NXP Semiconductors
Application Notes

Document Number:AN13388
Rev. 0, 11/2021

S32K3 Memories Guide

1. Introduction

The purpose of this application note is to provide a guideline to the readers about the memory features included in the S32K3 Product Family. This document details the available functions and best practices for running applications considering performance improvements.

You can find four kinds of memories inside S32K3 Product Family, the Flash memory, the SRAM, the Tightly Coupled Memory (TCM) and the Cache Memory. The S32K3 Product Family also have some modules with dedicated memory such like EMAC and CAN. This document will mainly focus on Flash Memory, TCM and SRAM.

The Flash memory is dedicated for program code and store data. Also, all devices in the family has a UTEST sector of 8 KB for store important configurations or to reserve information for the application. The S32K3 Product Family has devices from 512 KB to 8 MB of Flash program memory.

The RAM is integrated by the SRAM memory and the TCM. Part of the SRAM memory is available in standby mode. This means that the content of this memory are retained after setting the MCU in standby mode. The S32K3 product family leverages the TCM feature of ARM Cortex M7 architecture, whose main purpose is to provide a deterministic access time to the cores to some important data avoiding any delay in the access. This feature can be exploited in Real Time Operating Systems.

Contents

The Cache memory is a dedicated memory for the cores. This memory is not part of the system memory and doesn't have a physical address available for the programmer. This memory serves as an intermediate buffer between the processor and the main memory to reduce memory access time for the cores.

2. Features

S32K3 family devices memory features can be found in the *Table 1* and *2*.

Table 1. **S32K3 Memory features**

Feature	S32K310	S32K311	S32K341	S32K312	S32K322	S32K342
Core qty	1 x Cortex-M7	1 x Cortex-M7	1 x Cortex-M7 LS	1 x Cortex-M7	2 x Cortex-M7	1 x Cortex-M7 LS
Program flash memory (MB)	512 KB	1		2		
Data flash memory (KB)[1]	64		128			
Cache	I Cache 8 KB D Cache 8 KB					
Total RAM (KB)	128 KB (including 96 KB TCM)		256 KB (including 192 KB TCM)	192 KB (including 96 KB TCM)	256 KB (including 192 KB TCM)	
Standby RAM[2]	32 KB					

Table 2. **S32K3 Memory features**

Feature	S32K314	S32K324	S32K344	S32K328	S32K348	S32K338	S32K358
Core qty	1 x Cortex-M7	2 x Cortex-M7	1 x Cortex-M7 LS	2 x Cortex-M7	1 x Cortex-M7 LS	3 x Cortex-M7	1 x Cortex-M7 LS + 1 x Cortex-M7
Program flash memory (MB)	4			8			
Data flash memory (KB)[1]	128						
Cache	I Cache 8 KB D Cache 8 KB			TBD			
Total RAM (KB)	512 KB (including 96 KB TCM)	512 KB (including 192 KB TCM)		1152 KB (including 192 KB TCM)		1152 KB (including 384 KB TCM)	
Standby RAM[2]	32 KB			64 KB			

1. This represents the maximum available Data Flash memory. Refer to the S32K3 Reference Manual for limitations applying when HSE security firmware is installed
2. The Standby RAM is also included in the Total RAM

An important feature to remark is that all memories inside the S32K3 Product Family has Error Detection and Error Correction Code.

3. Flash memory

The flash memory on S32K3 devices is integrated by blocks. There are five blocks as maximum and two blocks as minimum. Detailed information is provided in the following table.

Table 3. **Flash memory architecture for S32K3**

Flash Blocks	S32K310	S32K311 S32K341	S32K312 S32K322 S32K342	S32K314 S32K324 S32K344	S32K328 S32K338 S32K348 S32K358
End Address UTEST Start Address	0x1B00_1FFF 8 KB 0x1B00_0000	0x1B00_1FFF 8 KB 0x1B00_0000	0x1B00_1FFF 8 KB 0x1B00_0000	0x1B00_1FFF 8 KB 0x1B00_0000	0x1B00_1FFF 8 KB 0x1B00_0000
End Address Block4 Data Flash Memory Start Address	0x1000_FFFF 64 KB 0x1000_0000	0x1000_FFFF 64 KB 0x1000_0000	0x1001_FFFF 128 KB 0x1000_0000	0x1001_FFFF 128 KB 0x1000_0000	0x1001_FFFF 128 KB 0x1000_0000
End Address Block3 Code Flash Memory 3 Start Address	Not Available	Not Available	Not Available	0x007F_FFFF 1 MB 0x0070_0000	0x00BF_FFFF 2 MB 0x00A0_0000
End Address Block2 Code Flash Memory 2 Start Address	Not Available	Not Available	Not Available	0x006F_FFFF 1 MB 0x0060_0000	0x009F_FFFF 2 MB 0x0080_0000
End Address Block1 Code Flash Memory 1 Start Address	Not Available	0x004F_FFFF 512 KB 0x0048_0000	0x005F_FFFF 1 MB 0x0050_0000	0x005F_FFFF 1 MB 0x0050_0000	0x007F_FFFF 2 MB 0x0060_0000
End Address Block0 Code Flash Memory 0 Start Address	0x0047_FFFF 512 KB 0x0040_0000	0x0047_FFFF 512 KB 0x0040_0000	0x004F_FFFF 1 MB 0x0040_0000	0x004F_FFFF 1 MB 0x0040_0000	0x005F_FFFF 2 MB 0x0040_0000

In the S32K3 Product Family the devices are available from 512 KB to 8 MB of Flash memory. The *Table 3* classifies the S32K3 Product Family devices by Flash memory size.

There are some regions inside the Flash memory that are protected to be used by the application cores. These are only available for HSE_B core. For more information about HSE_B refer to S32K3 Reference Manual.

There are three operations modes for the Flash memory. When the device is working in User mode the Flash memory array is accessible to execute a read, program or erase operation. The User mode is the default operating mode of the Flash memory. All the registers have read and write access. In low power mode the Flash memory is not accessible because its power source is turned off, so operations are not allowed in this mode. Finally the Utest mode is a test mode where the integrity of the Flash memory can be verified.

The Flash memory can perform multiple reads between different blocks by a single, dual or quad read feature, where in a multi-core scenario, if there are multiple threads running in parallel (on different sections/blocks of memory) those threads can occur simultaneously by a dual or quad read, this feature is controlled internally and not by the user. It also has the “Read-While-Write” (RWW) feature to be able to perform a read and a write simultaneously (applies only when operations are in different blocks); for example, in the S32K324, if the Core 0 application is performing a write process in block 0, then the Core 1 at the same time can read a data stored in the data flash block.

There are four important operations that we need to consider when we are working with the Flash memory:

- Read Flash memory
- Lock and Unlock sector or super sector
- Program Flash memory
- Erase Flash memory

3.1. Read

After reset, the Flash memory is in a default state which have the arrays and register available to be read by the controller. A read operation from Flash memory return a 256 bits of data length and register reads return 32 bits. For this operation is not necessary to consider a Lock or Unlock sector. The read operation is performed by the PFlash controller which is the interface between the system bus and the embedded Flash memory.

3.2. Write or Program

The minimum program size is 2 words (64 bits) and data must be 64 bit aligned. A maximum of 4 pages can be programmed at the same time, where 1 page are 8 words (256 bits). This mean that up to 1024 bits can be altered in a single program operation. When a program operation or write operation is made the ECC bits are calculated and stored. The ECC is handled on 64 bits doubleword. Eight bits of ECC are needed.

A program operation changes the logic value of a bit from 1 to 0, this means that a program operation from 0 to 1 is not allowed and the Flash memory needs to be erased before any program operation. When data Flash is used for EEPROM emulation, approved drivers by NXP can do an over-programming in a 64 bit ECC segment, this allows to over-program the same location up to 3 times without performing an erase operation in the sector. This feature can be used to change the record status of a data record without a previous erase which is frequently used in some EEPROM emulation techniques. It’s important to remark that it is only available and usable for approved drivers by NXP, please consult the RTD software for S32K3 devices for available FEE drivers.

Before a program operations occurs the sector that contains the specified address must be unlocked. If a locked sector or super sector is attempted to be programmed, program operation will fail and an error will be reported in the MCRS[PEP] bit.

The flow diagram for the program operation is explained in the *Figure 1*.

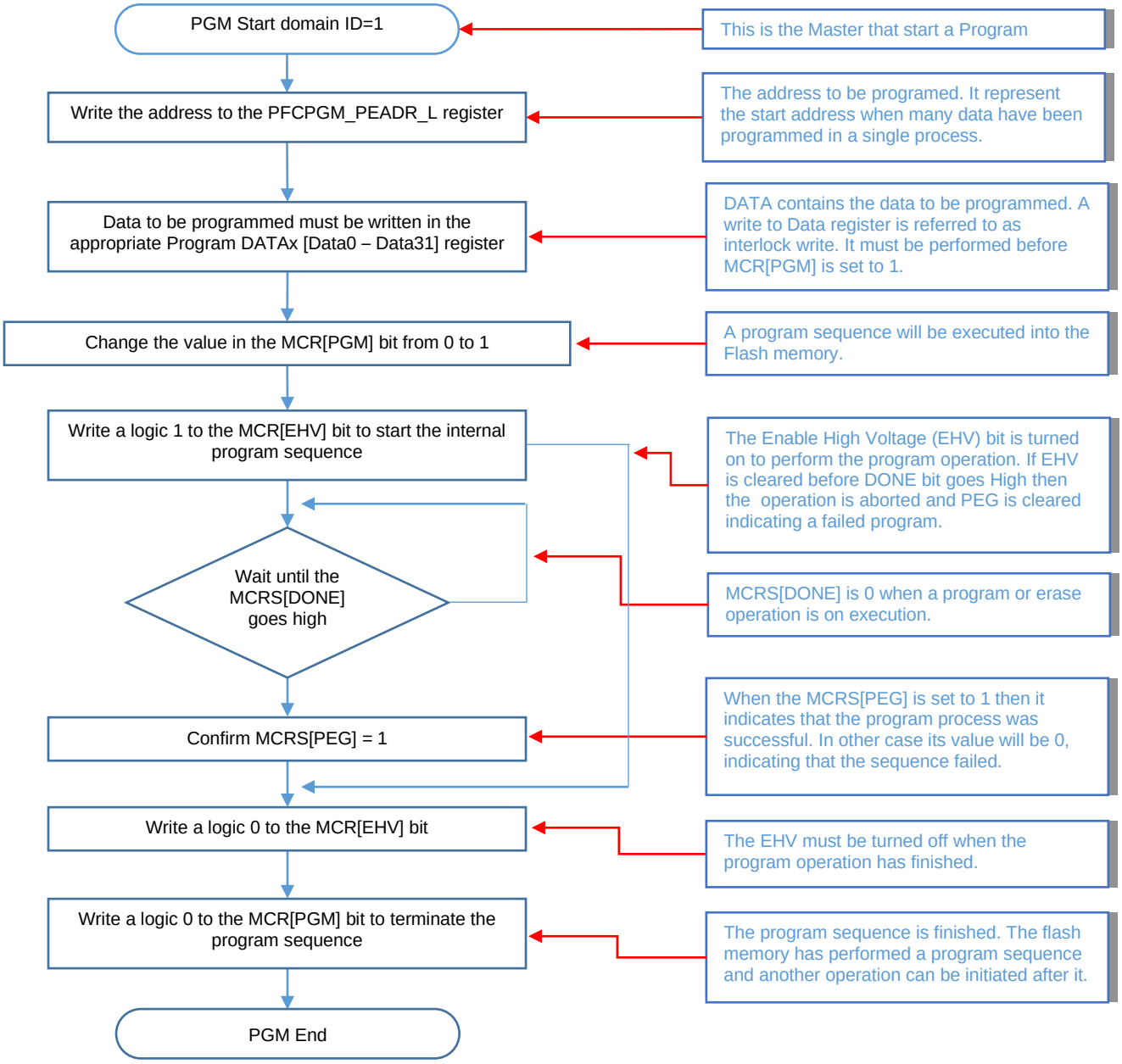

Figure 1. **Program sequence flow diagram**

3.3. Erase

The erase operation is the process to set all bits from a sector or block to 1. The minimum erase size can be performed in a sector, where a sector size is 8 KB. To erase a sector or block it must be unlocked previously to the erase operation. The erase process also clean the ECC bits.

The following flow diagram show the erase sequence.

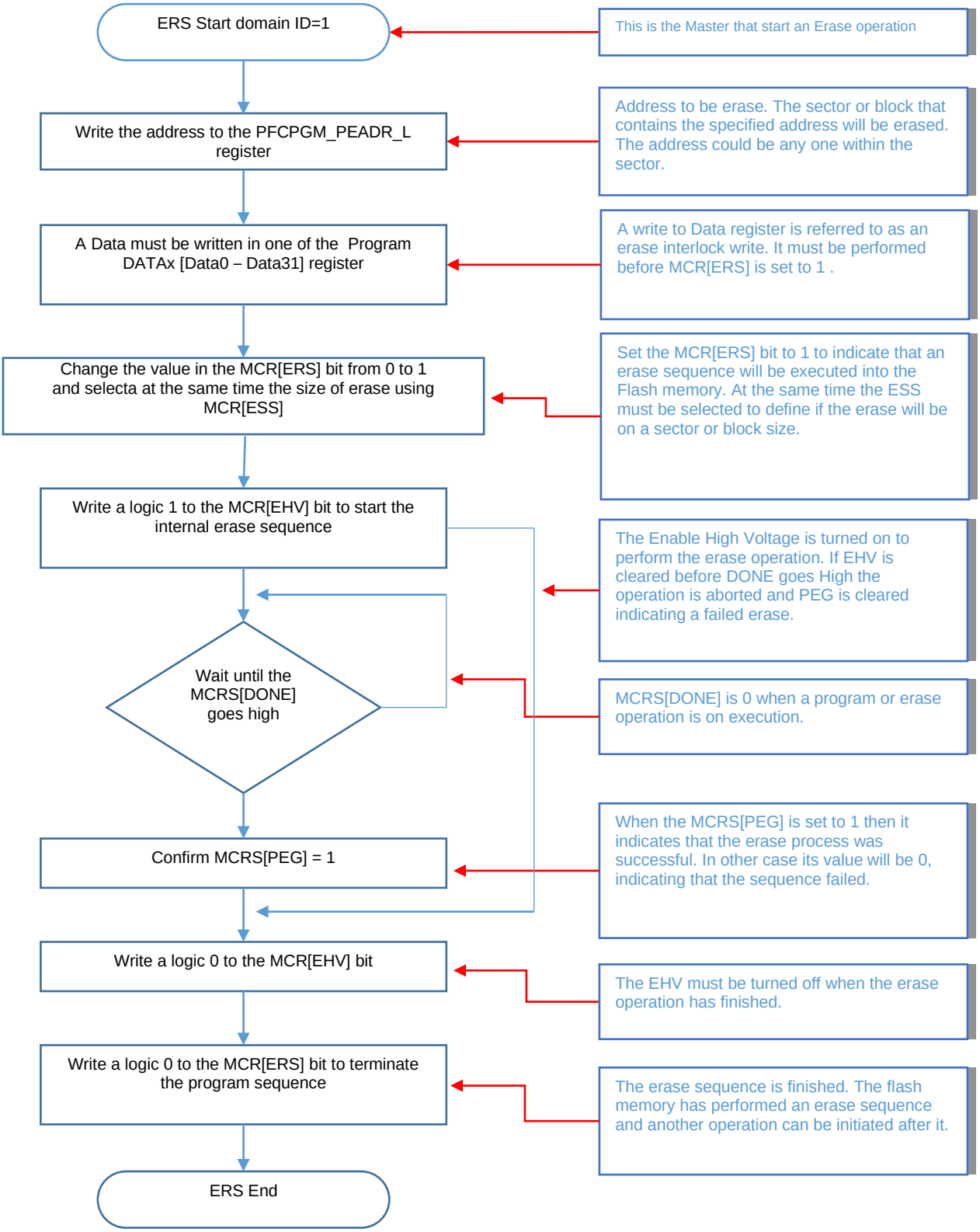

Figure 2. **Erase sequence flow diagram**

The following bare metal code shows the implementation of the write and erase operation, we can note that some functions are shared for both operations in the *Example 3*

Example 1. Erase flash function

```
tFLASH_STATUS FLASH_ErsSector (const void *dst)
{
  FLASH_InitSeq(dst);
  FLASH->DATA[0] = 0UL;               /* one and only one DATA register written */
  return FLASH_ExecSeq(FLASH_MCR_ERS_MASK);
}
```

Example 2. Write flash function

```
{
  register tFLASH_STATUS status = FLASH_PEG_ST;
  register uint8_t *pDst = (uint8_t*)dst, *pSrc = (uint8_t*)src, *pTmp;
  register uint32_t *pData;
  uint32_t tmp;

  while ((nbytes > 0L) && (status & FLASH_PEG_ST))
  {
    pData = (uint32_t*)((uint32_t)&FLASH->DATA[0] + ((uint32_t)pDst & FLASH_DATAX_MASK));
    FLASH_InitSeq (pDst);
    do {
      tmp = 0xffffffffUL;
      pTmp = (uint8_t*)((uint32_t)&tmp + ((uint32_t)pDst & FLASH_BYTES_MASK));
      do {
        *pTmp++ = *pSrc++; pDst++;
      } while ((nbytes-- > 0L) && ((uint32_t)pTmp & FLASH_BYTES_MASK));
      *pData++ = tmp;
    } while ((nbytes > 0L) && ((uint32_t)pData & FLASH_DATAX_MASK));
    status = FLASH_ExecSeq (FLASH_MCR_PGM_MASK);
  }
  return status;
}
```

Example 3. Common Erase/Write functions

```
#define FLASH_WritePEADR(dst)  do{ PFLASH->PFCPGM_PEADR_L=(uint32_t)dst; }while(0)
#define FLASH_GetPEID()  ((FLASH->MCR&FLASH_MCR_PEID_MASK)>>FLASH_MCR_PEID_SHIFT)
#define FLASH_ClrStatus(mask)  do{ FLASH->MCRS=mask; }while(0)
#define FLASH_GetStatus()  (tFLASH_STATUS)FLASH->MCRS

#define FLASH_InitSeq(addr)                                                  \
do{                                                                          \
  register uint8_t domain_id = XRDC->HWCFG1;                                 \
  /* entry semaphore loop                                          */       \
  do{ FLASH_WritePEADR (addr); } while(FLASH_GetPEID () != domain_id);       \
  /* clear any pending program & erase errors                      */       \
  FLASH_ClrStatus (FLASH_PES_ERR|FLASH_PEP_ERR);                             \
 }while(0)
tFLASH_STATUS FLASH_ExecSeq (register uint32_t mask)
{
register tFLASH_STATUS status;

  FLASH->MCR |= mask;                        /* initiate sequence            */
  FLASH->MCR |= FLASH_MCR_EHV_MASK;          /* enable high voltage          */
  while(!(FLASH->MCRS&FLASH_MCRS_DONE_MASK)); /* wait until MCRS[DONE]=1     */
  FLASH->MCR &=~FLASH_MCR_EHV_MASK;          /* disable high voltage         */
  status = FLASH_GetStatus();                /* read main interface status   */
  FLASH->MCR &=~mask;                        /* close sequence               */

  return status;
}
```

3.4. Locking and unlocking sector or super sector

A block is integrated by sectors and super sectors with sizes of 8 KB and 64 KB respectively. These sectors can be protected from write or erase operations by using the locking feature. The last 256 KB of the block has sector protection feature, while the rest has the Super Sector protection feature. The data flash has sector protection feature and the UTEST sector has an independent sector program protection. The S32K3 Product Family has some devices where Super Sector protection is not available due to the memory size, for more information about these devices, please review the S32K3 Product Family Reference Manual.

The following figure shows the sector and super sector distribution for a 1 MB block.

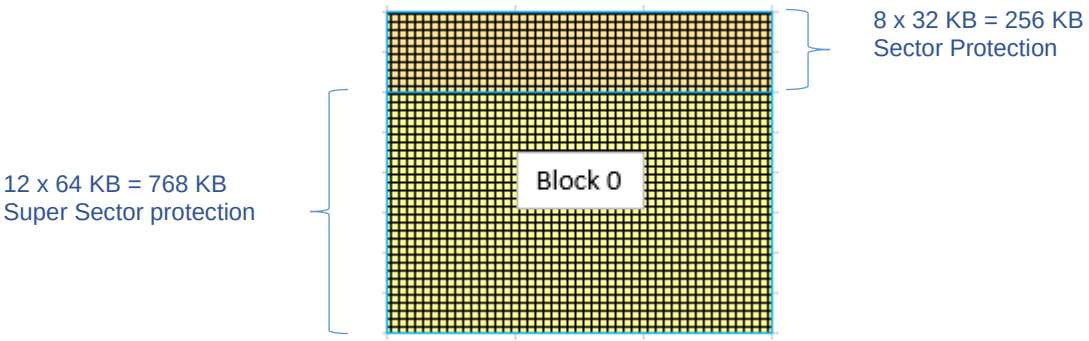

Figure 3. **Sector distribution inside 1MB block**

The lock and unlock process is controlled by the PFCBLKn_SSPELOCKn registers for super sectors and PFCBLKn_SPELOCKn registers for sectors. The PFCBLKn_SSPELOCKn register has 12 available bits, where each bit corresponds to each super sector, similarly, the PFCBLKn_SPELOCKn register has 32 available bits for 32 available sectors. If an unlock process is desired to allow a program or erase operation, then the corresponding register PFCBLKn_SSPELOCKn or PFCBLKn_SPELOCKn must be changed from 1 to 0. Writing 1 to any bit of PFCBLKn_SPELOCKn or PFCBLKn_SSPELOCKn will lock the sector or super sector against programming and erasing operations.

The PFCBLKn_SSPELOCKn and PFCBLKn_SPELOCKn registers values out of reset is 1 for all the bits, this means that all sectors are protected from program and erase operations after reset.

The *Figure 4* shows the steps to unlock a sector or super sector.

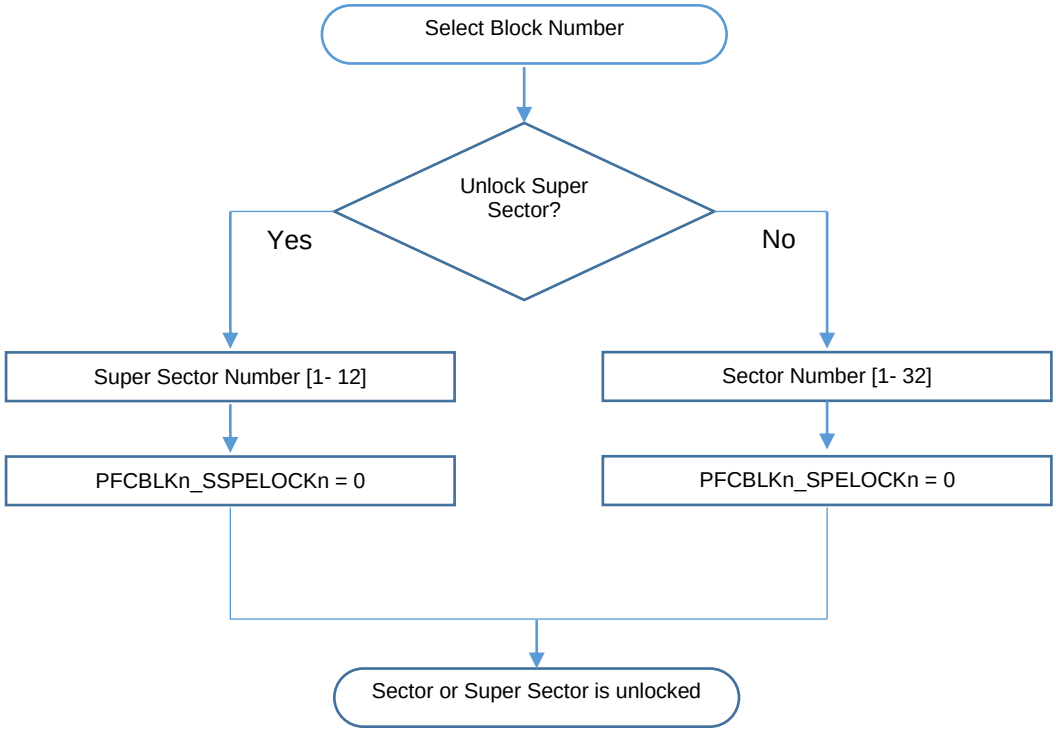

Figure 4. **Unlock process for Sector or Super Sector**

3.5. UTEST sector

The 8 KB UTEST sector is available in all devices from the S32K3 Product Family. In this sector is possible to store important information about the application, for example, version number, permanent parameters, configurations (boot or applications), etc. Inside the UTEST Sector there are some regions that are reserved for the SoC and its use is reserved for NXP. For more detail about available regions please consult the S32K3xx_DCF_client.xlsx attached in the S32K3 Product Family Reference Manual.

The UTEST Sector is an OTP (One Time Programmable) space when the Test mode seal is written. The Test mode seal is allocated in the UTEST Sector, for security it is programmed with the value 0x5A4B3C2D, this means that only new data or configuration could be appended in the UTEST sector and erase is not allowed.

The process to write a data in the UTEST Sector is the same process used to program a data in other blocks. The unlocking process is also the same but with the difference that the UTEST sector has its own register PFCBLKU_SPELOCK[SLCK] to lock or unlock the sector from program operations. As mentioned before, UTEST sector is an 8KB sector so there is only 1 bit to change in the PFCBLKU_SPELOCK register. Following the sector protection logic, if the SLCK bit is set to 0 then the UTEST sector is available for program operations.

The following example is part of a baremetal code that depict how to unlock the UTEST sector, this example can be used for other blocks.

Example 4. Unlocking UTEST sector

```
#define PFLASH_U_PFCBLKI_SPELOCK_COUNT           1u
typedef struct {
  …
__IO uint32_t PFCBLKU_SPELOCK[PFLASH_U_PFCBLKI_SPELOCK_COUNT]; /**< Block UTEST Sector Program Erase Lock, array
offset: 0x358, array step: 0x4 */
…
} PFLASH_Type, *PFLASH_MemMapPtr;

#define PFLASH_PFCBLK5_SPELOCK    PFLASH->PFCBLKU_SPELOCK[0]

#define PFLASH_Unlock(blocks,ssectors,sectors)                                          \
({                                                                                      \
  register uint32_t __t1=blocks,__t2=ssectors,__t3=sectors;                             \
  if (__t1 & PFLASH_BL0) { PFLASH_PFCBLK0_SSPELOCK&=~__t2; PFLASH_PFCBLK0_SPELOCK&=~__t3; } \
  if (__t1 & PFLASH_BL1) { PFLASH_PFCBLK1_SSPELOCK&=~__t2; PFLASH_PFCBLK1_SPELOCK&=~__t3; } \
  if (__t1 & PFLASH_BL2) { PFLASH_PFCBLK2_SSPELOCK&=~__t2; PFLASH_PFCBLK2_SPELOCK&=~__t3; } \
  if (__t1 & PFLASH_BL3) { PFLASH_PFCBLK3_SSPELOCK&=~__t2; PFLASH_PFCBLK3_SPELOCK&=~__t3; } \
  if (__t1 & PFLASH_BL4) {                                 PFLASH_PFCBLK4_SPELOCK&=~__t3; } \
  if (__t1 & PFLASH_BL5) {                                 PFLASH_PFCBLK5_SPELOCK&=~__t3; } \
})
/* unlock UTEST data flash sector                                          */
  PFLASH_Unlock (PFLASH_BL5, PFLASH_SS0, PFLASH_S0);
```

4. Tightly Coupled Memory

The Tightly Coupled Memory (TCM) is a memory implemented from the ARM(R) Cortex M7 architecture which main characteristic is to have a dedicated connection to the core. This memory is divided in Instruction TCM (I-TCM) and Data TCM (D-TCM). In the S32K3 Product Family there are 2 dedicated connections to the Cortex M7, one for the Instruction TCM and another for the Data TCM. Each CM7 core has an ITCM and DTCM memory available, so the sizes and addresses of the TCM memories depends on the number of cores available on the variant and its configuration (lockstep or decoupled), for more information please review the *Table 4 RAM memory architecture for S32K3*.

The TCM is memory mapped and it can be accessed by two possible buses, by the dedicated Core connection and by a backdoor access for any other Master on the AHBS bus. Each access has a defined start address for TCM and the end address is determinate by the size of the memory, for more information about the addresses please consult the *Table 4*.

For the dedicated access for the core, the I-TCM (Instruction-TCM) has a bus interface of 64 bits and the D-TCM (Data-TCM) a bus interface of 32 bits. The TCM memory can be accessed via 32 bits AHB interface by any other master, such as the eDMA, a different Core (when decoupled configuration), the EMAC and the HSE Core.

For devices in lockstep, the TCM of the secondary core is added to the Primary core so the accessible size to the primary core is augmented. In decoupling mode, the TCM is dedicate for each core.

The TCM can be used as System Memory or dedicated memory to the cores. When the TCM is used as System Memory on multicore devices all enabled cores and non-core masters can use the TCMs of the disabled core. In order to allow the usage of ITCM and DTCM of the disabled core as system memory, some register configuration is necessary. For more detail review the S32K3 Product Family reference manual and take as reference the example code in this section.

The following block diagram depicts the route that other masters use to get access to the TCM memory (S32K32x and S32K34x devices).

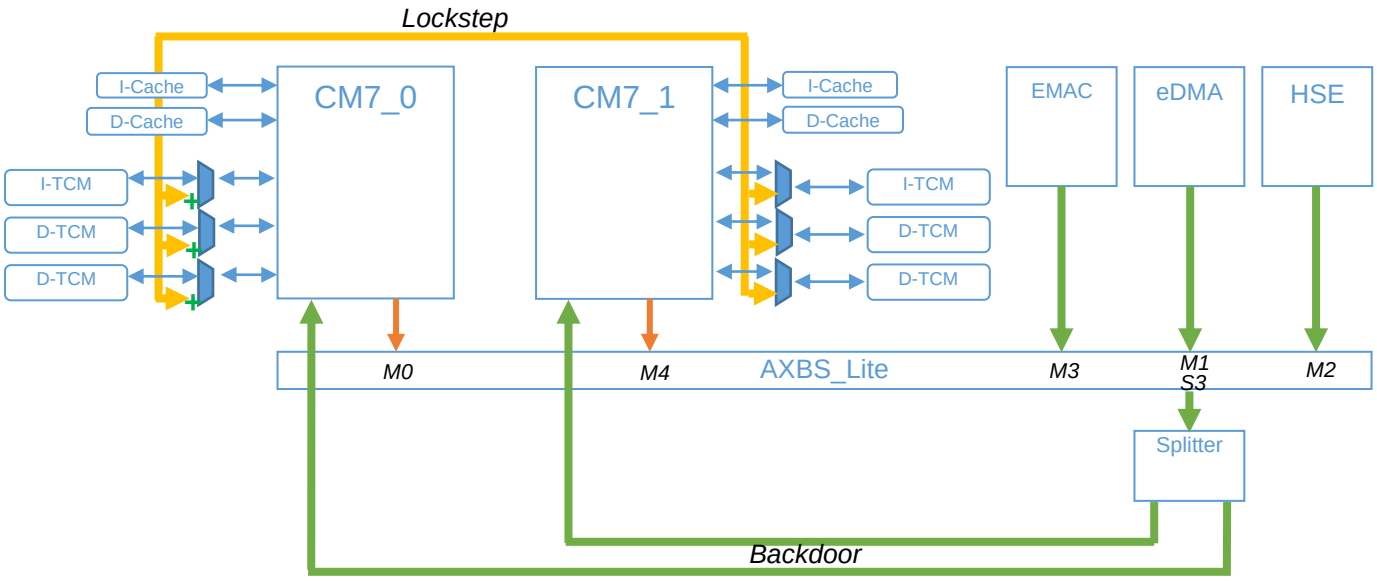

Figure 5. **TCM memory accesses for S32K32x and S32K34x (Direct and Backdoor)**

Due to the fact that the TCM is part of the memory map and it has a physical address, the user can decide the content to be stored in the TCM at compilation time. One advantage of using the TCM is that it has a deterministic access time (one clock cycle), so the TCM can be used to store critical data and code, for example, frequently updated variables, interrupt handlers, data processing, etc. This data is decided by the user and not by a control logic as the Cache memory.

As other memories, TCM has ECC protection, and after the chip's power on reset and before using it, the user must initialized it to avoid ECC errors, this write operation is required to set up the initial ECC code words after the chip's power on reset phase. In the S32K3 Product Family the cores can initialize ITCM and DTCM via the direct and Backdoor accesses. Also, the DTCM can be initialized by the DMA via the backdoor but the DMA cannot initialize the ITCM due to the 32 bits access that the backdoor provides (ITCM needs 64 bits writes to avoid ECC error).

As an example, the next code depicts how to use the TCM as system memory, how to initialize the ITCM via the Core M7_0, and how to initialize the DTCM via the DMA. Important notice, to initialize the DTCM the DMA needs to use the backdoor address in order to get access to the DTCM.

Example 5. TCM Configured as System RAM

```
/******************************** START TCM_Configured as System RAM ****************************/
MC_ME->PRTN2_COFB1_CLKEN |= MC_ME_PRTN2_COFB1_CLKEN_REQ62(1);     /* PRTN2_COFB1_CLKEN_[REQ62] = 1  */
                                                                 /* Enable clock for TCM for CM7_0 */
MC_ME->PRTN2_COFB1_CLKEN |= MC_ME_PRTN2_COFB1_CLKEN_REQ63(1);     /* PRTN2_COFB1_CLKEN_[REQ63] = 1  */
                                                                 /* Enable clock for TCM for CM7_1 */
DCM_GPR->DCMRWF4 |= DCM_GPR_DCMRWF4_cm7_0_cpuwait(1);             /* DCMRWF4[CM7_0_CPUWAIT] = 1     */
                                                                 /* Wait Mode for CM7_0            */
DCM_GPR->DCMRWF4 |= DCM_GPR_DCMRWF4_cm7_1_cpuwait(1);             /* DCMRWF4[CM7_1_CPUWAIT] = 1     */
                                                                 /* Wait Mode for CM7_1            */
MC_ME->PRTN0_CORE0_PCONF |= MC_ME_PRTN0_CORE0_PCONF_CCE(1);       /* PRTN0_CORE0_PCONF[CCE] = 1     */
                                                                 /* Enable Clock for CM7_0         */
MC_ME->PRTN0_CORE1_PCONF |= MC_ME_PRTN0_CORE1_PCONF_CCE(1);       /* PRTN0_CORE1_PCONF[CCE] = 1     */
                                                                 /* Enable Clock for CM7_1         */
/******************************** END TCM_Configured as System RAM ******************************/
```

Example 6. ITCM initialization by core

```
/********************************** START ITCM_Initialization by Core ****************************/
uint64_t* ITCM_begin = (uint64_t*)&__ITCM_START;         /* Variable to load ITCM start address   */
uint64_t* ITCM_size = (uint64_t*)&__ITCM_SIZE;           /* Variable to load ITCM size            */

while(ITCM_begin < ITCM_size)                            /* Loop to initialize ITCM region        */
{
        *ITCM_begin = (uint64_t)0x00;
        ITCM_begin ++;
}
/********************************** END ITCM_Initialization by Core ****************************/
```

Example 7. DTCM initialization by eDMA

```
/********************************** START DTCM_Initialization by eDMA ****************************/
/* Source Data Address */
TCD[0].TCD0_SADDR = 0x00400018;
/* Offset for the current Source Address = 0x00 (no offset applied)  */
TCD[0].TCD0_SOFF = 0;
/* Source Data Transfer Size = 010b 32 bits */
TCD[0].TCD0_ATTR = DMA_TCD_TCD0_ATTR_SSIZE(2) | DMA_TCD_TCD0_ATTR_DSIZE(2);
/* Number of bytes to transfer = 128KB */
TCD[0].NBYTES0.TCD0_NBYTES_MLOFFNO = (uint32_t)(&__DTCM_SIZE);
/* Last Source Address adjustment */
TCD[0].TCD0_SLAST_SDA = 0x00000000;
/* Destination Address = DTCM Backdoor Start Address (0x2100_0000h) */
TCD[0].TCD0_DADDR = (uint32_t)(__DTCMBD_START);
/* Offset for the current Destination Address = 0x04 (4 bytes offset applied)  */
TCD[0].TCD0_DOFF = DMA_TCD_TCD0_DOFF_DOFF(0x04);
/* Current Major Iteration Count = 1*/
TCD[0].CITER0.TCD0_CITER_ELINKNO = DMA_TCD_TCD0_CITER_ELINKNO_CITER(0x1);
/* Last Destination Address */
TCD[0].TCD0_DLAST_SGA = DMA_TCD_TCD0_DLAST_SGA_DLAST_SGA(-(uint32_t)(&__DTCM_SIZE));
/* Starting Major Iteration Count = 1 */
TCD[0].BITER0.TCD0_BITER_ELINKNO = DMA_TCD_TCD0_BITER_ELINKNO_BITER(0x1);
/* Channel Start transfer initiated */
TCD[0].TCD0_CSR = DMA_TCD_TCD0_CSR_START(0x1);

/* Loop waiting for Channel major loop has been completed  */
while(!(TCD[0].CH0_CSR & DMA_TCD_CH0_CSR_DONE_MASK));
/* Clear DONE bit field*/
TCD[0].CH0_CSR |= DMA_TCD_CH0_CSR_DONE_MASK;
/********************************** END DTCM_Initialization by eDMA ****************************/
```

5. SRAM

The S32K3 Product Family devices can be integrated from 1 SRAM and up to 3 SRAM blocks. The *Table 4* shows the RAM memory blocks available for each device. Remember that the TCM memory is considered a part of the RAM memory.

Table 4. **RAM memory architecture for S32K3**

RAM	S32K311 S32K310	S32K312	S32K314	S32K322	S32K324	S32K328	S32K342	S32K344	S32K348	S32K338	S32K358
End Address SRAM2 Start Address	Not Available	Not Available	Not Available	Not Available	Not Available	0x204B_FFFF 256 KB 0x2048_0000	Not Available	Not Available	0x204B_FFFF 256 KB 0x2048_0000	0x204B_FFFF 256 KB 0x2048_0000	0x204B_FFFF 256 KB 0x2048_0000
End Address SRAM1 Start Address	Not Available	Not Available	0x2044_FFFF 160 KB 0x2042_8000	Not Available	0x2044_FFFF 160 KB 0x2042_8000	0x2047_FFFF 256 KB 0x2044_0000	Not Available	0x2044_FFFF 160 KB 0x2042_8000	0x2047_FFFF 256 KB 0x2044_0000	0x2047_FFFF 256 KB 0x2044_0000	0x2047_FFFF 256 KB 0x2044_0000
End Address SRAM0 Start Address	0x2040_7FFF 32 KB 0x2040_0000	0x2041_7FFF 96 KB 0x2040_0000	0x2042_7FFF 160 KB 0x2040_0000	0x2040_FFFF 64 KB 0x2040_0000	0x2042_7FFF 160 KB 0x2040_0000	0x2043_FFFF 256 KB 0x2040_0000	0x2040_FFFF 64 KB 0x2040_0000	0x2042_7FFF 160 KB 0x2040_0000	0x2043_FFFF 256 KB 0x2040_0000	0x2043_FFFF 256 KB 0x2040_0000	0x2043_FFFF 256 KB 0x2040_0000
End Address DTCM2 Start Address	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	0x2001_FFFF 128 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000
End Address DTCM1 Start Address	Not Available	Not Available	Not Available	0x0000_FFFF 64 KB 0x2000_0000	0x0000_FFFF 64 KB 0x2000_0000	0x0000_FFFF 64 KB 0x2000_0000	Not Available	Not Available	Not Available	0x2000_FFFF 64 KB 0x2000_0000	Not Available
End Address DTCM0 Start Address	0x2000_FFFF 64 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000	0x2000_FFFF 64 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000
End Address ITCM2 Start Address	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	0x0000_FFFF 64 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000
End Address ITCM1 Start Address	Not Available	Not Available	Not Available	0x0000_7FFF 32 KB 0x0000_0000	0x0000_7FFF 32 KB 0x0000_0000	0x0000_7FFF 32 KB 0x0000_0000	Not Available	Not Available	Not Available	0x0000_7FFF 32 KB 0x0000_0000	Not Available
End Address ITCM0 Start Address	0x000_7FFF 32 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000	0x000_7FFF 32 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000

Each SRAM block has its own SRAM Controller (PRAMC) to support fast read/write accesses to the cores. The PRAM controller is the interface between the system bus and the integrated RAM array. The system bus supports 64 bit of data and the RAM array supports 64 bit of data + 8 bit for ECC.

Inside the SRAM memory there is a region that is available when the microcontroller is in Standby mode. The Standby SRAM is allocated at the first 32 KB of the SRAM memory. This region is sourced by the Power Domain 0, so when a standby mode is performed the information that is stored in the Standby region is retained. The rest of the SRAM memory is sourced by the Power Domain 1 and it is only available in Run Mode.

Just like the TCM memory, all the SRAM memory must be initialized to avoid ECC errors and this can be done by the core or the DMA as well. If the initialization is omitted then any read or write to the SRAM memory will generate an uncorrectable ECC error event.

Example 8. SRAM initialization by DMA

```
/********************************* START SRAM Initialization by DMA ******************************/
TCD[0].TCD0_SADDR = 0x00400018;
TCD[0].TCD0_SOFF = 0;
TCD[0].TCD0_ATTR = DMA_TCD_TCD0_ATTR_SSIZE(3) | DMA_TCD_TCD0_ATTR_DSIZE(3);
TCD[0].NBYTES0.TCD0_NBYTES_MLOFFNO = (uint32_t)(&__RAM_SIZE);
TCD[0].TCD0_SLAST_SDA = 0x00000000;
TCD[0].TCD0_DADDR = (uint32_t)(&__RAM_START);
TCD[0].TCD0_DOFF = DMA_TCD_TCD0_DOFF_DOFF(0x8);
TCD[0].CITER0.TCD0_CITER_ELINKNO = DMA_TCD_TCD0_CITER_ELINKNO_CITER(0x1);
TCD[0].TCD0_DLAST_SGA = DMA_TCD_TCD0_DLAST_SGA_DLAST_SGA(-(uint32_t)(&__RAM_SIZE));
TCD[0].BITER0.TCD0_BITER_ELINKNO = DMA_TCD_TCD0_BITER_ELINKNO_BITER(0x1);
TCD[0].TCD0_CSR = DMA_TCD_TCD0_CSR_START(0x1);

while(!(TCD[0].CH0_CSR & DMA_TCD_CH0_CSR_DONE_MASK));
TCD[0].CH0_CSR |= DMA_TCD_CH0_CSR_DONE_MASK;
/********************************* END SRAM Initialization by DMA ********************************/
```

5.1. Read

Read events can be configured to be completed with a zero wait state or one wait state response for any data size. The PRAM controller register Flow Trough Disable field PRCRx[FT-DIS] insert a wait state on read events prior to returning the data to the system bus. Insertions of wait states are important to consider when system frequency are greater than 120MHZ. This wait state doesn’t have any effect on write events. For more information about wait states please refer to the Gasket configurations in the Clocking Chapter of the S32K3 Reference Manual.

5.2. Write

Write operations can be made in 64 bits or less. When an aligned 64 bits write is performed the write operation is executed in a single phase cycle with a zero wait state. When a write less than 64 bits or unaligned write is performed a Read-Modify-Write action is executed in order to perform the write and recalculate the new ECC code. The RMW action will insert some cycles to the write process and this means that aligned 64 bits writes has better performance than unaligned < 64 bits writes to the SRAM memory.

The Read-Modify-Write can be explained in the following sequence:

1. The PRAMC performs a Single Error Correction (SEC)/Double Error Detection (DED) in the corresponding read data
2. The write data is merged with the previous read data and only bits of the write data are changed
3. A new ECC code is generated according to the new 64 bits
4. The new double word and ECC is written to RAM.

6. Use Cases

6.1. Flash vs TCM vs SRAM

One of the main features of S32K3 Product Family is the implementation of TCM memory. One of the advantages of TCM memory is a deterministic time for process task or use of data stored in this memory in order to avoid any latency in the data transfer between core with the SRAM or with the Flash Memory.

The following use case demonstrates the main difference between running code from Flash vs TCM and vs RAM memory. Also, this example helps the customer to learn how to run functions or data loaded in TCM or RAM instead of Flash memory. This use case was created for a S32K344 device and tested on the S32K3XXEVB-Q257 evaluation board, the following guidelines in this chapter are shown for S32 Design Studio with GCC toolchain, but the procedure is similar for other IDEs and toolchains.

6.1.1. Linker file

The first step is to check if our linker file has declared the TCM, Flash and SRAM memory regions with their corresponding size.

The memory regions on linker file should looks something like that:

Example 9. Linker memory definition and size

```
/************************* Linker script to configure memory regions. *****************************/
MEMORY
{
  ITCM       (RWX) : ORIGIN = 0x00000000, LENGTH = 0x10000
  PFLASH      (RX) : ORIGIN = 0x400000,   LENGTH = 0x3f4000
  DFLASH      (RX) : ORIGIN = 0x10000000, LENGTH = 0x20000
  DTCM        (RW) : ORIGIN = 0x20000000, LENGTH = 0x20000
  SRAM0_STDBY (RW) : ORIGIN = 0x20400000, LENGTH = 0x8000
  SRAM        (RW) : ORIGIN = 0x20408000, LENGTH = 0x48000
}
```

We need to define some sections inside each memory regions where we want to load the function or data. For this example we are going to define the next sections:

- ITCM_code (0x0000_0000): section on ITCM to load and running the TCM_Function.
- DTCM_data (0x2000_0000): section on DTCM to load some arrays and variables.
- SRAM_Function will use the standard .data section from the SRAM (start address 0x2040_8000).
- Flash functions will be loaded in the .text section by default.

The linker file will looks something like this:

Example 10. Sections for code and data for ITCM, DTCM and SRAM

```
__coderom_start__ = .;
  .ITCM_code : AT (__coderom_start__)
  {
    . = ALIGN(8);
    __coderam_start__ = .;
    *(.ITCM_code*)
    . = ALIGN(8);
    __coderam_end__ = .;
  } > ITCM
  . = __coderom_start__ + ( __coderam_end__ - __coderam_start__ );
  __coderom_end__ = .;

  __etext = ALIGN(8);

  .DTCM_data :
  {
    . = ALIGN(4);
        __DTCM_data__ = .;
  } > DTCM

  .standby_ram :
  {
    *(.standby_ram)
  } > SRAM0_STDBY

  /* Due ECC initialization sequence __data_start__ and __data_end__ should be aligned on 8 bytes */
  .data : AT (__etext)
  {
    . = ALIGN(8);
    __data_start__ = .;
    *(vtable)
    *(.data)
    *(.SRAM_Function)
    *(.data.*)
. = ALIGN(4);
    /* preinit data */
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP(*(.preinit_array))
```

6.1.2. Startup

To avoid ECC error ITCM, DTCM and SRAM are initialized. Also, the functions code needs to be copied to ITCM and SRAM. This process is according to the linker file and processed by the startup_code.

Example 11. Initialization for ITCM, DTCM and SRAM to avoid ECC errors according to the table specified in the linker file

```
  /**
    \brief   Early system init: ECC, TCM etc.
    \details This default implementation initializes ECC memory sections
             relying on .ecc.table properly in the used linker script.

   */
__STATIC_FORCEINLINE void __cmsis_cpu_init(void)
{
#if defined (__ECC_PRESENT) && (__ECC_PRESENT == 1U)
  typedef struct {
    uint64_t* dest;
```

```
      uint64_t  wlen;
  } __ecc_table_t;

  extern const __ecc_table_t __ecc_table_start__;
  extern const __ecc_table_t __ecc_table_end__;

  for (__ecc_table_t const* pTable = &__ecc_table_start__; pTable < &__ecc_table_end__; ++pTable) {
    for(uint64_t i=0u; i<pTable->wlen; ++i) {
      pTable->dest[i] = 0xDEADBEEFFEEDCAFEUL;
    }
  }
```

The copy of code and data from flash to ITCM and SRAM is also processed on the startup

Example 12. Copy code to ITCM, DTCM and SRAM

```
__STATIC_FORCEINLINE __NO_RETURN void __cmsis_start(void)
{
  extern void _start(void) __NO_RETURN;

  typedef struct {
    uint32_t const* src;
    uint32_t* dest;
    uint32_t  wlen;
  } __copy_table_t;

  typedef struct {
    uint32_t* dest;
    uint32_t  wlen;
  } __zero_table_t;

  typedef struct {
    uint32_t const* src;
    uint32_t* dest;
    uint32_t  wlen;
   } __copycode_table_t;

  extern const __copy_table_t __copy_table_start__;
  extern const __copy_table_t __copy_table_end__;
  extern const __zero_table_t __zero_table_start__;
  extern const __zero_table_t __zero_table_end__;
  extern const __copycode_table_t __copycode_table_start__;
  extern const __copycode_table_t __copycode_table_end__;

  for (__copy_table_t const* pTable = &__copy_table_start__; pTable < &__copy_table_end__; ++pTable) {
    for(uint32_t i=0u; i<pTable->wlen; ++i) {
      pTable->dest[i] = pTable->src[i];
    }
  }

  for (__zero_table_t const* pTable = &__zero_table_start__; pTable < &__zero_table_end__; ++pTable)      {
    for(uint32_t i=0u; i<pTable->wlen; ++i) {
      pTable->dest[i] = 0u;
    }
  }

   for (__copycode_table_t const* pTable = &__copycode_table_start__; pTable < __copycode_table_end__;
   ++pTable)
   {
    for(uint32_t i=0u; i<pTable->wlen; ++i) {
      pTable->dest[i] = pTable->src[i];
    }
  }

  _start();
}
```

6.1.3. Allocating code

The function attribute section needs to be used for the functions and data that needs to be allocated in SRAM, ITCM or DTCM

__attribute__((__section__(".Section_Name")))

Where the Section_Name is the section that was previously declared in the linker file.

Use this attribute before the function or variable.

Example 13. Use of the attribute section to allocate functions on ITCM, DTCM and SRAM

```
__attribute__((__section__(".DTCM_data")))uint32_t ClkCycleCounter[18] = {0};
__attribute__((__section__(".DTCM_data")))static uint8_t Counter;


/*******************************************************************************************/
/*                          ITCM Function                                                  */
/* Function loaded from ITCM. This function clean pDest and pSrc and copy the             */
/* content of the array dummy[1024] stored in flash to pDest and pSrc                      */
/* Inputs: *pDest = Pointer to Destination array                                           */
/*         *pSrc = Pointer to Source array                                                 */
/*         size = Array size to copy                                                      */
/* Output: ClkCycleCounter Array = Store the number of cycles taken since the              */
/*                       beginning of the function until the end. There are 18 positions   */
/*                       to store all function results                                     */
/*******************************************************************************************/
__attribute__((__section__(".ITCM_code")))
static void TCM_Function(uint8_t *pDest, uint8_t *pSrc, uint32_t size)
{
        /* Function example content */
        uint32_t i;
        for(i=0u; i<size; i++)
        {
          pSrc[i] = 0u;
          pDest[i] = 0u;
        }
        for(i=0u; i<size; i++)
        {
          pSrc[i] = pBuffer[i];
        }
}
__attribute__((__section__(".SRAM_Function")))
static void SRAM_Function(uint8_t *pDest, uint8_t *pSrc, uint32_t size)
{
        /* Function example content */
        uint32_t i;
        for(i=0u; i<size; i++)
        {
          pSrc[i] = 0u;
          pDest[i] = 0u;
        }
        for(i=0u; i<size; i++)
        {
          pSrc[i] = pBuffer[i];
        }

}
```

6.1.4. Results

The *Table 5* shows the number of cycles taken by functions and data on different memories. It is important to notice that the number of cycles is smaller when running from ITCM in comparison to running from Flash. This advantage can be used to reduce the number of cycles and timing for tasks that require deterministic times or avoid latencies. The *Table 5* also shows the use of enable Cache, Instruction Cache and Data Cache. When the D_Cache and I_Cache are enabled the number of cycles is reduced significantly. This is due to the fact that the Cache fetched all the code and data in its region. Flash and SRAM can be cacheable on S32K3, the TCM memory is zero wait access so no Cache is needed for ITCM and DTCM. This is a simple code and cache is enough to store all of it. However, in real application it is unlikely to place the entire code in Cache so the characteristics of TCM can be used as advantage.

Table 5. **Number of cycles taken by functions and data on different memories.**

	SRAM				DTCM			
	Cleaning 2 array buffers (size 1024 bytes)	Copying array from Flash to SRAM	Copying array from SRAM to SRAM	Total (Cleaning + Flash to SRAM + SRAM to SRAM)	Cleaning 2 array buffers (size 1024 bytes)	Copying array from Flash to DTCM	Copying array from DTCM to DTCM	Total (Cleaning + Flash to DTCM + DTCM to DTCM)
	Copying 1024 Bytes - Disable Cache and no optimizations							
Running Function from Flash	28780	36829	30366	95975	27510	36794	24351	88655
Running Function from SRAM	28956	36682	32398	98036	27505	36746	27718	91969
Running Function from ITCM	28838	33793	29687	92318	**27492**	**33817**	**24300**	**85609**
	Copying 1024 Bytes - Enable I Cache and no optimizations							
Code is fetched to I-Cache	28930	38602	29720	97252	27496	36708	24321	88525
	28911	38581	29709	97201	27500	36708	24322	88530
	28902	33806	29687	92395	**27492**	**33817**	**24300**	**85609**
	Copying 1024 Bytes - Enable I Cache and D Cache and no optimizations							
Code and Data is fetched to I-Cache and D-Cache	13428	14792	11315	39535	13329	14392	11314	39035
	13458	14392	11314	39164	13330	14392	11314	39036
	13458	14392	11314	39164	13330	14392	11313	39035

It can be noticed that while running code from ITCM and transferring data to DTCM we have the same result similar to the I-Cache run. This is because the TCM and core spend the same number of cycles as Cache.

6.2. SRAM standby

There are 32 KB of SRAM memory that are available when the MCU is in standby mode. This memory region allows to store critical data or important code that needs to be available when the MCU wakes from the standby mode. The MCU operates in two modes: Run mode and Standby mode. When it operates in run mode the MCU has available all the peripherals and modules. In Standby mode the MCU has available only a few of them because some of the power domains are disconnected from the power supply inside the chip. The purpose of this is to achieve a low power mode in order to save power consumption and wake up with an event, power on reset, destructive reset or a functional reset.

The next use case is a simple exercise that shows the Standby SRAM memory feature.

6.2.1. Linker file

First at all, in the linker file the SRAM Standby region need to be declared, please refer to the *Example 9. Linker memory definition and size*

As noted, the Standby SRAM is allocated at the first 32 KB of the SRAM Memory.

6.2.2. Main file

For the purpose of this example we are going to use three buffers:

- dummy_array[1024]: this is an array located in Flash, its size is 1024 bytes.

```
const uint8_t dummy_array[1024UL] =
{
  0x00,0x01,0x02,0x03,…
}
```

- SRAM_buffer[BUFFER_SIZE]: this is an array that will be located in the SRAM memory in any region after 0x20408000 address.

```
uint8_t SRAM_buffer [BUFFER_SIZE];
```

- SRAM_SB_buffer[BUFFER_SIZE]: this is an array that will be located in the Standby memory

```
__attribute((section(".standby_ram"))) uint8_t SRAM_SB_buffer [BUFFER_SIZE];
```

Where BUFFER_SIZE has been defined as 1024

```
#define BUFFER_SIZE       1024
```

In the main file array buffers are cleared and dummy array bytes are copied to SRAM_buffer[] and SRAM_SB_buffer[].

Example 14. Clear and copy data to Standby SRAM array and SRAM array

```
Uint32_t *pDest_SRAM_SB;
pDest_SRAM_SB = &SRAM_SB_buffer;

/* Clear buffer */
for(i=0; i<BUFFER_SIZE; i++)
{
    SRAM_SB_buffer[i] = 0;
    SRAM_buffer[i] = 0;
}
/* Copy dummy array into SRAM_buffer and SRAM_SB_buffer*/
for(i=0u; i<BUFFER_SIZE; i++)
{
    pDest_SRAM_SB[i] = dummy_array[i];
    SRAM_buffer[i] = dummy_array[i];
}
```

SRAM_SB_buffer[]

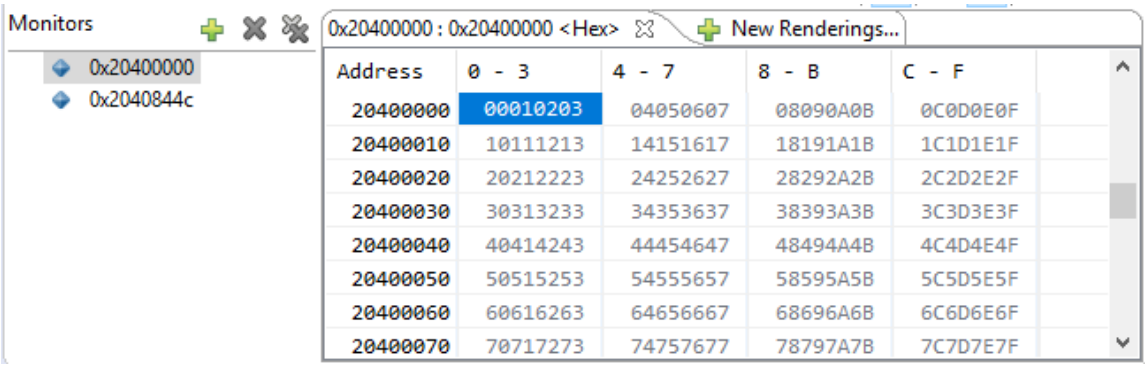

Figure 6. **Standby SRAM map memory**

SRAM_buffer[ ]

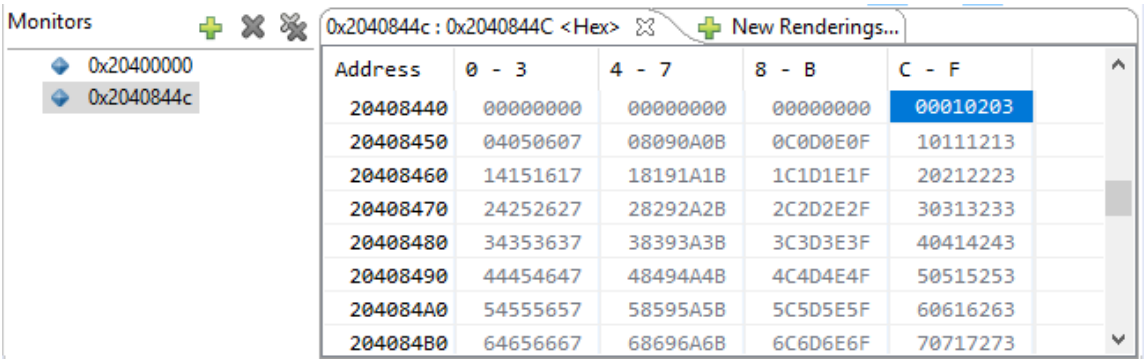

Figure 7. **SRAM map memory**

After that a dummy read is performed in order to validate that the data is available from Standby SRAM and SRAM memory.

Example 15. Dummy read to Standby SRAM array and SRAM array

```
read_SRAM_SB = *(uint32_t*) 0x20400000;
read_SRAM = *(uint32_t*) 0x20408830;
```

(x)= read_SRAM_SB	uint32_t	0x3020100 (Hex)
(x)= read_SRAM	uint32_t	0xe7e6e5e4 (Hex)

Figure 8. **Data read by dummy read**

The Standby entry sequence is performed by pressing SW4 in the interrupt handler.

Example 16. SW4 configuration to generate an interrupt to enter to Standby

```
/* Pin Configuration for PTB26 (SW4) */
    SIUL2->IREER0 &= ~SIUL2_IREER0_IREE13_MASK; /* Disable Rising Edge event */
    SIUL2->IFEER0 |= SIUL2_IFEER0_IFEE13_MASK;  /* Enable Falling Edge event */
    SIUL2->IMCR[541-512] |= SIUL2_IMCR_SSS(0b010); /* Enable Source Signal EIRQ[13] */
    SIUL2->MSCR[58] |= SIUL2_MSCR_IBE_MASK;        /* IBE=1: Input Buffer Enabled */
    SIUL2->DIRSR0 &= ~SIUL2_DIRSR0_DIRSR13_MASK;   /* Select Interrupt Request for PTB26 */
    SIUL2->DISR0 = 0xFFFFFFFFU;  /* Clear Status Flag Interrupt */
    SIUL2->DIRER0 |= SIUL2_DIRER0_EIRE13_MASK;  /* External Interrupt enable for EIRQ[13] */
```

Example 17. Interrupt routine to entry to Standby

```
void SIUL_1_Handler (void)
{
        /* Wait until SW4 (PTB26) release */
        while ((SIUL2->GPDI58 & SIUL2_GPDI58_PDI_n_MASK) == 1) { }
        /* Drive low on PTB30 to turn-off LED D32 */
        SIUL2->GPDO30 &= ~SIUL2_GPDO30_PDO_n_MASK;
        Standby_Entry_Sequence();
}
```

Once Standby_Entry_Sequence is performed the S32K3 enters into standby mode and the debugger is disconnected.

6.2.3. Wakeup

The wakeup process is performed by pressing the SW5 which is configured to generate an interrupt.

Example 18. SW5 configuration to generate a wakeup from Standby

```
/* Pin Configuration for PTB19 (SW5) */
// WKPU[38]
SIUL2->MSCR[51] = SIUL2_MSCR_IBE_MASK       /* IBE=1: Input Buffer Enabled */
                | SIUL2_MSCR_PUE_MASK;

WKPU->WIFEER_64 &= ~0x00000400;  /* Disable WKPU[38] falling edge */
WKPU->WIREER_64 |= 0x00000400;   /* Enable WKPU[38] rising edge */
WKPU->WIFER_64 |= 0x00000400;    /* WKPU[38] glitch filter enabled */
WKPU->WISR_64 = 0x00000400;      /* Write 1 to Clear WKPU[38] flag */
WKPU->IRER_64 |= 0x00000400;      /* Write 1 to Interrupt request WKPU[38] flag */
WKPU->NCR &= ~(WKPU_NCR_NDSS0_MASK | WKPU_NCR_NDSS1_MASK); /* NMI Destination Source Select  */
WKPU->NCR |=  WKPU_NCR_NWRE0_MASK | WKPU_NCR_NWRE1_MASK; /* NMI Wakeup Request Enable */
WKPU->WRER_64 |= 0x0000400;     /* Enable WKPU[38] input */
```

After wakeup the MCU can be reconnected to the Debugger selecting the following remarked option in the S32DS.

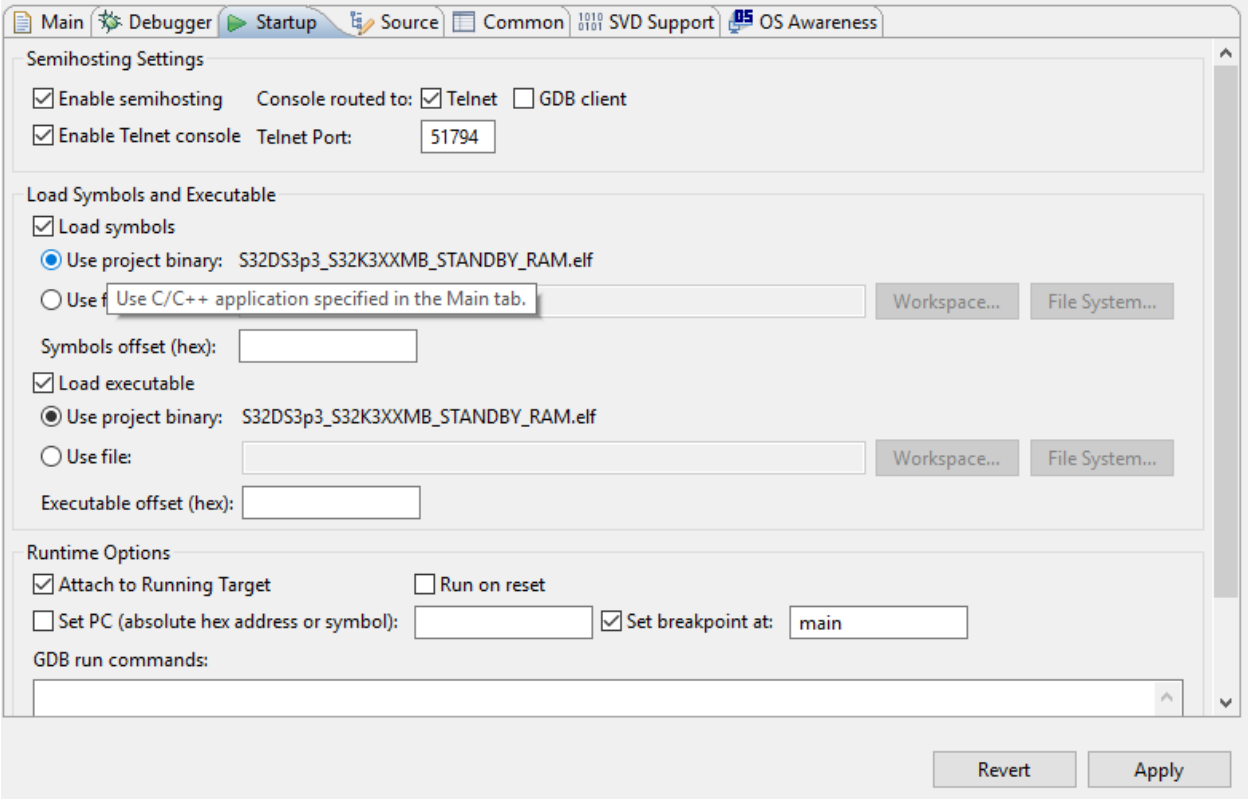

Figure 9. **Option to reconnect the debugger to the current code position**

The interrupt handler is shown in the *Example 19*.

Example 19. Interrupt routine is attended after wakeup

```
void WKPU_Handler (void)
{
        for(uint32_t i=0; i<0x02FFFFFF; i++)
        {
                __NOP();
        }
        read_SRAM_SB = *(uint32_t*) 0x20400000;
        read_SRAM = *(uint32_t*) 0x20408830;
        while(1);
}
```

Where

(x)= read_SRAM_SB	uint32_t	0x3020100 (Hex)
(x)= read_SRAM	uint32_t	0x0 (Hex)

Figure 10. **Data read by dummy read**

SRAM_SB_buffer[]

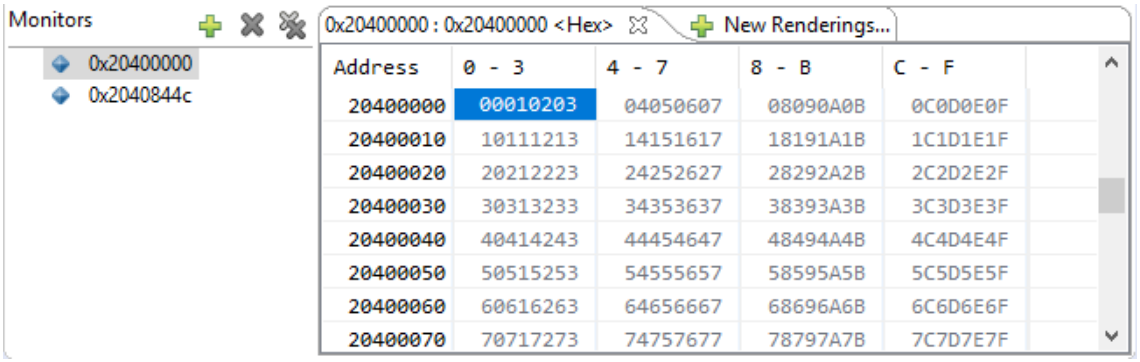

Figure 11. **Standby SRAM map memory**

SRAM_buffer[]

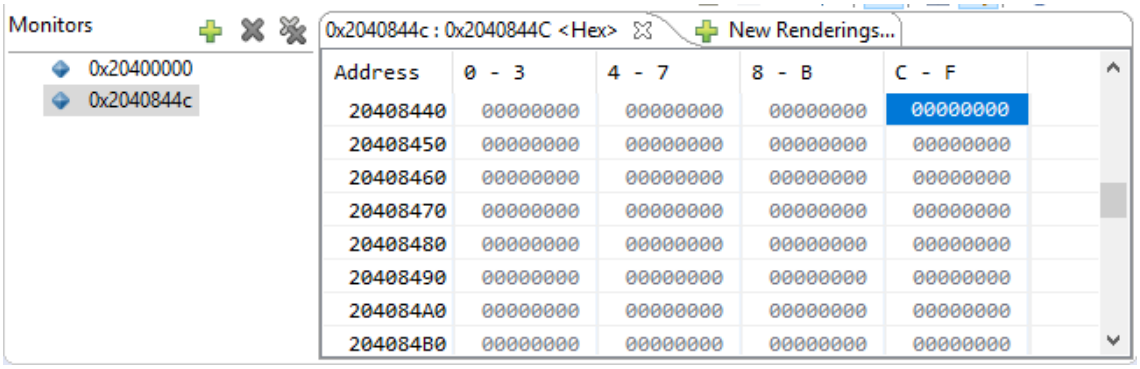

Figure 12. **SRAM map memory**

6.2.4. Results

As noted, the data stored in the Standby SRAM memory sourced by Standby domain is retained when the MCU is in standby mode and available after the wakeup. But the data in SRAM sourced by Run domain is not available and it needs to be initialized after the wakeup to avoid ECC errors. Is important to remark that after wakeup, the Standby SRAM doesn't need to be initialized to avoid ECC error, but the rest of the SRAM does need it, so a proper distinction should be performed in the Startup code. An example to make this distinction is depicted in the following code.

Example 20. Standby RAM initialization only after POR

```
/* Initialize STANDBY RAM if chip comes from POR */
  if (MC_RGM->DES & MC_RGM_DES_F_POR_MASK)
  {
    /* Initialize STANDBY RAM */
    cnt = (( uint32_t)(&__STDBYRAM_SIZE)) / 8U;
    pDest = (uint64_t *)(&__STDBYRAM_START);
    while (cnt--)
    {
      *pDest = (uint64_t)0xDEADBEEFCAFEFEEDULL;
      pDest++;
    }
    MC_RGM->DES = MC_RGM_DES_F_POR_MASK; /* Write 1 to clear F_POR */
  }
```

7. SW recommendations and conclusions

A correct use of different memories available in the S32K3 Product Family can represent a better performance of our application. The user needs to evaluate what functions and data of the application will be stored in the different memories in order to have the best performance.

The three important recommendation to get a good performance in applications are:

- Deterministic task should be considered to be inside the ITCM and DTCM because cores can avoid significant access time to other memories.
- Enable Caches has a great advantage for the applications due to it fetched code and data that cores need in a quickly time, Cache can be enabled and disabled in certain location of the code in order to have a better control of what code or data need to be fetched, is important to note that Cache needs to be invalidated before fetch new code or data, unfortunately the Cache size is small but we can use Tightly Coupled Memory to support the lack of Cache's space.
- The Standby SRAM is another advantage that can be explored in different applications because after our application wakes up from standby mode the data stored in Standby SRAM can be used without doing any other operation. A good example is sending the stored data when an external communication wakes up the MCU from standby mode.

8. References

- S32K3xx Reference Manual
- S32K3xx Data Sheet
- Customer Evaluation Board S32K3XXEVB-Q257
- Automotive SW - S32K3 - Real-Time Drivers for Cortex-M
- S32 Design Studio IDE

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN13388
Rev. 0
11/2021