Rev. 2,9/2023

S32G Secure Boot Customization

by John Li (nxa08200)

This Application note explains the Secure Boot features of S32G bootloader, considering the following customization:

- Add HSE FW update function.
- Add OTP Attribute setting function.
- Add IVT signature function.

To improve Secure Boot functionality.

History	Comments	Author
V1	● Create the doc	● John.Li
V2	● Translate to Eng	● John.Li

Contents

1 Reference Materials 2
2 S32G Secure Boot 3
2.1 IVT header format for the Secure Boot part 3
2.2 Secure Boot Flow 3
2.3 Secure Boot Configuration 4
2.4 HSE background of Secure Boot 6
3 Build the Project 7
3.1 Build the Compiling Environment 7
3.2 Create IVT Image 7
3.3 Burning Image 8
3.4 Bootloader Secure Boot Testing 9
4 Bootloader Secure Boot Codes and Function Description 9
4.1 EB Configuration 9
4.2 EB output codes 15
5 Customization 1：HSE FW update 22
5.1 Codes development 23
5.2 Testing 26
6 Customization 2：HSE OTP Attribute Setting 26
6.1 Code Development 27
6.2 Simulation test 34
7 Customization 3：IVT Signature 36
7.1 Codes Development 36
7.2 Simulation Testing 41

1 Reference Materials

Take S32G2 RDB2 as sample：

Num	Materials	Comments	How to get it
1	Platform_Software_Integration _S32G2_2022_06.exe	The bootloader project, please refer to its document <<Bootloader UserManual. Pdf>> for the Secure Boot chapter after installation	www.nxp.com/s32g
2	AN13750: Enabling Multicore Application on S32G2 using S32G2 Platform Software Integration – Application note	Bootloader project use guide, including Secure Boot configuration instructions (XueweiWang).	www.nxp.com/s32g
3	S32G_Bootloader_V*.pdf	Bootloader project cusotmizaiton application note, include the non-secure boot parts (Johnli)	https://community.nxp.com /t5/NXP-Designs-Knowledge -Base/ S32G-Bootloader- Customzition/ta-p/ 1519838
4	HSE_H/M Firmware Reference Manual	HSE FW Reference manual	www.nxp.com/s32g (Need secure file access right)
5	Secure Boot with HSE V0 - Draft B	Secure Boot doc	www.nxp.com/s32g (Need secure file access right)
6	HSE_DEMOAPP_S32G2_0_1_0_5.exe	HSE Demo App(S32DS version)	www.nxp.com/s32g
7	HSE_FW_S32G2_0_1_0_5.exe	HSE FW install package, Please refer its doc<<HSE_FW_H_S32G2XX_0.1.0.5 _HSE_Service_API_ Reference_Manual.pdf>>	www.nxp.com/s32g

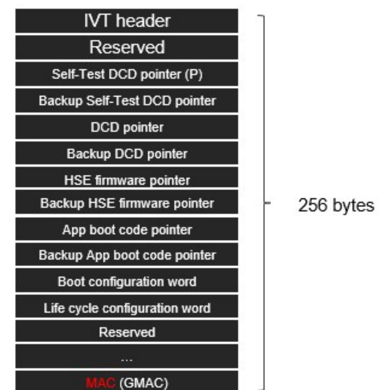
2 S32G Secure Boot

Secure Boot is divided into two parts: Secure Boot configuration and Secure Boot. After configuration is completed, the ROM code running on the HSE M7 core and the HSE FW are responsible for completing the authentication and decryption work related to Secure Boot.

2.1 IVT header format for the Secure Boot part

SECURE BOOT CONFIGURATION IN THE IVT

- IVT: Boot configuration word
  - BOOT_SEQ
    - 0b - Non-secure boot. BootROM executes application image without authentication
    - 1b - Secure boot. HSE_H firmware executes application image after authentication
- IVT: Life cycle configuration word
  - IN_FIELD
  - OEM_PROD

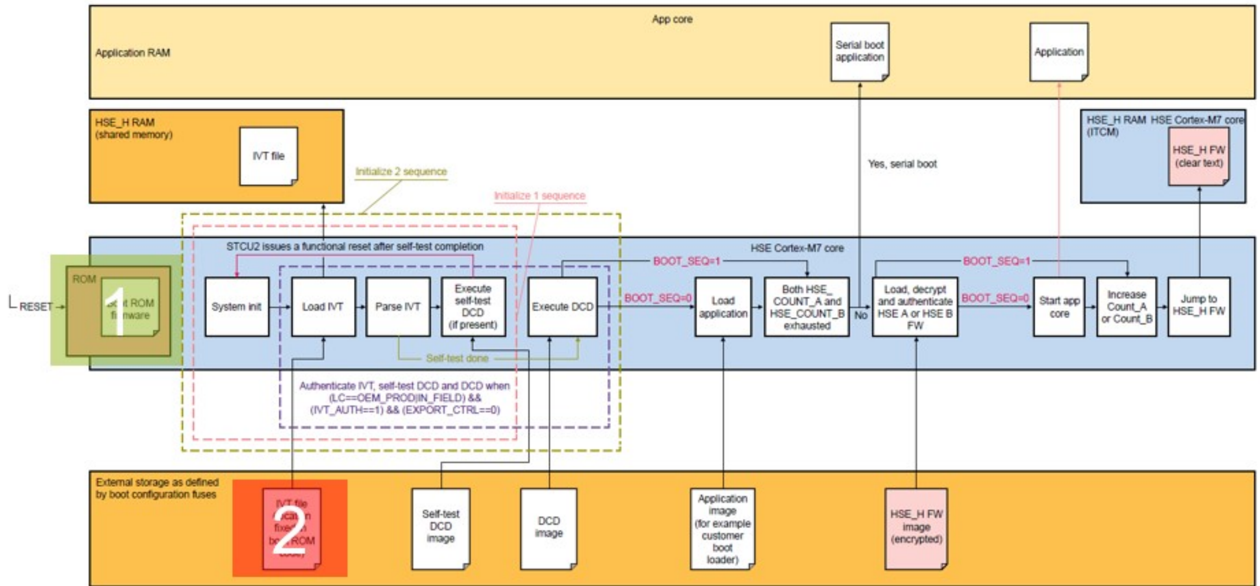


Note:

When creating an IVT image of Bootloader, it is important to ensure BOOT_ SEQ bit is not set, so the first startup is Non-secure boot. And because of BL_ ENABLE_ SECURE_BOOT compilation macro is configured to enabled in the Bootloader project, so the program determines that it is starting for the first time and will enter the Secure Boot configuration function. This function discovers that BOOT_ SEQ bit is not set, the Secure Boot configuration will be completed, and then the IVT BOOT will Set the BOOT_SEQ bit back to QSPI NOR, then restart and continue using Secure Boot.

2.2 Secure Boot Flow

HSE Secure Boot flow path as follows:



The whole Secure Boot flow path as follows:

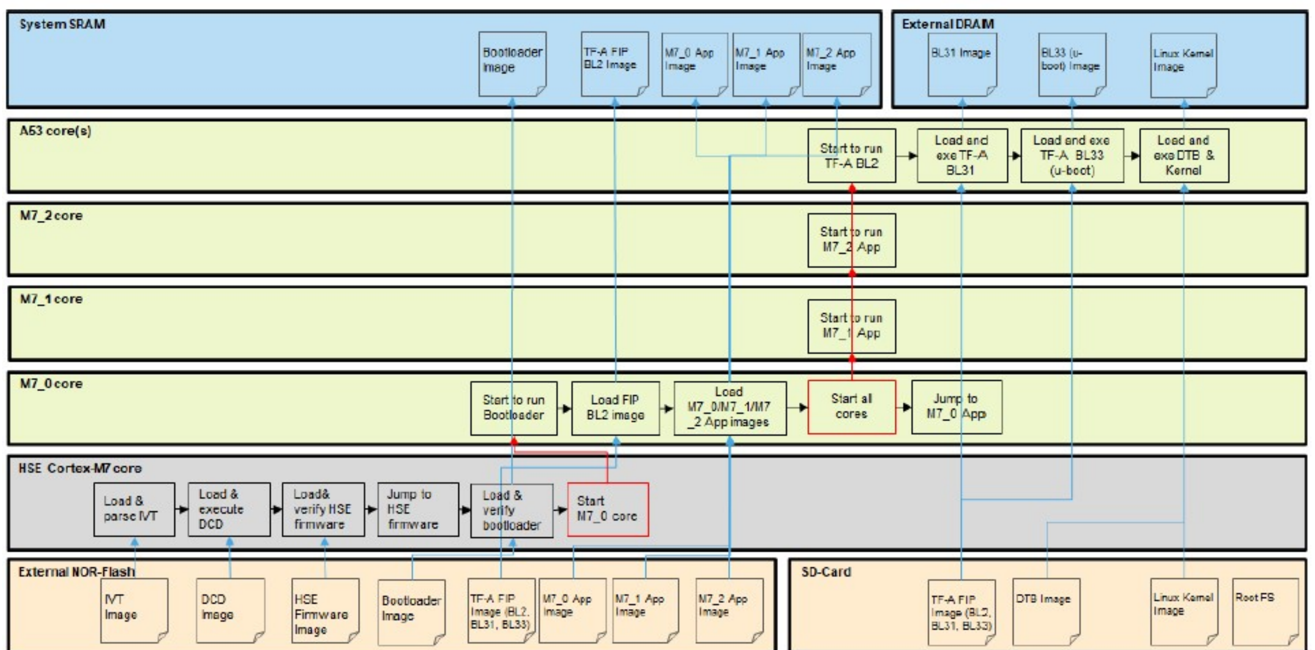


Figure 3. **Secure boot flow**

2.3 Secure Boot Configuration

The entire process of Secure Boot is mainly completed by HSE ROM code/HSE FW/Bootloader, so the key and difficulty lies in the configuration of Secure Boot. Generally speaking, customers have the following configuration methods:

- After Non Secure Boot, run the Autosar Crypto APP, call the MCAL interface to complete the configuration of Secure Boot, and then restart.
- After the Non Secure Boot is started, complete the configuration of the Secure Boot in the Bootloader, and then the Bootloader automatically restarts. After that, all Secure Boots are available. This is how the Bootloader project is designed.
- Mixed mode: Some configurations are automatically configured and restarted by the Bootloader. Some configurations are configured using the running APP.

Some configurations are suitable for automatic configuration using Bootloader, such as signature of Bootloader image, update of HSE FW, signature of IVT header, etc. However, there are also some configurations that are suitable for configuration using APP, which generally involve interaction with factory program burning servers, such as injection of keys and configuration of OTP Attribute. However, there is no fixed method for configuration, and each customer's approach is different.

Compared the document requirements, the configuration process for the Bootloader Secure Boot project and the HSE Demo Secure Boot project as follows:

steps		Bootloader Secure Boot Project	HSE Demo Secure Boot Project
1	format HSE key catalogs	format HSE key catalogs	format HSE key catalogs
2	Import the Keys	Inject RAM key and same NVM key for generating bootloader image CMAC (for verification)	Inject NVM key for generating/verifying bootloader image CMAC
3			Publish SYS_IMG
4			Store SYS_IMG into QSPI NOR
5	Configuring Secure Boot through SMR	Generate CMAC TAG for bootloader image	Same will more features support
6		Configure SMR for bootloader image, where NVM key is used	Same will more features support
7		Install SMR	Same will more features support
8		Install CR	Same will more feature support
9	Publish SYS_IMG to SRAM	Publish SYS_IMG to SRAM	Publish SYS_IMG to SRAM
10	Store SYS_IMG into QSPI NOR	Store SYS_IMG into QSPI NOR	Store SYS_IMG into QSPI NOR
11			HSE pink to blue update
12			Complete DCD/sefltest signature
13	Complete IVT signature		Complete IVT signature
14	Update IVT header to enable Secure Boot	Update IVT header to enable Secure Boot	Update IVT header to enable Secure Boot
15	Configure other OTP attributes		Configure other OTP attributes

16	Configure IVT Auth OTP Attribute to Enable IVT Protection		Configure IVT Auth OTP Attribute to Enable IVT Protection
17	Advanced LC to OEM_PROD Or IN_FIELD		Advanced LC to OEM_PROD Or IN_FIELD
18	Reboot	Reboot	Reboot

So if you only consider using the Bootloader project to achieve basic product level Secure Boot protection, consider adding the following customization functions (of course, you can also develop your own app to achieve this).

- The pink to blue update of HSE FW: which can improve startup speed and also increase the guarantee of preventing copying.
- The burning API for related OTP attributes: which is not tested, only the burning of ADKP is tested.
- The signature function of the IVT header: which is not tested for startup, only for signature testing.

This article did not consider the following:

- Bootloader image encryption: which will affect startup speed.
- SMR is only configured with one boot image Bootloader, and most applications require a Bootloader to start the M/A core and resolve related resource conflicts, such as clocks.
- The signature of DCD/selftest can be added by referring to the HSE Demo app when needed.

The customization part of this article mainly refers to the process of the HSE Demo APP, so the HSE Demo APP is a comprehensive reference platform for HSE functions

2.4 HSE background of Secure Boot

Including the following:

- Key catalog formatting
- Key import
- CMAC generation
- SYS_ IMG publish
- HSE FW_ IMG update
- SMR configuration
- CR configuration
- IVT AUTH
- OTP Attribute setting and reading, including life cycle evolution,
- etc…

Refer to the documents <<HSE_H/M Firmware Reference Manual>> and <<Secure Boot with HSE V *>> for relevant content.

3 Build the Project

3.1 Build the Compiling Environment

Refer the documents <<AN13750>> and <<S32G_Bootloader_V *. Pdf>>, a compilation environment can be built. We will not go into detail here. Please note:

- <<AN13750>> uses Secure Boot by default, and you can refer to this document, including the directory designation for HSE FW.
- <<S32G_Bootloader_V *. Pdf>> uses Non Secure Boot by default, but how to remove SDHC support and how to solve compilation errors can refer to this document.
- When only focusing on the Secure Boot section of the bootloader, we focus on the Secure Configuration and Boot sections, so the M/A section of bootloader startup is not described in this article. In code:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\generic\src\Bootloader.c

```
volatile int debug;
int main(void)
{…
debug =1;
while(debug); // Add debugging switcher interface to facilitate the use of lauterback for tracking debugging.
Bl_ConfigureSecureBoot();
while (1); // Stop continuing the boot for easy debugging.
if (E_OK != Bl_Run())
…
```

The above debugging switchers can be removed in the official code.

In addition, this project did not modify the image configuration of the bootloader entry.

3.2 Create IVT Image

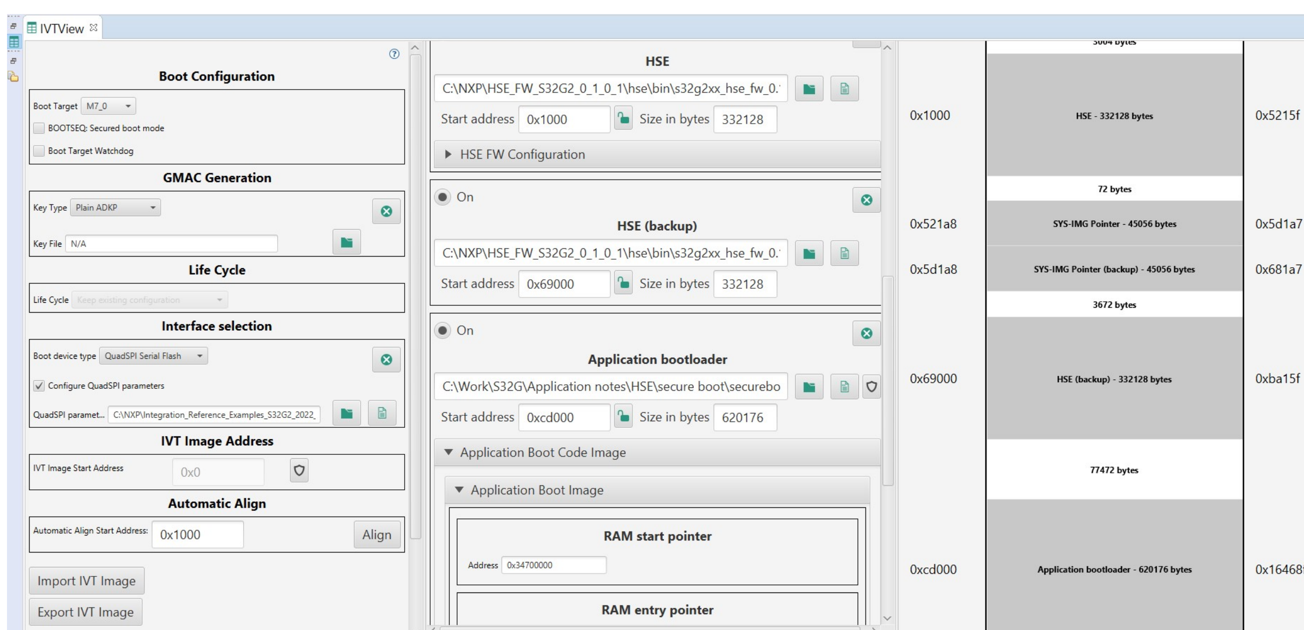
Refer to the documents <<AN13750>> and <<S32G_Bootloader_V *. Pdf>> for information on how to create an IVT image of a Bootloader.

Please note:

- Because we are considering developing the function of FW update in the future, when manufacturing IVT images, we prepare FW main images and FW backup images to facilitate FW backup.
- Also note that the sector size of the Fls driver is defined as: # define QSPI_ SECTOR_ SIZE 0x1000=4096, so we use 0x1000 when doing automatic align Start Address in the tool.
- For images that need to be updated, we set 0x1000 as the align, for example:

1. HSE FW Start address=0x1000
2. HSE FW Backup Start address=0x69000
3. App bootloader=0xcd000
4. In this case, we suggest setting the SYS-IMG address of the published file to 0x165000

As follow chart:



The maximum address of the image is 0x165000, which will be explained later.

3.3 Burning Image

Refer to the documents <<AN13750>> and <<S32G_Bootloader_V *. Pdf>> for instructions on how to burn the IVT image of the Bootloader.

Please note:

- Because this project does not contain any other images of the M/A core and only has one bootloader, we only burn the bootloader_blob.bin.

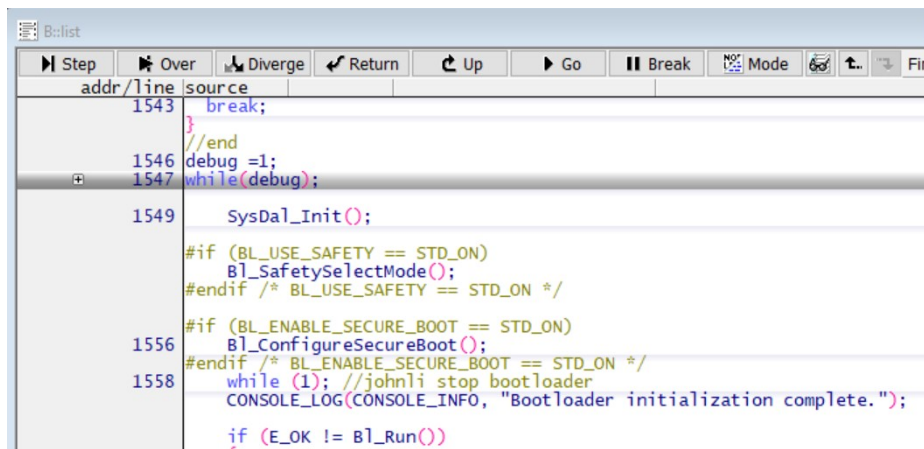
3.4 Bootloader Secure Boot Testing

Set the S32G RDB2 board to QSPI NOR for normal startup, and use Lauterbach to run the script after startup:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\build\cmm\connect_s32gxx_m7.cmm Connect the board and it will stop at:

while(debug);

Then switch the code mode to source code mode, and you can use Lauterbach to track:



Double click on “debug” and change “debug”=1 to “debug”=0 to continue running.

4 Bootloader Secure Boot Codes and Function Description

4.1 EB Configuration

Open the Bootloader project: The configuration of KEY installation information is as follows:

Bootloader_S32G2XX_ASR_4.4._M7->Bootloader(…)->Bootloader->General：

- Enable Secure Boot=checked //This flag indicates whether the Secure Boot sequence shall be activated or not
- Secure Boot Key Descriptor =/CryptoDal/CryptoDal/CryptoDalBswConfig/CryptoDescriptors_0 // Select an encryption descriptor from the security startup key descriptor. This key descriptor contains the configuration required for key image label generation and verification. For more information, please refer to the next descriptor section
- HSE System Image Address (1024 -> 4194304) =786432 //0xc0000 System Image Storage Address. At this flash address Bootloader will save the HSE system image

Set the HSE system image address to the given address in the QSPI flash memory. Ensure that the HSE system image address does not overlap with any configured memory areas of the HSE firmware, boot loader image, or application image. The reserved memory area of the system image will be 0x4000 bytes. If this region overlaps with any of the above images, the secure boot process or application loading from QSPI will fail. Because the maximum address of the image exported by our IVT tool is 0x165000, we have modified it to 1462272=0x165000.

Note:

The HSE System Image Address and subsequent 0x4000 bytes must be defined in QSPI Flash, aligned with the sector address. This is necessary because the boot loader needs to use the QSPI Flash driver to update the HSE system image.

Based on the above reasons, we need to add FLS driven sector mapping, including:

1. HSE FW Start address=0x1000: #define HSE_FW_IMAGE_SIZE 0x64000 //johnli for 400KB =64X0x1000
2. HSE FW Backup Start address=0x69000
3. In this case, we suggest setting the SYS-IMG address of the published file to 0x165000//# define HSE_ SYS_ IMAGE_ SIZE 0x4000=4X0x1000

Bootloader_S32G2XX_ASR_4.4._M7->Fls(…)->Fls->FlsSector:

The address that SYS-IMG was supposed to write to was 0xc000=786432, connecting four 4096 sectors=# define HSE_ SYS_ IMAGE_ SIZE 0x4000:

0	FlsSector_0	0	FLS_EXT_SECTOR	1	16	4096	0
1	FlsSector_1	1	FLS_EXT_SECTOR	1	16	782336	4096
2	FlsSector_2	2	FLS_EXT_SECTOR	1	16	4096	786432
3	FlsSector_3	3	FLS_EXT_SECTOR	1	16	4096	790528
4	FlsSector_4	4	FLS_EXT_SECTOR	1	16	4096	794624
5	FlsSector_5	5	FLS_EXT_SECTOR	1	16	4096	798720
6	FlsSector_6	6	FLS_EXT_SECTOR	1	16	4096	802816
7	FlsSector_7	7	FLS_EXT_SECTOR	1	16	4096	806912

Now we need to configure the sector mapping for all sectors that need to be erased and written. We will modify the mapping as follows:

- Crypto (V3.0.
- CryptoDal (V
- EcuC (V3.0.2,
- EcuM (V3.0.2,
- Eep (V3.0.2, A
- Fls (V3.0.2, A
  - Fls
- Gpt (V3.0.2, A

FlsSector

Ind...	Name	Fl...	Fls Physical Sector	Fls ...	Fl...	Fls Se...	Fls Sect...	Fls Sector Hard...
0	FlsSector_0	0	FLS_EXT_SECTOR	1	16	4096	0	0x0
1	FlsSector_1	1	FLS_EXT_SECTOR	100	16	4096	4096	0x1000
2	FlsSector_2	2	FLS_EXT_SECTOR	4	16	4096	413696	0x65000
3	FlsSector_3	3	FLS_EXT_SECTOR	100	16	4096	430080	0x69000
4	FlsSector_4	4	FLS_EXT_SECTOR	152	16	4096	839680	0xcd000
5	FlsSector_5	5	FLS_EXT_SECTOR	4	16	4096	1462272	0x165000

FlsSector_1 is the HSE FW address, FlsSector_3 is the HSE FW backup address, FlsSector_5 is the address of SYS-IMG publish. (FlsSector_0/2/4 does not require operation, but the logical sector requires continuous addresses, which are also added to ensure consistency with the physical address configuration).

Note:

Due to the logic sector size of both S32DS IVT tools and Fls driver configurations being 4096B, corresponding modifications need to be made in the source code to prevent flash operation failures caused by align errors, as follows:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\generic\include\Bootloader.h

```
#define BL_ALIGN_4096B(x) BL_ALIGN_IMAGE_B(x, 12) //johnli add
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\BootloaderSpecific.c

```
static void Bl_SaveConfiguration(…)
{…
    uint32_t u32SysImageSizeAligned = BL_ALIGN_4096B(u32SysImageSize); //modified from
BL_ALIGN_10246B
```

In this way, the Fls driver can operate on the target address we need to write.

Bootloader_S32G2XX_ASR_4.4._M7->CryptoDal(…)->CryptoDal->Crypto Driver Abstraction Layer->CryptoDescriptors_0:

- Encryption Engine = CRYPTO // Encryption Engine used for encryption
- Encryption Algorithm= AES_256_CMAC // Encryption Algorithm for which the key is set
- Crypto Key = 0x85, 0xe3, 0xe6, 0x39, 0x1b, 0x13, 0xc2, 0xa3, 0x23, 0x69, 0xb2, 0x36, 0x80, 0x50, 0x4c, 0xbf, 0x1c, 0x12, 0x7b, 0x10, 0xd2, 0x36, 0x7f, 0xf6, 0x8c, 0x0c, 0x35, 0x6b, 0xa8, 0x86, 0x99, 0x0c //32X8=256bit. Encryption key value. Needs to be 16/32 coma separated values, for each bite.
- Crypto Key engine reference = /Crypto/Crypto/CryptoKeys/AES256CMAC_RAM_KEY // Encryption key engine internal reference, containing info about key storage. Used to generate Tag

It is recommended to store the key used for Tag generation in the RAM key directory, as it will no longer be needed or accessible after powering on and resetting. This key is only used before the secure boot takes effect

- Crypto Key alternate reference = /Crypto/Crypto/CryptoKeys/AES256CMAC_NVM_KEY // Encryption key engine alternate reference, used for key mirroring. For Tag verification, the backup key reference must be stored in the NVM key directory, as this key slot will be used to check if the application image has not been modified after any power on reset issued in the future

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto>CryptoKey->AES256CMAC_RAM_KEY:

- CryptoKeyId (0 -> 4294967295) =0 // Identifier of the Crypto Driver key.
- CryptoKeyTypeRef = /Crypto/Crypto/CryptoKeyTypes/Crypto_KT_AES256_CMAC // Refers to a pointer

in the CRYPTO to a CryptoKeyType. The CryptoKeyType provides the information about which key elements are contained in a CryptoKey.

->AES256CMAC_NVM_KEY:

- CryptoKeyId (0 -> 4294967295) =1
- CryptoKeyTypeRef =/Crypto/Crypto/CryptoKeyTypes/CryptoKeyType_0

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto>CryptoKeyType->Crypto_KT_AES256_CMAC:

- CryptoKeyElementRef= /Crypto/Crypto/CryptoKeyElements/Crypto_KE_AES256_CMAC_RAM_GENERATE //Refers to a Crypto Key Element, which holds the data of the Crypto Key Element.

-> CryptoKeyType_0:

- CryptoKeyElementRef= /Crypto/Crypto/CryptoKeyElements/ /Crypto/Crypto/CryptoKeyElements/Crypto_KE_AES256_CMAC_NVM_VERIFY

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto>CryptoKeyElement->Crypto_KE_AES256_CMAC_RAM_GENERATE->General:

- CryptoKeyElementFormat = CRYPTO_KE_FORMAT_BIN_OCTET // Defines the format for the key element. This is the format used to provide or extract the key data from the driver.
- CryptoKeyElementReadAccess= CRYPTO_RA_ALLOWED // Define the reading access rights of the key element.
- CryptoKeyElementSize (1 -> 4294967295)= 32 //Maximum size of the Crypto Key Element value, in bytes. Will be used by Crypto driver to reserve internal memory for those Crypto Key Elements that do not use a HSE key
- CryptoKeyElementWriteAccess = CRYPTO_WA_ALLOWED //Defines the writing access rights of the key element
- Use HSE Key = checked // Vendor specific: Enables or disables the usage of a HSE key.
- HSE Key Catalog Group Ref = /Crypto/Crypto/RamKeyGroup_AES // Vendor specific: The 'HSE Key Catalog Group Ref' identifies the key group where the key is located in the NVM or RAM key catalog.
- HSE Key Slot (0 -> 255) =0 // Vendor specific: Slot of the key inside the key group selected above in the 'HSE Key Catalog Group Ref'.
- HSE Key Counter (0 -> 268435455) = 0 Vendor specific: 28 bits counter used to prevent the rollback attacks on key. When updating a key value and attributes the new counter value must be greater than the current counter value. At counter saturation(0xFFFFFFF)the key cannot be updated anymore.

- HSE SMR Flags (0x0 -> 0xffffffff) = 0x0 // Vendor specific: A map of bits that define which Secure Memory Region (SMR), indexed from 0 to 31, should be verified before the key can be used. Set to zero means not used.

->HseKeyFlags:

- HseKeyFlag_2=USAGE_SIGN //Vendor specific: The key flag specifies the operations or restrictions that can be applied to a key.

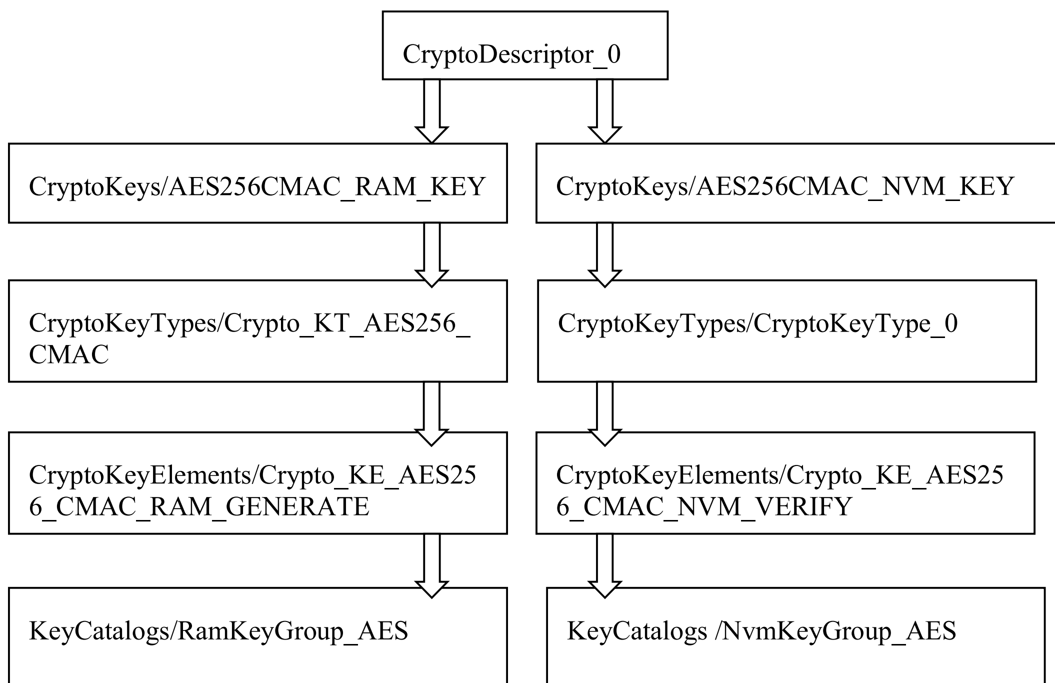
Same way to Analysis Crypto_KE_AES256_CMAC_NVM_VERIFY, difference on:

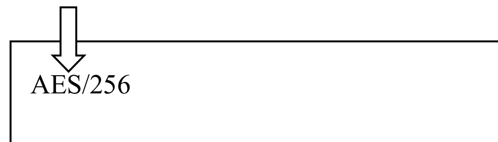
- HSE Key Catalog Group Ref= /Crypto/Crypto/NvmKeyGroup_AES
- HseKeyFlag_0/2/3= USAGE_ENCRYPT/VERIFY/ENCRYPT

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto>KeyCatalogs->NvmKeyGroup_AES

- Key Type = AES //Vendor specific: Specifies the key type. It provides information about the interpretation of key data
- Number of key slots (1 -> 256) = 10 // Vendor specific: The number of key slots in the current key group.
- Max key length in bits (1 -> 65535) = 256 //Vendor specific: The maximum length of the key (in bits). All stored keys can have keyBitLen lower or equal to MaxKeyBitLen
- Key Owner= OWNER_CUST // Vendor specific: Specifies the key group owner.

Conclusion:





Output the configuration codes:

C:\EB\tresos\workspace\Bootloader_S32G2XX_ASR_4.4_M7\output\src\CryptoDal_PbCfg.c

```
VAR(uint8, AUTOMATIC) CryptoDal_CryptoKey_0[32] = {0x85, 0xe3, 0xe6, 0x39, 0x1b, 0x13, 0xc2,
0xa3, 0x23, 0x69, 0xb2, 0x36, 0x80, 0x50, 0x4c, 0xbf, 0x1c, 0x12, 0x7b, 0x10, 0xd2, 0x36, 0x7f, 0xf6,
0x8c, 0x0c, 0x35, 0x6b, 0xa8, 0x86, 0x99, 0x0c};

const CryptoDal_ConfigType CryptoDal_Config[CRYPTODAL_MAX_CIPHERS] = {

  {

    32,    /* Key Length (bytes) */

    CRYPTODAL_AES_256_CMAC, /* Encryption Algorithm */

    CRYPTODAL_CRYPTO, /* Encryption Engine */

    CryptoDal_CryptoKey_0, /* Secret Key */

    0, /* Encryption engine internal reference */

    1 /* Encryption engine internal alternate reference */

  },
```

Other EB configuration:

Bootloader_S32G2XX_ASR_4.4._M7->CryptoDal(…)->CryptoDal->General:

- UseCrypto=checked // This flag indicates whether CryptoDal is enabled or not.
- EnableHashingService=checked // This flag indicates whether hash support is enabled or not.
- EnableSMRSupport =checked // This flag indicates whether Secure Memory region support is enabled or not
- EnableCMACGeneration=checked // This flag indicates Message Authentication generation is enabled or not
- EnableKeyCatalogsFormat=checked // This flag enables the key catalog formatting service.

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto> CryptoDriverObject-> CryptoDriverObject_0

->General:

- CryptoQueueSize (0 -> 4294967295) =6 // Size of the queue in the Crypto Driver. Defines the maximum number of jobs in the Crypto Driver Object queue. If it is set to 0, queueing is disabled in the Crypto Driver Object. Note: The node value will be used as the element number when declaring an array variable for the QUEUE feature. So the maximum value depends on the memory space of each platform.
- MU Instance = MU_0 // Vendor specific: Selects one of the MU (Messaging Units) instances available on the platform to use for communication with HSE.
- Algorithms Type = CRYPTO_SYMMETRIC_ALGORITHMS // Vendor specific: Determines if the crypto algorithms (primitives) associated with the Crypto Driver Object are symmetric or asymmetric.

->CryptoPrimitiveRef->:

0:/Crypto/Crypto/CryptoPrimitives/CryptoPrimitive_0

1:/Crypto/Crypto/CryptoPrimitives/CryptoPrimitive_1

Bootloader_S32G2XX_ASR_4.4._M7->Crypto(…)->Crypto-> CryptoPrimitives-> CryptoPrimitives-> CryptoPrimitive_0

- CryptoPrimitiveAlgorithmFamily = CRYPTO_ALGOFAM_AES
- CryptoPrimitiveAlgorithmMode = CRYPTO_ALGOMODE_NOT_SET
- CryptoPrimitiveAlgorithmSecondaryFamily = CRYPTO_ALGOFAM_NOT_SET
- CryptoPrimitiveService = MAC_VERIFY

-> CryptoPrimitive_1

- CryptoPrimitiveAlgorithmFamily = CRYPTO_ALGOFAM_AES
- CryptoPrimitiveAlgorithmMode = CRYPTO_ALGOMODE_NOT_SET
- CryptoPrimitiveAlgorithmSecondaryFamily = CRYPTO_ALGOFAM_NOT_SET
- CryptoPrimitiveService = MAC_GENERATE

4.2 EB output codes

The parts related to Secure Boot in the Bootloader code startup process are as follows:

```
Startup.s:
\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\m7
|-> SystemInit // Initialize the system (MPU, interrupts)
|->main:
\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\generic\src\bootloader.c
|   |->SysDal_Init
```

```
|  |  |->SysDal_StartUpInit
|  |  |  |->SysDal_McuPlatformInitSeq
|  |  |  |  |->Mcu_Init，Mcu_SetMode，Mcu_InitClock(McuClockSettingConfig_0);，Mcu_DistributePllClock();// clock initial
|  |  |  |->SysDal_WakeUpInit
|  |  |  |  |->InitBlockOneCallout
|  |  |  |  |  |->// Peripheral driver initialization
…
|  |  |  |  |  |-> CryptoDal_Init(&CryptoDal_Config[0]);
|  |  |  |  |  |  |-> CryptoDal_Crypto_Init
|  |  |  |  |  |  |  |-> Crypto_Init(NULL_PTR); // /* Initialize hardware module */
|  |  |  |  |  |  |  |  |->Crypto_Ipw_Init //#define    Crypto_Ipw_Init(partitionId) \
        (Crypto_Hse_Init(partitionId))
|  |  |  |  |  |  |  |  |  |->Hse_Ip_Init
|  |  |  |  |  |  |  |  |->Crypto_HandleNvramInfo
|  |  |  |  |  |  |  |  |->Crypto_Util_InitJobQueues
|  |  |  |  |  |  |  |->Crypto_Exts_FormatKeyCatalogs  /* Format key catalogs */
|  |->Bl_ConfigureSecureBoot
|  |  |-> if (E_NOT_OK == Bl_IsSecureBootActive())//Check whether Secure Boot was configured previously.
/* Secure Boot is not active */ // So if the Secure Boot mode was not set in the previous IVT, but the project has enabled Secure Boot, it is considered as the first startup. Therefore, after executing the signature, installing SMR, exporting SYS Image, and modifying the IVT Secure Boot mode configuration, all subsequent starts will be Secure Boot.
      Status = E_NOT_OK;
|  |  |->Bl_GetBootloaderParams(&u32BootAppFlashAddr, &u32BootAppRamAddr,
                          &u32BootAppSize); // Read the relevant information of the system startup image from the IVT header
|  |  |->CryptoDal_GenerateCmac(
        (uint8_t *) u32BootAppFlashAddr, &Bl_BootAppTag[0],
        BL_SEC_BOOT_KEY_INDEX, u32BootAppSize, &u32TagSize);// Generate CMAC signature for this image
BL_SEC_BOOT_KEY_INDEX=0, so it used Crypto_KE_AES256_CMAC_RAM_GENERATE。
|  |  |  |->CryptoDal_pGlobalConfigPtr[u8KeyIndex].eEncryptionEngine;


|  |  |  |->CryptoDal_Crypto_GenerateCmac(pInputData, pTag, u8KeyIndex, u32InputDataSize, pTagSize);
// The following call first import the Key
|  |  |  |  |->Crypto_KeyElementSet(u8CryptoKeyIndex, CRYPTO_KEY_MATERIAL_U32, CryptoDal_Crypto_pGlobalConfigPtr[u8KeyIndex].u8CryptoKey,
```

```
                                CryptoDal_Crypto_pGlobalConfigPtr[u8KeyIndex].u8EncryptionKeyLen))
|  |  |  |  |   |->Crypto_Ipw_ImportKey=Crypto_Hse_ImportKey
|  |  |  |  |   |   |->Crypto_Hse_EccLoadPlainPairKey
pHseSrvDescriptor->srvId = HSE_SRV_ID_IMPORT_KEY;
         pImportKeyReq->pKeyInfo        =
HSE_PTR_TO_HOST_ADDR(&Crypto_Hse_apMuState[u8MuInstance]->HseKeyInfo);
         pImportKeyReq->targetKeyHandle =
Crypto_aKeyElementList[u32KeyMaterialKeyElemIdx].u32HseKeyHandle;

         /* Only for encrypted ECC is needed*/
         pImportKeyReq->cipher.cipherKeyHandle     = HSE_INVALID_KEY_HANDLE;
         pImportKeyReq->keyContainer.authKeyHandle = HSE_INVALID_KEY_HANDLE;

         RetVal = Crypto_Hse_SendMsg(u8MuInstance, u8MuChannel, pHseSrvDescriptor, NULL_PTR);
|  |  |  |   |->Crypto_KeySetValid(u8CryptoKeyIndex))
if (CRYPTODAL_AES_256_CMAC == eAlgorithm)
            {
                CryptoDal_Crypto_CmacJobPrimitiveInfo.cryIfKeyId                = u8CryptoKeyIndex;
                CryptoDal_Crypto_CmacJob.jobPrimitiveInputOutput.inputPtr       = pInputData;
                CryptoDal_Crypto_CmacJob.jobPrimitiveInputOutput.inputLength    = u32InputDataSize;

                CryptoDal_Crypto_CmacJob.jobPrimitiveInputOutput.outputPtr      = pTag;
                CryptoDal_Crypto_CmacJob.jobPrimitiveInputOutput.outputLengthPtr = pTagSize;
                eRetVal = Crypto_ProcessJob(CRYPTODAL_CRYPTO_DRIVER_ID, &CryptoDal_Crypto_CmacJob);
            }///* Crypto driver object used */
#define CRYPTODAL_CRYPTO_DRIVER_ID          (0U)
/* --- Structure of the job to be passed to Crypto driver, requesting CMAC with specific keys
---------------------------------------------------------------------------- */
static VAR(Crypto_JobType, AUTOMATIC) CryptoDal_Crypto_CmacJob =
{
   1U,                                    /* jobId                  - Identifier for the job structure */
   CRYPTO_JOBSTATE_IDLE,                  /* jobState               - Determines the current job state */
   {
      NULL_PTR,                           /* inputPtr               - Pointer to the input data. */
      0,                                  /* inputLength            - Contains the input length in bytes. */
```

```
    NULL_PTR,                    /* secondaryInputPtr        - Pointer to the secondary input data (for MacVerify, SignatureVerify). */
    0U,                          /* secondaryInputLength     - Contains the secondary input length in bytes. */
    NULL_PTR,                    /* tertiaryInputPtr         - Pointer to the tertiary input data (for MacVerify, SignatureVerify). */
    0U,                          /* tertiaryInputLength      - Contains the tertiary input length in bytes. */
    NULL_PTR,                    /* outputPtr                - Pointer to the output data. */
    0U,                          /* outputLengthPtr          - Holds a pointer to a memory location containing the output length in bytes. */
    NULL_PTR,                    /* secondaryOutputPtr       - Pointer to the secondary output data. */
    NULL_PTR,                    /* secondaryOutputLengthPtr - Holds a pointer to a memory location containing the secondary output length in bytes. */
    0U,                          /* input64                  - Versatile input parameter */
    NULL_PTR,                    /* verifyPtr                - Output pointer to a memory location holding a Crypto_VerifyResultType */
    NULL_PTR,                    /* output64Ptr              - Output pointer to a memory location holding an uint64. */
    CRYPTO_OPERATIONMODE_SINGLECALL          /* mode         - Indicator of the mode(s)/operation(s) to be performed */
  },
  &CryptoDal_Crypto_CmacJobPrimitiveInfo,         /* jobPrimitiveInfo     - Pointer to a structure containing further information,
                                                     depends on the job and the crypto primitive */
  &CryptoDal_Crypto_CmacInfoType,                 /* jobInfo              - Pointer to a structure containing further information,
                                                     depends on the job and the crypto primitive */
  NULL_PTR                                        /* jobRedirectionInfoRef - Pointer to a structure containing further information
                                                     on the usage of keys as input and output for jobs. */
};
//Install SMR/CR
|  |  |  |->Bl_ConfigureBootloaderSMR(&SMR_CR_Config, u32BootAppFlashAddr,
                   u32BootAppRamAddr, u32BootAppSize,
                   &Bl_BootAppTag[0], &u32TagSize);
{
  pSMR_CR_Config->u32FlashStorageAddr = u32AppFlashAddress;
  pSMR_CR_Config->u32RamDestAddr = u32AppRamAddr;
  pSMR_CR_Config->u32AppSize = u32AppSize;
```

```
    pSMR_CR_Config->pTag = pTag;
    pSMR_CR_Config->pTagSize = pTagSize;
    pSMR_CR_Config->u8KeyIndex = BL_SEC_BOOT_KEY_INDEX; //
/* Key index in the cryptodal descriptors list */
#define BL_SEC_BOOT_KEY_INDEX 0
    pSMR_CR_Config->u8EntryIndex = 1;
    pSMR_CR_Config->u8CoreID = HSE_APP_CORE0; //#define HSE_APP_CORE0        ((hseAppCore_t)0U)    /**<
@brief Core0 */
}
|  |  |  |->CryptoDal_SMR_CR_Install(&SMR_CR_Config);
|  |  |  |   |->CryptoDal_Crypto_SMR_CR_Install
/* Fetch internal decryption engine key reference */
|  |  |  |   |     |->u8CryptoKeyIndex =
CryptoDal_Crypto_pGlobalConfigPtr[pSMR_CR_Config->u8KeyIndex].u8EncryptionKeyAlternateRef;
/* Register key in using the alternate slot. This descriptor shall reside in NVM for verification usage */
|  |  |  |   |     |->Crypto_KeyElementSet(u8CryptoKeyIndex, CRYPTO_KEY_MATERIAL_U32,
CryptoDal_Crypto_pGlobalConfigPtr[pSMR_CR_Config->u8KeyIndex].u8CryptoKey,
                        CryptoDal_Crypto_pGlobalConfigPtr[pSMR_CR_Config->u8KeyIndex].u8EncryptionKeyLen);
|  |  |  |   |     |->MemLib_MemSet(&SmrEntry, 0, sizeof(SmrEntry));
/* Configure SMR entry */
    SmrEntry.pSmrSrc                              = (HOST_ADDR)pSMR_CR_Config->u32FlashStorageAddr;
    SmrEntry.pSmrDest                             = (HOST_ADDR)pSMR_CR_Config->u32RamDestAddr;
    SmrEntry.smrSize                              = pSMR_CR_Config->u32AppSize;
    SmrEntry.checkPeriod                          = 0;
    SmrEntry.configFlags                          = HSE_SMR_CFG_FLAG_QSPI_FLASH; //#define
HSE_SMR_CFG_FLAG_QSPI_FLASH         ((hseSmrConfig_t)0x0U)
    SmrEntry.authKeyHandle                        = CRYPTODAL_SMR_CMAC_KEY_HANDLE(u8CryptoKeyIndex);//
#define HSE_KEY_CATALOG_ID_NVM       ((hseKeyCatalogId_t)1U)    /**< @brief NVM key catalog */
    SmrEntry.authScheme.macScheme.macAlgo             = HSE_MAC_ALGO_CMAC;   #define
HSE_MAC_ALGO_CMAC           ((hseMacAlgo_t)0x11U)
    SmrEntry.authScheme.macScheme.sch.cmac.cipherAlgo  = HSE_CIPHER_ALGO_AES;// #define
HSE_CIPHER_ALGO_AES         ((hseCipherAlgo_t)0x10U)
    SmrEntry.pInstAuthTag[0]                      = 0;
    SmrEntry.pInstAuthTag[1]                      = 0;

    /* Create a SMR install request */ //server request
```

```
  SmrInstallRequest.accessMode                    = HSE_ACCESS_MODE_ONE_PASS;//HSE 访问模式
  SmrInstallRequest.entryIndex                  = pSMR_CR_Config->u8EntryIndex;
  SmrInstallRequest.pSmrEntry                    = (HOST_ADDR)&SmrEntry;
  SmrInstallRequest.pSmrData                     = (HOST_ADDR)pSMR_CR_Config->u32FlashStorageAddr;
  SmrInstallRequest.smrDataLength                  = pSMR_CR_Config->u32AppSize;
  SmrInstallRequest.pAuthTag[0]                   = (HOST_ADDR)pSMR_CR_Config->pTag;
  SmrInstallRequest.authTagLength[0]               = *((uint32_t*)pSMR_CR_Config->pTagSize);

  MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));
  HseSrvDescriptor.srvId                  = HSE_SRV_ID_SMR_ENTRY_INSTALL; // #define
HSE_SRV_ID_SMR_ENTRY_INSTALL      ((hseSrvId_t)(HSE_SRV_VER_0 | 0x00000501UL)) //SMR install server
ID
  HseSrvDescriptor.hseSrv.smrEntryInstallReq = SmrInstallRequest;
|  |  |  |  |     |->CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
/* Configure a Core Reset Entry for Advanced Secure Boot */
  CoreResetEntry.coreId = pSMR_CR_Config->u8CoreID;
  CoreResetEntry.preBootSmrMap = (1UL << pSMR_CR_Config->u8EntryIndex);
  CoreResetEntry.pPassReset = pSMR_CR_Config->u32RamDestAddr;
  CoreResetEntry.crSanction = HSE_CR_SANCTION_KEEP_CORE_IN_RESET; //#define
HSE_CR_SANCTION_KEEP_CORE_IN_RESET   ((hseCrSanction_t)0x7455U) //CR The sanction method is core in
reset
  CoreResetEntry.startOption = HSE_CR_AUTO_START; #define HSE_CR_AUTO_START
((hseCrStartOption_t)0x35A5U)

  /* Create the request to HSE using service descriptor */
  HseSrvDescriptor.srvId                            = HSE_SRV_ID_CORE_RESET_ENTRY_INSTALL; #define
HSE_SRV_ID_CORE_RESET_ENTRY_INSTALL ((hseSrvId_t)(HSE_SRV_VER_0 | 0x00000503UL))//CR install
server
  HseSrvDescriptor.hseSrv.crEntryInstallReq.crEntryIndex  = 0;
  HseSrvDescriptor.hseSrv.crEntryInstallReq.pCrEntry      = HSE_PTR_TO_HOST_ADDR(&CoreResetEntry);

//new sys-img publish
|  |  |  |   |->CryptoDal_GetSysImage(
            &Bl_HseSysImage[0], &u32SysImageOffSet, &u32SysImageSize)
@details  Shall be used to retrieve the System Image from the HSE subsystem. Whenever a new operation has been
registered in the
```

HSE (Non volatile key, OTFAD configuration, SMR/CR configuration), a system configuration is being kept in HSE internal RAM.

Upon this request, the HSE will publish the system image using the device encryption key.

```
| | | | |    |->CryptoDal_Crypto_GetSysImage(pImage, pSysImageOffset, pHseImageSize);
  /* Clear previous request */
   MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));
   HseSrvDescriptor.srvId                          = HSE_SRV_ID_GET_SYS_IMAGE_SIZE;
   HseSrvDescriptor.hseSrv.getSysImageSizeReq.pSysImageSize = (HOST_ADDR)&u32HSEImageSize;
   if (E_OK == eRetVal)
   {
      | | | | | |   |->CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor);
/* Clear previous request */
   MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));

   HseSrvDescriptor.srvId                          = HSE_SRV_ID_PUBLISH_SYS_IMAGE;
   HseSrvDescriptor.hseSrv.publishSysImageReq.publishOptions = HSE_PUBLISH_ALL_DATA_SETS;
   HseSrvDescriptor.hseSrv.publishSysImageReq.pPublishOffset = (HOST_ADDR)(pSysImageOffset);
   HseSrvDescriptor.hseSrv.publishSysImageReq.pBuffLength    = (HOST_ADDR)(pHseImageSize);
   HseSrvDescriptor.hseSrv.publishSysImageReq.pBuff          = (HOST_ADDR)(pImage);

   if (E_OK == eRetVal)
   {
        | | | | | |   |->if (HSE_SRV_RSP_OK != CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
| | | |   |->Bl_SaveConfiguration(&Bl_HseSysImage[0], BL_SYS_IMAGE_STORAGE_ADDR,
                      u32SysImageOffSet, u32SysImageSize); ///* HSE system image storage address */
#define BL_SYS_IMAGE_STORAGE_ADDR 786432   EB configure it.
| | | | |    |->Fls_Write(u32SysImageStorageAddr + u32SysImageOffset,
         (const uint8 *) pSysImage, u32SysImageSizeAligned);
/* Update IVT configuration */
   MemLib_MemCpy(&Bl_FlashMirror[0], IVT_ADDR, QSPI_SECTOR_SIZE);
   pIvtUpdated = (Bl_ImageVectorTableType *) &Bl_FlashMirror[0];
   pIvtUpdated->u32BootCfgWord = IVT_BCW_SEC_BOOT_CONFIG_MASK; // Reset Secure Boot enable for IVT here
   pIvtUpdated->u32SysImageAddr = u32SysImageStorageAddr;
   pIvtUpdated->u32ExtFlashType = IVT_ADDR;
```

```
    pIvtUpdated->u32FlashPageSize = QSPI_SECTOR_SIZE;
|  |  |  |  |     |->Fls_Erase(IVT_ADDR, QSPI_SECTOR_SIZE);
|  |  |  |  |     |->Fls_Write(IVT_ADDR, (const uint8 *) &Bl_FlashMirror[0], QSPI_SECTOR_SIZE);
|  |  |  |   |->Mcu_PerformReset();
```

Based on the above Secure Boot code analysis, the secure boot execution process includes the following steps:

1. After the first cold start (Image is written to the QSPI flash along with HSE firmware, DCD, QSPI header configuration data, and IVT header), the boot loader performs the following steps when running in a non secure environment:
   - Parse the IVT to obtain the existing configuration of the Bootloader (storage address, start address, and image size).
   - By generating authentication tags and using the key configured in CryptoDal to authenticate the boot image, the AES-256-CMAC algorithm is used.
   - The above keys are also stored in the HSE non volatile key catalog for use with reset (POR) after a power outage.
   - The secure memory area SMR is registered in the HSE system image, including image parameters (key, storage address, image size, start address, target boot kernel, and previously calculated authentication label).
   - The HSE system image Sys Image, which contains a non-volatile key catalog and secure memory area configuration, is exported to RAM memory for applications to save to flash memory.
   - The system image Sys Image is saved to the flash memory of the EB configuration address. This ensures that the previously registered SMR and key will be loaded and executed by HSE on the next reboot.
   - IVT updates with new configuration parameters (secure boot enabled and system image address updated), as shown in the following figure, and issues a power on reset (POR). In addition, users can also observe in the data dump below that the IVT header residing in flash memory has been modified (the BOOTSEQ bit at 0x28 is now turned on, and the HSE system image address has been updated).
2. If the previous operation is successful, on the next reset, HSE will check the authenticity of the Bootloader image and launch the corresponding configured kernel. If any application image bytes are modified, the kernel will remain reset and booted, and the process will fail.

5 Customization 1：HSE FW update

This chapter explains how to add the HSE FW update function.

5.1 Codes development

1. Implement a function to read HSE FW related information from the IVT header:

Function call:：
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
        uint32_t u32FwBlueImageSize = HSE_FW_IMAGE_SIZE;
        uint32_t u32FwPinkImageSize = HSE_FW_IMAGE_SIZE;
       Bl_GetHSEFwParams(&u32HseFwImageFlashAddr, &u32HseBackupFwImageFlashAddr);
      static void Bl_GetHSEFwParams(uint32_t *pHseFwAddr, uint32_t *pHseBackupFwAddr)
{
       *pHseFwAddr=pIVTConfig->u32HSEAddr;
        *pHseBackupFwAddr=pIVTConfig->u32HSEBckAddr;
}
```

static Bl_ImageVectorTableType *pIVTConfig = IVT_ADDR； // /* Image Vector Table configuration address */#define IVT_ADDR 0x0 IVT header address is QSPI NOR AHB memory start address: 0x0.

Add HSE_FW_IMAGE_SIZE macro as follow information:

15.2.3 Image sizes

The below table specifies the maximum image sizes for the HSE firmware.

Table 135: Maximum image sizes

HSE image	HSE_H image size	HSE_M i
FW-IMG (as provided by NXP)	357KB **Note**: It is recommended to allocate 400KB (or more) in flash for each primary and back-up image	261KB **Note**: It i (or more) back-up
SYS-IMG	44KB / 48KB on S32ZE **Note:** It is recommended to allocate 48KB in flash.	44KB **Note:** It i in flash

```
#define HSE_FW_IMAGE_SIZE 0x64000 //johnli for 400KB
```

2. Implement HSE FW update function

```
/* HSE FW  Image */
static uint8_t Bl_HseFwImage[HSE_FW_IMAGE_SIZE]; //johnli add  prepared a memory buffer to store FW.
```

Function call:

```
status = CryptoDal_GetFwImage(&Bl_HseFwImage[0], &u32HseFwImageFlashAddr， &u32FwPinkImageSize, &u32FwBlueImageSize);
```

Dal function:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_GetFwImage(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pImage,
                              VAR(uint32, AUTOMATIC) u32FwPinkImageFlashAddress,
                              P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwPinkImageSize,
                              P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwBlueImageSize)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
      eRetVal = CryptoDal_Crypto_GetFwImage(pImage, u32FwPinkImageFlashAddress, pFwPinkImageSize, pFwBlueImageSize);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implementation function:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_GetFwImage(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pImage,
                              VAR(uint32, AUTOMATIC) u32FwPinkImageFlashAddress,
                              P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwPinkImageSize,
                              P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwBlueImageSize)
{
   VAR(hseSrvDescriptor_t, CRYPTODAL_VAR) HseSrvDescriptor;
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_OK;
   VAR(uint32, AUTOMATIC) u32FwBlueImageSize = (*pFwBlueImageSize);
   VAR(uint32, AUTOMATIC) u32FwPinkImageSize = (*pFwPinkImageSize);
   VAR(uint32, AUTOMATIC) u32FwPinkImageFlashAddress = (*pFwPinkImageFlashAddress);

   // Obtain the size of FW. Note that although this function is implemented in code, in fact, it is only necessary to transfer the size parameter to 0 when converting the running FW, so this function is only useful when converting the non running FW.
   // u32FwPinkImageSize=GetHseFwSize((uintptr_t)u32FwPinkImageFlashAddress);
        // Update FW from pink to blue sends FW updated service, takes pink FW as input, and outputs blue FW to memory.The address of pink FW is the address of QSPI NOR AHB memory.
      //update FW from pink to blue
        //get the pink FW size
        /* Clear previous request */
   MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));

        /* Fill the service descriptor */
   HseSrvDescriptor.srvId                      = HSE_SRV_ID_FIRMWARE_UPDATE;
   HseSrvDescriptor.hseSrv.firmwareUpdateReq.pInFwFile = HSE_PTR_TO_HOST_ADDR(&u32FwPinkImageFlashAddress);
```

```
#if 1
   HseSrvDescriptor.hseSrv.firmwareUpdateReq.inFwFileLength = 0x0;
#else
   HseSrvDescriptor.hseSrv.firmwareUpdateReq.inFwFileLength = u32FwPinkImageSize;
#endif
   HseSrvDescriptor.hseSrv.firmwareUpdateReq.pOutFwBuffer    = HSE_PTR_TO_HOST_ADDR(pImage);
   HseSrvDescriptor.hseSrv.firmwareUpdateReq.pFwBufferLength         =
HSE_PTR_TO_HOST_ADDR(pFwBlueImageSize);
   if (E_OK == eRetVal)
   {
      if (HSE_SRV_RSP_OK != CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
                  {
                           eRetVal = E_NOT_OK;
                  }
   }

   return eRetVal;
}
```

3. Implement the function of updating HSE FW image.

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
//write blue image to hse main fw
Bl_SaveFwConfiguration(&Bl_HseFwImage[0],u32HseFwImageFlashAddr,u32FwBlueImageSize);
```

   Final implementation function:

```
static void Bl_SaveFwConfiguration(uint8_t *pFwImage,
                                   uint32_t u32FwImageStorageAddr,
                                   uint32_t u32FwImageSize)
{
   uint32_t u32FwImageSizeAligned = BL_ALIGN_4096B(u32FwImageSize);
        Fls_Erase(u32FwImageStorageAddr,
            u32FwImageSizeAligned);
   while (MEMIF_IDLE != Fls_GetStatus())
   {
      Fls_MainFunction();
   }
   Fls_Write(u32FwImageStorageAddr,
            (const uint8 *) pFwImage, u32FwImageSizeAligned);
   while (MEMIF_IDLE != Fls_GetStatus())
   {
      Fls_MainFunction();
   }
  …
}
```

   Header file：

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\include\CryptoDal.h

```
    FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_GetFwImage(P2VAR(uint8, AUTOMATIC,
CRYPTODAL_CONST) pImage,
                                  VAR(uint32, AUTOMATIC) u32FwPinkImageFlashAddress,
```

```
                                P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwPinkImageSize,
                                P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwBlueImageSize);
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\include\CryptoDal_Crypto.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_GetFwImage(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pImage,
                                VAR(uint32, AUTOMATIC) u32FwPinkImageFlashAddress,
                                P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwPinkImageSize,
                                P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) pFwBlueImageSize);
```

Note:

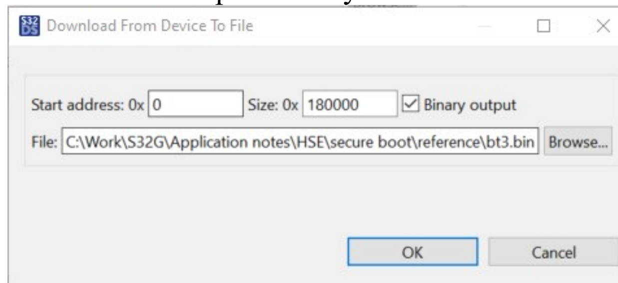
It is recommended to back up the original pink image to the HSE FW backup image flash address after updating the pink image to blue image. The customer can debug it by himself.

5.2 Testing

After the Secure Boot succeeds, reset the RDB2 board to the download mode, and use flash tools to read the image:

Click: Upload target and algorithm to hardware,

Click: Download from device to file Export Binary:



Then compare with the original image, check the address 0x1000, and you can see that the original pink image starts with D4, while the new blue image starts with D6, and it can be used to secure boot normally.

6 Customization 2：HSE OTP Attribute Setting

This chapter consider to set the follow OTP fuse:

- ADKP
- Authorization mode， from default pass-word to challenge-response(Seldom used by customers)
- Lifecycle
- IVTAuth

Because the above OTP operations are one-time, we only compile code and do not do actual tests and can only be simulated by Lifecycle. ADKP needs to actually burn before making IVT signatures. **Be very careful!!!**

6.1 Code Development

1. Implement OTP Attribute setting/reading API

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_SetAttribute(VAR(hseAttrId_t, AUTOMATIC) attrId,
                    VAR(uint32_t, AUTOMATIC) attrLen,
                    P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAttr
                                                    )
{
   VAR(hseSrvDescriptor_t, CRYPTODAL_VAR) HseSrvDescriptor;
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_OK;
    /* Clear previous request */
   MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));
    /* Fill the service descriptor */
     HseSrvDescriptor.srvId                                = HSE_SRV_ID_SET_ATTR;
     HseSrvDescriptor.hseSrv.setAttrReq.attrId             = attrId;
     HseSrvDescriptor.hseSrv.setAttrReq.attrLen  = attrLen;
     HseSrvDescriptor.hseSrv.setAttrReq.pAttr              = HSE_PTR_TO_HOST_ADDR (pAttr);
    if (E_OK == eRetVal)
   {
      if (HSE_SRV_RSP_OK != CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
                {
                          eRetVal = E_NOT_OK;
                }
   }
}

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_GetAttribute(VAR(hseAttrId_t, AUTOMATIC) attrId,
                    VAR(uint32_t, AUTOMATIC) attrLen,
                    P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAttr
                                                    )
```

```
{
    VAR(hseSrvDescriptor_t, CRYPTODAL_VAR) HseSrvDescriptor;
    VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_OK;
     /* Clear previous request */
    MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));
     /* Fill the service descriptor */
      HseSrvDescriptor.srvId                                    = HSE_SRV_ID_GET_ATTR;
      HseSrvDescriptor.hseSrv.setAttrReq.attrId                 = attrId;
      HseSrvDescriptor.hseSrv.setAttrReq.attrLen  = attrLen;
      HseSrvDescriptor.hseSrv.setAttrReq.pAttr                  = HSE_PTR_TO_HOST_ADDR (pAttr);
     if (E_OK == eRetVal)
     {
        if (HSE_SRV_RSP_OK != CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
                    {
                                  eRetVal = E_NOT_OK;
                    }
     }
}
```

2. Implement ADKP burning function：

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
/*
    The application debug key/password (ADK/P) to be programmed - 128-bits
    Used by `HSE_ProgramAdkp` function to program ADK/P in fuses
*/
static uint8_t applicationDebugKeyPassword[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};
CryptoDal_ProgAdkp(&applicationDebugKeyPassword[0]);
```

   Dal function:

   C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_ProgAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
       eRetVal = CryptoDal_Crypto_ProgAdkp(pAdkp);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_ProgAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp)
{
   return CryptoDal_Crypto_SetAttribute(HSE_APP_DEBUG_KEY_ATTR_ID,sizeof(hseAttrApplDebugKey_t),pAdkp);
}
```

3. Implement the Auth mode to CR function:

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
CryptoDal_SetAuthModeToCR();
```

Dal function:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SetAuthModeToCR()
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
       eRetVal = CryptoDal_Crypto_SetAuthModeToCR();
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_SetAuthModeToCR(void)
{
   hseAttrDebugAuthMode_t debugAuthMode;
   debugAuthMode=HSE_DEBUG_AUTH_MODE_CR;
      return
CryptoDal_Crypto_SetAttribute(HSE_DEBUG_AUTH_MODE_CR,sizeof(hseAttrDebugAuthMode_t),&debugAuthMode);
}
```

4. Implement the lifecycle advanced function:

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
CryptoDal_AdvanceLifecycle(HSE_LC_OEM_PROD);
CryptoDal_AdvanceLifecycle(HSE_LC_IN_FIELD);
```

Dal function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_AdvanceLifecycle(hseAttrSecureLifecycle_t targetLifeCycle)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
       eRetVal = CryptoDal_Crypto_AdvanceLifecycle(targetLifeCycle);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE)
CryptoDal_Crypto_AdvanceLifecycle(VAR(hseAttrSecureLifecycle_t, AUTOMATIC) u32targetLifecycle)
{
      return
CryptoDal_Crypto_SetAttribute(HSE_SECURE_LIFECYCLE_ATTR_ID,sizeof(hseAttrSecureLifecycle_t),&u32targetLifecycle);
```

```
}
```

5. Implement the IVT Auth burning function:

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
CryptoDal_SetIvtAuth();
```

   Dal function:

   C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SetIvtAuth()
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
      eRetVal = CryptoDal_Crypto_SetIvtAuth();
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

   Final implement function:

   C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
/* Enable IVT/DCD/ST-DCD authentication */

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_SetIvtAuth(void)

{

    hseAttrConfigBootAuth_t ivtAuth;

     /* Enable IVT/DCD/ST-DCD signature verification by BootROM. */

   ivtAuth = HSE_IVT_AUTH;

    return
CryptoDal_Crypto_SetAttribute(HSE_ENABLE_BOOT_AUTH_ATTR_ID,sizeof(hseAttrSecureLifecycle_t),&ivtAuth);

}
```

6. The main function call:

```
#define HSE_PROG_OTP_ATTR 1

#if defined(HSE_PROG_OTP_ATTR)

volatile uint8_t gProgramAdkp = 0U;

volatile uint8_t gSetDebugAuthModeToChallengeResponse = 0U;

volatile uint8_t gAdvanceLifecycleToOemProd = 0U;

volatile uint8_t gAdvanceLifecycleToInField = 0U;
```

```
volatile uint8_t gSetIvtAuth = 0U;
/*
    The application debug key/password (ADK/P) to be programmed - 128-bits
    Used by `HSE_ProgramAdkp` function to program ADK/P in fuses
*/
static uint8_t applicationDebugKeyPassword[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};
#endif
StatusType Bl_ConfigureSecureBoot(void)
{…
        if (E_OK == status)
        {
                #if defined(HSE_PROG_OTP_ATTR)
                                //program the OTP attribute
                                /* Program Application debug key/password example. Current Lifecycle must be Customer Delivery */
    if(0xDA == gProgramAdkp)
    {
        CryptoDal_ProgAdkp(&applicationDebugKeyPassword[0]);
    }

    /* Set Debug Authorization mode to Challenge-Response (Default mode is password-based) */
    if(0xDA == gSetDebugAuthModeToChallengeResponse)
    {
        CryptoDal_SetAuthModeToCR();
    }

    /* Advances Lifecycle to OEM Production Lifecycle */
    if(0xDA == gAdvanceLifecycleToOemProd)
    {
        CryptoDal_AdvanceLifecycle(HSE_LC_OEM_PROD);
    }
```

```
/* Advances Lifecycle to Infield Lifecycle */
if(0xDA == gAdvanceLifecycleToInField)
{
    CryptoDal_AdvanceLifecycle(HSE_LC_IN_FIELD);
}


        /* Enable IVT/DCD/ST-DCD authentication */
        if(0xDA == gSetIvtAuth)
        {
                CryptoDal_SetIvtAuth();
        }
        #endif
        //end
…
```

Header file define:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\include\CryptoDal.h

```
#include "hse_srv_attr.h" //johnli for compiling
…
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_ProgAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pAdkp);
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SetAuthModeToCR(void);
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_AdvanceLifecycle(hseAttrSecureLifecycle_t targetLifeCycle);
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_CheckLifecycle(P2VAR(hseAttrSecureLifecycle_t, AUTOMATIC, CRYPTODAL_CONST) ptargetLifeCycle);
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SetIvtAuth(void);
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\include\CryptoDal_Crypto.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_ProgAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp);

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_SetAuthModeToCR(void);

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_AdvanceLifecycle(VAR(hseAttrSecureLifecycle_t, AUTOMATIC) u32targetLifecycle);

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_CheckLifecycle(P2VAR(uint32, AUTOMATIC, CRYPTODAL_CONST) ptargetLifecycle);

FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_SetIvtAuth(void);
```

6.2 Simulation test

Because OTP is one-time, the above code is only compiled, not tested.

The following simulation test simulates the advance of lifecycle and develops the following test code:

Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
#define HSE_PROG_OTP_ATTR_TEST 1
#if defined(HSE_PROG_OTP_ATTR_TEST)
volatile uint8_t gAdvanceLifecycleToSimulOemProd = 0xDA;
volatile uint8_t gAdvanceLifecycleToSimulInField = 0xDA;
hseAttrSecureLifecycle_t gLifecycle=HSE_LC_CUST_DEL
#endif
…
    StatusType Bl_ConfigureSecureBoot(void)
    {…
       if (E_NOT_OK == Bl_IsSecureBootActive())
       {…}
       else
         {
    #if defined(HSE_PROG_OTP_ATTR_TEST)
                      /* Advances Lifecycle to simulated OEM Production Lifecycle, do not prog fuse, reset recovery */
      if(0xDA == gAdvanceLifecycleToSimulOemProd)
      {
          CryptoDal_AdvanceLifecycle(HSE_LC_SIMULATED_OEM_PROD);
      }
        CryptoDal_CheckLifecycle(&gLifecycle);
      /* Advances Lifecycle to simulated Infield Lifecycle  do not prog fuse, reset recovery*/
      if(0xDA == gAdvanceLifecycleToSimulInField)
      {
          CryptoDal_AdvanceLifecycle(HSE_LC_SIMULATED_IN_FIELD);
      }
      CryptoDal_CheckLifecycle(&gLifecycle);
    #endif
```

CryptoDal_CheckLifecycle function implement: C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE)
CryptoDal_CheckLifecycle(P2VAR(hseAttrSecureLifecycle_t, AUTOMATIC, CRYPTODAL_CONST)
ptargetLifeCycle)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
       eRetVal = CryptoDal_Crypto_CheckLifecycle(ptargetLifeCycle);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_CheckLifecycle(P2VAR(uint32,
AUTOMATIC, CRYPTODAL_CONST) ptargetLifecycle)
{
      return
CryptoDal_Crypto_GetAttribute(HSE_SECURE_LIFECYCLE_ATTR_ID,sizeof(hseAttrSecureLifecycle_t),ptargetLifecycle);
}
```

Use lauterbach for tracking. After setting the lifecycle to HSE_ LC_ SIMULATED_ OEM_ PROD and HSE_ LC_ SIMULATED_ IN_ FIELD, the read lifecycle is HSE_ LC_ OEM_ PROD and HSE_ LC_ IN_ FIELD.

Note that the above functions run in non secure configure mode, and the direct reading is valid after setting. If they are not configured to enter the simulation mode after reset, they are invalid, because the simulation mode will not burn fuse.

Suggestion:

For the configuration of OTP attributes, it is recommended to read before burning fuse each time to confirm whether there has been an over write operation, so as to prevent excessive burning. Therefore, it is recommended to add a call to the read operation interface before writing. The next chapter has an example of ADKP. It is recommended that other attributes also operate similarly.

7 Customization 3：IVT Signature

Sign the IVT header to prevent tampering.

Note:

- ADKP needs to be installed in advance when IVT signs, so we open the ADKP burning operation in the previous chapter.
- IVT start certification needs to meet two conditions: 1: IVT_ AUTH=1, 2: LC advanced to OEM_ PROD/IN_ FIELD, since we do not intend to advance the chip, we only test the configuration function.
- Note that for the entire IVT section, including DCD and self-test DCD (if any), TAG heads need to be added. This section only explain the code for the IVT section from HSE Demo porting to bootloader. For the DCD and self-test DCD sections, you can refer to this method for porting.
- Note that when burning ADKP, the application needs to pull GPIO to VDD_ EFUSE power supply ,which can refer to the App VddEfusiePower function in HSE Demo, which was not explained in this article, can be implemented using the DIO driver in MCAL.
- Power on to open VDD_EFUSE, There are two methods for GPIO pull. Method 1 uses the efuse marker in IVT, which is operated by HSE FW. Therefore, the delay requirement in the document is to be greater than 1ms and requires actual testing and register checking, as follows:

Table 128: VDD_EFUSE configuration word

Bit #	Description
31	GPIO Polarity: - 0: the VDD_EFUSE is powered when driving the GPIO low - 1: the VDD_EFUSE is powered when driving the GPIO high
16:30	GPIO MSCR number as mentioned in the IO Mux sheet for the GPIO (refer to **[REF02]**); based on device type, it can be in the ranges from below table.
0:15	Delay provided in microseconds. Must be greater than 1 millisecond. **Note**: The delay period must be measured from the moment the GPIO switch occurs and the NCSPD_STAT4 bit is set (refer to [REF02]). Note that the measurement may differ between the boards if different external components are used. A large enough delay must be programmed to ensure that the VDD_EFUSE supply is powered.

Table 129: MSCR number

Device Type	SUIL2_0_MSCR	SUIL2_1_MSCR	SUIL2_4_MSCR	SUIL2_5_MSCR
S32G2/G3	0 - 101	112 - 190	N/A	N/A
S32R45	0 - 101	102 - 133	N/A	N/A

Method 2: Using Self DCD or DDR Init DCD, place the DCD code that pulls GPIO at the beginning of the DCD segment. This way, after executing the DCD, the running time to HSE FW is much longer than 1ms.

7.1 Codes Development

1. Remove the former IVT store codes:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
static void Bl_SaveConfiguration(…)
{
#if 0
   /* Update IVT configuration */
…
    #endif
}

static void Bl_SaveFwConfiguration(…)
{…
#if 0
   /* Update IVT configuration */
…
    #endif
}
```

2. Before calling the IVT GMAC generation function, burn ADKP as the key:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
volatile uint8_t gProgramAdkp = 0xDA;
StatusType Bl_ConfigureSecureBoot(void)
{…
hseAttrApplDebugKey_t adkpHash;
…
if(0xDA == gProgramAdkp)
   {
       if(HSE_SRV_RSP_OK != CryptoDal_GetAdkp(&adkpHash[0U])) // First, judge whether ADKP has been burned. If not, burn it.
       {
       CryptoDal_ProgAdkp(&applicationDebugKeyPassword[0]);
       }
   }
```

3. Implement the ADKP reading function.

Dal function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\include\CryptoDal.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_GetAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
       eRetVal = CryptoDal_Crypto_GetAdkp(pAdkp);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_GetAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pAdkp);
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_GetAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp)
{
   return
CryptoDal_Crypto_GetAttribute(HSE_APP_DEBUG_KEY_ATTR_ID,sizeof(hseAttrApplDebugKey_t),pAdkp);
}
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\include\CryptoDal_Crypto.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_Crypto_GetAdkp(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pAdkp);
```

4. Implement the IVT store function:

   Function call:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\swc\bootloader\platforms\S32G2XX\src\ BootloaderSpecific.c

```
//johnli add for signed IVT
```

```
                        Bl_SaveSignedIvtConfiguration(BL_SYS_IMAGE_STORAGE_ADDR,
                                    u32HseBackupFwImageFlashAddr,
                                    u32HseFwImageFlashAddr);
```

Final implement function:

```
static void Bl_SaveSignedIvtConfiguration(uint32_t u32SysImageStorageAddr,
                                    uint32_t u32FwBackupImageStorageAddr,
                                    uint32_t u32FwImageStorageAddr)
{
  Bl_ImageVectorTableType *pIvtUpdated;
   hseAttrApplDebugKey_t adkpHash;
     /* Update IVT configuration */
   MemLib_MemCpy(&Bl_FlashMirror[0], IVT_ADDR, QSPI_SECTOR_SIZE);
   pIvtUpdated = (Bl_ImageVectorTableType *) &Bl_FlashMirror[0];

   pIvtUpdated->u32BootCfgWord = IVT_BCW_SEC_BOOT_CONFIG_MASK;
     pIvtUpdated->u32SysImageAddr = u32SysImageStorageAddr;// This function also saves sys_ Img publish
related parameters
   pIvtUpdated->u32HSEAddr = u32FwImageStorageAddr; // This function also saves fw_ Img update related
parameters
     pIvtUpdated->u32HSEBckAddr=u32FwBackupImageStorageAddr;
   pIvtUpdated->u32ExtFlashType = IVT_ADDR;
   pIvtUpdated->u32FlashPageSize = QSPI_SECTOR_SIZE;
  CryptoDal_SignIvt((uint8_t *)pIvtUpdated,0x10,(uint8_t *)pIvtUpdated->u32GMAC[0]);// his function generates
IVT GMAC tagGMAC tag
   Fls_Erase(IVT_ADDR, QSPI_SECTOR_SIZE);
   while (MEMIF_IDLE != Fls_GetStatus())
   {
      Fls_MainFunction();
   }
   Fls_Write(IVT_ADDR, (const uint8 *) &Bl_FlashMirror[0], QSPI_SECTOR_SIZE);
   while (MEMIF_IDLE != Fls_GetStatus())
   {
      Fls_MainFunction();
   }
}
```

5. Implement the IVT GMAC Tag generation function:

   Dal function:

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\include\CryptoDal.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SignIvt(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pImage,
                                   VAR(uint32, AUTOMATIC) inTaglenth,
                                      P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pOutTagAddr);
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\generic\src\CryptoDal.c

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE) CryptoDal_SignIvt(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pImage,
                                   VAR(uint32, AUTOMATIC) inTaglenth,
                                   P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST) pOutTagAddr)
{
   VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_NOT_OK;
   if (NULL_PTR != CryptoDal_pGlobalConfigPtr)
   {
#if CRYPTODAL_USE_CRYPTO
      eRetVal = CryptoDal_Crypto_SignIvt(pImage, inTaglenth, pOutTagAddr);
#endif /* CRYPTODAL_USE_CRYPTO */
   }
   return eRetVal;
}
```

Final implement function:
C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\include\CryptoDal_Crypto.h

```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE)CryptoDal_Crypto_SignIvt(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pImage,
                                  VAR(uint32, AUTOMATIC)inTaglenth,
                                  P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pOutTagAddr);
```

C:\NXP\Integration_Reference_Examples_S32G2_2022_06\code\framework\realtime\bsw\dal\cryptodal\platforms\S32G2XX\src\ CryptoDal_Crypto.c

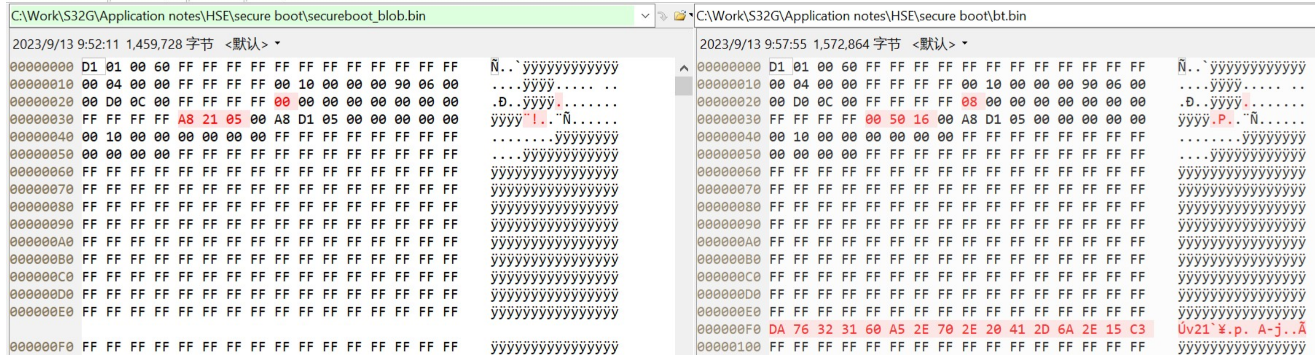
```
FUNC(Std_ReturnType, CRYPTODAL_APP_CODE)CryptoDal_Crypto_SignIvt(P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pInImage,
                                  VAR(uint32, AUTOMATIC)inTagLength,
                                  P2VAR(uint8, AUTOMATIC, CRYPTODAL_CONST)pOutTagAddr)
{
```

```
    VAR(hseSrvDescriptor_t, CRYPTODAL_VAR) HseSrvDescriptor;
    VAR(Std_ReturnType, AUTOMATIC) eRetVal = E_OK;

                        /* Clear previous request */
    MemLib_MemSet(&HseSrvDescriptor, 0, sizeof(HseSrvDescriptor));

            /* Fill the service descriptor */
            HseSrvDescriptor.srvId                                    = HSE_SRV_ID_BOOT_DATA_IMAGE_SIGN;
            HseSrvDescriptor.hseSrv.bootDataImageSignReq.pInImage = (HOST_ADDR)(pInImage);
            HseSrvDescriptor.hseSrv.bootDataImageSignReq.inTagLength = inTagLength;
            HseSrvDescriptor.hseSrv.bootDataImageSignReq.pOutTagAddr = (HOST_ADDR)(pOutTagAddr);


            if (E_OK == eRetVal)
            {
                    if (HSE_SRV_RSP_OK != CryptoDal_Crypto_HseSrv_Request(&HseSrvDescriptor))
                    {
                            eRetVal = E_NOT_OK;
                    }
            }
}
```

7.2 Simulation Testing

This article does not actually test the start authentication effect of IVT AUTH (involving life cycle advanced), so the IVT GMAC Tag is verified as follows:

Use flash tools to read the IVT head image (as described previously):



Comparison of native IVT images exported by S32DS IVT tool: Export IVT Image->C format

```
/* IVT binary */
const uint8_t ivt_binary[256] = {

            /* HEADER */
            0xd1, 0x1, 0x0, 0x60,

            /* reserved_1 */
            0xff, 0xff, 0xff, 0xff,
            /* Self-Test DCD */
            0xff, 0xff, 0xff, 0xff,
            /* Self-Test DCD (backup) */
```

```
        0xff, 0xff, 0xff, 0xff,
        /* DCD */
        0x0, 0x4, 0x0, 0x0,
        /* DCD (backup) */
        0xff, 0xff, 0xff, 0xff,
        /* HSE */
        0x0, 0x10, 0x0, 0x0,
        /* HSE (backup) */
        0x0, 0x90, 0x6, 0x0,
        /* Application bootloader */
        0x0, 0xd0, 0xc, 0x0,
        /* Application bootloader (backup) */
        0xff, 0xff, 0xff, 0xff,
        /* boot_config */
        0x0, 0x0, 0x0, 0x0,
        /* life_cycle_config */
        0x0, 0x0, 0x0, 0x0,
        /* reserved_2 */
        0xff, 0xff, 0xff, 0xff,
        /* hse_fw_config */
        0xa8, 0x21, 0x5, 0x0, 0xa8, 0xd1, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x10, 0x0, 0x0,
        0x0, 0x0, 0x0, 0x0, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0, 0x0, 0x0, 0x0,
        0xff, 0xff, 0xff, 0xff,
        /* reserved_3 */
        0xff,…0xff,
        /* gmac */
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

};
```

You can see the BOOT_SEQ bit has been set, the SYS-IMG address has been updated, and the GMAC value of the IVT header has been generated.