User Manual

DA16200 DA16600 Security Tool

UM-WI-015

Abstract

This User Manual provides instructions on how to implement and use the Security Tool of DA16200 and DA16600.

Contents

Figures

Tables

1 Terms and Definitions

AHB	AMBA High-performance Bus
CM	Chip Master
CMPU	Chip Master Process Unit
DCU	Debug Control Unit
DM	Device Master
DMPU	Device Master Process Unit
HUK	Hardware Unique Key
LCS	Life Cycle State
OEM	Original Equipment Manufacturer
OTP	One Time Programmable
PoR	Power on Reset
RMA	Return Merchandise Authorization
RoT	Root of Trust
SB	Secure Boot
SD	Secure Debug

2 References

[1] DA16200, Datasheet, Renesas Electronics

[2] UM-WI-056, DA16200 DA16600 FreeRTOS Getting Started Guide , User Manual, Renesas Electronics

[3] UM-WI-003, DA16200 DA16600 Host Interface and AT Command, User Manual, Renesas Electronics

[4] UM-WI-046, DA16200 DA16600 FreeRTOS SDK Programmer Guide, User Manual, Renesas Electronics

3 DA16200/DA16600 Security

3.1 Security Engine

DA16200/DA16600 uses the ARM CryptoCell-312 as its security engine that provides security services for the platform such as Secure Boot and Key Management with acceleration for cryptographic operations. Many of the security services are implemented in the ROM code, for example, the Secure Boot process. The cryptography and management service are integrated into the Operating System (OS) and are used with mbedTLS for the TLS and SSL protocols.

3.2 H/W Components

Figure 1 shows a block diagram of the DA16200/DA16600 security engine.

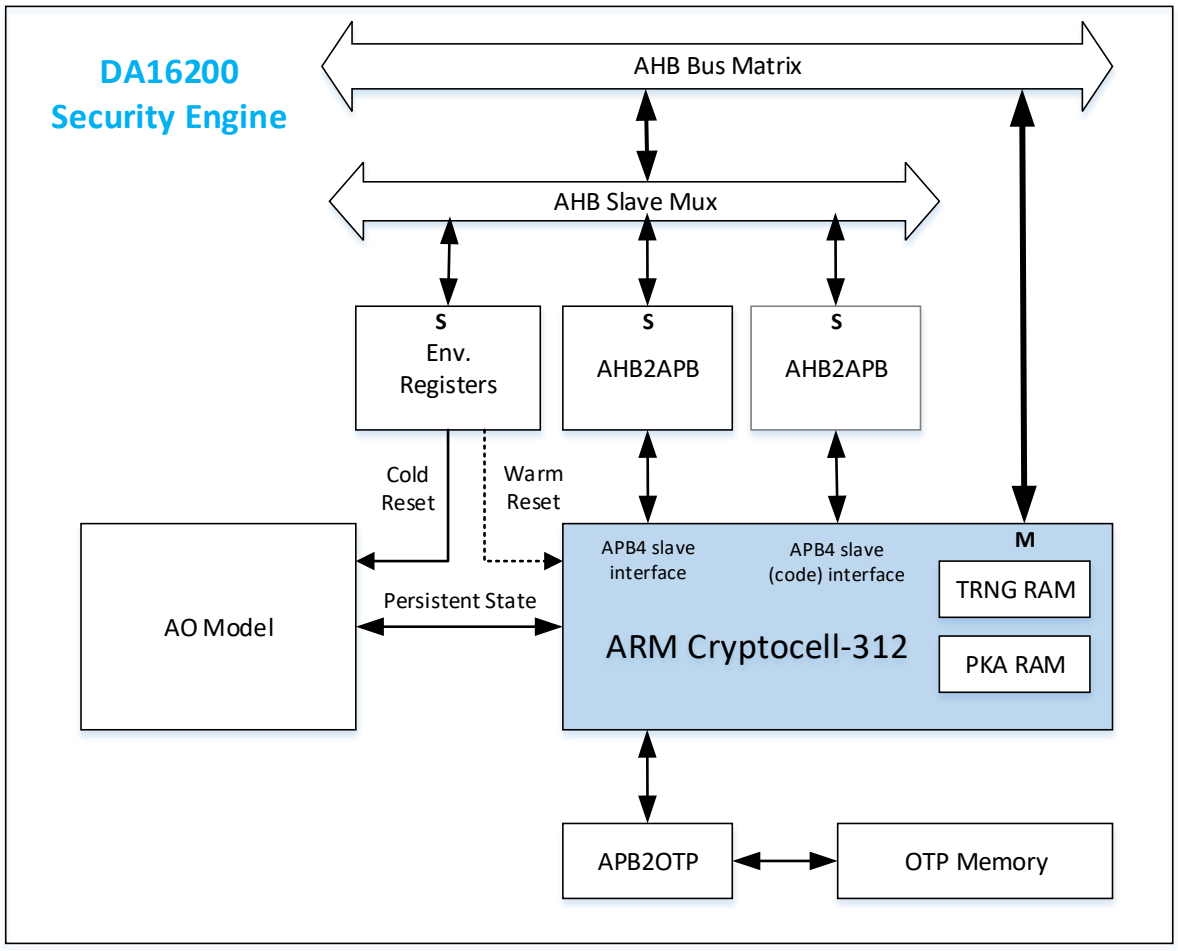

Figure 1: Block Diagram of DA16200/DA16600 Security Engine

The host processor is able to access CC312's SRAM and registers, as well as the One Time Programmable (OTP) memory in DA16200/DA16600. The CC312 security engine can initialize transactions with the system memory or other DMA slaves through the AMBA High-performance Bus (AHB) Master (Marked by 'M' in Figure 1).

The CC312 security engine is connected to the external OTP memory via the Advanced Peripheral Bus (APB4) Master interface, and OTP memory holds the device root key (HUK) and life cycle state

(LCS).The specific area of OTP memory that is controlled by the CC312 security engine is only accessible by the CC312 and thus acts as the Root-of-Trust for DA16200 and DA16600.

The Always On (AO) module must survive a power down of the CC312 to keep the critical state of the embedded system. The AO module includes the following components:

- Security Life cycle States
- Debug Control Unit (DCU) and DCU Lock registers
- Lock-Bits Register

3.3 S/W Architecture

Secure Boot services run from the ROM in DA16200/DA16600. The crypto services, which are accelerated by CC312 H/W, can be used with mbedTLS APIs. See the user manual, Ref. [4] for the mbedTLS APIs. Figure 2 shows the security software architecture in DA16200/DA16600.

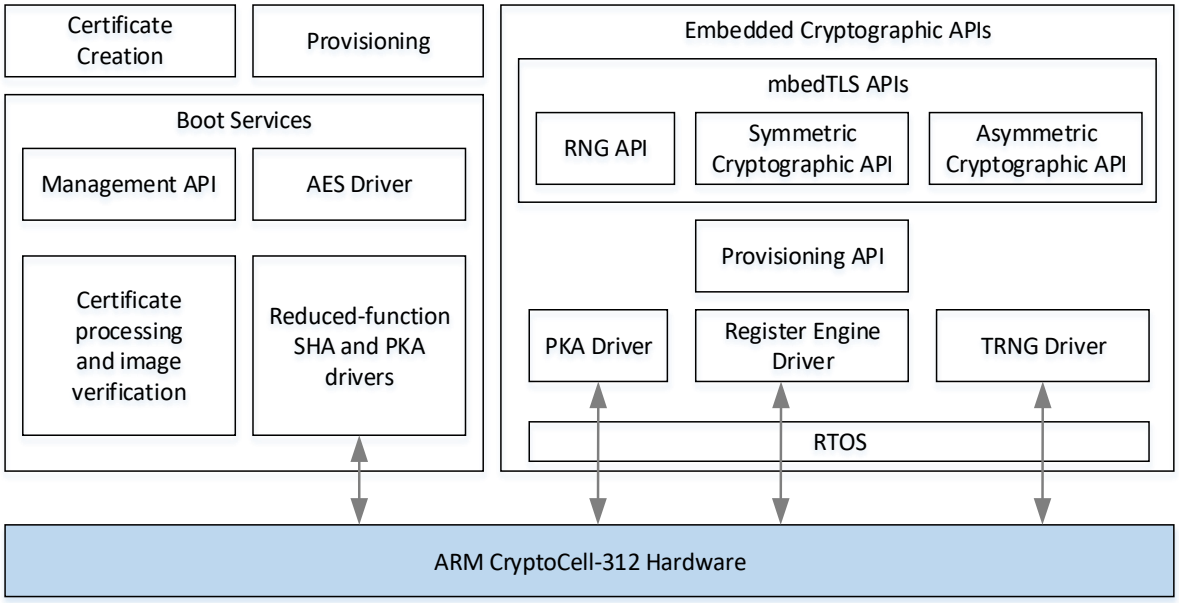

Figure 2: DA16200/DA16600 Security S/W Architecture

DA16200/DA16600 supports a FreeRTOS based SDK. See the SDK programmer guide Ref. [4] for Serial Flash Memory Map of FreeRTOS images.

- Images in FreeRTOS SDK
  - UEboot image (XXUEBOOTXX.img) built from SDK contains a bootloader (UEboot) binary
  - RTOS image (XXRTOSXX.img) built from SDK contains the RTOS binaries

4 Security Features

4.1 Security Services

4.1.1 Secure Boot

The DA16200/DA16600 provides a secure boot function that allows trusted images signed with a key matching the registration information in the system during the boot process to ensure the system's platform integrity. In the production step, it is necessary to register the key information for authentication in the OTP memory, which is protected by CC312.

4.1.2 Secure Debug

DA16200/DA16600 supports a Secure Debug function that provides hardware protection of the debug port to prevent an external security attack. When a developer needs to enable this port for system debugging, Secure Debug uses the authenticated key with the signed debug certificate to remove the hardware protection, to allow debugging tasks.

4.1.3 Secure Asset

Secure Asset is a cryptographic service provided to protect data stored in external storage (Serial Flash memory). Data can be encrypted or decrypted with the provisioning key stored in the chip. Production-Line Provisioning is used to protect the data used in the mass production process, and Asset Provisioning is used to protect the data used during system operation.

4.2 Secret Keys

This chapter describes the required security keys in DA16200/DA16600. For the security feature in DA16200/DA16600, several security keys are required and should be stored in OTP memory before production. All secret keys are burned with the Security Tool. All hardware keys are accessed only by CryptoCell-312 and cannot be read by the CPU depending on the security LCS.

4.2.1 HUK

Hardware Unique Key (HUK) is called device key and a secret value that is burned into OTP memory, and is read by H/W as part of the secure boot sequence and is no longer accessible for reading. HUK can only be used by the AES engine, and only for the derivation of other keys. It must be unique per device. For this uniqueness, it is generated as the seed value derived from TRNG in CC312. The SoC ID is derived from this key. A SoC ID is required in Secure Debug, which will be described further on. A SoC ID is only valid in the Secure LCS state. HUK will be generated by the Security Tool.

4.2.2 Platform Key

The Platform key (Krtl) is placed in DA16200/DA16600 and used for provisioning during the production life cycle (CM and DM LCS). The Platform key will be provided by Renesas Electronics when requested. It is a 128-bit AES class key and a random 128-bit value. A key derived from this key is used to encrypt the provisioning assets such as Chip Master keys and Device Master keys, which are described in the following section. This key is only for use in CM (Chip Master) and DM (Device Master) LCS and is locked by H/W in all other LCS. Krtl should not be exposed to others for any reason. Our Security Tool uses this key in Secure Production and will remove the key after use.

4.2.3 Chip Master Key

Chip Master (CM) keys are burned in OTP memory at production time and used as a back-up key for DM keys. The CM keys are generated in the Security Tool. There are two types of CM keys:

- CM Provisioning Key (Kpicv): The Kpicv is a 128-bit AES key used for an asset provisioning flow
- CM Encryption Key (Kceicv): The Kceicv is a 128-bit AES key used to encrypt or decrypt software images as part of the Secure Boot process.

4.2.4 Device Master Key

Device Master (DM) keys are burned in OTP memory at production time. They are generated in the Security Tool. There are two types of DM keys:

- DM Provisioning Key (Kcp): This is a 128-bit AES key that is used for asset provisioning
- DM Encryption Key (Kce): This 128-bit AES key is used to encrypt or decrypt S/W images as part of the Secure Boot process.

4.2.5 RoT

The Root-of-Trust (RoT) is a hash of the public key. Every public key has a corresponding private key that must be preserved and not exposed for security reasons. These public and private key pairs are generated in the Security Tool.

There are two RoT keys: Hbk0 and Hbk1. Hbk0 is a hash of the CM public key generated by the Security Tool and is a back-up RoT for Hbk1 (a hash of the DM public key), which is normally used for Secure Boot and Secure Debug. Both Hbk0 and Hbk1 should be burned to the OTP memory as RoT. Hbk0 and Hbk1 are used in order to validate the authentication of an image with certificate data.

Here are the summary of Hbk0 and Hbk1.

- Hbk0

  A 128-bit truncated SHA-256 digest of a CM public key. Used as a back-up key for Hbk1, mainly used for Secure Boot and Secure Debug
- Hbk1

  A 128-bit truncated SHA-256 digest of a DM public key. Used as a main RoT key

4.3 OTP Memory

OTP memory is used to store keys and configure data. DA16200/DA16600 has 2 KB of OTP memory. Mandatory configuration data must be burned at the offsets given in Table 1.

Table 1: Configuration Data and Key in OTP Memory

32-bit Word (Note 1)	Description	Read	Write
0x00-0x07	HUK	Readable in CM LCS	Writable in CM or RMA LCS
0x0B	Kpicv	Readable in CM LCS	Writable in CM or RMA LCS
0x0C-0x0F	Kceicv	Readable in CM LCS	Writable in CM or RMA LCS
0x10	CM programmed flags.	See the following table	Writable in CM or RMA LCS
0x11-0x18	RoT pubkey If split into CM and DM keys: CM key(Hbk0): 0x11-0x14 DM key(Hbk1): 0x15-0x18	Readable in all LCS	Writable in CM or DM LCS
0x19-0x1C	Kcp	Readable in CM LCS or DM LCS	Writable in DM or RMA LCS
0x1D-0x20	Kce	Readable in CM or DM LCS	Writable in DM or RMA LCS
0x21	DM programmed flags	See the following table	Writable in all LCS
0x27	General purpose configuration flags		Writable in CM LCS
0x28-0x2B	DCU 128bits lock mask that allows S/W to lock the required debug bits	Readable in all LCS	Writable in CM or DM LCS
0x40-0x1FE	Code and data sections that a developer may use	Readable in all LCS	

Note 1 The word area from 0x00 ~ 0x2B is not accessible by the CPU and is accessible only by the H/W Security engine in DA16200/DA16600.

The word area from 0x00 ~ 0x2B should be burned into OTP memory at production time. For this purpose, special binary images called CMPU and DMPU are required. CMPU is a binary image containing the HUK, Hbk0, and CM keys. DMPU is a binary image containing the Hbk1 and DM keys. The CMPU and DMPU binary images are generated by the Security Tool during the Secure Production process. See Section 5.

Table 2 shows the CM programmed flags that are located at address 0x10 in the OTP memory.

Table 2: CM-Programmed Flags

Bits	Usage	Read Access	Write Access
[7:0]	Number of zero bits in HUK.	Readable only in CM LCS; masked for reading in any other LCS	Writeable in CM LCS and RMA LCS.
[14:8]	Number of zero bits in Kpicv (128-bit).	Readable only in CM LCS.	Writeable in CM LCS and RMA LCS.
[15]	Kpicv "not in use" bit. If Kpicv is not in use, this bit	Readable in all security life-cycle states.	Writeable in CM LCS and RMA LCS.

	is set by the IFT.		
[22:16]	Number of zero bits in Kceicv.	Readable only in CM LCS.	Writeable in CM LCS and RMA LCS.
[23]	Kceicv "not in use" bit. If Kceicv is not in use, this bit should be set by the IFT.	Readable in all security life-cycle states.	Writeable in CM LCS and RMA LCS.
[30:24]	Number of zero bits in Hbk0	Readable in all security life-cycle states.	Writeable in CM LCS and RMA LCS.
[31]	Hbk0 "not in use" bit. If Hbk0 is not in use, this bit should be set by the IFT.	Readable in all security life-cycle states.	Writeable in CM LCS and RMA LCS.

Table 3 shows the DM programmed flags that are located at address 0x21 in OTP memory.

Table 3: DM-Programmed Flags

Bits	Usage	Read Access	Write Access
[7:0]	Number of zero bits in Hbk1 or Hbk.	Readable in all security life-cycle states.	Writeable in DM LCS and RMA LCS.
[14:8]	Number of zero bits in Kcp (128-bit).	Readable only in CM LCS and DM LCS.	Writeable in DM LCS and RMA LCS.
[15]	Kcp "not in use" bit. If Kcp is not in use, this bit should be set by the OFT.	Readable in all security life-cycle states.	Writeable in DM LCS and RMA LCS.
[22:16]	Number of zero bits in Kce.	Readable only in CM LCS and DM LCS.	Writeable in DM LCS and RMA LCS.
[23:23]	Kce "not in use" bit. If Kce is not in use, this bit should be set by the OFT.	Readable in all security life-cycle states.	Writeable in DM LCS and RMA LCS.
[29:24]	Reserved.	Always readable.	Always writeable.
[30]	DM RMA LCS flag.	Readable in all security life-cycle states.	Writeable in CM LCS, DM LCS and Secure LCS.
[31]	CM RMA LCS flag.	Readable in all security life-cycle states.	Writeable in CM LCS, DM LCS and Secure LCS, only if the CM RMA locking bit in the AO module is not set.

4.4 Life Cycle States

Life cycle states have CM, DM, Secure, and RMA. Figure 3 shows the transitions of Life Cycle States (LCS) of DA16200/DA16600. This enables the device to behave differently in each LCS, protecting any security assets once they have been introduced into the device and reducing the risk of IP theft and reverse engineering.

Figure 3 shows the LCS transitions:

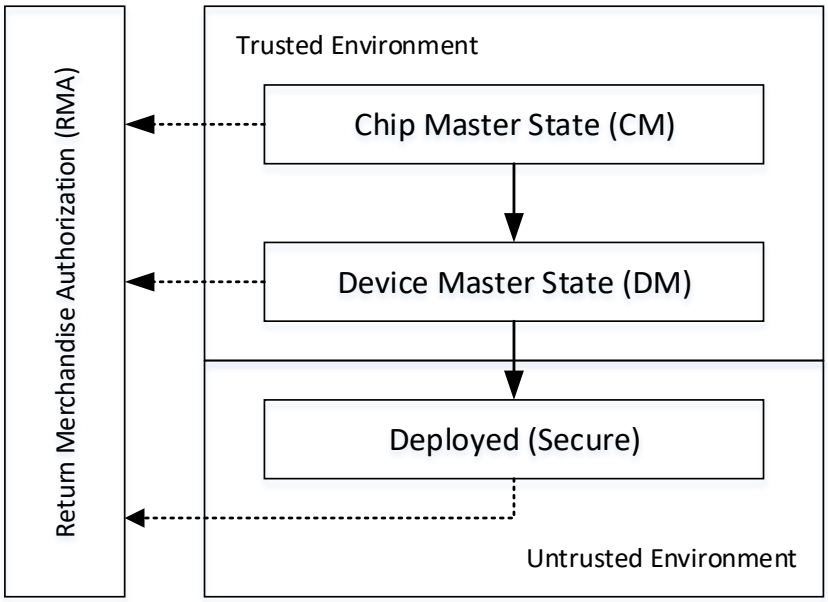

Figure 3: LCS Transitions

4.4.1 CM LCS

The device is in CM LCS if the following is true:

- CM-programmed flags: OTP word 0x10 = 0
- DM-programmed flags: OTP word 0x21 = 0

Thus, the default H/W state is CM LCS. In this LCS, all debug interfaces (UART and JTAG) are enabled.

A CMPU package binary image that is generated with the Security Tool includes the following assets and should be burned into OTP in CM LCS:

- HUK: OTP word 0x00-0x07
  The number of zero bits in HUK: Bits[7:0] of OTP word 0x10
- Hbk0: OTP word 0x11-0x14
  The number of zero bits in Hbk0: Bits[30:24] of OTP word 0x10
- General purpose configuration (GPPC) flags, OTP word 0x27
- CM DCU locking if Hbk0 is used

Once these assets are burned, the device does a Power on Reset (PoR) to transition to DM LCS.

4.4.2 DM LCS

The device is in DM LCS if the following is true:

- CM-programmed flags: OTP word 0x10 ≠ 0
- DM programmed flags: Bits[7:0] of OTP word 0x21 = 0

In this LCS, all debug interfaces (UART and JTAG) are still enabled.

A DMPU package binary image that is generated with Security Tool includes the following assets and it should be burned into OTP in DM LCS:

- Hbk1: OTP word 0x15-0x18

  The number of zero bits in Hbk1: Bits[7:0] of OTP word 0x21
- Optional: DM DCU locking if Hbk1 is used

Once these assets are burned, the device does a PoR to transition to Secure LCS

4.4.3 Secure LCS

The Deployed (Secure) LCS is used for devices out of the manufacturing line and in the field. It permits the execution of security functions but blocks all debugging and testing capabilities. Use of Secure Boot is mandatory in this LCS. The device is in Secure LCS if the following is true:

- CM programmed flags: OTP word 0x10 ≠ 0
- DM programmed flags: Bits[7:0] of OTP word 0x21 ≠ 0

Secure LCS is the state changed with the DMPU process and see Section 5.2 for more information. This is the state that should be applied at mass production when secure boot is required. Once in this state, the debug interface such as JTAG cannot be used anymore for security reasons. To enable the disabled debug interface, a firmware image with a Debug certificate must be used as described in Section 5.5.

4.4.4 RMA LCS

The Return Merchandise Authorization (RMA) LCS is a terminal state for devices that are returned to a Chip maker (for example, Renesas Electronics) for analysis of fatal failures. When a device is put into RMA LCS, it loses its existing secret keys, but regains full access to all debugging and testing capabilities. All cryptographic engines are usable for test purposes, but the root keys change for each boot phase.

- HUK is replaced with a different random value with each boot cycle. Therefore, any previously-saved data that is protected by a key derived from HUK is lost
- Kce and Kceicv are invalidated so that Secure Boot can be used only in non-encrypted mode
- Kcp and Kpicv are invalidated, so that provisioning can no longer be done based on the previous values

There are two separate certificates needed to enter a device into RMA LCS - CM RMA and DM RMA. CM RMA is a certificate image with CM RoT (Hbk0) chain and will remove CM keys in OTP.

DM RMA is a certificate image with DM RoT (Hbk1) chain and will remove DM keys. For the detailed information about the process, see Section 5.6.

4.5 Boot Services

The boot services in DA16200/DA16600 include the Secure Boot and Secure Debug certificate-based mechanisms that use an RSA private-public key scheme. Secure Boot and Secure Debug are based on the following elements:

- OTP secrets

  Provisioned to the device during the device manufacturing stage (CM LCS or DM LCS).

- ROM code

  A code library linked into the ROM of the device.

- RSA scheme verification

Verification of Secure Boot and Secure Debug is done over a certificate chain that is two or three certificates long. Each certificate is signed and verified with an RSA PSS scheme (RSA 3072 Private-Public Key scheme and compliant to PKCS#1 Ver. 2.1, RSA-PSS).

4.5.1 Secure Boot

Secure Boot guarantees that only authenticated software images (optionally encrypted) are loaded on a target system. A certificate is a message used to prove ownership of a public key. The certificate contains information about the public key, the authentication hash of the next key, and the signature that verifies contents.

The general structure of a certificate is as show in Figure 4.

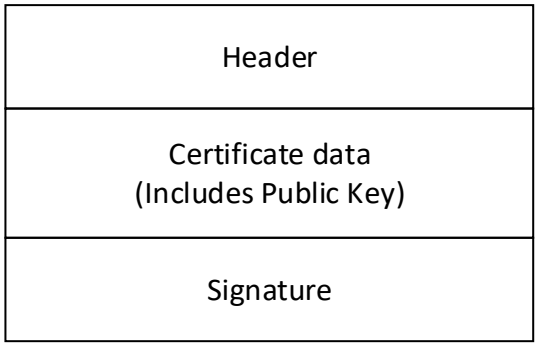

Figure 4: General Structure of a Certification

A signature is generated by encrypting a hash of the Certificate Data using a private key. Signature verification is done using a public key to decrypt a hash of the same Certificate Data. If the Certificate Data is compromised for any reason, then the decrypted hash of the Certificate Data would be different from the original signature and certificate verification fails.

DA16200/DA16600 uses a Certificate Chain for secure certificate verification.

A three-level “self-signed” certificates chain is used, which are a series of certificates that contain the public keys and are signed with a corresponding private key.

The Secure Boot certificate chain is composed of key certificates and content certificates.

- Key certificates

  Mainly the 1st or 2nd certificate in the certificates chain.

- Content certificates

  The last certificate in a certificates chain, which is used to load and validate software components.

Figure 5 shows a three-certificate chain.

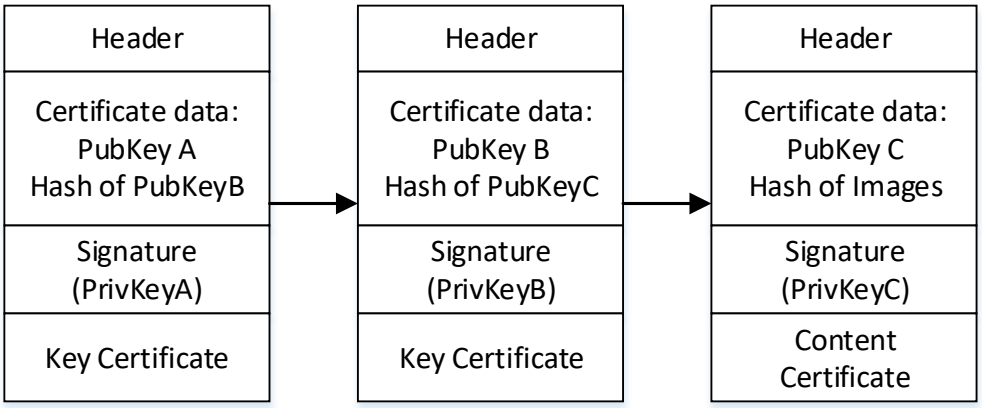

Figure 5: Three-Certificate Chain

- Three-level SB certificate scheme

  The order of three-level SB certificate chain is: master key certificate → key certificate → content certificate.

Even when a key used in a 3rd or 2nd certificate is leaked, it can be replaced with another key if the private key used in the first certificate is not compromised.

4.5.1.1 Secure Boot Flow

To verify a certificate, follow the steps below in DA16200/DA16600:

- Get the public key from the certificate and calculate its hash (HBK1, or HBK0)
- Verify the calculated hash:
  - If it is the first certificate in the chain, compare it with the hash value stored in the OTP
  - Otherwise, compare it with the saved hash from the previous certificate in the chain
- Verify the RSA signature with the public key of the certificate
- Save the public key hash of the next certificate, unless it is the last certificate in the chain

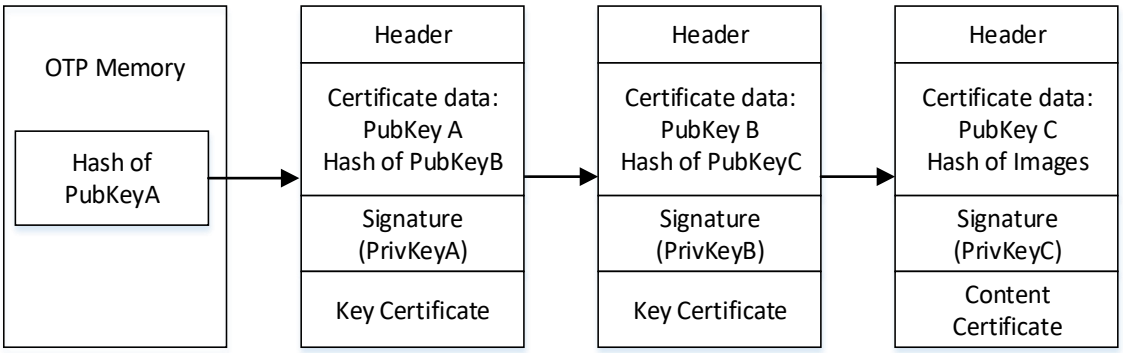

Figure 6: Secure Boot Flow

The entire certificate chain mentioned above is included in the built Image of DA16200/DA16600. And it is impossible for an unauthorized image to boot because of the verification process with this certificates chain.

Figure 7 shows the overall certificate-verification process.

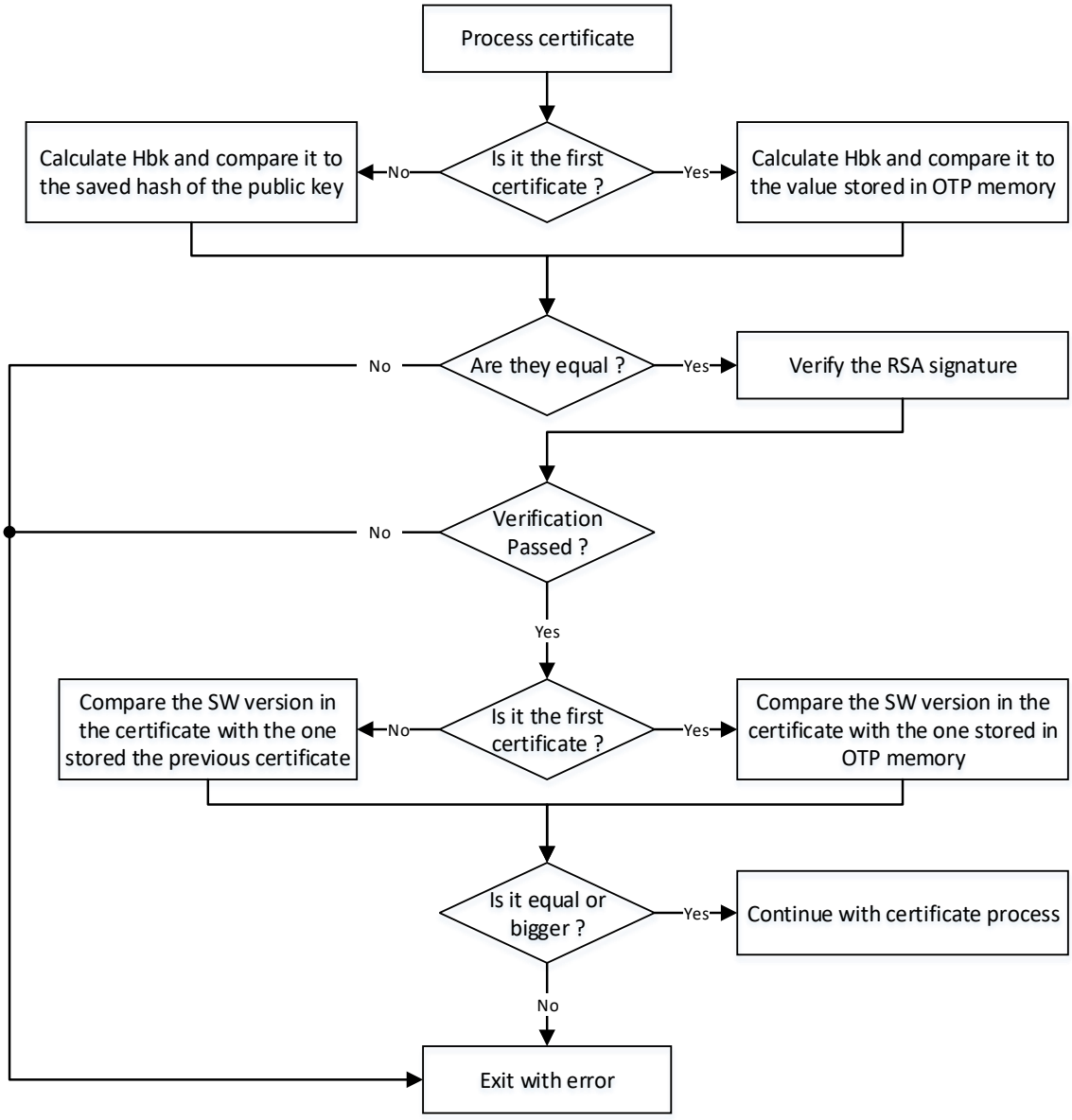

Figure 7: Overall Certificate-Verification Process

Figure 8 shows how the content-certificate process is done in a loop to process every S/W image that is signed in the certificate.

The content certificate contains the following information for every image that must be verified:

- The address that the S/W image is loaded to [load address]
- The flash address that the S/W image is stored in [storage address]
- The size of the S/W image

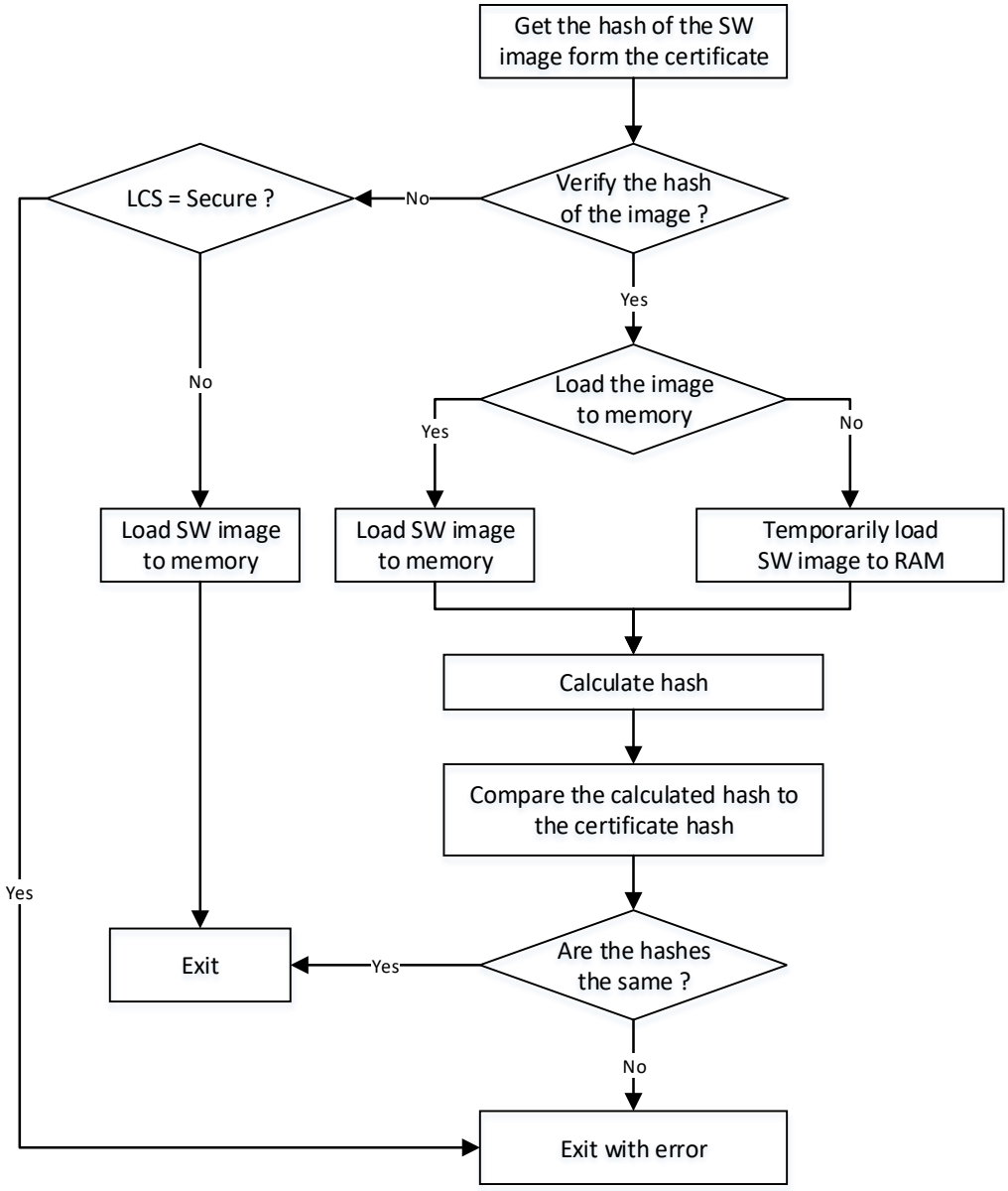

Figure 8: Certification Contents in S/W Images

Figure 9 shows the structure of the built image of the DA16200/DA16600. The key and content certificate chain are included in the image.

Besides certificates, the following contents are included in the image of DA16200/DA16600.

- Serial Flash Discoverable Parameters (SFDP) information to control serial flash memory
- Debug certificate (optional)
- S/W component (maximum of 3 components available)

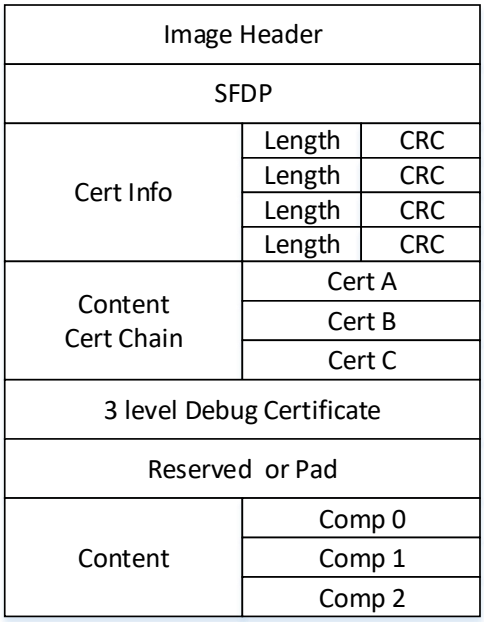

Figure 9: Certification Contents in DA16200/DA16600

CertA and CertB are key certificates, and CertC is a content certificate as shown in Figure 9. Content can be a UEboot image or an RTOS built from SDK.

- UEboot image (XXUEBOOTXX.img) built from our SDK contains a bootloader (UEboot) binary as S/W component (Comp0)
- RTOS image (XXRTOSXX.img) built from our SDK includes an RTOS binary as a S/W component (Comp0)

All CertA, CertB, and CertC are generated with the Security Tool and attached to each binary (UEboot and RTOS) to make a bootable image for DA16200/DA16600.

- CertA and CertB are the same for all images while CertC is different for each image
- CertA with a Hbk1 (DM RoT) chain is generated with the name of "sboot_hbk1_3lvl_key_chain_issuer.bin" in the dmpublic directory in the Security Tool
- CertB with Hbk1 is generated with the name "sboot_hbk1_3lvl_key_chain_publisher.bin" in the dmpublic directory
- CertC is different from each image, because CertC contains the information of each image such as content and size as described before
- CertC for the UEboot binary is generated with the name "sboot_hbk1_ueboot_cert.bin" in the dmpublic directory
- CertC for the RTOS binary has the name "sboot_hbk1_cache_cert.bin"

4.5.2 Secure Debug

Secure Debug is a certificate-based mechanism that uses an RSA private-public key scheme. It enables secure debugging of the device.

Secure Debug supports the following operations:

- Perform boot-time verification of debug certificates that enable authenticated debugging of secure domains. The secure domains are controlled by the Debug Control Unit (DCU) registers on the SoC
- Allow an authorizing party to shift the device into RMA LCS by using the same certificate mechanism (This is called "Secure RMA")

There are two-certificate chains in the debug certificate: an enabler certificate and a developer certificate. An enabler debug certificate can enable certain debug interfaces for a developer to debug a certain device. The developer enters SoC-ID of the target device to extend this to an actual debug certificate.

Figure 10 shows a three-level SD certificate scheme.

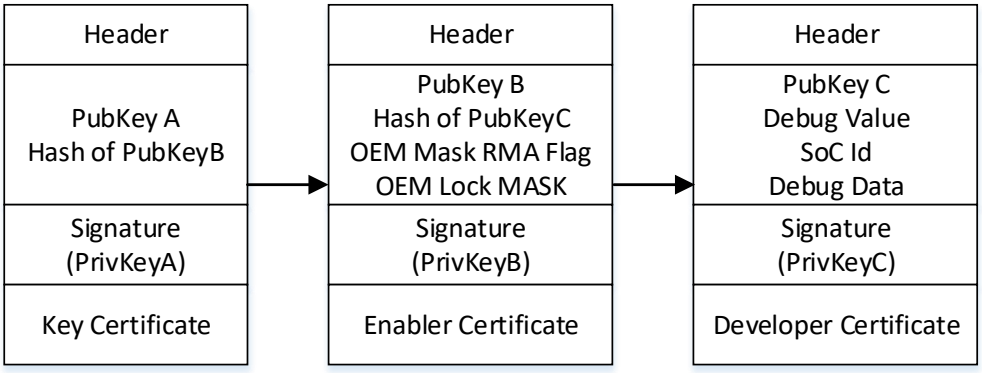

Figure 10: Three-Level SD Certificate Scheme

Note that the developer certificate can be generated with a SoC ID. If this does not match the target device, then the debug interfaces (JTAG and UART) will not be enabled. Once a debug certificate is verified during the boot sequence of the device, the permitted debug interfaces in the DCU mask of the enabler certificate will be enabled (UART0 and JTAG for DA16200/DA16600) for the designated device of the SoC-ID in the developer certificate.

An enabler certificate has the following fields.

Table 4: Items in Enabler Certificate

Items	Condition	Description
RMA-mode	Mandatory if debug-mask is not defined. Cannot be defined together with debug-mask.	Defines whether to use this certificate for entry into RMA LCS, by setting to a non-zero value. Set when "Secure RMA" is run in the Security Tool.
debug-mask	Mandatory if RMA-mode is not defined. Cannot be defined together with RMA-mode.	The DCU mask allowed by the enabler. A 128-bit mask. Set when "Secure Debug" is run in the Security Tool.
debug-lock	Mandatory if RMA-mode is not defined.	An additional DCU lock mask required by the enabler and is a 128-bit mask. These bits are added to the OTP-based mask.

A developer certificate has the following fields.

Table 5: Items in Developer Certificate

Items	Condition	Description
SoC-ID	Mandatory.	SoC-ID of the device. Developers can enable debug interfaces of the device with this SoC-ID. If try to enable the debug interface of the device with a different SoC-ID, it fails.
debug-mask	Mandatory if RMA-mode is not defined. Cannot be defined together with RMA-mode.	The DCU mask allowed by the developer. A 128-bit mask.

A debug certificate is generated at Secure Debug in the Security Tool, which includes a debug-mask configuration in the enabler certificate and a SoC-ID for the developer certificate. An RMA certificate is generated at Secure RMA in the Security Tool, which includes an RMA-mode configuration in the enabler certificate and a SoC-ID for the developer certificate.

4.6 Device Provisioning

Device provisioning refers to burning secret keys and assets in the OTP memory of a device in a secure manner. The CM keys and assets in Table 6 should be burned in the OTP in CM LCS, and the DM keys and assets in Table 7 should be burned in the OTP in DM LCS.

Table 6: CM Keys and Assets in CM LCS

Key names or Assets	Functions
Kpicv and Kceicv	CM key
Hbk0	Root Of Trust
Asset	CM DCU lock bits
Asset	Configuration bits (General Purpose Flag)

A CM Provisioning Utility (CMPU) package binary contains all of the above items and is generated when "Secure Production" is run in the Secure Tool.

Table 7: DM Keys and Assets in DM LCS

Key names or Assets	Functions
Kcp and Kce	DM key
Hbk1	Root Of Trust
Asset	DM DCU lock bits

After the secrets and asset are burned in the OTP memory, LCS automatically changes to Secure LCS, and the JTAG debug interface in DA16200/DA16600 is disabled. In order to enable the JTAG debug interface again, the debug certificate scheme should be applied. The platform key (Krtl) is required to generate a CMPU and DMPU package binary to encrypt all assets as described in Section 4.2.2.

4.7 Secure Asset

After device provisioning, secret keys in the OTP memory can be used to encrypt or decrypt user data in the flash memory. It provides APIs and procedures to encrypt and manage data to be stored in FLASH with the AES CCM method.

4.7.1 API for Secure Assets

Secure assets are encrypted data stored in FLASH. Data decryption is done with the key derived from the provisioning key Kpicv or Kcp that is stored in the OTP memory. Therefore, there is no risk of key disclosure. The Security Tool supports the creation of the secure asset, encrypted with a key derived from the provisioning key. The DA16200/DA16600 SDK provides an API to decrypt assets with the key derived from the OTP memory keys by the H/W Crypto engine.

The Secure Asset Service uses a CMAC algorithm based on AES encryption and has a file size restriction:

- The valid size of unencrypted data must be multiplied of 16 bytes
- The maximum size of unencrypted data cannot exceed 512 bytes
- The maximum size of the secure asset is 560 bytes including the header size

The decryption API provided by the SDK is FC9K_Secure_Asset(). See Table 8.

Table 8: Secure Asset Runtime APIs

```
extern UINT32 FC9K_Secure_Asset(
       UINT32 Owner
       , UINT32 AssetID
       , UINT32 *InAssetData
       , UINT32 AssetSize
       , UINT8 *OutAssetData
);
```

- Owner

  Key type number. Use **1** for Kpicv, or **2** for Kcp.
- AssetID

  ID information used in the encryption process.
- InAssetData

  Secure Asset Data. This data must be loaded into SRAM since this function does not access FLASH.
- AssetSize

  Size of Secure Asset Data.
- OutAssetData

  Decrypted Asset Data. This data must be allocated in SRFAM since this function does not run a memory allocation.

The Secure Asset is generated with "CM.4.secuasset.bat" in the folder SBOOT.

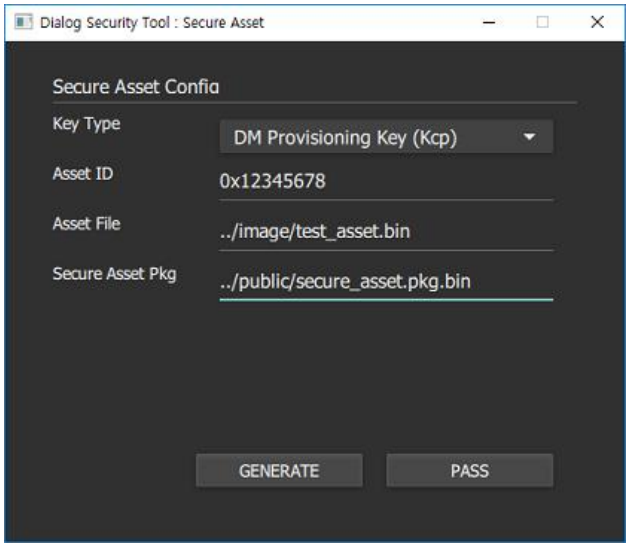

Figure 11: Encryption Process of Secure Asset

Table 9 shows example code for decrypting a Secure Asset in FLASH.

Table 9: Secure Asset Decryption Process

```
UINT32  status;
UINT32 assetsiz, encassetsiz;
UINT8 *asset;
UINT8 *dump_encasset_hex = NULL;
UINT32 address ;

dump_encasset_hex = APP_MALLOC((512+48)); // header + asset

address = htoi(argv[1]);
encassetsiz = htoi(argv[2]);

status = sbrom_sflash_read( address, dump_encasset_hex, encassetsiz);

if( status == TRUE ){
      asset = CRYPTO_MALLOC(512);

      assetsiz = FC9K_Secure_Asset(2              // 1 : Kpicv, 2 : Kcp
              , 0x00112233                        // Asset ID
              , (UINT32 *)dump_encasset_hex       // secure asset
              , encassetsiz                       // size of secure asset
              , asset);                           // decrypted asset

      if( assetsiz > 0 ){
        CRYPTO_DBG_DUMP(0, asset, assetsiz);
      }

      CRYPTO_FREE(asset);
}

PP_FREE(dump_encasset_hex);
```

4.7.2 Secure Storage

The Secure Storage is a concept similar to the Secure Asset, but some features are different. Secure Storage is encrypted with a key derived from one of the followings: user key, root key, Kcp or Kpicv. It also supports full services to encrypt raw data and decrypt secure data, but the Secure Asset only supports one-way function used to decrypt assets.

The following table shows the functions and related definition items for Secure Asset.

Table 10: Secure Asset Runtime APIs

```
typedef enum  {
    ASSET_USER_KEY = 0,
    ASSET_ROOT_KEY = 1,
    ASSET_KCP_KEY = 2,
ASSET_KPICV_KEY = 4,
} AssetKeyType_t;

typedef struct {
    UINT8     *pKey;
    size_t    keySize;
} AssetUserKeyData_t;

typedef struct {
        uint32_t  token;
        uint32_t  version;
        uint32_t  assetSize;
} AssetInfoData_t;

#define CC_RUNASSET_PROV_TOKEN     0x416E7572UL
#define CC_RUNASSET_PROV_VERSION   0x10000UL

extern INT32 FC9K_Secure_Asset_RuntimePack(
       AssetKeyType_t KeyType
       , UINT32 noncetype
       , AssetUserKeyData_t *KeyData
       , UINT32 AssetID
       , char *title
       , UINT8 *InAssetData
       , UINT32 AssetSize
       , UINT8 *OutAssetPkgData
     );

extern INT32 FC9K_Secure_Asset_RuntimeUnpack(
       AssetKeyType_t KeyType
       , AssetUserKeyData_t *KeyData
       , UINT32 AssetID
       , UINT8 *InAssetPkgData
       , UINT32 AssetPkgSize
       , UINT8 *OutAssetData
     );
```

- AssetKeyType_t

This defines the type of the derived key stored in the OTP to be applied to the key derivation function CMAC to be used for encryption. ASSET_ROOT_KEY means Huk, ASSET_KCP_KEY means Kcp, and ASSET_KPICV_KEY means Kpicv. If the user-defined key is used in addition to the key stored in OTP, it should be defined as ASSET_USER_KEY and KeyData should be set as input value.

- AssetUserKeyData_t

This is a structure to define a user-defined key when ASSET_USER_KEY is used. The user-defined key defines 128/192/256 bits, pKey defines the buffer pointer of Key Data, and keySize defines 16/24/32 Bytes, which means key length.

- FC9K_Secure_Asset_RuntimePack()

This function encrypts raw input data with an AES CCM method and has the following parameters:

  - KeyType
    - Defines the type of decryption key to use for encryption.
  - Noncetype
    - Defines how to generate the nonce information used in the encryption process. '0' is the Nonce generated by TRNG, and '0xFFFFFFFF' is the Nonce generated by PRNG.
  - KeyData
    - This means the parameter to input User Key when KeyType is defined as ASSET_USER_KEY.
  - AssetID
    - This is the ID information used in the encryption process.
  - Title
    - This is a parameter to enter the title information of the Runtime Asset Package.
  - InAssetData
    - This is the data pointer of the raw data to be encrypted.
  - AssetSize
    - This is the size of the raw data and must be defined as 16 Bytes multiple for AES, which is a block cipher.
  - OutAssetPkgData
    - This is the data pointer of the encrypted Runtime Asset Package. Since the function does not perform internal memory allocation, the data buffer for the output data should be pre-allocated and allocated to Raw Data Size + 48 bytes, considering 48 bytes of information field to be additionally tagged.
  - If the Return Value is less than 0, it means error. If the Return Value is larger than 0, it means size information of output data OutAssetPkgData.

- FC9K_Secure_Asset_RuntimeUnpack()

  This is a function to decrypt the encrypted input Runtime Asset Package, and the input parameter needs to input the encryption parameter applied to function FC9K_Secure_Asset_RuntimePack().

  - KeyType
    - This should match the type of decryption key used in encryption.
  - KeyData
    - If KeyType is defined as ASSET_USER_KEY, it should match key information used as User Key.
  - AssetID
    - This should match the ID information used for encryption.
  - InAssetPkgData

  - This is the data pointer of the Runtime Asset Package to be decoded.
- AssetPkgSize
  - This is the size of the Runtime Asset Package, which means Raw Data Size + 48 bytes.
- OutAssetData
  - This is the data pointer of the decoded raw data, and the size is the raw data size.
- If the Return Value is less than 0, it means an error. If the Return Value is larger than 0, it means size information of output data OutAssetData.

Table 11 and Table 12 show example code to implement Secure Storage in FLASH that use the Runtime Pack / Unpack function.

Table 11: Encryption Process

```
{
        UINT32  status;
        UINT32 assetid, assetoff;
        INT32  assetsiz, pkgsiz;
        UINT8  *assetbuf, *pkgbuf;

        assetid = htoi(argv[2]); // Asset ID
        assetoff = htoi(argv[3]); // FLASH Offset
        assetsiz = htoi(argv[4]); // plaintext, InAssetPkgData size

        assetsiz = (((assetsiz + 15) >> 4)<< 4); // 16B aligned
        PRINTF(" Aligned Asset Size:%d\n", pkgsiz);

        assetbuf = APP_MALLOC(assetsiz);
        pkgbuf   = APP_MALLOC(assetsiz + 48);

        if( assetbuf == NULL ){
                return;
        }
        if( pkgbuf == NULL ){
                APP_FREE(assetbuf);
                return;
        }

        // Step 1. Read Raw Data from FLASH
        pkgsiz = 0;
        status = sbrom_sflash_read( assetoff, assetbuf, assetsiz);

        // Step 2. AES Encryption
        if( status > 0 ){
                pkgsiz  = FC9K_Secure_Asset_RuntimePack(ASSET_ROOT_KEY
                                        , 0
                                        , NULL, assetid, "RunPack"
                                        , assetbuf, assetsiz, pkgbuf );
        }
        // Step 3. Write Runtime Package Data to FLASH
        if( pkgsiz > 0 ){
                PRINTF("PKG Size:%d\n", pkgsiz);
                sbrom_sflash_write(assetoff, pkgbuf, pkgsiz);
        }

        APP_FREE(pkgbuf);
        APP_FREE(assetbuf);
```

```
        }
```

Table 12: Decryption Process

```
{
 UINT32  status;
 AssetInfoData_t AssetInfoData;
 UINT32 assetid, assetoff, flagwrite;
 INT32  assetsiz, pkgsiz;
 UINT8  *assetbuf, *pkgbuf;

 assetid = htoi(argv[2]); // Asset ID
 assetoff = htoi(argv[3]);  // FLASH Offset
 flagwrite = htoi(argv[4]); // Test only. flash write option flag

 // Step 1. Read Info Block of Runtime Asset Package
 status = sbrom_sflash_read(assetoff
              , (UINT8 *)(&AssetInfoData), sizeof(AssetInfoData_t));
 if( status == 0 ){
       PRINTF("SFLASH Read Error:%x\n", assetoff);
       return;
 }
 if( (AssetInfoData.token == CC_RUNASSET_PROV_TOKEN)
  && (AssetInfoData.version == CC_RUNASSET_PROV_VERSION) ){
       assetsiz = AssetInfoData.assetSize;
       PRINTF("Stored PKG Size:%d\n", assetsiz);
       pkgsiz = assetsiz + 48;
 }else{
       PRINTF("Illegal Asset Package:%X.%X\n"
              , AssetInfoData.token, AssetInfoData.version );
       return;
 }

 assetbuf = APP_MALLOC(assetsiz);
 pkgbuf   = APP_MALLOC(pkgsiz);

 if( assetbuf == NULL ){
       return;
 }
 if( pkgbuf == NULL ){
       APP_FREE(assetbuf);
       return;
 }

 // Step 2. Read Runtime Asset Package form FLASH
 assetsiz = 0;
 status = sbrom_sflash_read( assetoff, pkgbuf, pkgsiz);

 // Step 3. AES Decryption
 if( status > 0 ){
       assetsiz = FC9K_Secure_Asset_RuntimeUnpack(ASSET_ROOT_KEY
                           , NULL, assetid, pkgbuf, pkgsiz, assetbuf );

 }

 if( assetsiz > 0 ){
```

```
                PRINTF("ASSET:%d\n", assetsiz);
                CRYPTO_DBG_DUMP(0, assetbuf, assetsiz);

                // Step 4. Test only. Write Raw Data to FLASH
                if( flagwrite == 1 ){
                        sbrom_sflash_write(assetoff, assetbuf, assetsiz);
                }
        }else{
                PRINTF("ASSET:decryption error (%x)\n", assetsiz);
        }

        APP_FREE(pkgbuf);
        APP_FREE(assetbuf);
    }
```

4.7.3 Secure NVRAM

The contents in NVRAM can be encrypted with the runtime APIs for security. Huk, Kpicv, and Kcp in the OTP are used in Secure NVRAM. When the NVRAM APIs are used, the developer can read and write certain items in the NVRAM area on the flash memory. See Ref.[4]. for more information.

When Secure NVRAM is enabled by the following commands, the items to write to the flash will be encrypted before writing, and the items to read will be decrypted when reading from the flash internally.

```
[DA16200] nvram.nvedit secure 1          // Key selection: 1 HUK, 2 Kpicv, 4 Kcp
[DA16200] nvram.nvedit save sflash       // Activates Secure NVRAM. Henceforth,
encryption and decryption will be performed internally whenever read or write to the
NVRAM occurs.
```

4.7.3.1 Cryptographic Acceleration

MbedTLS APIs are used for cryptographic functions in DA16200/DA16600. MbedTLS is an open source SSL library that enables developers to include cryptographic and SSL/TLS capabilities in their embedded products, with a minimal coding footprint.

Developers can choose between H/W-accelerated cryptographic operations and the S/W cryptographic implementation of Mbed TLS for each feature supported by both Mbed TLS and CryptoCell-312:

- The Mbed TLS cryptographic implementation provides an interface to the standard cryptographic operations. For example, AES, RSA or ECC.
- The dedicated CryptoCell-312 APIs provide an interface to the non-standard or specific CryptoCell-312 operations. For example, key derivation using HUK.

Mbed TLS and CryptoCell-312 are flexible in terms of which features are compiled in each. To control which components are Mbed TLS-based or CryptoCell-312-based, developers must edit the config-cc312.h configuration file. This file is located in crypto/inc/mbedtls/config.h. It includes all the flags that are supported by Mbed TLS, with the additional XXX_ALT definitions. These XXX_ALT definitions are for the components that are accelerated by the H/W.

By default, Renesas Electronics SDK comes with the minimal required features that CryptoCell-312 accelerates. See Ref. [4] on how to use mbedTLS APIs. Table 13 shows the supported H/W acceleration crypto algorithms in DA16200/DA16600.

Table 13: H/W Acceleration Crypto Algorithms

Algorithm	Mode	Key Sizes
AES	ECB, CBC, CTR, OFB,CMAC, CBC-MAC, AES-CCM, AES-CCM*, AES-GCM	128 bits, 192 bits and 256 bits
AES key wrapping	N/A	All
Chacha and Chacha-Poly1305	N/A	N/A
Diffie-hellman • ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography • Public-Key Cryptography Standards (PKCS) #3: DiffieHellman Key Agreement Standard.	N/A	1024 bits, 2048 bits and 3072bits
ECC key generation	N/A	NIST curves and 25519 curves
ECIES	N/A	NIST curves and 25519 curves
ECDSA	N/A	NIST curves and ED25519
ECDH	N/A	NIST curves and 25519 curves
Hash	SHA1, SHA224 and SHA256	N/A
HKDF	N/A	N/A
HMAC	SHA1, SHA224 and SHA256	N/A
KDF NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions	CMAC or HMAC	N/A
RSA PKCS#1 operations • Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications. • Public-Key Cryptography Standards (PKCS) #1 v1.5: RSA Encryption.	Encryption and signature schemes	2048 bits, 3072 bits and 4096bits
RSA key generation	N/A	2048 bits and 3072 bits

5 Security Tool

The Security Tool is provided to generate secret keys, certificates, and secure binary images for DA16200/DA16600. There are four major things developers can do with the Security Tool:

- Generate RoT (Hbk0, Hbk1) and CM/DM secret keys (Kpicv, Kceicv, Kcp, and Kce). It also generates CMPU and DMPU binary which contain all keys to be burned into OTP memory.
- Build Secure Boot images (secure bootloader and RTOS Images) that run on DA16200/DA16600.
- Generate Secure Debug certificates and images.
- Generate RMA certificates and image.

Figure 12 shows the top window of the Security Tool when running "CM.1.secuman.bat" at SBOOT directory in our SDK.

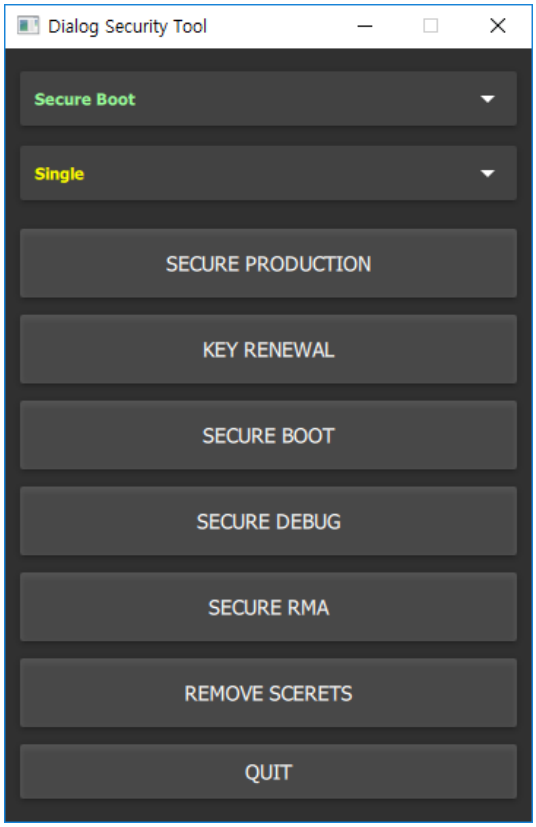

Figure 12: Top Window of the Security Tool

5.1 Role Selection

Three roles are available:

- Single Manager

"Single" is a top manager who is responsible to generate and manage all secret keys of the product. Only the Single Manager has the authority to generate, renew or remove the secret keys. Most importantly, the private key that corresponds to the RoT (Hbk0 and Hbk1) in the OTP memory should be kept and maintained by the Single Manager.

The Single Manager has the responsibility to keep the private key to itself and not expose the private key for any reason. If exposed, there is no guarantee of security so that products that have the corresponding RoT in the OTP should be recalled. For this reason, pay extra attention when a developer takes on the role of "Single" manager.

- SB Publisher

The “SB Publisher" role has to generate the third certificate, for example, the content certificate, which is needed for Secure Boot in a three-level certificate scheme and to rebuild Secure bootable images with it (all UEboot, RTOS, and images).

Only the Secure Boot menu is enabled for this role. The main responsibility of this role is to remove the debug certificate in the image after Secure Debug. A debug certificate is in place in the images after running Secure Debug. An image with a debug certificate will enable debug interfaces. Use this role to remove the debug certificate and build only Secure Boot images that disable debug interfaces for security.

- SB/SD Publisher

The "SB/SD Publisher" role has to generate the third certificate, for example, the content certificate, which is needed for Secure Boot in a three-level certificate scheme and to rebuild Secure Boot images with it (all UEboot, RTOS, and images).

In addition, the "SB/SD Publisher" role has to generate the Debug certificate for Secure Debug with the SoC-ID of the target device enabling the debug interface (JTAG port) of the target device and to rebuild Secure bootable images (only the UEboot image is rebuilt).

Only Secure Boot and Secure Debug menus are enabled for the SB/SD Publisher role. See Figure 13.

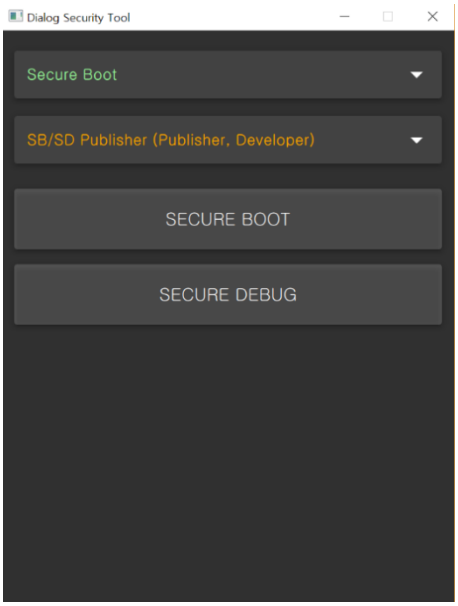

Figure 13: Secure Boot and Secure Debug Menu

When Secure Debug is selected, a popup window appears requesting the Soc-ID of the target device. See Figure 14.

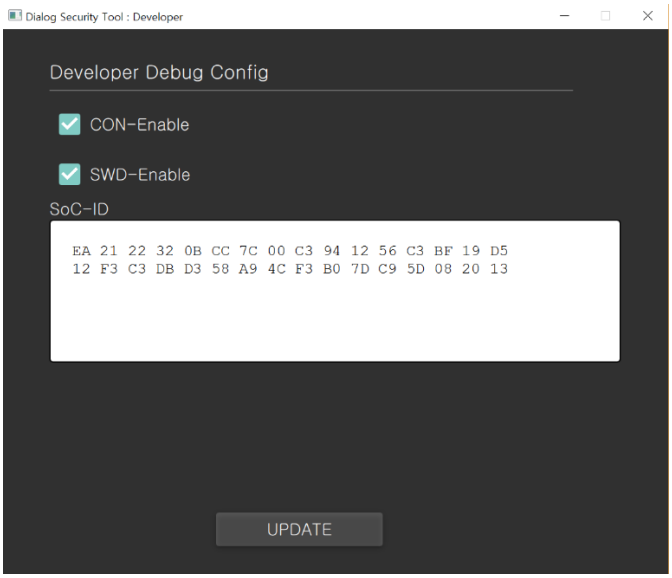

Figure 14: Request Soc-ID in Secure Debug

The SoC-ID of the target device can be checked with the following console command:

```
[/DA16200] sys.socid
```

Copy the SOC-ID with the `sys.socid` command to the SoC-ID field in the Security Tool window. Note that SoC-ID is only valid when the target device is in Secure LCS. The SB/SD Publisher role is useful when the developer wants to make their third party (or developer) debugging the end-product in the field and not expose secrets. The third party can make a secure bootable image with this role and debug the product.

5.2 Secure Production

Secure Production generates all the secret keys such as CM keys, DM keys, and keys for the 2nd certificate and 3rd certificate. And the certificate chains that use the generated keys are generated to create Secure Boot and Secure Debug images. Note that SDK should be built before executing the Secure Production process.

Table 14 shows which files are generated when Secure Production is used.

Table 14: Secret Keys for Secure Production

Items	CM/DM Keys	Directory	Generated Files
CM keys	CM keys	cmsecret	OTP keys: cmkey_pair.pem, kceicv.bin, kpicv.bin
			Private keys for Secure Boot: cmissuer_keypair.pem, cmpublisher_keypair.pem
			Private keys for Secure Debug: cmenabler_keypair.pem, cmdeveloper_keypair.pem
DM keys	DM keys	dmsecret	OTP keys: dmkey_pair.pem, kce.bin, Kcp.bin

			Private keys for Secure Boot: dmissuer_keypair.pem, dmpublisher_keypair.pem
			Private keys for Seucre Debug: dmenabler_keypair.pem, dmdeveloper_keypair.pem
certificates for Secure Boot	with CM keys	cmpublic	sboot_hbk0_3lvl_key_chain_issuer.bin, sboot_hbk0_3lvl_key_chain_publisher.bin
	with DM keys	dmpublic	sboot_hbk1_3lvl_key_chain_issuer.bin, sboot_hbk1_3lvl_key_chain_publisher.bin, and content certificates for UEboot and RTOS images
certificates for Secure Debug	with CM keys	cmpublic	sdebug_hbk0_3lvl_key_chain_enabler.bin, sdebug_hbk0_3lvl_key_chain_developer.bin
	with DM keys	dmpublic	sdebug_hbk1_3lvl_key_chain_enabler.bin, sdebug_hbk1_developer_pkg.bin

To enter CM and DM secret keys into OTP memory, special binaries called CMPU package and DMPU package are also generated after Secure Production.

- CMPU and DMPU package binary contains the items in Table 14

Note that "Krtl.key" (the platform key) should be in place in the **cmsecret** directory in order to run Secure Production. If there is no proper platform key in the **cmsecret** directory, not able to run Secure Production. The platform key will be provided by Renesas Electronics upon request.

After successful Secure Production, the platform key will be deleted by the Security Tool for security concerns. The platform key should not be exposed for any reason.

When the **Secure Production** button is clicked on the Security Tool, a popup window appears preventing developers remove the files by mistake.

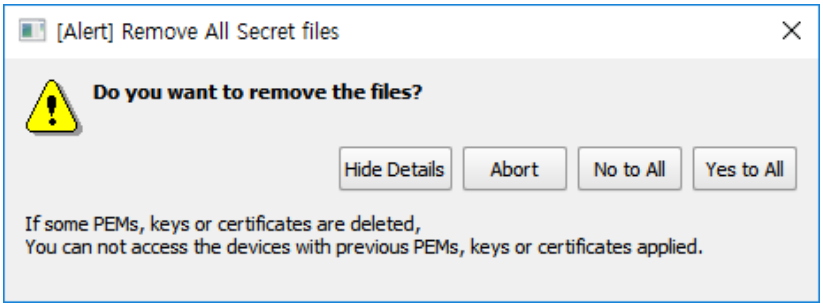

Figure 15: Prevent Accidental Removal of Secret Keys in Secure Production

When the Security Tool is used for the first time, select **Yes to All**. The Secure Production process starts logging on both the console window and the log file in the **example** directory.

The log messages for Secure Production are saved in **secure_production.txt** file in the **example** directory. The file allows developers to check procedure or error messages.

Pay extra attention when it is not the first time using the Security Tool and selected **Yes to All**. If using the Security Tool again, the previously generated secret keys and certificates will be lost and regenerated keys and certificates.

After successful Secure Production, **cmpu.pkg.bin** and **dmpu.pkg.bin** files are in the **public** directory. At production time, these package binaries should be downloaded to Sflash memory at the address shown in Table 15.

Table 15: CMPU/DMPU Download Address in Sflash

Binary	Start Address
Cmpu.pkg.bin	0x001F_2000
Dmpu.pkg.bin	0x001F_3000

Note that the addresses in Table 15 are the default address in our SDK and can be changed under the circumstances. And UEboot binary for the production version should be used for mass production.

When the pre-built binary is used :

There are UEboot binaries provided in the **image** directory and the developer must set them for the respective purposes.

Table 16: UEboot Binary Definition of Secure Boot, Non-Secure Boot and RMA

UEboot Binary Name	Purpose
DA16xxx_ueboot.bin.SecureBoot	Production version UEboot
DA16xxx_ueboot.bin.NoneSecure	Normal version UEboot
DA16xxx_ueboot.bin.RMA	RMA version UEboot

Before the SDK is built, UEboot binaries should be renamed to build a bootable UEboot image (DA16xxx_ueboot_xxx.img). After the SDK is built, a bootable UEboot image is available in the **public** directory.

Table 17: UEboot Binary Setting for Secure Boot, Non-Secure Boot and RMA

SDK	Secure Type	UEboot Binaries Setting
FreeRTOS	Secure Boot	DA16xxx_ueboot.bin.Secure.4MB → DA16xxx_ueboot.bin
	Non-Secure Secure Debug	DA16xxx_ueboot.bin. NonSecure.4MB → DA16xxx_ueboot.bin
	RMA	DA16xxx_ueboot.bin. RMA.4MB → DA16xxx_ueboot.bin

When the UEboot binary is built from DA16x00 UEboot Project:

UEboot binary should be built according to the Secure Production process described in the Table 17.

For building UEboot according to its purpose, following definitions in the UEboot_initialize.c should be defined or undefined beforehand.

Table 18: Proper Definitions for Secure Production

Definition	SUPPORT_SECURE_REGION_LOCK	SUPPORT_DA16X_RMA_OTP_ERASE
Secure Boot	undef	undef
Non Secure/Secure Debug	define	undef
RMA	undef	define

UEboot image will be generated in apps/da16x00/get_started/image folder.

For the CMPU and DMPU process, all UEboot, and RTOS images should be downloaded to Sflash beforehand.

To download the UEboot image, run the following command at the MROM prompt and select the production version UEboot image.

- `[MROM] loady boot`

In case of FreeRTOS SDK, download the RTOS image.

- `[MROM] loady 23000   // for RTOS image`
- `Power OFF and ON`
- `[DA16200] reset      // to enter into MROM`

To download the CMPU binary, run the next command at the MROM prompt and select cmpu.pkg.bin.

- `[MROM] loady 1f2000 1000 bin`

To download the DMPU binary, run the next command at the MROM prompt and select dmpu.pkg.bin.

- `[MROM] loady 1f3000 1000 bin`

The command(sys.sprod) in the RTOS image is used to write secrets into the OTP memory. Therefore, an RTOS image should be run to provision the secrets in the CMPU and DMPU binaries. The developer needs to boot with RTOS. To do so, press the power off/on button, or use the **boot** command at the MROM prompt. hbk0 and CM keys can be burned into the OTP memory with the below-mentioned command on the [DA16200] prompt in RTOS.

- `[DA16200] sys.sprod`

When successful, the following message is output:

- Product.CMPU: 0

After the power off/on is pressed, the LCS of the DA16200 will change from CM LCS to DM LCS.

hbk1 and DM keys can be burned into the OTP memory with command:

- `[DA16200] sys.sprod`

When successful, the following message is output:

- Product.DMPU: 0

After the power off/on button is pressed, the LCS of the DA16200 will change from DM LCS to Secure LCS, in which JTAG is disabled and only enabled again with a Debug Certificate. Once completed, the CMPU and DMPU binary in the flash should be deleted for security reasons. Command `sys.sprod` will erase the binaries on the flash.

- `[DA16200] sys.sprod`

Command `sys.sprod` will output some messages similar to that shown in Table 18.

Table 18: Success Message to Change from DM to Secure LCS

```
CC_BsvSocIDCompute return SocID
        7D D2 00 E0 F1 06 43 F5 AF 5A 17 3F BF A6 8E 3D
        03 4C B7 DA AA 6D DB 39 51 0B F5 D5 62 7E 2C 8F
Product.CMPU: Erased
Product.DMPU: Erased
Product.SLock: 1
Product.State: Secure Boot Scenario - Good
```

The example shows the SoC ID of the device (it will be different from your device) and the status of the CMPU and DMPU binary (whether they are erased or not). Command `Product SLock` shows the status of a control bit in the OTP. If the value is 1, then the DA16200 performs a secure boot.

After all the above-mentioned procedures are completed, the production version of UEboot should be replaced with a normal version of UEboot (rename "DA16xxx_ueboot.bin.NoneSecure" to "DA16xxx_ueboot.bin" in the "image" directory and build the SDK) with the following command at the MROM prompt to download the image.

- `[MROM] loady boot`

The following table summarizes which directories are the most important ones after Secure Production and that should not be exposed for any reason because of security.

Table 19: Directory Definition for Secure Production

Directory	Contents
cmsecret	CM private keys and encryption keys (private/public key pair, Kceicv, and Kpicv)
cmpublic	1st and 2nd certificate for Secure Boot and Secure Debug that use Hbk0 (CM root key)
dmsecret	DM private keys and encryption keys (private/public key pair, Kcp, and Kce)
dmpublic	1st, 2nd, and 3rd certificate for Secure Boot and Secure Debug that use Hbk1 (DM root key)

Secure Boot images with the certificate chain based on the above-mentioned keys are generated in the **public** directory.

- Secure Boot images in FreeRTOS SDK
  - UEboot image (XXUEBOOTXX.img) built from our SDK contains a bootloader (UEboot) binary
  - RTOS image (XXRTOSXX.img) built from our SDK contains the RTOS binaries.

If the target device already went through the CMPU and DMPU process and the above images were downloaded to the Sflash at the proper address, it will boot correctly.

5.3 Key Renewal

When one of the 2nd and 3rd private keys is exposed for any reason, those private keys need to be changed with the Key Renewal menu. Be cautious before clicking Key Renewal. When you click the button, all previously generated 2nd, 3rd private keys and certificates are deleted and regenerated from the start. Note that RoT (1st private key) cannot be changed.

If you click the **Key Renewal** button in the Security Tool, the confirmation popup window shown in Figure 16 displays to prevent developers from accidentally deleting it.

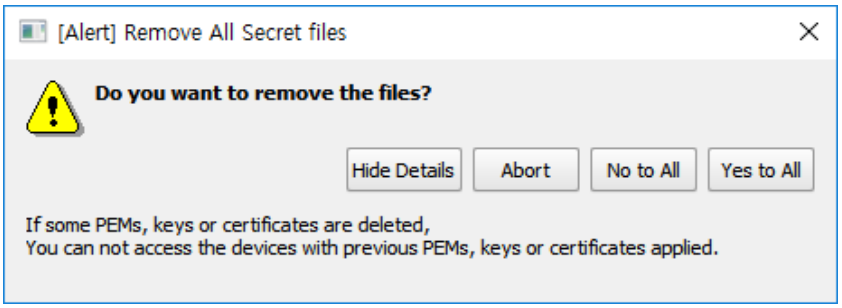

Figure 16: Prevent Accidental Removal of Secret Keys in Key Renewal

To renew the key, select **Yes to All**. The previously generated 2nd and 3rd private keys and the certificates are deleted and regenerated.

The following table summarizes which directories are updated after key renewal.

Table 20: Directory Definition for Key Renewal

Directory	Contents
dmsecret	2nd/3rd private keys for Secure Boot and Secure Debug.
dmpublic	1st/2nd/3rd certificates for Secure Boot and Secure Debug that use Hbk1 (DM root key).
dmpubkey	2nd/3rd public keys.
dmtpmcfg	Configurations.
public	Images with the certificate chain for Secure Boot.

Secure Boot images with the certificate chain based on the renewed keys are generated in the **public** directory.

The **key_renewal.txt** file in the **example** directory is a log file for the Key Renewal process. The file can be used to check the log or read error messages that occurred.

5.4 Secure Boot

After running the Secure Debug menu, the generated image contains a Debug Certificate but no Content Certificate chain. See Figure 17.

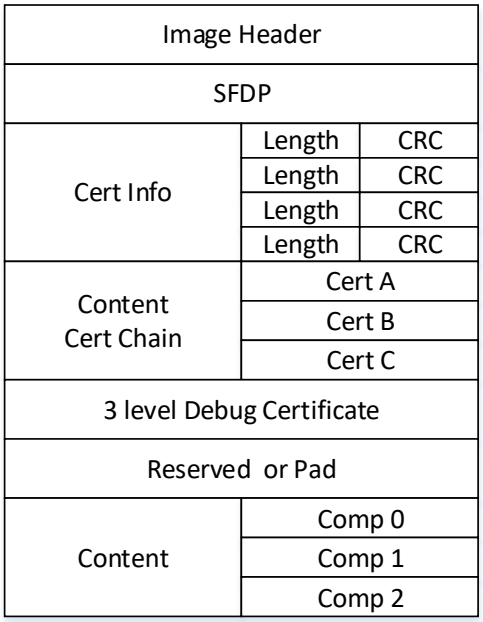

Figure 17: Debug Certificate of Secure Debug Menu

For Secure Boot, an image with a Content Certificate chain is required without a Debug certificate. To generate images for Secure Boot, run the **Secure Boot**. Secure Boot images with the certificate chain are generated in the **public** directory.

- Secure Boot images in FreeRTOS SDK
  - UEboot image (XXUEBOOTXX.img) contains a bootloader (UEboot) binary
  - RTOS image (XXRTOSXX.img) contains RTOS binaries

The **secure_debug.txt** in the **example** directory is a log file for Secure Boot process. The file can be used to check the log and read error messages that occurred.

5.5 Secure Debug

The debug port in the DA16200/DA16600 JTAG is disabled by default when entered into Secure LCS. When this debug port needs to be re-enabled for debug purposes, then a Secure Debug image should be used. There is an optional Debug certificate field in an image.

At the boot sequence, a check is done to see whether the Debug certificate exists in the image. If a Debug certificate exists, then the SoC ID in the Debug certificate is checked to see if it matches with the target device. When it does match, the debug port is enabled and boot.

When `Secure Debug` is run in the Security Tool, the window shown in Figure 18 will display to enter the SoC ID of the target device. Use `sys.socid` command in the console to check what the SoC-ID is of the target device.

- `[DA16200] sys.socid`

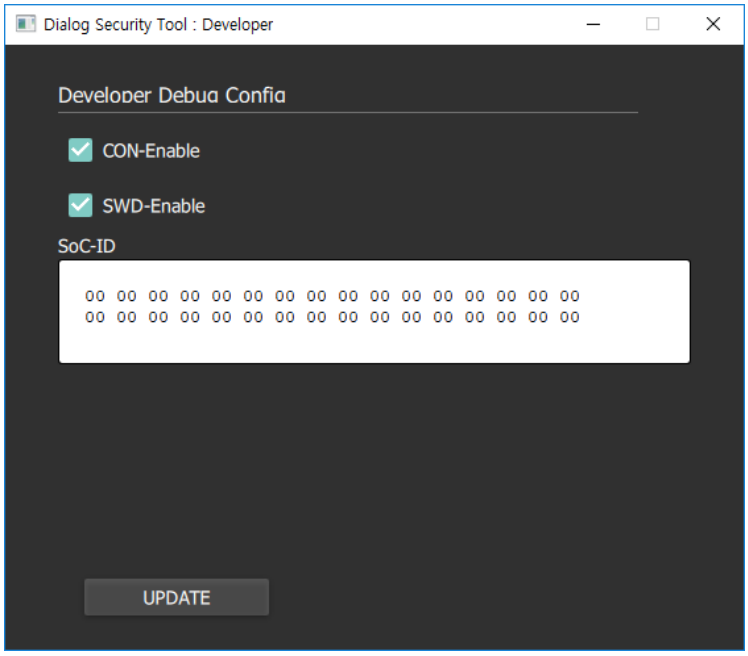

Figure 18: Window to Enter SoC ID in Secure Debug

You can copy the SoC-id from the console command to the window shown in Figure 18 and then click **UPDATE**. The following table summarizes which directories are updated from Secure Debug.

Table 21: Directory Definition for Secure Debug

Directory	Contents
dmpublic	Developer certificate with the SoC-ID.
public	Images with Debug certificate.

Secure Debug images with the Debug certificate are generated in the **public** directory.

- Secure Debug images in FreeRTOS SDK
  - UEboot image (XXUEBOOTXX.img) includes a bootloader (UEboot) binary
  - RTOS image (XXRTOSXX.img) includes the RTOS binaries

The **secure_debug.txt** in the **example** directory is a log file for the Secure Debug process. The file can be used to check the log and read error messages that occurred.

5.6 Secure RMA

As described earlier, the LCS of the chip should be changed to RMA-LCS before the chip is sent to the chip maker (for example, Renesas Electronics) for analysis.

A Debug certificate that has an RMA flag enabled (RMA certificate) is required to enter a device into RMA LCS. In addition, to erase secret keys in the OTP memory, a specific UEBoot binary for RMA is required. This UEboot binary for RMA is provided in the SDK with the name UEbootXXRMAXX.bin. Like Secure Debug, Secure RMA is allowed for a specific device and a SoC-ID is required for the RMA certificate.

When changing to RMA-LCS, secret keys in the OTP memory such as Kpicv, Kceicv, Kcp, and Kce are erased to prevent that the developer's secret keys are exposed and the debug port (JTAG) is re-enabled for debugging purposes.

When running Secure RMA, the window in Figure 19 will be displayed to enter the SoC-ID in the RMA certificate for the target device. Copy and paste the SoC-ID from console command `sys.socid` to the Security Tool window and then click **UPDATE**.

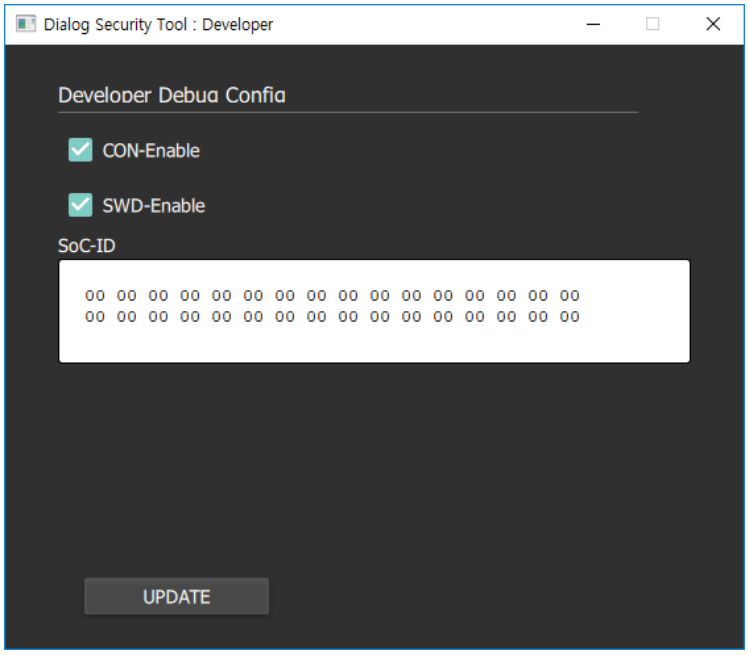

Figure 19: Window to Enter SoC ID in RMA

There are two images with an RMA certificate generated in the **public** directory: DA16xxx_rma.img and DA16xxx_rma_icv.img. Image DA16xxx_rma.img is for the RMA image with DM keys and will erase the DM keys in the OTP. Image DA16xxx_rma_icv.img is for the RMA image with CM keys and will erase the CM keys in the OTP.

After UEboot for RMA to the Sflash is updated, do the following for the RMA process.

- `[MROM] loady boot [RMA version of UEboot]`

To run an RMA image with DM keys, run the following command at the MROM prompt and download DA16xxx_rma.img.

- `[MROM] loady 1f2000 1000 bin`

After downloading, you need to reboot the system and set the DM RMA flag using the following command at the [DA16200] prompt:

- `[DA16200] sys.sbrom sflash 1f2000`
- `Power OFF and ON     // for POR`

To run an RMA image with CM keys, run the following command at the MROM prompt and download DA16xxx_rma_icv.img.

- `[MROM] loady 1f2000 1000 bin`

After downloading, you need to reboot the system and set the CM RMA flag using the following command at the [DA16200] prompt:

- `[DA16200] sys.sbrom sflash 1f2000`
- `Power OFF and ON     // for POR`

All HUK, CM and DM keys are erased from OTP in the UEBoot initialization phase during POR boot.

To check if the device entered properly into RMA, use command `sys.socid`.

After the above steps are done, the Non-Secure UEBoot image should be in place again on the Sflash.

- `[MROM] loady boot [Non-Secure of UEboot]`

Table 22 summarizes which directories are updated from Secure RMA.

Table 22: Directory Definition for Secure RMA

Directory	Contents
cmpublic	Debug certificate with RMA enabled (RMA certificate) with CM key chain (Hbk0).
dmpublic	Debug certificate with RMA enabled (RMA certificate) with DM key chain (Hbk1).
public	Images with RMA certificate with both DM key chain and CM key chain (DA16xxx_rma.img and DA16xxx_rma_icv.img).

The **secure_rma.txt** in the **example** directory is a log file for the Secure RMA process. The file can be used to check the log and read error messages that occurred.

5.7 Remove Secrets

When the developer wants to have a third party (or developer) to debug the end-product in the field, the developer should run Remove Secrets before the SBOOT directory is delivered to the third party, to remove all important secret keys and certificates. Note that before running this menu, the original SBOOT directory should be already backed-up in a safe location because all secret keys will be removed. Then, the third party can make its own debug images with the SBOOT and IAR environment.
After debugging is complete by the 3rd party, the developer should apply the resolving patch codes from the third party to the SDK and build the SDK, which makes UEboot, RTOS binaries in SDK that are copied to the **image** directory.

When clicking the Remove Secrets, a confirmation window shows. See Figure 20.

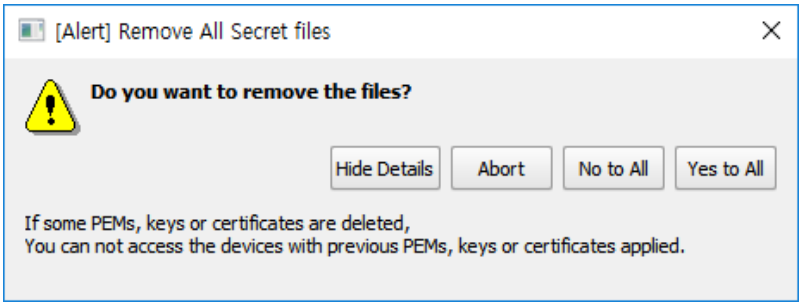

Figure 20: Prevent Accidental Removal of Secret Keys in Secure RMA

Select **Yes to All** to remove all secrets. Next, the window in Figure 21 shows.

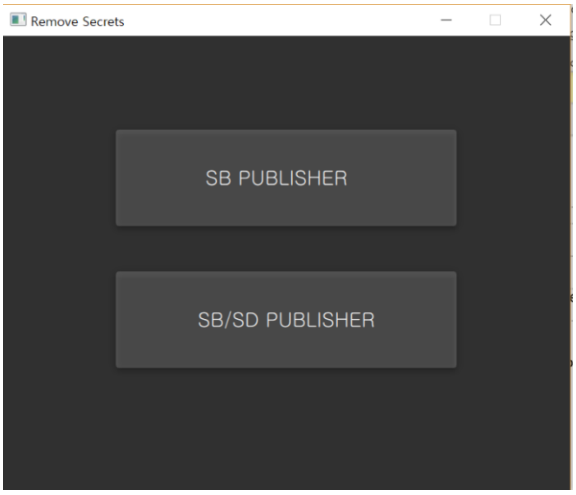

Figure 21: Remove Secret Keys in Secure RMA

This is to determine to who to send SBOOT and what files should be removed accordingly. Files that will be removed according to the selected target are summarized in Table 23.

Table 23: Directory Definition to Remove Secret Keys in Secure RMA

Target	Directory	Removed files
SB Publisher	cmsecret	All files
	cmpublic	All files
	dmsecret	All files except dmpublisher_keypair.pem
	dmpublic	enc.kce.bin, enc.kcp.bin, and all sdebug_* files
SB/SD Publisher	cmsecret	All files
	cmpublic	All files
	dmsecret	All files except dmpublisher_keypair.pem, dmdeveloper_keypair.pem
	dmpublic	enc.kce.bin, enc.kcp.bin, sdebug_hbk1_enabler_rma_pkg.bin, sdebug_hbk1_developer_rma_pkg.bin

After this, SBOOT can be sent to the third party (or developer) for debugging or development purposes.

Revision History

Revision	Date	Description
2.1	31-Jul-2023	● Updated the reference section ● Updated Section 5.2
2.0	16-Nov-2022	● Changed company name from Dialog to Renesas ● Updated Table 1: change user area range in OTP memory
1.9	28-Mar-2022	● Updated logo, disclaimer, and copyright.
1.8	9-Dec-2021	Added Table 17: UEboot Binary Setting for Secure Boot, Non-Secure Boot and RMA
1.7	23-Nov-2021	● Updated RMA procedure ● Add the guide of security tool in FreeRTOS SDK ● Change title and file name from DA16200 to the DA16200 DA16600
1.6	15-May-2020	Updated User Manual for Security Tool v2.0
1.5	21-04-2020	Added: Remove CMPU and DMPU Binary
1.4	16-Dec-2019	● Added Write CM and DM package at Sflash ● Added Change Life Cycle Status (LCS) ● Added Change to secure boot mode
1.3	16-Dec-2019	Editorial review
1.1	11-Sept-2019	Updated: How to generate and burn secret keys
1.0	03-Jul-2019	Preliminary DRAFT Release

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

RoHS Compliance

Renesas Electronics' suppliers certify that its products are in compliance with the requirements of Directive 2011/65/EU of the European Parliament on the restriction of the use of certain hazardous substances in electrical and electronic equipment. RoHS certificates from our suppliers are available on request.

Important Notice and Disclaimer

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers skilled in the art designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only for development of an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising out of your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu

Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

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:

https://www.renesas.com/contact/