Freescale Semiconductor, Inc.
Application Notes

Document Number: AN5178

Rev.0, 11/2015

SHE on MPC5746C/MPC5748G

By: Himanshu Singhal, Pradip Singh

1. Introduction

This application note describes the features offered by the Hardware Security Module (HSM), which has been implemented on the MPC5746C/MPC5748G devices. This module implements the security functions described in the Secure Hardware Extension (SHE) functional specification. This application note provides an overview why HSM should be implemented and how it can be used in typical automotive application use cases to protect application code and inter-module communication. This application note shows the basic features and provides the user guidance of the most typical functions to be used with SHE implementation on HSM in the form of a Green Hills example project. However, it is not within the scope of this application note to discuss the details of the SHE specification. It focuses on the hardware features provided by SHE implementation on HSM. It is implied that the user is acquainted with the content of the SHE specifications.

Contents

Why is cryptography needed?

Today, the modern electronic industry has the same problem which Julius Caesar faced 2000 years ago; about transmitting information in secure and trusted manner between two communicating parties. Cryptography helps in exchanging secure information and provides the authenticity. In the automotive area, cryptography helps in implementing the following use cases:

- Immobilizers
- Component protection
- Secure flash updates
- Protecting data sets (for example: mileage)
- Feature management via Digital-Right-Management (DRM)
- Secure communication
- IP protection
- Car to X communication

Many more use cases exist and will be there in upcoming time. It should be noted that SHE implementation on HSM is not intended to be used for encrypting the code flash contents.

1.1. AES algorithm

SHE defines the Advanced Encryption Standard (AES) algorithm, described in**Appendix A**, which is used for cryptographic operations.

1.2. Cipher modes overview

Block ciphers like the AES algorithm, work with a defined granularity, often 64 bits or 128 bits. The simplest way to encode data is to split the message in the cipher specific granularity. In this case, the cipher output will depend only on the key and the input value. The drawback of this cipher mode, called Electronic Code Book (ECB), is that the same input values will be decoded into the same output values. This provides attackers the opportunity to use statistical analysis (for example, in a normal text some letter combinations occur much more often than others).

To overcome this issue other cipher modes were developed like the Cipher-block chaining (CBC), Cipher feedback (CFB), Output feedback (OFB), and Counter (CTR) mode.

SHE implementation on HSM supports only the ECB and the CBC modes which are described in the following sections.

1.2.1. Electronic codebook (ECB)

As described above this mode is the simplest one. And each block has no relationship with other blocks of the same message or information. **Figure 1** shows the block diagram of the ECB mode.

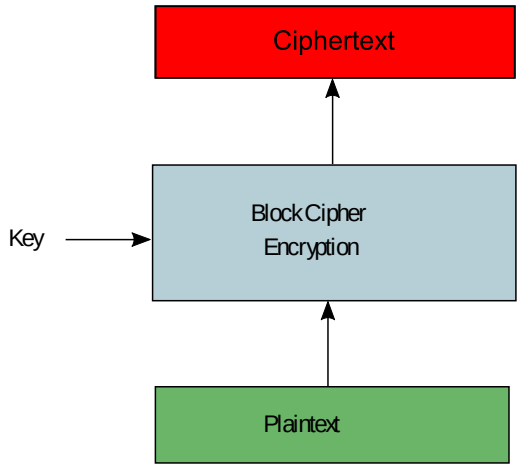

Figure 1: ECB block diagram

Figure 2 shows the drawback of the ECB mode. Figure is still visible in the encoded form, which is unsecure.

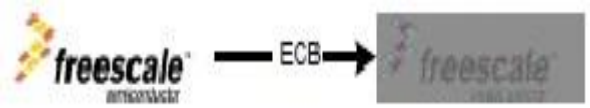

Figure 2: Encoding using ECB mode

1.2.2. Cipher-block chaining (CBC)

The Cipher-block chaining (CBC) mode, invented in 1976, is one of the most important cipher modes. In this mode the output of the last encoding step is XOR'ed with the input block of the actual encoding step. Because of this, an additional value for the first encoding step called Initialization Vector (IV) is necessary. In this method each cipher block depends on the plain text blocks processed up to that point.

Figure 3 shows the block diagram of the CBC mode.

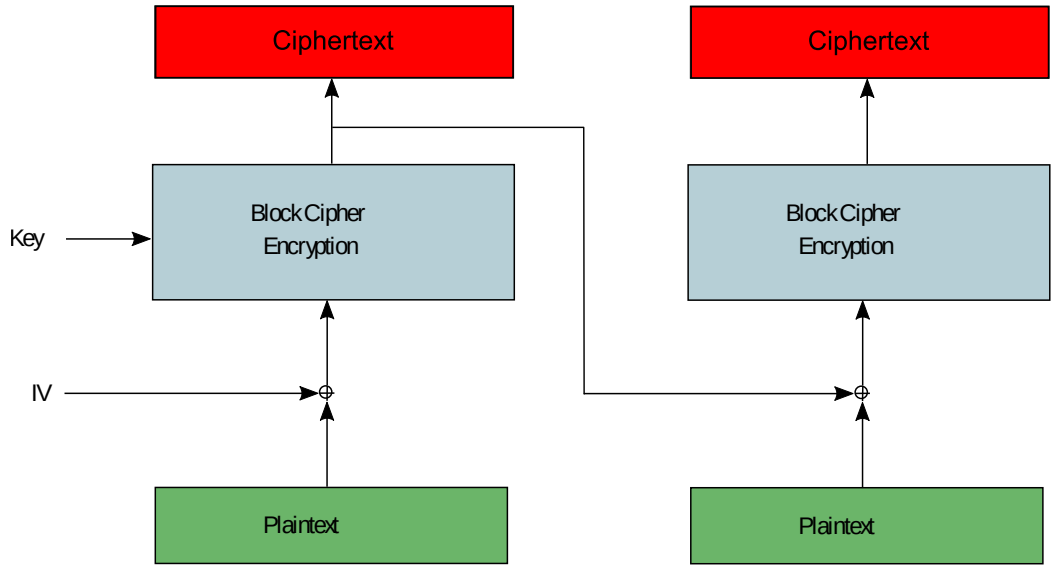

Figure 3: CBC block diagram

Figure 4 shows the encoding result of Freescale logo using the CBC cipher mode. The difference from the ECB mode(as shown in Figure 2) can be seen here. In many applications ECB mode may not be appropriate.

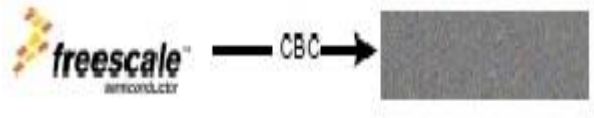

Figure 4: Encoding using CBC mode

1.2.3. CMAC

A CMAC provides a method for authenticating messages and data. CMAC uses the AES algorithm. The CM AC algorithm accepts a secret key as input and an arbitrary length message to be authenticated, and outputs a CMAC. The CMAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

Figure 5 shows the components of a CMAC scheme.

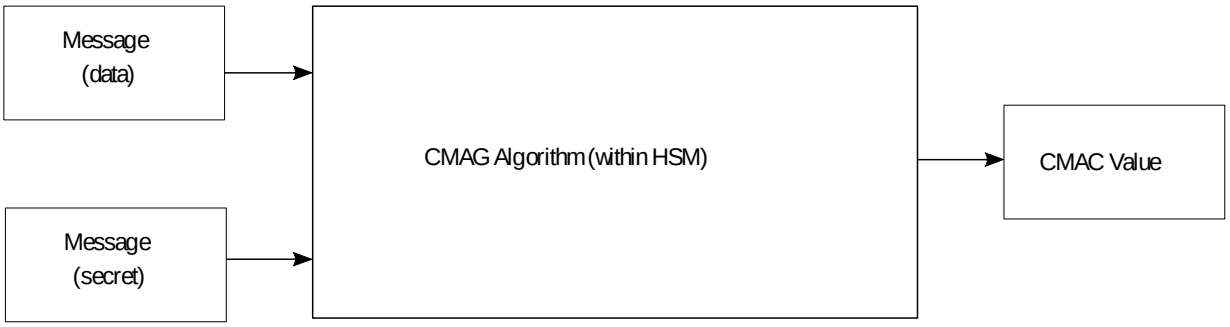

Figure 5: Components of a CMAC scheme

1.3. Typical automotive security use cases

Following sections describe automotive use cases and how they could be supported by the SHE implementation on HSM module. Many of these use cases assume that the application code was verified with the SHE implementation on HSM secure-boot-function.

1.3.1. Secure mileage

In the past, the mileages of cars were illegally reduced to increase the resale value of the vehicle. This essenti ally created a negative impact on the OEM reputation and increased quality and warranty questions. For this reason OEM has a strong interest in preventing any illegal manipulation of the mileage. SHE implementation on HSM could help in protecting the mileage data. The principle idea is that the mileage is stored encrypted in a non-volatile flash area. Initially, the encoded mileage is read from the flash memory (for example, EEPROM or EEPROM emulation) into the data memory. Before the value is used, it has to be decrypted by the SHE implementation on HSM and whenever the mileage is stored periodically back into the non-volatile memory the SHE implementation on HSM has to encode the mileage value again. This encoding and decoding will only work if the SHE implementation on HSM verified the application code without any failures before.

1.3.2. Immobilizers

Today immobilizers are standard equipment in every modern car. They prevent cars being stolen without the car key. Additionally, the reduction of the overall number of stolen vehicles has a positive effect on insurance premiums.

Following is an example of how a simple immobilizer implementation looks like.

A car key includes a transponder, a small cipher unit and a unique cryptographic key. The immobilizer unit sends a random value, generated by the SHE implementation on HSM, to the car key. The car key encrypts this value with the internal AES engine and sends the result back to the immobilizer. The immobilizer has the same secret key stored in the SHE implementation on HSM and is able to decrypt back the random value.

Now, the immobilizer code is able to verify the answer from the car key if the result is correct and the engine could be started.

1.3.3. Component protection

Component protection prevents dismantling single ECU's from a car and re-using it in other ones.
Often cars are stolen specifically to re-sell the single components into the aftermarket.

The OEM can now address following issues with a secure component protection scheme:

- Reduce the number of stolen cars
- Prevent any negative impact on reputation and quality
- Protect own aftermarket business

A component protection system based on the SHE implementation on HSM may look like this. The most valuable ECU's will include a controller which has a HSM. A master node which may be assigned by design or dynamically with a specific algorithm will poll all ECU's of the component protection system and request a specific answer (for example: unique ID in encoded from). In this case only ECU's with the right secret key will be able to send back a validresponse.Additionally, the master node can cross-check the unique ID with a database of all assembled modules in this specific car.

This component check can be done periodically while the car is used. If the system detects an unauthorized ECU in the car network it is able to react on it.

1.3.4. Flash programming/firmware updates

SHE implementation on HSM supports secure key storage area programming by the means of Cipher based message authentication code (CMAC) calculation. The application code will verify each block of the new flash image by re-calculating the CMAC value and will compare it with the offline pre-calculated value which is part of the flash image. This check will only be verified when the same secret key is used for the CMAC calculation.

2. HSM

The Hardware Security Module (HSM) is a subsystem, meant to address advanced security features, freeing the main core, from security tasks. It includes a secure core, security specific peripherals, like AES (Advanced Encryption Standard) cryptographic core, and local memories. It implements the security functions described in the secure Hardware Extension (SHE) Functional Specification Version 1.1.

Please refer to HSM Security Manual for block diagram and other information.

2.1. HSM features

SHE implementation on HSM implements a comprehensive set of cryptographic functions including secure storage for cryptographic keys, AES- 128 encryption and decryption, secure boot, AES-128 CMAC

authentication, and random number generation. As an introduction to the user, these features are explained in the following sections. This introduction is intended to give a basic understanding of the features and the demo code supplied with this application note facilitates the first steps with the SHE implementation on HSM itself. To get a more detailed overview of the register set please refer to the MPC5746C/MPC5748G Security Reference Manual.

NOTE

Contact support team to get the security reference manual.

2.2. Details of contents of Key Storage Area

2.2.1. Default secure code flash content

When SHE enabled parts are received from the factory, the secure code 64 K flash is populated with SHE firmware and the secret key (SK). The secure key storage area is otherwise erased if the parts are not SHE enabled. The Public flash as shown in **Figure 6** below refers to the flash area available to the user for application use.

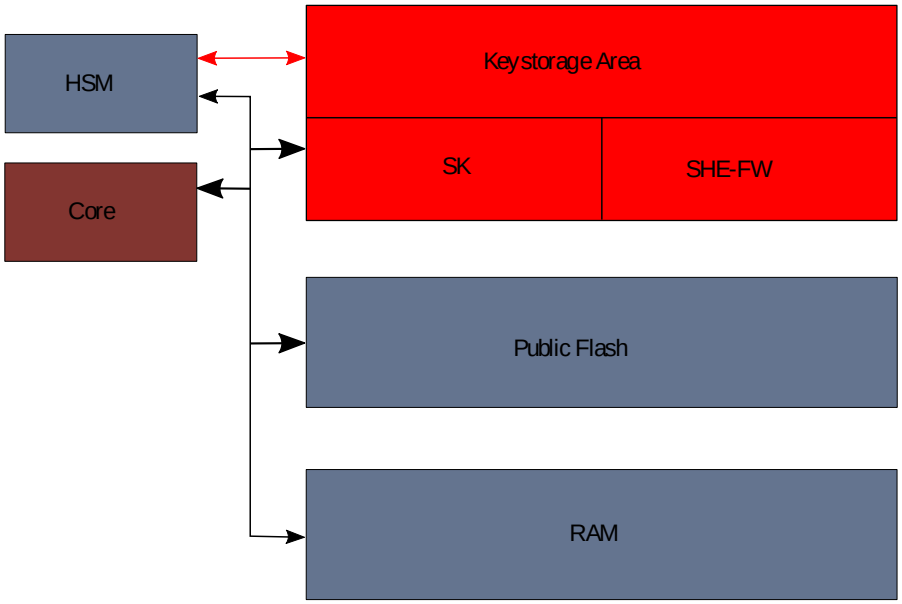

Figure 6: Flash content default state

2.2.2. User defined secure data flash contents

Once the user wants to use the SHE module he may program keys for application use into the secure code flash. KEY_1 to KEY_20 (20 keys) are user keys which can be programmed by the user with secret information for application use. Additionally MASTER_ECU_KEY, BOOT_MAC_KEY, and BOOT_MAC may also be programmed by the user. Please refer to **section 2.2.3** for details. Additionally the user may program application code into the Public flash area as on every other device.

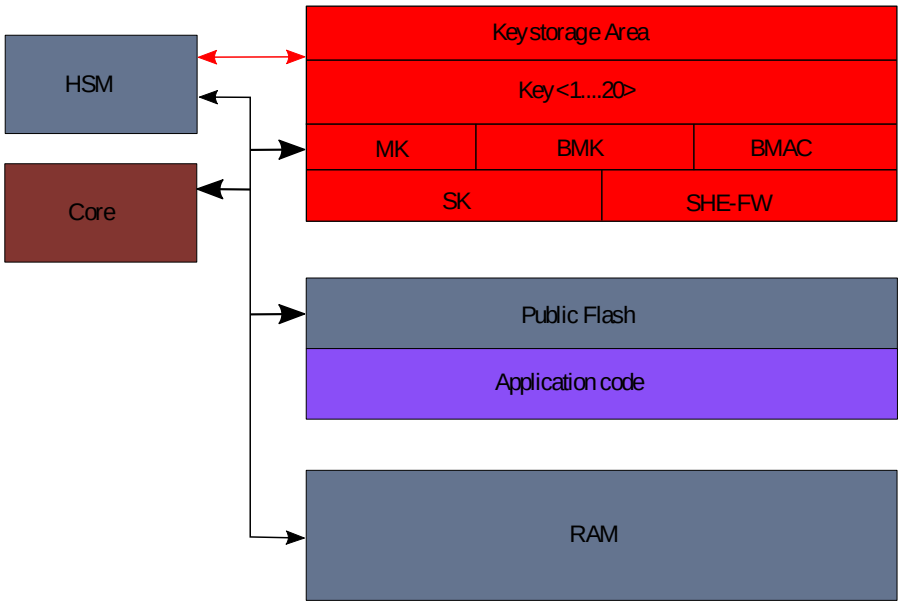

Figure 7: User-defined flash content

2.2.3. Adding user-defined content to Key Storage Area

In general, knowledge of a specific key is needed in order to update that specific key. MASTER_ECU_KEY is a key with special meaning.
It can be used to authorize updating other keys (BOOT_MAC_KEY, BOOT_MAC and KEY_1 to KEY_20) without knowledge of those keys. Refer to Table 5 "*Memory Update Policy*" of the SHE Specification. To add user keys the protocol as defined in the SHE specification must be used (in *section 9.1 Description of memory update protocol*). This ensures confidentiality, integrity, authenticity and provides protection against replay attacks. SHE implementation on HSM requires that in order to update the memory containing the keys the following must be calculated and passed to SHE implementation on HSM:

- K1 = KDF(KAuthID, KEY_UPDATE_ENC_C)
- K2 = KDF(KAuthID, KEY_UPDATE_MAC_C)
- M1 = UID'|ID|AuthID - 256 bits
- M2 = ENCCBC,K1,IV=0(CID'|FID'|"0...0"94|KID') -128 bits
- M3 = CMACK2(M1|M2) – 128 bits

Details of how to generate K1, K2 and M1 to M3 are given in **Appendix B.1.1**. These values will typically be derived on an offline computer and created as arrays in a header file. Registers HSM_CMD.PARAM_1, HSM_CMD.PARAM_2 and HSM_CMD.PARAM_3 are populated with the addresses of arrays M1, M2 and M3. The HSM_LOAD_KEY command must then be issued. In order to check that the update is performed correctly SHE implementation on HSM calculates:

- M4 = UID|ID|AuthID|M4* - 256 bits
- M5 = CMACK4(M4) – 128 bits

Registers HSM_CMD.PARAM_4 and HSM_CMD.PARAM_5 must be populated with the addresses of M4 and M5 respectively. Details of how to generate M4 and M5 are given in **Appendix B.1.7**.

2.3. Secure storage of cryptographic keys

SHE implementation on HSM provides secure and non-volatile storage for cryptographic keys as described in the SHE functional specification. The keys are stored in 15 memory slots, with one ROM slot, 13 non volatile slots and one RAM slot as shown in Table 1 . The first four slots have a dedicated use; the other slots are available for application specific keys. KEY<n> can be extended to 20 by using KBS bit. KEY_1 to KEY_10 are defined for KBS=0 and KEY_11 to KEY_20 are defined for KBS=1. The BOOT_MAC slot is loaded with a MAC value used by the secure boot process. This can be performed automatically by the SHE implementation on HSM under specific circumstances or by user software. All other slots are used for encryption or message authentication keys. The SECRET_KEY slot is programmed with a random value during device fabrication. All HSM encryption and message authentication commands specify a key by its KeyID.

Table 1 Key slot

Slot Name	KeyID	Memory Type	Counter [bits]	Key Flags(every flag has a size of one bit)						Default Factory State
				VERIFY_ ONLY	WRITE _PROT	BOOT_ PROT	DEBUG_ PROT	KEY_ USAGE	WILDCARD	
SECRET_KEY	0x0	Read-only	-	-	-	-	-	-	-	Written by Freescale
MSTER_ECU_KEY	0x1	NVM	28							Empty
BOOT_MAC_KEY	0x2		28							Empty
BOOT_MAC	0x3		28							Empty
KEY<n> n=1…10	0x4-0xD		28							Empty
RAM_KEY	0xE	RAM	28							Undefined after every reset
UID		Read-only	-	-	-	-	-	-	-	Written by Freescale

2.3.1. Key attributes

Each key has six flags associated with it, which determine how and under what conditions the key can be used.

2.3.1.1. VERIFY_ONLY

If set, the key can only be used for CMD_VERIFY_MAC.

2.3.1.2. WRITE_PROT

If set, the key cannot be updated even if an authorizing key (secret) is known. This flag should be set with caution. Setting this flag is an irreversible step, which will prevent the part from being reset to factory state. See section **Appendix C.1** for details.

2.3.1.3. BOOT_PROT

If set, the key cannot be used if the MAC value calculated in the SECURE_BOOT step did not match the BOOT_MAC value stored in secure key storage area.

2.3.1.4. DEBUG _PROT

If set, the memory slot is disabled if HSM to Host Status (HSM2HTS)[EXT_DEBUGGER] = 1.

2.3.1.5. KEY_USAGE

This flag determines if a key can be used for encryption/decryption or for MAC generation/verification (CMAC). If the flag is set, the key is used for MAC generation/verification. If clear, the key is used for encryption.

2.3.1.6. Wildcard

If set, the key cannot be updated by supplying a special wildcard (UID=0).

2.3.1.7. Key counter

Each user key has a counter which must be increased on every update. The counter is 28-bit long. The new counter value is used in the derivation of M2 when a key is being updated. Refer to section B.1.5.

2.3.1.8. Key ID

Each key has an identifying number associated with it. This number is used to identify the key being updated and the key authorizing the update. The following table shows the KeyID and KBS bit for each key.

Table 2 Key IDs

Key	KBS	KeyID
SECRET_KEY	x	0x00
MASTER_ECU_KEY	x	0x01
BOOT_MAC_KEY	x	0x02
BOOT_MAC	x	0x03

KEY_1	0	0x04
KEY_2	0	0x05
KEY_3	0	0x06
KEY_4	0	0x07
KEY_5	0	0x08
KEY_6	0	0x09
KEY_7	0	0x0A
KEY_8	0	0x0B
KEY_9	0	0x0C
KEY_10	0	0x0D
KEY_11	1	0x04
KEY_12	1	0x05
KEY_13	1	0x06
KEY_14	1	0x07
KEY_15	1	0x08
KEY_16	1	0x09
KEY_17	1	0x0A
KEY_18	1	0x0B
KEY_19	1	0x0C
KEY_20	1	0x0D
RAM_KEY	x	0x0E

2.4. AES-128 encryption and decryption

SHE implementation on HSM supports AES-128 encryption and decryption in ECB (Electronic Codebook) and CBC (Cipher Block Chaining) modes of operation as described in section 1.2. The key is selected from one of the memory slots which must be enabled for the encryption (KEY_USAGE =0. Refer to **section 2.3.1** for details). A plain text key can be loaded into the RAM_KEY slot using the LOAD_PLAIN_KEY command for keys that are not stored in a non-volatile memory slot. However, as this method implies a potential security risk, this might only be useful for development or debug purposes only.

2.5. AES-128 CMAC authentication

SHE implementation on HSM uses the AES-128 CMAC algorithm for message authentication. The key for the CMAC operation is selected from one of the memory slots which must be enabled for the authentication (KEY_USAGE =1. Refer to **section 2.3.1** for details). A plain text key can be loaded into the RAM_KEY slot using the LOAD_PLAIN_KEY command for keys that are not stored in a non-volatile memory slot. The VERIFY_MAC command supports comparison of a calculated MAC with an input MAC value.

2.6. Random number generation

SHE implementation on HSM has both a Pseudo Random Number Generator (PRNG) and a True Random Number Generator (TRNG). The PRNG has a 128-bit state variable and uses AES in output feedback mode to generate pseudo random values. A key derived from the SECRET_KEY is used for the PRNG. The RND command updates the state of the PRNG and returns the 128-bit
random value. The EXTEND_SEED command can be used to add entropy
to the PRNG state. The PRNG state is initialized after each reset with the INIT_RNG command which uses the TRNG to generate a 128-bit seed value for the PRNG. The HSM.HSM2HTS[RAND_INIT] flag is set when the PRNG is initialized. The INIT_RNG and RND commands use the TRNG to generate truly random values. The TRNG hardware runs off of a slower clock derived from the system clock.
Random values generated by the TRNG are checked with a statistical test to verify proper operation
of the TRNG. If the test fails, a TRNG error (EC=0x12) is returned. Due to the statistical nature of this test, there is a very small probability ($<10^{-9}$) that a properly operating TRNG will return an error. If a TRNG error is returned, the command can be issued again.

2.7. Unique ID

Unique Identifier Number (UID) is unique for every part and is programmed into the UTEST flash area when it is tested in wafer form. UID is 120 bits long. UID can be used during inter ECU communications to confirm that external controllers have not been substituted. If Wildcard is disabled for a specific key, then that key cannot be updated without specific knowledge of the UID of the part being updated. Refer to **Section 2.3.1** for details. UID is also used in the process for resetting part to their factory state. Refer to **Appendix C.1**.

UID can be obtained by issuing the HSM_GET_ID command.

2.7.1. Example code for retrieving UID from UTEST flash area

```
uint32_t  get_id_challenge[4] = {0xE6FE097D, 0xBC723E2C, 0xF0EA416F, 0xE68AD33E}
; /* user selects                          these values*/
uint32_t  GET_ID_UID[4];
uint32_t  UID_MAC[4];

while  (HSM.HSM2HTS.B.BUSY==1){}
 /*wait  until  HSM  is  idle*/
HSM_CMD.CMD = 0x10;                              /* Get UID*/
HSM_CMD.PARAM_1= (vuint32_t)&get_id_challenge; /*  input  challenge  value*/
HSM_CMD.PARAM_2= (vuint32_t)&GET_ID_UID;       /*  output  UID*/
HSM_CMD.PARAM_3= 0;                              /* holds the value of
HSM2HTS[7:0]*/
HSM_CMD.PARAM_4= (vuint32_t)&UID_MAC;
/*  output  challenge  response  */
HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;             /*HSM_CMD base address */
HSM.HT2HSMF.B.CMD_INT = 1;                       /* send command */
```

The SHE implementation on HSM will return 0 in UID_MAC if the MASTER_ECU_KEY is empty. UID_MAC is populated with a 128-bit MAC calculated over the concatenation of a 128-bit input

challenge value, UID and HSM.HSM2HTS[7:0]. GET_ID_UID is 128 bits with the 8 least significant bits set to 0.

2.8. Updating user keys

After a part's user keys are programmed into the secure key storage area and the part is no longer in its factory state, it may be necessary to update one or more
keys. SHE describes a mechanism for doing this and this has been implemented in the SHE implementation on HSM via the HSM_LOAD_KEY command. If a key has Write Protection (WC) flag set, it will no longer be possible to update that key.

2.8.1. Authorization

In order to keep keys secure, SHE requires that an authorizing key (secret) be known before an update to a specific key can be attempted.

Table 3 Key update overview

Key (Secret) (needed to update a key)					
Slot to update	**MASTER_ECU_KEY**	**BOOT_MAC_KEY**	**BOOT_MAC**	**KEY <n>**	**RAM_KEY**
MASTER_ECU_KEY	√				
BOOT_MAC_KEY	√	√			
BOOT_MAC	√	√			
KEY <n>	√			√	
RAM_KEY				√	

Knowledge of MASTER_ECU_KEY enables updating of all user keys except RAM_KEY. Knowledge of BOOT_MAC_KEY enables BOOT_MAC and BOOT_MAC_KEY to be updated. Knowledge of a specific KEY_<N> enables that specific KEY_<N> to be updated. Knowledge of any KEY_<N> enables the RAM_KEY to be updated.

2.8.2. Update process

The process for updating a given key is the same as that described in **section 2.3**. If the key to be updated is not Wildcard protected; the UID=0 may be used in the generation of M1 and M3. Otherwise the UID will need to be established for the part being updated and this UID used in the generation of M1 and M3. UID can be established as described in **section 2.7**. In **section 2.3** the key being updated has initial
value of 0xF and the authorizing key is the key itself. This is a very

specific case for a part in its factory state. Substitution of the authorizing key value will be required in all other cases.

2.8.3. Erasing all keys

A procedure for erasing the user content of the secure key storage in flash is described in **Appendix C.1**.

2.9. Secure Boot

2.9.1. Authenticating boot code

SHE implementation on HSM has a mechanism, which allows users to authenticate boot code in flash. The MCU can be configured so that on every boot a section of code is authenticated and the generated MAC will be compared with a value previously stored in secure key storage area. This is supported on every HSM reset with valid boot header programmed.

The key used to authenticate the boot code is called BOOT_MAC_KEY. A value for comparison is stored in secure key storage area and is called BOOT_MAC Firmware picks the image start address and image length from boot header which application program in boot location as specified in BAF chapter of MCU specific reference manual. A snapshot of boot header is shown in Table 4 below. The image offset is at 0x08 and image length is at offset 0x0C. It is advisable to include boot header as part of secure boot. If the boot code is not authenticated keys which are marked as boot protected cannot be used.

Table 4 Application boot header structure

Address Offset	Content
0x00	Boot Header Configuration
0x04	CPU2 Reset Vector
0x08	Boot Image Address
0x0C	Boot Image Size
0x10	CPU0 Reset Vector
0x14	CPU1 Reset Vector
0x18	Reserved for future uses

2.9.2. Adding BOOT_MAC to secure key storage area automatically using the HSM

Parts from the factory have no user keys stored in the secure key storage area. SHE implementation on HSM will calculate and store BOOT_MAC in secure key storage area if the following sequence is followed:

1. Program the code flash with code to be protected (including boot image address and boot image size parameters at address offset 0x08 and 0x0C)
2. Program BOOT_MAC_KEY into secure key storage area (other user keys may be programmed at this time too), See **section 2.3**.
3. Reset the part; HSM calculates BOOT_MAC and stores it in secure key storage area.
4. Reset the part again; HSM confirms previously calculated BOOT_MAC and set HSM.HSM2HTS [BOOT_OK] =1 (Secure Boot OK bit)

After this procedure keys marked as Boot Protected can be used by application code. On subsequent booting, provide BOOT_MAC_KEY has not been erased and the code flash is not erased, SHE implementation on HSM will calculate a MAC over the identified boot code and if the output value matches the value stored in secure key storage area (BOOT_MAC) set HSM.HSM2HTS[BOOT_OK] =1. This process is represented in Figure 8.

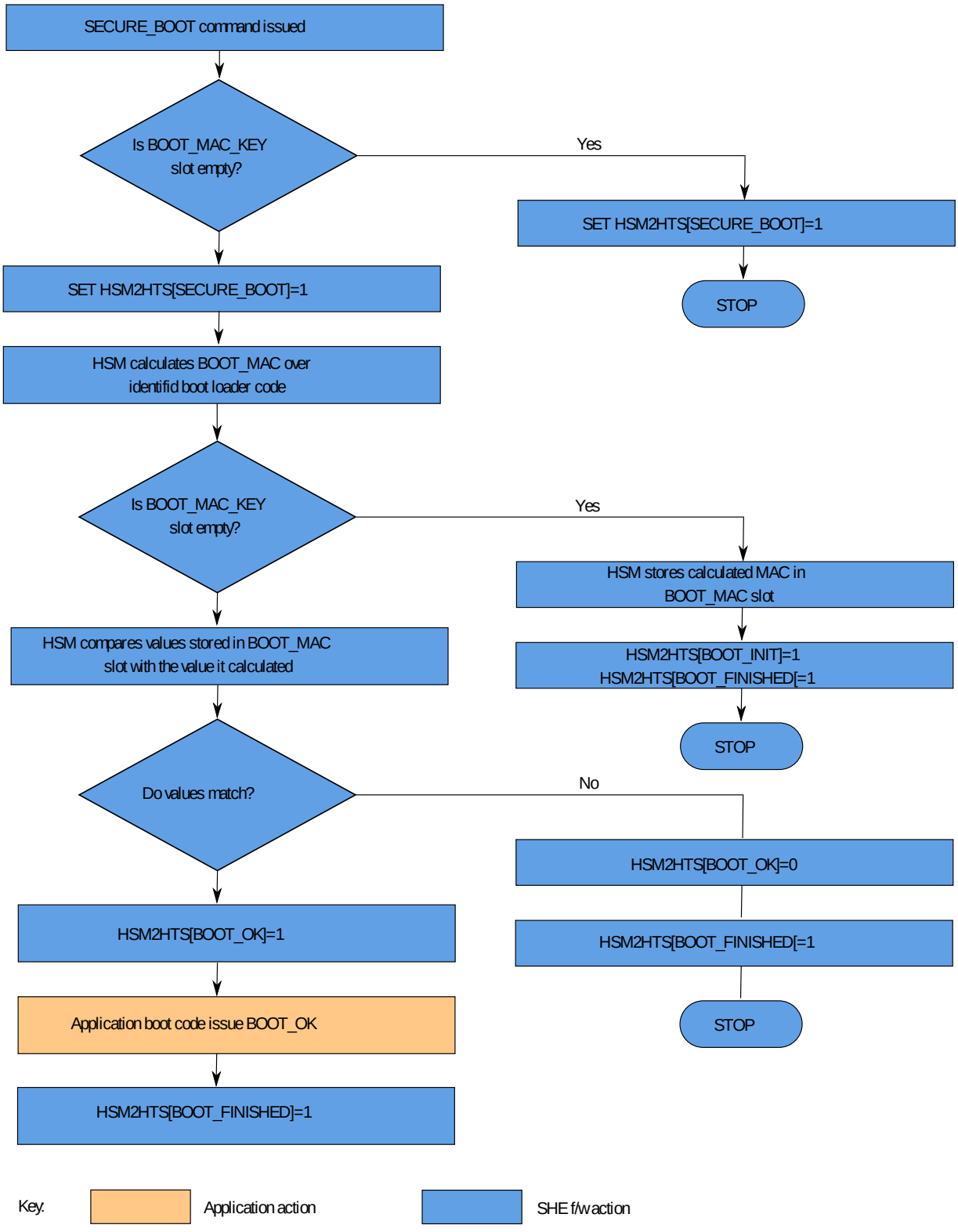

Figure 8: HSM Boot Process on MPC5746C/MPC5748G

2.10. Code flash update procedure

During software development and at other times during an ECU's life cycle it may be necessary to change the code flash
which is authenticated by the BOOT_MAC. This means that the BOOT_MAC calculated by the SHE implementation on HSM will not match the BOOT_MAC stored in the secure key storage area. In this scenario cryptographic services which used keys marked as Boot Protected
will be unavailable. The BOOT_MAC stored in secure key storage area must be updated to avoid this situation. There are 2 scenarios which lead to different methods for updating the stored BOOT_MAC.

2.10.1. Scenario 1: No key is write protected and all user keys can be erased and re-programmed

In this case the DEBUG_CHAL and DEBUG_AUTH commands can be used to set the secure key storage area back to its factory state. See **Appendix C.1** Secure key storage area to its Factory State. The DEBUG_CHAL and DEBUG_AUTH commands will
only work on a part which has no keys marked as Write Protected. All user keys must be known in this scenario, as they will all be erased by this process and one must know them in order to restore them to their previous values. After successfully running the DEBUG_CHAL and DEBUG_AUTH commands, the secure key storage area will be erased. New keys can be programmed into the secure key storage area as described in **section 2.3** . The procedure which causes the SHE implementation on HSM to generate BOOT_MAC should be followed. Refer to section 2.9.2.

2.10.2. Scenario 2: One or more keys are write-protected and all user keys cannot be erased. (Or not all user keys are known)

In this case the DEBUG_CHAL and DEBUG_AUTH commands cannot be used to set the secure key storage area back to its factory
state. In order to update the BOOT_MAC a new value for it must be derived and the key updated as described in **section 2.8**.

There are two methods which can be used to derive the new BOOT_MAC. These are described in the followi ng sections.

2.10.3. Method 1: Use the LOAD_RAM_KEY command and HSM to generate the new BOOT_MAC

In this method the RAM key is loaded by issuing the HSM_LOAD_PLAIN_KEY command. The HSM_GENERATE_MAC command can be used to derive the new
BOOT_MAC. If the code flash has already been programmed the following code can be used to derive the new BOOT_MAC value.

```
#define BOOT_BLOCK_START_ADDR 0;
volatile unsigned  long  long  length  ;
```

```
  vuint32_t *  code_start_ptr;
  vuint32_t *  code_length_ptr;
  vuint32_t *  boot_block_ptr;

  boot_block_ptr  =  (uint32_t  *  )BOOT_BLOCK_START_ADDR  ;
  code_start_ptr  =  boot_block_ptr  +  1;  /*  get  the  code  start  address

                                              from  the  code  flash*/
  code_length_ptr  =  boot_block_ptr  +  2;  /*  get  the  code  length  in  by
  tes  from  the  code flash*/ length  =  (*code_length_ptr)  *  8;

/* HSM_GENERATE_MAC takes bits as its input  */
  HSM_CMD.PARAM_1= (uint32_t) &BOOT_MAC_KEY;
  HSM_CMD.CMD= HSM_LOAD_PLAIN_KEY; /* Load the BOOT_MAC_KEY  as  plain  text  t
  o the                                       RAM_KEY*/
  HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;              /* command structure base
address */
  HSM.HT2HSMF.B.CMD_INT = 1;                                  /* send command */
  hsm_done();

  if (HSM.HSM2HTS.B.ERROR_CODE != HSM_NO_ERR) {failcount++;}

  HSM_CMD.PARAM_1= HSM_RAM_KEY;                       /* RAM  key  */
  HSM_CMD.PARAM_2= (unsigned long long)&length;    /*  msg  length  */
  HSM_CMD.PARAM_3= (vuint32_t)*code_start_ptr;
  HSM_CMD.PARAM_4= (vuint32_t) &NEW_BOOT_MAC;
  HSM_CMD.CMD = HSM_GENERATE_MAC;         /* generate the new BOOT_MAC value*/
  HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;           /* command structure base
address */
  HSM.HT2HSMF.B.CMD_INT = 1;                /* send command */
  hsm_done();
```

The new BOOT_MAC value can be used to update the BOOT_MAC values stored in secure key storage area.

2.10.3.1. Method 2: Generate the new BOOT_MAC offline

In this method an external program must be used. The hex data for binary image should be input to a program which can
calculate a new BOOT_MAC value using the BOOT_MAC_KEY. The new BOOT_MAC value can be used to update the BOOT_MAC values stored in secure key storage area.

2.11. HSM boot modes

On MPC5746C/MPC5748G three boot modes are supported. These control whether the main cores of the MPC5746C/MPC5748G are gated by the SHE implementation on HSM in flash boot modes.

2.11.1. Parallel boot mode

SHE f/w allows application boot before it starts the secure boot process. Hence application can be booted in parallel to secure boot operation.

2.11.2. Sequential boot mode

SHE f/w allows application boot only when secure boot process is complete. The result of the secure boot is immaterial. SHE f/w allows application irrespective of secure boot is passed, failed or is not supported.

2.11.3. Strict sequential boot mode

The SHE f/w allows application boot only when secure boot is success. It does not allow booting the application when secure boot is failed or secure boot is not supported.

2.11.4. Configuring boot modes

The user programs the boot configuration as 64 bit value starting at address 0x004000C8. The 64-bit value is needed because this is the smallest unit one can write in flash.

The SHE firmware controls various types of boot configuration by setting the HSM_READY bit. The user also has to program the HSM boot options in HSM enable and configuration DCF records as described in security reference manual of respective MCU. If a value other than mentioned in **Table 5** is programmed then SHE firmware assumes it as strict sequential boot.

Table 5 Boot configuration values

Boot Configuration	Value
Parallel boot	0xFFFFFFAB_XXXXXXXX
Sequential boot	0xFFFFFFCD_XXXXXXXX
Strict Sequential boot	0xFFFFFFEF_XXXXXXXX

3. Example use cases

To ease the introduction for the user, an example project is delivered with this application note. This code should be general example of how to use the HSM during a typical life cycle of a device from delivery through reprogramming as well as device recovery back into the default state again. It is not intended to showcase a complete application but should be understood as a starting point to facilitate the entry and provide the user with the basic prerequisites to start with the HSM. Furthermore, the user should be aware that this is an example code only and not intended for production software whatsoever. It was tested on the MPC574XG- 256DS Freescale evaluation motherboard with a MPC574XG-MB daughter

card. The example project contains several use cases. The project is using the GHS compiler 6.1.4 and Lauterbach debugger. The software concept of examples is described in **Appendix D**.

4. Conclusion

By showing the reasons for cryptography in general, and example automotive security use cases the need for cryptography becomes obvious. To protect the cryptographic keys from software attacks, the control over those keys moved from the software domain to the hardware domain. The MPC5746C and MPC5748G devices offer the security features specified in the Secure Hardware Extension (SHE) functional specification completely in hardware offering a higher security standard to OEM's in the future when using this device.

The discussions and explanations in this document provide an overview of the features the SHE implementation on HSM implements and how
these features can be used. With the example code demonstrating from first silicon, over re-programming up to setting everything back into the default state will give the reader the means to start working with this module.

5. Acronyms and definitions

Table 6 Acronyms and definitions

Term	Definition
AES	Advanced Encryption Standard
HSM	Hardware Security Module
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CMAC	Cipher based message authentication code
CTR	Counter (cipher mode)
ECB	Electronic Codebook
GHS	Green Hills
OEM	Original Equipment Manufacturer
OFB	Output Feedback
POR	Power on Reset
PRNG	Pseudo Random Number Generator
RCHW	Reset Configuration Half Word
RNG	Random Number Generator
SHE	Secure Hardware Extension
SK	Secret Key

6. Firmware programming and configuration

Please refer to security firmware block guide for details. Please contact Freescale support team for the same: http://www.freescale.com/SUPPORTHOME

7. References

- *SHE - Secure Hardware Extension functional specification Version1.1 (rev 439)* available on http://www.automotive-his.de/
- *MPC5746CRM/MPC5748GRM* available on https://www.freescale.com
- *[FIPS197] NIST/FIPS: Announcing the Advanced Encryption Standard (AES),* November 26, 2001 available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
- *Security Firmware Block Guide* – Contact: http://www.freescale.com/SUPPORTHOME
- *MPC5746C/MPC5748G* Security Reference manual – Contact: http://www.freescale.com/SUPPORTHOME

Appendix A

A.1 AES algorithm

The Advanced Encryption Standard (AES) algorithm was selected and specified by the US National Institute of Standard and Technology (NIST) [FIPS197] after a public championship. The algorithm is named after after the designers: Joan Daemen and Vincent Rijmen, and is thus is called *Rijndael-Algorithm*.

The AES algorithm is a symmetric cipher; for this encryption and decryption use the same key. The key value is 128, 192 or 256 bits width. Independent of the key size the block size is always 128 bits width according to the NIST standard3. **Figure 9** shows a general program flow of AES-128 (AES with 128-bit width key).

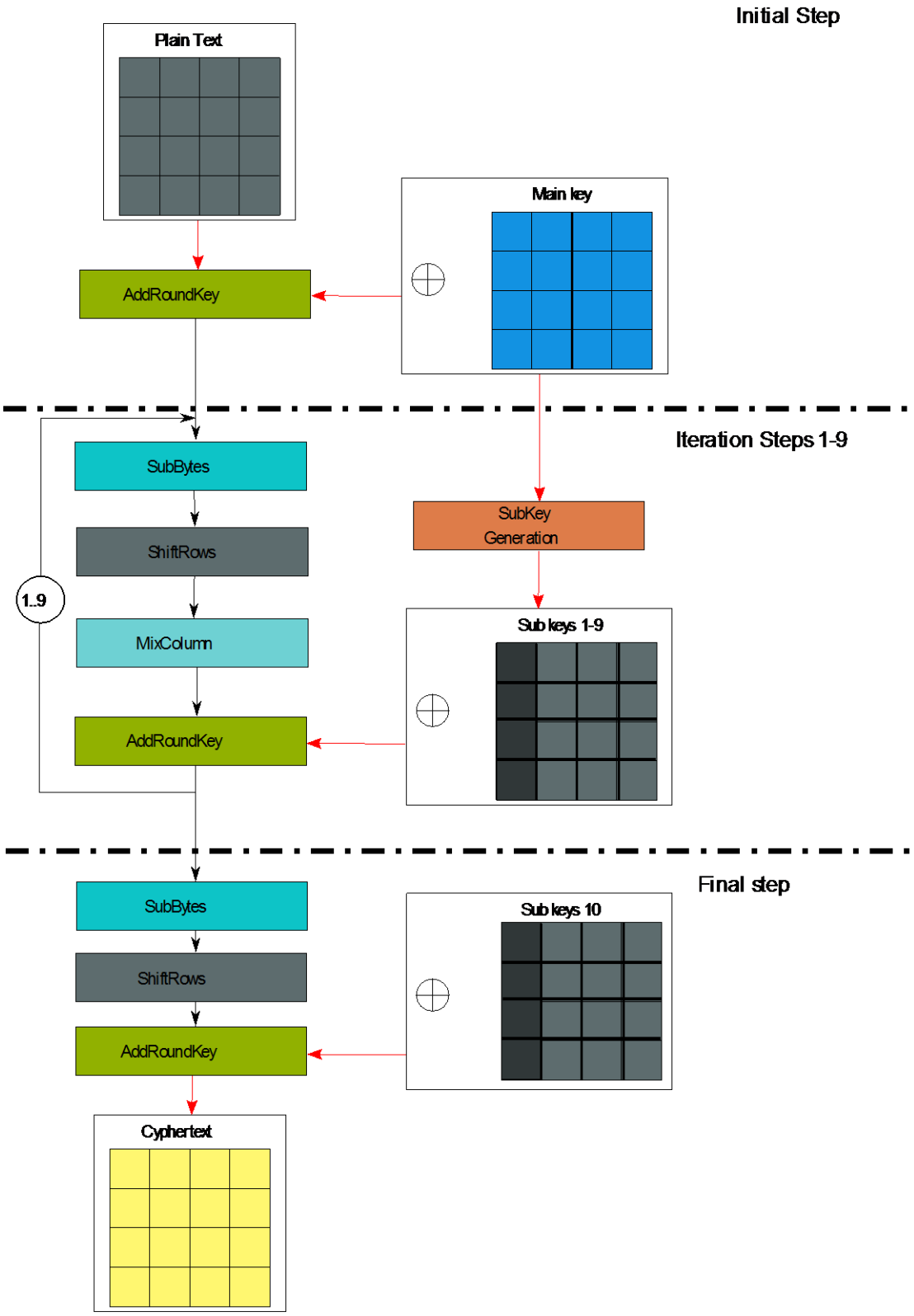

Figure 9 Flow of AES-128

The algorithm is based on the following four functions: SubBytes, ShiftRows, MixColumns and AddRoundKey. The sub- keys are derived from the main key.

A.1.1 AES basic functions

The functional description below is focused only on a 128-bit key implementation. But implementations for longer keys (192-bit or 256-bit) are similar.

A.1.1.1 SubBytes

The SubBytes function replaces each byte of the 128-bit input value by a value of a 16x16 constant array which is called S-Box. The high-nibble of each input byte is used as the y- and the low-nibble as the x-coordinate of the S-Box.

A.1.1.2 ShiftRows

The ShiftRows function interprets the input value as a 4x4 array and rotates the second row by one, the third row by two and the fourth row by 3 bytes to the left.

A.1.1.3 MixColumns

The input values are interpreted again as a 4x4 array. Every column will be modulo multiplied with a predefined matrix.

A.1.1.4 AddRoundKey

The 128-bit input value is xor'ed with the iteration specific sub-key value.

A.1.1.5 SubKey Generation

The SubKeys are generated by allocating a 44x4 byte array as shown in **Figure 10**.

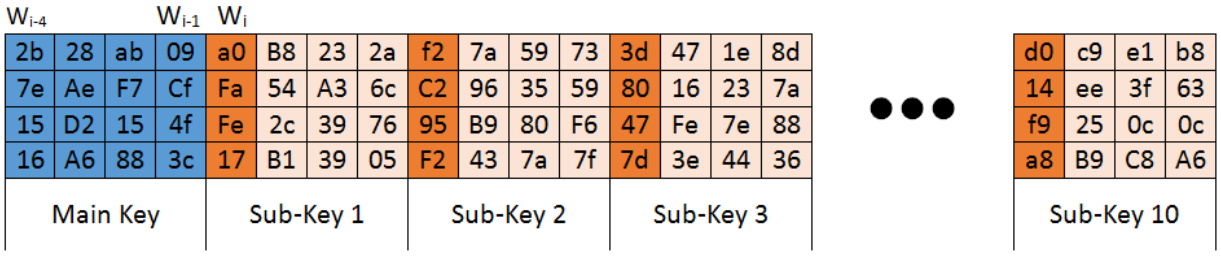

Figure 10 44x4 key array

The first 4x4 block is filled up with the main key. The remaining cells are determined by the following two rules:

Rule 1: Columns where the index could be divided by 4 are calculated by the following:

1. Rotate the column before (W_{i-1}) by one byte up.
2. Replace the four byte values by the S-Box like in the SubBytes functions.

3. XOR the column with the column of a pre-defined matrix called Rcon.

Rule 2: All other columns are generated by XOR of w_{i-1} and w_{i-4}.

Appendix B

B.1 Memory Update Protocol

B.1.1 Generating M1, M2, M3

In order to generate M1, M2 and M3 the following steps must be performed. Below is the source code for `hsm_kdf` and `hsm_done` functions.

```
void hsm_kdf(uint32_t *plaintext, uint32_t *dst, uint32_t numblocks)
{
    uint32_t *temp;
    uint32_t iv[4],cipher[4];
    uint8_t ptr_plaintext=0;

/*Initialise output to zero*/
    temp=dst;
    temp[0] = 0UL;
    temp[1] = 0UL;
    temp[2] = 0UL;
    temp[3] = 0UL;
/*Initialise Initial Vector to zero*/
    iv[0] = 0UL;
    iv[1] = 0UL;
    iv[2] = 0UL;
    iv[3] = 0UL;
/*Initialise Cipher to zero*/
    cipher[0]=0;
    cipher[1]=0;
    cipher[2]=0;
    cipher[3]=0;

    while(numblocks)
    {
  /*Load IV as RAM key*/
  HSM_load_plain_key((uint32_t)&iv);
  HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1;           /* send command */
  hsm_done();

  /*Calculate ECB ENC over 128 bit chunk of Plaintext*/
  HSM_encrypt_ECB(1UL,(uint32_t)&plaintext[4*(ptr_plaintext)],(uint32_t)&cipher
);
  HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1;           /* send command */
  hsm_done();

  numblocks--;
  iv[0] = iv[0] ^ cipher[0] ^ plaintext[0+4*(ptr_plaintext)];
      iv[1] = iv[1] ^ cipher[1] ^ plaintext[1+4*(ptr_plaintext)];
      iv[2] = iv[2] ^ cipher[2] ^ plaintext[2+4*(ptr_plaintext)];
```

```
      iv[3] = iv[3] ^ cipher[3] ^ plaintext[3+4*(ptr_plaintext)];
  ptr_plaintext++;
    }

/*Copy final OUTi to dst*/
    temp[0] = iv[0];
    temp[1] = iv[1];
    temp[2] = iv[2];
    temp[3] = iv[3];
}

void hsm_done(void)
{
    while(HSM.HSM2HTF.B.CMD_COMPLETE == 0 |  HSM.HSM2HTS.B.BUSY == 1);
    HSM.HSM2HTF.R = 0x00000002;               /* clear cmd_complete */
}
```

B.1.2 Generate K1

`K1 = KDF(KAuthID, KEY_UPDATE_ENC_C)`

- `KDF` is key derivation function which derives a secret key (`K1`) from a secret value.
- `KAuthID` Authorizing key value. In the case where a part from the factory has no keys programmed (t he Secure key storage area is erased) the value stored in flash does not have a valid checksum and HSM does not copy it to RAM at initialization, hence this value, in the HSM's RAM, is zero. In this case we are using `AuthID` = `ID` (i.e. the authorizing key will be the key itself)
- `KEY_UPDATE_ENC_C` – Constant value defined by SHE as: 0x01015348_45008000_00000000_000000B0

Code for generating K1:

```
for(i=0;i<KEY_SIZE_IN_WORD;i++)
   plaintext[i]=MASTER_ECU_KEY[i];
for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*2;i++)
   plaintext[i]=KEY_UPDATE_ENC_C[i-KEY_SIZE_IN_WORD];

hsm_kdf((uint32_t *)&plaintext, (uint32_t *) &K1, 2UL);
```

B.1.3 Generate K2

`K2 = KDF(KAuthID,KEY_UPDATE_MAC_C)`

- `KEY_UPDATE_MAC_C` – Constant value defined by SHE as : 0x01025348_45008000_00000000_000000B0

Code for generating K2:

```
for(i=0;i<KEY_SIZE_IN_WORD;i++)
   plaintext[i]=MASTER_ECU_KEY[i];
for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*2;i++)
   plaintext[i]=KEY_UPDATE_MAC_C[i-KEY_SIZE_IN_WORD];
```

```
hsm_kdf((uint32_t *)&plaintext, (uint32_t *) &K2, 2UL);
```

B.1.4 Generate M1

```
M1 = UID'|ID|AuthID
```

- `AuthID` can be either ID (number of key being updated) or MASTER_ECU_KEY number (0x1)
- UID' can be 0 (Wildcard value) because WC flag = 0 on parts from the factory
- UID is 120 bit and ID and AuthID are 4 bits each

Code for generating M1:

```
for(i=0;i<15;i++)
      M1_t[i] = uid[i];
  M1_t[15] = ID_AuthID;

   M1[0] = (uint32_t)M1_t[0]<<24 | (uint32_t)M1_t[1]<<16 | (uint32_t)M1_t[2]<<8 | (uint32_t)M1_t[3] ;
   M1[1] = (uint32_t)M1_t[4]<<24 | (uint32_t)M1_t[5]<<16 | (uint32_t)M1_t[6]<<8 | (uint32_t)M1_t[7] ;
   M1[2] = (uint32_t)M1_t[8]<<24 | (uint32_t)M1_t[9]<<16 | (uint32_t)M1_t[10]<<8 | (uint32_t)M1_t[11] ;
   M1[3] = (uint32_t)M1_t[12]<<24 | (uint32_t)M1_t[13]<<16 | (uint32_t)M1_t[14]<<8 |
(uint32_t)M1_t[15] ;
```

B.1.5 Generate M2

```
M2 = ENCCBC,K1,IV=0(CID'|FID'|"0...0"94|KID')
```

Run a CBC encryption using `K1` (as defined previously) with Initial Value (IV) = 0

- The message for encryption is a concatenation of:
- CID - the new counter value (28 bits). 0x0000001 in this case
- FID - New Protection flags - WP | BP | DP | KU | WC|VERIFY_ONLY (6 bits) 94 zeros to fill first 128 bit block with zeros.
- KID - The new key value (128 bits)

Code for generating M2:

```
// starting address of 32 bit counter value is KeyUpdateStruct[20]
  count_val = (uint32_t)KeyUpdateStruct[20]<<24 |
              (uint32_t)KeyUpdateStruct[21]<<16 |
              (uint32_t)KeyUpdateStruct[22]<<8 |
               (uint32_t)KeyUpdateStruct[23];

// 8 bit flag is at KeyUpdateStruct[19]
  flag_val = (uint32_t)KeyUpdateStruct[19];
  M2_t[0] = ((count_val<<4) & 0xFFFFFFF0) | ((flag_val>>2) & 0x0000000F);


  flag_val = (uint32_t)KeyUpdateStruct[19];
  M2_t[1] = (flag_val<<30) & 0x30000000;
  for(i=2;i<KEY_SIZE_IN_WORD;i++)
    M2_t[i] = 0x00000000;
  for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*2;i++)
  M2_t[i] = (uint32_t)KeyUpdateStruct[4*(i-KEY_SIZE_IN_WORD)+0]<<24 |
          (uint32_t)KeyUpdateStruct[4*(i-KEY_SIZE_IN_WORD)+1]<<16 |
```

```
          (uint32_t)KeyUpdateStruct[4*(i-KEY_SIZE_IN_WORD)+2]<<8 |
          (uint32_t)KeyUpdateStruct[4*(i-KEY_SIZE_IN_WORD)+3];

// Load K1 in RAM key
   HSM_load_plain_key((uint32_t)&K1);                /*Load RAM key*/
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;                    /* send command */
   hsm_done();
   for (i = 0;i<8; i++)
   {
      M2[i] = 0;
   }

   /*CBC ENCRYPT using K1 to get M2*/
   HSM_encrypt_CBC(2, (uint32_t) &M2_t, (uint32_t) &M2, (uint32_t)&IV);
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;                  /* send command */
   hsm_done();
```

B.1.6 Generate M3

`M3 = CMACK2(M1|M2)`

A CMAC is performed over `M1` and `M2` using key `K2`

Code for generating M3:

```
 HSM_load_plain_key((uint32_t)&K2);              /*Load K2 in RAM key*/
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;                  /* send command */
   hsm_done();

/*Generate MAC  over (M1|M2)*/
   for(i=0;i<KEY_SIZE_IN_WORD;i++)
      M3_t[i]=M1[i];
   for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*3;i++)
      M3_t[i]=M2[i-KEY_SIZE_IN_WORD];

 /*debug_cmac*/
   HSM_CMD.CMD = 0x5;                /* GenMac*/
   HSM_CMD.PARAM_1 = (uint32_t)0xE;
   HSM_CMD.PARAM_2 = (uint32_t)&messagedebugcmaclen;
   HSM_CMD.PARAM_3 = (uint32_t)&M3_t;
   HSM_CMD.PARAM_4 = (uint32_t)&M3;
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;              /* send command */
   hsm_done();
```

B.1.7 Generating M4, M5

When the HSM _LOAD_KEY command is issued HSM derives M4 and M5. These values can be independently generated offline and compared against those generated by the HSM.

B.1.8 Generating K3

```
K3 = KDF(KEYID,KEY_UPDATE_ENC_C)
```

- KID – The new key value (128 bits)
- KEY_UPDATE_ENC_C – Constant value defined by SHE as: 0x01025348_45008000_00000000_000000B0

Code for generating K3:

```
for(i=0;i<KEY_SIZE_IN_WORD;i++)
 plaintext[i]=(uint32_t)KeyUpdateStruct[4*i+0]<<24 |
          (uint32_t)KeyUpdateStruct[4*i+1]<<16 |
          (uint32_t)KeyUpdateStruct[4*i+2]<<8 |
          (uint32_t)KeyUpdateStruct[4*i+3];
for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*2;i++)
  plaintext[i]=KEY_UPDATE_ENC_C[i-KEY_SIZE_IN_WORD];

hsm_kdf((uint32_t *)&plaintext, (uint32_t *) &K3, 2UL);
```

B.1.9 Generate M4

```
M4 = UID|ID|AuthID|M4*
```

- M4 is a concatenation of:
  - UID - Unique ID (120 bits)
  - ID - number of key updated (4 bits)
  - AuthID - number of key authorizing the update (4 bits)
  - M4* - the encrypted counter value; prior to encryption the counter value (28 bits) is padded with a 1 and 99 0's. The key for the ECB encryption is K3 (derived as above). M4* = ENCECB K3 (CID(28bit)| 1 | 99bit zeros)

Code for generating M4:

```
count_val = (uint32_t)KeyUpdateStruct[20]<<24 |
           (uint32_t)KeyUpdateStruct[21]<<16 |
          (uint32_t)KeyUpdateStruct[22]<<8 |
          (uint32_t)KeyUpdateStruct[23];
M4_t[0] = ((count_val<<4) & 0xFFFFFFF0) | 0x00000008;
for(i=1;i<KEY_SIZE_IN_WORD;i++)
       M4_t[i] = 0x00000000;

// Load K3 in RAM key
 HSM_load_plain_key((uint32_t)&K3);
 HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
 HSM.HT2HSMF.B.CMD_INT = 1;  /* send command */
```

```
hsm_done();

for (i = 0;i<4; i++)
  {
     M4_te[i] = 0;
  }

  /*EBC ENCRYPT */
  HSM_encrypt_ECB(1, (uint32_t)&M4_t, (uint32_t)&M4_te);
  HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1;  /* send command */
  hsm_done();

  for(i=0;i<KEY_SIZE_IN_WORD;i++)
      M4_i[i]=M1[i];
  for(i=KEY_SIZE_IN_WORD;i<KEY_SIZE_IN_WORD*2;i++)
      M4_i[i]=M4_te[i-KEY_SIZE_IN_WORD];
```

B.1.10 Generate K4

`K4 = KDF (KID, KEY_UPDATE_MAC_C)`

- `KID` – The new key value (128 bits)
- `KEY_UPDATE_MAC_C` = 0x01025348_45008000_00000000_000000B0

B.1.11 Generate M5

$M5 = CMAC_{K4}(M4)$

B.2 Example code for updating a key (secret)

```
uint32_t  M1  [4]  =  {0xFFFFFFFF,  0xFFFFFFFF,  0xFFFFFFFF,  0xFFFFFF11};
uint32_t  M2  [8]  =  {0xff8b75f7,  0x3e6ad5a1,  0x729423c6,  ..  , 0xf0cc28ec};
uint32_t  M3  [4]  =  {0x57f51382,  0x4cfd1ba7,  0xd7593939,  0x4c8d0036};
uint32_t  M4_output  [8]  ;
uint32_t  M5_output  [4]  ;
while  (HSM.HSM2HTS.B.BUSY  ==1){}          /*wait  until  HSM  is  idle*/

HSM_CMD.PARAM_1 =  (vuint32_t)&M1  ;        /*  address  where  HSM
will  look  for  M1*/
HSM_CMD.PARAM_2 =  (vuint32_t)&M2  ;        /*  address  where  HSM
                                               will  look  for  M2*/
HSM_CMD.PARAM_3 =  (vuint32_t)&M3;           /*  address  where  HSM
                                               will  look  for  M3*/
HSM_CMD.PARAM_4 =  (vuint32_t)&M4_output;    /*  address  where  HSM
                                             will  write  M4*/
HSM_CMD.PARAM_5 =  (vuint32_t)&M5_output;    /* address  where  HSM
                                               will  write  M5*/
HSM_CMD.CMD = 0x7;                           /* HSM Load Key*/
HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD; /* command structure base address*/
HSM.HT2HSMF.B.CMD_INT = 1;                   /* send command */
```

Appendix C

C.1 Resetting secure key storage area to its factory state

SHE describes a mechanism for resetting the secure key storage area to the state it was in when it left the factory. The mechanism is only applicable if no user key has been write protected. The process is described in section 11 of the SHE spec "failure analysis of SHE/Resetting of SHE". HSM has implemented this mechanism by way of two commands. These are HSM_DEBUG_CHAL and HSM_DEBUG_AUTH. Successfully issuing these commands will result in the part having no user keys (MASTER_ECU_KEY, BOOT_MAC, BOOT_MAC_KEY, KEY1 ...KEY10 are all erased).

The TRNG must be initialized prior to the HSM_DEBUG_CHAL command being issued. The TRNG is initialized by executing the HSM_INIT_RNG command. The TRNG is used in deriving a challenge value.

```
/* initialize RNG */
while  (HSM.HSM2HTS.B.BUSY  ==1){}    /*wait  until  HSM  is  idle*/
HSM_CMD.CMD = HSM_INIT_RNG;           /* 0x0A */
HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;
HSM.HT2HSMF.B.CMD_INT = 1;            /* send command */
hsm_done();
```

The HSM2HTS [RAND_INIT] bit must be checked to confirm that the TRNG was correctly initialized

```
if  (HSM.HSM2HTS.B.RAND_INIT ==
   0)  {failcount++;}                /*  check  RIN  bit  is  set*/
```

The HSM will provide a challenge value when the HSM_DEBUG_CHAL command is issued.

```
/* generate challenge value */
HSM_CMD.PARAM_1 =  (vuint32_t)&challenge  ;
                                      /*  challenge  is  declared
                                          as  4  x  uint32_t  */
HSM_CMD.CMD = HSM_DEBUG_CHAL;   ;     /* 0x12 */
HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;
HSM.HT2HSMF.B.CMD_INT = 1;            /* send command */
hsm_done();
```

The UID for the part in question is added to challenge output and an authorization value is derived using KDEBUG. KDEBUG is defined as:

```
KDEBUG = KDF (MASTER_ECU_KEY, DEBUG_KEY_C)
DEBUG_KEY_C = 0x01035348_45008000_00000000_000000B0
```

The authorization value is calculated as follows:

```
AUTHORIZATION= CMACKDEBUG(CHALLENGE|UID)
```

For development purposes this may be calculated using the HSM

```
/* load RAM_key with KDEBUG*/
HSM_CMD.PARAM_1 =  (uint32_t)  &KDEBUG;
HSM_CMD.CMD=  HSM_LOAD_PLAIN_KEY;  ;    /* 0x08 */
```

```
  HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1; /* send command */
  hsm_done();

  challenge_UID  [0]  =  challenge[0];
  challenge_UID  [1]  =  challenge[1];
  challenge_UID  [2]  =  challenge[2];
  challenge_UID  [3]  =  challenge[3];
  challenge_UID  [4]  =  UID[0];
  challenge_UID  [5]  =  UID[1];
  challenge_UID  [6]  =  UID[2];
  challenge_UID  [7]  =  UID[3];

/* generate  CMAC  based  on  challenge|UID  using  KDEBUG  */
  HSM_CMD.PARAM_1 =  HSM_RAM_KEY;             /*  RAM  key  */
  HSM_CMD.PARAM_2 =  (unsigned  long  long)&length;

  /*  msg  length  :  248  in  this
                                                  case  (UID  is  120 bits  )*/
  HSM_CMD.PARAM_3 =  (vuint32_t)&challenge_UID;
  HSM_CMD.PARAM_4 =  (vuint32_t)&authorization;
  HSM_CMD.CMD=  HSM_GENERATE_MAC;   ;          /* 0x05 */
  HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1;                   /* send command */
  hsm_done();
```

AUTHORIZATION is passed to HSM and `HSM_DEBUG_AUTH` is issued.

```
/* issue authorization command  */
  HSM_CMD.PARAM_1 =  (vuint32_t)&authorization ;
  HSM_CMD.CMD=  HSM_DEBUG_AUTH;
  HSM.HT2HSMS.R = (uint32_t) &HSM_CMD;
  HSM.HT2HSMF.B.CMD_INT = 1;                 /* send command */
  hsm_done();
```

If the `HSM_DEBUG_AUTH` command is successfully executed, HSM will respond with `HSM_NO_ERROR`.

Appendix D

D.1 Example code

All examples are running on Freescale Evaluation board. Basic mode initialization of the part is done and the PLL is setup to run at 160 MHz. All examples are running from flash. The examples were tested using the GreenHills Compiler 6.1.4 and a Laughterbach Debugger.

Following is the list of functionalities checked for the HSM module:

- ECB_Test
- CBC_Test
- Generate_MAC_Test
- Verify_MAC_Test
- UID_Test
- Debug_Protocol_Test
- Memory_Update_Protocol_Test
- Memory_Update_Protocol_Wildcard_Test

D.1.1 ECB_Test

This section contains the code for ECB_Test. For other functionalities refer to the attached project file.

```
//load ecbkey as RAM_KEY
   HSM_load_plain_key((uint32_t) &ecbkey); /* ebckey is loded to RAM_KEY for
                                              encryption and decryption */
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;              /* send command */
   hsm_done();

// encrypt ecbPlainText to ecbCypheredText1 using ecbkey.
   HSM_encrypt_ECB(1, (uint32_t) &ecbPlainText, (uint32_t) &ecbCypheredText1);
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;              /* send command */
   hsm_done();

// decrypt ecbPlainText2 to ecbCypheredText1 using ecbkey.
   HSM_decrypt_ECB(1,(uint32_t)&ecbPlainText2,(uint32_t)&ecbCypheredText1);
   HSM.HT2HSMS.R         = (uint32_t) &HSM_CMD;
   HSM.HT2HSMF.B.CMD_INT = 1;             /* send command */
   hsm_done();

// code for HSM_encrypt_ECB
    HSM_CMD.CMD = 0x1;                         /* encrypt ECB */
```

```
    HSM_CMD.PARAM_1 = 0xE;                  /* key ID = RAM_KEY (Plain key)*/
    HSM_CMD.PARAM_2 = nBLOCKS;              /* n 128-bit blocks */
    HSM_CMD.PARAM_3 = cypher_address;       /* plain text address */
    HSM_CMD.PARAM_4 = plain_address;        /* cypher text address */
    HSM_CMD.PARAM_5 = 0x00000000;

// code for HSM_decrypt_ECB
    HSM_CMD.CMD = 0x3;                        /* decrypt ECB */
    HSM_CMD.PARAM_1 = 0xE;                    /* key ID = RAM_KEY (Plain key)*/
    HSM_CMD.PARAM_2 = nBLOCKS;                /* n 128-bit blocks */
    HSM_CMD.PARAM_3 = cypher_address;         /* plain text address */
    HSM_CMD.PARAM_4 = plain_address;          /* cypher text address */
    HSM_CMD.PARAM_5 = 0x00000000;
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN5178

Rev. 0, 11/2015