

S32K14x FlexNVM & CSEc Workshop

John Floros, Alejandro Cervantes

FAEs

October 2018 | AMF-AUT-T3374



SECURE CONNECTIONS
FOR A SMARTER WORLD

Company External – NXP, the NXP logo, and NXP secure connections for a smarter world are trademarks of NXP B.V. All other product or service names are the property of their respective owners. © 2018 NXP B.V.

Agenda

- Introduction of FlexNVM
- Why do we need security?
- NXP Layered security model
- S32K Overview
- Lab #1 Enable the S32K CSEc
- SHE Specification Overview
- Lab #2 How to Store Secret Keys
- CSEc Details
- Lab #3 Encrypt Image
- Lab #4 Erase CSEc Keys
- Lab #5 Disable CSEc
- Use Cases

S32K Overview



S32K144 Block Diagram

High performance

- ARM Cortex M4F up to 112MHz w FPU
- eDMA from 57xxx family

Software Friendly Architecture

- High RAM to Flash ratio
- Independent CPU and peripheral clocking
- 48MHz 1% IRC – no PLL init required in LP
- Registers maintained in all modes
- Programmable triggers for ADC → no SW delay counters or extra interrupts

Functional safety

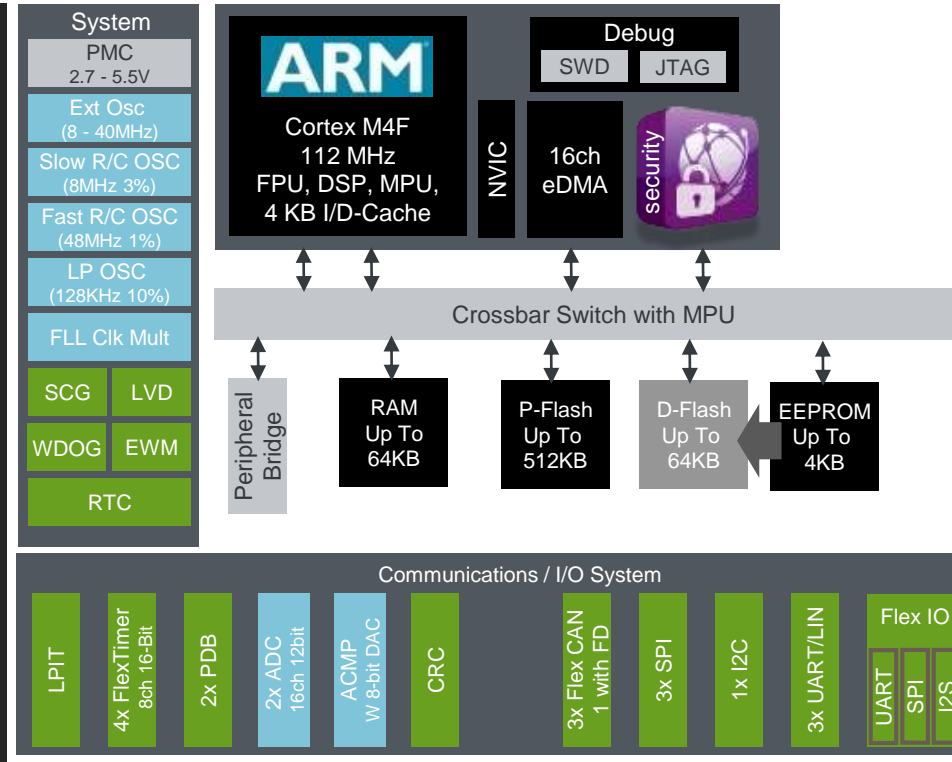
- ISO26262 support for ASIL B or higher
- Memory Protection Unit
- ECC on Flash/Dataflash and RAM
- Independent internal OSC for Watchdog
- Diversity between ADC and ACMP
- Diversity between SPI/SCI and FlexIO
- Core self test libraries
- Scalable LVD protection
- CRC

Low power

- Low leakage technology
- Multiple VLP modes and IRC combos
- Wake-up on analog thresholds

Security

- CSEc (SHE-spec)



MCU Core
and Memories

5V Analogue
Components

Digital
Components

Packages & IO

- Open-drain for 3.3 V and hi-drive pins
- Powered ESD protection
- Packages: 100 BGA, 64 LQFP, 100 LQFP

Operating Characteristics

- Voltage range: 2.7V to 5.5V
- Temperature (ambient): -40°C to +125°C

Targeting General Purpose Applications



Body control module



Human machine interface



Wireless charging



Battery Management



Tire pressure receiver

Product Longevity



Climate control



Door/Window/sunroof



Near Field Communication



Lighting



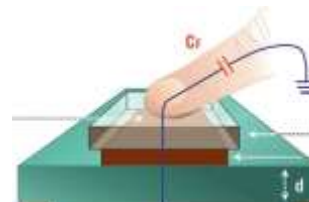
Secure transmission / encryption in cars



Motorbike ECU/ABS



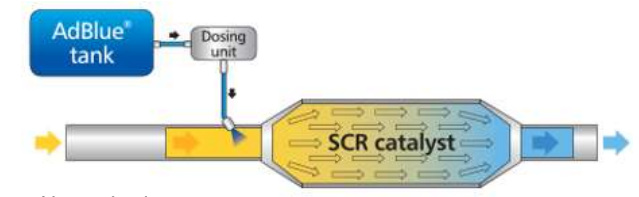
PMSM/BLDC motorcontrol



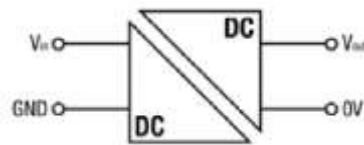
Touch sensing



Park assist



Nox reduction systems



DC/DC converters



E-shifter

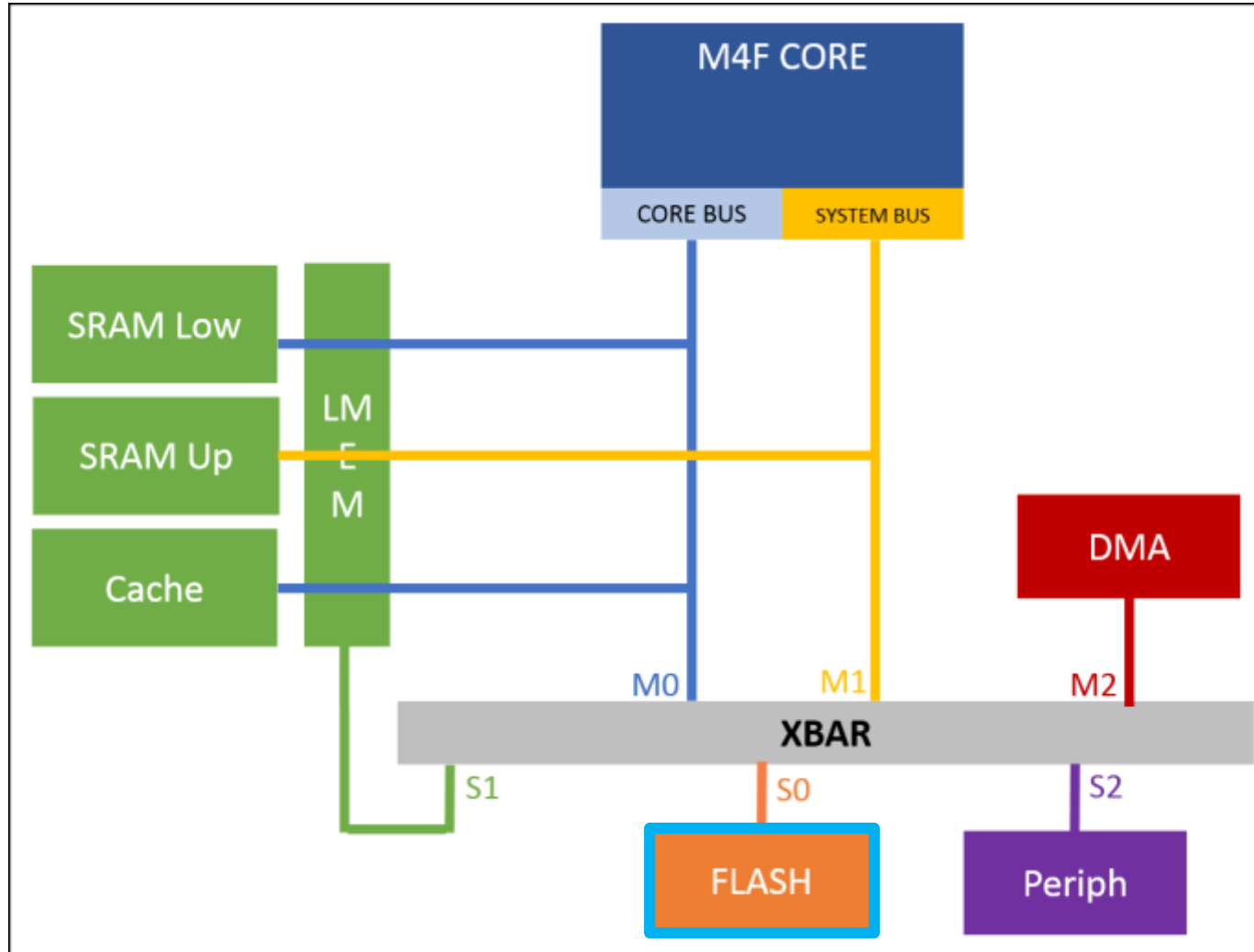


Rear view camera tilt

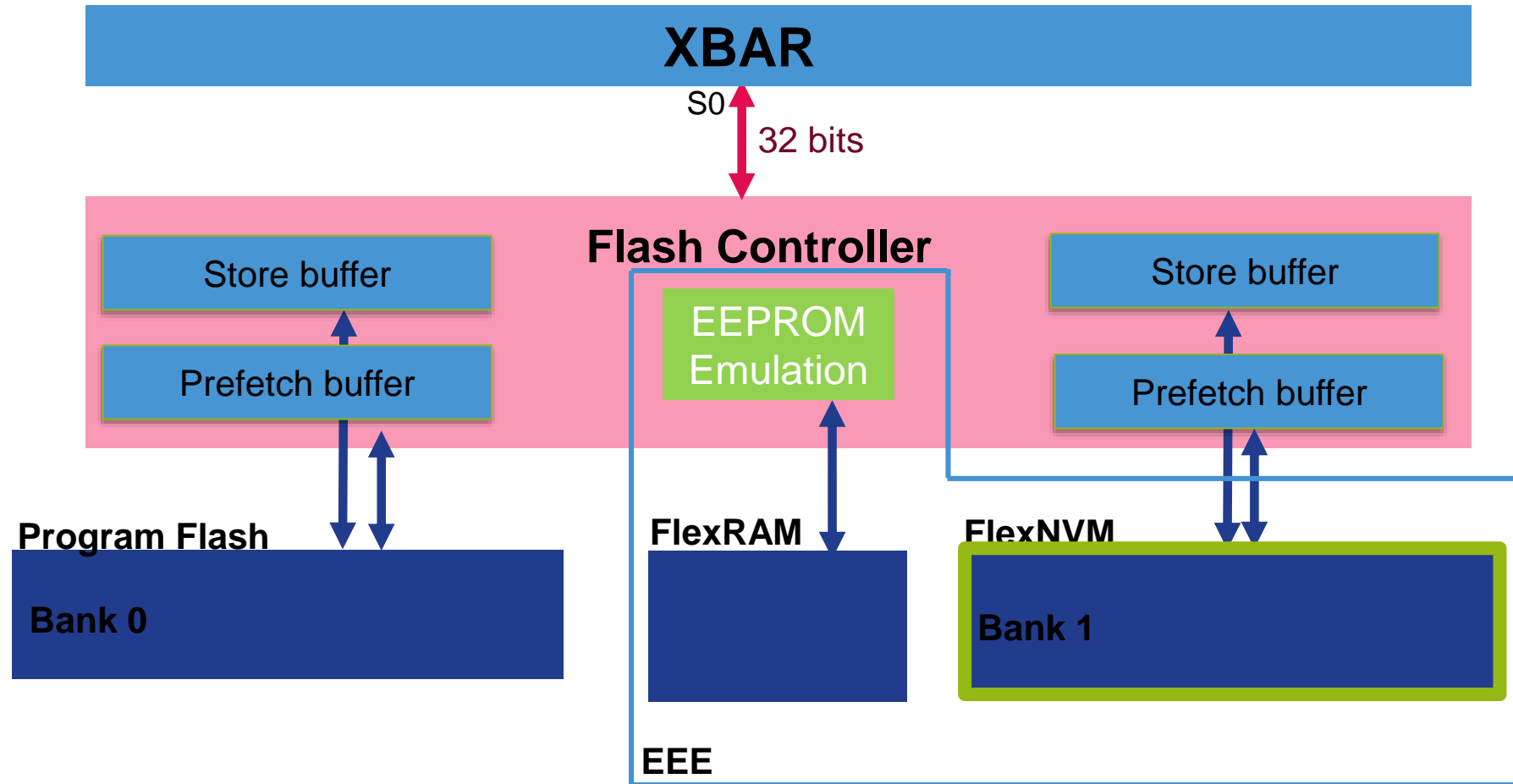


Steering wheel electronics

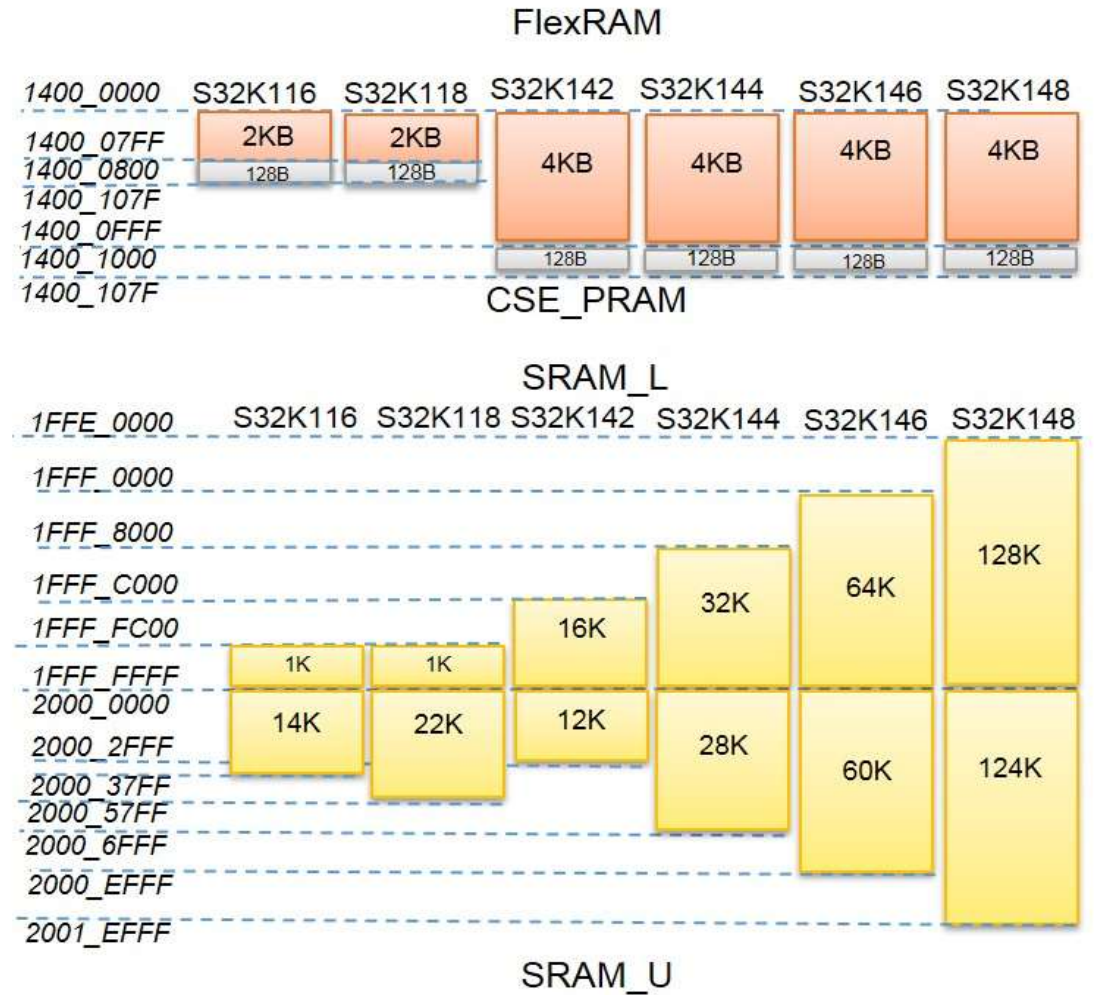
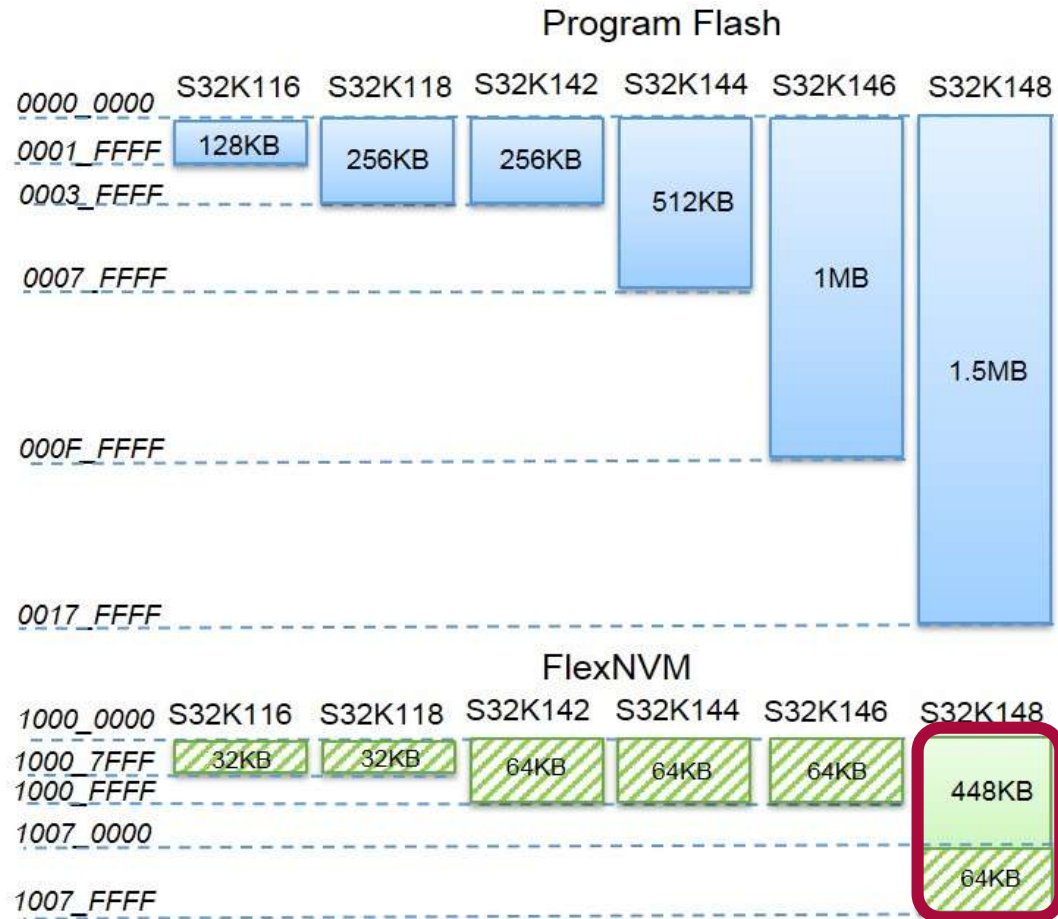
Introduction to S32K Memory



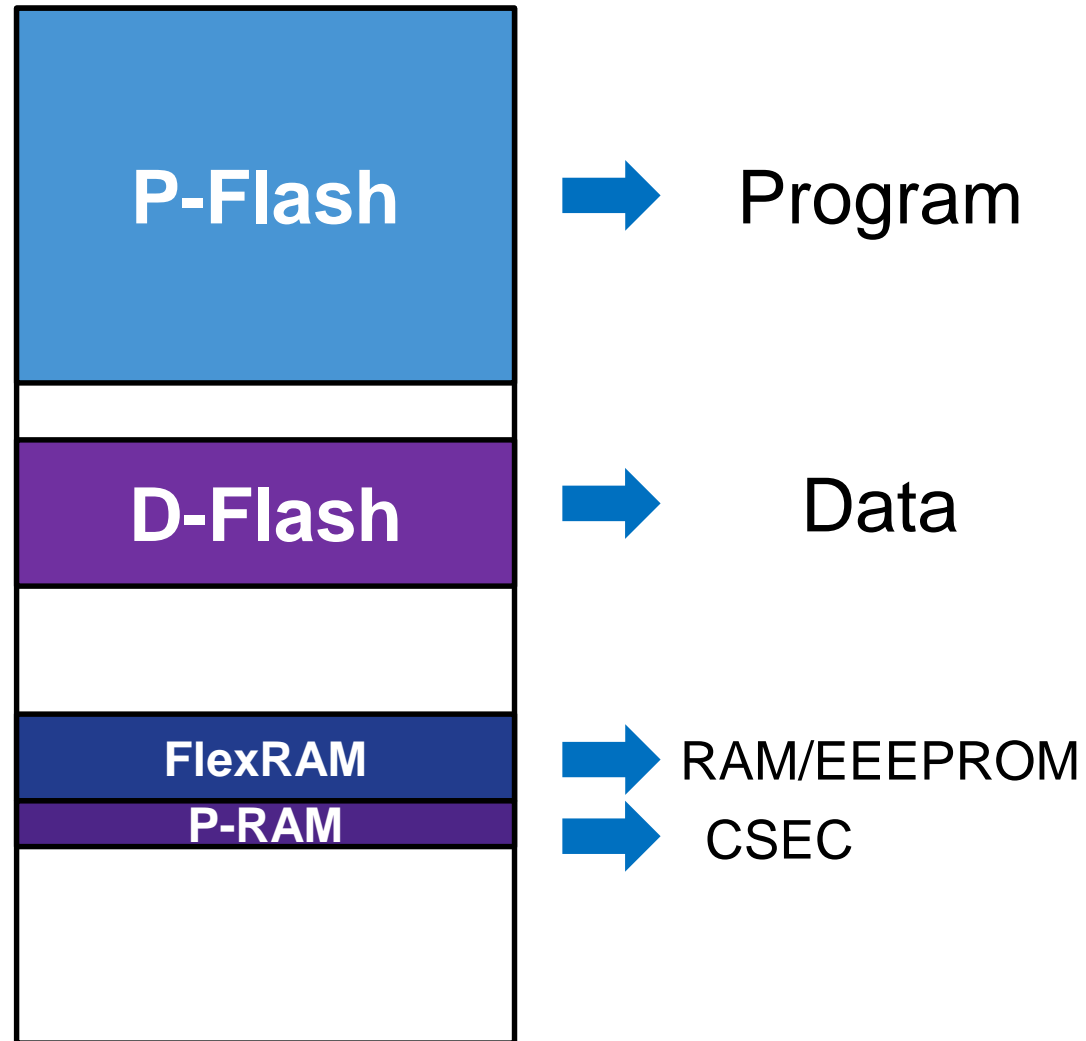
Flash Block



Memory Mapping



S32K144 Memory Architecture



Special Considerations

Only one at the same time!

		Pflash			FlexNVM			FlexRAM			CSEc
		Read	Program Phrase	Erase Flash Sector	Read	Program Phrase	Erase Flash Sector	Read	EEPROM Write	RAM write	Any command
Program Flash	Read					Allowed	Allowed		Allowed		Allowed
	Program Phase				Allowed			Allowed		Allowed	Allowed
	Erase Flash Sector				Allowed			Allowed		Allowed	Allowed
FlexNVM	Read		Allowed	Allowed							
	Program Phase	Allowed						Allowed		Allowed	
	Erase Flash Sector	Allowed						Allowed		Allowed	
FlexRAM	Read		Allowed	Allowed	Allowed	Allowed					
	EEPROM Write	Allowed									
	RAM Write		Allowed	Allowed	Allowed	Allowed					
CSEc	Any command	Allowed	Allowed	Allowed							

LAB #1: Flex-NVM Partition



Partition Flex – NVM

- Task

- Configure S32K144 Clock
- Validate NVM factory settings
- Configure NVM memory for: D-Flash, EEPROM and CSEc

- Learn

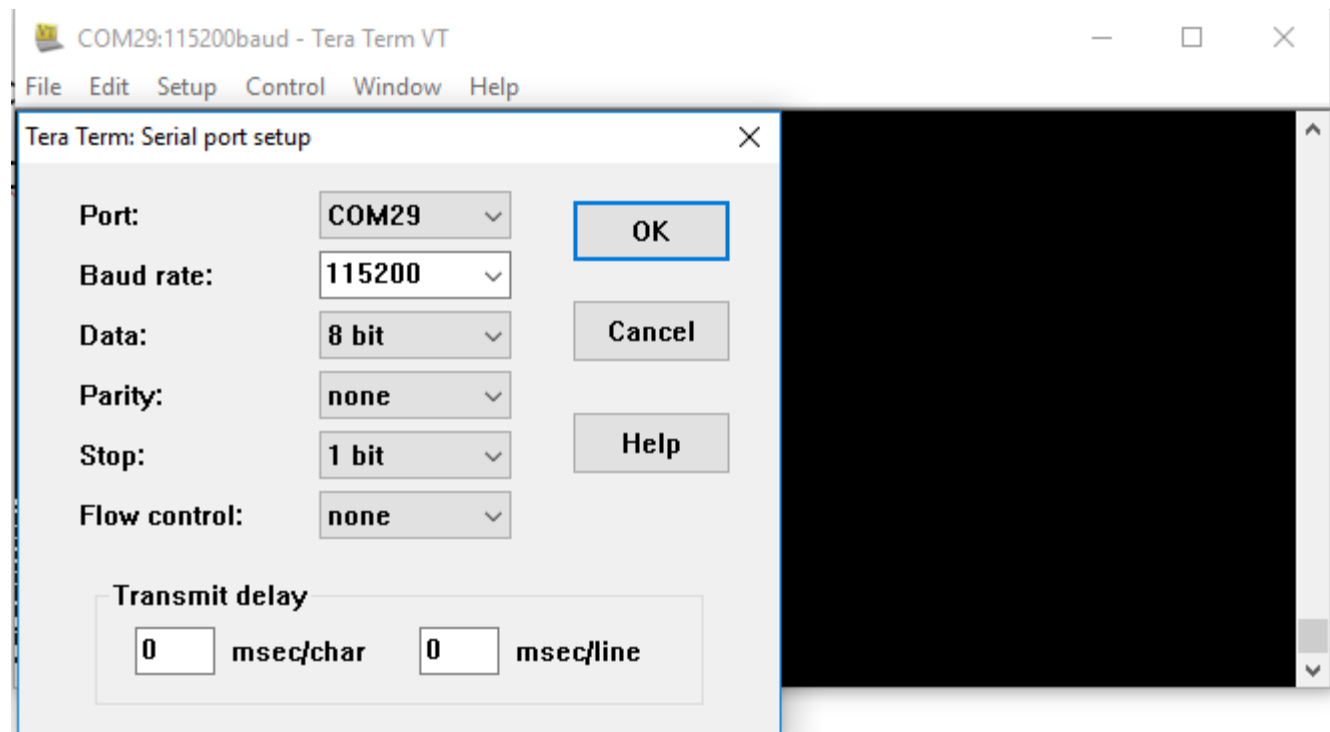
- Identify if the device can be enabled for CSEc operation?
- How to verify EEPROM status?
- How to use command interface to configure EEPROM and enable CSEc?
- Flash status register

- Note

- You will need to run S32K144_NVM_Lab1 project for this Lab.

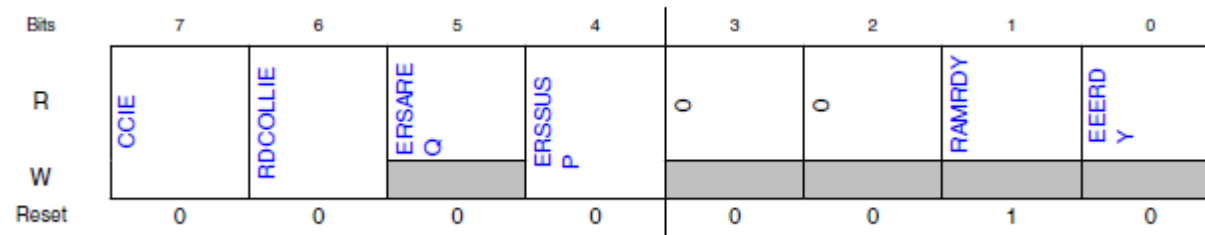
Debugging by Serial Session

- Open a Tera Term sesión:
 1. File -> New connection -> Serial connection
 2. Setup -> Serial Port ->



What is the Default Configuration Status?

- By default CSEc and emulated-EEPROM functions are disabled
 - FTFC_FCNFG – Flash Memory Module Flash Configuration Register



EEERDY = 0 : FlexRAM is not available for emulated EEPROM operation

RAMRDY = 1 : FlexRAM is available for traditional RAM operations only

Register	Hex	Bin
FTFC		
FSTAT	0x80	10000000
MGSTAT0 (bit 0)	0x0	0
FPVIOL (bit 4)	0x0	0
ACCERR (bit 5)	0x0	0
RDCOLERR (bit 6)	0x0	0
CCIF (bit 7)	0x1	1
FCNFG	0x02	00000010
EEERDY (bit 0)	0x0	0
RAMRDY (bit 1)	0x1	1
ERSSUSP (bit 4)	0x0	0
ERSAREQ (bit 5)	0x0	0
RDCOLLIE (bit 6)	0x0	0
CCIE (bit 7)	0x0	0

Does My Part Have a CSEc?

- How to locate if your part has CSEc in it or not?
 - SIM_SDID – System Integration Module System Device Identification register



SIM	
SIM	
CHIPCTL	0x00300000
FTMOPT0	0x00000000
LPOCLKS	0x00000003
ADCOPT	0x00000000
FTMOPT1	0x00000000
MISCTRL0	0x00000000
SDID	0x144F04E0
FEATURES (bits 7-0)	0xE0
PACKAGE (bits 11-8)	0x4
REVID (bits 15-12)	0x0
RAMSIZE (bits 19-16)	0xF
DERIVATE (bits 23-20)	0x4
SUBSERIES (bits 27-24)	0x4
GENERATION (bits 31-28)	0x1

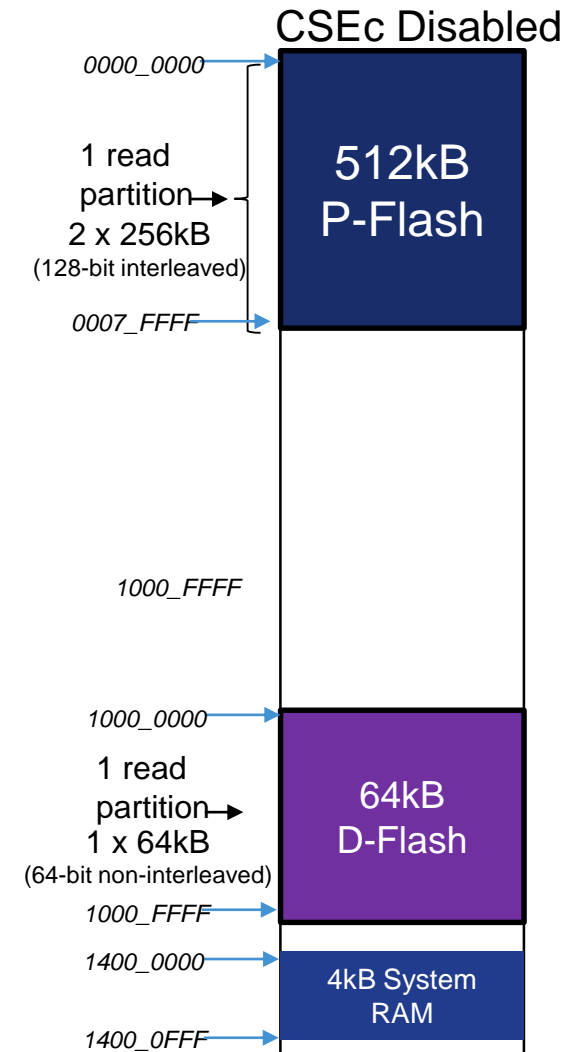
Features on this part are:

Security = CSEc enabled
CAN-FD Enabled
FlexIO Enabled

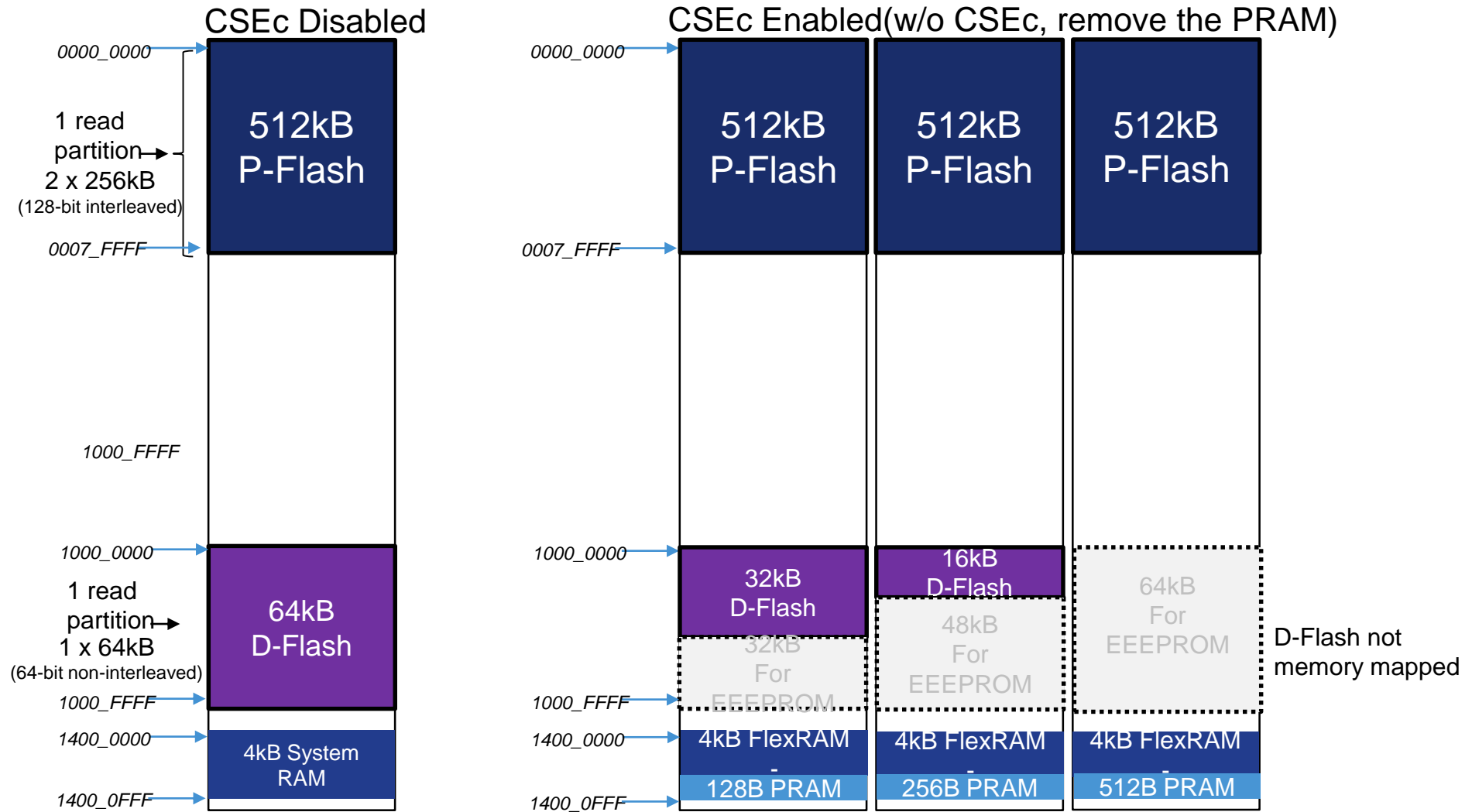
7-0	Features
FEATURES	Specifies the supported features of the chip. 1 Feature is present 0 Feature is not present Each bit in this field represents the following:
	<ul style="list-style-type: none">• Bit 7: Security• Bit 6: ISO CAN-FD• Bit 5: FlexIO• Bit 4: QuadSPI• Bit 3: ENET• Bit 2: Reserved• Bit 1: SAI• Bit 0: Reserved

To Enable the CSEc the Memory Need to Be Partitioned for EEPROM

- To access the CSEc feature set, the part must be configured for EEE operation, using the PGMPART command.
- By enabling security features and configuring a number of user keys, the total size of the 4 Kbyte FlexRAM used for EEPROM will be reduced by the space required to store the user keys.
- The user key space will then effectively be unaddressable space in the FlexRAM.



S32K144 P/D-Flash Memory Mapping



Enable CSEc and Configure Flash for EEPROM & D-Flash Operations

- FTFC Program Partition Command(PRGPART) configures part for CSEc and EEPROM operations
 - Issue PRGPART command using FCCOB[0-F] registers
 - FCCOB[0-F] are located inside FTFC module and are part of Host Interface

FCCOB Number	Content Description	Data	Comments
0	Program Partition Command	0x80	Command
1	CSEc Key Size	0x03	0x00 – CSEc Disabled; 0x01 – 5 Keys; 0x02 – 10 Keys; 0x03 – 20 Keys
2	SFE	0x00	Verify only functionality disabled
3	FlexRAM load during reset option (only bit 0 used):	0x00	0 - FlexRAM loaded with valid EEPROM data during reset sequence 1 - FlexRAM not loaded during reset sequence
4	EEPROM Data Set Size Code	0x02	4k of FlexRAM reserved for EEPROM operation
5	FlexNVM Partition Code	0x03	32k of FlexNVM is reserved for EEPROM-backup operation and 32K is used for D-Flash

NOTE:
EESIZE must be 0x2 for 4kB

DEPART must not be 0x0; For Lab set to 0x3

Enable EEPROM, CSEc & D-Flash With Flash Command Write Sequence

```
eeeprom_status EEPROM_init(eeprom_size eflash_size)
{
    eeprom_status ret = EEPROM_OK;
    uint8_t params = 0;
    uint8_t flashCommandBuffer[FLASH_COMMAND_BUFFER_SIZE] = {0};

    /* Check if EEE is already implemented */
    if (eFlexNVMnotPartitioned == EEPROM_initialized())
    {
        /* Launch partition code */
        params = 0;
        memset((void *)&flashCommandBuffer[0], 0x00, (size_t)FLASH_COMMAND_BUFFER_SIZE);

        flashCommandBuffer[params++] = 0x80; /* FCCOB0: Selects the PGMPART command */
    #if CSEC_KEYS_ENABLED
        flashCommandBuffer[params++] = 0x03; /* FCCOB1: CSEc Key Size 3: 512 bytes of EEE will be used for Keys*/
    #else
        flashCommandBuffer[params++] = 0x00; /* FCCOB1: CSEc Key Size 0: No space for Keys*/
    #endif
        flashCommandBuffer[params++] = 0x00; /* FCCOB2: SFE */
    #if FLEXRAM_IS_LOADED_IN_RST
        flashCommandBuffer[params++] = 0x00; /* FCCOB3: FlexRAM loading reset option: 0 - FlexRAM loaded with valid EEPROM*/
    #else
        flashCommandBuffer[params++] = 0x01; /* FCCOB3: FlexRAM loading reset option: 1 - FlexRAM not loaded; option 0: FlexRam Loaded */
    #endif
    #if (defined(S32K11x_SERIES))
        flashCommandBuffer[params++] = 0x03; /* FCCOB4: EEPROM data set size code: EEESIZE = 3 (2kB) */
    #else
        flashCommandBuffer[params++] = 0x02; /* FCCOB4: EEPROM data set size code: EEESIZE = 2 (4kB) */
    #endif
        flashCommandBuffer[params++] = eflash_size; /* FCCOB5: FlexNVM Partition code: DEPART */
    }
```

Enable CSEc With Flash Command Write Sequence

See FCCOB Program Partition command requirements (below) from S32K1xx reference manual. Also, below is part of the C-Code used for the this Lab.

Offset	Register	Width (In bits)	Access	Reset value
4h	Flash Common Command Object Registers (FCCOB0)	8	RW	00h
5h	Flash Common Command Object Registers (FCCOB1)	8	RW	00h
6h	Flash Common Command Object Registers (FCCOB2)	8	RW	00h
7h	Flash Common Command Object Registers (FCCOB3)	8	RW	00h
8h	Flash Common Command Object Registers (FCCOB4)	8	RW	00h
9h	Flash Common Command Object Registers (FCCOB5)	8	RW	00h
Ah	Flash Common Command Object Registers (FCCOB6)	8	RW	00h
Bh	Flash Common Command Object Registers (FCCOB7)	8	RW	00h
Ch	Flash Common Command Object Registers (FCCOB8)	8	RW	00h
Dh	Flash Common Command Object Registers (FCCOB9)	8	RW	00h
Eh	Flash Common Command Object Registers (FCCOB10)	8	RW	00h
Fh	Flash Common Command Object Registers (FCCOB11)	8	RW	00h

FCCOB Number	FCCOB Contents [7:0]
0	0x80 (PGMPART)
1	CSEc Key Size
2	SFE
3	FlexRAM load during reset option (only bit 0 used): 0 - FlexRAM loaded with valid EEPROM data during reset sequence 1 - FlexRAM not loaded during reset sequence
4	EEPROM Data Set Size Code ¹
5	FlexNVM Partition Code ²

```
/* Enables CSEc by issuing the Program Partition Command, procedure: Figure 33-8 in RM, Configures for all 24 Keys */
uint8_t configure_part_CSEc(void)
{
    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != FTFC_FSTAT_CCIF_MASK); /* Wait until any ongoing flash operation is completed */
    FTFC->FSTAT = (FTFC_FSTAT_FPVIOL_MASK | FTFC_FSTAT_ACCERR_MASK); /* Write 1 to clear error flags */
    /*
    FTFC->FCCOB[2] = 0x03; /* FCCOB1 = 2b11, 1 - 20 keys
    FTFC->FCCOB[3] = 0x80; /* FCCOB0 = 0x80, program partition command */
    FTFC->FCCOB[1] = 0x00; /* FCCOB2 = 0x00, SFE = 0, VERIFY_ONLY attribute functionality disable */
    FTFC->FCCOB[0] = 0x00; /* FCCOB3 = 0x00, FlexRAM will be loaded with valid EEPROM data during reset sequence */
    FTFC->FCCOB[7] = 0x02; /* FCCOB4 = 0x02, 4k EEPROM Data Set Size */
    FTFC->FCCOB[6] = 0x04; /* FCCOB5 = 0x04, no data flash, 64k(all) EEPROM backup */

    FTFC->FSTAT = FTFC_FSTAT_CCIF_MASK; /* Start command execution by writing 1 to clear CCIF bit */

    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != FTFC_FSTAT_CCIF_MASK); /* Wait until ongoing flash operation is completed */

    flash_error_status = FTFC->FSTAT; /* Read the flash status register for any Execution Error */

    return flash_error_status;
}
```

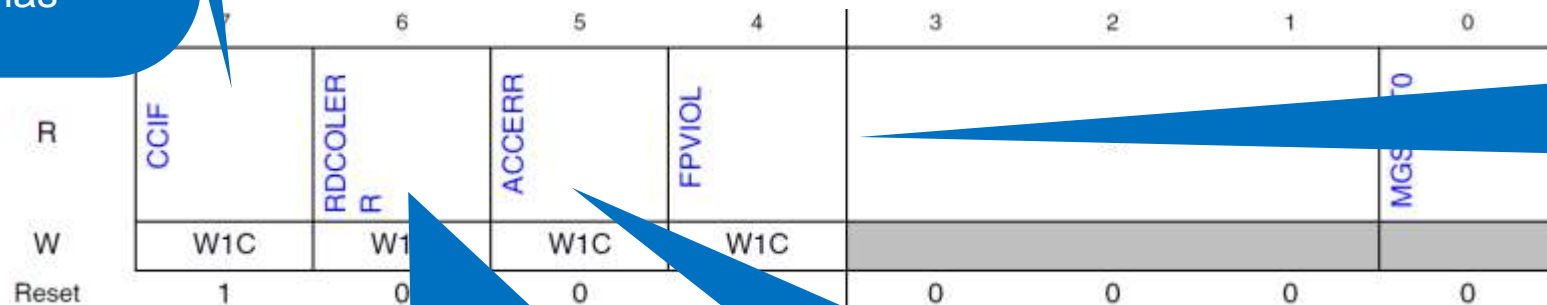

FTFC_FSTAT – Flash Status Register

Command Complete
Interrupt Flag

CCIF=0 : FTFC
command or emulated
EEPROM operation or
CSEc operation in
progress
CCIF=1 : FTFC
command or emulated
EEPROM operation or
CSEc operation has
completed

Check for following before & after issuing command

- Flash operation may be going on
 - Chances of conflict
 - Always check CCIF flag
- Check for potential flash errors
 - If they exist take necessary actions and clear them



Read Collision
Error Flag

- Read while manipulating

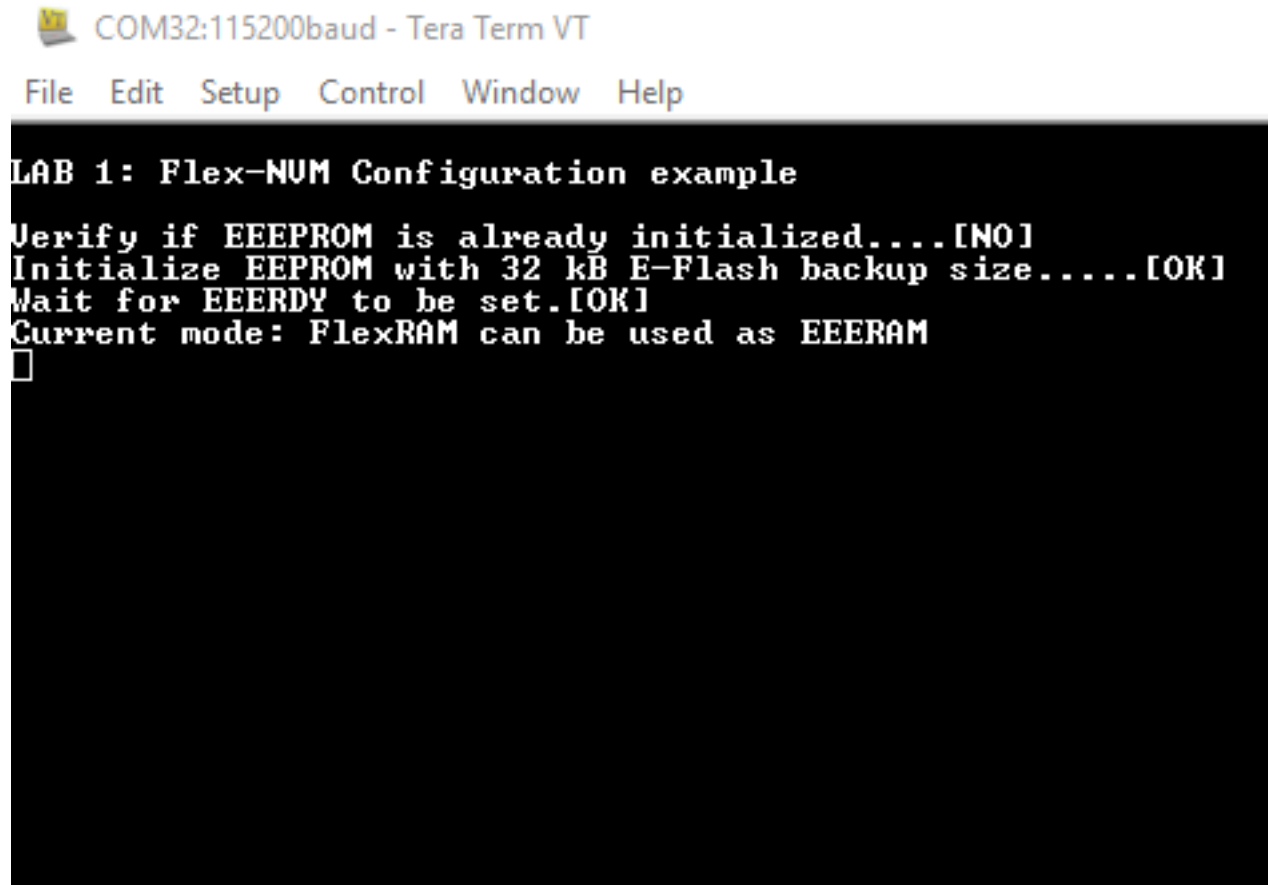
Flash Access Error
Flag

- Illegal access or
- Illegal command

Flash Protection
Violation Flag

- Attempt to program or
erase an address in a
protected area

Output Message on Serial Terminal



A screenshot of a serial terminal window titled "COM32:115200baud - Tera Term VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The terminal output is as follows:

```
LAB 1: Flex-NUM Configuration example
Verify if EEPROM is already initialized....[NO]
Initialize EEPROM with 32 kB E-Flash backup size.....[OK]
Wait for EEERDY to be set.[OK]
Current mode: FlexRAM can be used as EEERAM
█
```

Verify Whether Device is Configured Correctly?

- FTFC_FCNFG – Flash Memory Module Flash Configuration Register

Register	Hex	Bin
FTFC		
FSTAT		
FCNFG	0x01	00000001
EEERDY (bit 0)	0x1	1
RAMRDY (bit 1)	0x0	0
ERSSUSP (bit 4)	0x0	0
ERSAREQ (bit 5)	0x0	0
RDCOLLIE (bit 6)	0x0	0
CCIE (bit 7)	0x0	0

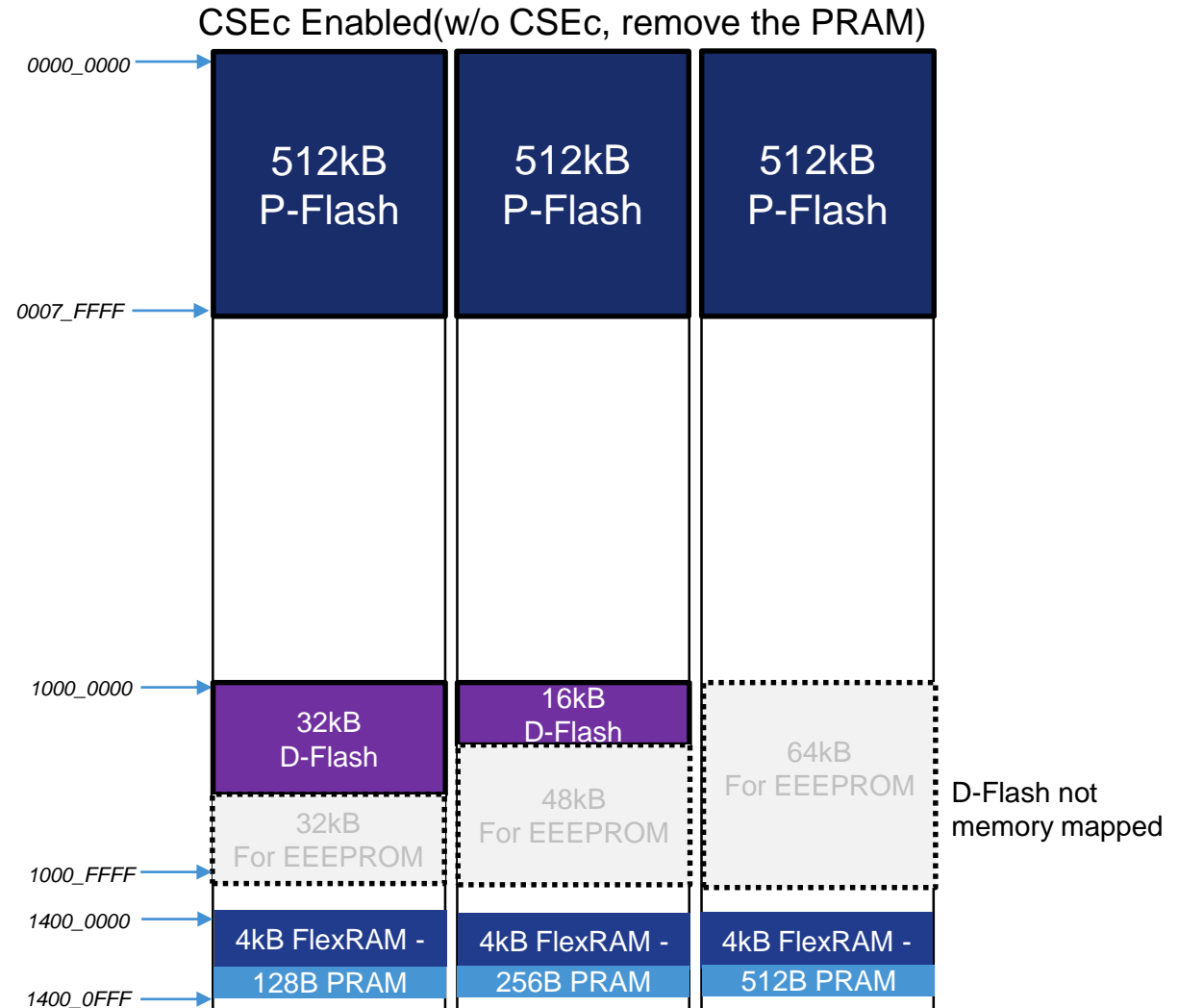
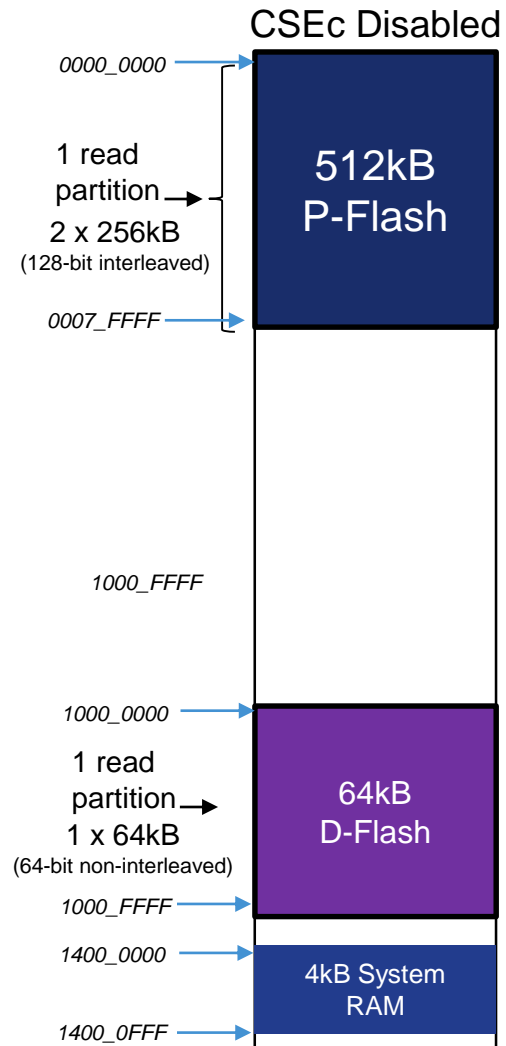
RAMRDY = 0 : FlexRAM is **NOT** available for traditional RAM operations only.

EEERDY = 1 : FlexRAM is available for emulated EEPROM operation

- Try to access the memory area reserved for keys

Address	0 - 3	4 - 7	8 - B	C - F
14000DE0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
14000DF0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
14000E00	00000000	00000000	00000000	00000000
14000E10	00000000	00000000	00000000	00000000
14000E20	00000000	00000000	00000000	00000000

Recap



Lab 2: D-Flash Writing



Lab 2. D-Flash Writing

- Task

- Write information into D-Flash memory through the FCCOB
- Validate the writing/erase function in the first sector of the D-Flash

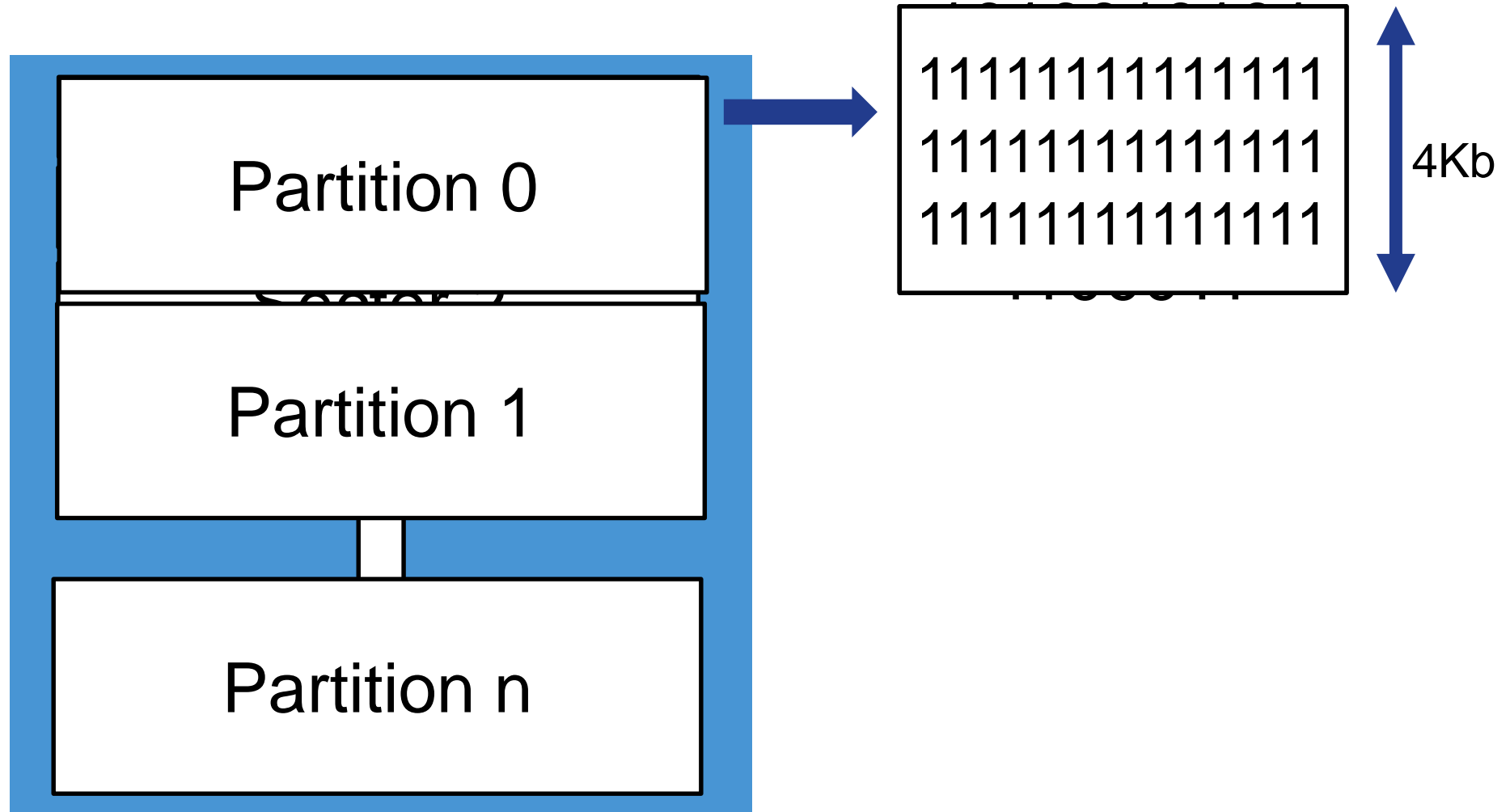
- Learn

- Use of FCCOB commands for writing and erasing D-Flash.
- Understand the differences between blocks and sectors

- Note

- You will need to run **S32K144_NVM_Lab2** project for this Lab.

Memory Block

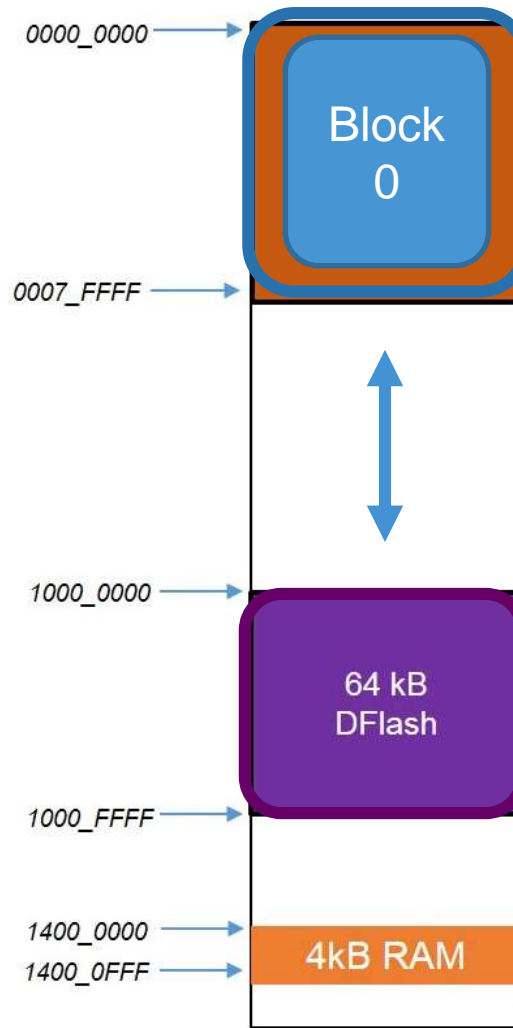


Sector and Bank

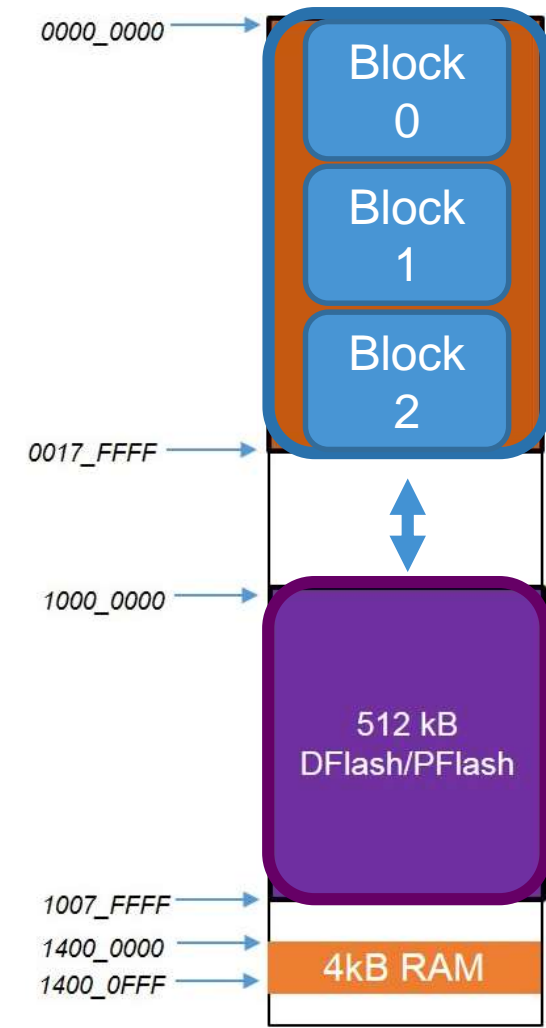
- Sector: smaller erasable area that can be erased from flash
- Bank: partitions in which the flash block is divided into.

P-Flash / D-Flash

- FlexNVM is configured by default as D-Flash.
- P-Flash is split into 512kB blocks known as read partitions.
- Read-While-Write feature applies between read partitions.



S32K144



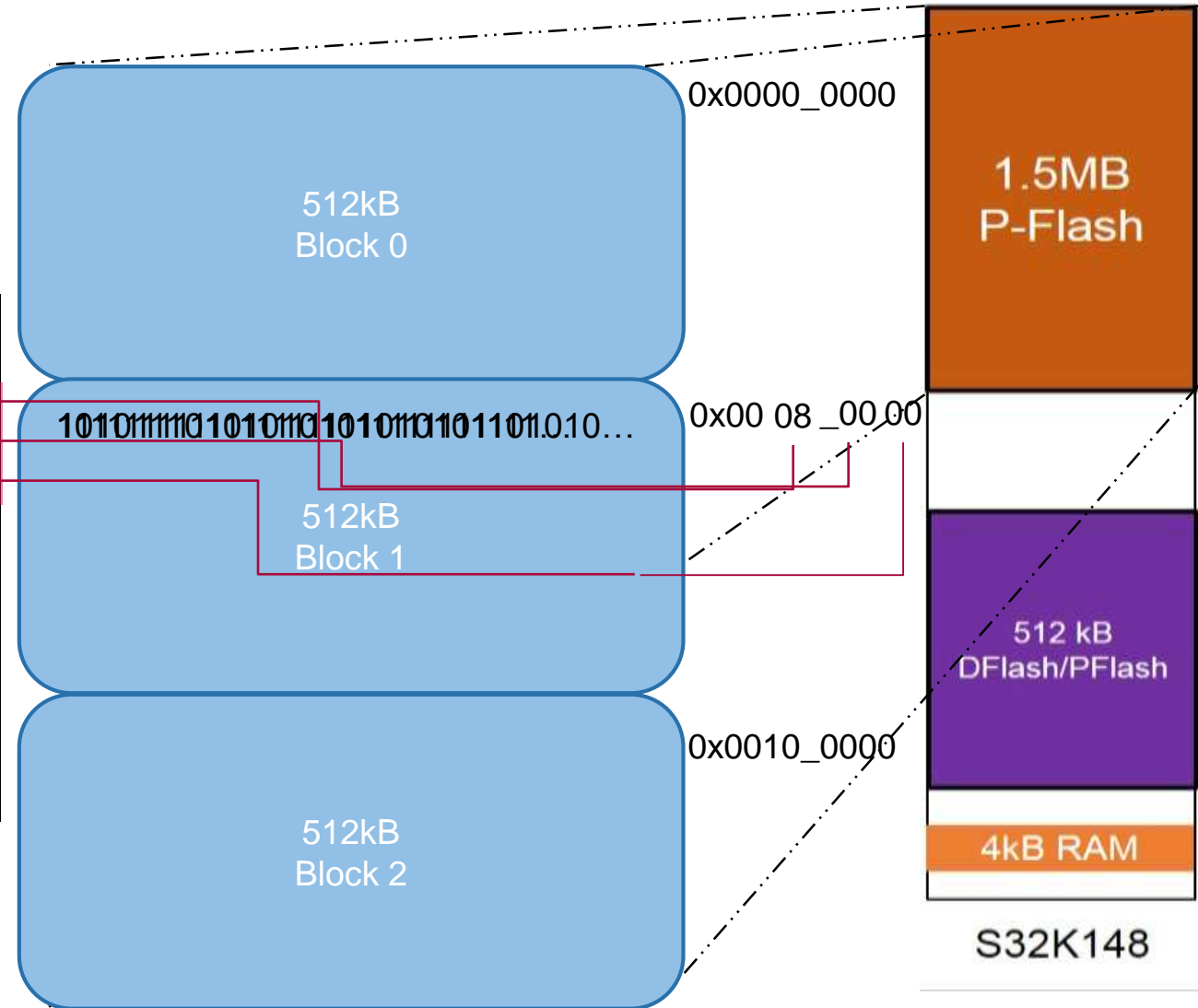
S32K148

P-Flash / D-Flash

Erase Flash Block Command

FCCOB Number	Typical Command Parameter Contents [7:0]
0	FCMD (a code that defines the FTFC command) 0x08
1	Flash address [23:16] 0x08
2	Flash address [15:8] 0x00
3	Flash address [7:0] 0x00
4	Data Byte 0
5	Data Byte 1
6	Data Byte 2
7	Data Byte 3
8	Data Byte 4
9	Data Byte 5
A	Data Byte 6
B	Data Byte 7

FTFC->FSTAT |= FTFC_FSTAT_CCIF_MASK;



On S32 Design Studio

1. Verify if there are space in the FlexNVM for D-Flash

```
/* Check DEPART value in SIM module to check how the DFlash was partitioned */  
uint8_t depart = (SIM->FCFG1 & SIM_FCFG1_DEPART_MASK) >> SIM_FCFG1_DEPART_SHIFT;
```

In case there is no configuration for D-Flash the program will stay in an infinite loop

```
case (0b0100):  
case (0b1000):  
    /* 0 kB for DFlash, EEPROM is 64 kB */  
    DFlash_size = 0;  
    printf("\r\n D-Flash is 0Kb and EEPROM is 64Kb \r\n\r\n");  
    printf("\r\n To continue is necessary load Lab 1 first.... \r\n\r\n");  
    for(;;){} //Infinite Loop  
    break;
```

On S32 Design Studio..

- Erasing the first and second sectors of D-flash

```
/* Erase first 2kB sector */  
Flash_eraseDPFlashSector(DFLASH_ADDRESS + DFLASH_SECTOR_SIZE_1);  
  
/* Erase second 2kB sector */  
Flash_eraseDPFlashSector(DFLASH_ADDRESS + DFLASH_SECTOR_SIZE_2);
```

- Changing the 32-bit address for a 24-bit address (Adding 1 or 0 in the most significant bit for P-Flash or D-Flash)

```
__attribute__((section(".code_ram"))) void Flash_eraseDPFlashSector(uint32_t address)  
{  
    uint32_t temp = 0;  
    /* check if address is on DFlash range */  
    if ((address >= DFLASH_ADDRESS) && (address < (DFLASH_ADDRESS + 0x10000)))  
    {  
        /* DFlash command starts 0x80000000 */  
        temp = address + 0x8000000U - DFLASH_ADDRESS;  
    }  
}
```

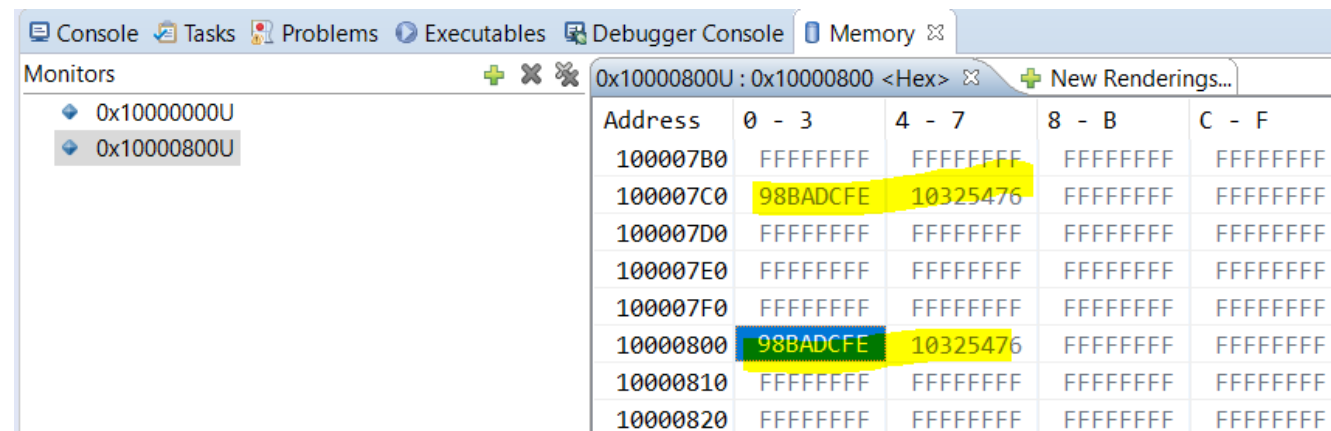

On S32 Design Studio

Writing 3 times:

- 2 in the first sector of D-Flash and 1 in the beginning of the second sector

```
/* Write to the FIRST sector: Location 0 */  
Flash_writeDPFlash(DFLASH_ADDRESS + DFLASH_SECTOR_SIZE_1, dataToWrite);  
/* Write to the first sector: Location 0x7C0 */  
Flash_writeDPFlash(DFLASH_ADDRESS + 0x7C0, dataToWrite);  
  
/* Write to the SECOND sector: Location 0 */  
Flash_writeDPFlash(DFLASH_ADDRESS + DFLASH_SECTOR_SIZE_2, dataToWrite);
```

- Open Memory view to validate the data into the D-Flash



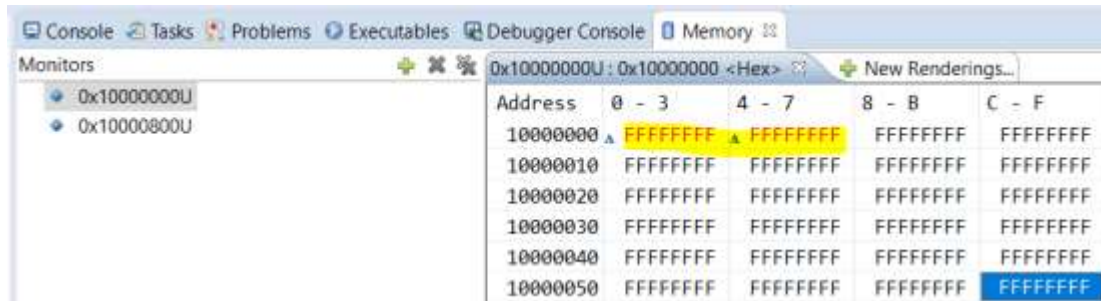
Address	0 - 3	4 - 7	8 - B	C - F
100007B0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007C0	98BADCFE	10325476	FFFFFFFF	FFFFFFFF
100007D0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007E0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007F0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000800	98BADCFE	10325476	FFFFFFFF	FFFFFFFF
10000810	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000820	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF

On S32 Design Studio

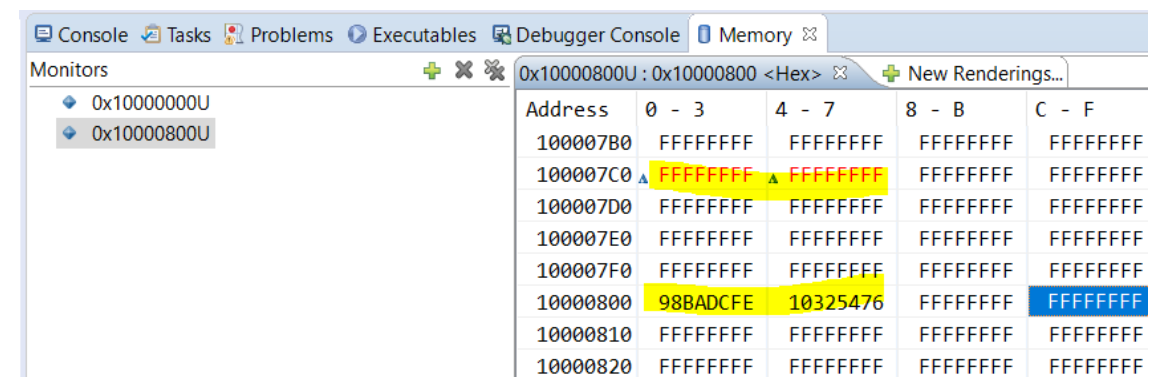
- Use Erase command to erase in a “random” locality of sector 1

```
/* Erase a random location of FIRST sector */  
Flash_eraseDPFlashSector(DFLASH_ADDRESS + 0x7C0);
```

- Open Memory view to validate that the data in 0x7C0 and 0x000 was erased but no the data in sector 2



Address	0 - 3	4 - 7	8 - B	C - F
10000000	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000010	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000020	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000030	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000040	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000050	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF



Address	0 - 3	4 - 7	8 - B	C - F
100007B0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007C0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007D0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007E0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
100007F0	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000800	98BADCFE	10325476	FFFFFFFF	FFFFFFFF
10000810	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
10000820	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF

Lab 3: EEPROM vs. EEPROM Quick Writes



Lab 3. EEPROM vs. EEPROM Quick Writes

- Task

- Write information into EEPROM
- Write information into EEPROM using Quick Writes configuration

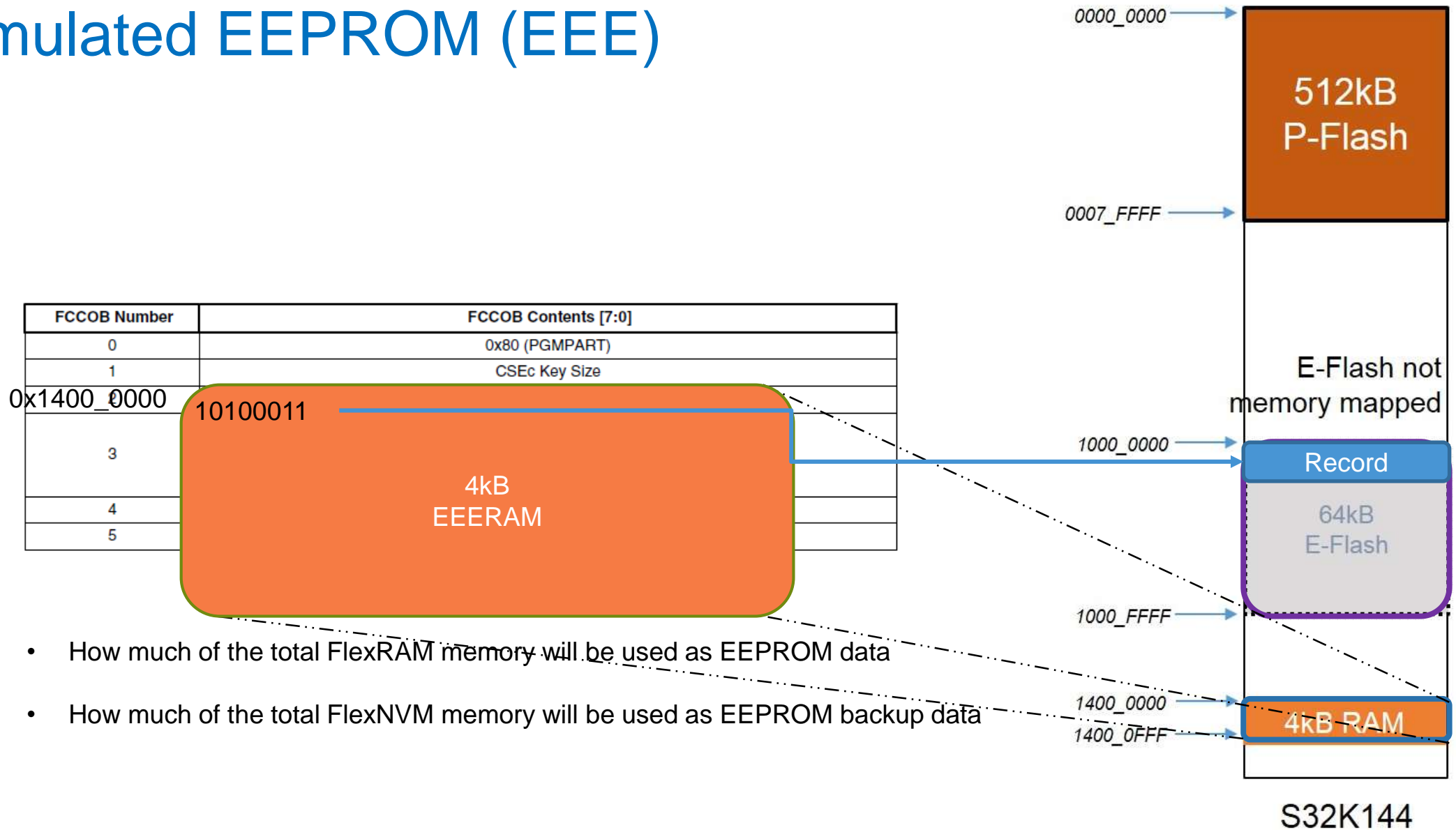
- Learn

- Advantages of writing into EEPROM using the FlexRam
- Configuration of FlexRam for QuickWrites
- Differences between EEPROM and EEPROM Quick Writes

- Note

- You will need to run S32K144_NVM_Lab3 project for this Lab.

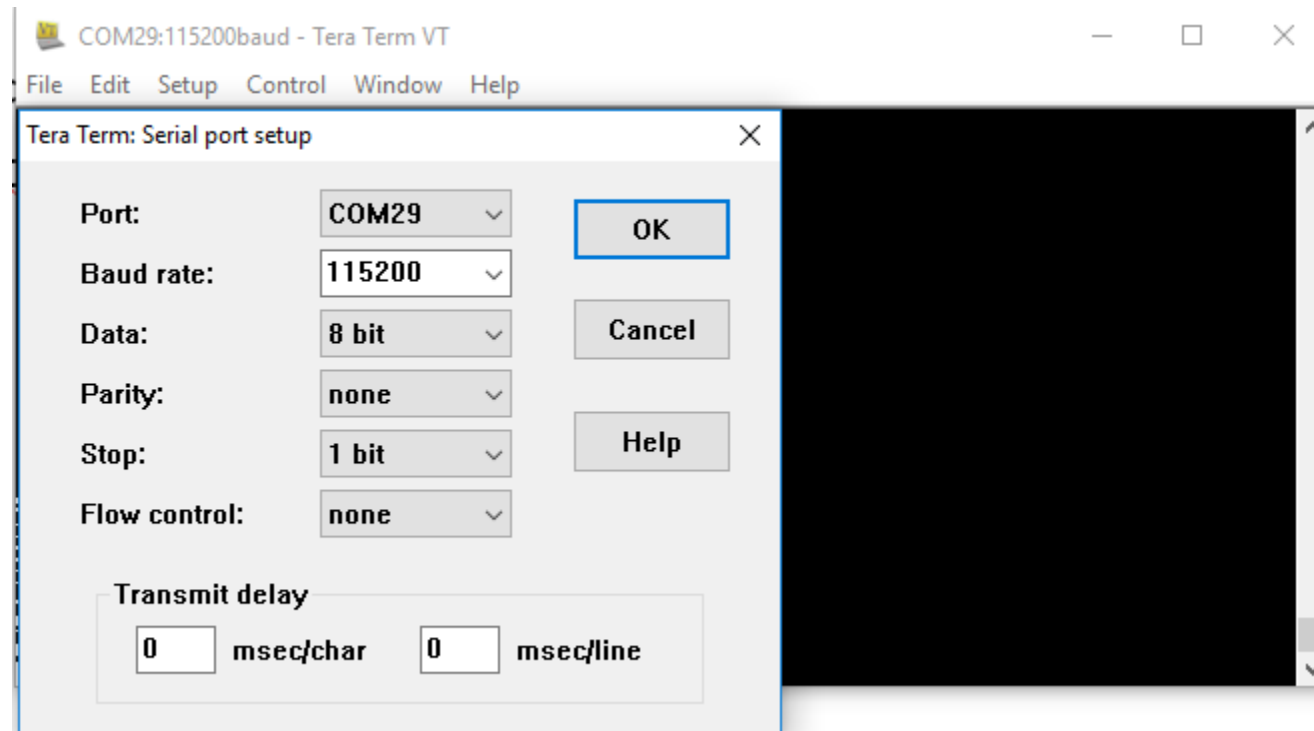
Emulated EEPROM (EEE)



- How much of the total FlexRAM memory will be used as EEPROM data
- How much of the total FlexNVM memory will be used as EEPROM backup data

Debugging by Serial Session

- Open a Tera Term sesión:
 1. File -> New connection -> Serial connection
 2. Setup -> Serial Port ->



Lab 3. EEPROM vs EEPROM Quick Writes

Using the EEPROM_write function a buffer size of 128 bytes (32-bit aligned) Will be writing into FlexRam memory space (0x1400 0000)

```
ret = EEPROM_write((uint32_t)(FLEXRAM_START_ADDR), (uint32_t *)buffer, BUFFER_SIZE);
if (ret != EEPROM_OK)
{
    printf("[FAIL]\n\rLoop forever\r\n");
    /* Error */
    while (1)
    {
        __asm("nop");
    }
}
else
{
    printf("[OK]\r\nPrint FlexRAM content to check EEE data\r\n");
}
```

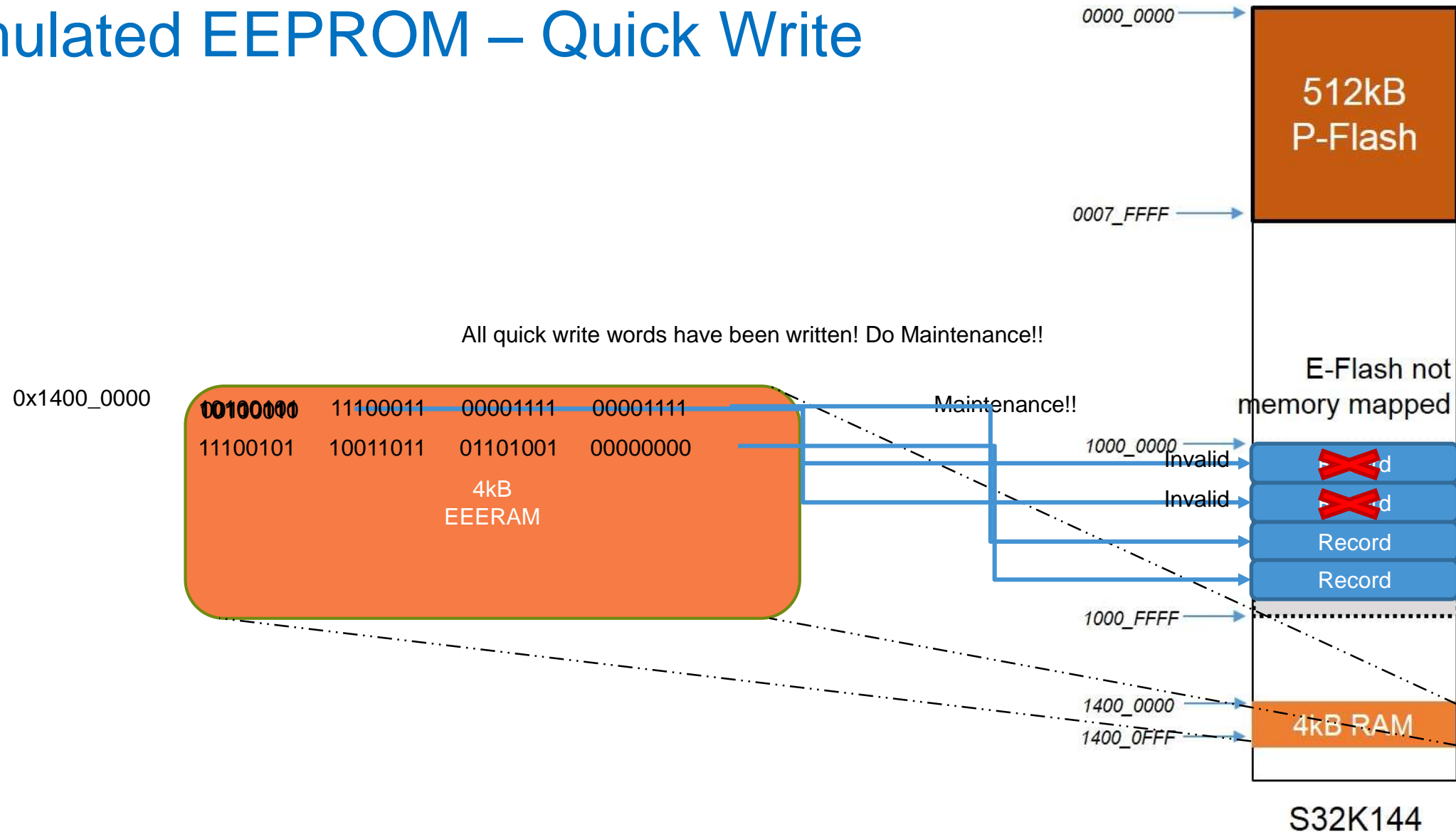
Lab 3. Configuring FlexRam for QuickWrites

- Change to FlexRam for QuickWrites configuration.
 - Config FlexRam for saving 128 bytes

```
printf("Configure FlexRAM for %d quick writes mode\r\n", QUICK_WRITES_BYTES);  
ret = SetRAM_mode(eFlexRAMforQuickWrites, QUICK_WRITES_BYTES, NULL);
```

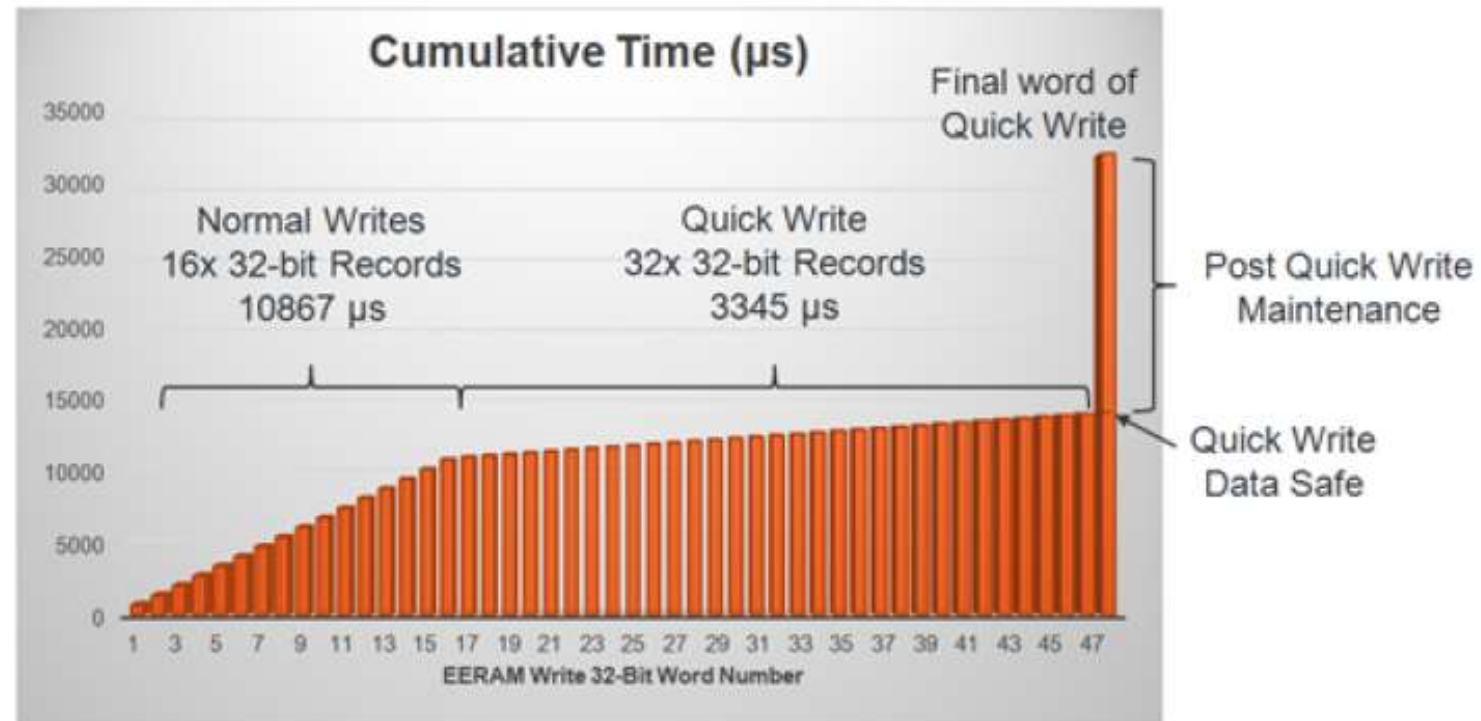
```
switch(mode)  
{  
case eFlexRAMforNormalWrites:  
case eCompleteQuickWriteProcess:  
case eTraditionalRAM:  
    /* No more FCCOBs parameters are needed for these options */  
    break;  
case eQuickWriteQuery:  
    if (quickWriteStatusPtr == NULL)  
    {  
        ret = eEEPROM_INVALID_ARGS;  
    }  
    break;  
case eFlexRAMforQuickWrites:  
    if ((quickWriteBytes != 0) && (quickWriteBytes <= 512) && ((quickWriteBytes & 0x3) == 0))  
    {  
        flashCommandBuffer[params++] = 0x00; /* FCCOB2: Reserved */  
        flashCommandBuffer[params++] = 0x00; /* FCCOB3: Reserved */  
        /* Use 512 bytes (maximum allowed value) for quick writes */  
        flashCommandBuffer[params++] = ((quickWriteBytes >> 8) & 0xFF); /* FCCOB4: Number of FlexRAM bytes allocated for EEPROM quick writes */  
        flashCommandBuffer[params++] = (quickWriteBytes & 0xFF); /* FCCOB5: Number of FlexRAM bytes allocated for EEPROM quick writes */  
    }  
    _
```


Emulated EEPROM – Quick Write



How Quick Are Quick Writes?

The data writing speed is the same as normal writes, the difference is on the data maintenance

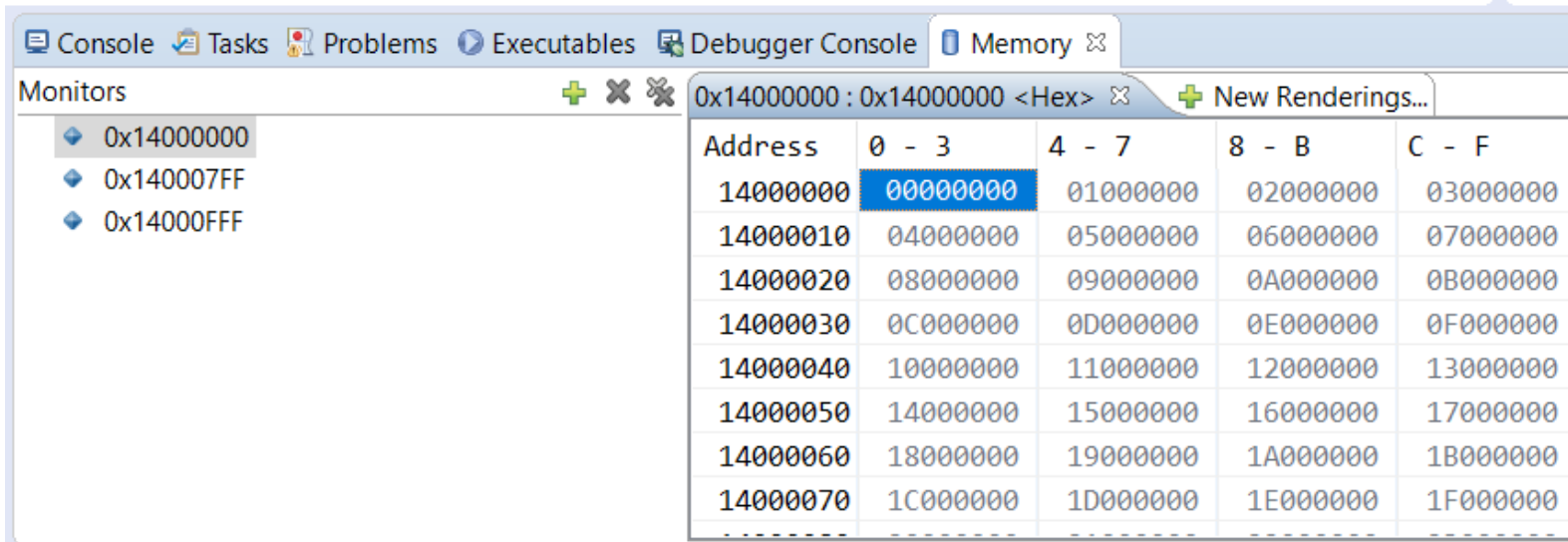


*EEE records were written over existing data (not pre-erased array).

**Unit had 9000 previous program / erase flash cycles.

Lab 3. EEPROM Quick Writes

Validate the storage of the 128 bytes into the FlexRam by checking the Memory window or in terminal



0x14000000 : 0x14000000 <Hex> ✕				
Address	0 - 3	4 - 7	8 - B	C - F
14000000	00000000	01000000	02000000	03000000
14000010	04000000	05000000	06000000	07000000
14000020	08000000	09000000	0A000000	0B000000
14000030	0C000000	0D000000	0E000000	0F000000
14000040	10000000	11000000	12000000	13000000
14000050	14000000	15000000	16000000	17000000
14000060	18000000	19000000	1A000000	1B000000
14000070	1C000000	1D000000	1E000000	1F000000

```
Printing FlexRAM area
[0x14000000]: 0x00000000
[0x14000004]: 0x00000001
[0x14000008]: 0x00000002
[0x1400000C]: 0x00000003
[0x14000010]: 0x00000004
[0x14000014]: 0x00000005
[0x14000018]: 0x00000006
[0x1400001C]: 0x00000007
[0x14000020]: 0x00000008
[0x14000024]: 0x00000009
[0x14000028]: 0x0000000A
[0x1400002C]: 0x0000000B
[0x14000030]: 0x0000000C
[0x14000034]: 0x0000000D
[0x14000038]: 0x0000000E
[0x1400003C]: 0x0000000F
[0x14000040]: 0x00000010
[0x14000044]: 0x00000011
```

Recommendations

- Any software driver that uses CSEc, EEPROM (writes only) or Flash controller commands must not be placed in FlexNVM's PFlash
- Any Configuration Data (constant parameters) that must be read during a CSEc or EEPROM write or program/erase operation must not be placed in FlexNVM's Dflash
- Any ISR associated to an interrupt that has to be served during CSEc or EEPROM write or program/erase operation must not be placed in FlexNVM's Pflash. The same restriction applies to the functions called from ISRs.

In Conclusion

- The FlexNVM can be used as:
 - D-Flash
 - P-Flash
 - EEPROM
- The FlexNVM is an important part of other modules functionality such as CSEC and EEPROM emulation.
- Some recommendations should be followed as the ones presented in this video. Mainly, due to **only one transaction can be executed in the same partition!**

References

- [AN12003. Using S32K148 FlexNVM Memory](#)
- [AN11983. Using the S32K1xx EEPROM Functionality](#)
- [AN5401. Getting started with CSEc security module.](#)
- [S32K14x Series Reference Manual](#)

CSEc Configuration



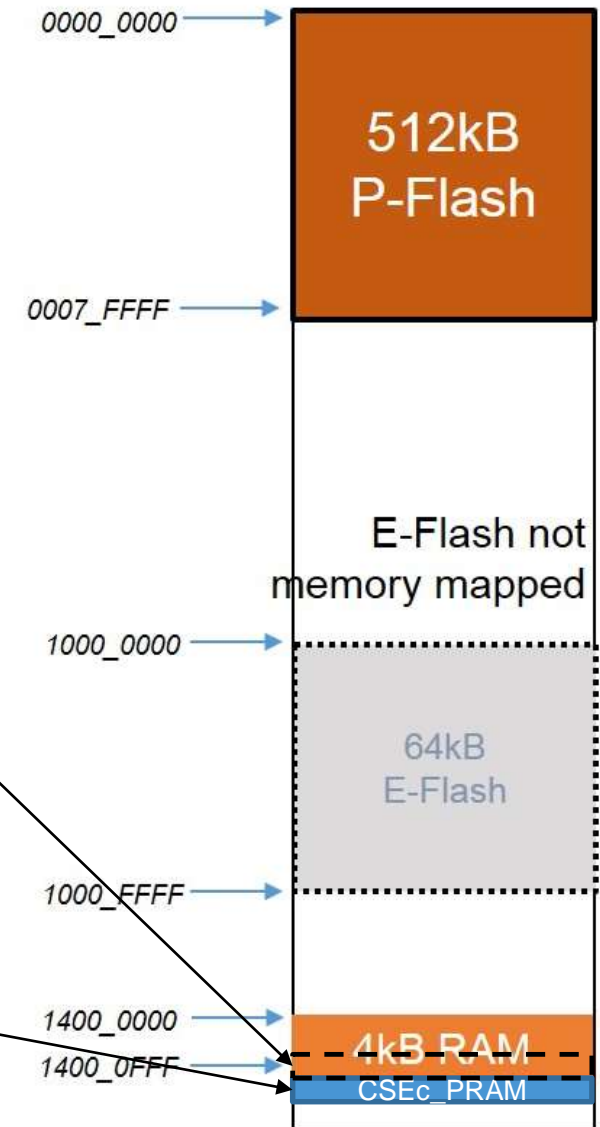
CSEc

FCCOB Number	FCCOB Contents [7:0]
0	0x80 (PGMPART)
1	CSEc Key Size
2	SFE
3	FlexRAM load during reset option (only bit 0 used): 0 - FlexRAM loaded with valid EEPROM data during reset sequence 1 - FlexRAM not loaded during reset sequence
4	EEPROM Data Set Size Code ¹
5	FlexNVM Partition Code ²

Memory space (not available and subtracted from EEERAM)

2'b00	Number of User Keys ({MASTER_ECU_KEY, BOOT_MAC_KEY, BOOT_MAC, KEY<n>})	Number of Bytes (subtracts from the total 4K EEERAM space)
2'b00	Zero	0
2'b01	1 to 5 keys	128 Bytes
2'b10	1 to 10 keys	256 Bytes
2'b11	1 to 20 keys	512 Bytes

Memory available for setting/getting CSEc parameters

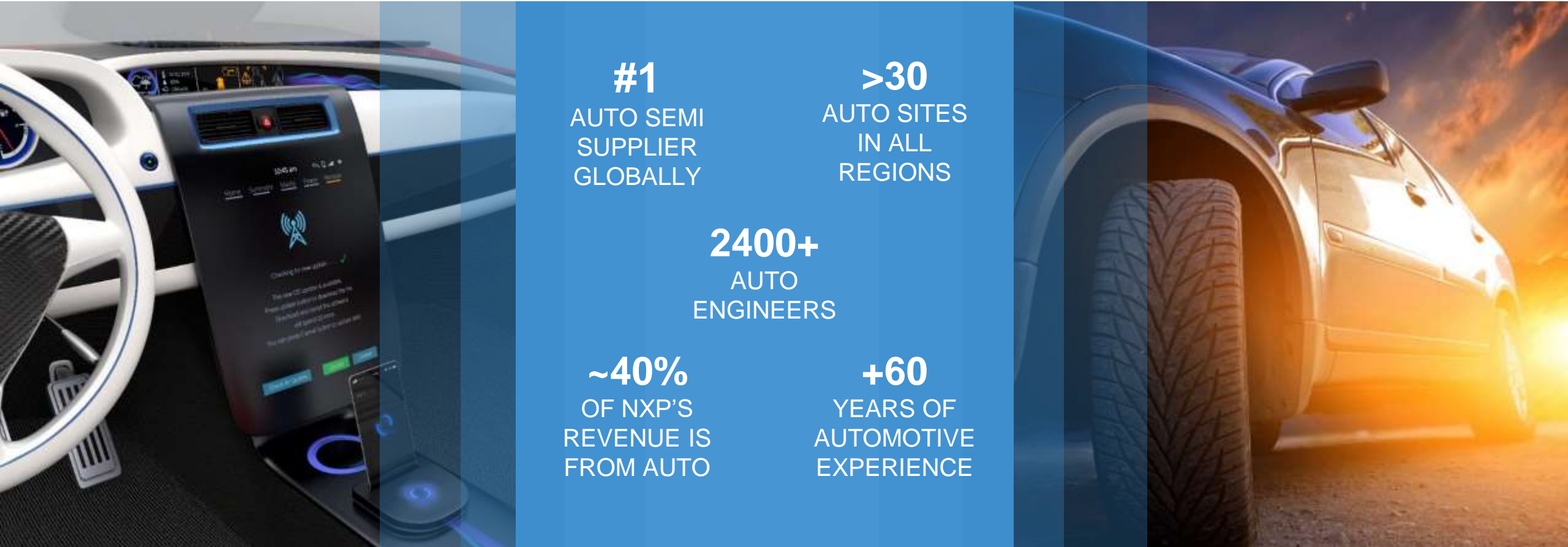


S32K144

The Need for Security



NXP – #1 Global Auto Semi Powerhouse



#1
AUTO SEMI
SUPPLIER
GLOBALLY

>30
AUTO SITES
IN ALL
REGIONS

2400+
AUTO
ENGINEERS

~40%
OF NXP'S
REVENUE IS
FROM AUTO

+60
YEARS OF
AUTOMOTIVE
EXPERIENCE

Increasing Connectivity = Increasing Risks

FBI: Estimated 3 Trillion USD Annual Damage from Hacking

Requiring maximum protection of . . .



Privacy

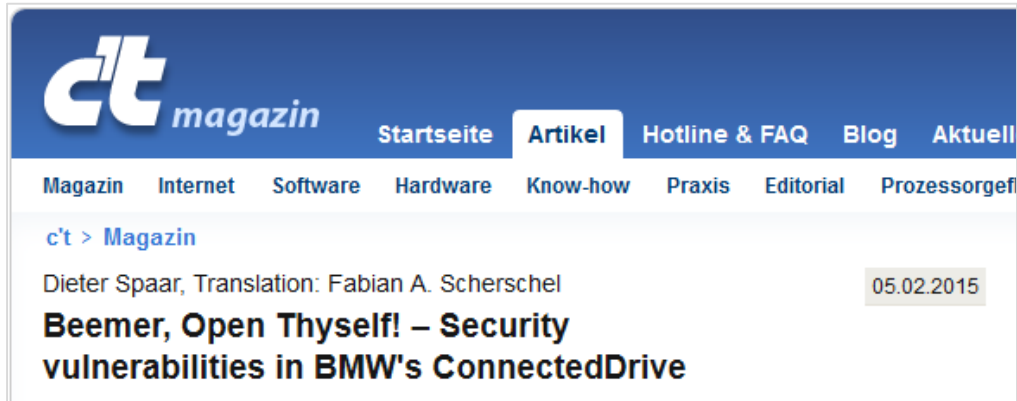


Personal Assets



Lives

Car Hacking is 'Hot'



ct magazin

Startseite Artikel Hotline & FAQ Blog Aktuell

Magazin Internet Software Hardware Know-how Praxis Editorial Prozessorgef

c't > Magazin

Dieter Spaar, Translation: Fabian A. Scherschel 05.02.2015

Beemer, Open Thyself! – Security vulnerabilities in BMW's ConnectedDrive



JALOPNIK

Damon Lavrinc

Filed to: CAR HACKING · 2/18/15 5:40pm

How A 14-Year-Old Hacked A Car With \$15 Worth Of Radio Shack Parts



Forbes / Security

2 FREE Issues of F

JUL 14, 2015 @ 12:00 PM 26,209 VIEWS

Tesla Model S Digital Weaknesses To Be Exposed By Hackers Next Month



Hackers Remotely Kill a Jeep on the Highway—With Me in It

BUSINESS DESIGN ENTERTAINMENT GEAR SCIENCE SECURITY

ANDY GREENBERG SECURITY 07.21.15 6:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT



BBC Sign in News Sport Weather Shop Earth More

NEWS

Home Video World UK Business Tech Science Magazine Entertainment & Arts

Technology

Car hack uses digital-radio broadcasts to seize control

By Chris Vallance 22 July 2015



engadget

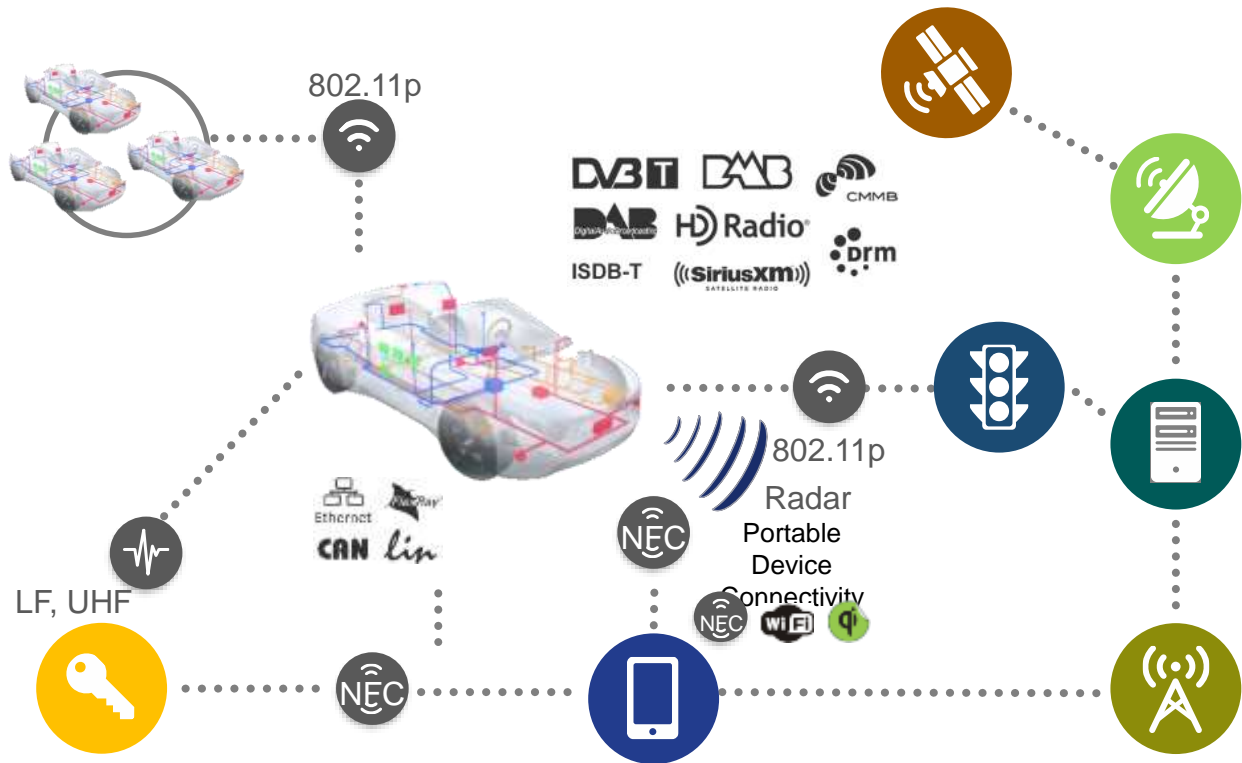
OnStar hack remotely starts cars, GM working on a fix

by Jessica Conditt | @jessconditt | July 30th 2015 At 1:58pm

The Connected Car...

A Cloud-connected Computer Network on Wheels

- **A networked computer**
 - up to 100 ECUs per car
 - and many sensors
 - inter-connected by wires
 - more and more software
- **Increasingly connected to its environment**
 - to vehicles & infrastructure
 - to user devices
 - to cloud services



... is an Attractive Target for Hackers!

Valuable Data

- Collection of data/info
- Storage of data
- Diagnostic functions



Protect Privacy

High Vulnerability

- Increasing number of nodes
- More advanced features
- X-by-Wire



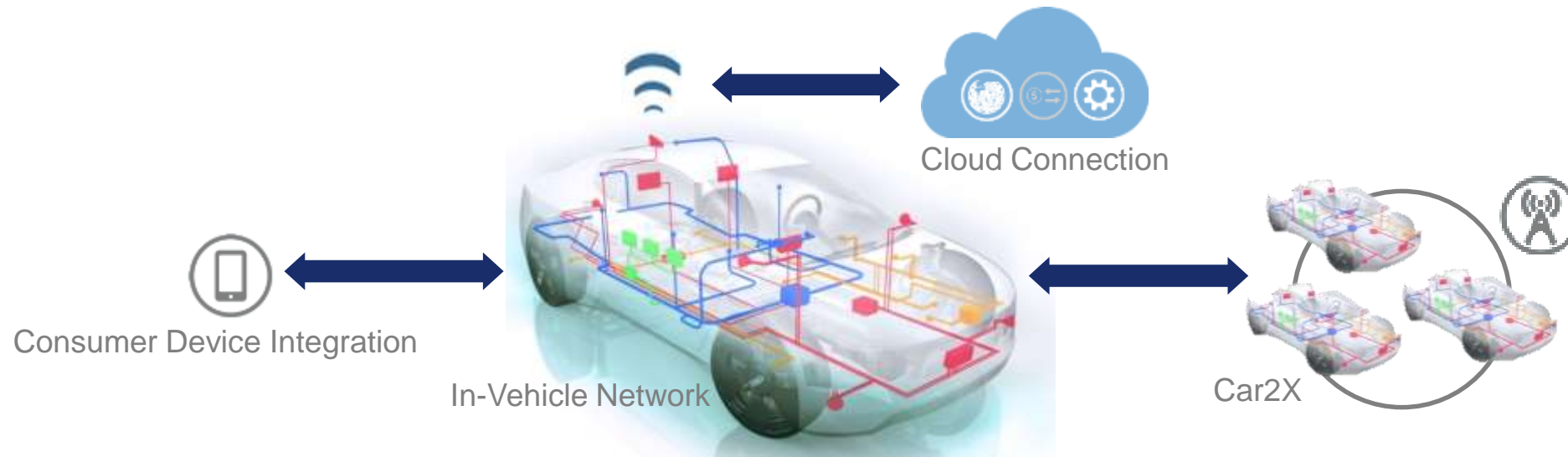
Increase Safety

Easy (Remote) Access

- Fully Connected Car
- External & internal interfaces
- Wired & wireless interfaces



Prevent Unauthorized Access



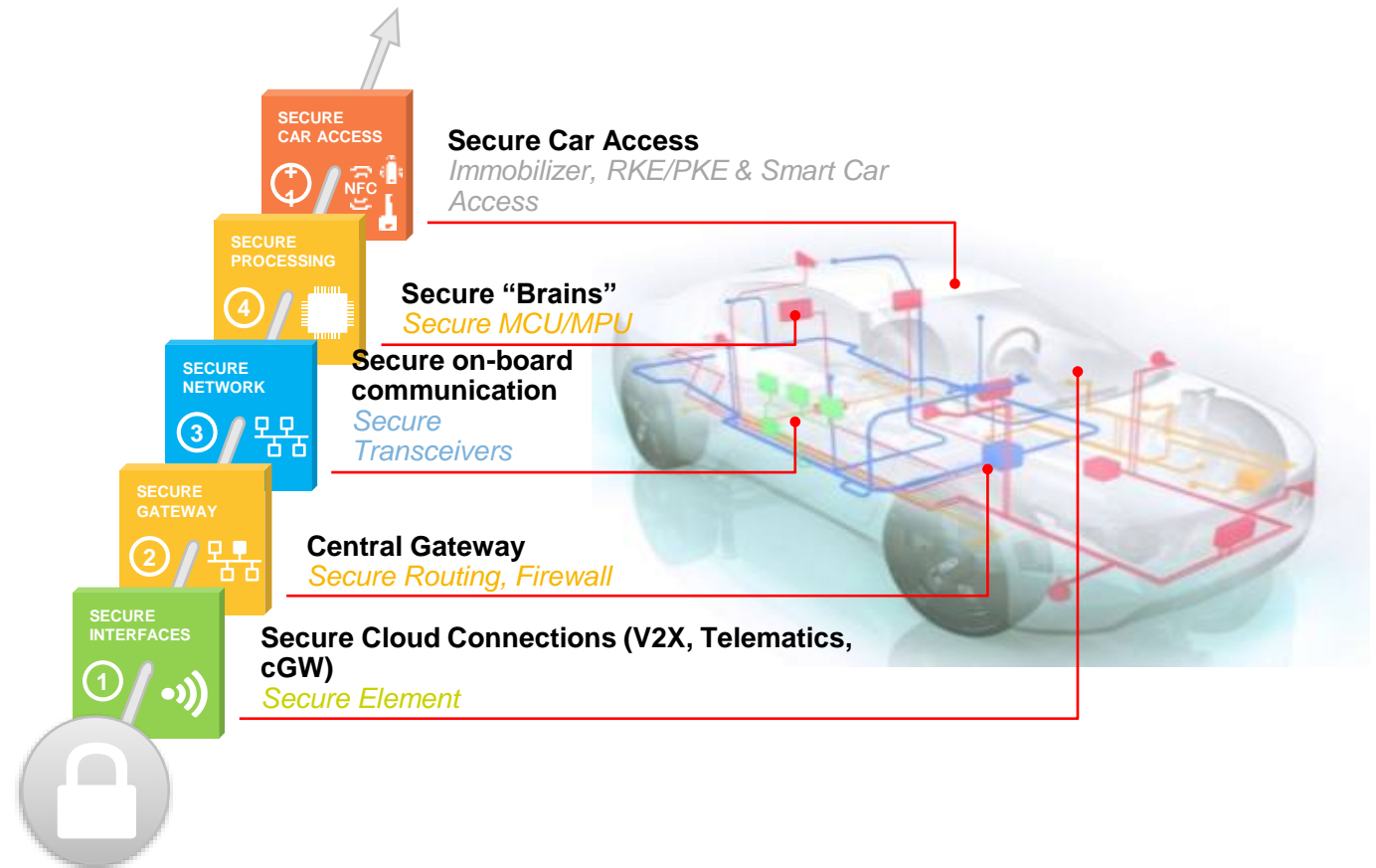
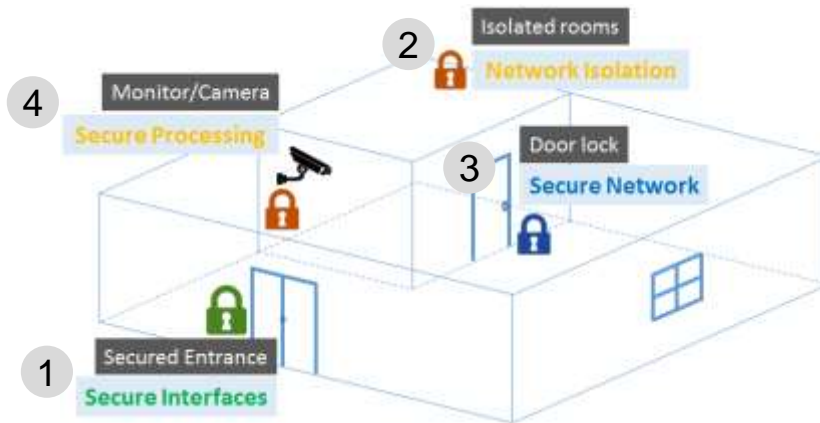
Layered Security Model



NXP Automotive Vehicle Security Architecture (4 +1 Solution)

Security Requires a Layered Approach For Connected Cars like your Home.

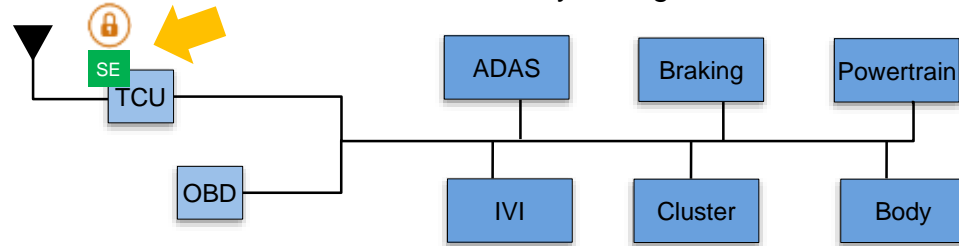
- Multiple security techniques, at different levels (a.k.a. defense-in-depth)
- To mitigate the risk of one component of the defense being compromised or circumvented



The 4 Layers to Securing an Automobile

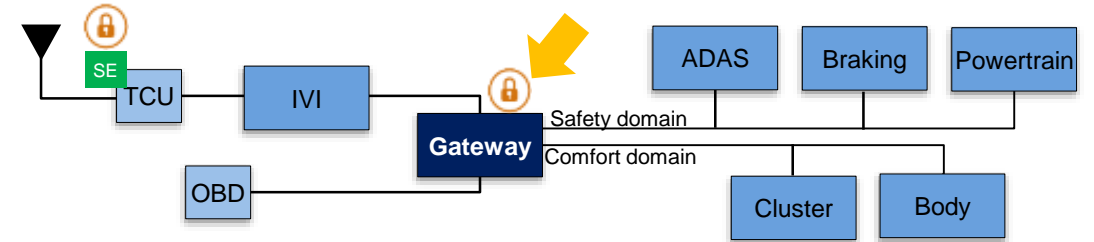
Layer 1: Secure Interface

Secure M2M authentication, secure key storage



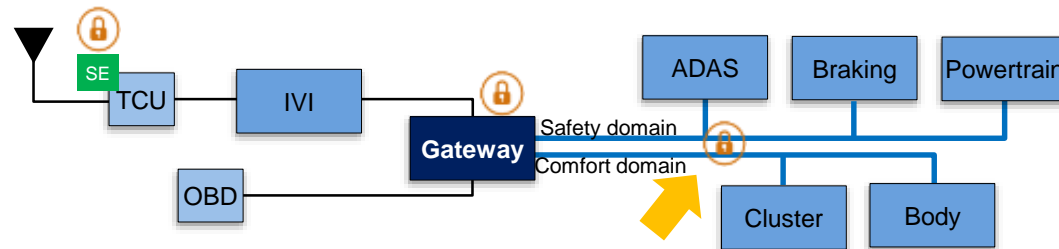
Layer 2: Secure Gateway

Domain isolation, firewall/filter, centralized intrusion detection (IDS)



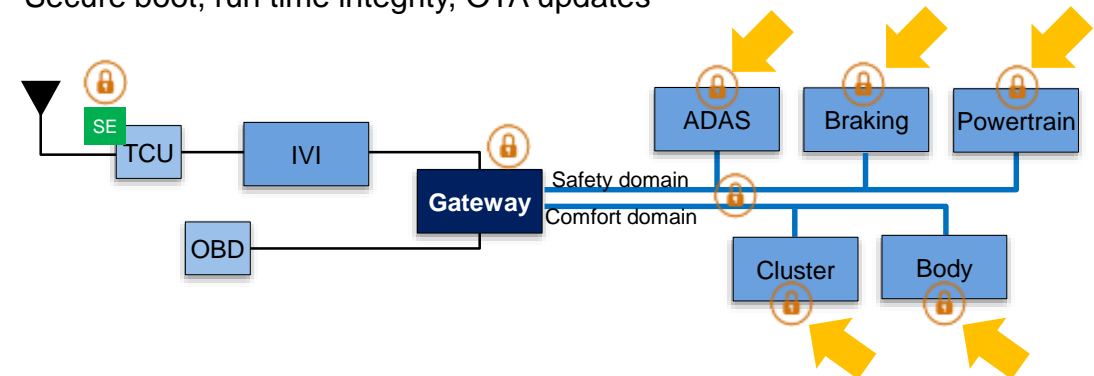
Layer 3: Secure Network

Message authentication, CAN ID killer, distributed intrusion detection (IDS)



Layer 4: Secure Processing

Secure boot, run time integrity, OTA updates



Layer +1 – Secure Car Access: What is It?

Immobilizer



- Car theft protection



Remote Keyless Entry (RKE)



Consisting of:

- Car theft protection
- Remote car door lock and unlock



Passive Keyless Entry (PKE)



Consisting of:

- Car Theft protection
- Remote car door lock and unlock
- Passive keyless entry
- Passive Start



Smart Car Management



Car-key communication for:

- Remote start
- Car finder
- Alarm Systems
- Tire pressure information
- Fuel level / Charging state
- Door lock status



Connected Keyless Entry

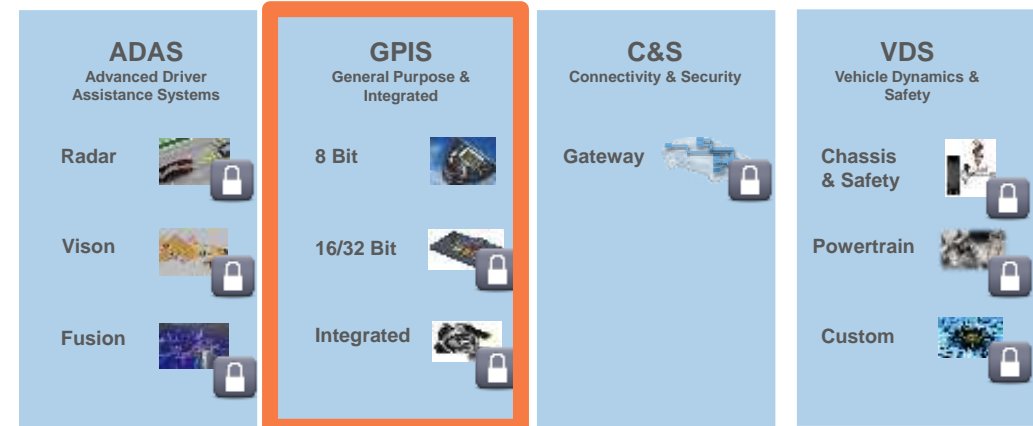
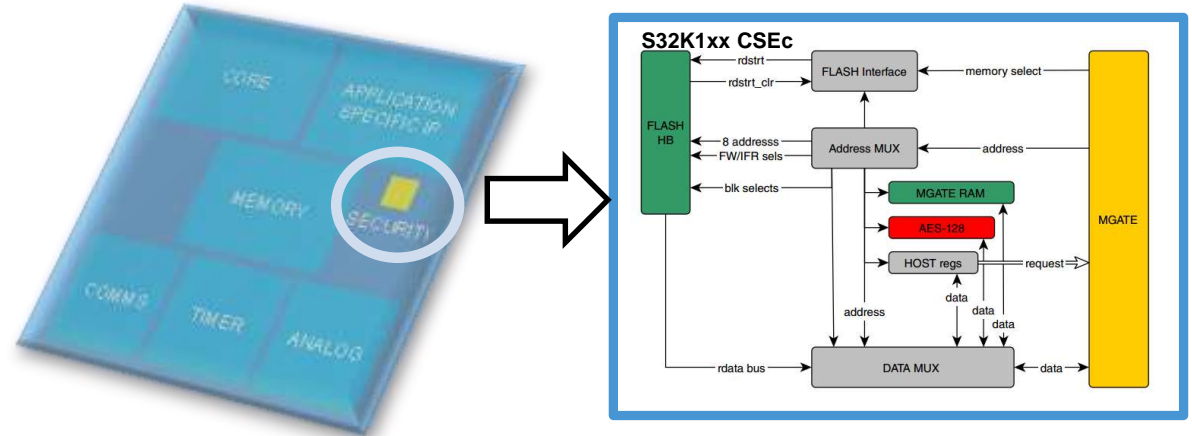


- Car Access via NFC enabled phones/wearables
- NFC key advantage: secure transport of keys
- Alternative: Car access via phone using BLE and key fob as 'Gateway'



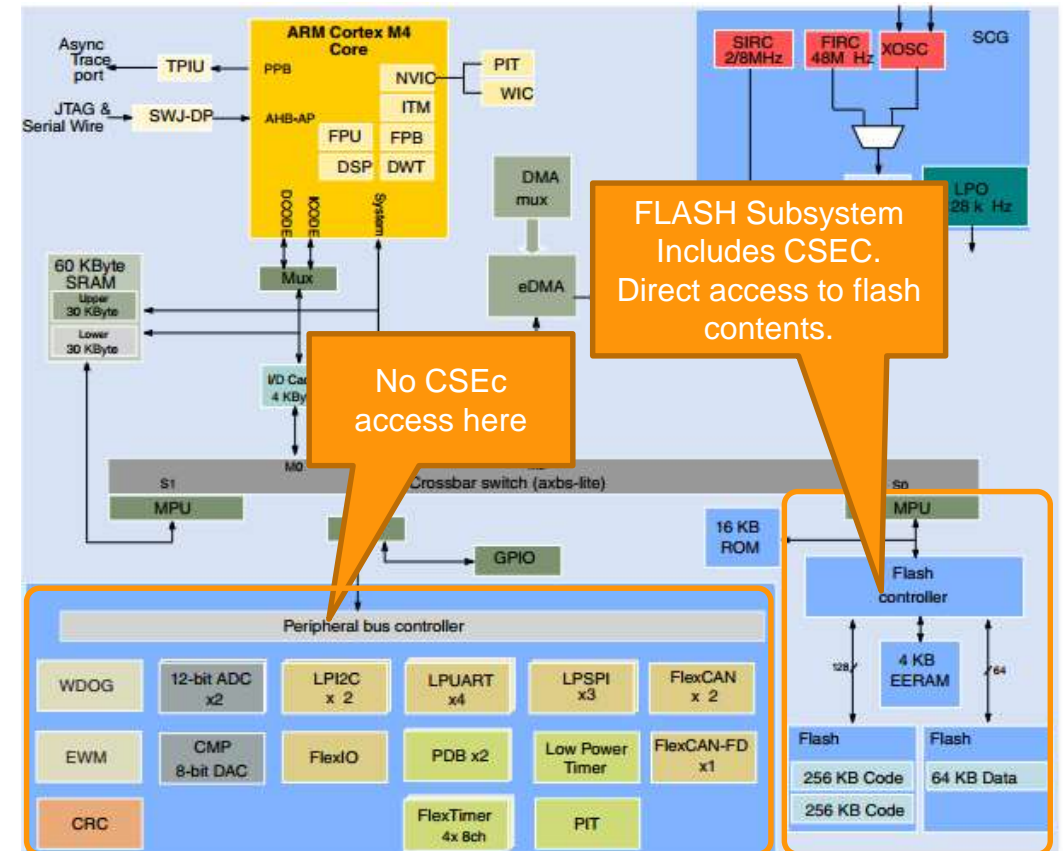
Layer 4 – Secure Processing: What is It?

- Secure MCU - Defined by hardware accelerated Crypto capability
- IP can be applied to any MCU/Processor
- Use cases:
 - CAN Message authentication
 - Secure boot – FW auth.
 - Key storage
 - Encryption
 - OTA software updates in the field



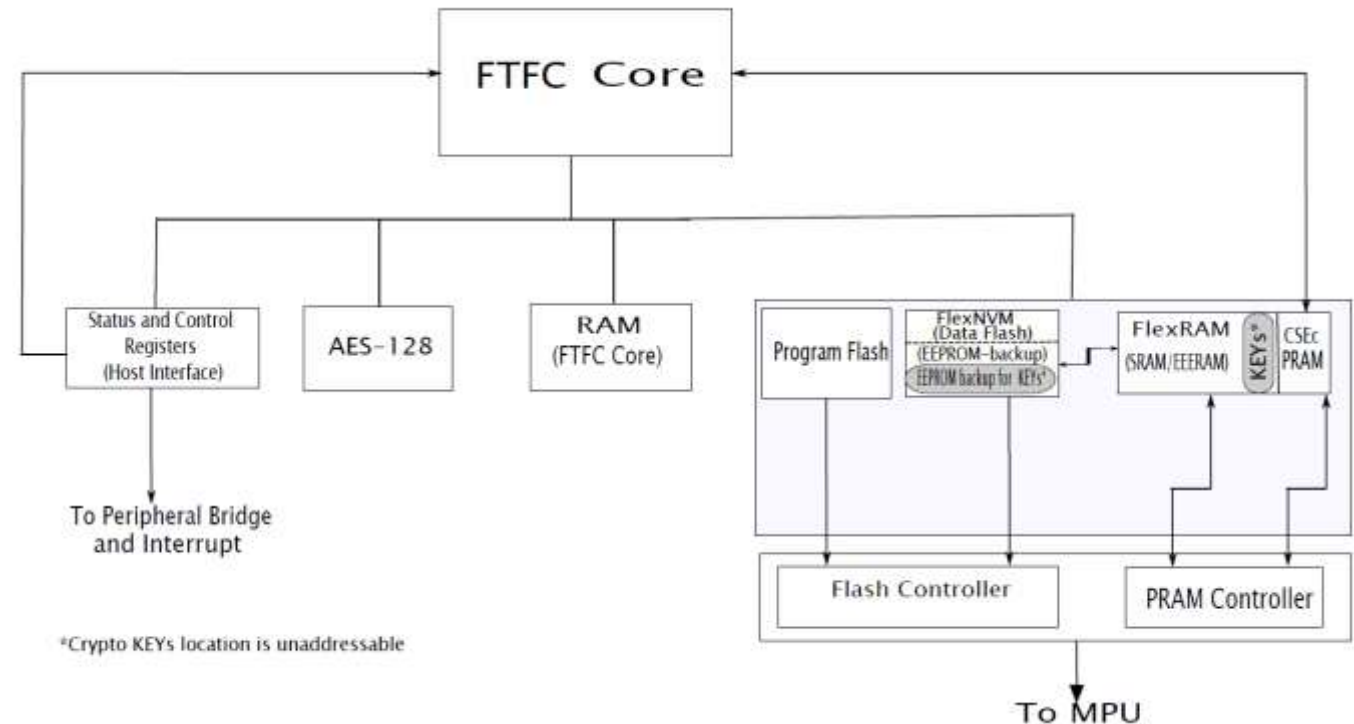
S32K Security Module (CSEc) – Overview

- Implemented directly in the flash system (close to the secure information)
- Direct memory access to the flash data for fast and simple secure boot support
- Data in SRAM / Peripheral are accessible via Core or DMA transfers.
- Supports the complete SHE Specification and the enhanced SHE+ features (more keys etc.)
- Small easy to use security implementation



S32K Security Module (CSEc) – Overview

- FTFC core is utilized for processing both FLASH as well as CSEc commands
- Host Interface is used to issue flash commands and read back flash-subsystem status(including CSEc).
 - Registers like FCCOB
- FlexNVM and FlexRAM are configured to work together to emulate EEPROM
- Part of emulated-EEPROM is used as secure storage
- CSEc_PRAM is programming interface for CSEc operations



S32K Security Module (CSEc) – Commands

- CSEc Commands to FTFC.
- CCOB command set is effectively extended to include SHE commands related to ECB, CBC and CMAC features.
- Similar protocol to the FCCOB commands, CCOB interface will be locked until completion.
- CSEc command constructed by writing data to a Parameter Memory (PRAM) followed by a command header.
- Operation Start as indicated by CCIF, transition from 1 to 0.
- Operation complete: CCIF transition from 0 to 1. User read PRAM to verify results.

Figure 32-11. Generic PRAM interface

Bits	[127:0]															
Bits	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0
WD	Word 0				Word 1				Word 2				Word 3			
	Byte															
Page	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	FuncID	FuncFormat	CallSeq	KeyID	Error Bits			Command Specific								
1	Data Input to CSEc OR Data Output from CSEc															
2																
3																
4																
5																
6																
7																

SHE Specification Overview



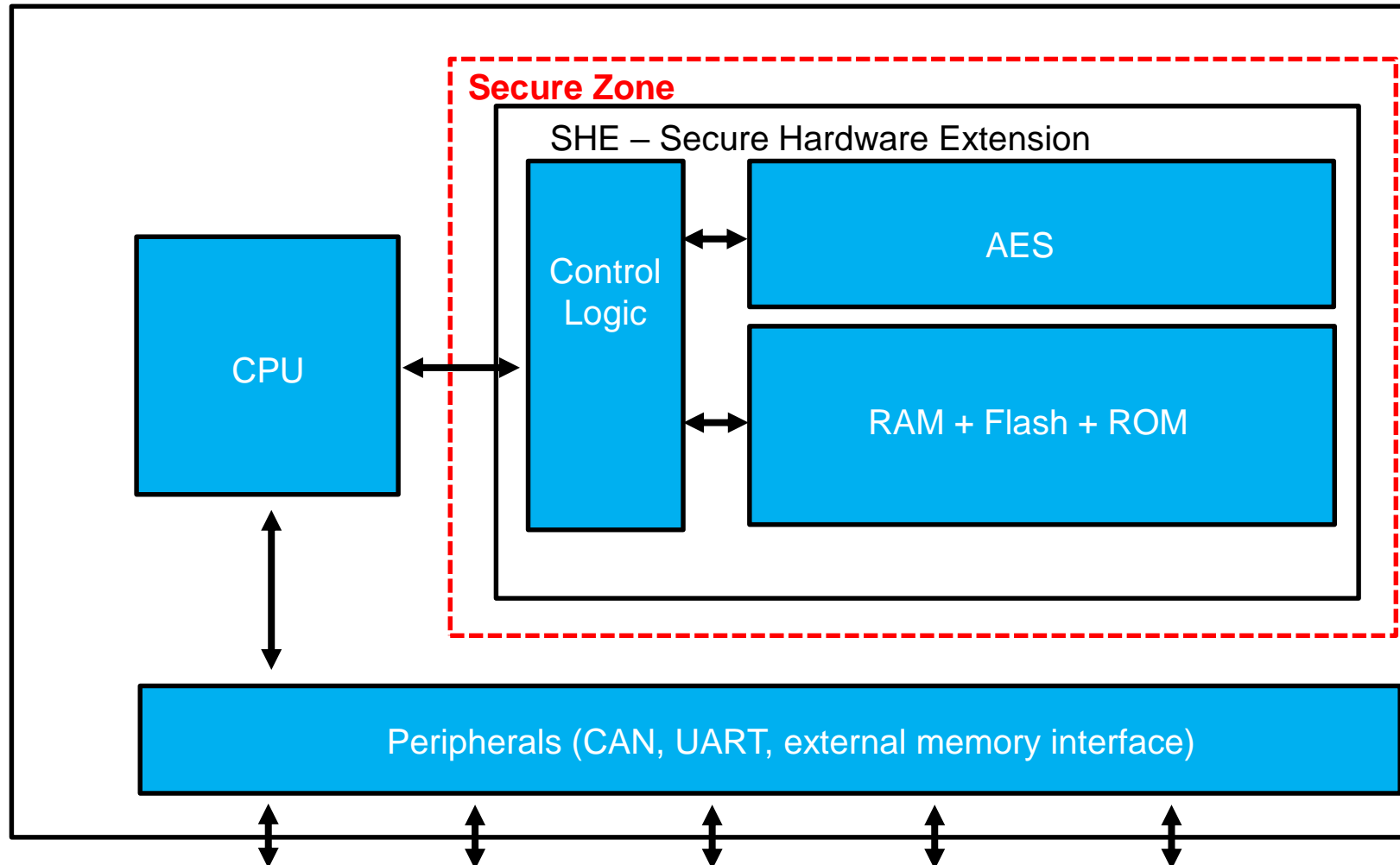
SHE – Secure Hardware Extension: Introduction

- The Secure Hardware Extension (SHE) is an on-chip extension to any given microcontroller. It is intended to move the control over **cryptographic keys** from the software domain into the **hardware domain** and therefore protect those keys from software attacks. However, it is not meant to replace highly secure solutions like TPM chips or smart cards, i.e. no tamper resistance is required by the specification.
- **The main goals are**
 - Protect cryptographic keys from software attacks
 - Provide an authentic software environment
 - Let the security only depend on the strength of the underlying algorithm and the confidentiality of the keys
 - Allow for distributed key ownerships
 - Keep the flexibility high and the costs low

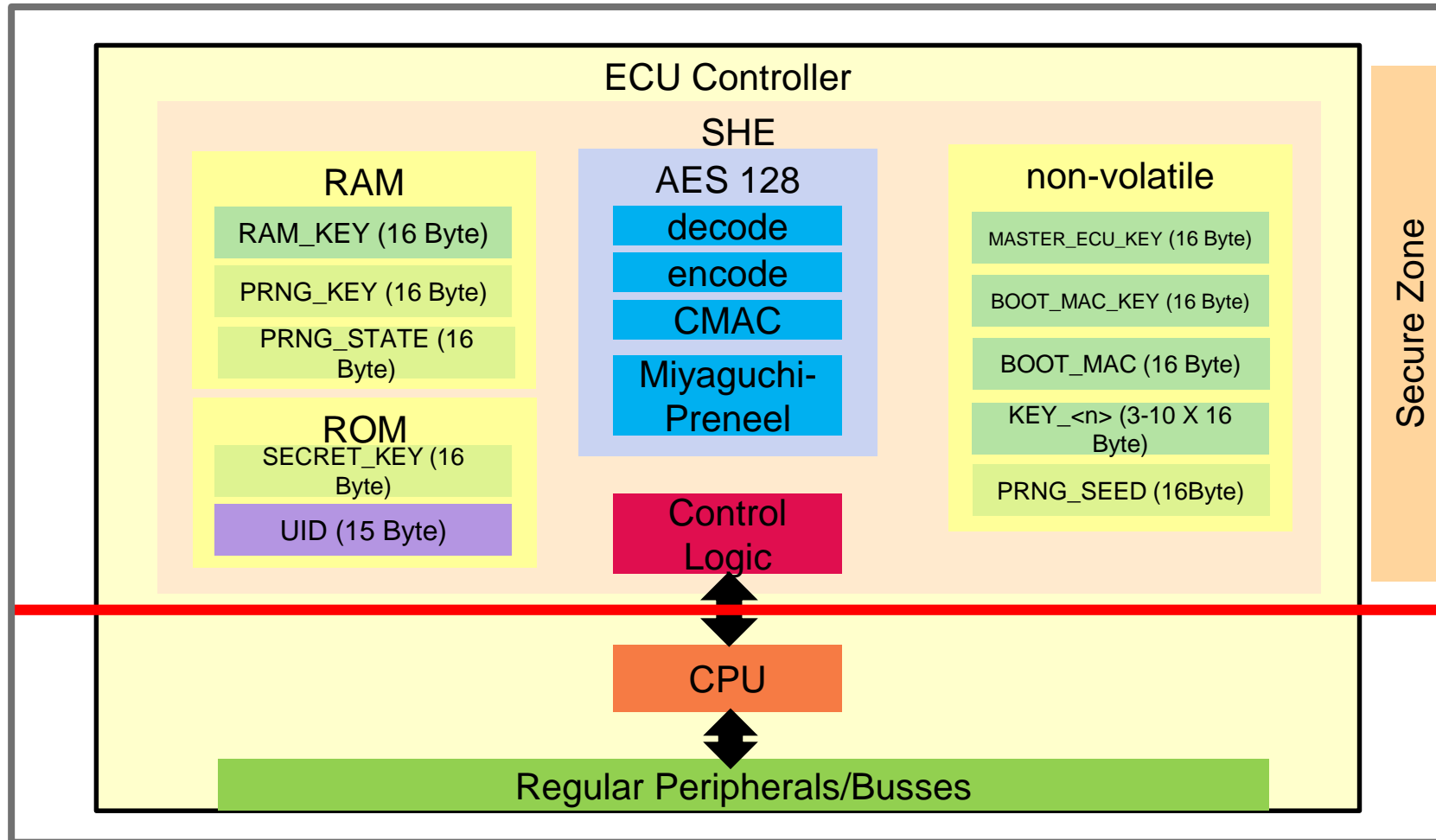
SHE Specification – Introduction

- Note – all information is in reference to the official HIS / SHE (Secure Hardware Extension) specification version 1.1 – Rev:: 439 - 01.04.2009
- The Re-view of the Spec. was done by Freescale/NXP in an early phase
- Key features to attain goals of the SHE specification are:
 - A secure storage for crypto keys
 - Crypto algorithm acceleration (AES-128)
 - Secure Boot mechanism to verify custom firmware after reset
 - Offers 19 security specific functions
 - Up to 10 general and 5 special purpose crypto keys

Simplified Block Diagram of the SHE Specification



Detailed Block Diagram of the SHE Specification



SHE Specification – Algorithms

- Encryption / Decryption

- SHE has to support the **Electronic Cipher Cook mode (ECB)** for processing single blocks of data and the **Cipher Clock Chaining mode (CBC)** for processing larger amounts of data

- MAC Generation / Verification

- The MAC generation and verification has to be implemented as a CMAC using the **AES-128** as specified by [NIST800_38B]

- Compression Function

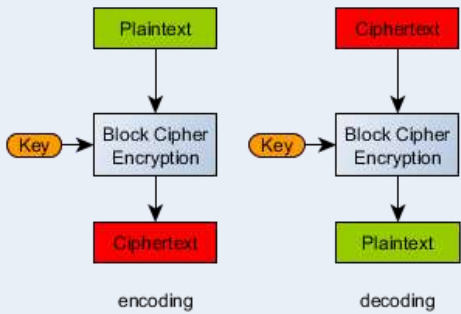
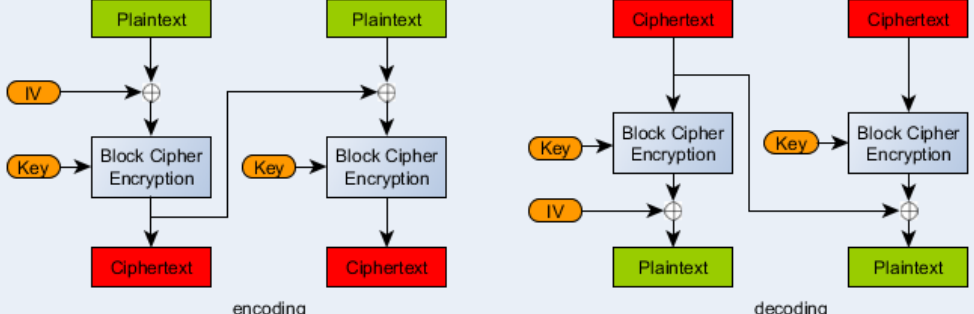




- The **Miyaguchi-Preneel** construction (see [HAC] Algorithm 9.43) with the AES as block cipher is used as compression function within SHE. Messages have to be preprocessed before feeding them to the compression algorithm, i.e. they have to be padded and parsed into 128 bit chunks.

- Key Derivations

- Keys are derived using the **Miyaguchi-Preneel** compression algorithm based on [NIST800_108].

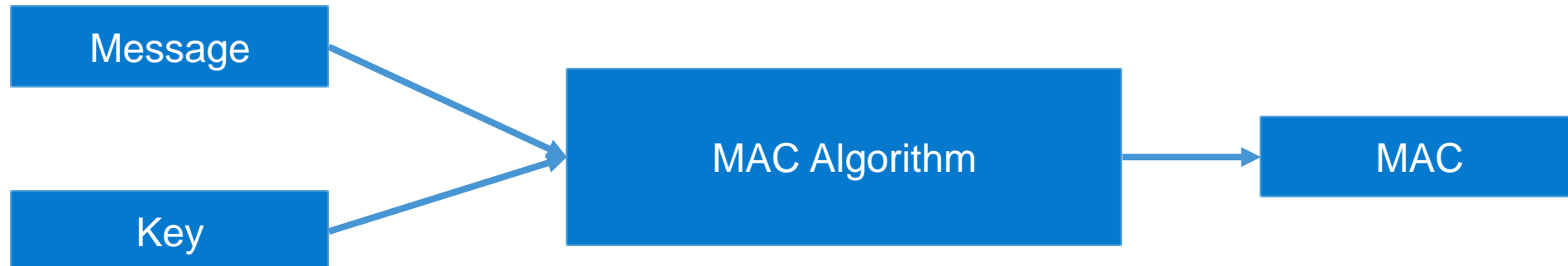
SHE Specification – Cipher Modes

AES
Encryption/
Decryption in
ECB or CBC
mode

	Electronic codebook (ECB)	Cipher-block chaining (CBC)
Scheme	Each block is encoded/decoded independently from the others	Previous result is XORed with actual plaintext
Diagram	 <p>The diagram illustrates the Electronic Codebook (ECB) mode. It shows two separate processes: encoding and decoding. In encoding, a 'Plaintext' block (green) is input to a 'Block Cipher Encryption' block (blue), which also receives a 'Key' (orange). The output is a 'Ciphertext' block (red). In decoding, a 'Ciphertext' block (red) is input to a 'Block Cipher Encryption' block (blue), which also receives a 'Key' (orange). The output is a 'Plaintext' block (green). The processes are independent of each other.</p>	 <p>The diagram illustrates the Cipher-Block Chaining (CBC) mode. It shows two separate processes: encoding and decoding. In encoding, a 'Plaintext' block (green) is XORed with an 'IV' (orange) and then input to a 'Block Cipher Encryption' block (blue), which also receives a 'Key' (orange). The output is a 'Ciphertext' block (red). In decoding, a 'Ciphertext' block (red) is input to a 'Block Cipher Encryption' block (blue), which also receives a 'Key' (orange). The output is XORed with the previous 'Ciphertext' block (red) to produce a 'Plaintext' block (green). The processes are dependent on each other.</p>
Pro	Random access possible	Secure for long messages, decryption can be parallelized
Cont	Insecure for longer messages (statistical analysis)	Encryption takes longer since you have to wait for each block
Example	 	 

SHE Specification – CMAC Generator

- Cipher based Message Authentication Code (CMAC)
- A MAC algorithm inputs:
 - Secret key
 - Message of arbitrary length
- A MAC algorithm output:
 - MAC value
 - The MAC value protects both a message's data integrity as well as its authenticity.



SHE: Non-volatile Memory Slots

(Including Extension on CSEc)

Key Name	Key Block Select (KBS)	Address (KeyID)	Memory Area	Description
SECRET_KEY	X	0x0	ROM	Inserted during chip fabrication by the semiconductor manufacturer and should not be stored outside of SHE
UID (Unique identification)	X	0x0	ROM	A serial number of at most 120 bits. Inserted during chip fabrication by the semiconductor manufacturer.
MASTER_ECU_KEY	X	0x1	NVM	Only used for updating other memory slots inside of SHE
BOOT_MAC_KEY	X	0x2	NVM	Used by the secure booting mechanism to verify the authenticity of the software. The BOOT_MAC_KEY may also be used to verify a MAC.
BOOT_MAC	X	0x3	NVM	Used to store the MAC of the Bootloader of the secure booting mechanism and may only be accessible to the booting mechanism of SHE
KEY_<1 - 10>	1'b0	0x4 – 0xD	NVM	Can be used for arbitrary functions. The usage of the keys has to be selected between encryption/decryption or MAC generation/verification on programming time by setting the key usage flag accordingly. <i>KEY_<11 – 17> is extended on CSEc over SHE</i>
KEY_<11 - 17>	1'b1	0x4 – 0xA	NVM	
RESERVED	1'b0 / 1'b1	0xE	RESERVED	RESERVED
RAM_KEY	X	0xF	SRAM	Can be used for arbitrary operations. Can be exported if it was loaded as plaintext.

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overall data bits
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overlaid Data
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

If set, the key cannot ever be updated even if an authorizing key (secret) is known

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Courier	Overall data bits
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

If set, key cannot be used if MAC value comparison failed at Boot

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overall Data bit
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

If set, key cannot be used if a debugger is (or has ever been) connected to the MCU

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overlaid data bits
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

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

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overlaid data
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

Determines if a key can be used for encryption/decryption or for MAC generation/verification (CMAC).

Set: MAC

Clear: Encryption/Decryption

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Keys

Key values moved from public memory space to secure memory space.

The secure memory space is only accessible by the security module. Application work with key references!

	Write Protection	Secure Boot Failure	Debugger Activation	Wildcard UID	Key Usage	Plain Key	Counter	Overall data bits
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
RAM_KEY						X		129
SECRET_KEY		X ¹	X ¹					128
UID								120

28 bit counter. Must be increase on every update.

¹ SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

SHE Specification – Functions

#	SHE – Functions	Usage
1	CMD_ENCRYPT_ECB	Encryption / Decryption
2	CMD_ENCRYPT_CBC	
3	CMD_DECRYPT_ECB	
4	CMD_DECRYPT_CBC	
5	CMD_GENERATE_MAC	Signing / Authentication
6	CMD_VERIFY_MAC	
7	CMD_LOAD_KEY	Key Management
8	CMD_LOAD_PLAIN_KEY	
9	CMD_EXPORT_RAM_KEY	
10	CMD_INIT_RNG	Random Number System
11	CMD_EXTEND_SEED	
12	CMD_RND	
13	CMD_SECURE_BOOT	Secure Boot
14	CMD_BOOT_FAILURE	
15	CMD_BOOT_OK	
16	CMD_GET_STATUS	Module Handling
17	CMD_GET_ID	
18	CMD_CANCEL	
19	CMD_DEBUG	

Can anybody add/update keys?

No!

User must know the authorizing key before updating a key

Note: In factory, for the very first time: use default value of key – i.e. all 1s

$$\begin{aligned}
 K_1 &= \text{KDF}(K_{\text{AuthID}}, \text{KEY_UPDATE_ENC_C}) \\
 K_2 &= \text{KDF}(K_{\text{AuthID}}, \text{KEY_UPDATE_MAC_C}) \\
 M_1 &= \text{UID}'|\text{ID}|\text{AuthID} \\
 M_2 &= \text{ENC}_{\text{CBC}, K_1, \text{IV}=0}(\text{C}_{\text{ID}}'|\text{F}_{\text{ID}}'|"0...0"_{95}|K_{\text{ID}}') \\
 M_3 &= \text{CMAC}_{K_2}(M_1|M_2)
 \end{aligned}$$

CMD_LOAD_KEY
stores key value in secure
NVM

Note:

To be able to update a key you have to know the actual key value or the MASTER_ECU_KEY value.

SHE Specification – Memory Update Protocol

- To add user keys the protocol as defined in the SHE specification must be used.
- This ensures confidentiality, integrity, authenticity and protects against replay attacks.
- To update the memory containing the keys the following must be calculated and passed to CSE: K1, K2, M1 ,M2 and M3.

Key	Calculation	Size
K1	$KDF(K_{AuthID}, KEY_UPDATE_ENC_C)$ KDF is key derivation function	128 bit
K2	$KDF(K_{AuthID}, KEY_UPDATE_MAC_C)$ KDF is key derivation function	128 bit
M1	UID' ID AuthID - 256 bits	128 bit
M2	$ENC_{CBC,K1,IV=0}(C_{ID}' F_{ID}' "0...0"_{95} K_{ID}')$ CBC encryption using K1	256 bit
M3	$CMAC_{K2}(M_1 M_2)$ CMAC calculation using K2	128 bit

SHE – Random Number Generators

- **Use Case**
 - Key generation
 - Noise/Salt to prevent re-play attacks
- **Pseudo Random Number Generation (PRNG)**
 - re-producible value generated by a deterministic algorithm
 - digital IP
 - fast
- **TRUE Random Number Generation (TRNG)**
 - value generated via measurement of physical effects (e.g. thermal noise)
 - includes analog elements (e.g. simple A/D-converter)
 - slow

Lab #2 How to Store Secret Keys



How to Store Secret Keys

- Task

- Initialize pseudo random number generator
- Generate random number
- Generate secret keys
- Store secret keys into secure memory

- Learn

- How to use CSEc programming interface(CSEc PRAM) for your security operations?

- Note

- You will need to run S32K144_EVB_CSEc_Step2_CreateStore_Keys project for this Lab.

How to Store Keys in the CSEC Secure Memory

- How to store keys?
 - Recall: all you got is CSEc PRAM
 - Use LOAD_KEY command to store keys to the secure memory slot
 - Takes Crypto key, Security Flags and Counter value in an **encrypted form only**
 - On command completion outputs encrypted values only
 - This is to **validate successful key update**
- The first Key to be loaded is MASTER_ECU_KEY
 - It is used to update all other keys
- By default all keys have value all 1s(i.e.0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF)

Random Number Generator

- Pseudo random number generator
- Command: CMD_INIT_RNG

Input Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x0A	0x00	0x00	KeyID	Error Bits		Reserved									
1	Reserved															
2																
3																
4																
5																
6																
7																

Input Parameters: What is an Encrypted Form?

Refer AN5401 for detail description

AuthID: the keyID of the key that authorizes the key update

CID' – the new counter value (28 bits). Starts from 0x0000001

FID' – New Protection flags

- For SFE == 0x00:
WRITE_PROT | BOOT_PROT |
DEBUG_PROT | KEY_USAGE |
WILD_CARD (5 bits)
- For SFE == 0x01:
WRITE_PROT | BOOT_PROT |
DEBUG_PROT | KEY_USAGE |
WILD_CARD | VERIFY_ONLY (6 bits)

- SHE specification defines the secure memory update protocol

– Supply keys in terms of M1, M2 and M3

▪ M1 = UID'|ID|AuthID – 128 bits

▪ M2 = ENC_{CBC, K1, IV=0}(CID'|FID'|"0...0"95|KID') – 256 bits : SFE==0x00

M2 = ENC_{CBC, K1, IV=0}(CID'|FID'|"0...0"94|KID') – 256 bits : SFE==0x01

▪ M3 = CMAC_{K2}(M1|M2) – 128 bits

• K1 = KDF(KEY_{AuthID}, KEY_UPDATE_ENC_C)

• K2 = KDF(KEY_{AuthID}, KEY_UPDATE_MAC_C)

KEYID' – The new key value (128 bits)

KEY_{AuthID}: Authorizing key value

CSEc PRAM input for LOAD_KEY command

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0x07	0x00	0x01	KeyID	Error	bits	Reserved										
1								M1 [0:15]									
2								M2 [0:15]									
3																	
4	3'0 KBS(1bit) KeyIDx(4bit)							M3 [0:15]									
5								Reserved									
6																	
7																	

Output Parameters

- $M4 = \text{UID}|\text{ID}|\text{AuthID}|M4^*$
- $M5 = \text{CMAC}_{K4}(M4)$

CSEc PRAM input for LOAD_KEY command

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x07	0x00	0x01	KeyID	Error Bits	Reserved										
1	M1 [0:15]															
2	M2 [0:15]															
3																
4	M3 [0:15]															
5	M4 [0:31]															
6																
7	M5 [0:15]															

Let's Load the KEY

- Used CSEc to generate M1, M2, M3, M4 and M5
- Load keys using LOAD_KEY command
- Compare user generated M4/M5 value with CSEc PRAM returned M4/M5 value
 - This validates that key write was successful

Keys Used In This Lab

Keys used in this lab can be found in CSEc_keys.h in the project

```
#ifndef CSEC_KEYS_H_
#define CSEC_KEYS_H_

uint32_t BLANK_KEY_VALUE[4] = {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF}; //When key value is not
written it is all FFs
uint32_t MASTER_ECU_KEY_VALUE[4] = {0xD275F12C, 0xA863A7B5, 0xF933DF92, 0x6498FB4D}; //MASTER_ECU_KEY
uint32_t BOOT_MAC_KEY_VALUE[4] = {0x12340000, 0x00000000, 0x00000000, 0x00005678}; //BOOT_MAC_KEY
uint32_t KEY_1_VALUE[4] = {0x2FF8B03C, 0x5C540546, 0x5A9C94BD, 0x2D863279}; //KEY_1
uint32_t KEY_11_VALUE[4] = {0x85852FF8, 0xE7860C89, 0xB3AB9D63, 0xB8D6288F}; //KEY_11
uint32_t RAM_KEY_VALUE[4] = {0x68B674CB, 0x8198A250, 0x3A285100, 0xF4DDC40A}; //RAM_KEY

#endif /* CSEC_KEYS_H_ */
```

Code to Create and Load Keys Into CSEc

Code can be found in main.c in the project

```
csec_error = INIT_RNG();

/* Load MASTER_ECU_KEY */
calculate_M1_to_M5(M1, M2, M3, M4, M5, BLANK_KEY_VALUE, MASTER_ECU_KEY_VALUE, MASTER_ECU_KEY, MASTER_ECU_KEY, 1, 0); /* Calculate M1 to M5 in Software */
csec_error = LOAD_KEY(M4_out, M5_out, M1, M2, M3, MASTER_ECU_KEY); /* Load the key using SW calculated M1 to M3, and it returns M4 and M5 */
result = compare_results(M4, M4_out); /* Compare M4 generated by SW with the M4_out returned by CSEc */

/* Load KEY_1 */          /* Calculate M1 to M5 in Software, Authorizing Key = Master ECU Key */
calculate_M1_to_M5(M1, M2, M3, M4, M5, MASTER_ECU_KEY_VALUE, KEY_1_VALUE, MASTER_ECU_KEY, KEY_1, 1, 0);
csec_error = LOAD_KEY(M4_out, M5_out, M1, M2, M3, KEY_1); /* Load the key using M1 to M3, returns M4 and M5 */
result = compare_results(M4, M4_out); /* Compare M4 generated by SW with the M4_out returned by CSEc */

/* Load KEY_11 */          /* Calculate M1 to M5 in Software, Authorizing Key = Master ECU Key, Key Usage=1(for CMAC operations) */
calculate_M1_to_M5(M1, M2, M3, M4, M5, MASTER_ECU_KEY_VALUE, KEY_11_VALUE, MASTER_ECU_KEY, KEY_11, 1, 0x04);
csec_error = LOAD_KEY(M4_out, M5_out, M1, M2, M3, KEY_11); /* Load the key using M1 to M3, returns M4 and M5 */
result = compare_results(M4, M4_out); /* Compare M4 generated by SW with the M4_out returned by CSEc */
```

Initialize Random Number Generator (RNG)

Code can be found in CSEc_functions.c in the project

```
/* Initialize Random Number Generator */
uint16_t INIT_RNG(void)
{
    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != FTFC_FSTAT_CCIF_MASK); //Check for the ongoing FLASH command

    FTFC->FSTAT = (FTFC_FSTAT_FPVIOL_MASK | FTFC_FSTAT_ACCERR_MASK); // Write 1 to clear error flags

    /* Start command by writing Header */
    CSE_PRAM->RAMn[0].DATA_32= (CMD_INIT_RNG << 24) | (CMD_FORMAT_COPY << 16) | (CALL_SEQ_FIRST << 8) | (0x00);
    //Write to Page0 Word0, Value = 0x0A000000

    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != 0x80); //Check for the ongoing FLASH command

    csec_error_bits = CSE_PRAM->RAMn[1].DATA_32 >> 16; //Read Page0 Word1, Error Bits

    return csec_error_bits;
}
```

Load Keys

Code can be found in CSEc_functions.c in the project

```
uint16_t LOAD_KEY(uint32_t *M4_out, uint32_t *M5_out, uint32_t *M1_in, uint32_t *M2_in, uint32_t *M3_in, uint8_t key_id)
{
    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != FTFC_FSTAT_CCIF_MASK); //Check for the ongoing FLASH command

    FTFC->FSTAT = (FTFC_FSTAT_FPVIOL_MASK | FTFC_FSTAT_ACCERR_MASK); // Write 1 to clear error flags

    for(i=4,j=0; i<8; i++,j++) //Write to Page1
    CSE_PRAM->RAMn[i].DATA_32 = M1_in[j];

    for(i=8,j=0; i<16; i++,j++) //Write to Page2-3
    CSE_PRAM->RAMn[i].DATA_32 = M2_in[j];

    for(i=16,j=0; i<20; i++,j++) //Write to Page4
    CSE_PRAM->RAMn[i].DATA_32 = M3_in[j];

    /* Start command by writing Header */
    CSE_PRAM->RAMn[0].DATA_32= (CMD_LOAD_KEY << 24) | (CMD_FORMAT_COPY << 16) | (CALL_SEQ_FIRST << 8) | key_id; // Write to Page0 Word0, Value =
    0x07000000 | key_id

    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != 0x80); //Check for the ongoing FLASH command

    csec_error_bits = CSE_PRAM->RAMn[1].DATA_32 >> 16; //Read Page0 Word1, Error Bits

    for(i=20,j=0; i<28; i++,j++) //Read from Page5-6
    M4_out[j] = CSE_PRAM->RAMn[i].DATA_32;

    for(i=28,j=0; i<32; i++,j++) //Read from Page7
    M5_out[j] = CSE_PRAM->RAMn[i].DATA_32;

    return csec_error_bits;
}
```

CSEC Details

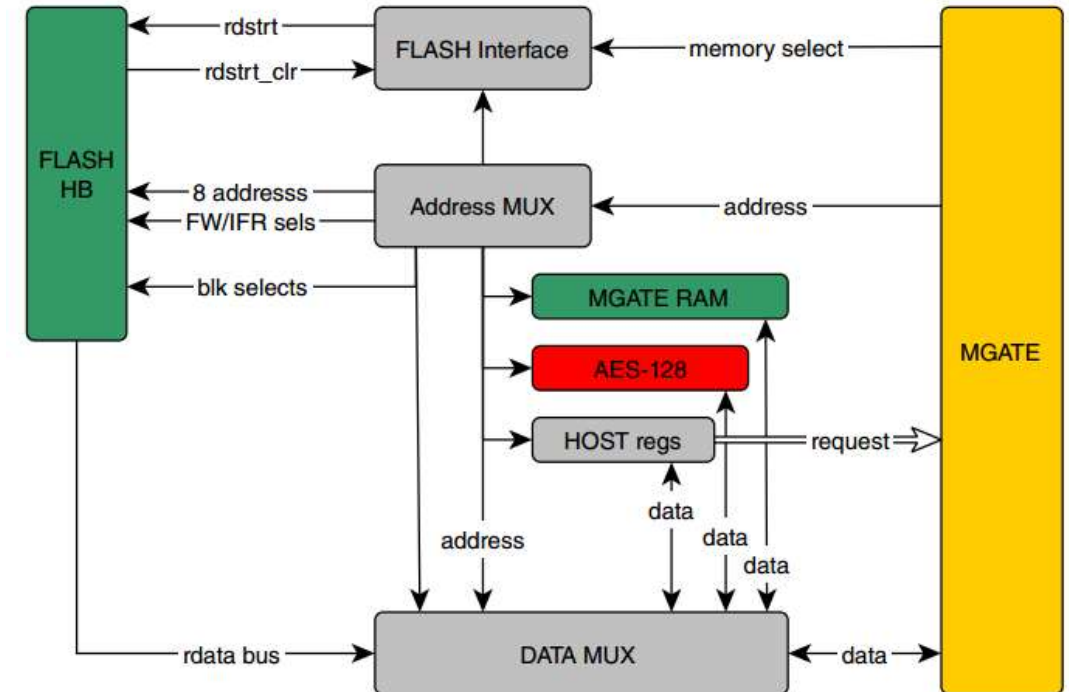


CSEc Security Block Diagram

Supports OEM Requirements for End Node Security

Supports >SHE functionality

- Secure key storage
- AES-128 encryption/decryption
- AES-128 Cypher-based Message Authentication Code (CMAC) calculation and authentication
- True and Pseudo random number generation
- User configurable Secure Boot Mode (Sequential, Strict, or Parallel Boot)



CSEc: CSE PRAM Interface Structure

The PRAM interface can be thought of a 128-bit wide SRAM with eight 128-bit pages

Figure 32-11. Generic PRAM interface

Bits	[127:0]															
Bits	31:2 4	23:1 7	15:8	7:0	31:2 4	23:1 7	15:8	7:0	31:2 4	23:1 7	15:8	7:0	31:2 4	23:1 7	15:8	7:0
WD	Word 0				Word 1				Word 2				Word 3			
	Byte															
Page	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Func D	Func Form at	CallS eq	KeyID	Error Bits		Command Specific									
1	Data Input to CSEc OR Data Output from CSEc															
2																
3																
4																
5																
6																
7																

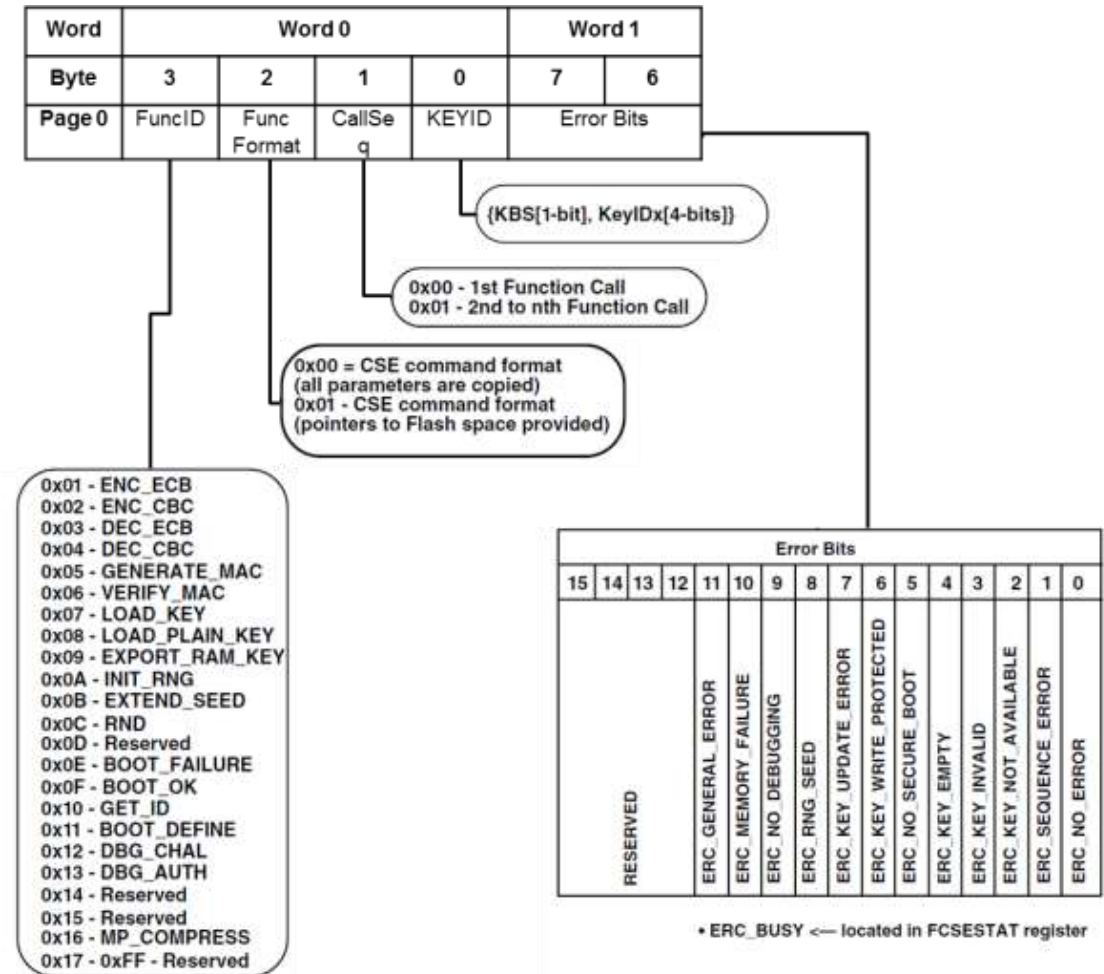
(2) Any associated control information such as 'MESSAGE_LENGTH'.

(3) The last write which is to the command header. This is because writing to the command header (any write to any of the bytes 0-3) triggers the macro to lock the PRAM interface so CSEc operation may start.

(1) User should first enter the data as required, in 128-bit blocks, as many blocks as desired (within the seven pages allowed at one given time).

S32K Security Module (CSEc) – Commands Header

- FuncID: CSEc ID to execute
- Func Format: specify data transfer to CSEc: parameters directly copied to PRAM or pointer method
- CallSeq: long data could be managed
- Key ID: SHE key index (KeyIdx) and key block selec (KBS)
- Error bits: Located in FCESTAT



CSEc PRAM – Command Header

Word	Word 0				Word 1	
Byte	3	2	1	0	7	6
Page 0	FuncID	Func Format	CallSeq	KEYID	Error Bits	

{KBS[1-bit], KeyIDx[4-bits]}

0x00 - 1st Function Call
0x01 - 2nd to nth Function Call

0x00 = CSE command format
(all parameters are copied)
0x01 = CSE command format
(pointers to Flash space provided)

- 0x01 - ENC_ECB
- 0x02 - ENC_CBC
- 0x03 - DEC_ECB
- 0x04 - DEC_CBC
- 0x05 - GENERATE_MAC
- 0x06 - VERIFY_MAC
- 0x07 - LOAD_KEY
- 0x08 - LOAD_PLAIN_KEY
- 0x09 - EXPORT_RAM_KEY
- 0x0A - INIT_RNG
- 0x0B - EXTEND_SEED
- 0x0C - RND
- 0x0D - Reserved
- 0x0E - BOOT_FAILURE
- 0x0F - BOOT_OK
- 0x10 - GET_ID
- 0x11 - BOOT_DEFINE
- 0x12 - DBG_CHAL
- 0x13 - DBG_AUTH
- 0x14 - Reserved
- 0x15 - Reserved
- 0x16 - MP_COMPRESS
- 0x17 - 0xFF - Reserved

Error Bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED				ERC_GENERAL_ERROR	ERC_MEMORY_FAILURE	ERC_NO_DEBUGGING	ERC_RNG_SEED	ERC_KEY_UPDATE_ERROR	ERC_KEY_WRITE_PROTECTED	ERC_NO_SECURE_BOOT	ERC_KEY_EMPTY	ERC_KEY_INVALID	ERC_KEY_NOT_AVAILABLE	ERC_SEQUENCE_ERROR	ERC_NO_ERROR

• ERC_BUSY ← located in FCSESTAT register

Directs to the Key to be used for this operation

Indicates continuation of same function with more data
Useful in case when data size can not fit into the CSEc PRAM.

Defines “how to access data?”
Pointer method only available for MAC commands
Pointer can only points to flash locations

User Accessible Functions/Commands/APIs

Error codes: updated after every command execution

CSEc: Activity Conflict With Flash/EEPROM Operations

- CSEc is shared with Flash controller. CSEc command is not accepted during Flash command is executed (CCIF=0)
 - Ex1) Program Flash is programmed/erased
 - Ex2) EEPROM is programmed.

NOTE from RM

1. It is not possible to concurrently execute CCOB commands related to Program, Erase (or other standard FTFC flash commands) along with CSEc commands.
2. Execution of a CSEc command while in Erase Suspend (ERSSUSP) will result in the Suspended Erase operation being aborted (not able to be resumed).
3. It is also not possible to execute a different CSEc command in the middle of a continuation of an ongoing CSEc command.
4. It is possible to execute a FCCOB command in the middle of a continuation of an ongoing CSEc command, BUT the result is the existing CSEc command will be canceled.
5. Starting execution of CCOB commands or CSEc commands will lock out the CCOB interface, the EEERAM and the PRAM. The lock is in place until the requested command completes.

CSEc: Example - CBC Encryption – CallSeq Usage

CallSeq = 0

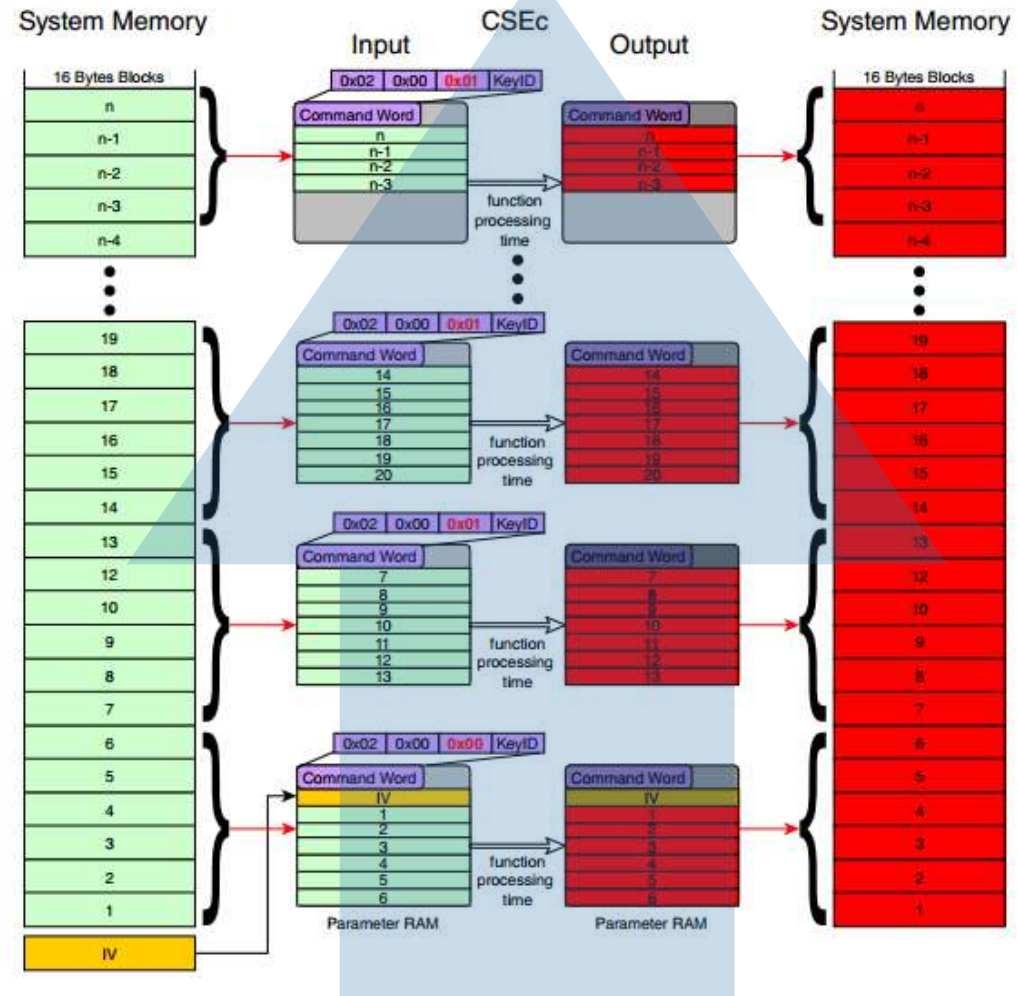
Input Parameters															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x00	KeyID	Clear Bits										
1	Plaintext 1 (0-15)														
2	Plaintext 2 (0-15)														
3	Plaintext 3 (0-15)														
4	Plaintext 4 (0-15)														
5	Plaintext 5 (0-15)														
6	Plaintext 6 (0-15)														
7	Plaintext 7 (0-15)														

Output Parameters															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x00	KeyID	Clear Bits										
1	Ciphertext 1 (0-15)														
2	Ciphertext 2 (0-15)														
3	Ciphertext 3 (0-15)														
4	Ciphertext 4 (0-15)														
5	Ciphertext 5 (0-15)														
6	Ciphertext 6 (0-15)														
7	Ciphertext 7 (0-15)														

CallSeq = 1

Input Parameters															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x01	KeyID	Clear Bits										
1	Plaintext 7 (0-15)														
2	Plaintext 8 (0-15)														
3	Plaintext 9 (0-15)														
4	Plaintext 10 (0-15)														
5	Plaintext 11 (0-15)														
6	Plaintext 12 (0-15)														
7	Plaintext 13 (0-15)														

Output Parameters															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x01	KeyID	Clear Bits										
1	Ciphertext 7 (0-15)														
2	Ciphertext 8 (0-15)														
3	Ciphertext 9 (0-15)														
4	Ciphertext 10 (0-15)														
5	Ciphertext 11 (0-15)														
6	Ciphertext 12 (0-15)														
7	Ciphertext 13 (0-15)														

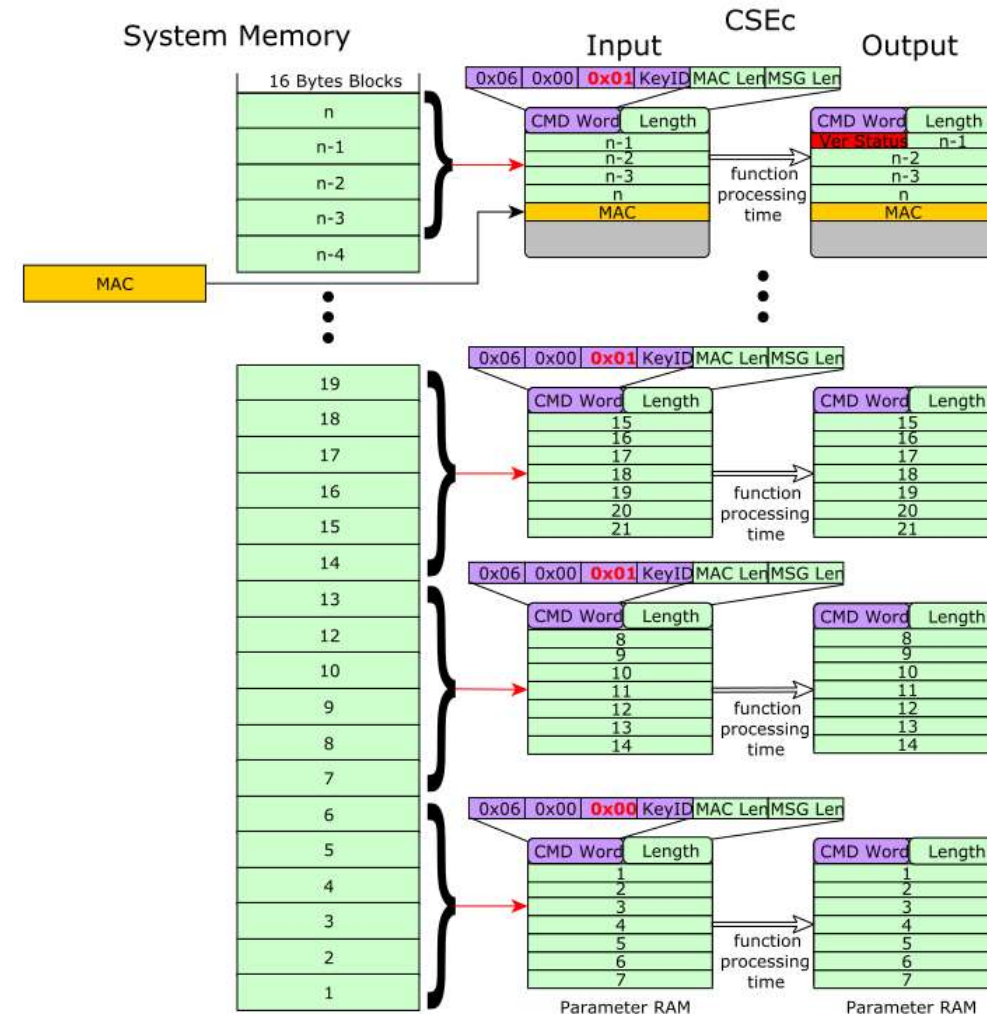


S32K Security Module (CSEc) – Keys

Key name	KBS	Key Index
SECRET_KEY	X	0x0
UID	X	0x0
MASTER_ECU_KEY	X	0x1
BOOT_MAC_KEY	X	0x2
BOOT_MAC	X	0x3
KEY 01 – KEY 10	0	0x4-0xA
KEY 11 – KEY 17	1	0x4-0xA (CSEC Ext.)
RAM_KEY	X	0xF

S32K Security Module (CSEc) – CMAC Verification

- The Verify MAC command verifies a MAC of a given MESSAGE
- Two options
 - Data Directly copied to PRAM
 - Pointer method
- Command Parameters
 - Key ID
 - Message Length
 - Message
 - MAC
 - MAC Length



S32K Security Module (CSEc) – CMAC Command

- Generate MAC command operates on a MESSAGE using a key
- Two options
 - Data Directly copied to PRAM
 - Pointer method
- Command Parameters
 - Key ID
 - Message Length
 - Message

Command Parameters

Parameter	Direction	Width
KEY_ID	IN	5
MESSAGE_LENGTH	IN	64
MESSAGE	IN	n * 128
MAC	OUT	128
MAC = CMACKEY, KEY_ID (MESSAGE, MESSAGE_LENGTH)		
Error Codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

Data Directly Copied to PRAM

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0x05	0x00	0x00	KeyID	Error Bits	Reserved								MESSAGE_LENGTH			
1	DATA 1 [0:15]																
2	DATA 2 [0:15]																
3	DATA 3 [0:15]																
4	DATA 4 [0:15]																
5	DATA 5 [0:15]																
6	DATA 6 [0:15]																
7	DATA 7 [0:15]																

Pointer Method

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x05	0x01	0x00	KeyID	Error Bits	Reserved							MESSAGE_LENGTH			
1	Flash Start Address				Reserved											
2	Reserved															
3																
4																
5																
6																
7																

S32K Security Module (CSEc) – Boot Define

- Allow user to define the Boot size
- User to select the boot mode
- Input Parameters
 - Boot size
 - Boot Flavor

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x11	0x00	0x00	KeyID	Error Bits	Reserved										
1	Reserved											Boot Flavor	BOOT_SIZE			

Table continues on the next page...

Lab #3 Encrypt Image



Encrypt Image Task

- Task

- Encrypt an image using the CSEc

- Learn

- How to use CSEc to encrypt data using Keys?

- DIY

- Use ENC_ECB command to encrypt the image
- Use ENC_CBC command to encrypt the image
- What do you expect to see?
- Use KEY_11 vs. KEY_1, what did you see?

- Note

- You will need to run S32K144_EVB_CSEc_Step3_EncryptLogo project for this Lab.

Lab Technical Details (Misc)

- EVB Configuration
 - SOSC w/ 8MHz crystal
 - $VCO_CLK = 8 \times 40 = 320\text{MHz}$, $SPLL_CLK = 320 / 2 = 160\text{MHz}$
 - $CORE_CLK / SYS_CLK = 160 / 2 = 80\text{MHz}$
 - $BUS_CLK = 80 / 2 = 40\text{MHz}$, $FLASH_CLK = 80 / 3 = 26.6\text{MHz}$
- The size of NXP logo is 80x200 pixels in RGB565 format (1pixel=2byte), thus 32000 bytes in total
- To maximize the throughput, the pre-encrypted NXP logo bitmap data is transferred from Flash to SRAM by DMA in the initialization routine

Lab Software Requirements

- Display plain text array of the NXP image on LCD (Done)
- Press SW2 to Encrypt plain text array (Need to Do)
- During encryption display encrypted cipher text array on the LCD (Done)
- The encrypted logo is gradually drawn from top to bottom on the LCD (Done)
- Elapsed times and encryption and display data rates are shown on LCD (Done)
- Press reset to start again.

Crypto Tasks

- ECB Encoding
- Command:
CMD_ENC_ECB

Input Parameter

	Byte															
Page	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x01	0x00	0x00	KeyID	Error Bits		Reserved								PAGE_LENGTH	
1	PLAIN_TEXT 1 [0:15]															
2	PLAIN_TEXT 2 [0:15]															
3	PLAIN_TEXT 3 [0:15]															
4	PLAIN_TEXT 4 [0:15]															
5	PLAIN_TEXT 5 [0:15]															
6	PLAIN_TEXT 6 [0:15]															
7	PLAIN_TEXT 7 [0:15]															

Output Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x01	0x00	0x00	KeyID	Error Bits		Reserved									
1	CIPHER_TEXT 1 [0:15]															
2	CIPHER_TEXT 2 [0:15]															
3	CIPHER_TEXT 3 [0:15]															
4	CIPHER_TEXT 4 [0:15]															
5	CIPHER_TEXT 5 [0:15]															
6	CIPHER_TEXT 6 [0:15]															
7	CIPHER_TEXT 7 [0:15]															

Crypto Tasks

- ECB Decoding
- Command:
CMD_DEC_ECB

Input Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x03	0x00	0x00	KeyID	Error Bits	Reserved										PAGE_LENGTH
1	CIPHER_TEXT 1 [0:15]															
2	CIPHER_TEXT 2 [0:15]															
3	CIPHER_TEXT 3 [0:15]															
4	CIPHER_TEXT 4 [0:15]															
5	CIPHER_TEXT 5 [0:15]															
6	CIPHER_TEXT 6 [0:15]															
7	CIPHER_TEXT 7 [0:15]															

Output Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x03	0x00	0x00	KeyID	Error Bits	Reserved										
1	PLAIN_TEXT 1 [0:15]															
2	PLAIN_TEXT 2 [0:15]															
3	PLAIN_TEXT 3 [0:15]															
4	PLAIN_TEXT 4 [0:15]															
5	PLAIN_TEXT 5 [0:15]															
6	PLAIN_TEXT 6 [0:15]															
7	PLAIN_TEXT 7 [0:15]															

Crypto Tasks

- CBC Encoding
- Command: CMD_ENC_CBC

Input Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x00	KeyID	Error Bits		Reserved								PAGE_LENGTH	
1	IV [0:15]															
2	PLAIN_TEXT 1 [0:15]															
3	PLAIN_TEXT 2 [0:15]															
4	PLAIN_TEXT 3 [0:15]															
5	PLAIN_TEXT 4 [0:15]															
6	PLAIN_TEXT 5 [0:15]															
7	PLAIN_TEXT 6 [0:15]															

Output Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x00	KeyID	Error Bits		Reserved									
1	IV [0:15]															
2	CIPHER_TEXT 1 [0:15]															
3	CIPHER_TEXT 2 [0:15]															
4	CIPHER_TEXT 3 [0:15]															
5	CIPHER_TEXT 4 [0:15]															
6	CIPHER_TEXT 5 [0:15]															
7	CIPHER_TEXT 6 [0:15]															

Crypto Tasks

- CBC Decoding
- Command: CMD_DEC_CBC

Input Parameter

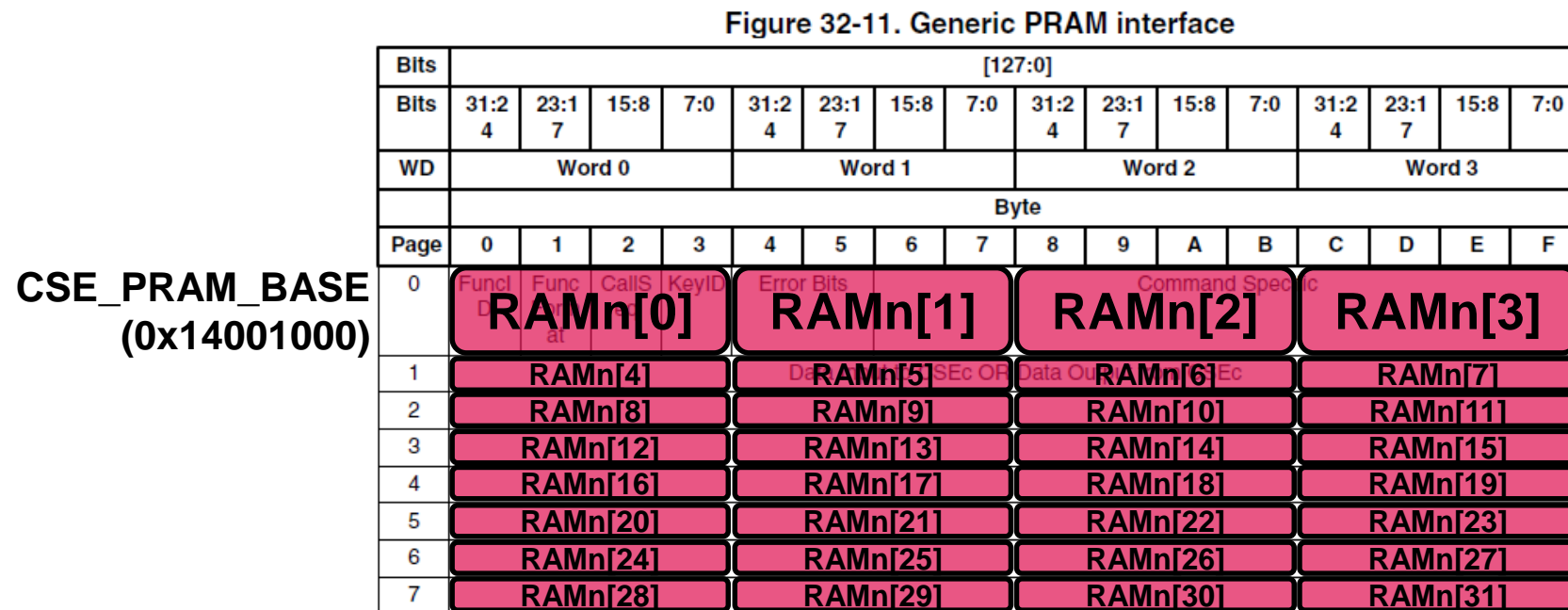
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x04	0x00	0x00	KeyID	Error Bits	Reserved										PAGE_LENGTH TH
1	IV [0:15]															
2	CIPHER_TEXT 1 [0:15]															
3	CIPHER_TEXT 2 [0:15]															
4	CIPHER_TEXT 3 [0:15]															
5	CIPHER_TEXT 4 [0:15]															
6	CIPHER_TEXT 5 [0:15]															
7	CIPHER_TEXT 6 [0:15]															

Output Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x04	0x00	0x00	KeyID	Error Bits	Reserved										
1	IV [0:15]															
2	PLAIN_TEXT 1 [0:15]															
3	PLAIN_TEXT 2 [0:15]															
4	PLAIN_TEXT 3 [0:15]															
5	PLAIN_TEXT 4 [0:15]															
6	PLAIN_TEXT 5 [0:15]															
7	PLAIN_TEXT 6 [0:15]															

CSE_PRAM Macro Definition in Header File

- Below text macro has been defined in “S32K144.h” header file
- The “RAMn[]” array is defined with “uint32_t”



Example: `CSE_PRAM->RAMn[i].DATA_32 = plain_text[j];`

Software Coding Example – ENC_CBC Case

```
/* Encode the data using CBC: Cipher Block Chaining Mode
 * For simplicity this function is developed for up to first 6 pages of data(96 bytes)
 */
uint16_t ENC_CBC(uint32_t *cipher_text, uint32_t *IV, uint8_t key_id, uint32_t *plain_text, uint16_t page_length)
{
    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != FTFC_FSTAT_CCIF_MASK); //Check for the ongoing FLASH command
    FTFC->FSTAT = 0x30; // Clear old errors
    for(i=4,j=0; i<8; i++,j++) //Write to Page1
    {
        CSE_PRAM->RAMn[i].DATA_32 = IV[j];
        for(i=8,j=0; i<(page_length*4+8); i++,j++) // Fill all other pages, word by word
        {
            CSE_PRAM->RAMn[i].DATA_32 = plain_text[j];
        }
        CSE_PRAM->RAMn[3].DATA_32= page_length; // Write to Page0 Word3, Value = Number of Pages
    }

    /* Start Command by writing Header */

    CSE_PRAM->RAMn[0].DATA_32=(CMD_ENC_CBC << 24) | (CMD_FORMAT_COPY << 16) | (CALL_SEQ_FIRST << 8) | key_id; // Write to Page0 Word0, Value = 0x02000000 | key_id
    while((FTFC->FSTAT & FTFC_FSTAT_CCIF_MASK) != 0x80); //Check for the ongoing FLASH command
    csec_error_bits = CSE_PRAM->RAMn[1].DATA_32 >> 16; //Write to Page0 Word1

    for(i=8,j=0; i<(page_length*4+8); i++,j++)
    {
        cipher_text[j] = CSE_PRAM->RAMn[i].DATA_32; //Read Page0 Word1, Error Bits
    }

    return csec_error_bits;
}
```

Figure 32-14. Encrypt CBC input parameter format

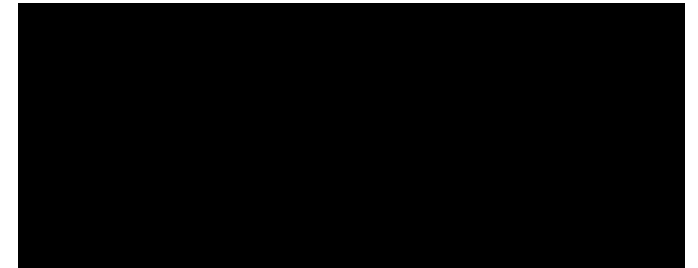
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0x02	0x00	0x00	KeyID	Error Bits		Reserved									PAGE_LENGTH	
1	IV [0:15]																
2	PLAIN_TEXT 1 [0:15]																
3	PLAIN_TEXT 2 [0:15]																
4	PLAIN_TEXT 3 [0:15]																
5	PLAIN_TEXT 4 [0:15]																
6	PLAIN_TEXT 5 [0:15]																
7	PLAIN_TEXT 6 [0:15]																

Figure 32-15. Encrypt CBC output parameter format

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x02	0x00	0x00	KeyID	Error Bits		Reserved									
1	IV [0:15]															
2	CIPHER_TEXT 1 [0:15]															
3	CIPHER_TEXT 2 [0:15]															
4	CIPHER_TEXT 3 [0:15]															
5	CIPHER_TEXT 4 [0:15]															
6	CIPHER_TEXT 5 [0:15]															
7	CIPHER_TEXT 6 [0:15]															

Display the Image

- Run S32K144_EVB_CSEc_Step3_EncryptLogo
- Press SW2
- Should see Blank image
- How much time did it take?

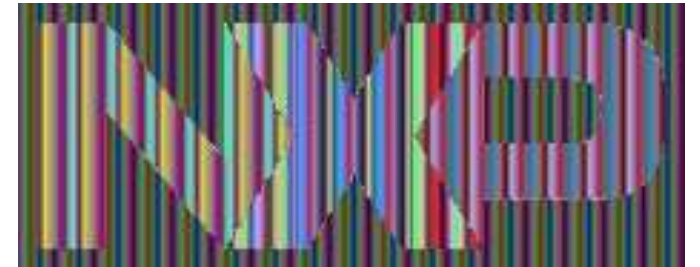


Time = 0.105s

CSEc Functions can be found in CSEc_functions.c

DIY: Encrypt the Image Using CBC Mode

- ECB Encoding Decoding using KEY_1
- Command: CMD_ENC_ECB
- Function: ENC_ECB
- Run project with your modification, press SW2
- How much time did it take?

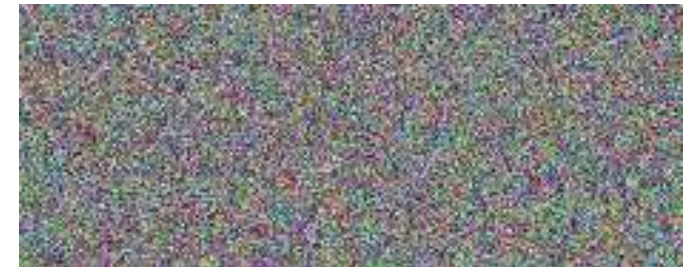


Time = 0.147s

CSEc Functions can be found in CSEc_functions.c

DIY: Encrypt the Image Using CBC Mode

- ECB Encoding Decoding using KEY_1
- Command: CMD_ENC_CBC
- Function: ENC_CBC
- Run project with your modification, press SW2
- How much time did it take?



Time = 0.153s

CSEc Functions can be found in CSEc_functions.c

DIY: Encrypt the Image Using CBC Mode

- ECB Encoding Decoding using **KEY_11**
- Command: `CMD_ENC_CBC`
- Function: `ENC_CBC`
- Run project with your modification, press SW2
- What did you see? Why?



Time = 0.147s

CSEc Functions can be found in `CSEc_functions.c`

Lab #4 Erase CSEC Keys



Erase CSEc Keys

- Task

- Erase all keys

- Learn

- Erase Keys based on process defined by SHE and Implement by the CSEc?

- Note

- You will need to run S32K144_EVB_CSEc_Step4_Erase_CSEc_Keys project for this Lab.

Erase CSEc Keys

- SHE describes a process to reset the secure flash to the state it was in when it left the factory which the CSEc has implemented.
- This can only be done if no user keys have been write protected.
- CMD_DBG_CHAL and CMD_DBG_AUTH FCCOB commands are used to erase the secure flash.
- What do you mean by secure flash back to factory?
 - The device does not have user keys (MASTER_ECU_KEY, BOOT_MAC, BOOT_MAC_KEY, KEY1..KEY17 are all erased)

Erase CSEc Keys

It is a 2 step process

1. Issue CMD_DBG_CHAL command request a random number (let say CHALLENGE – 128-bits)
 2. Issue CMD_DBG_AUTH command to return the authorization parameter to CSEc (let say AUTHORIZATION – 128bits)
- AUTHORIZATION value can be generated using CHALLENGE and MASTER_ECU_KEY
 - $K = \text{KDF}(\text{KEYMASTER_ECU_KEY}, \text{DEBUG_KEY_C})$
 - $\text{AUTHORIZATION} = \text{CMAC}_K(\text{CHALLENGE} \parallel \text{UID})$

Erase CSEc Keys Challenge/Authorization

- Issue Challenge
- Command: CMD_DBG_CHAL

Input Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x12	0x00	0x00	KeyID	Error Bits	Reserved										
1	Reserved															
2																
3																
4																
5																
6																
7																

Output Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x12	0x00	0x00	KeyID	Error Bits	Reserved										
1	CHALLENGE [0:15]															
2	Reserved															
3																
4																
5																
6																
7																

Erase CSEc Keys Challenge/Authorization

- Return authorization
- Command: CMD_DBG_AUTH

$K = \text{KDF}(\text{KEYMASTER_ECU_KEY}, \text{DEBUG_KEY_C})$
 $\text{AUTHORIZATION} = \text{CMAC}_K(\text{CHALLENGE} \parallel \text{UID})$

Input Parameter

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x13	0x00	0x00	KeyID	Error Bits		Reserved									
1	AUTHORIZATION [0:15]															
2	Reserved															
3																
4																
5																
6																
7																

LAB #5 Disable CSEC



Disable CSEc (Reset Device to Factory State)

- Task

- Disable CSEc and reset device to factory state

- Learn

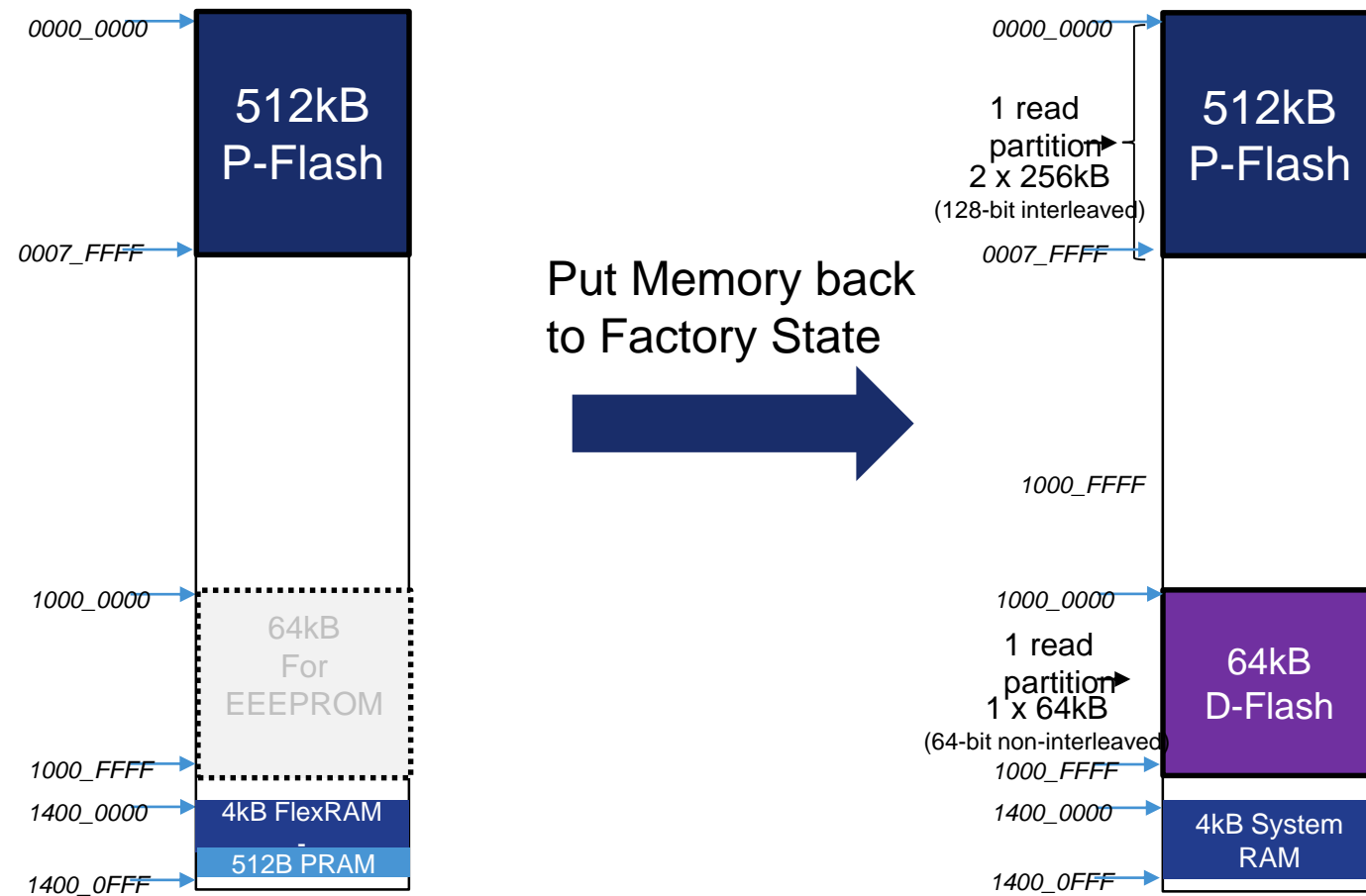
- How to put back the device so that FlexRAM is no longer used for EEPROM (factory state) which will also disable the CSEc.

- Note

- You will need to run S32K144_EVB_CSEc_Step5_Disable_CSEc project for this Lab.

Erase All Flash

To disable the CSEc the Data Flash 0 IFR must be erased which was written to during step 1 to configure the CSEc and use the FlexRAM with Data Flash as EEPROM and secure Flash.



Erase All Flash Block Command

- Erase All Flash Blocks
- Command: Erase All Blocks

Input Parameter

FCCOB Number	FCCOB Contents [7:0]
0	0x44 (ERSALL)

- Power Cycle Device
- Like you did after Lab 1 Check for:

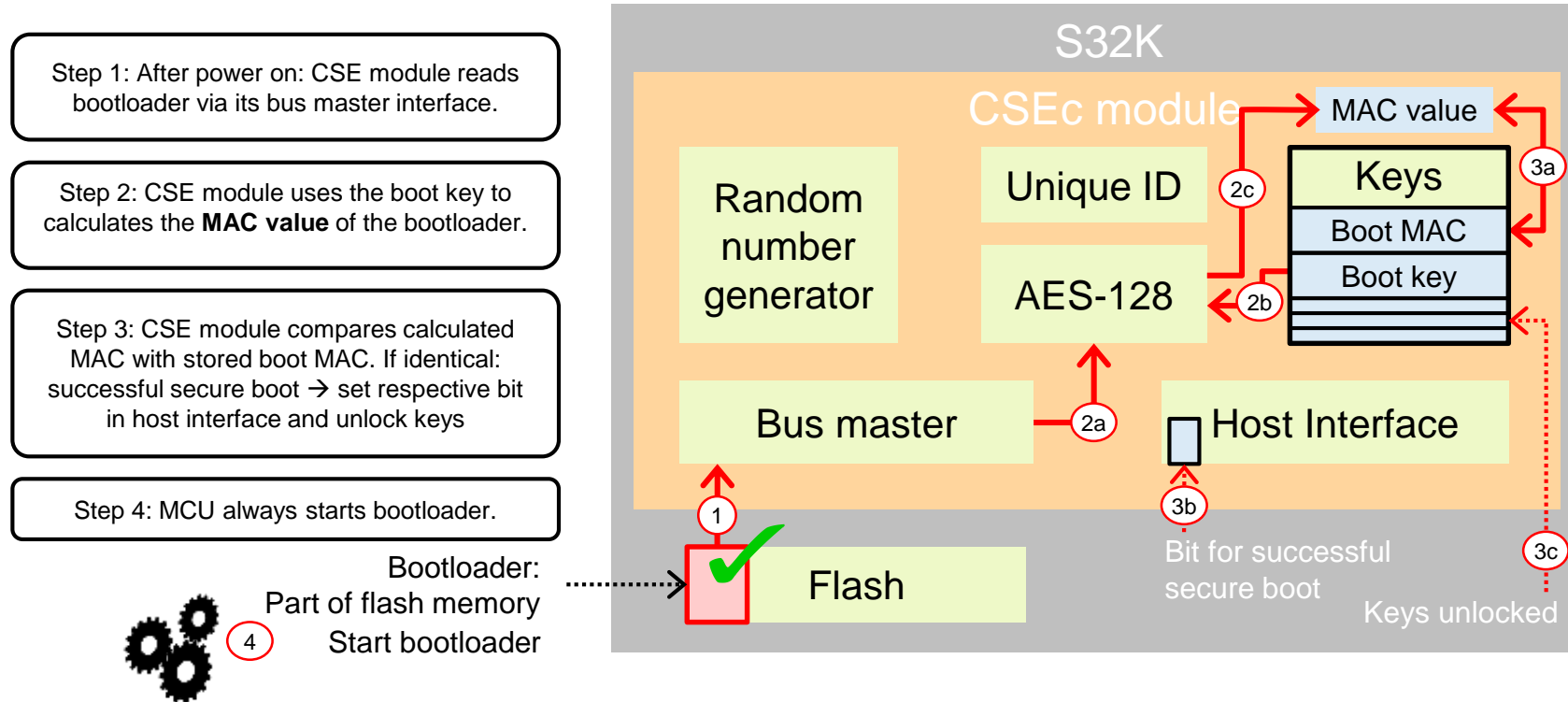
FCNFG[RAMRDY] == 1 & FCNFG[EEERDY] == 0

- i.e. emulated-EEPROM is not available now and we are back to the factory state of the MCU

Use Cases

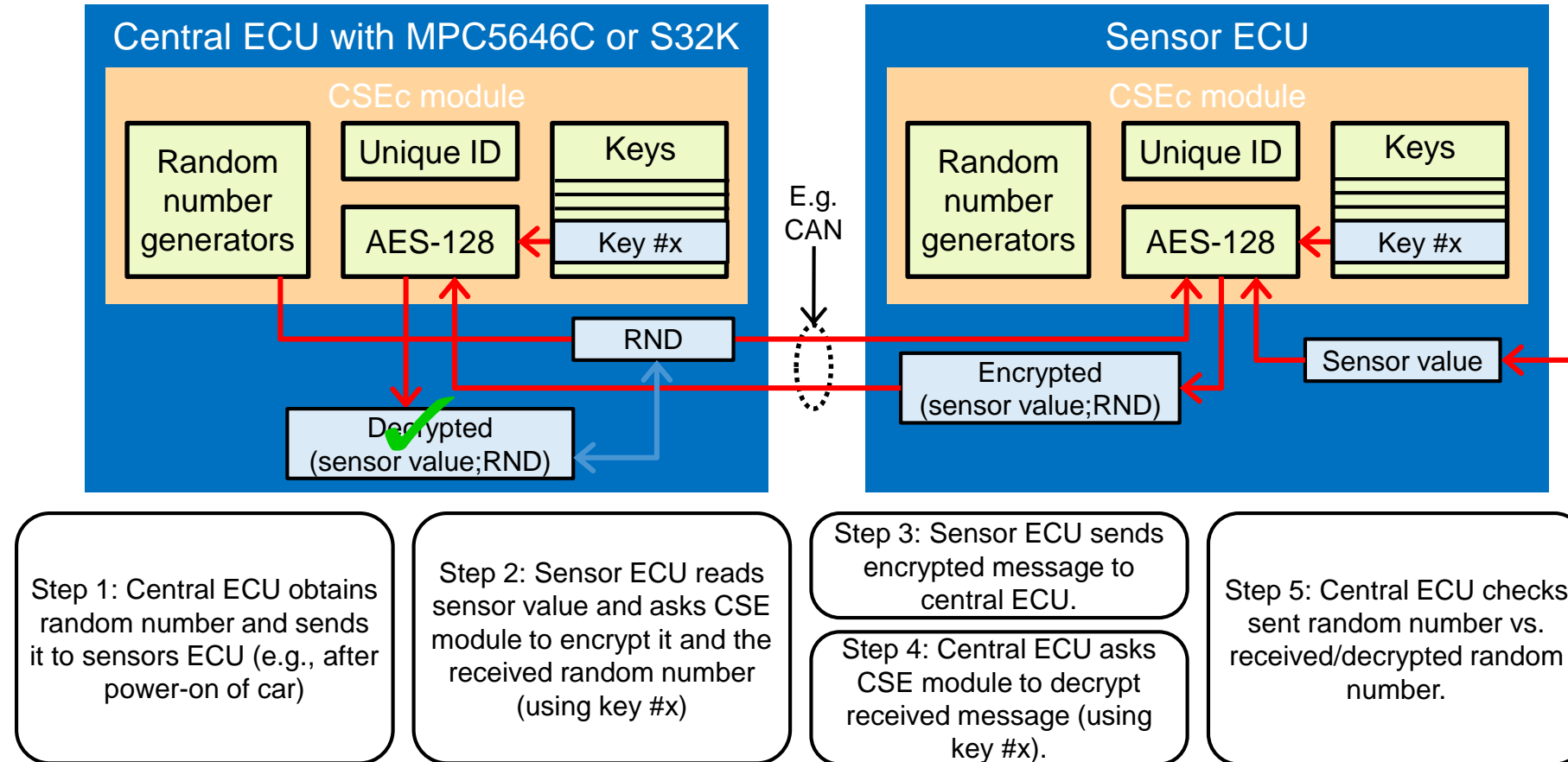


Secure Boot – Check Boot Loader for Integrity and Authenticity



- **MAC** protects against modification of bootloader and depends on the (secret) boot key → integrity and authenticity of bootloader.
- Only if calculated MAC value matches stored boot MAC value: successful secure boot → set respective bit in host interface and unlock keys for further usage (see next demos)

Secure Communication

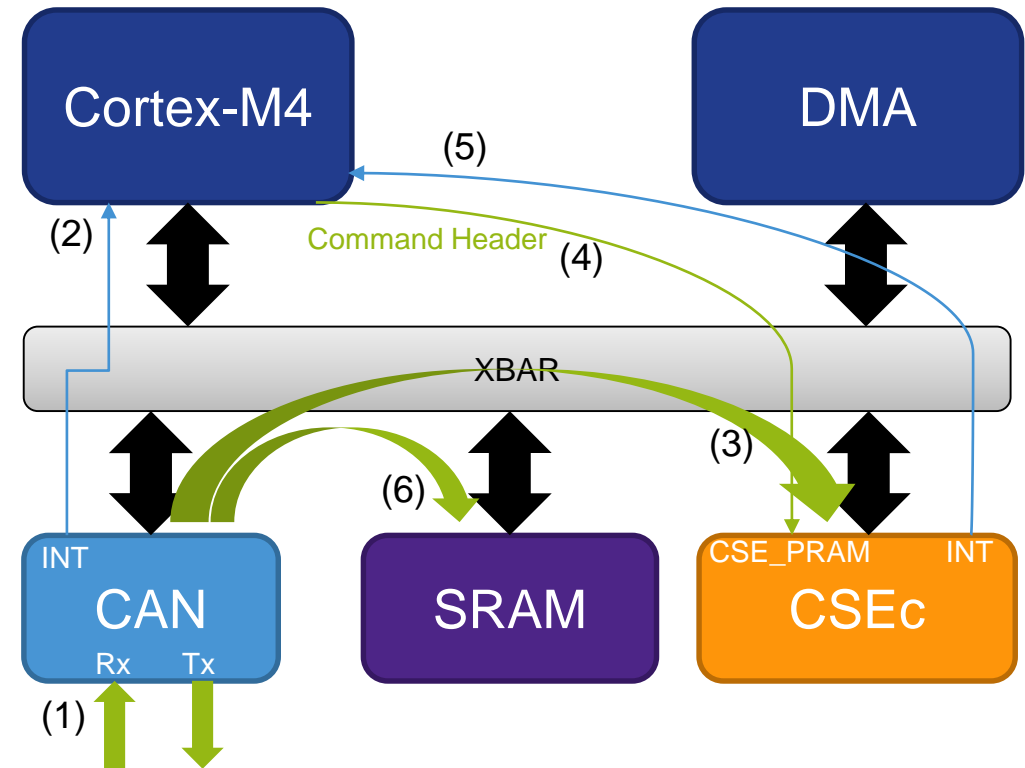


- Random number: protects against replay attacks.
- Encryption: protects against eavesdropping.
- Random number and encryption: ensures data integrity and authenticity.

CSEc: Module Interaction & Data-Flow

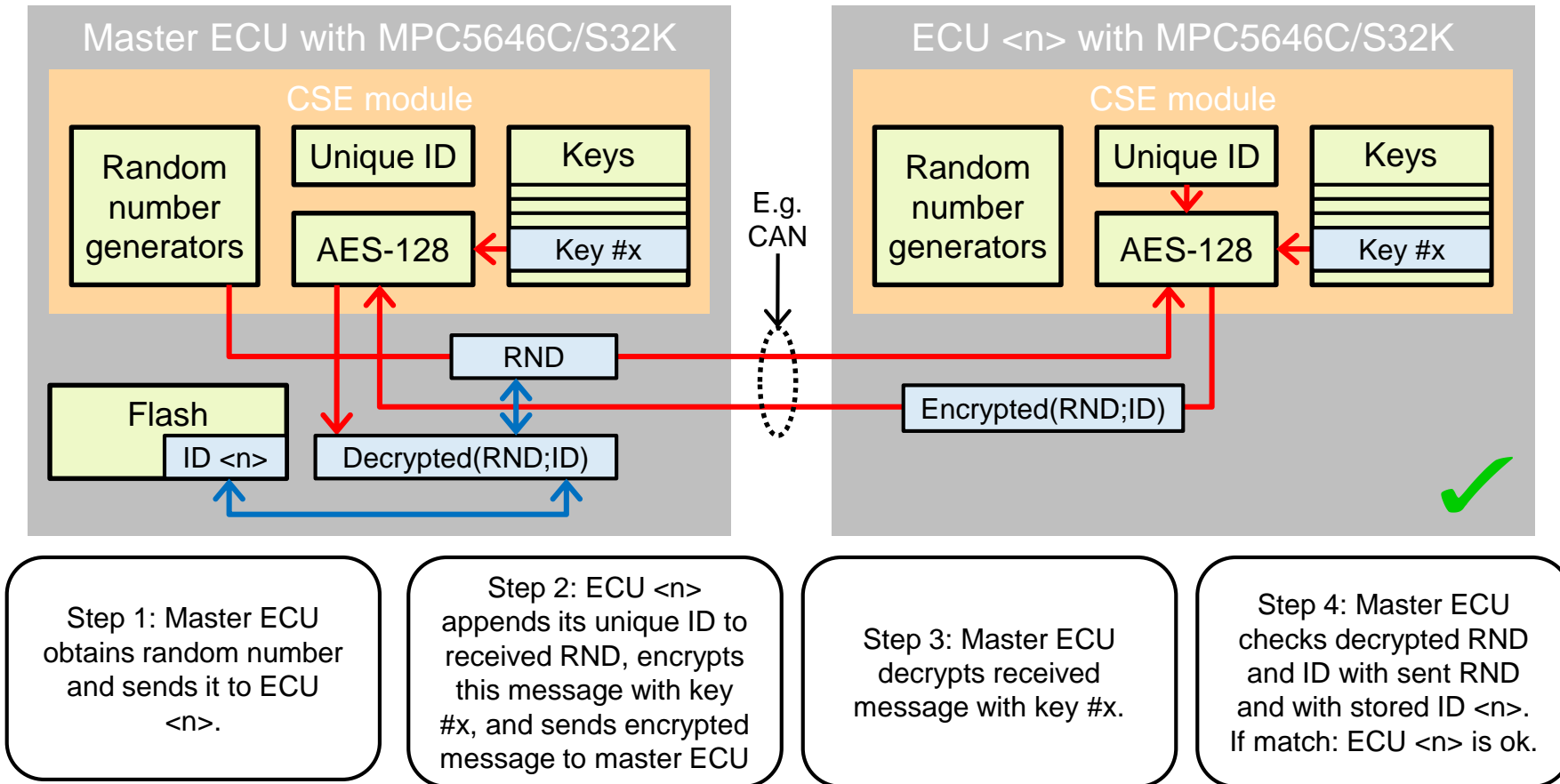
Scenario: CAN Rx Message Authentication

1. CAN data stored in local buffer
2. FlexCAN triggers interrupt to CORE/DMA
3. Transfer Data to CSEc Memory (max. 12 CAN message á 8 Bytes + 16 Byte CMAC)
4. Trigger CSEc CMAC calculation/verification
5. CSEc triggers interrupt to Core
6. Core processed message data



Component Protection

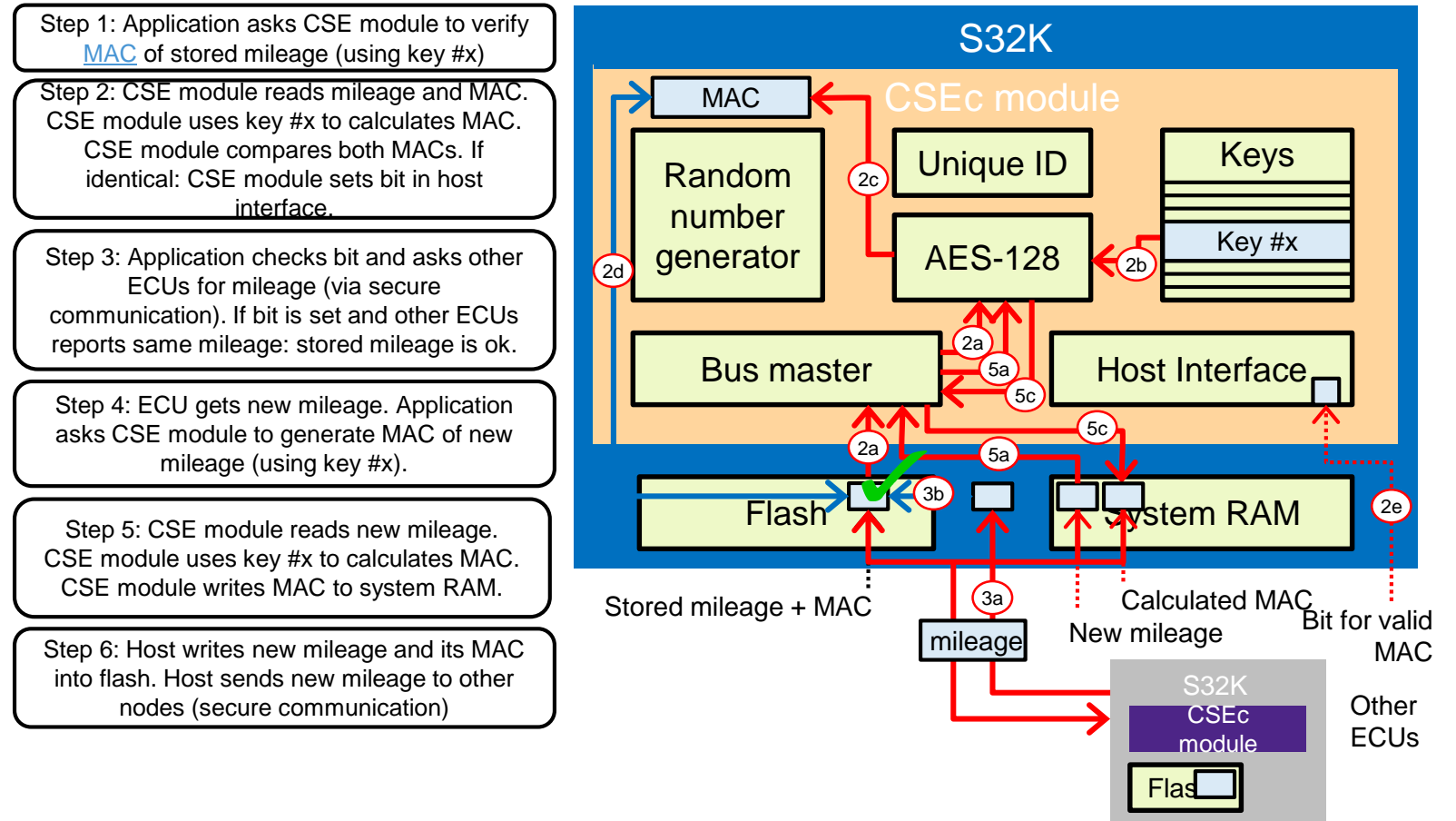
(Detect replacement or Modification of Components)



Replacement or modification of ECU <n> will change its unique ID and/or keys. Both will be detected with this proposal for component protection.

Mileage Protection – Protect Mileage Meter Against Modification

- MAC protects mileage against modification.
- Distributing mileage on other ECUs protects against replay-attacks (i.e., overwriting mileage and MAC with read old mileage and its MAC).



Summary



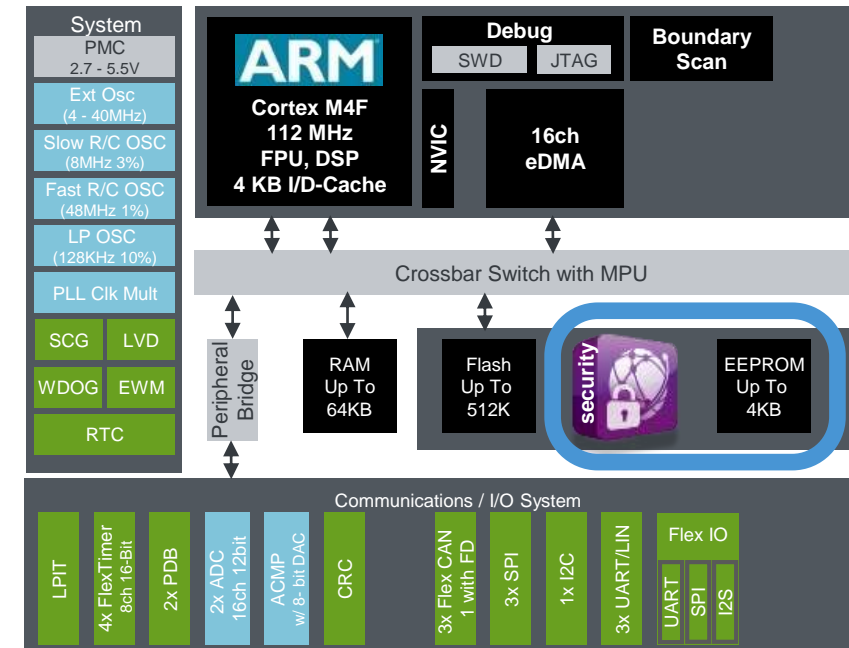
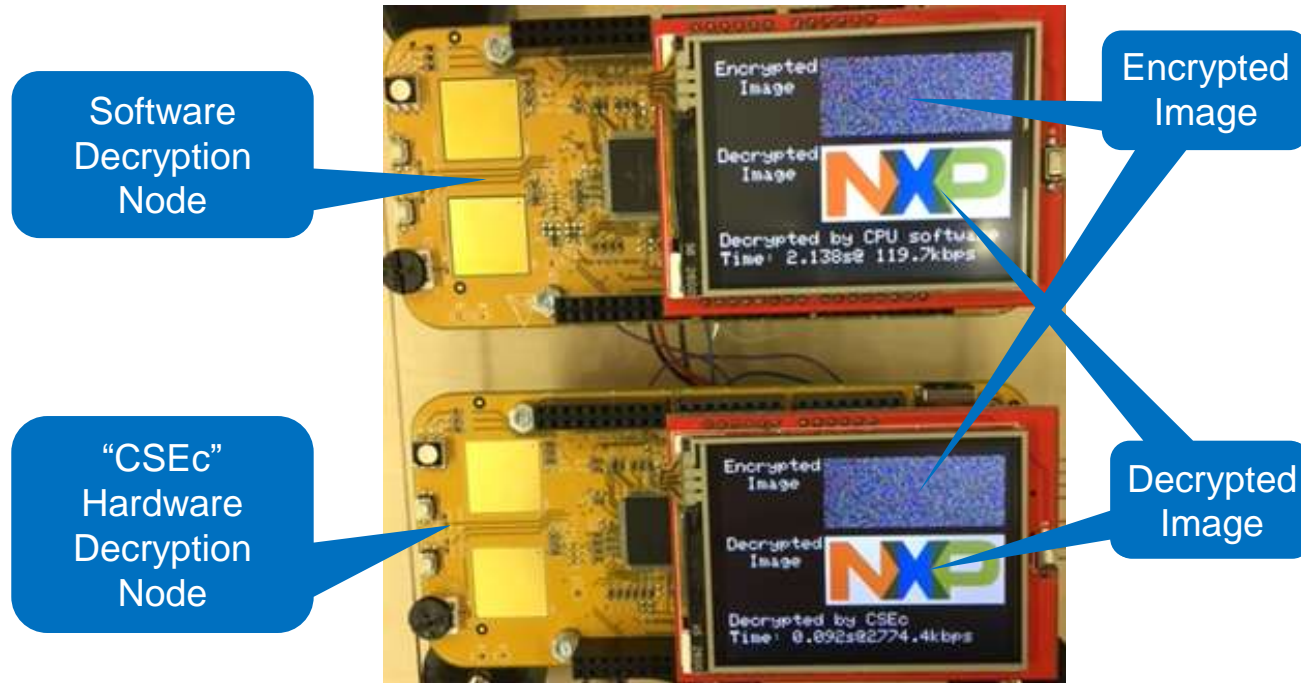
Summary – CSEc Workshop

- Demonstrated how to enable the CSEc through the Configuration/Partitioning of the FlexRAM and Data Flash EEPROM.
- How to create and store Secret keys using the CSEc interface PRAM.
- In the Do It Yourself portion you encrypted the NXP logo in both EBC and CBC mode using the CSEc AES-128 hardware and secret key.
- Demonstrated how to clear the secret keys (as long as they are not locked).
- Showed how to restore the device back to factory state.
- Additionally the workshop demonstrated how much time the encryption takes in both EBC and CBC mode.

Summary – CSEc in Your Application

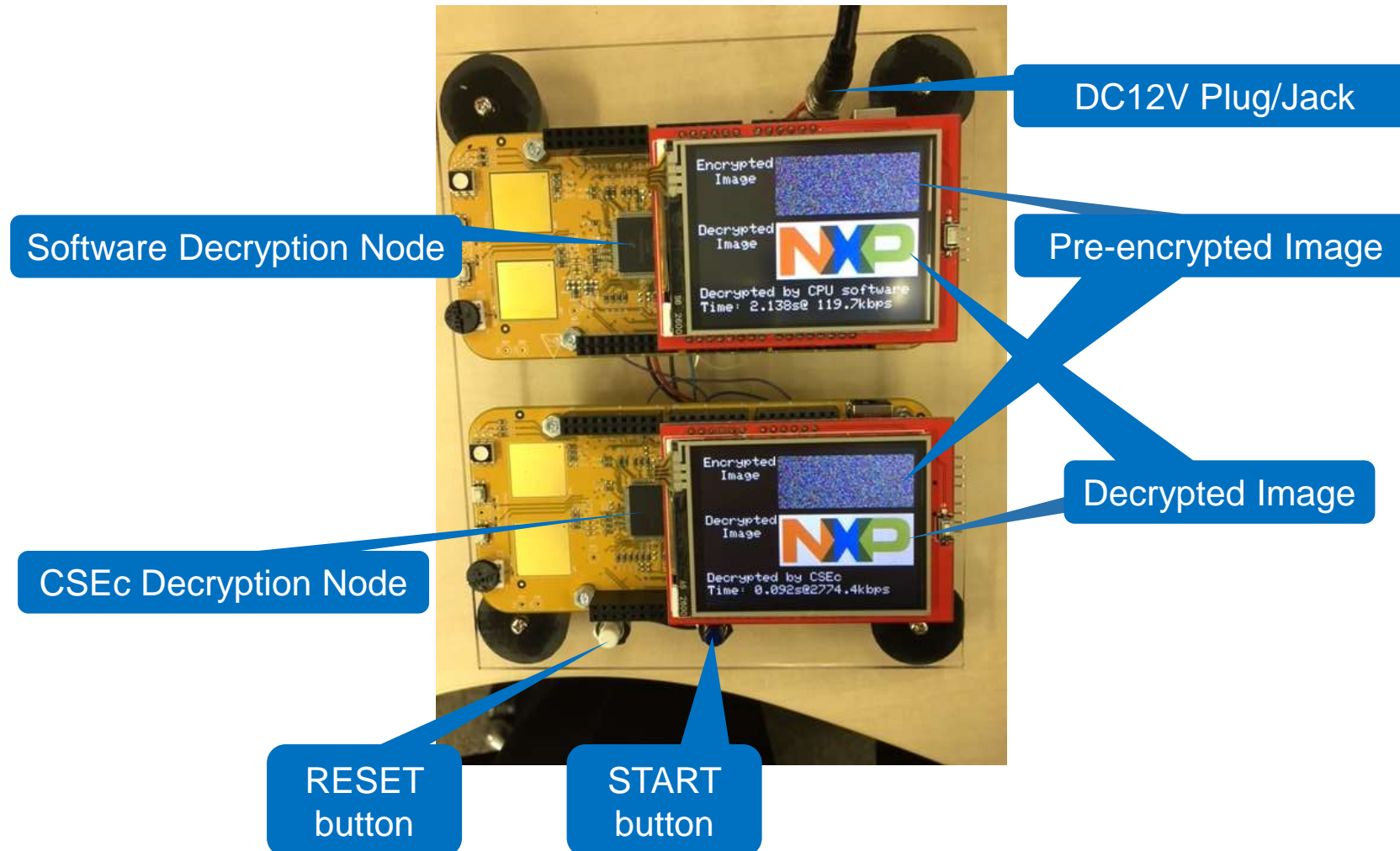
- CSEc can help you to encrypt your data and generate CMAC values to verify data. The CSEc is compliant with “SHE” security standard which means 1) comes with dedicated AES-128 hardware for much faster security processing, and 2) keys used for the encryption / decryption / authentication are stored in the special EEPROM storage which CPU can't access.
- Secure communication is and Firmware authentication is possible with S32K. All of S32K1xx family products from 128KB to 2MB flash products equip the “Cryptography-Service-Engine-compressed” (or CSEc) for secured communication (message encryption and/or authentication), and firmware code authentication.
- Future Proof – Combining with CAN FD feature, S32K MCU family is the best choice to achieve more secure automotive applications from small body control nodes to in-vehicle network gateway.

Summary – S32K144 CSEc Image Decryption Demo



- Fast execution of encryption/decryption/authentication by hardware
- Complaint with "SHE" standard specification with secured key storage
- Available in all S32K1xx products from 128KB to 2MB flash memory

Summary – CSEc Image Decryption Demo



AUTOSAR Cryptographic Service Engine Driver (CSEc)

Complex Device Driver for Autosar

Implementing AUTOSAR 4.0 conventions
(AUTOSAR 4.2 support in progress)

Driver configuration using Autosar tooling

Supporting S32K142, S32K144, S32K148
Support for S32K146 in progress

Easy to integrate into Autosar Crypto stacks (4.0/4.2)

Main services:

Secure cryptographic key storage

AES-128 encryption and decryption (ECB and CBC)

AES-128 CMAC calculation and verification

True and pseudo random number generation

Miyaguchi-Preneel compression function

Secure boot mode (user configurable)

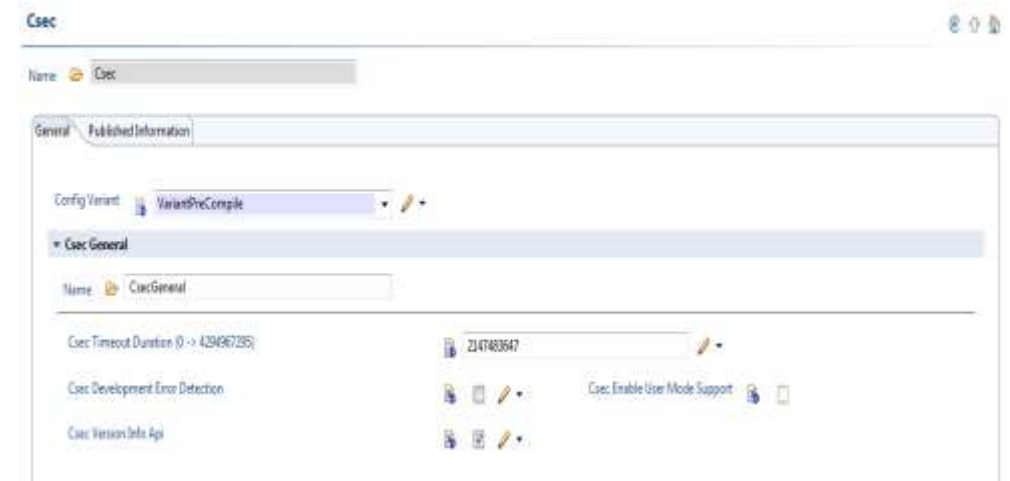
Implementing Synchronous access to CSEc HW

Driver API abstracting CSEc HW commands

Validated with multiple compilers

Fully compatible with NXP Autosar MCAL product

Compatible with running in parallel with EEPROM or Flash
drivers (sharing HW peripherals)



Configurable parameters in Elektrobit Tresos

AUTOSAR CSEc Sample Application

- Part of CSEc driver installer
- Demonstrates usage of CSEc driver in parallel with synchronous EEP and FLS drivers
- After each FLS, EEP and CSEC operation, messages are printed over UART
- Self-contained application delivered with a built system (makefiles) that compiles and delivers the elf file to be programmed with a debugger/programmer.

Initializes CSEc HW (S32K14x partitioned for emulated EEPROM, 512 bytes subtracted from EERAM space for keys)

Loads master key and user key 2

Encrypts a 16-byte buffer using AES – 128 ECB protocol

Reads the UID of the chip

Erases the stored keys (master key and user key 2)

```
MCAL SAMPLE APPLICATION: OS not present
CSEC: Master key updated successfully
EEP write accepted
CSEC: User key updated successfully
EEP write finished successfully
FEE init finished successfully
CSEC: Encryption successful
EEP read accepted
FEE erase immediate successful
CSEC: Get ID executed successfully
```

Hyperterminal log

S32K142
S32K144
S32K148

Used Compilers:

GHS: 2015.1.4
IAR: V7.50.3
GCC: 4.9.3 20150529



SECURE CONNECTIONS
FOR A SMARTER WORLD

www.nxp.com

NXP, the NXP logo, and NXP secure connections for a smarter world are trademarks of NXP B.V. All other product or service names are the property of their respective owners. © 2018 NXP B.V.