恩智浦半导体
应用笔记

文件编号：AN13388
第 0 版，2021 年 11 月

S32K3 存储器指南

1. 介绍

本应用笔记将帮助读者了解 S32K3 系列芯片的存储器的特性。本文档详细介绍了在考虑性能提升的情况下运行应用程序的最佳实践及其可用功能。

S32K3 系列芯片有 4 种存储器：Flash、SRAM、TCM 和 Cache 存储器。S32K3 系列芯片还内置了一些带有专用存储器的模块，如 EMAC 和 CAN。本文档主要介绍 Flash、TCM 和 SRAM 存储器。

Flash 专用于编写代码和存储数据。此外，S32K3 系列中的所有芯片都有一个 8KB 的 UTEST 扇区，用于存储重要配置或为应用程序保留信息。S32K3 系列芯片的 Flash program 存储器从 512KB 到 8MB 不等。

RAM 由 SRAM 和 TCM 组成。SRAM 存储器的部分区域在 Standby 状态下可用。这意味着，在将 MCU 设置为 Standby 模式后，该存储器的内容将被保留。S32K3 系列芯片利用了 Arm Cortex M7 架构的 TCM 特性，其主要目的是让一些重要数据在内核访问上有确定性的时间保证，避免访问中出现任何延迟。实时操作系统可以利用该特性。

目录

Cache 存储器是内核的专用存储器。该存储器不是系统存储器的一部分，也没有可供编程人员使用的物理地址。该存储器用作处理器和主存储器之间的缓冲区，以减少内核访问存储器的时间。

2. 特性

S32K3 系列芯片的存储器特性见 表 1 和 表 2。

表 1. S32K3 系列芯片的存储器特性

特性	S32K310	S32K311	S32K341	S32K312	S32K322	S32K342
内核数量	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 存储器 (MB)	512 KB	1		2		
Data Flash 存储器 (KB)[1]	64	128				
Cache	I Cache 8 KB D Cache 8 KB					
总 RAM (KB)	128 KB（包括 96 KB 的 TCM）		256 KB（包括 192 KB 的 TCM）	192 KB（包括 96 KB 的 TCM）	256 KB（包括 192 KB 的 TCM）	
Standby RAM[2]	32 KB					

表 2. S32K3 存储器的特性

特性	S32K314	S32K324	S32K344	S32K328	S32K348	S32K338	S32K358
内核数量	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 存储器 (MB)	4			8			
Data Flash 存储器 (KB)[1]	128						
Cache	I Cache 8 KB D Cache 8 KB			待定			
总 RAM (KB)	512 KB（包括 96 KB 的 TCM）	512 KB（包括 192 KB 的 TCM）		1152 KB（包括 192 KB 的 TCM）		1152 KB（包括 384 KB 的 TCM）	
Standby RAM[2]	32 KB			64 KB			

1. 这是最大可用 Data Flash 存储器。如需了解安装 HSE 安全固件时的应用限制，请参见《S32K3 参考手册》
2. Standby RAM 也包含在总 RAM 中

需要注意的一个重要特性是，S32K3 系列芯片中的所有存储器都具有错误检测和纠错码（ECC）功能。

3. Flash 存储器

S32K3 芯片上的 Flash 存储器是由多个块组成的，最多 5 个块，最少两个块。下表提供了详细信息。

表 3. S32K3 的 Flash 存储器架构

Flash 块	S32K310	S32K311 S32K341	S32K312 S32K322 S32K342	S32K314 S32K324 S32K344	S32K328 S32K338 S32K348 S32K358
UTEST	结束地址 0x1B00_1FFF 8 KB 起始地址 0x1B00_0000	结束地址 0x1B00_1FFF 8 KB 起始地址 0x1B00_0000	结束地址 0x1B00_1FFF 8 KB 起始地址 0x1B00_0000	结束地址 0x1B00_1FFF 8 KB 起始地址 0x1B00_0000	结束地址 0x1B00_1FFF 8 KB 起始地址 0x1B00_0000
Block4 Data Flash 存储器	结束地址 0x1000_FFFF 64 KB 起始地址 0x1000_0000	结束地址 0x1000_FFFF 64 KB 起始地址 0x1000_0000	结束地址 0x1001_FFFF 128 KB 起始地址 0x1000_0000	结束地址 0x1001_FFFF 128 KB 起始地址 0x1000_0000	结束地址 0x1001_FFFF 128 KB 起始地址 0x1000_0000
Block3 Code Flash 存储器 3	NA	NA	NA	结束地址 0x007F_FFFF 1 MB 起始地址 0x0070_0000	结束地址 0x00BF_FFFF 2 MB 起始地址 0x00A0_0000
Block2 Code Flash 存储器 2	NA	NA	NA	结束地址 0x006F_FFFF 1 MB 起始地址 0x0060_0000	结束地址 0x009F_FFFF 2 MB 起始地址 0x0080_0000
Block1 Code Flash 存储器 1	NA	结束地址 0x004F_FFFF 512 KB 起始地址 0x0048_0000	结束地址 0x005F_FFFF 1 MB 起始地址 0x0050_0000	结束地址 0x005F_FFFF 1 MB 起始地址 0x0050_0000	结束地址 0x007F_FFFF 2 MB 起始地址 0x0060_0000
Block0 Code Flash 存储器 0	结束地址 0x0047_FFFF 512 KB 起始地址 0x0040_0000	结束地址 0x0047_FFFF 512 KB 起始地址 0x0040_0000	结束地址 0x004F_FFFF 1 MB 起始地址 0x0040_0000	结束地址 0x004F_FFFF 1 MB 起始地址 0x0040_0000	结束地址 0x005F_FFFF 2 MB 起始地址 0x0040_0000

S32K3 系列芯片的 Flash 存储器容量从 512 KB 到 8 MB 不等。表 3 按 Flash 存储器的大小对 S32K3 系列芯片进行了分类。

Flash 存储器中有一些区域受到保护，可供应用程序内核使用。这些区域仅适用于 HSE_B 内核。如需了解有关 HSE_B 的更多信息，请参见《S32K3 参考手册》。

Flash 存储器有 3 种操作模式。当芯片在用户模式（User mode）下工作时，可以访问 Flash 存储器阵列以执行读取、编程或擦除操作。用户模式是 Flash 存储器的默认工作模式。所有寄存器都具有读写权限。在低功耗模式下，Flash 存储器的电源被关闭，无法访问，在该模式下不允许操作。最后，在 Utest 模式下，可以验证 Flash 存储器的完整性。

Flash 存储器可以通过单次读取、两次读取或四次读取功能在不同块之间执行多次读取，在多核芯片中，如果有多个线程并行运行（在存储器的不同部分/块上），可以同时对这些线程进行两次读取或四次读取，该功能由内部控制，而不是由用户控制。它还具有“边读边写”（RWW）功能，可同时执行读取和写入操作（仅适用于在不同的块中执行操作的情况）；例如，在 S32K324 中，如果 Core 0 应用程序正在块 0 中执行写入操作，与此同时，Core 1 可以读取存储在 Data Flash 块中的数据。

在使用 Flash 存储器时，需要考虑 4 个重要的操作：

- 读取 Flash 存储器
- 锁定和解锁扇区或超级扇区
- 对 Flash 进行编程
- 擦除 Flash 存储器

3.1. 读取

复位后，Flash 存储器处于默认状态，控制器可以读取阵列和寄存器。读取 Flash 会返回 256 位长度的数据，读取寄存器将返回 32 位长度的数据。对于读取操作，不必考虑锁定或解锁扇区。读取操作由 PFlash 控制器执行，该控制器是系统总线和嵌入式 Flash 存储器之间的接口。

3.2. 写入或编程

最小编址单位为 2 个字（64 位），数据必须 64 位对齐。最多可同时编程 4 页，1 页为 8 个字（256 位）。这意味着一次编程操作最多可以更改 1024 位。在执行编程操作或写入操作时，存储器会计算并存储 ECC 位。ECC 按 64 位双字处理。ECC 需要 8 位。

编程操作将位的逻辑值从 1 改为 0，这意味着不允许从 0 改为 1 的编程操作，并且在执行任何编程操作之前需要擦除 Flash 存储器。当 Data Flash 用于 EEPROM 仿真时，恩智浦批准的驱动程序可以在 64 位 ECC 段中进行重复编程（over-programming），允许对同一位置最多进行 3 次重复编程，无需在扇区中执行擦除操作。此功能可用于更改数据记录的记录状态，而无需事先擦除数据。EEPROM 仿真技术经常用到此功能。需要注意，它仅适用于恩智浦批准的驱动程序，请咨询适用于 S32K3 芯片的 RTD 软件，了解可用的 FEE 驱动程序。

在执行编程操作之前，必须先解锁包含指定地址的扇区。对锁定的扇区或超级扇区执行编程操作，操作会失败，并且 MCRS[PEP]位会报告错误。

编程操作的流程图如图 1 所示。

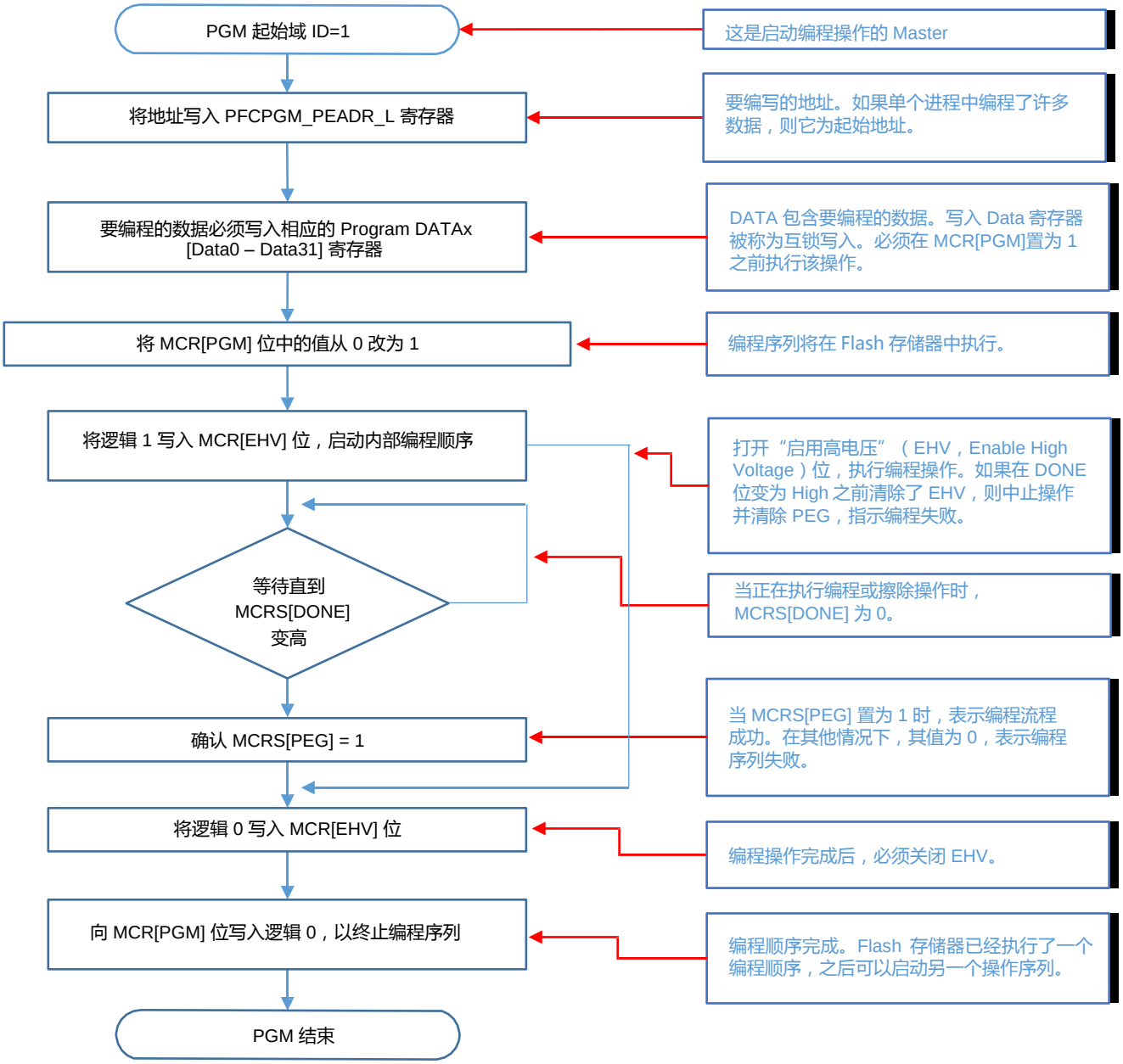


图 1. 编程顺序流程图

3.3. 擦除

擦除操作是指将扇区或块中的所有位设置为 1 的流程。最小擦除单位可以是扇区，扇区大小为 8KB。要擦除扇区或块，必须在擦除操作之前将其解锁。擦除流程也会清除 ECC 位。

下面的流程图展示了擦除顺序。

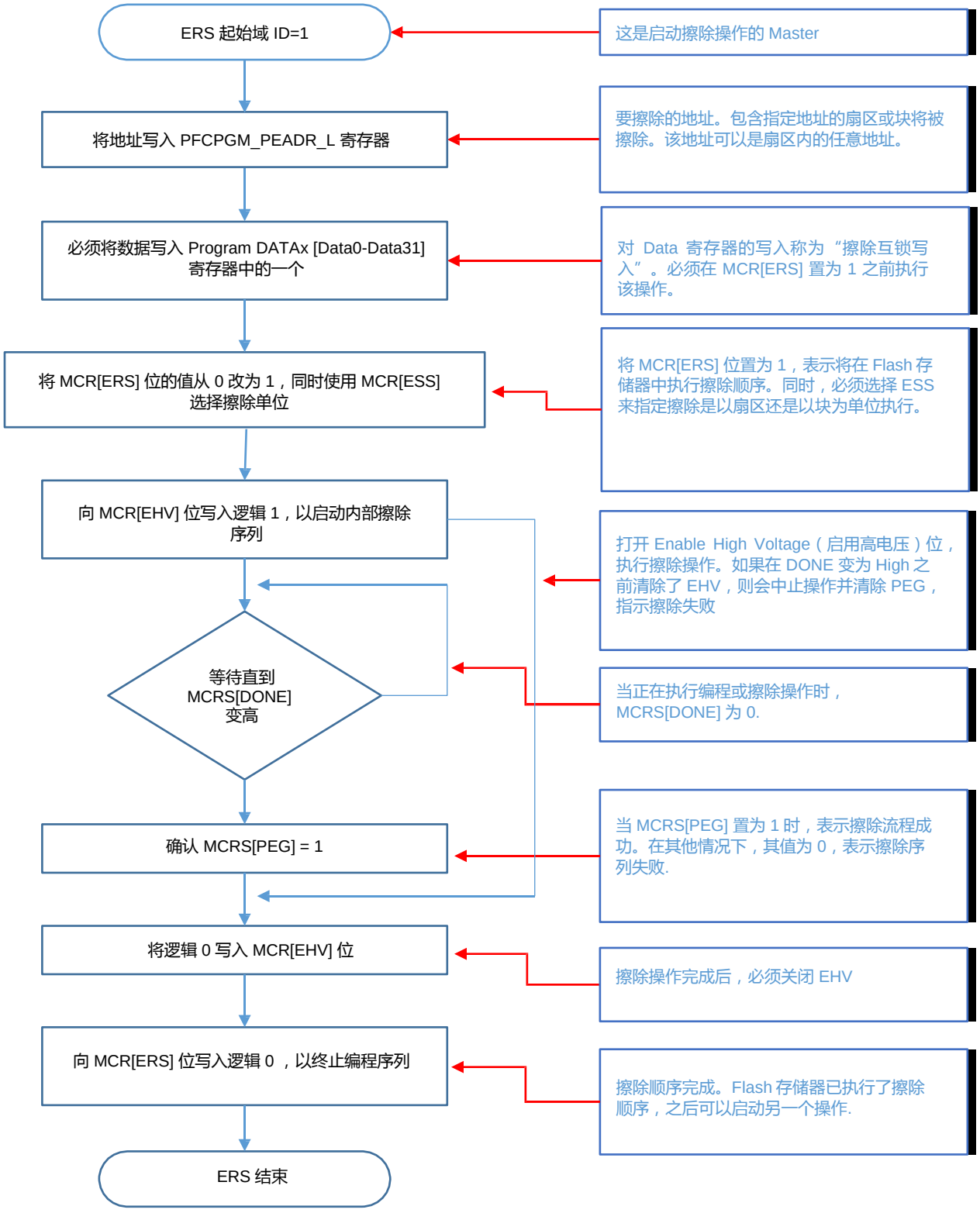


图 2. 擦除顺序流程图

下面的裸机代码展示了写入和擦除操作实施，注意，在示例 3 中，这两个操作共享了一些函数。

示例 1. 擦除 flash 函数

```
tF  \SH_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);
}
```

示例 2. 写入 flash 函数

```
{
  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;
}
```

示例 3. 常见的擦除/写入函数

```
#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. 锁定和解锁扇区或超级扇区

块由大小分别为 8KB 和 64KB 的扇区和超级扇区组成。使用锁定功能可保护这些扇区不受写入或擦除操作的影响。块最后的 256 KB 部分具有扇区保护功能，而其余部分则具有超级扇区保护功能。Data Flash 具有扇区保护功能，UTEST 扇区具有独立的扇区编程保护。S32K3 系列芯片的某些成员由于内存大小原因而无法提供超级扇区保护，如需了解这些产品的更多信息，请参见《S32K3 系列芯片参考手册》。

下图展示了 1MB 块的扇区和超级扇区分布。

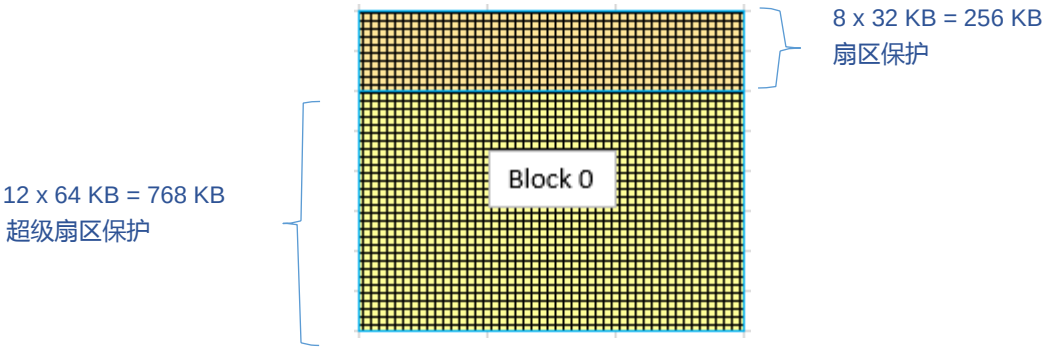


图 3. 1MB 块内的扇区分布

超级扇区的锁定和解锁流程由 PFCBLKn_SSPELOCKn 寄存器控制，扇区的锁定和解锁流程由 PFCBLKn_SPELOCKn 寄存器控制。PFCBLKn_SSPELOCKn 寄存器有 12 个可用位，其中每个位对应于一个超级扇区，同样，PFCBLKn_SPELOCKn 寄存器有 32 个可用位，对应于 32 个可用扇区。如果需要执行解锁流程来允许编程或擦除操作，则应将相应的寄存器 PFCBLKn_SSPELOCKn 或 PFCBLKn_SPELOCKn 从 1 改为 0。向 PFCBLKn_SPELOCKn 或 PFCBLKn_SSPELOCKn 的任何位写入 1 将锁定扇区或超级扇区，以防编程和擦除操作。

PFCBLKn_SSPELOCKn 和 PFCBLKn_SPELOCKn 寄存器所有位的 out of reset 值为 1，这意味着所有扇区在复位后都受到保护，不受编程和擦除操作的影响。

图 4 展示了解锁扇区或超级扇区的步骤。

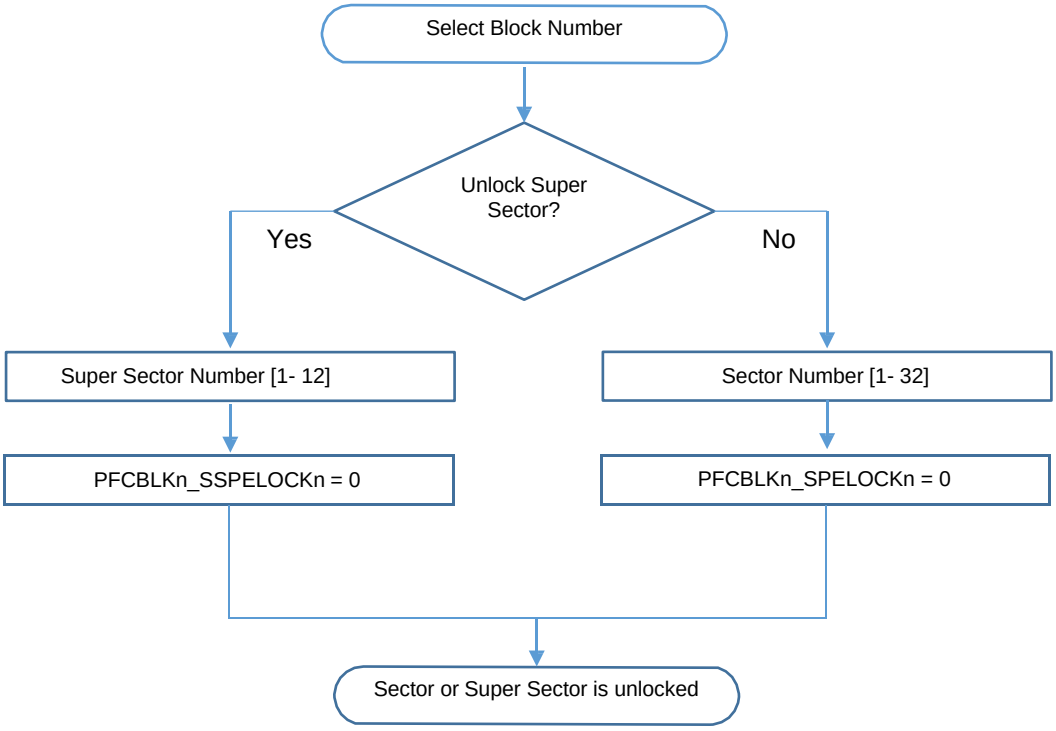


图 4. 扇区或超级扇区的解锁流程

3.5. UTEST 扇区

S32K3 系列芯片的所有成员都具有 8KB UTEST 扇区。在该扇区中，可以存储有关应用程序的重要信息，例如版本号、永久参数、配置（启动或应用）等。在 UTEST 扇区中，有一些区域是为 SoC 保留的，供恩智浦使用。如需了解可用区域的更多详细信息，请参见《S32K3 参考手册》中随附的 S32K3xx_DCF_client.xlsx。

当写入测试模式密封（Test mode seal）时，UTEST 扇区是一个 OTP（一次性可编程）空间。测试模式密封在 UTEST 扇区中分配，为安全起见，它的值被编写为 0x5A4B3C2D，这意味着只有新的数据或配置可以附加在 UTEST 扇区中，不允许擦除。

在 UTEST 扇区中写入数据的流程与在其他块中写入数据的流程相同。解锁流程也是一样的，但不同之处在于，UTEST 扇区有自己的 PFCBLKU_SPELOCK[SLCK] 寄存器，用于在编程操作中锁定或解锁扇区。如前所述，UTEST 扇区是 8KB 扇区，因此 PFCBLKU_SPELOCK 寄存器中只有 1 个位可更改。按照扇区保护逻辑，如果 SLCK 位设置为 0，则可对 UTEST 扇区进行编程操作。

下面的示例是裸机代码的一部分，介绍如何解锁 UTEST 扇区，此示例可用于其他块。

示例 4. 解锁 UTEST 扇区

```
#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. TCM

紧耦合存储器（TCM）是采用 Arm（R）Cortex M7 架构的存储器，其主要特点是与内核有专用连接。该存储器分为 Instruction TCM（I-TCM）和 Data TCM（D-TCM）。在 S32K3 系列芯片中，有两个与 Cortex M7 的专用连接，一个用于 I-TCM，另一个用于 D-TCM。每个 CM7 内核都有一个可用的 ITCM 和 DTCM 存储器，因此 TCM 存储器的大小和地址取决于 S32K3 产品上可用的内核数量及其配置（锁步或解耦），如需了解更多信息，请查看表 4 S32K3 的 RAM 存储器架构。

TCM 是内存映射的，可以通过两类总线进行访问。一种是专用的内核连接总线，另一种是 Backdoor 访问总线（供 AHBS 总线上其他 Master 访问）。每次访问都有一个定义的 TCM 起始地址，结束地址由存储器大小决定，如需了解有关地址的更多信息，请参见表 4。

对于专用的内核访问，I-TCM（Instruction-TCM）使用 64 位总线接口，D-TCM（Data-TCM）使用 32 位总线接口。任何其他的 Master（如 eDMA、其他内核（在解耦配置下）、EMAC 和 HSE 内核）都可以通过 32 位 AHB 接口访问 TCM 存储器。

对于锁步模式下的芯片，备用内核的 TCM 被添加到主内核，以扩大主内核的可访问大小。在解耦模式下，TCM 专用于每个内核。

TCM 可用作系统存储器，也可以用作内核专用存储器。当 TCM 用作多核芯片上的系统存储器时，所有启用的内核和非内核 master 都可以使用禁用的内核的 TCM。为了将禁用内核的 ITCM 和 DTCM 用作系统存储器，需要对寄存器做一些配置。如需了解更多详细信息，请参见《S32K3 系列芯片参考手册》，并参考本章节中的示例代码。

下面的框图介绍了其他 master 访问 TCM 存储器（S32K32x 和 S32K34x 系列芯片）所用的路径。

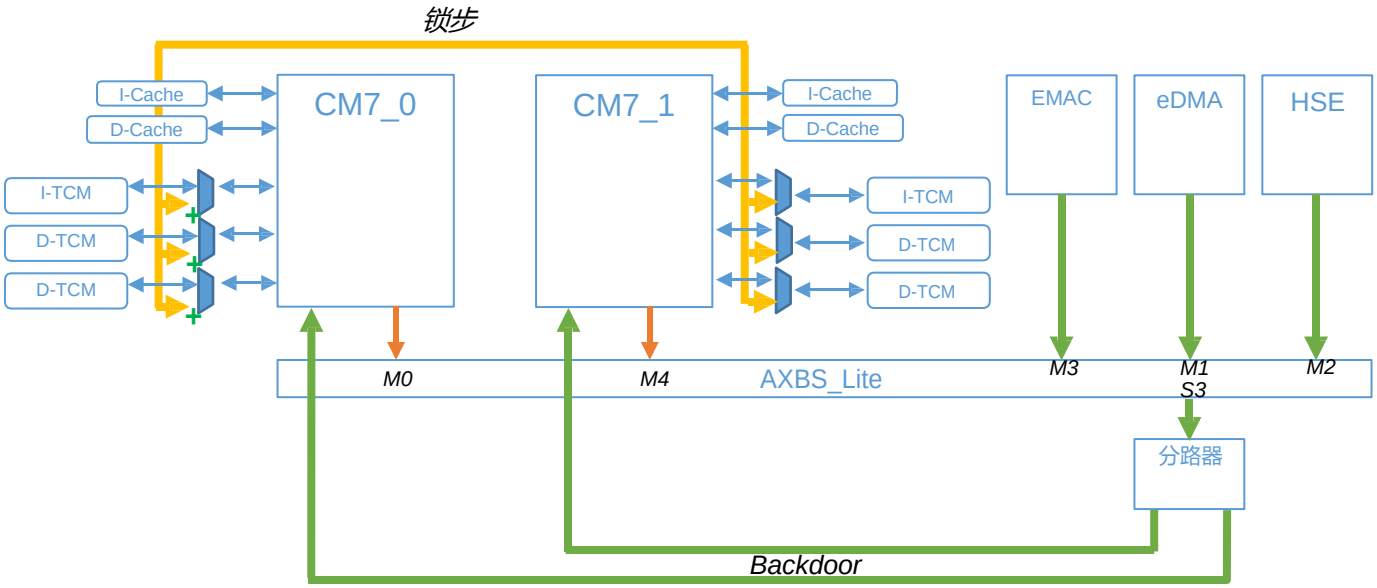


图 5. S32K32x 和 S32K34x 的 TCM 存储器访问（直接访问和 Backdoor 访问）

由于 TCM 位于存储器映射中，并且有一个物理地址，用户可以决定在编译时要存储在 TCM 中的内容。使用 TCM 的一个好处是，它具有确定的访问时间（一个时钟周期），因此 TCM 可用于存储关键数据和代码，例如，频繁更新的变量、中断处理程序、数据处理等。这些数据由用户决定，而不是由作为 Cache 存储器的控制逻辑决定。

与其他存储器一样，TCM 具有 ECC 保护功能，在芯片上电复位后使用前，用户必须对其进行初始化以避免 ECC 错误，在芯片上电复位阶段后，需要进行写入操作来设置初始 ECC 码字。在 S32K3 系列芯片中，内核可以通过直接访问和 Backdoor 访问方式对 ITCM 和 DTCM 进行初始化。此外，DMA 可通过 Backdoor 对 DTCM 进行初始化，但由于 Backdoor 提供 32 位接入，DMA 无法对 ITCM 进行初始化（ITCM 需要 64 位写入以避免 ECC 错误）。

例如，下面的代码描述了如何将 TCM 用作系统存储器，如何通过 M7_0 内核初始化 ITCM，以及如何通过 DMA 初始化 DTCM。重要提示：要初始化 DTCM，DMA 需要使用 Backdoor 地址才能访问 DTCM。

示例 5. TCM 配置为系统 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 ****************************/
```

示例 6. 通过内核进行 ITCM 初始化

```
/********************************** 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 ******************************/
```

示例 7. 通过 eDMA 进行 DTCM 初始化

```
/********************************** 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

S32K3 系列芯片可以包含 1 个到 3 个 SRAM 块。表 4 展示了每个芯片可用的 RAM 存储器块。需要注意：TCM 存储器被视为 RAM 存储器的一部分。

表 4. S32K3 的 RAM 存储器架构

RAM	S32K311 S32K310	S32K312	S32K314	S32K322	S32K324	S32K328	S32K342	S32K344	S32K348	S32K338	S32K358
SRAM2	NA	NA	NA	NA	NA	0x204B_FFFF 256 KB 0x2048_0000	NA	NA	0x204B_FFFF 256 KB 0x2048_0000	0x204B_FFFF 256 KB 0x2048_0000	0x204B_FFFF 256 KB 0x2048_0000
SRAM1	NA	NA	0x2044_FFFF 160 KB 0x2042_8000	NA	0x2044_FFFF 160 KB 0x2042_8000	0x2047_FFFF 256 KB 0x2044_0000	NA	0x2044_FFFF 160 KB 0x2042_8000	0x2047_FFFF 256 KB 0x2044_0000	0x2047_FFFF 256 KB 0x2044_0000	0x2047_FFFF 256 KB 0x2044_0000
SRAM0	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
DTCM2	NA	NA	NA	NA	NA	NA	NA	NA	NA	0x2001_FFFF 128 KB 0x2000_0000	0x2001_FFFF 128 KB 0x2000_0000
DTCM1	NA	NA	NA	0x0000_FFFF 64 KB 0x2000_0000	0x0000_FFFF 64 KB 0x2000_0000	0x0000_FFFF 64 KB 0x2000_0000	NA	NA	NA	0x2000_FFFF 64 KB 0x2000_0000	NA
DTCM0	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
ITCM2	NA	NA	NA	NA	NA	NA	NA	NA	NA	0x0000_FFFF 64 KB 0x0000_0000	0x0000_FFFF 64 KB 0x0000_0000
ITCM1	NA	NA	NA	0x0000_7FFF 32 KB 0x0000_0000	0x0000_7FFF 32 KB 0x0000_0000	0x0000_7FFF 32 KB 0x0000_0000	NA	NA	NA	0x0000_7FFF 32 KB 0x0000_0000	NA
ITCM0	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

每个 SRAM 块都有自己的 SRAM 控制器（PRAMC），支持对内核的快速读/写操作。PRAM 控制器是系统总线和集成的 RAM 阵列之间的接口。系统总线支持 64 位数据，RAM 阵列支持 64 位数据+8 位 ECC。

在 SRAM 存储器中，有一个区域在微控制器处于 Standby 模式（Standby mode）时也可用。SRAM 存储器将前 32 KB 分配给了 Standby SRAM。该区域由电源域 0 供电，因此在 Standby 模式下，存储在 Standby 区域中的信息将被保留。SRAM 存储器的其余部分由电源域 1 供电，仅在运行模式下可用。

与 TCM 存储器一样，所有 SRAM 存储器都必须初始化以避免 ECC 错误，也可以通过内核或 DMA 进行初始化。如果省略了初始化步骤，那么对 SRAM 存储器的任何读取或写入操作都会导致不可纠正的 ECC 错误事件。

示例 8. 通过 DMA 进行 SRAM 初始化

```
/********************************* 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. 读取

无论数据大小，读取事件都可以配置为以零等待状态（zero wait state）或一等待状态（one wait state）的响应作为事件完成的表现。在将数据返回到系统总线之前，PRAM 控制器 Flow Trough Disable field PRCRx[FT-DIS]寄存器在读取事件上插入等待状态。当系统频率大于 120 MHz 时，等待状态的插入非常重要。此等待状态对写入事件没有任何影响。如需了解等待状态的更多信息，请参见《S32K3 参考手册》的“时钟”章节中的 Gasket 配置。

5.2. 写入

写入操作能以 64 位或更少的位进行。当执行对齐 64 位写入时，写入操作以零等待状态在单相周期中执行。当执行小于 64 位的写入或未对齐写入时，将执行读取-修改-写入（RMW）操作，以执行写入并重新计算新的 ECC 代码。RMW 操作将向写入流程插入一些周期，这意味着对于 SRAM 存储器执行对齐 64 位写入比未对齐的小于 64 位的写入的性能更好。

读取-修改-写入可按照以下顺序进行解释：

1. PRAMC 在相应的读取数据中执行单次纠错（SEC）/两次错误检测（DED）。
2. 写入数据与之前的读取数据合并，只更改写入数据的位。
3. 根据新的 64 位生成新的 ECC 代码。
4. 新的双字和 ECC 被写入 RAM。

6. 用例

6.1. Flash、TCM、SRAM 存储器的比较

S32K3 系列芯片的一个主要特性是 TCM 存储器实施。TCM 存储器的一个优势是，处理任务或使用该存储器中存储的数据的时间是确定的，以避免在内核与 SRAM 或 Flash 存储器之间传输数据时出现任何延迟。

下面的用例演示了从 Flash 运行代码与从 TCM 和 RAM 存储器运行代码的主要区别。此外，该示例帮助客户了解如何运行加载到 TCM 或 RAM 而不是 Flash 存储器中的函数或数据。该用例是为 S32K344 创建的，并在 S32K3XXEVB-Q257 评估板上进行了测试，本章节的以下指南适用于带有 GCC 工具链的 S32 Design Studio，但其他 IDE 和工具链的过程也类似。

6.1.1. Linker 文件

第一步是检查我们的 Linker 文件是否声明了 TCM、Flash 和 SRAM 存储器的区域及其相应的大小。

Linker 文件上的存储器区域应如下所示：

示例 9. Linker 中的存储器定义和大小

```
/************************* 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
}
```

我们需要在每个存储器区域中定义一些部分，用于加载函数或数据。对于该示例，我们将定义以下部分：

- ITCM_code (0x0000_0000)：ITCM 中用于加载和运行 TCM_Function 的部分。
- DTCM_data (0x2000_0000)：DTCM 中用于加载一些阵列和变量的部分。
- SRAM_Function 将使用 SRAM 中的 standard .data 部分（起始地址为 0x2040_8000）。
- Flash 函数将默认加载到 .text 部分。

Linker 文件将如下所示：

示例 10. ITCM、DTCM 和 SRAM 的代码和数据部分

```
__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. 启动

为了避免 ECC 错误，ITCM、DTCM 和 SRAM 进行了初始化。此外，需要将函数代码复制到 ITCM 和 SRAM。该流程根据 Linker 文件进行，并由 startup_code 进行处理。

示例 11. 根据 Linker 文件中指定的表对 ITCM、DTCM 和 SRAM 进行初始化以避免 ECC 错误

```
  /**
    \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;
    }
  }
```

启动时也会处理将代码和数据从 Flash 复制到 ITCM 和 SRAM 的流程。

示例 12. 将代码复制到 ITCM、DTCM 和 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. 分配代码

函数属性部分用于那些要分配给 SRAM、ITCM 或 DTCM 的函数和数据

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

其中，Section_Name 是之前在 Linker 文件中声明的部分。

在函数或变量之前使用该属性。

示例 13. 使用属性部分在 ITCM、DTCM 和 SRAM 中分配函数

```
__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. 结果

表 5 展示了不同存储器上函数和数据所占用的周期数。需要注意，与从 Flash 运行相比，从 ITCM 运行时的周期数更小。这一优势可用于为需要确定时间或需要避免延迟的任务减少周期数和计时。表 5 还展示了启用 Cache、Instruction Cache 和 Data Cache 的用法。当启用 D_Cache 和 I_Cache 时，周期数会显著减少。这是因为 Cache 获取了其区域中的所有代码和数据。S32K3 系列芯片的 Flash 和 SRAM 可缓存，TCM 存储器为零等待访问，因此 ITCM 和 DTCM 不需要 Cache。这是一个简单的代码，Cache 足以存储所有代码。然而，在实际应用中，不太可能将全部代码放在 Cache 中，因此可以利用 TCM 的特点。

表 5. 函数和数据在不同存储器上占用的周期数

	SRAM				DTCM			
	清理 2 个阵列缓冲区（大小为 1024 字节）	将阵列从 Flash 复制到 SRAM	将阵列从 SRAM 复制到 SRAM	总计（清理 + 从 Flash 复制到 SRAM +从 SRAM 复制到 SRAM）	清理 2 个阵列缓冲区（大小为 1024 字	将阵列从 Flash 复制到 DTCM	将阵列从 DTCM 复制到 DTCM	总计（清理 + 从 Flash 复制到 DTCM + 从 DTCM 复制到 DTCM）
	复制 1024 字节 – 关闭 Cache，不进行优化							
从 Flash 运行函数	28780	36829	30366	95975	27510	36794	24351	88655
从 SRAM 运行函数	28956	36682	32398	98036	27505	36746	27718	91969
从 ITCM 运行函数	28838	33793	29687	92318	**27492**	**33817**	**24300**	**85609**
	复制 1024 字节 - 启用 I Cache，不进行优化							
代码被提取到 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**
	复制 1024 字节 - 启用 I Cache 和 D Cache，不进行优化							
代码和数据被提取到 I-Cache 和 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

当从 ITCM 运行代码并将数据传输到 DTCM 时，我们得到的结果与 I-Cache 运行类似。这是因为 TCM 和内核的周期次数与 Cache 相同。

6.2. SRAM Standby

当 MCU 处于 Standby 模式时，SRAM 内存中有 32 KB 可用。此存储器区域允许存储 MCU 从 Standby 模式唤醒时需要的关键数据或重要代码。MCU 有两种工作模式：运行模式和 Standby 模式。在运行模式下运行时，MCU 可使用所有外设和模块。在 Standby 模式下，MCU 只能使用其中的几个，因为一些电源域与芯片内的电源断开了。其目的是实现低功耗模式，以节省能耗，并在事件、上电复位、破坏性复位或功能性复位时唤醒。

下一个用例是一个简单的实践，展示了 Standby SRAM 存储器的特性。

6.2.1. Linker 文件

首先，需要在 Linker 文件中声明 SRAM Standby 区域，请参考示例 9 Linker 中的存储器定义和大小。

如前所述，Standby SRAM 分配给了 SRAM 存储器的前 32KB。

6.2.2. Main 文件

在本示例中，我们将使用 3 个缓冲区：

- dummy_array[1024]：这是一个位于 Flash 中的阵列，大小为 1024 字节。

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

- SRAM_buffer[BUFFER_SIZE]：这是一个位于 SRAM 存储器中的阵列，在 0x20408000 地址之后的任何区域。

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

- SRAM_SB_buffer[BUFFER_SIZE]：这是一个位于 Standby 存储器中的阵列

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

其中 BUFFER_SIZE 被定义为 1024

```
#define BUFFER_SIZE       1024
```

在 Main 文件中，阵列缓冲区被清除，dummy 阵列字节被复制到 SRAM_buffer[]和 SRAM_SB_buffer[]中。

示例 14. 清除并将数据复制到 Standby SRAM 阵列和 SRAM 阵列

```
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[]

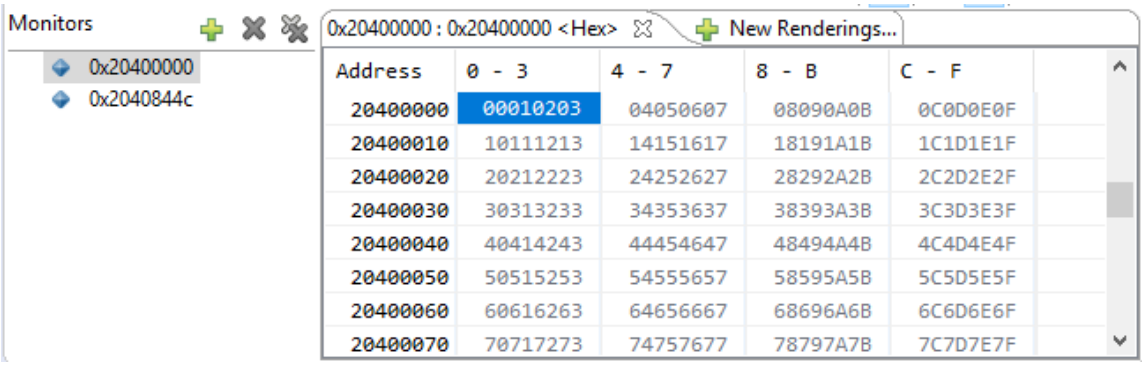


图 6. Standby SRAM 映射存储器

SRAM_buffer[ ]

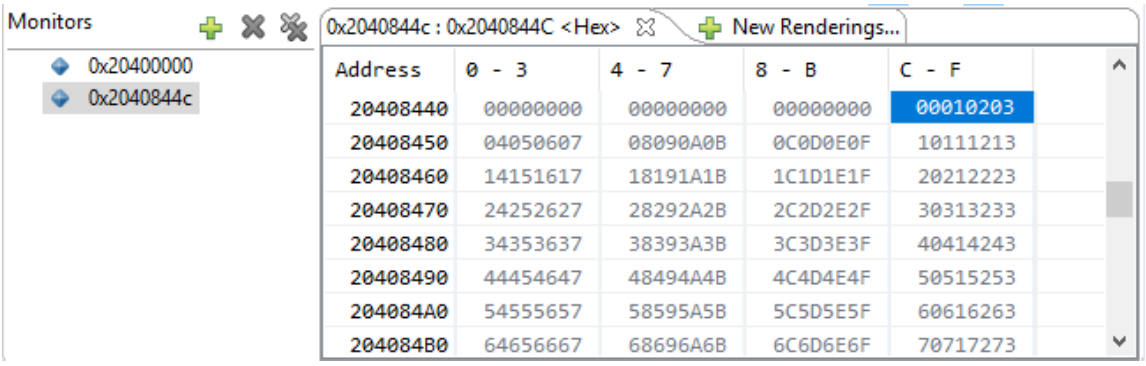
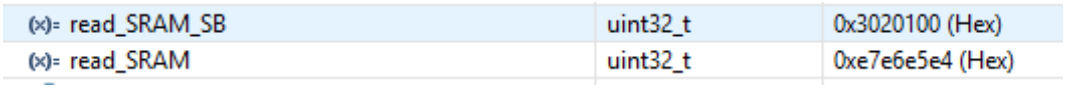


图 7. SRAM 映射存储器

之后执行 dummy 读取，以验证数据是否可从 Standby SRAM 和 SRAM 存储器中获取。

示例 15. 伪读取到 Standby SRAM 阵列和 SRAM 阵列

```
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)

图 8. 通过伪读取方式读取数据

在中断处理程序中按下 SW4，执行 Standby 输入序列。

示例 16. SW4 配置生成中断，进入 Standby 状态

```
/* 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] */
```

示例 17. 中断例程，进入 Standby 模式

```
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();
}
```

执行了 Standby_Entry_Sequence 后，S32K3 进入 Standby 模式，调试器被断开。

6.2.3. 唤醒

按下被配置为产生中断命令的 SW5 执行唤醒流程。

示例 18. SW5 配置，用于从 Standby 模式下唤醒

```
/* 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 */
```

唤醒后，MCU 可重新连接到调试器，在 S32DS 中选择以下标注的选项。

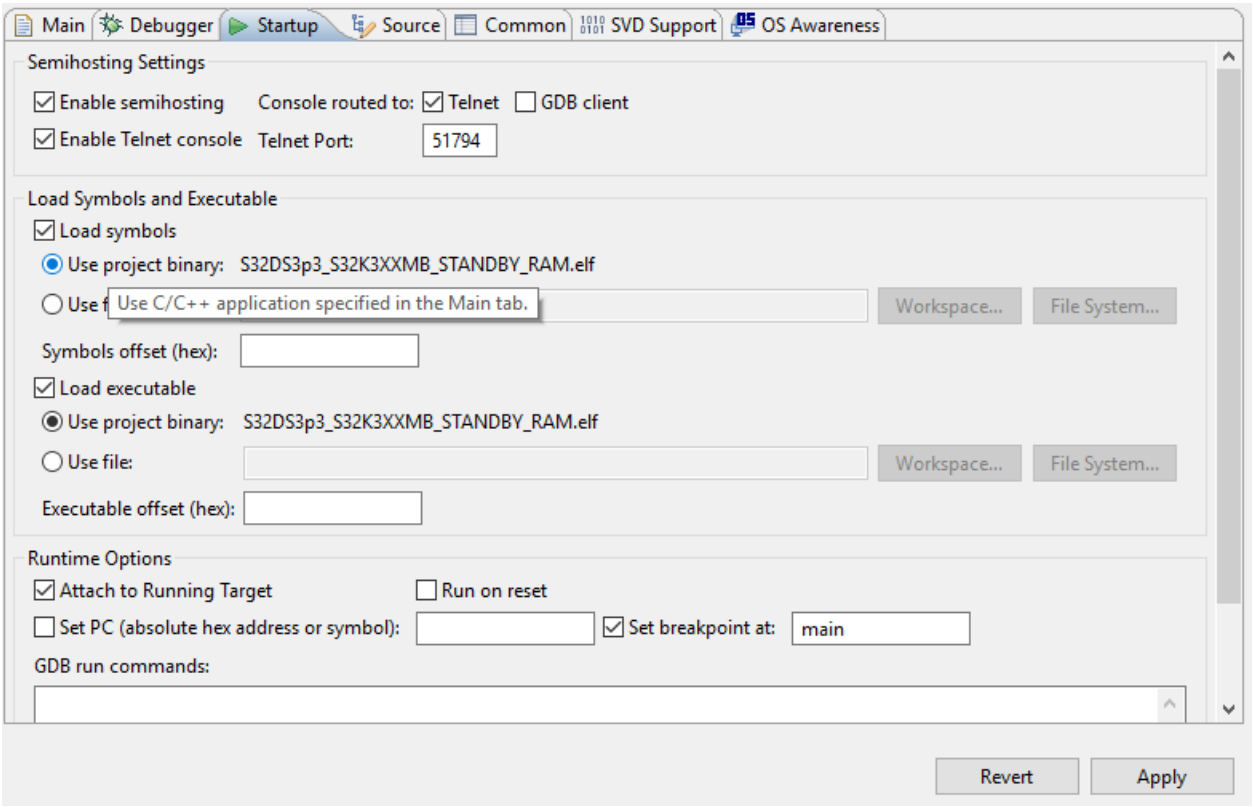


图 9. 将调试器重新连接到当前代码位置的选项

中断处理程序如示例 19 所示。

示例 19. 唤醒后参加中断例程

```
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);
}
```

其中

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

图 10. 通过 dummy 读取方式读取数据

SRAM_SB_buffer[]

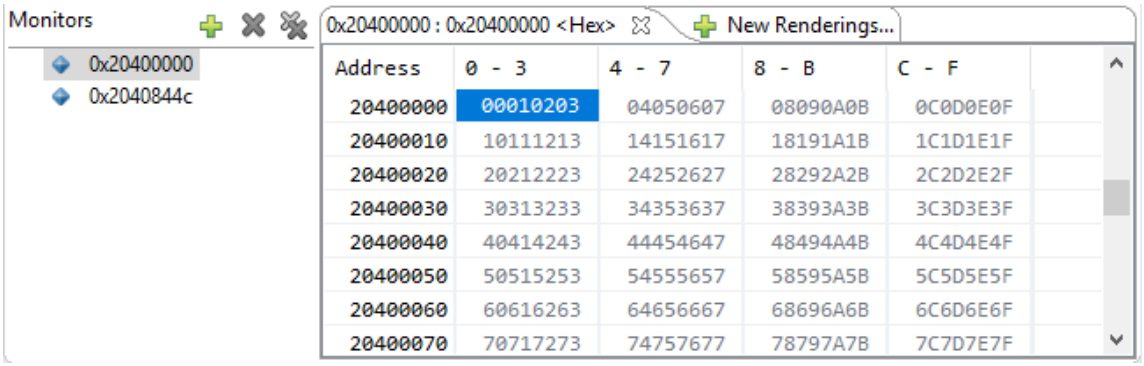


图 11. Standby SRAM 映射存储器

SRAM_buffer[]

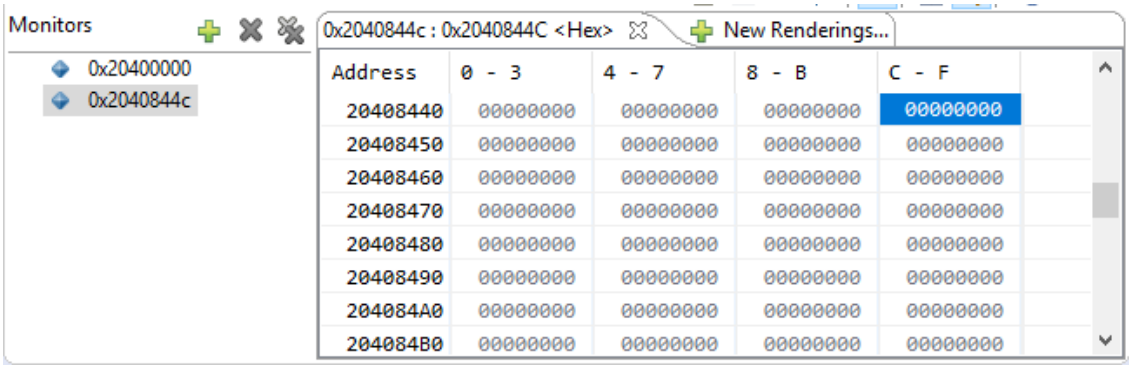


图 12. SRAM 映射存储器

6.2.4. 结果

如上所述，当 MCU 处于 Standby 模式时，存储在由 Standby（Standby）域供电的 Standby SRAM 存储器中的数据被保留，在唤醒后可用。但运行（Run）域供电的 SRAM 中的数据不可用，需要在唤醒后进行初始化，以避免 ECC 错误。需要注意，唤醒后，Standby SRAM 不需要进行初始化以避免 ECC 错误，但 SRAM 的其余部分却需要进行初始化，因此应在启动（Startup）代码中进行适当的区分。下面的代码描述了进行这种区分的示例。

示例 20. 仅在 POR 后对 Standby RAM 进行初始化

```
/* 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. 软件建议和结论

正确使用 S32K3 系列芯片中的不同存储器可以提高应用程序的性能。用户需要评估应将应用程序的哪些函数和数据存储在哪些存储器中，以获得最佳性能。

要在应用中获得良好性能，须遵循 3 个重要建议：

- 应考虑在 ITCM 和 DTCM 内执行确定性任务，因为内核不需要访问其他存储器，从而节省大量的时间。
- 启用 Cache 对应用程序有很大的优势，因为它可以快速获取内核需要的代码和数据，可以在代码的特定位置启用和关闭 Cache，以便更好地控制需要获取哪些代码或数据。要注意，在获取新的代码或数据之前，需要先使 Cache 失效，遗憾的是 Cache 很小，但我们可以使用紧耦合存储器来弥补 Cache 空间的不足。
- 在不同应用中还可以利用 Standby SRAM 的优势，因为在我们的应用程序从 Standby 模式唤醒后，可以使用 Standby SRAM 中存储的数据而无需执行任何其他操作。举例来说，当外部通信将 MCU 从 Standby 模式唤醒后可发送存储的数据。

8. 参考资料

- S32K3xx 参考手册
- S32K3xx 数据手册
- 客户评估板 S32K3XXEVB-Q257
- 面向 Cortex-M 的汽车软件 S32K3 实时驱动程序
- S32 Design Studio IDE

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN13388
Rev. 0
11/2021