of attempts, the `read` API will return an error code to the caller. Otherwise a success code is returned and the generated random number will be transferred to the caller-supplied data buffer.

3.1 SCE HAL Module Operational Notes and Limitations

3.1.1 SCE HAL Module Operational Notes

- Synergy S7 and S5 devices have the SCE7 and therefore support AES, TRNG, RSA, HASH, and DSA.
- Synergy S3 devices have the SCE5 and therefore support AES, TRNG, and GHASH. GHASH is supported as part of the AES GCM mode. Synergy S3 devices do not support SHA1/SHA256 HASH functionality.
- Synergy S1 devices support AES and TRNG.
- If an unsupported module is added to the project, then a compiler warning will be generated indicating this fact.
- All crypto APIs may return `SSP_ERR_ASSERTION` on null pointer input or invalid input parameters. All APIs return error codes documented in `sf_crypto_err_t` or `ssp_err_t` which are within the width of the type uint32_t.
- Crypto hardware engine does not support re-entrancy. When the crypto hardware engine is busy performing a task, any new request will receive a status error code `SSP_ERR_CRYPTO_SCE_RESOURCE_CONFLICT`.
- Endianness configuration parameter usage:
  - The default mode is big endian, where the input and output parameters (example: keys, payload, and initialization vector (IV)) are required to be in uint32_t data type.
  - The little endian mode allows the user to have uint8_t/byte array for input and output parameters (example: keys, payload, and IV) and they should be cast to (uint32_t *).
  - The endianness configuration is set at the initialization of the SCE module and remains in effect until the module is closed. All data should be formatted accordingly.
  - Example:
    - Set the big endian mode when the data is in uint32_t array and big endian format:
      ```
      uint32_t  test_data[5] =
      {0x84983E44, 0x1C3BD26E, 0xBAAE4AA1, 0xF95129E5, 0xE54670F1};
      ```
    - Set the little endian mode when the same data is in byte array format
      ```
      uint8_t  test_data_byte_array[20] =
      {0x84, 0x98, 0x3E, 0x44, 0x1C, 0x3B, 0xD2, 0x6E, 0xBA, 0xAE, 0x4A, 0xA1,
      0xF9, 0x51, 0x29, 0xE5, 0xE5, 0x46, 0x70, 0xF1};
      ```

3.1.2 SCE HAL Module Limitations

- The AES `encrypt()` and `decrypt()` functions do not support data padding. These functions operate on data lengths that are multiples of 16 bytes. (Data padding needs to be handled by the user application.) AES GCM mode may require support for authentication data that may not be a multiple of 16 bytes. To support this, `zeroPaddingEncrypt()` and `zeroPaddingDecrypt()` function APIs are provided for the AES GCM mode.
- AES `createKey` API is implemented only for GCM Mode for generating device specific encrypted key.
- The TDES `encrypt()` and `decrypt()` functions do not support data padding. These functions operate on data lengths that are multiples of 8 bytes. (Data padding needs to be handled by the user application.)
- Refer to the most recent SSP release notes for the most up-to-date limitations on this module.

4. Including the SCE HAL Module in an Application

This section describes how to include the SCE HAL module in an application using the SSP configurator.

Note: This section assumes you are familiar with creating a project, adding threads, adding a stack to a thread and configuring a block within the stack.

To add the SCE Driver to an application, simply add it to a thread using the stacks selection sequence given in the following table. (The default name for the SCE HAL module is r_sce. This name can be changed in the associated Properties window.)

Table 9 SCE Driver Selection Sequence

Resource	ISDE Tab	Stacks Selection Sequence
g_sce_aes_0 AES Driver on r_sce_aes	Threads	New Stack> Driver> Crypto> AES Driver on r_sce_aes
g_sce_arc4_0 ARC4 Driver on r_sce_arc4	Threads	New Stack> Driver> Crypto> ARC4 Driver on r_sce_arc4
g_sce_dsa_0 DSA Driver on r_sce_dsa	Threads	New Stack> Driver> Crypto> DSA Driver on r_sce_dsa
g_sce_hash_0 HASH Driver on r_sce_hash	Threads	New Stack> Driver> Crypto> HASH Driver on r_sce_hash
g_sce_rsa_0 AES Driver on r_sce_aes	Threads	New Stack> Driver> Crypto> RSA Driver on r_sce_rsa
g_sce_tdes TDES Driver on r_sce_tdes	Threads	New Stack> Driver> Crypto> TDES Driver on r_sce_tdes
g_sce_trng TRNG Driver on r_sce_trng	Threads	New Stack> Driver> Crypto> TRNG Driver on r_sce_trng

When the Crypto Drivers on r_sce are added to the thread stack as shown in the following figure, the configurator automatically adds any needed lower-level modules. Any drivers that need additional configuration information will be box text highlighted in Red. Modules with a Gray band are individual modules that stand alone. Modules with a Blue band are shared or common and need only be added once and can be used by multiple stacks.

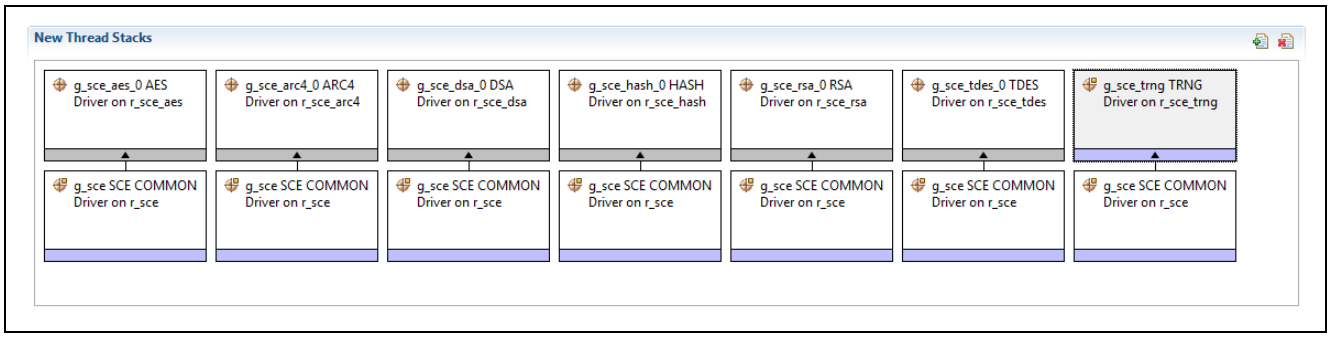

Figure 2 SCE HAL Module Stack (AES, ARC, DSA, HASH, RSA, TDES, and TRNG included)

5. Configuring the SCE HAL Module

The SCE HAL module must be configured for the desired operation. The SSP configuration window will automatically identify (by highlighting the block in red) any required configuration selections, such as interrupts or operating modes, which must be configured for lower-level modules for successful operation. Furthermore, only those properties that can be changed without causing conflicts are available for modification. Other properties are 'locked' and are not available for changes, and are identified with a lock icon for the 'locked' property in the Properties window in the ISDE. This approach simplifies the configuration process and makes it much less error prone than previous 'manual' approaches to configuration. The available configuration settings and defaults for all the user-accessible properties are given in the properties tab within the SSP configurator and are shown in the following tables for easy reference.

One of the properties most often identified as requiring a change is the interrupt priority; this configuration setting is available within the Properties window of the associated module. Simply select the indicated module and then view the Properties window; the interrupt settings are often toward the bottom of the properties list, so scroll down until they become available. Also note that the interrupt priorities listed in the Properties window in the ISDE will include an indication as to the validity of the setting based on the targeted MCU (CM4 or CM0+). This level of detail is not included in the following configuration properties tables, but is easily visible with the ISDE when configuring interrupt-priority levels.

Note: You may want to open your ISDE, create the module and explore the property settings in parallel with looking over the following configuration table settings. This will help orient you and can be a useful 'hands-on' approach to learning the ins and outs of developing with SSP.

Table 10 AES HAL Module Parameters and Settings

ISDE Property	Value	Description
Name	g_sce_aes_0	Module name
Key Length	User defined, default is 128 bits. Allowed values for S7G2 and S5D9 devices: 128, 192, or 256 bits. Allowed values for S3A7 and S124 devices: 128 or 256 bits	Key length used for encryption/decryption operations by this instance of the driver
Channel	User defined, default is CBC. Allowed values for S7G2 and S5D9 devices: ECB, CBC, CTR, GCM, XTS. Allowed values for S3A7 device: ECB, CBC, CTR, GCM. Allowed values for S124 device: ECB, CBC, CTR	Block cipher chaining mode used for encryption/decryption operations by this instance of the driver

Table 11 ARC4 HAL Module Parameters and Settings

ISDE Property	Value	Description
Name	g_sce_arc4_0	Module name
Key Length	Default 0	Key length in number of bytes
Key Name	g_arc4_0_key	Key name- must be defined as unit8_array type data in user code

Table 12 DSA HAL Module Parameters and Settings

Parameter	Value	Description
Name	g_sce_dsa_0	Module name
Key Length	User defined, default is (2048, 256) bits. Allowed values for S7G2 and S5D9 devices: (1024, 160), (2048, 224) or (2048, 256) bits Allowed values for S3A7 and S124 devices: Not available.	Key length used for signing/verification operations by this instance of the driver.

Table 13 HASH HAL Module Parameters and Settings

Parameter	Value	Description
Name	g_sce_hash_0	Module Name
Algorithm	User defined, default is SHA256. Allowed values for S7G2 and S5D9 devices: SHA1, or SHA256. Allowed values for S3A7 and S124 devices: Not available.	Algorithm used for computing the message digest/hash on the message data.

Table 14 RSA HAL Module Parameters and Settings

Parameter	Value	Description
Name	g_sce_rsa_0	Module Name
Key Length	User defined, default is 2048 bits. Allowed values for S7G2 and S5D9 devices: 1024 or 2048 bits. Allowed values for S3A7 and S124 devices: Not available.	Key length used for signing/verification/encryption/decryption operations by this instance of the driver

Table 15 TDES HAL Module Parameters and Settings

Parameter	Value	Description
Name	g_sce_tdes	Module Name
Chaining Mode	ECB, CBC, CTR	Chaining mode selection

Table 16 TRNG HAL Module Parameters and Settings

Parameter	Value	Description
Name	g_sce_trng	Module Name
Max. Attempts	User defined, default is 2	Sets the maximum number of attempts when a newly generated random number differs from the previously generated random number

6. Using the SCE HAL Module in an Application

The typical steps in using the SCE HAL module in an application are:

1. Use the `open` API to initialize the module.
2. Specify parameters using the associate API (AES addAdditionalAuthenticationData, for example).
3. Use the `open` API to start the function (AES open for example).
4. Encrypt data using the `encrypt` API.
5. Decrypt data using the `decrypt` API.
6. Close the SCE instance using the SCE `close` API.

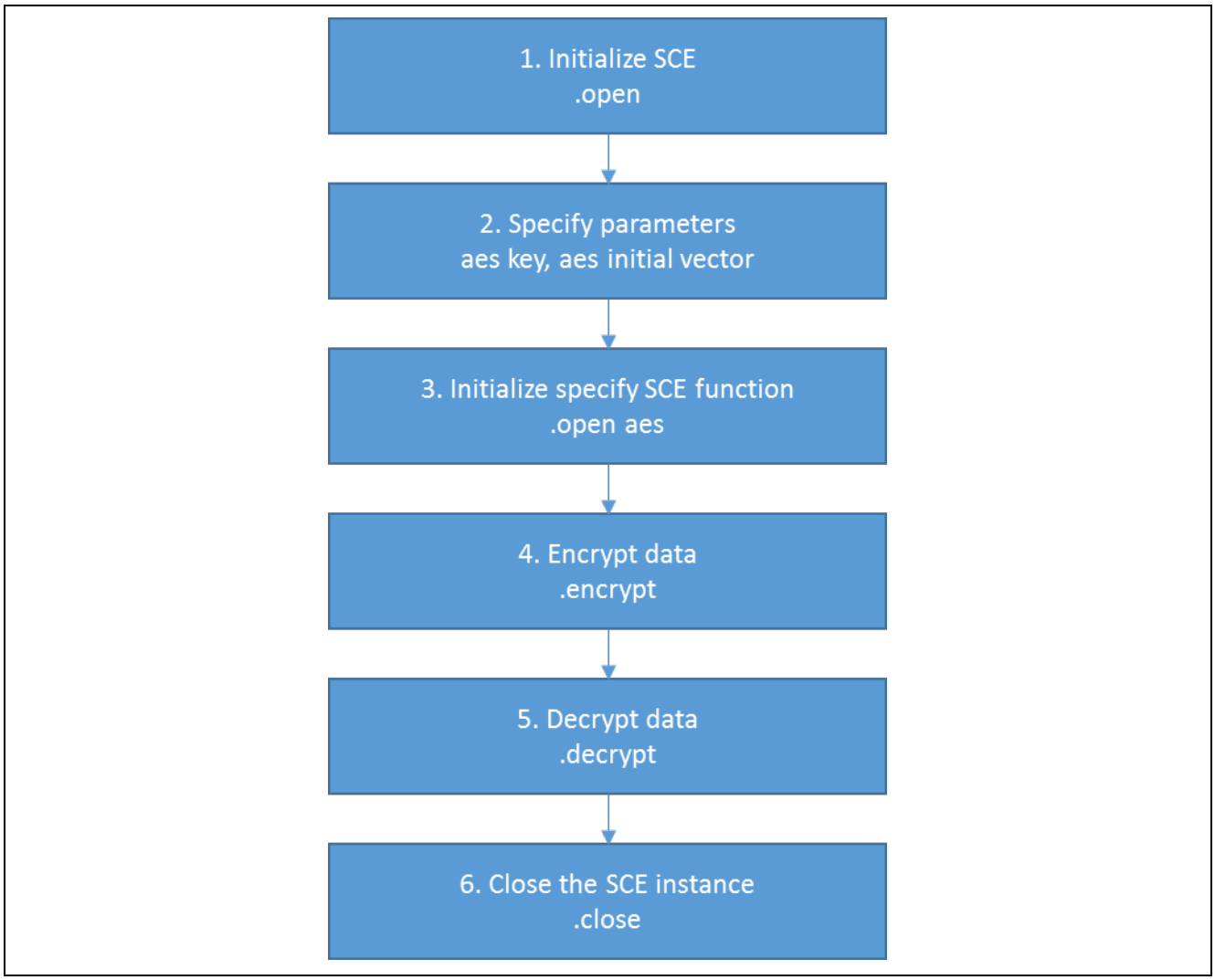

Figure 3 Flow Diagram of a Typical Cryptography Application

Specific use examples are provided below.

1. To use the SCE module.
   - Use the `open` API to initialize the SCE and the SCE HAL module (R_SCE) through the SCE common driver.
   - Configure the endianness (little endian or big endian) for the input/output data for all the HAL APIs. The big endian mode is configured by default. See HAL Module Operational notes for details on endianness configuration.
   - The `open` function cannot be called again until the module is closed.
2. To use the AES functions:
   - Initialize an AES interface instance with the `open` API.
   - Specify the configuration parameters for the instance to use the associated APIs. AES key sizes available are 128-bit, 192-bit, or 256-bit. Chaining modes supported are ECB, CBC, CTR, GCM, and XTS.
   - Encrypt data using the `encrypt` API.
   - Decrypt data using the `decrypt` API
   - Close the interface instance with `close` API.
3. To use the TDES functions:
   - Initialize the TDES interface instance with `open` API.

- Specify the configuration parameters for the instance to use the associated APIs. Select the TDES chaining mode ECB, CBC, or CTR.
- Encrypt data using the `encrypt` API.
- Decrypt data using the `decrypt` API
- Close the interface instance with the `close` API.

4. To use the ARC4 functions:
   - Initialize an ARC4 interface instance with `open` API.
   - Specify the configuration parameters for the instance to use the associated APIs. ARC4 key can be specified by length (anywhere from 64 bits to 2048 bits) and location.
   - The key to be used can be set through the `keySet` API.
   - To encrypt or decrypt data, use the `arc4Process` API.
   - Close the interface instance with `close` API.
5. To use the RSA functions:
   - Supported key lengths are 1024 bits and 2048 bits.
   - Select the RSA interface instance based on the desired key length and initialize the selected RSA interface instance with open API.
   - To encrypt data using an RSA public key use the `encrypt` API.
   - To decrypt data using an RSA private key use the `decrypt` API.
   - To decrypt data using anRSA private key which is in the CRT format, use the `decryptCrt` API.
   - To generate signature for a given padded hash using an RSA private key which is in the standard forma,t use the `sign` API.
   - To generate signature for a given padded hash using an RSA private key which is in the CRT format, use the `signCrt` API.
   - To verify the signature for a given padded hash using an RSA public key which is in the standard format, use the `verify` API.
   - Close the interface instance with `close` API.
6. To use the DSA functions:
   - Supported key lengths are (1024, 160) bits, (2048, 224) bits and (2048, 256) bits
   - Select the DSA interface instance based on the desired key length and initialize the selected DSA interface instance with the `open` API.
   - To generate signature using the DSA private key, use the `hashSign` API.
   - To verify signature using the DSA public key, use the `hashVerify` API.
   - Close the interface instance with the `close` API.
7. To use the HASH algorithms:
   - SHA1 and SHA256 hash methods are supported.
   - Select the HASH interface instance based on the desired hash method and initialize the selected HASH interface instance with the `open` API.
   - To compute the message digest, use the `hashUpdate` API.
   - Close the interface instance with the `close` API.
8. To use the True Random Number Generator functions:
   - Initialize a TRNG interface instance with the `open` API.
   - Generate random number with the `read` API.

- Close the interface instance with the `close` API.

9. Close the SCE and the SCE HAL module using the `close` API.

7. The SCE HAL Module Application Project

The application project associated with this module guide demonstrates the aforementioned steps in a full design. The project can be found using the link provided in the References section at the end of this document. You may want to import and open the application project within the ISDE and view the configuration settings for the SCE HAL modules. You can also read over the code (in `hal_entry.c`, `sce_aes_mg`, `sce_dsa_mg.c`, `sce_hash_mg.c`, `sce_rsa_encryption_mg.c`, `sce_rsa_signature_mg.c`, and `sce_trng_mg.c`) which are used to illustrate the SCE APIs in a complete design.

The application project demonstrates the use of the SCE APIs. The application project demonstrates data encryption and decryption, digital signature and verification, data-hash calculation, and random number generation. The following table identifies the target versions for the associated software and hardware used by the application project:

Table 17 Software and Hardware Resources Used by the Application Project

Resource	Revision	Description
e² studio	5.3.1 or later	Integrated Solution Development Environment
SSP	1.2.0 or later	Synergy Software Platform
IAR EW for Renesas Synergy	7.71.2 or later	IAR Embedded Workbench for Renesas Synergy
SSC	5.3.1 or later	Synergy Standalone Configurator
SK-S7G2	v3.0 to v3.1	Starter Kit

A simple flow diagram of the application project is given in the following figure:

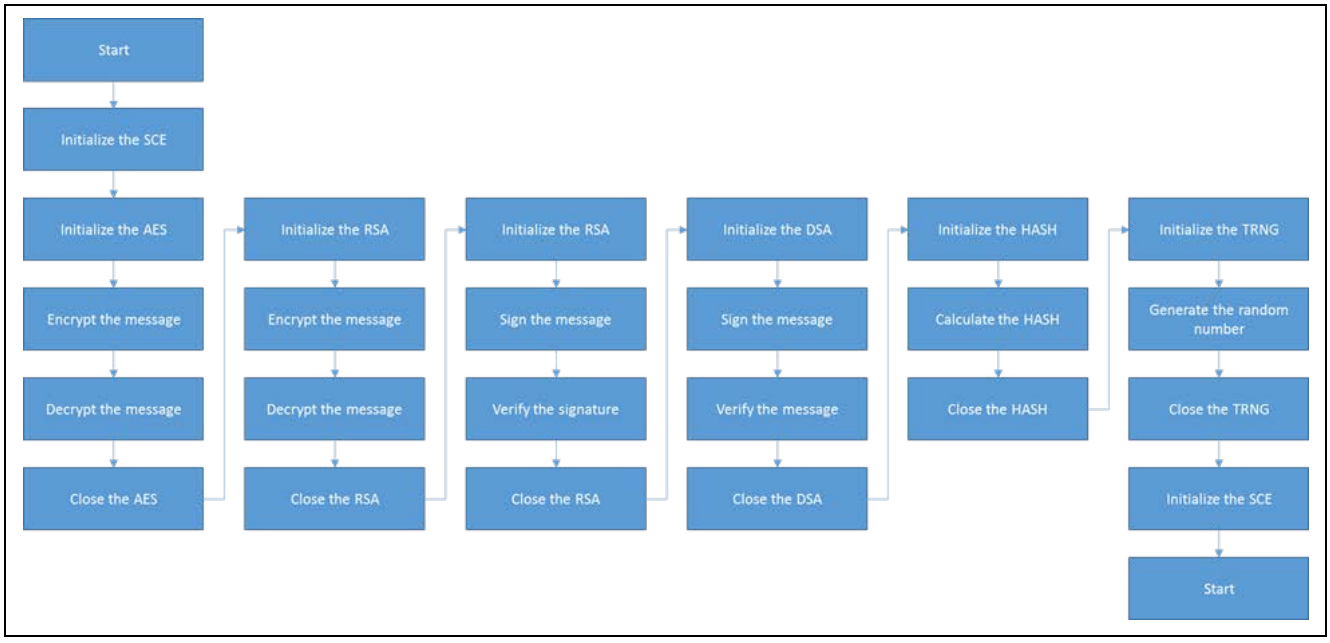

Figure 4 SCE HAL Module Application Project Flow Diagram

The complete application project can be found using the link provided in the References section at the end of this document. The files `hal_entry.c, sce_aes_mg, sce_dsa_mg.c, sce_hash_mg.c, sce_rsa_encruption_mg.c, sce_rsa_signature_mg.c`, and `sce_trng_mg.c` are located in the project once it has been imported into the ISDE. You can open these files within the ISDE and follow along with the description provided to help identify key uses of APIs.

The first section of the `hal_entry.c` includes the auto-generated header file which references the SCE instance structures. Within the function `hal_entry.c`, some sample data that can be operated on is created. Once created in the correct format, the SCE HAL module is opened and six example crypto-functions are called: aes_example, rsa_encryption_example, rsa_signature_example, dsa_example, hash_example, and trng_example.

The aes_example is in the file `sce_aes_mg.c`. This function first declares the AES keys and AES initial vector required for the AES functions. The SCE AES driver is opened and then the sample data is encrypted by calling the API `g_sce_aes_0.p_api->encrypt()` with the required parameters. The result of the encryption process is stored in the array `encrypted_message`. Setting a breakpoint before and after this API call can show the result of the encryption process.

The encrypted message is then decrypted be calling the API `g_sce_aes_0.p_api->decrypt()` with the required parameters. The result of the decryption process is stored in the array `decrypted_message`. Setting a breakpoint before and after this API call can show the result of the decryption process.

The final section of the aes_example is a comparison between the starting message and the decrypted message. If these messages match, then the application will continue. If not, the application will wait in a `while(1)` loop.

After successful completion of the sce_aes example, the rsa_encryption_example in the file `sce_rsa_encryption_mg.c` is executed. This function first declares the exponents and modulus that make up the RSA public and private keys required for the RSA functions. The SCE RSA HAL module is opened and then the sample data is encrypted by calling the API `g_sce_rsa_0.p_api->encrypt()` with the required parameters. The result of the encryption process is stored in the array `encrypted_message`. Setting a breakpoint before and after this API call can show the result of the encryption process.

The encrypted message is then decrypted be calling the api `g_sce_rsa_0.p_api->decrypt()`with the required parameters. The result of the decryption process is stored in the array `decrypted_message`. Setting a breakpoint before and after this API call can show the result of the decryption process.

The final section of the rsa_example is a comparison between the starting message and the decrypted message. If these messages match, the application will continue. If not, the application will wait in a `while(1)` loop.

After successful completion of the rsa_encryption_example, the rsa_signature_example in the file `sce_rsa_signature_mg.c` is executed. This function first declares the exponents and modulus that make up the RSA public and private keys required for the RSA sign and verify functions. The SCE RSA HAL module is opened and the sample data is signed by calling the API `g_sce_rsa_0.p_api->sign()` with the required parameters. The result of the sign process is stored in the array `signature`. The message is then verified with the previously created signature by calling the API `g_sce_rsa_0.p_api->verify()` with the required parameters. If the verify process is successful, the application will continue. If not, the application will wait in a `while(1)` loop.

After successful completion of the rsa_signature_example, the dsa_example in the file sce_dsa_mg.c is executed. This function first declares the keys and domain variables that make up the DSA public and private keys required for the DSA sign and verify functions. The SCE DSA HAL module is opened and the sample data is signed by calling the API `g_sce_dsa_0.p_api->hashSign()` with the required parameters. The result of the sign process is stored in the array `signed_message`. The message is then verified with the previously created signature by calling the API `g_sce_dsa_0.p_api->verify()` with the required parameters. If the verify process is successful, the application will continue. If not, the application will wait in a `while(1)` loop.

After successful completion of the dsa_example, the hash_example in the file `sce_hash_mg.c` is executed. This function first declares the storage area required for the hash functions. The SCE HASH driver is opened and then a hash of the sample data is generated by calling the API `g_sce_hash_0.p_api->hashUpdate()` with the required parameters. If the `hashUpdate` is successful, the application will continue. If not, the application will wait in a `while(1)` loop.

After successful completion of the hash_example, the trng_example in the file `sce_trng_mg.c` is executed. This function first declares the size and the storage area required for the random number-generation function. The SCE TRNG HAL module is opened and the specified number of random numbers are generated by calling the API `g_sce_trng_0.p_api->read()` with the required parameters. If the read function is successful, the application will continue. If not, the application will wait in a `while(1)` loop.

The application has now completed the SCE functions and will wait in a while(1) loop.

The key elements in constructing a simple SCE HAL module application are selecting and configuring a specified stack. In this application project, five stacks are used: the AES HAL module on r_sce_aes, the HASH HAL module on r_sce_hash, the RSA HAL module on r_sce_rsa, the DSA HAL module on r_sce_dsa, and the TRNG Driver on r_sce_trng.

The following table shows the configuration of the stacks used in the application example:

Table 18 SCE HAL Layer Interface API Summary

Driver	Module	Property	Value
AES	g_sce_aes_0 AES Driver on r_sce_aes	Name	g_sce_aes_0
		Key Length	128
		Chaining Mode	ECB
RSA	g_sce_rsa_0 RSA Driver on r_sce_rsa	Name	g_sce_rsa_0
		Key Length	1024
DSA	g_sce_dsa_0 DSA Driver on r_sce_dsa	Name	g_sce_dsa_0
		Key Length	(1024, 160)
HASH	g_sce_hash_0 HASH Driver on r_sce_hash	Name	g_sce_hash_0
		Key Length	SHA256
TRNG	g_sce_trng TRNG Driver on r_sce_trng	Name	g_sce_trng_0
		Max. Attempts	2

8. Customizing the SCE HAL Module for a Target Application

Some configuration settings will normally be changed by the developer from those shown in the application project. For example, the AES key-length and chaining mode may be changed.

Data encryption, decryption, signing, and verifying requires a set of keys depending on the chosen method and key length. The keys used in this application project was done using the OpenSSL toolkit, which can be found here: http://gnuwin32.sourceforge.net/packages/openssl.htm.

The following pages detail how the keys were generated:

Download the binaries zip file and extract it.

Open the command window and navigate to the openssl folder and then to bin folder.

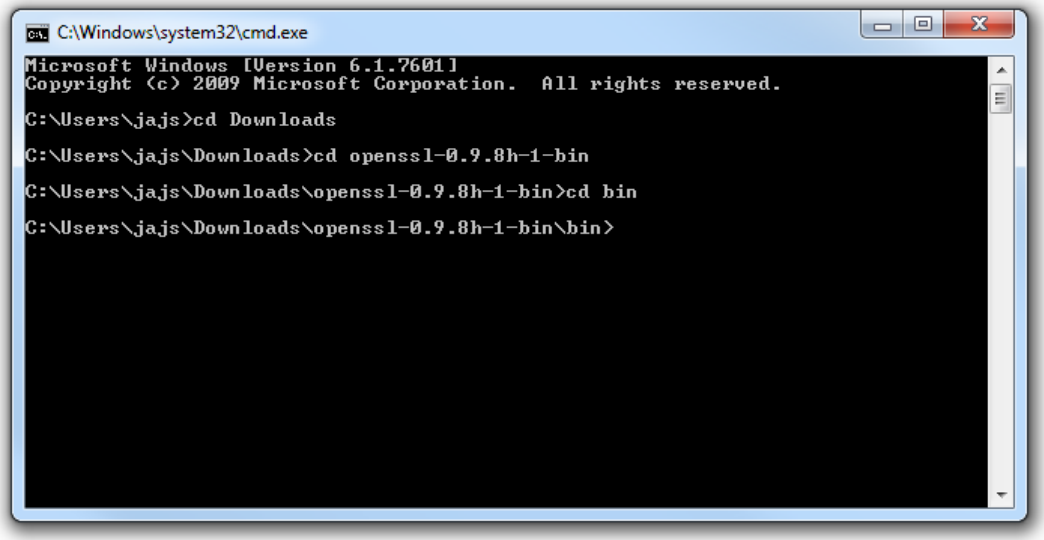

Figure 5 Navigation to openssl binary folder

8.1 AES key generation

Generation of the AES key and initial vector can be generated using the command `openssl aes-128-ecb -P` (for 128-bit key length and ECB chaining mode). The user will be asked to enter the password and verify it. It is used to calculate the hash to generate the password. It can be any password. The generated key and IV should be used to encrypt/decrypt the data using the AES algorithm. Please remember about proper formatting for 32-bit length integer array defined in a hexadecimal form.

Figure 6 Key and initial vector for AES

8.2 RSA key generation

To generate the RSA key with OpenSSL, run the command `openssl genrsa -out rsa-openssl.pem 1024` (for a 1024-bit key length) and then `openssl rsa -in rsa-openssl.pem -pubout -outform DER -text`. The program will present the data on the screen in DER format (1-byte hexadecimal separated with colon) which must be converted to C-based notation of 32-bit hexadecimal number representation. The example of using these data to combine the private and public keys is presented in the previous section. If some number begins with byte `00:` this byte must be ignored.

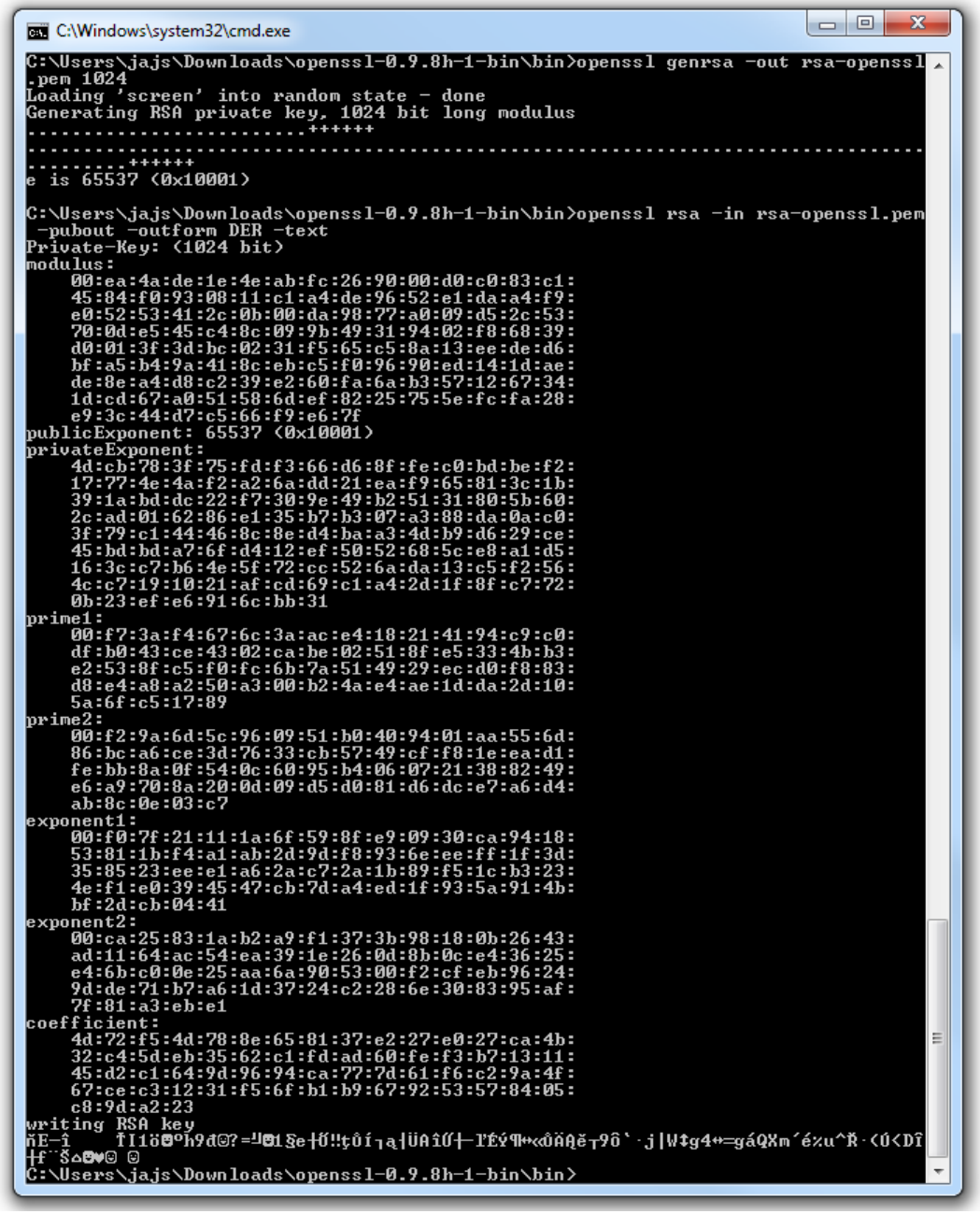

Figure 7 Key and initial vector for RSA

8.3 DSA key generation

Generation of the key and parameters for DSA signing and verification can be done with a set of commands: `openssl dsaparam -out dsa-param-openssl.pem 1024` then openssl gendsa -out dsa-openssl.pem dsa-param-openssl.pem and finally `openssl dsa -in dsa-openssl.pem -pubout -outform DER -text`. The program will present the data on the screen in DER format which must be converted to C-based notation of 32-bit hexadecimal number representation. The example of using these data to combine the private and public keys is presented in the previous section. If a number begins with byte `00:` this byte must be ignored.

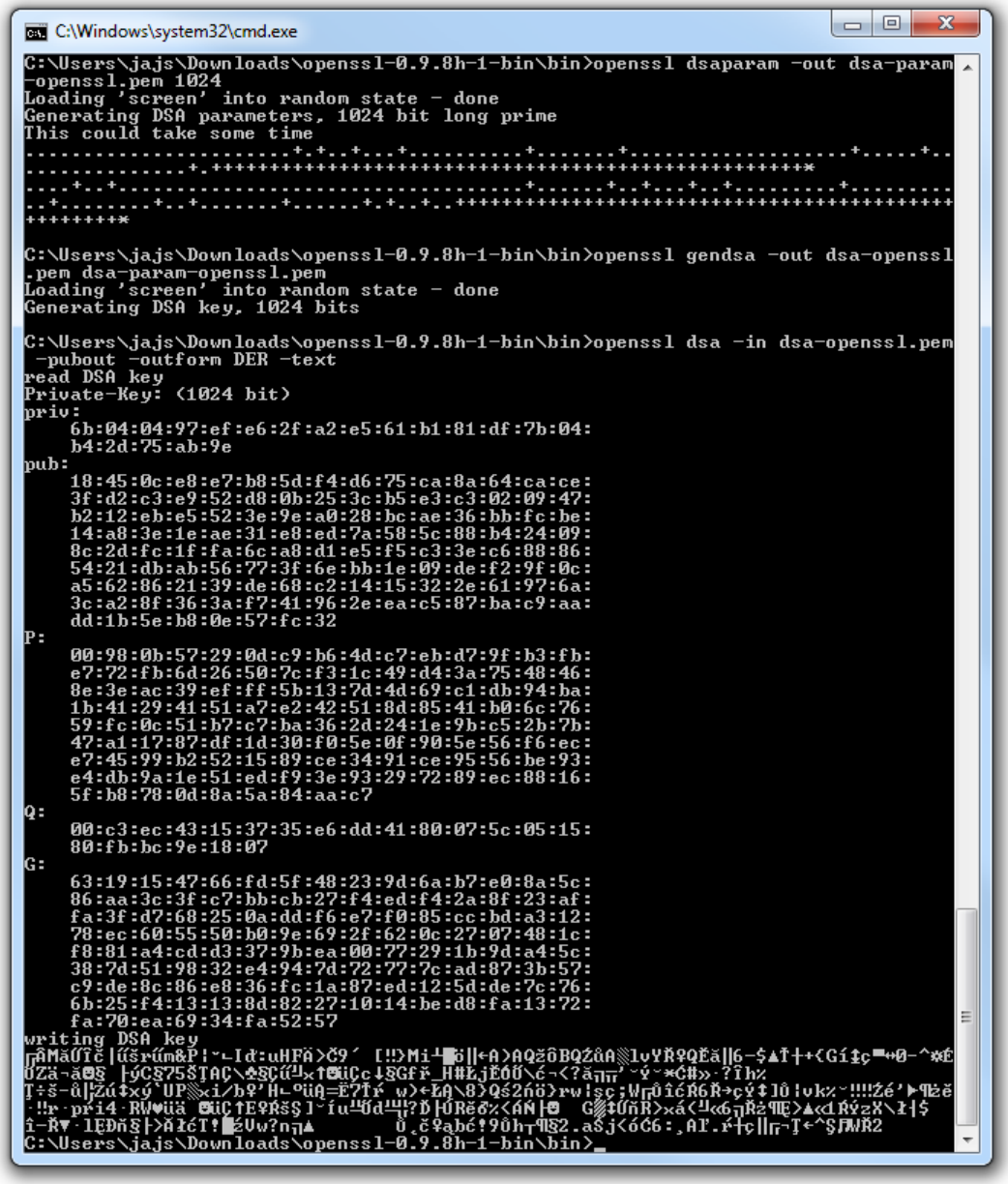

Figure 8 Key and parameters for DSA

9. Running the SCE HAL Module Application Project

To run the SCE HAL module application project and to see it executed on a target kit, you can simply import it into your ISDE, compile, and run debug.

To implement the SCE application in a new project, follow the steps for defining, configuring, auto-generating files, adding code, compiling and debugging on the target kit. Following these steps is a hands-on approach that can help make the development process with SSP more practical, while just reading over this guide will tend to be more theoretical.

Note: The following steps are described in sufficient detail for someone experienced with the basic flow through the Synergy development process. If these steps are not familiar, refer to the first few chapters of the SSP User's Manual for a description of how to accomplish these steps.

To create and run the SCE HAL module application project, simply follow these steps:

1. Create a new Renesas Synergy project for the SK-S7G2 board (S7G2-BSP) called `sce_hal`.
2. Select the S7G2-SK BSP project template and create the project.
3. Open `Configuration.xml` from the generated project and select the **Threads** tab.

4. Add five SCE stacks in the HAL/Common thread from **New Stack > Driver > Crypto** and configure their parameters**.**
5. Click on the **Generate Project Content** button.
6. Add the code from the supplied project file `hal_entry.c` or copy the file over the generated `hal_entry.c` file.
7. Copy the files `sce_aes_mg.c`, `sce_dsa_mg.c`, `sce_functions_mg.h`, `sce_hash_mg.c`, `sce_rsa_encryption_mg.c`, `sce_rsa_signature_mg.c` and `sce_trng_mg.c` to the project **src** folder in the project directory.
8. Connect to the host PC via a micro USB cable to J19 on SK-S7G2.
9. Start to debug the application.

9.1 AES and RSA encryption result presentation

After the data is decrypted, the decrypted message and original message are compared. If they differ, the program will start the infinite loop. To inspect the decrypted message, place a breakpoint after decrypting the data in `aes.c` or `rsa.c` file and watch the variable `decrypted_message`.

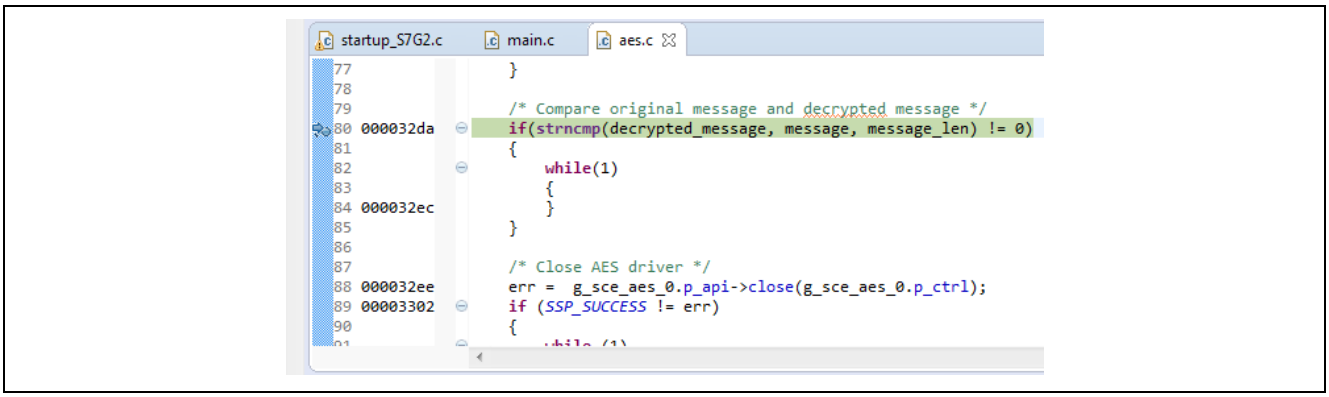

Figure 9 Breakpoint placed at string compare in “aes.c” file

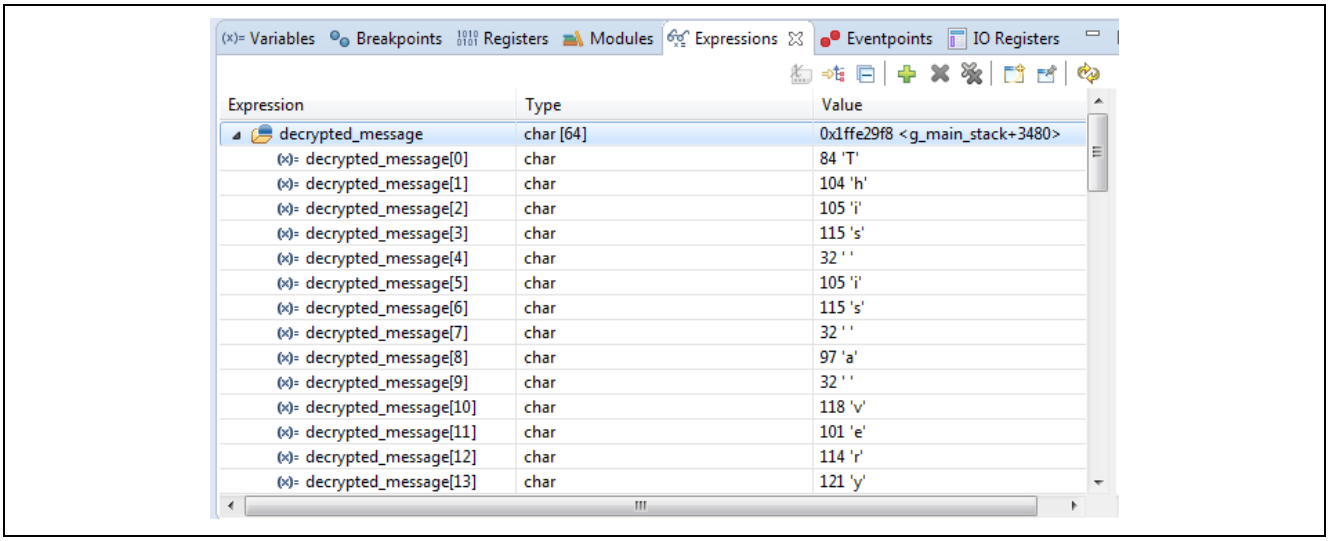

Figure 10 Result of data decryption

As seen in the preceding figure, the `decrypted_message` variable contains the same text as the original message. Otherwise the program would start the infinite loop.

9.2 RSA and DSA signature result presentation

To present the result of data signature and verification, place the breakpoint after verifying the signature and inspect `err`. The value `SSP_SUCCESS` indicates the positive verification. Otherwise, the result would be `SSP_ERR_INVALID_MODE` and the program would start the infinite loop.

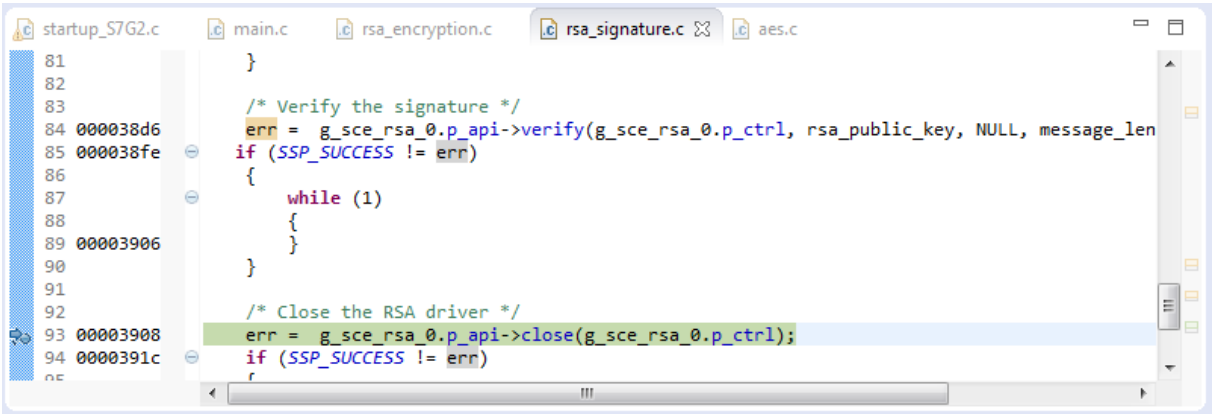

Figure 11 Breakpoint placed after signature verification in “rsa_signature.c” file

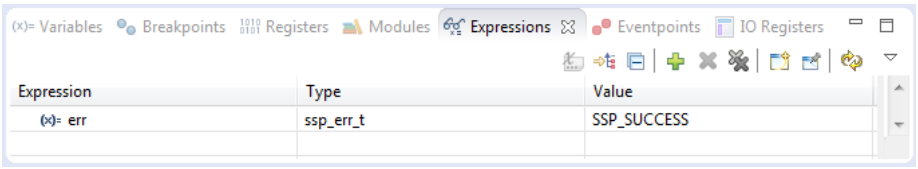

Figure 12 Result of the signature verification

9.3 HASH and TRNG result presentation

To present the result of a hash function or the random-number generation, place a breakpoint after calculating or generating it and watch the `hash` or `random_number` variable.

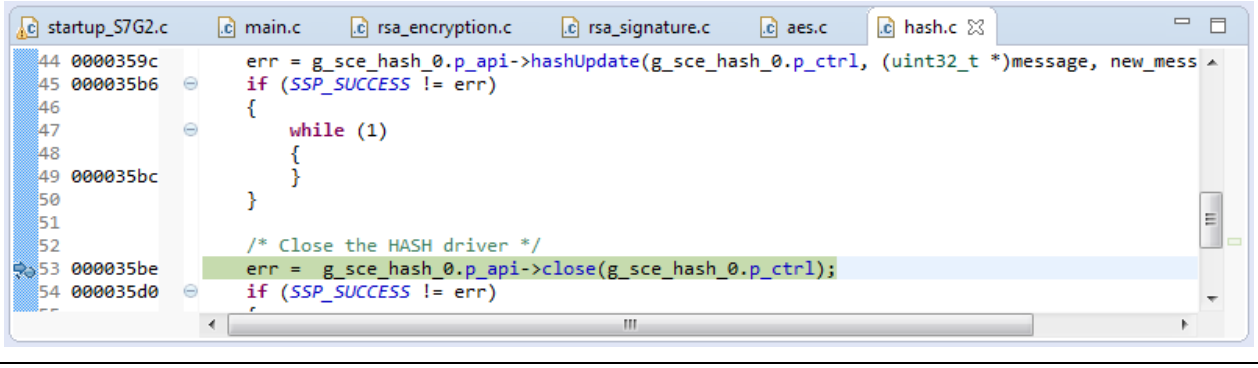

Figure 13 Breakpoint after hash calculation

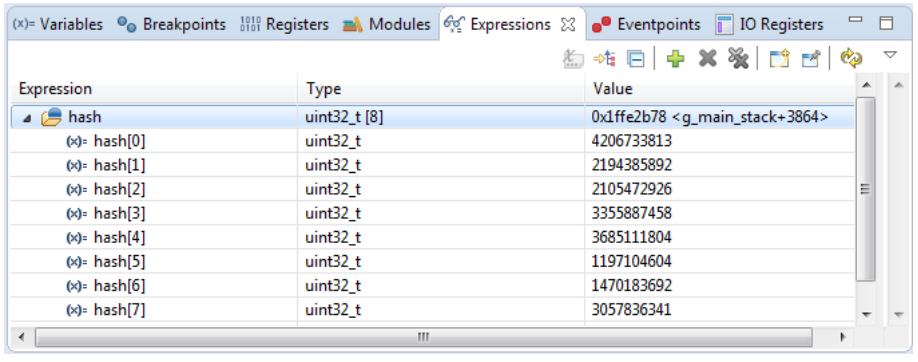

Figure 14 Hash value

10. SCE HAL Module Conclusion

This module guide has provided all the background information needed to select, add, configure, and use the module in an example project. Many of these steps were time consuming and error-prone activities in previous generations of

embedded systems. The Renesas Synergy Platform makes these steps much less time consuming and removes the common errors like conflicting configuration settings or incorrect selection of lower-level modules. The use of high-level APIs (as demonstrated in the application project) illustrates additional development-time savings by allowing work to begin at a high level and avoiding the time required in older development environments to use, or, in some cases, create, lower-level drivers.

11. SCE HAL Module Next Steps

After you have mastered a simple SCE module project, you may want to review a more complex example. Other application projects and application notes that demonstrate SCE HAL use can be found as described in the References section at the end of this document.

12. SCE HAL Module Reference Information

SSP User Manual: Available in html format in the SSP distribution package and as a pdf from the Synergy Gallery.

Links to all the most up-to-date r_sce module reference materials and resources are available on the Synergy Knowledge Base: https://en-us.knowledgebase.renesas.com/English_Content/Renesas_Synergy%E2%84%A2_Platform/Renesas_Synergy_Knowledge_Base/R_SEC_Module_Guide_Resources.

Website and Support

Support: https://synergygallery.renesas.com/support

Technical Contact Details:

- America: https://www.renesas.com/en-us/support/contact.html
- Europe: https://www.renesas.com/en-eu/support/contact.html
- Japan: https://www.renesas.com/ja-jp/support/contact.html

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Mar 16, 2017	-	Initial Release
1.01	Sep 14, 2017	16	Updated Hardware and Software Resources Table

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 disputes 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, drawing, chart, program, algorithm, 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 not alter, modify, copy, or otherwise misappropriate 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, copy or otherwise misappropriation of Renesas Electronics products.
5. 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; and 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.

   Renesas Electronics products are neither intended nor 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 damages (space and 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 third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the 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 radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond such specified ranges.
7. 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. Further, Renesas Electronics products are not subject to radiation resistance design. Please ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of 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 by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use 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.
9. 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 not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.3.0-1 November 2016)

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 6.0