AN4240
Application note

Introduction to the Cryptographic Service Engine (CSE) module for SPC56ECxx and SPC564Bxx devices

Introduction

This application note provides an easy introduction to the usage of the CSE module inside the SPC56ECxx and SPC564Bxx family of devices.

The CSE module implements the security functions described in the Secure Hardware Extension (SHE) functional specification version 1.1.

Three examples show main the features of the cryptographic service engine and in the same time the differences between the Electronic Code Book (ECB), and the Cipher Block Chaining (CBC) mode of the Advanced Encryption Standard (AES) algorithm, defined by SHE specification.

In particular, first example shows how to load cryptographic keys into secure flash in order to permit the usage of the cryptographic module. Second application code shows that if a data or an image has a low variance, the CBC cipher mode provides a best performance in terms of message encryption in comparison with the ECB cipher mode. Last example code shows how to release a Secure Boot in order to prevent application code from being altered by an unauthorized party cipher.

Contents

List of tables

List of figures

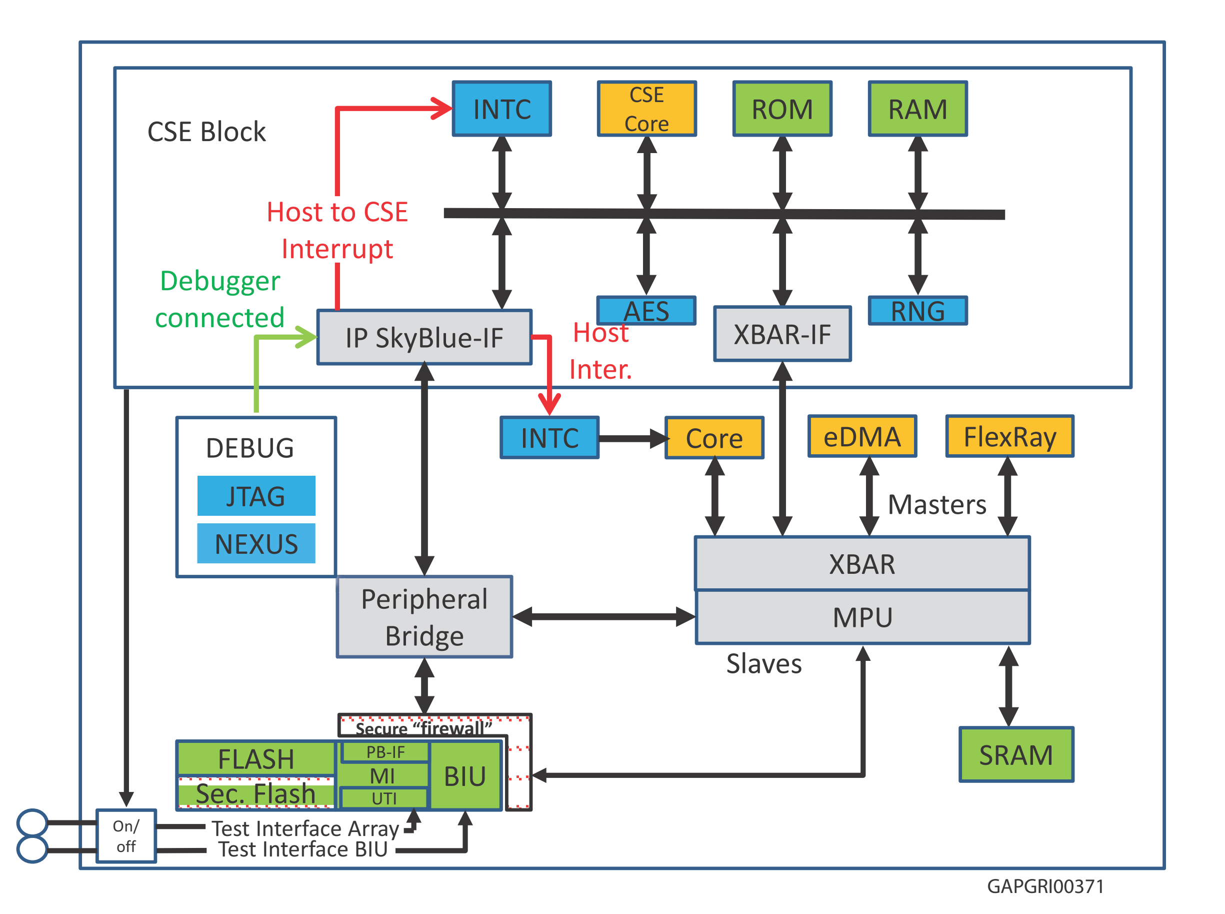
1 CSE IP block overview

Cryptographic Service Engine (CSE) is a peripheral module that implements the security functions described in the Secure Hardware Extension (SHE) Functional Specification Version 1.1.

The CSE is an on-chip extension of the microcontroller. It is intended to move the control over cryptographic keys from software domain into the hardware domain and therefore protect those keys from software attacks.

CSE design includes a host interface (via peripheral bridge) with a set of memory mapped registers used by CPU to issue commands (for example Get_ID, INIT_CSE, etc). Furthermore a system bus interface (via xbar IF) allows the CSE to directly access the system memory. Here the crypto module behaves like any other master. Through the host interface the user configures and controls the CSE module, for example putting the module into low power mode, enabling interrupts for finished command processing or suspending command processing. The status and error register gives further system information.

Figure 1. CSE IP block



Two dedicated blocks of system flash memory are used by the CSE for secure key and firmware storage. These blocks are not accessible by other masters from system and therefore are called secure flash. The command processing is done by a 32-bit CSE core with attached ROM and RAM running at system frequency of SoC. See *Figure 1.*

After system boot, the core comes out of reset and executes reset code from the module ROM. This code will load the firmware from the secure flash into the module RAM and start executing from there. This reduces the flash accesses by the crypto core.

The AES block is a slave to the crypto internal bus. It processes the encryption (plain text to ciphertext) and decryption (ciphertext to plaintext) and offers AES CMAC authentication.

The 32-bit secure core works at 120 MHz with a throughput of 100 Mbit/sec.

2 AES-128 encryption and decryption overview

The CSE module supports two cipher modes: Electronic Code Book (ECB) and Cipher Block Chaining (CBC) and CMAC authentication mode.

Block ciphers like AES algorithm works with a granularity of 64 or 128 bits. This means that if someone wants to encode data, it is necessary to split the message with an equivalent granularity.

In the ECB cipher mode the output cipher message depends not only on the chosen cryptographic key and the input data, but same input is encrypted in the same manner.

This could provide an opportunity to hack the message using an example statistical analysis.

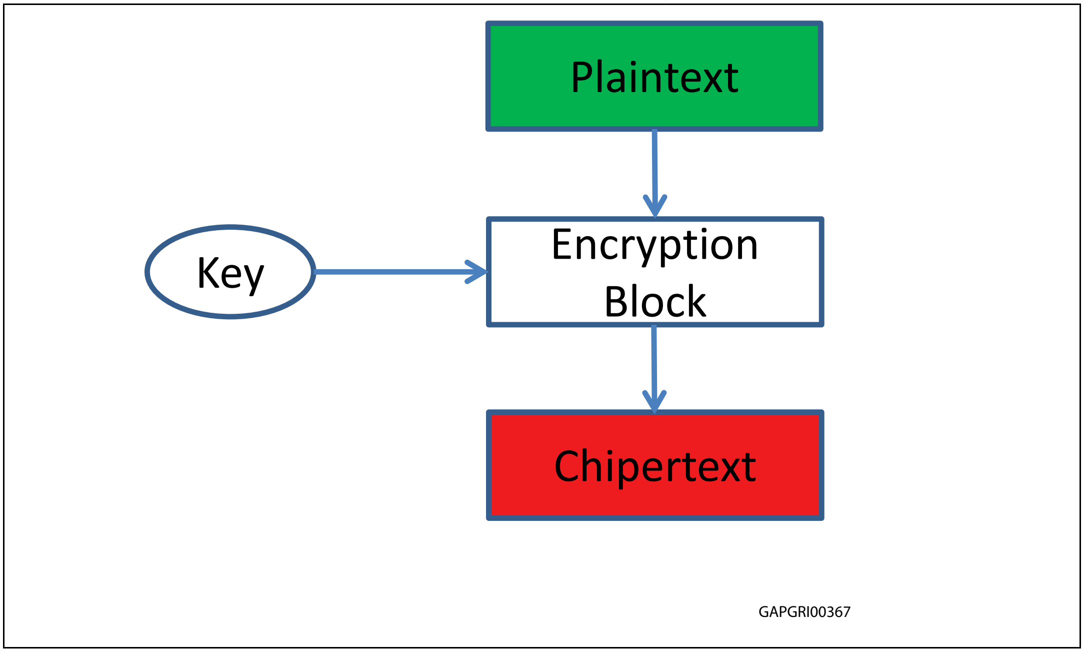
For this reason, the CBC cipher mode introduces more entropy to the encryption process using a chaining structure which is described in the following sections.

2.1 Electronic Code Book (ECB)

This cipher mode is the easiest one, because the input message (plaintext) is passed to the cryptographic block with a key and the output message (ciphertext) is obtained directly, see *Figure 2.*

This means that input message with a low variance could be encrypted in an equivalent manner and it could have a low security threshold.

Figure 2. ECB block diagram

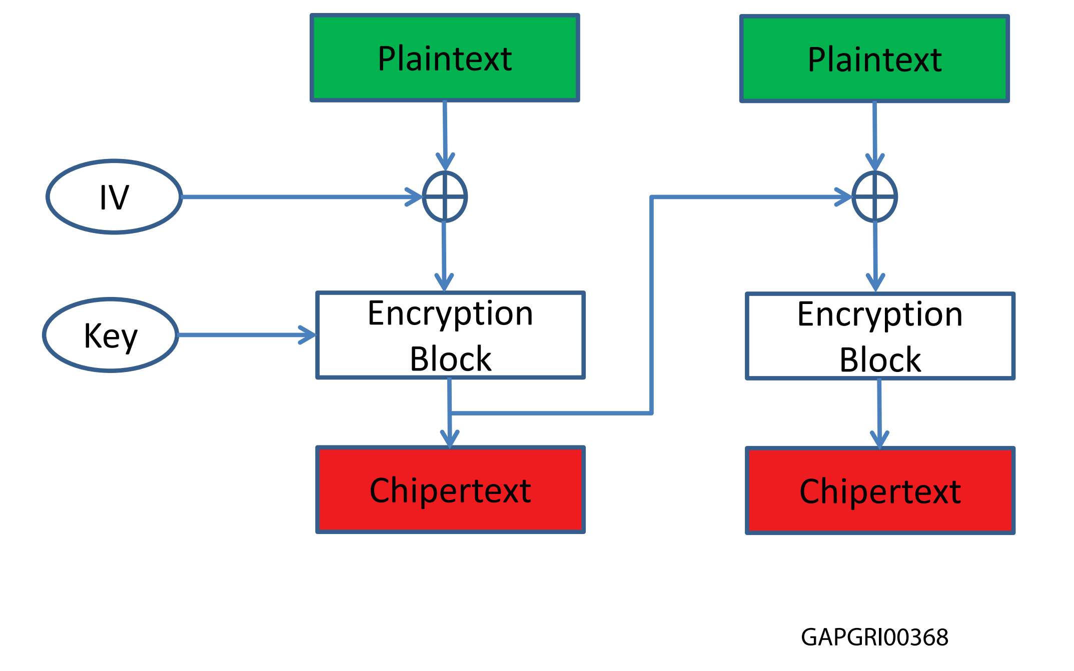


2.2 Cipher Block Chaining (CBC)

The Cipher Block Chaining mode allows an higher level of entropy because the output of the first ciphertext is derived by an initialization vector and a cryptographic key.
The chaining structure permits using the previous ciphertext as cryptographic key for the next encryption block, as described in *Figure 3.*

Using this mode, each cipher block depends on the previous processed plaintext block.

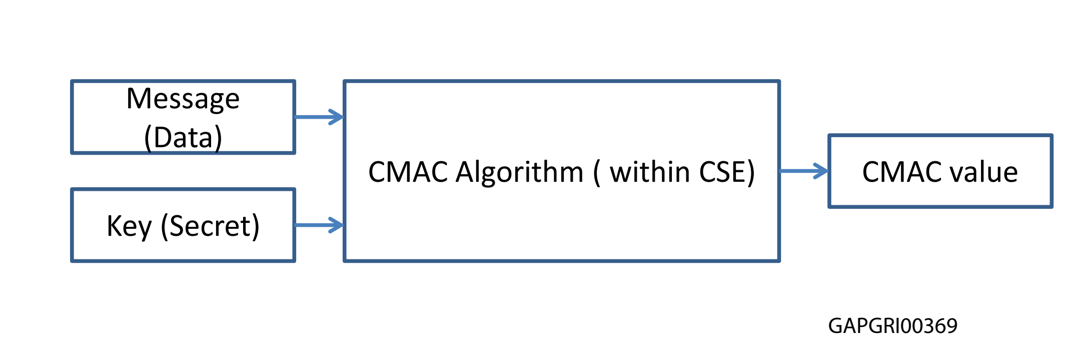
Figure 3. CBC block diagram



2.3 CMAC (Cipherbased Message Authentication Code)

A CMAC provides a method for authenticating messages and data. The CMAC algorithm accepts as input a secret key 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 change in the message content.

Figure 4. CMAC Scheme



3 Secure Boot procedure

The CSE module allows authentication boot code in flash. This sentence is misunderstood because the CSE is not intended to be used to encrypt the code flash content.

The Secure Boot procedure starts when the System Status and Configuration Module (SSCM) releases a SECURE_BOOT command. After that the CSE module downloads from the internal secure flash the firmware and the valid keys.

Note: The device must be previously configured with valid cryptographic keys in order to issue a Secure Boot.

The sequence to authenticate Boot Code is as follows:

1) program the code flash with the boot code.
This implies that the boot code includes the start address and length parameters at address RCHW+4 and RCHW+8 as shown in *Table 1*:

Table 1. Secure boot example structure to add at the start of boot sectorr

Address	Content	Comment
0x0	0x015A	Reset Configuration Half Word
0x4	0x10	Start Address for BOOT_MAC calculation
0x8	0x1000	Length of code to be authenticated in bytes (4KB)
0xC	–	Skipped for 64bit boundary
0x10	Code starts here	First address of code

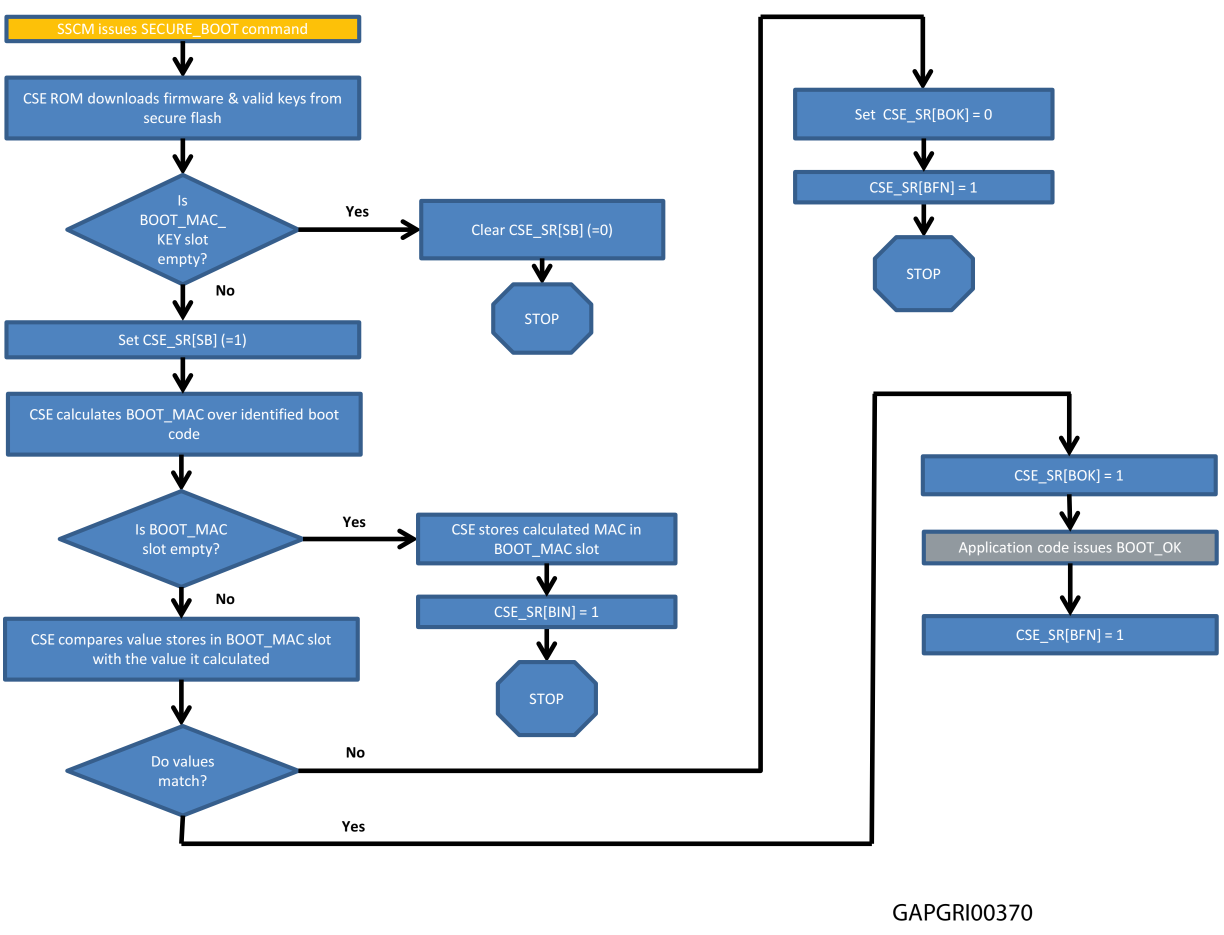
In this case the boot code starts at 0x10 and CSE authenticates 4 KB of code.

2) Program valid cryptographic keys (BOOT_MAC_KEY) into secure flash

3) Reset the device twice; the first time CSE calculates the BOOT_MAC over the indentified boot code and stores this value in the secure flash, the second time CSE compares the BOOT_MAC with the previous one stored in the secure flash. If they match then CSE sets Secure Boot OK bit (CSE.SR[BOK]=1).

After this procedure, keys marked as Boot Protected are used by application code. On subsequent booting, provided BOOT_MAC_KEY and the code flash are not erased, CSE calculates a MAC over the identified boot code. If the output value matches the value stored in secure flash (BOOT_MAC).
Figure 5. represents this process.

Figure 5. Secure Boot Mode procedure


SSCM issues SECURE_BOOT command
CSE ROM downloads firmware & valid keys from secure flash
Is BOOT_MAC_ KEY slot empty?
Yes
Clear CSE_SR[SB] (=0)
STOP
No
Set CSE_SR[SB] (=1)
CSE calculates BOOT_MAC over identified boot code
Is BOOT_MAC slot empty?
Yes
CSE stores calculated MAC in BOOT_MAC slot
CSE_SR[BIN] = 1
STOP
No
CSE compares value stores in BOOT_MAC slot with the value it calculated
Do values match?
No
Yes
Set CSE_SR[BOK] = 0
CSE_SR[BFN] = 1
STOP
CSE_SR[BOK] = 1
Application code issues BOOT_OK
CSE_SR[BFN] = 1
GAPGRI00370

4 Secure storage for cryptographic keys

The CSE provides secure, and non-volatile storage for cryptographic keys as described in the *SHE Functional Specification*. The keys are stored in fifteen memory slots, with one ROM slot, thirteen non-volatile slots, and one RAM slot as shown in *Table 2.* The first four slots have a dedicated usage, the other slots are available for application specific keys. The BOOT_MAC slot is loaded with a MAC value used by the secure boot process. All other slots are used for encryption or message authentication keys. The SECRET_KEY slot is programmed with a random value during device fabrication same as the Unique Identifier Number (UID). It is unique for every part and is programmed into the secure flash when it is tested in wafer form. UID is 120 bits long. UID is used during inter ECU communications to confirm that external controllers is not substituted. SECRET KEY may only be used to import/export keys.

All CSE encryption and message authentication commands specify a key, by its Key ID.

Table 2. Memory Slots

Slot Name	Key ID	Type
SECRET_KEY	0x0	ROM
MASTER_ECU_KEY	0x1	non-volatile
BOOT_MAC_KEY	0x2	non-volatile
BOOT_MAC	0x3	non-volatile
KEY_1	0x4	non-volatile
KEY_2	0x5	non-volatile
KEY_3	0x6	non-volatile
KEY_4	0x7	non-volatile
KEY_5	0x8	non-volatile
KEY_6	0x9	non-volatile
KEY_7	0xA	non-volatile
KEY_8	0xB	non-volatile
KEY_9	0xC	non-volatile
KEY_10	0xD	non-volatile
RAM_KEY	0xE	RAM

Table 3. describes that each memory slot holds a 128-bit value, a 28-bit counter and five security flags.

Table 3. Key attributes

Flag Name	Description
WRITE_PROT	If set, memory slot cannot be updated
BOOT_PROT	If set, memory slot is disabled if Secure Boot is not enabled
DEBUG_PROT	If set. memory slot is disabled if a debugger is connected

Table 3. Key attributes (continued)

Flag Name	Description
KEY_USAGE	If set, memory slot holds a MAC key, otherwise it holds an encryption key
WILDCARD	If set, memory slot cannot be updated with the wildcard UID

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 is used to authorize updating other keys (BOOT_MAC_KEY, BOOT_MAC, BOOT_MAC_KEY and all KEY_1 to KEY_10) without knowledge of those keys. See *Table 4: Memory Update Policy* of the SHE specification, reported here for convenience:

Table 4. Memory Update Policy

Slot to update	MASTER ECU KEY	BOOT MAC KEY	BOOT MAC	KEY<n>	RAM KEY
MASTER ECU KEY	X				
BOOT MAC KEY	X	X			
BOOT MAC	X	X			
KEY<n>	X			X	
RAM KEY				X	

Table 4 shows that the knowledge of the MASTER ECU KEY allows updating of all the other keys. This implies that it is the most important key as it allows to reset the device to its factory state, (with all the security memory slots empty) only if no key has the WRITE_PROT flag set.

Setting this flag is an irreversible step and for this reason, the user has to be very carefull during the creation of the key attributes of the cryptographic keys.

5 Application code structure

This section gives an overview of the application code implemented to test some CSE features as the encryption/decryption using ECB and CBC cipher modes and an example code in order to release a Secure Boot loading some cryptographic keys in the secure slots.

5.1 Peripherals and tools used

- Device: SPC564xB/C
- Compiler: GHS v6.1
- Debugger: Lauterbach
- System clock: 120 MHz by PLL initialization
- CGM (Clock Generation Module): PLL is configured as system clock
- CSE module
- ME: Mode entry in order to configure the Magic Carpet
- SWT: Software Watchdog in order to disable the periodic reset

5.2 Code Implementation

In order to load the cryptographic keys in the secure flash of the device, to release a secure boot and then start the demo application, three different GHS projects are implemented.

First project is needed to load the cryptographic keys obtained off line, using an executable file precompiled in order to obtain all the five “M” parameters and the 2 K parameters to generate 10 user Crypto Keys, BOOT_MAC_KEY and MASTER_ECU_KEY. For more information read the *Appendix A* at the end of this document.

Second project is used to release a Secure Boot OK, adding in the start up file and in the locator file, few lines of code and implementing the Secure Boot Procedure as describe in *Figure 5.*

Third project is the demo application code which shows the main differences between the ECB and CBC cipher mode, as described in the previous paragraph.

The first application code configures the device in order to have the following:

- Basic configuration procedure:
  - Disable Software Watchdog
  - Initialize Magic Carpet
  - initialize PLL to 120 MHz from 40 MHz of the external oscillator
- Initialization of the CSE module
- Updating a blank sample and loading the cryptographic keys into the secure flash
  - Issue a Get UID command
  - Issue a LOAD KEY command

The second application code configuress the device in order to:

- Add in the ctr0.ppc file the code size and the start address of the code to be secured
- Add in the locator file the reserved section needed to release the secure boot
- Reset the device two times in order to issue Secure Boot OK, as described in the Secure boot procedure *Figure 5.*

Finally the third application code configures the device in order to:

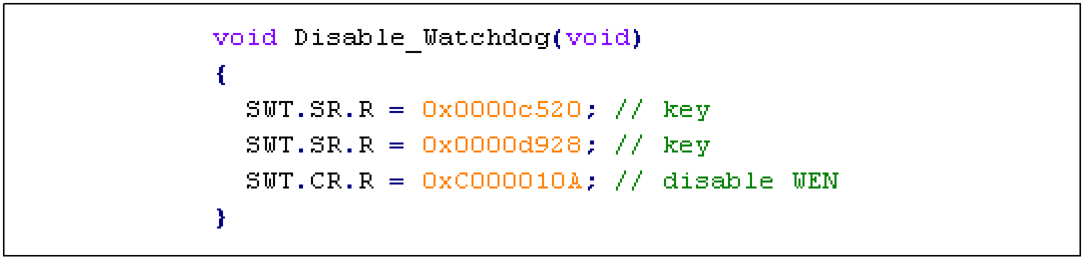
- Issue a CSE BOOT OK command to verify if the content of the flash is not modified
- Issue an encryption ECB and a CBC command over a preloaded figure (ST Logo)
- Issue a decryption ECB and a CBC command over the encrypted figure to show the plain text figure again

5.2.1 Disable Software Watchdog

This function allows disabling the Software Watchdog using a key word and permit the access to the control register of the SWT module.

The SWT is disabled in order to avoid the period execution of watchdog service that could issue a system reset:

Figure 6. Software Watchdog disable function

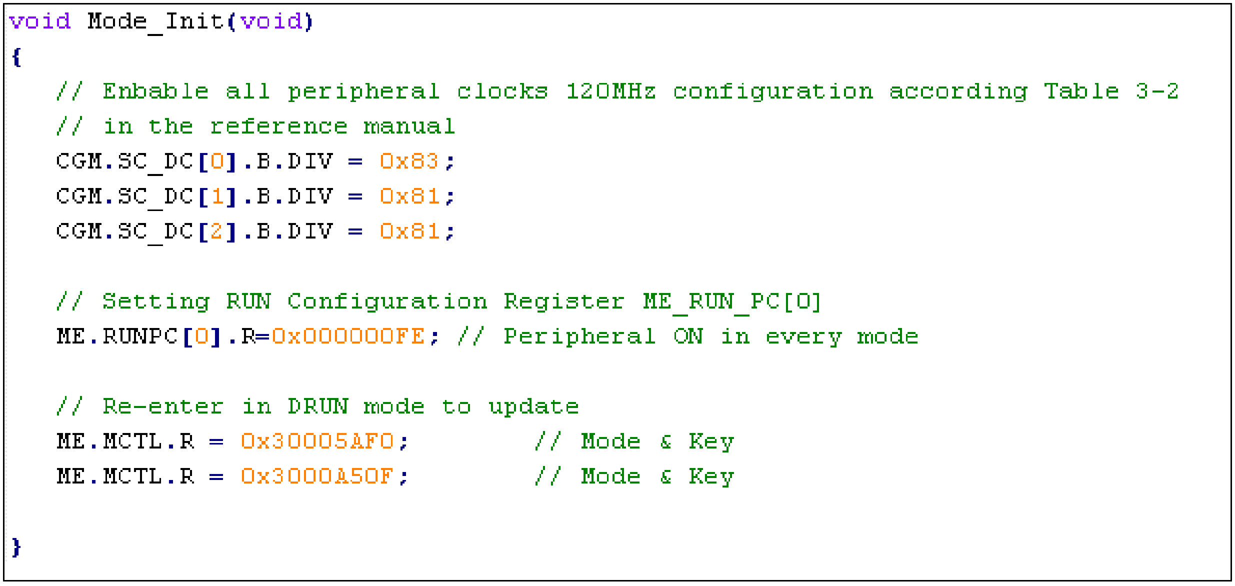


```
void Disable_Watchdog(void)
{
  SWT.SR.R = 0x0000c520; // key
  SWT.SR.R = 0x0000d928; // key
  SWT.CR.R = 0xC000010A; // disable WEN
}
```

5.2.2 Mode Init

This function allows to configure the mode entry of the device in order to turn on all the peripherals considering the right dividers for a system clock of 120 MHz.

Figure 7. Mode Init function



```
void Mode_Init(void)
{
   // Enbable all peripheral clocks 120MHz configuration according Table 3-2
   // in the reference manual
   CGM.SC_DC[0].B.DIV = 0x83;
   CGM.SC_DC[1].B.DIV = 0x81;
   CGM.SC_DC[2].B.DIV = 0x81;

   // Setting RUN Configuration Register ME_RUN_PC[0]
   ME.RUNPC[0].R=0x000000FE; // Peripheral ON in every mode

   // Re-enter in DRUN mode to update
   ME.MCTL.R = 0x30005AF0;        // Mode & Key
   ME.MCTL.R = 0x3000A50F;        // Mode & Key

}
```

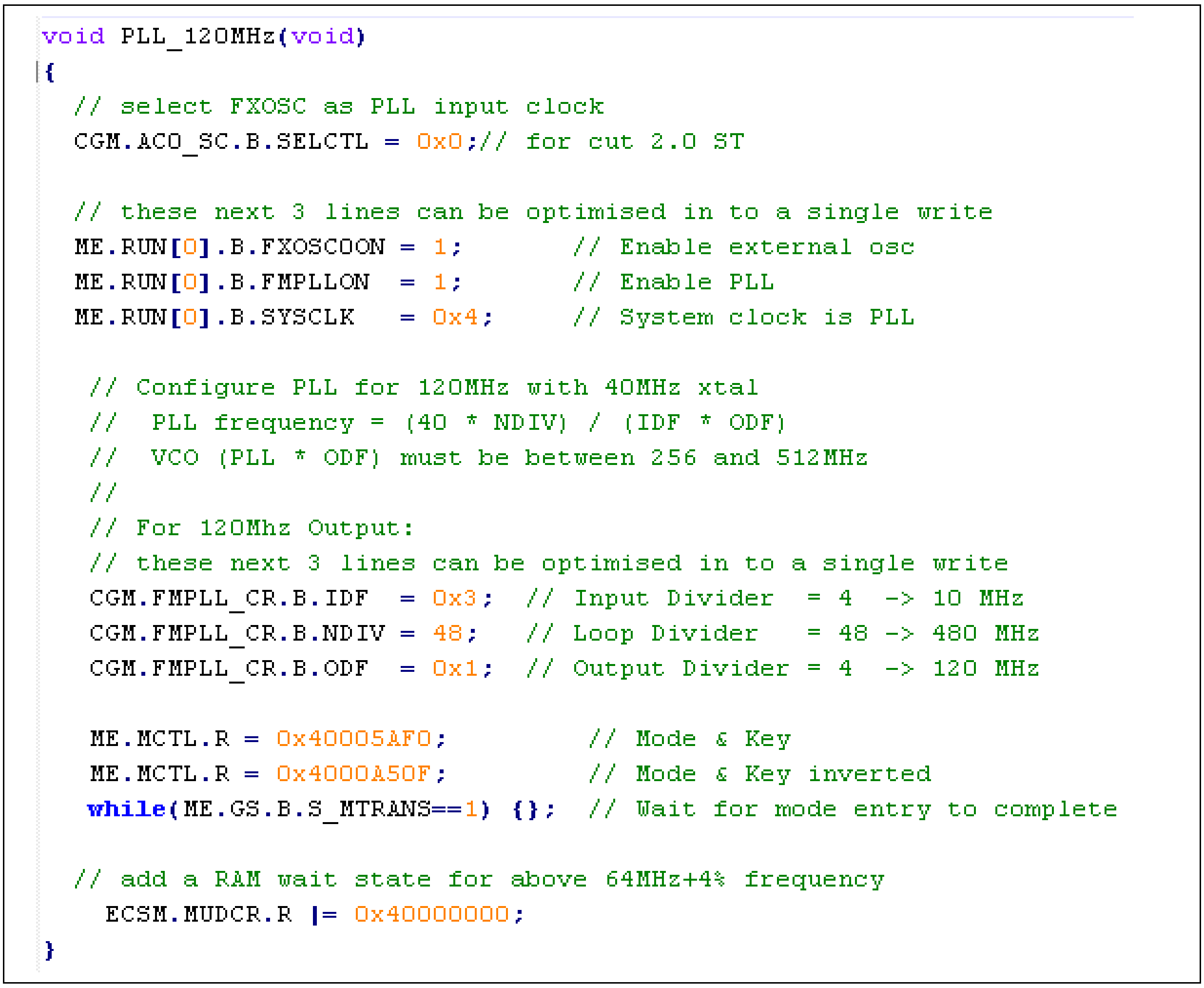
5.2.3 PLL configuration function

This function permits to select the clock source for the PLL. In this case the PLL is driven by the FXOSC oscillator at 40 MHz. Then it configures the RUN[0] mode in order to enable the external oscillator, turning on the PLL and setting the PLL as system clock.

The CGM module allows to set several dividers (Input, loop and output) in order to set the ouput of the PLL to be at 120 MHz frequency.

In order to complete the mode entry two keys are written in the Mode Entry MCTL register.

Figure 8. PLL init function

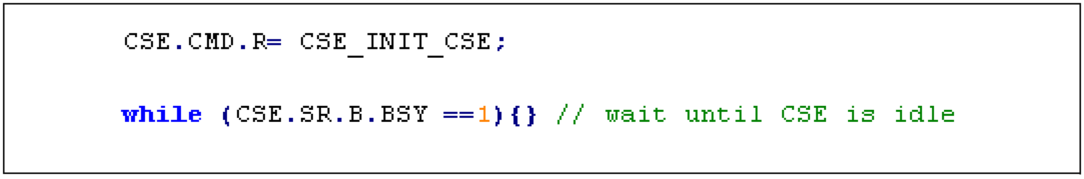


```
void PLL_120MHz(void)
{
  // select FXOSC as PLL input clock
  CGM.AC0_SC.B.SELCTL = 0x0;// for cut 2.0 ST

  // these next 3 lines can be optimised in to a single write
  ME.RUN[0].B.FXOSC0ON = 1;       // Enable external osc
  ME.RUN[0].B.FMPLLON  = 1;       // Enable PLL
  ME.RUN[0].B.SYSCLK   = 0x4;     // System clock is PLL

   // Configure PLL for 120MHz with 40MHz xtal
   //  PLL frequency = (40 * NDIV) / (IDF * ODF)
   //  VCO (PLL * ODF) must be between 256 and 512MHz
   //
   // For 120Mhz Output:
   // these next 3 lines can be optimised in to a single write
   CGM.FMPLL_CR.B.IDF  = 0x3;  // Input Divider  = 4  -> 10 MHz
   CGM.FMPLL_CR.B.NDIV = 48;   // Loop Divider   = 48 -> 480 MHz
   CGM.FMPLL_CR.B.ODF  = 0x1;  // Output Divider = 4  -> 120 MHz

   ME.MCTL.R = 0x40005AF0;          // Mode & Key
   ME.MCTL.R = 0x4000A50F;          // Mode & Key inverted
   while(ME.GS.B.S_MTRANS==1) {};   // Wait for mode entry to complete

  // add a RAM wait state for above 64MHz+4% frequency
    ECSM.MUDCR.R |= 0x40000000;
}
```

5.2.4 CSE initialization function

The INIT_CSE command loads the command processor firmware and memory slot data from the CSE Flash blocks into local memory. It does not execute the secure boot protocol. The CSE firmware version is loaded into the CSE_P1 register. This command must be issued before any other command when using device boot modes, that do not support secure boot.

Figure 9. CSE init function

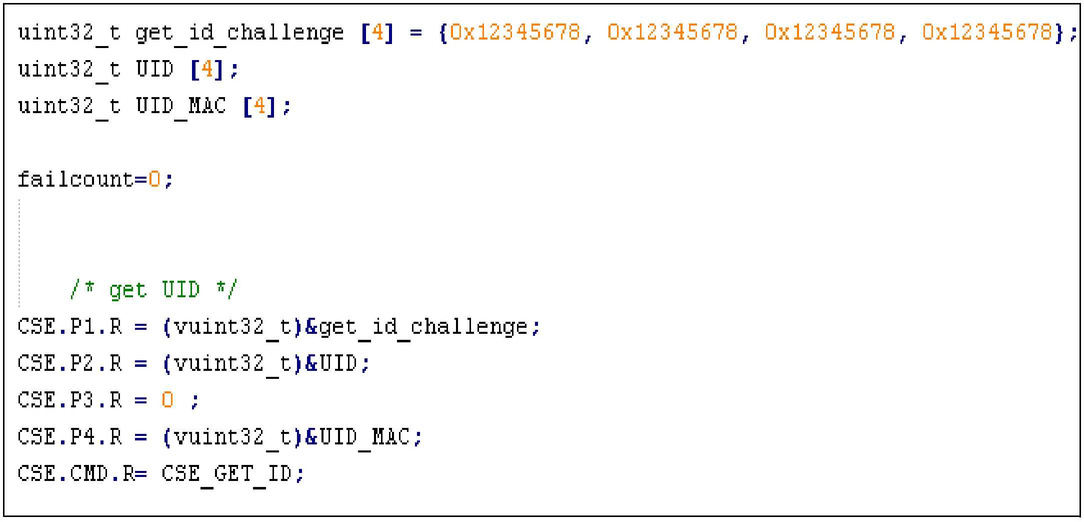


```
CSE.CMD.R= CSE_INIT_CSE;

while (CSE.SR.B.BSY ==1){} // wait until CSE is idle
```

5.2.5 Load Crypto Keys function

The update blank part function allows to get the UID value of the device, a 120-bit read only unique identification number which is programmed during device fabrication. The UID is used in the memory update procedure and is available for application specific uses.

Figure 10. Get UID function

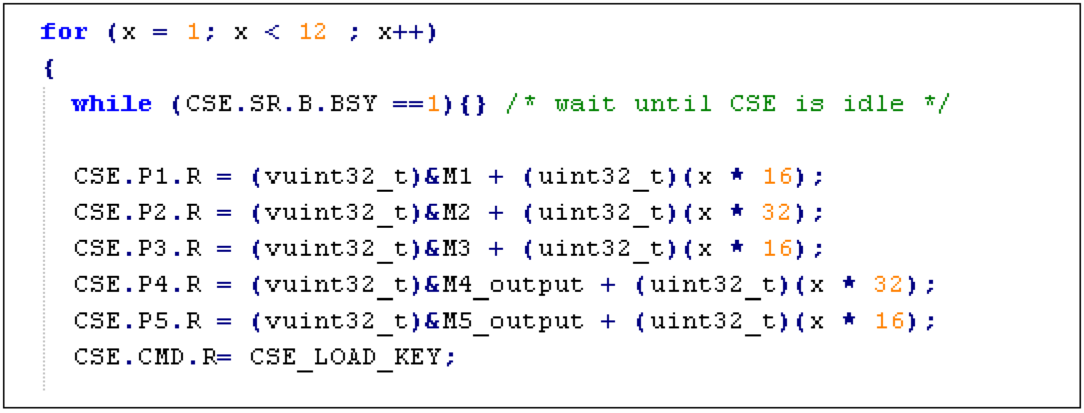


```
uint32_t get_id_challenge [4] = {0x12345678, 0x12345678, 0x12345678, 0x12345678};
uint32_t UID [4];
uint32_t UID_MAC [4];

failcount=0;


    /* get UID */
CSE.P1.R = (vuint32_t)&get_id_challenge;
CSE.P2.R = (vuint32_t)&UID;
CSE.P3.R = 0 ;
CSE.P4.R = (vuint32_t)&UID_MAC;
CSE.CMD.R= CSE_GET_ID;
```

The UID value is derived here in order to be used later to check if the updating procedure has been processed correctly.

Furthermore, few lines of code is used to load the Cryptographic keys in order to update eleven keys: ten USER KEYS and BOOT_MAC_KEY.

Figure 11. Load Key function



```
  for (x = 1; x < 12 ; x++)
  {
    while (CSE.SR.B.BSY ==1){} /* wait until CSE is idle */

    CSE.P1.R = (vuint32_t)&M1 + (uint32_t)(x * 16);
    CSE.P2.R = (vuint32_t)&M2 + (uint32_t)(x * 32);
    CSE.P3.R = (vuint32_t)&M3 + (uint32_t)(x * 16);
    CSE.P4.R = (vuint32_t)&M4_output + (uint32_t)(x * 32);
    CSE.P5.R = (vuint32_t)&M5_output + (uint32_t)(x * 16);
    CSE.CMD.R= CSE_LOAD_KEY;
```

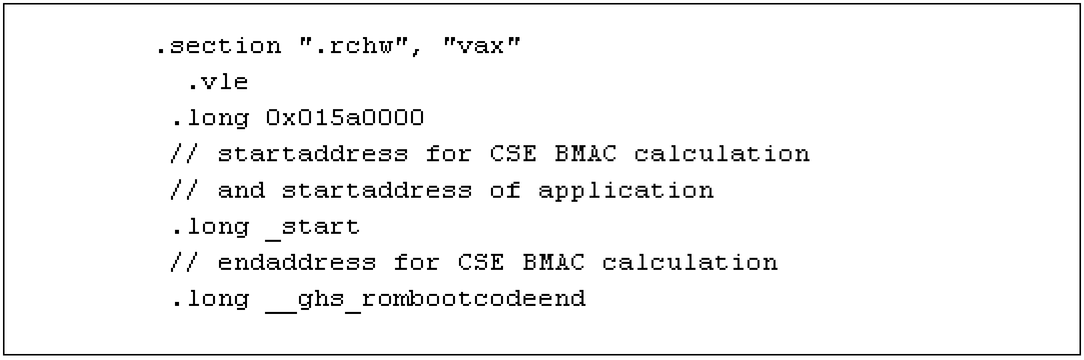
In this case, M1 to M5 parameters have been calculated off line. In order to check that the updating procedure was performed correctly, CSE calculates M4 and M5 to compare those values with the values obtained off line.

How to obtain those parameters is explained in the *Appendix A*.

5.2.6 App code to be added to have a secure boot

In order to issue a Secure Boot OK it is necessary to add in the crt0.ppc file, the following lines:

Figure 12. Secure boot code

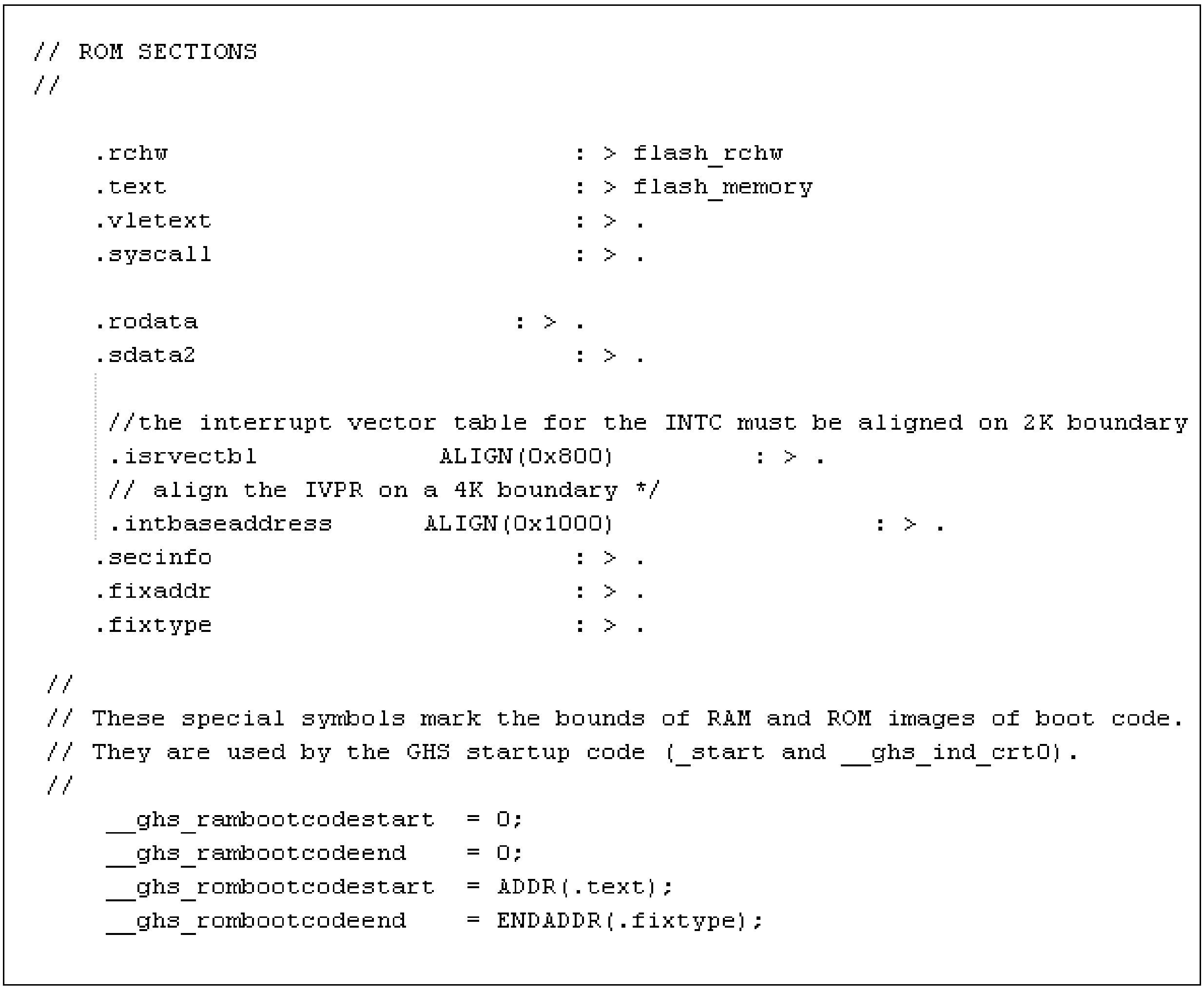


```
        .section ".rchw", "vax"
          .vle
         .long 0x015a0000
         // startaddress for CSE BMAC calculation
         // and startaddress of application
         .long _start
         // endaddress for CSE BMAC calculation
         .long __ghs_rombootcodeend
```

The few lines of code create a section called rchw (reset configuration half word) just at the first address of the CFLASH memory adding the start address for BOOT CMAC calculation, the start address of the application and the lenght of the code to be authenticated, as described in *Figure 12*.

In the locator file it is also necessary add the same section at the beginning of CFLASH and define the size of the code to be authenticated, as shown *Figure 13.*:

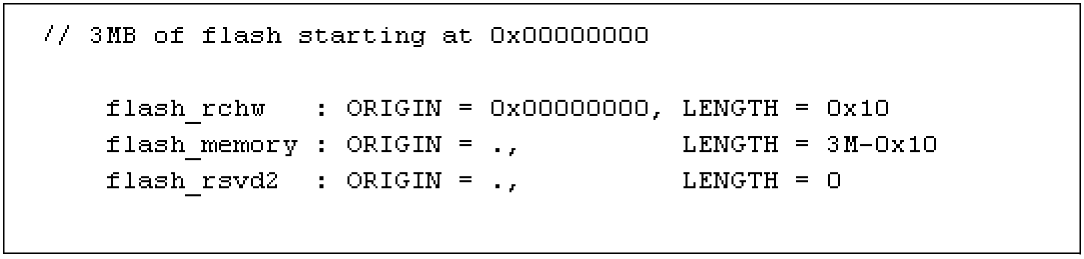
Figure 13. Locator file



```
// ROM SECTIONS
//

    .rchw                               : > flash_rchw
    .text                               : > flash_memory
    .vletext                            : > .
    .syscall                            : > .

    .rodata                         : > .
    .sdata2                             : > .

     //the interrupt vector table for the INTC must be aligned on 2K boundary
     .isrvectbl              ALIGN(0x800)         : > .
     // align the IVPR on a 4K boundary */
     .intbaseaddress       ALIGN(0x1000)                 : > .
    .secinfo                            : > .
    .fixaddr                            : > .
    .fixtype                            : > .

 //
 // These special symbols mark the bounds of RAM and ROM images of boot code.
 // They are used by the GHS startup code (_start and __ghs_ind_crt0).
 //
     __ghs_rambootcodestart  = 0;
     __ghs_rambootcodeend    = 0;
     __ghs_rombootcodestart  = ADDR(.text);
     __ghs_rombootcodeend    = ENDADDR(.fixtype);
```

where:

Figure 14. Locator memory setup



```
// 3MB of flash starting at 0x00000000

    flash_rchw    : ORIGIN = 0x00000000, LENGTH = 0x10
    flash_memory : ORIGIN = .,          LENGTH = 3M-0x10
    flash_rsvd2   : ORIGIN = .,          LENGTH = 0
```

Once the BOOT_MAC_KEY is loaded in the device, the CSE is able to calculate the CMAC of the secure boot as defined in the startup file after a first reset. With a second reset the CSE will check if the content of the flash has been modified, comparing the CMAC previously calculated over the authenticated code versus the new CMAC calculated over the same flash content. If they match the CSE release a Secure Boot OK.

5.2.7 Demo application code

As described before the last project allows, once CSE has released a secure boot, to use the cryptographic keys to encrypt and decrypt a logo.

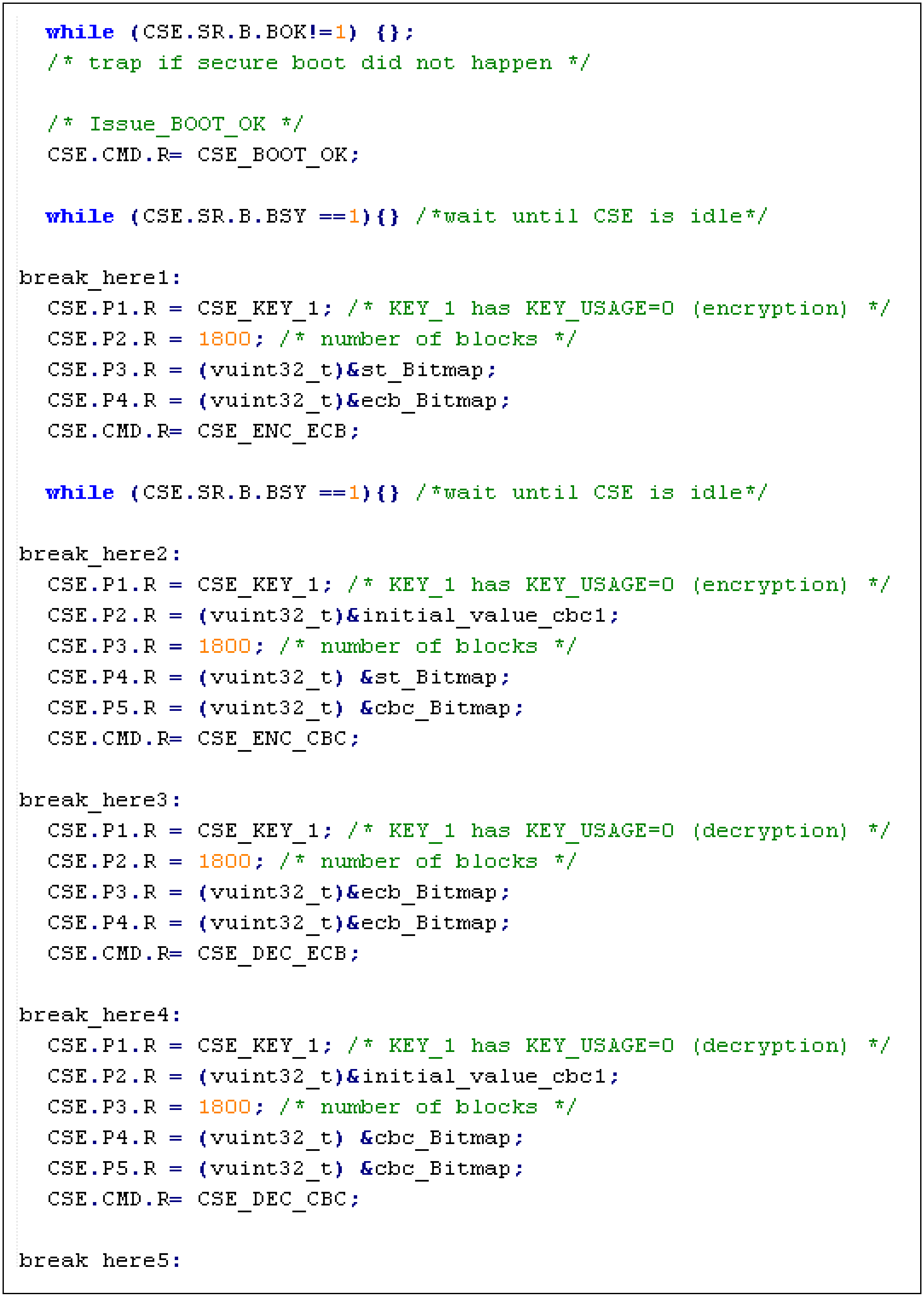
The following lines of code permit the encryption and decryption commands using the ECB and CBC cipher modes over a preloaded image.

The Cryptographic key used, is the KEY1 which has the KEY USAGE flag equal to 0 in order to be used for encryption/decryption feature.

For the CBC cipher mode, an initialize vector is used to add more entropy to the process.

The st_Bitmap is the ST logo of 1800 blocks size obtained using an image bitmap converter.

Figure 15. Encryption/Decryption functions

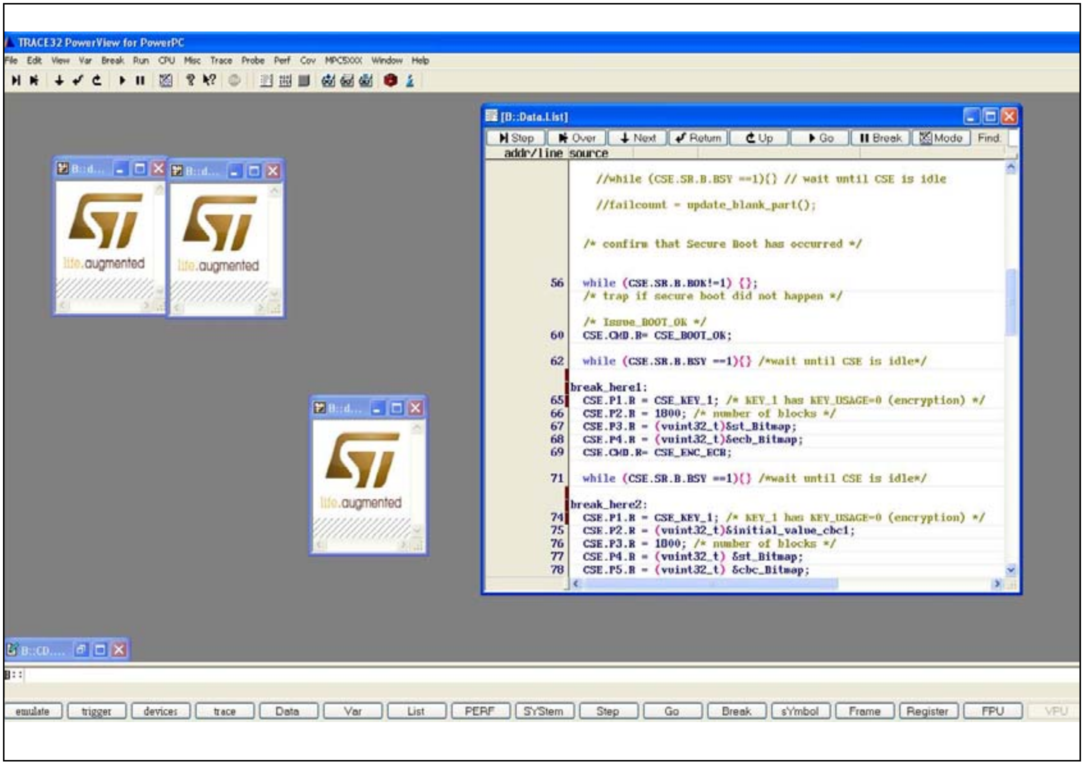


```
  while (CSE.SR.B.BOK!=1) {};
  /* trap if secure boot did not happen */

  /* Issue_BOOT_OK */
  CSE.CMD.R= CSE_BOOT_OK;

  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here1:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&st_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_ENC_ECB;

  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here2:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &st_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_ENC_CBC;

break_here3:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&ecb_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_DEC_ECB;

break_here4:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &cbc_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_DEC_CBC;

break_here5:
```

6 Application demo results

To better understand the differences between the two cipher modes the following figures will show an ECB encryption and a CBC encryption starting from the same image:

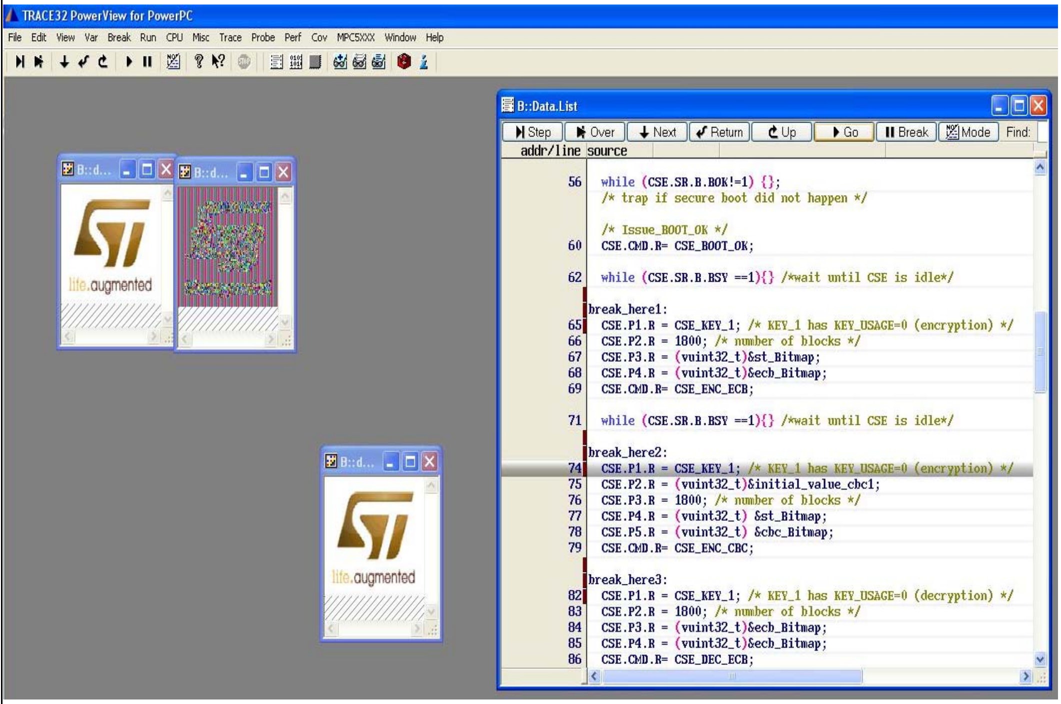
Figure 16. Demo Starting point


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
[B::Data.List]
Stop Over Next Return Up Go Break Mode Find
addr/line source
B::d...
life.augmented
B::CD....
B::
emulate trigger devices trace Data Var List PERF SYStem Step Go Break sYmbol Frame Register FPU VPU

```
     //while (CSE.SR.B.BSY ==1){} // wait until CSE is idle

     //failcount = update_blank_part();


   /* confirm that Secure Boot has occurred */


   while (CSE.SR.B.BOK!=1) {};
   /* trap if secure boot did not happen */

   /* Issue_BOOT_OK */
   CSE.CMD.R= CSE_BOOT_OK;

   while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here1:
   CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
   CSE.P2.R = 1800; /* number of blocks */
   CSE.P3.R = (vuint32_t)&st_Bitmap;
   CSE.P4.R = (vuint32_t)&ecb_Bitmap;
   CSE.CMD.R= CSE_ENC_ECB;

   while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here2:
   CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
   CSE.P2.R = (vuint32_t)&initial_value_cbc1;
   CSE.P3.R = 1800; /* number of blocks */
   CSE.P4.R = (vuint32_t) &st_Bitmap;
   CSE.P5.R = (vuint32_t) &cbc_Bitmap;
```

Figure 16 shows that the ECB encryption should not be suggested for encryption of message with a low variance because, for the same nature of the algorithm, identical plaintext blocks of messages are encrypted into identical ciphertext blocks. This does not hide the data pattern well. In some sense it does not provide a high level of confidentiality and security.

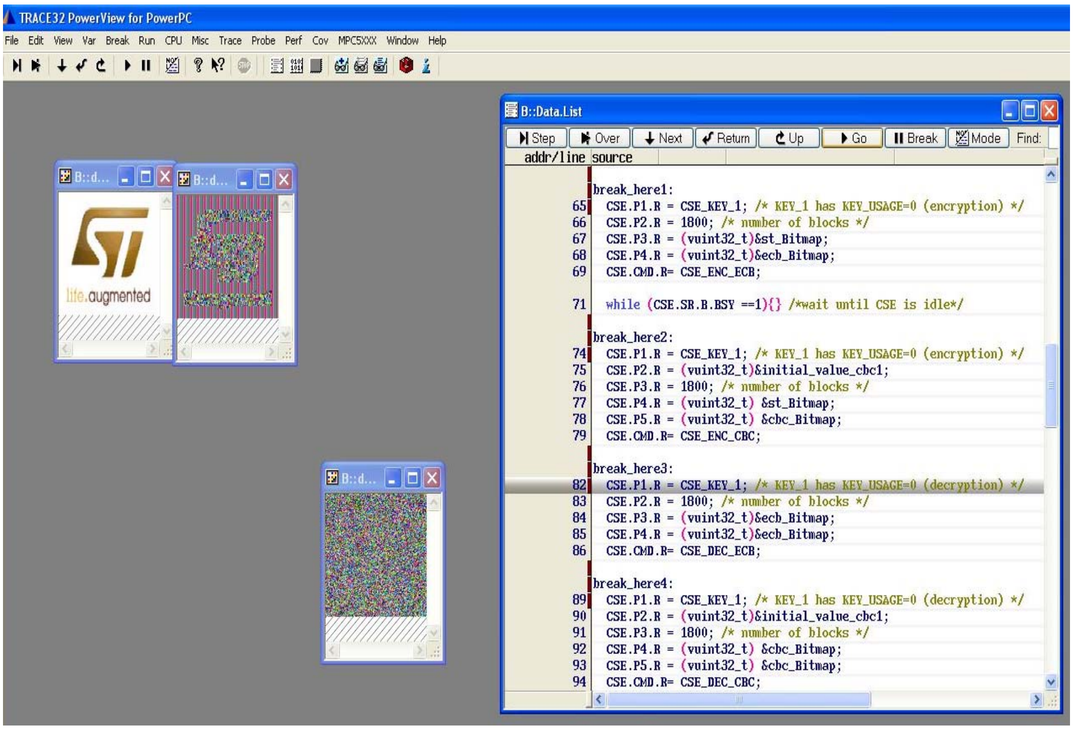
Figure 17. ECB encryption



```
  while (CSE.SR.B.BOK!=1) {};
  /* trap if secure boot did not happen */

  /* Issue_BOOT_OK */
  CSE.CMD.R= CSE_BOOT_OK;

  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here1:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&st_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_ENC_ECB;

  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here2:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &st_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_ENC_CBC;

break_here3:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&ecb_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_DEC_ECB;
```

While in the CBC encryption, having a chaining structure, each block of plaintext is XORed with the previous ciphertext block before encryption. In this way, each ciphertext block is independent on all plaintext block processed up to that point. To make each message unique, an initialization vector (IV) must be used in the first block.

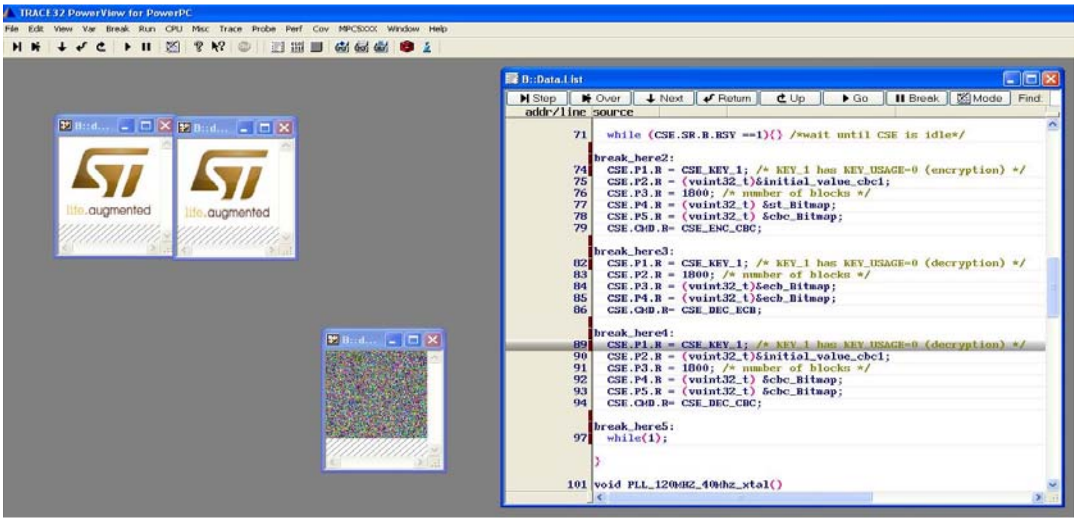
Figure 18. CBC encryption


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::d...
life.augmented
B::Data.List
Step Over Next Return Up Go Break Mode Find:
addr/line source

```
break_here1:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&st_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_ENC_ECB;

  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here2:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &st_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_ENC_CBC;

break_here3:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&ecb_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_DEC_ECB;

break_here4:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &cbc_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_DEC_CBC;
```

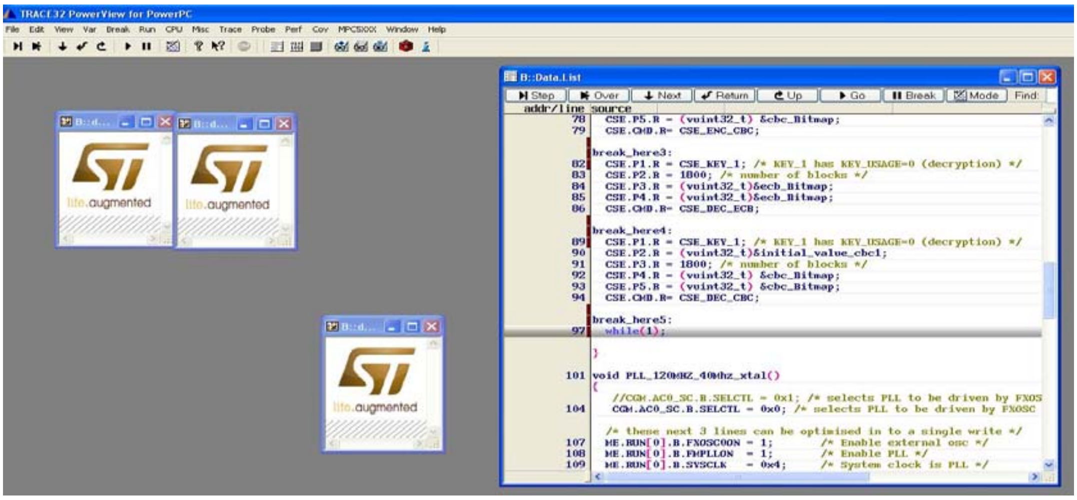
Here are the following decryption of the encrypted images:

Figure 19. ECB decryption


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::d...
life.augmented
life.augmented
B::Data.List
Step Over Next Return Up Go Break Mode Find:
addr/line source

```
  while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

break_here2:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (encryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &st_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_ENC_CBC;

break_here3:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&ecb_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_DEC_ECB;

break_here4:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &cbc_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_DEC_CBC;

break_here5:
  while(1);

}

void PLL_120MHZ_40Mhz_xtal()
```

Figure 20. CBC decryption


TRACE32 PowerView for PowerPC
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::d...
life.augmented
B::Data.List
Step Over Next Return Up Go Break Mode Find:
addr/line source

```
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_ENC_CBC;

break_here3:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = 1800; /* number of blocks */
  CSE.P3.R = (vuint32_t)&ecb_Bitmap;
  CSE.P4.R = (vuint32_t)&ecb_Bitmap;
  CSE.CMD.R= CSE_DEC_ECB;

break_here4:
  CSE.P1.R = CSE_KEY_1; /* KEY_1 has KEY_USAGE=0 (decryption) */
  CSE.P2.R = (vuint32_t)&initial_value_cbc1;
  CSE.P3.R = 1800; /* number of blocks */
  CSE.P4.R = (vuint32_t) &cbc_Bitmap;
  CSE.P5.R = (vuint32_t) &cbc_Bitmap;
  CSE.CMD.R= CSE_DEC_CBC;

break_here5:
  while(1);

}

void PLL_120MHZ_40Mhz_xtal()
{
   //CGM.AC0_SC.B.SELCTL = 0x1; /* selects PLL to be driven by FXOS
   CGM.AC0_SC.B.SELCTL = 0x0; /* selects PLL to be driven by FXOSC

  /* these next 3 lines can be optimised in to a single write */
  ME.RUN[0].B.FXOSCOON = 1;        /* Enable external osc */
  ME.RUN[0].B.FMPLLON  = 1;        /* Enable PLL */
  ME.RUN[0].B.SYSCLK   = 0x4;      /* System clock is PLL */
```

7 Conclusion

To protect the cryptographic keys from software attacks, the control over those keys are moved from the software domain to the hardware domain. The SPC564xB/C device is the first ST device which offers 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 CSE module implements and how these features are used.

The three example codes show a mini-life cycle in order to load cryptographic keys, issue a secure boot and encrypt and decrypt an image using two cipher modes, showing the superiority of the CBC cipher mode against the ECB one.

Appendix A How to generate the M1-M5 parameters

A.1 Memory update protocol

SHE requires that in order to update the memory containing the keys are the following 5 parameters must be calculated and passed to CSE:

- K1 = KDF(K_{AuthID}, KEY_UPDATE_ENC_C)
  - KDF is defined as key derivation function
  - KAuthID is Authorizing key value. Part from factory has no keys programmed. In this case AuthID = ID (for example Authorizing key is the key itself) is used
  - KEY_UPDATE_ENC_C is a constant value defined by SHE as:
  - 0x01015348_45008000_00000000_000000B0
- K2 = KDF(K_{AuthID}, KEY_UPDATE_MAC_C)
  - KEY_UPDATE_MAC_C is a constant value defined by SHE as:
  - 0x01025348_45008000_00000000_000000B0
- M1 = UID’|ID|AuthID - 128 bits
  - AuthID is either ID (number of key being updated) or MASTER_ECU_KEY number(0x1)
  - UID is 0 (Wildcard value) because WC flag = 0 on parts from factory
  - UID is 120 bit and ID and AuthID are 4 bits each
  - ID is the identification number of key we want to update
- M2 = ENCCBC,K1,IV=0(CID’|FID’|“0...0"95|KID’) - 256 bits
  - ENCCBC is the encryption using K1 (as defined) with IV = 0
  - The message to encrypt is a concatenation of :
  - CID : the new counter value (28 bit). 0x0000001 in this case
  - FID : New Protection flags –WP|BP|DP|KU|WC (5 bits)
  - 95 zeros to fill first 128 bit block with zeros
  - KID : the new key value (128 bit)
- M3 = CMACK2(M1|M2) – 128 bits
  - A CMAC is performed using K2 over concatenation of M1 and M2 (128+256 = 384 bit input size)

To complete the list, other parameters are needed to calculate offline M4 and M5 and make a match to check if the cryptographic keys updating procedure has been performed correctly:

- K3 = KDF(KEYID, KEY_UPDATE_ENC_C)
  - KEYID is the new key value
- K4 = KDF(KEYID, KEY_UPDATE_MAC_C)
- M4 = UID|ID|AuthID|M4* (256 bit size)
  - UID : Unique ID (120 bit)
  - ID: Identification number of key to be updated (4 bits)
  - AuthID: Identification number of key authorizing the update (4 bits)
- M4*: ENC_ECB, K3(CID|CIDPAD)
  - where an ECB encryption is performed using K3 as key over the concatenation of:
  - CIDPAD = 0x8000000000000000000000000 (1 and 99 0’s)

  - CID = counter value (28 bit)
- M5 = CMAC,K4(M4) (128 bit size)
  - A CMAC is performed over using key K4 over M4

Note: *If a key has it is Write Protect (WP) attribute set, the key cannot ever be updated or erased. See Table 3. Key Attributes. Write Protection should only be used when the user is absolutely certain that the key is never changed or erased. Setting Write Protection on any single key means that the part cannot be reset to its factory state using the DEBUG CHALLENGE/AUTHORIZATION sequence.*

In order to generate M1-M5 parameters some precompiled executable files have been used.

The following sources were downloaded from www.hoozi.com and modified. Original author is Niyaz PK.

AES_ENC_CBC_CMD.c

AES_ENC_ECB_CMD.c

AES_MP_KDF_CMD.c

The following sources was based on an original program by Junhyuk Son and Jicheol Lee:

AES_CMAC_CMD.c

A.2 Cryptographic keys used

For the demo code, a set of pre-calculated keys and values are used which are shown in *Table 5.* These values found in some of the header files provided with the examples. To use user-defined keys, the user needs to use offline scripts to calculate the necessary values.

Table 5. Examples of Keys used for the project

Slot name	Key ID [hex]	Address offset [hex]	Key flags [bin]	128-bit key word 0 word 1word 2 word 3
BOOT_MAC_KEY	0x2	0x060	00011	12340000 00000000 00000000 00005678
KEY_1	0x4	0x0A0	00001	2FF8B03C 5C540546 5A9C94BD 2D863279
KEY_2	0x5	0x0C0	00001	85852FF8 E7860C89 B3AB9D63 B8D6288F
KEY_3	0x6	0x0E0	01001	A36FF144 FB6D5E2C DA0D2894 DA0D2894
KEY_4	0x7	0x100	00101	86078C1A BCDCC6B6 C52C851D E5652BF5
KEY_5	0x8	0x120	00011	043A1A50 DB3954D2 22FEB37F 1F678FCA
KEY_6	0x9	0x140	00001	4B957750 4B957750 6F75C3E0 5C8DCD59
KEY_7	0xA	0x160	00011	2B7E1516 28AED2A6 ABF71588 09CF4F3C
KEY_8	0xB	0x180	00111	10AF4B5B 024195B9 1730D7F5 94C87E19
KEY_9	0xC	0x1A0	00001	93346F4C 6A8ABCCD 37D52249 291F4138
KEY_10	0xD	0x1C0	01011	68B674CB 8198A250 3A285100 F4DDC40A

Appendix B References

B.1 Reference documents

- *SPC564Bxx, SPC56ECxx 32-bit MCU family built on the embedded Power Architecture®* (RM0070, Doc ID 18196)
- *32-bit MCU family built on the Power Architecture® for automotive body electronics applications* (SPC564Bxx, SPC56ECxx Data sheet, Doc ID 17478)
- SHE - Secure Hardware Extension functional specification Version1.1 (rev 439) available on www.automotive-his.de
- [FIPS197] NIST/FIPS: Announcing the Advanced Encryption Standard (AES); November 26, 2001; http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

Revision history

Table 6. Revision history

Date	Revision	Changes
17-May-2013	1	Initial release.
17-Sep-2013	2	Updated Disclaimer.

Please Read Carefully:

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan -
Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com