Application Note

MSPM0 系列中的闪存多组功能

Hao Mengzhen

摘要

随着对具有严格实时要求的汽车系统等应用需求的不断增长，在完成闪存存储器擦除或编程操作之前，获取关于系统功能的关键信息变得至关重要。应用需要升级固件，尤其是在系统于更新过程中发生断电的情况下。这可能会导致许多问题，例如传输错误或信息丢失。基于这些原因，TI 推出了具备双组闪存存储器的 MSPM0 MCU，专为满足以上需求而设计。双组闪存存储器允许在一个存储体中执行代码，同时在另一个存储体中进行擦除或编程操作。这样可以避免编程操作期间 CPU 停滞，并保护系统免受电源故障或其他错误的影响。本《应用手册》概述了 MSPM0L、MSPM0G 系列的闪存存储器双组功能，并展示了如何在客户项目中实现存储体交换功能。

内容

1 非易失性存储器 (NVM) 基本介绍 2
1.1 闪存保护 4
2 客户安全代码 (CSC) 简介 5
2.1 客户安全代码 (CSC) 执行概述 5
2.2 CSC 内存映射 7
2.3 客户安全代码 (CSC) 执行程序 7
3 存储体交换示例实现 9
3.1 客户安全代码项目准备 9
3.2 应用代码项目准备 12
4 常见用例介绍 14
5 DATA 存储体简介 15
5.1 数据存储体保护 15
5.2 DATA 存储体擦除写入操作 15
6 总结 16
7 参考资料 16

商标

所有商标均为其各自所有者的财产。

1 非易失性存储器 (NVM) 基本介绍

非易失性存储器系统提供了片上可编程闪存存储器，用于存储可执行代码和数据、器件引导配置以及 TI 在工厂预编程的参数。NVM 分为一个或多个存储体，每个存储体中的存储器进一步映射到一个或多个逻辑存储区域和系统地址空间，以供应用使用。“表 1-1”定义了重要的闪存存储体术语，用作本《应用手册》其余部分的参考。

表 1-1. NVM 系统术语

术语	定义	尺寸
闪存字	闪存编程和读取操作的基本数据大小（也是针对系统的读取总线宽度）	64 个数据位（含 ECC 为 72 位）
字线	扇区内的一组闪存字，在扇区擦除前具有最大编程操作限制	16 个闪存字（128 个数据字节，可选 16 个 ECC 字节）
扇区	一起擦除的一组字线（闪存的最小擦除分辨率）	8 个字线（1024 个数据字节，可选 128 个 ECC 字节）
存储体	在一次操作中可以批量擦除的一组扇区。在同一时间针对给定的存储体只能进行一个读取、编程、擦除或验证操作。	变量

MSPM0 器件上的 NVM 系统支持多达 5 个闪存存储体（编号为 BANK0 到 BANK4）。存在的闪存存储体数量取决于器件。要确定特定器件的存储体方案，请查看特定器件数据表中“闪存存储器”章节的详细说明部分。

在给定的闪存存储体内，若存在正在进行的编程或擦除操作，则该操作会暂停对该存储体的所有读取请求，直到操作完成并且闪存控制器已经释放了对存储体的控制。在具有多个闪存存储体的器件上，对其他闪存存储体的读取请求不受不同闪存存储体中操作的影响。因此，在一些应用场景中（如下述场景），若存在多个存储体，则可以提高性能：

- 实时固件更新：应用可以在将新映像写入单独的闪存存储体时，继续从另一个闪存存储体执行代码功能。
- 数据记录和 EEPROM 仿真：应用可以在将数据写入一个单独的存储体的同时继续执行。
- 根据每个组中存储器所支持的功能，每个组中的存储器映射到一个或多个逻辑区域。存在三种区域：FACTORY、NONMAIN（配置 NVM）和 MAIN。一些器件还具有独立的 DATA 存储体，可用于保存数据或进行 EEPROM 仿真。有关 DATA 存储体的更多详细信息，请参阅“节 5”。

表 1-2. 闪存区域

闪存区域	区域内容	可执行	使用者	编程者
FACTORY	器件 ID 和其他参数	否	应用	仅限 TI（不可修改）
NONMAIN（配置 NVM）	器件引导配置（BCR 和 BSL）	否	引导 ROM	TI、用户
MAIN（闪存）	应用代码和数据	是	应用	用户

具有单个存储体的器件在 BANK0（唯一存在的存储体）上实现 FACTORY、NONMAIN 和 MAIN 区域，而不提供 DATA 存储体。具有多个存储体的器件也在 BANK0 上实现 FACTORY、NONMAIN 和 MAIN 区域，但是还包括可实现 MAIN 或 DATA 存储体的其他存储体（BANK1 至 BANK4）。

备注

在 NVM 上执行操作

在擦除或重新编程操作过程中，任何中断（例如拔出器件、移除 SWD 跳线、意外触发复位、取消代码下载、IDE 崩溃等）都可能导致设备永久失效，即“变砖”。NONMAIN 配置不当也会导致器件永久锁定。

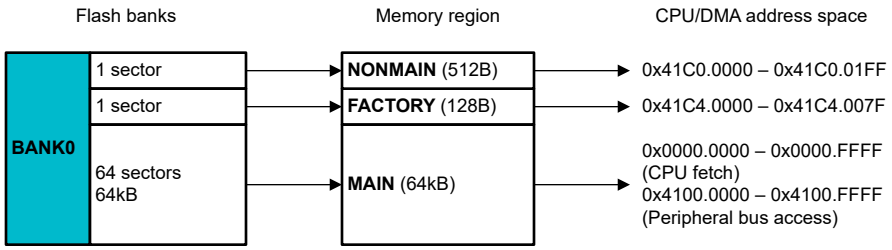

图 1-1. 存储器组织示例 - 单组配置

图 1-2 是一个三存储体配置的示例，其中 512KB 的 MAIN 区域跨 BANK0 和 BANK1 进行拆分，并在 BANK2 中提供一个 16KB 的 DATA 存储体。与单组示例一样，BANK0 中包含 NONMAIN 和 FACTORY 区域。此示例支持在 DATA 存储体中进行 EEPROM 仿真，而不会停止对 MAIN 的取指令操作。它还支持双映像应用，即可以在不影响 BANK1 MAIN 区域取指令的情况下向 BANK0 的 MAIN 区域写入数据，反之亦然。大多数 MAIN 区域大小 ≥256KB 的器件都实现了某种形式的多组配置。

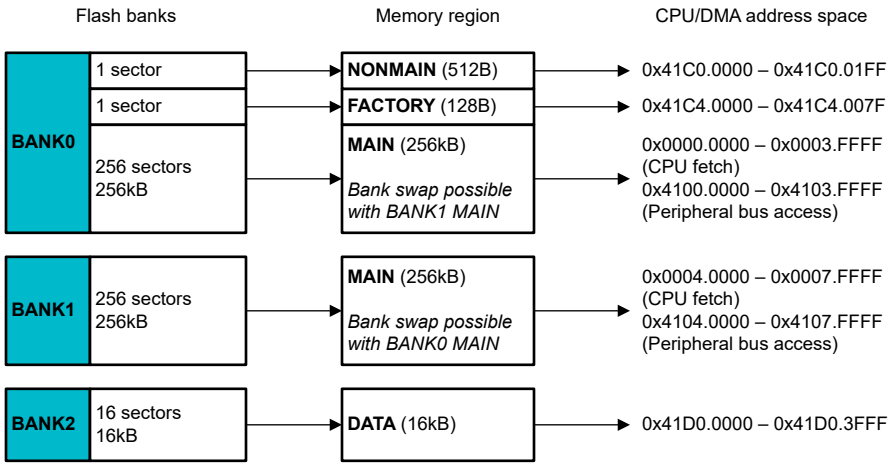

图 1-2. 存储器组织示例 - 多组配置

对于多组器件，BANK0 和 BANK1 被视为物理存储体 0 (PB0) 和物理存储体 1 (PB1)。这些存储体可以根据存储体交换配置切换地址，但在 BOOTRST 后，入口点始终为 PB0。

例如，逻辑存储体 0 (LB0) 始终映射到地址 0x0000.0000 - 0x0003.FFFF；逻辑存储体 1 (LB1) 则映射到 0x0004.0000 - 0x0007.FFFF。这两个逻辑存储体可以映射到 PB0 或 PB1。

后续章节将重点介绍如何利用多组特性来满足应用要求。

1.1 闪存保护

为了满足多种安全要求，有多种类型的闪存存储器保护机制可供使用。

表 1-3. MSPM0 双组器件上的存储器保护机制

存储器保护	说明
存储体交换	在双组或四组器件中，可执行的存储体（或对）会获得读取和执行权限并失去写入或擦除权限。另一个存储体（或对）可读以及可写但不可执行。这种机制强制执行的策略是任何固件更新只能保存在当前会话中的可写存储体中，但绝不能执行。
写保护	1. 由 TI 引导代码强制执行的写保护（NONMAIN 配置）。 2. 由客户安全代码强制执行的写保护，旨在进一步保护应用程序可以更新但不得修改的数据。 3. 存储体交换上下文中的写保护（在“节 3”一节中介绍）。
读取-执行保护	闪存存储器区域可配置为读取-执行保护；根据配置，对该区域的读取和指令取访问将返回错误。CPU、DMA 和调试器访问均以相同的方式处理。
IP 保护	闪存存储器可以配置为读取保护；根据配置，对该区域的读取访问将返回错误，而指令取访问则被允许。CPU、DMA 和调试器访问均以相同的方式处理。
数据存储体保护	可将闪存 DATA 存储体的一个区域配置为读写保护，以阻止读取或写入，或同时阻止这两种类型的访问。CPU、DMA 和调试器访问均以相同的方式处理。

备注

请在《技术参考手册》的“安全”章节中查找每种闪存保护方法的详细操作说明。

2 客户安全代码 (CSC) 简介

客户安全代码 (CSC) 是客户所拥有的软件，用于在执行 BOOTRST 和 SYSRST 操作后配置额外的高级安全设置。该解决方案适用于具有高级安全功能的 MSPM0 系列，例如器件系列 MSPM0Gx51x 和 MSPM0Lx22x 等。TI 在 SDK 中基于公开可用的 MCUboot 提供了一个参考实现，展示了如何使用许多此类附加功能。此属性控制是否提供第二级安全性和基于闪存的可信代码。与 SDK 中的示例（如 customer_secure_image_with_bootloader）搭配使用时，这为设备上的新映像更新和验证提供了完整的解决方案。

备注

客户安全代码并非是将映像放置到设备上的必要条件。在参考实现中，这一过程是由应用程序完成的，因此该过程可以保持更新。

客户安全代码可应用于支持该功能的设备，这些设备可以拥有任意数量的存储体。根据所使用的具体器件和器件上存在的功能，完整的功能集和执行流程会有所不同。通常，客户拥有的安全代码会执行并实现额外的安全功能：

- 存储体互换决策
- 安全固件更新
- 安全密钥存储
- 闪存读取-执行防火墙
- 闪存 IP 防火墙
- SRAM 写入执行互斥

2.1 客户安全代码 (CSC) 执行概述

本节将详细介绍执行流程的总体架构，并通过简化的内存映射来展示两个闪存存储体的使用方式。该器件的执行过程分为两个不同的阶段：*特权*流和*非特权*流。以下流程图使用一条分隔线来区分上部分的特权阶段和下部分的非特权阶段，展示了这两个阶段的区别。

- 发出 INITDONE 标志着从特权状态正式过渡到非特权状态，并启用安全功能。
- 客户安全代码在这两个阶段都会运行，因此同一段代码包含两条执行路径，具体取决于当前状态。
- 特权状态必须发生在非特权状态之前；并且如果没有 BOOTRST，便无法切换回特权状态。
- 这满足某些安全引导文档中所述的一次性可信执行环境 (TEE) 要求。
- 非特权模式启用了额外的功能，例如防火墙和读取或执行存储体策略。
- 应用只会在非特权模式下运行。
- 客户安全代码可以通过读取 INITDONE 是否已发出来确定状态。
- SYSRST 不会切换到特权状态，且安全策略保持不变。

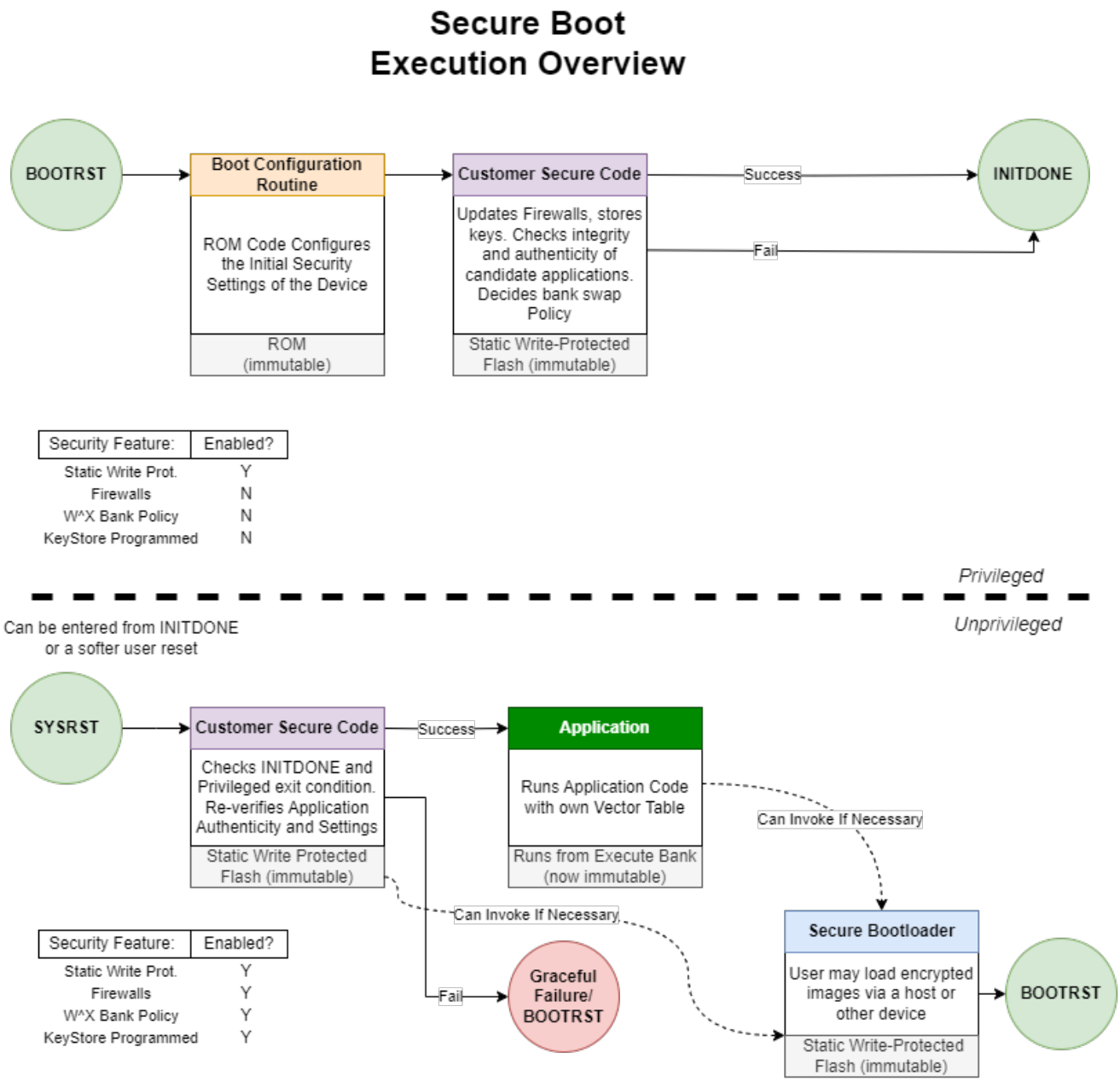

图 2-1. 客户安全代码 (CSC) 执行概述

2.2 CSC 内存映射

下图展示了每个存储体仅包含单个映像插槽的元件存储器映射。下文说明了存储体交换策略。

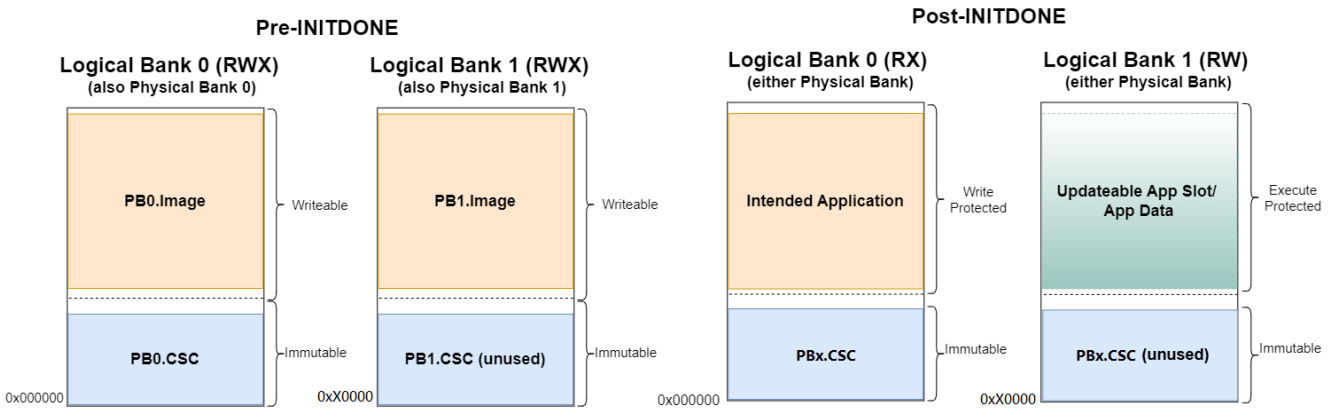

图 2-2. CSC 存储器映射

存储器映射显示存储体交换的基本行为以及 CSC 的执行方式。

在 INITDONE 之前，客户安全代码始终从 PB0 运行。INITDONE 之后，客户安全代码将根据存储体交换的执行情况，从 PB0 或 PB1 中运行。如果执行了存储体交换功能，则 PB1 会重新映射到地址 0x0。

用蓝色表示的客户安全代码在两个存储体中是相同的。这意味着两个 CSC 均被编译为相对于 0x0000.0000 运行，并且两者完全相同，符号重复。无论处于哪种状态，仅需运行 LB0 中的代码。因此，无论哪个物理存储体位于此区域，CSC 都将按预期运行。

2.3 客户安全代码 (CSC) 执行程序

图 2-3 展示了启用安全功能的应用中的引导和启动序列。在 BOOTRST 时，TI 引导代码开始执行。成功引导后，引导代码会发出 BOOTDONE。此时，SYSCTL 向器件发出 SYSRST 以触发从闪存执行。根据引导配置记录，这会触发启动主应用程序（如果此配置中不存在 CSC）或启动 CSC（如果配置了 CSC）。CSC 负责确定执行存储体、内存区域保护、将安全密钥初始化到密钥库等事项。当客户安全代码通过向 SYSCTL.SECCFG.INITDONE MMR 写入值来发出 INITDONE 时，SYSCTL 会发出第二个 SYSRST。器件再次从映射到闪存的 0x0 开始执行，而 CSC 会执行第二次。这一次，CSC 发现 INITDONE 之前已经发出过（这是通过读取 SYSCTL.SECCFG.SECSTATUS.INITDONE 位来确定的），因此会直接调用主应用程序。

备注

请参阅《*MSPM0 L 系列 32MHz 微控制器技术参考手册*》中的“安全”章节，以获取更多关于寄存器配置的详细信息。

安全执行流程是 CSC_EXISTS = YES 情况下的路径。在这种情况下，可以观察到在 BOOTRST 之后，有两个 SYSRST 会在主应用程序启动之前发出。在首个 SYSRST 之后，客户启动代码开始执行。这会配置安全性并发出 INITDONE。此时会锁定并强制执行安全配置。随即会发出第二个 SYSRST 以重新开始执行启动代码。在第二个 SYSRST 时，由于 INITDONE 为 YES，因此会启动主应用程序。

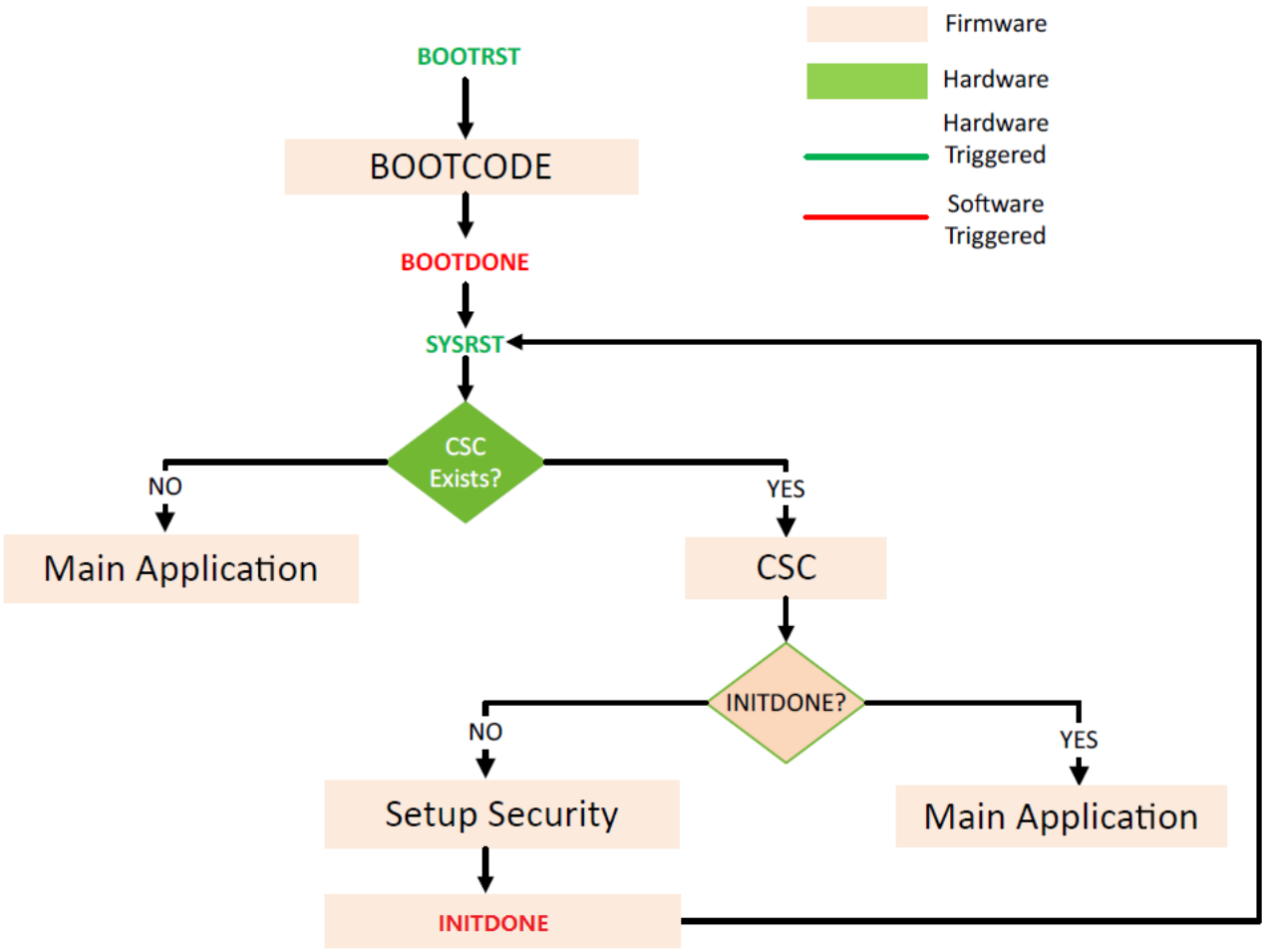

图 2-3. 安全引导和启动序列

以使用存储体交换功能为例，CSC 会指定哪个存储体持有较新的已验证应用程序映像。如果该存储体是物理存储体 0（与 CSC 执行的源存储体相同），则存储体 1 为只读写，不具有执行权限。如果确定了正确的应用程序映像位于物理存储体 1 上，则 CSC 必须发出存储体交换请求。除映像身份验证之外，CSC 还会设置其他安全配置（本文档稍后将介绍这些配置）。CSC 通过向 SYSCTL.SECCFG.INITDONE 寄存器写入 PASS 值 (0x1) 以及 KEY 值 0x9d 来指示 CSC 执行结束。成功向 INITDONE 寄存器写入值会触发第二个 SYSRST 操作，在此期间，存储体交换以及所有其他安全配置都将生效。下一章节将详细介绍如何实现存储体交换功能。

3 存储体交换示例实现

在双组或四组器件中，可执行的存储体（或对）会获得读取和执行权限并失去写入或擦除权限。另一个存储体（或对）可读以及可写但不可执行。这种机制强制执行的策略是任何固件更新只能保存在当前会话中的可写存储体中，但绝不能执行。在发生后续的 BOOTRST 后，客户安全代码会运行，并需要验证更新，决定是否将更新后的映像设置为可执行。如果更新后的映像存在于上部存储体，则将 USEUPPER 配置为 1。如果更新后的映像不存在于上部存储体中，则不需要其他操作。当客户安全代码发出 INITDONE 时，此存储体交换将生效，这会交换上部和下部存储体的执行或写入权限，并将上部存储体（或对）重新映射到闪存存储器地址空间的下半部分，同时将下部存储体（或对）重新映射到闪存存储器地址空间的上半部分。

双组交换功能是客户安全代码功能的一部分，并遵循前一章中所述的客户安全代码执行序列。在应用方面，客户通常需要准备以下项目以实现存储体交换功能。

1. 客户安全代码项目。
2. Bank0/1 中的应用代码项目。

3.1 客户安全代码项目准备

客户安全代码项目分为两个部分。

- NONMAIN 配置
- 客户安全代码应用代码

3.1.1 在 NONMAIN 中启用客户安全代码 (CSC)

从空项目开始。打开 Sysconfig，并在“配置 NVM (NONMAIN)”中勾选“*启用 CSC 策略*”和“*启用闪存存储体交换策略*”的复选框。

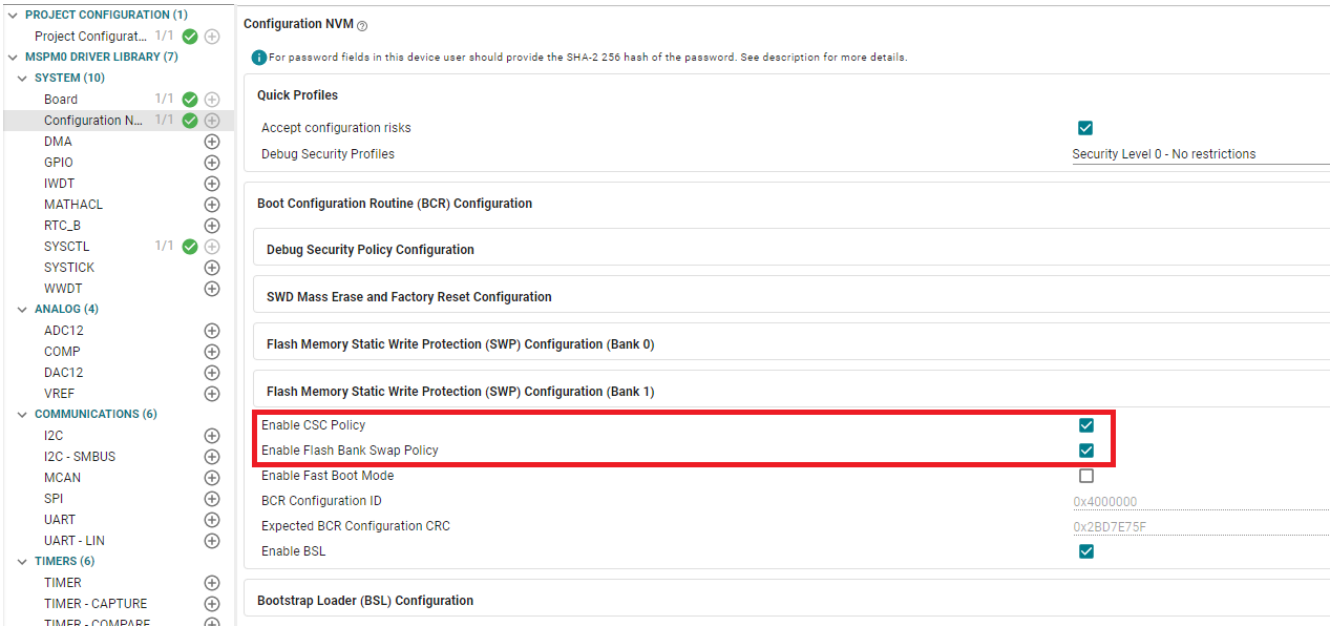

图 3-1. 启用 CSC 策略的 NONMAIN 配置

保存 NONMAIN 配置并使用新的 NONMAIN 信息对 MCU 进行编程。TI 建议单独对 NONMAIN 进行编程。出于安全目的，MSPM0 器件始终需要有效的配置 NVM (NONMAIN)。更新配置非易失性存储器 (NVM) 时，将擦除旧的配置 NVM 配置，并对新的配置进行编程。在擦除或重新编程操作过程中，任何中断（例如拔出器件、移除 SWD 跳线、意外触发复位、取消代码下载、IDE 崩溃等）都可能导致设备永久失效，即“变砖”。配置 NVM 配置不当也会导致器件永久锁定。

通过以下步骤更改擦除配置：在“项目”-> “属性”-> “调试”-> “MSPM0 内存设置”-> “擦除方法”中，选择“*擦除 MAIN 和 NONMAIN 内存*”（参见上述警告）。此设置允许客户使用新配置擦除 NONMAIN 并进行编程。对于其他 IDE 或编程工具，启用擦除 NONMAIN 设置以将 NONMAIN 配置编程到芯片的过程与此类似。

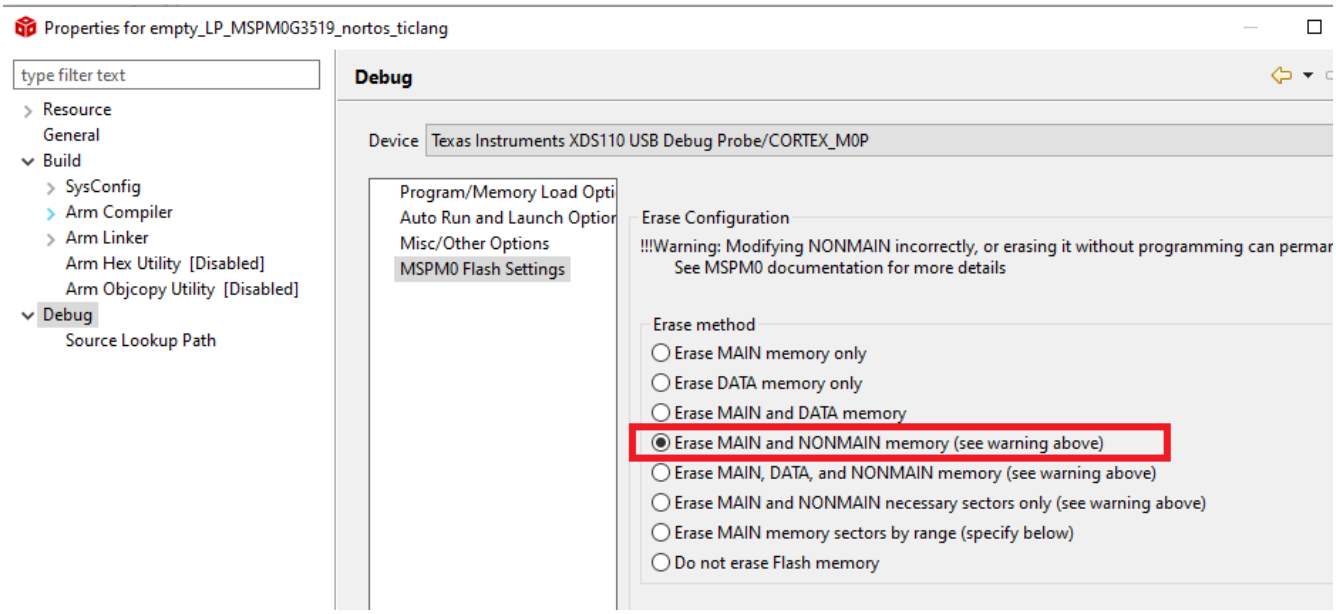

图 3-2. 擦除方法配置

3.1.2 客户安全代码应用代码的实现：存储提及交换功能

客户安全代码需要知道应用映像的位置以执行某些安全功能。因此，首先需要进行的工作是为客户安全代码和应用代码配置闪存存储器的布局。相应地更改链接器文件。链接器文件配置示例如下所示。将闪存存储器分为 FLASH_CSC 和 FLASH_APP 两部分。将 SECTIONS.intvecs 设置为 FLASH_CSC 的起始位置，并将相关部分也分配给 FLASH_CSC。

备注

在可交换存储体的配置中，这些闪存存储器保护会自动镜像到两个存储体。这要求 BANK0 和 BANK1 之间的客户安全代码必须相同，并且应用代码从相同的偏移地址开始。

```
-uinterruptVectors
--stack_size=256
--define=_CSC_SIZE_=(6*1024)
/* Note: SRAM is partitioned into two separate sections SRAM_BANK0 and SRAM_BANK1
to account for SRAM_BANK1 being wiped out upon the device entering any low-power
mode stronger than SLEEP. Thus, this is up to the end-user to enable SRAM_BANK1 for
applications where the memory is considered lost outside of RUN and SLEEP Modes.
*/
MEMORY
{
    FLASH_CSC       (RX)  : origin = 0x00000000,  length = _CSC_SIZE_
    FLASH_APP       (RX)  : origin = _CSC_SIZE_,  length = (0x00040000 - _CSC_SIZE_)
    SRAM_BANK0      (RWX) : origin = 0x20200000,  length = 0x00010000
    SRAM_BANK1      (RWX) : origin = 0x20210000,  length = 0x00010000
    BCR_CONFIG      (R)   : origin = 0x41C00000,  length = 0x000000FF  /*Boot configuration
routine*/
    BSL_CONFIG      (R)   : origin = 0x41C00100,  length = 0x00000080  /*Bootstrap loader*/
    DATA            (R)   : origin = 0x41D00000,  length = 0x00004000
}

SECTIONS
{
    .intvecs      : > 0x00000000
    .text         : palign(8) {} > FLASH_CSC
    .const        : palign(8) {} > FLASH_CSC
    .cinit        : palign(8) {} > FLASH_CSC
    .pinit        : palign(8) {} > FLASH_CSC
    .rodata       : palign(8) {} > FLASH_CSC
    .ARM.exidx    : palign(8) {} > FLASH_CSC
    .init_array   : palign(8) {} > FLASH_CSC
    .binit        : palign(8) {} > FLASH_CSC
    .TI.ramfunc   : load = FLASH_CSC, palign(8), run=SRAM_BANK0, table(BINIT)

    .vtable    :  > SRAM_BANK0
    .args      :  > SRAM_BANK0
    .data      :  > SRAM_BANK0
    .bss       :  > SRAM_BANK0
    .sysmem    :  > SRAM_BANK0
    .TrimTable :  > SRAM_BANK0
    .stack     :  > SRAM_BANK0 (HIGH)

    .BCRConfig  : {} > BCR_CONFIG
    .BSLConfig  : {} > BSL_CONFIG
    .DataBank   : {} > DATA
}
```

以下示例展示了如何根据客户安全代码的启动序列来实现存储体交换功能。软件首先会检查 INITDONE 位。如果尚未发出 INITDONE，请在此处设置安全代码。在这种情况下，首先通过调用 DL_SYSCTL_executeFromUpperFlashBank() 函数来配置存储体交换功能。然后，延迟适当的时间，并通过调用 DL_SYSCTL_issueINITDONE () 函数来发出 INITDONE 位。调用 DL_SYSCTL_issueINITDONE () 函数会触发 SYSRST。在 SYSRST 之后，MCU 在 BANK1 处开始运行。客户还可以在代码中添加存储体交换条件。如果条件匹配，则软件会调用存储体交换函数并运行 BANK1 应用项目。客户可以在代码中添加其他 CSC 函数。别忘了在所有 CSC 函数执行完后发出 INITDONE。系统复位后，如果未启用存储体交换功能，则软件可以通过调用 start_app () 函数来运行位于 BANK0 中的应用项目。

```
int main(void)
{
    SYSCFG_DL_init();

    if (!(DL_SYSCTL_isINITDONEIssued())) {
        if(bankswap == true){                       // Add bank swap conditions if needed
            DL_SYSCTL_executeFromUpperFlashBank(); // Set swap bank0 to bank1
            delay_cycles(160);
            DL_SYSCTL_issueINITDONE();              // Issue INITDONE to trigger System Reset ->
swap to bank1
        }else
        {
                                                    // Add other CSC function if needed
            DL_SYSCTL_issueINITDONE();              // Then issue INITDONE to trigger System Reset
        }
    }else
    {
        start_app((uint32_t *) (0x1800));           // Jump to BANK0 app start address
    }
}
```

下面提供了 start_app() 函数的示例。输入参数是应用的起始地址。链接器文件配置中的存储器映射已更改。在这种情况下，BANK0 中的应用代码从 0x1800 开始。软件会将 SP 值和复位向量复位为应用代码的向量表。然后，将 PC 设置为应用代码的复位处理程序地址。该过程使 MCU 开始运行 BANK0 中的应用代码。

```
static void start_app(uint32_t *vector_table)
{
    /* Reset the SP with the value stored at vector_table[0] */
    __asm volatile(
        "LDR R3,[%[vectab],#0x0] \n"
        "MOV SP, R3       \n" ::[vectab] "r"(vector_table));

    /* Set the Reset Vector to the new vector table (Resets to 0x000) */
    SCB->VTOR = (uint32_t) vector_table;

    /* Jump to the Reset Handler address at vector_table[1] */

    ((void (*)(void))(*(vector_table + 1)))();
}
```

3.2 应用代码项目准备

在此过程中，唯一需要做的是在链接器文件中配置闪存存储器的布局。这样做是为了避免在同一存储体内，应用代码和客户安全代码重叠。请注意，根据客户安全代码政策，两个存储体上的应用代码必须从同一地址开始。以下示例展示了如何为应用代码配置链接器文件。

```
-uinterruptVectors
--stack_size=256
--define=_CSC_SIZE_=(6*1024)

/* Note: SRAM is partitioned into two separate sections SRAM_BANK0 and SRAM_BANK1
 * to account for SRAM_BANK1 being wiped out upon the device entering any low-power
 * mode stronger than SLEEP. Thus, this is up to the end-user to enable SRAM_BANK1 for
 * applications where the memory is considered lost outside of RUN and SLEEP Modes.
 */

MEMORY
{
    FLASH_APP       (RX)  : origin = _CSC_SIZE_, length = (0x00040000 - _CSC_SIZE_)
    SRAM_BANK0      (RWX) : origin = 0x20200000, length = 0x00010000
    SRAM_BANK1      (RWX) : origin = 0x20210000, length = 0x00010000
    DATA            (R)   : origin = 0x41D00000, length = 0x00004000
}

SECTIONS
{
    .intvecs:   > _CSC_SIZE_
    .text   : palign(8) {} > FLASH_APP
    .const  : palign(8) {} > FLASH_APP
    .cinit  : palign(8) {} > FLASH_APP
    .pinit  : palign(8) {} > FLASH_APP
    .rodata : palign(8) {} > FLASH_APP
    .ARM.exidx    : palign(8) {} > FLASH_APP
    .init_array   : palign(8) {} > FLASH_APP
    .binit        : palign(8) {} > FLASH_APP
    .TI.ramfunc   : load = FLASH_APP, palign(8), run=SRAM_BANK0, table(BINIT)

    .vtable :   > SRAM_BANK0
    .args   :   > SRAM_BANK0
    .data   :   > SRAM_BANK0
    .bss    :   > SRAM_BANK0
    .sysmem :   > SRAM_BANK0
    .TrimTable :  > SRAM_BANK0
    .stack  :   > SRAM_BANK0 (HIGH)

    .DataBank   : {} > DATA
}
```

客户可以根据项目要求在应用代码项目中添加其他功能。当应用代码项目准备好进行编程时，请记得更改 IDE 或编程器中的擦除方法，因为该应用项目不包含 NONMAIN 配置。TI 建议使用“*仅擦除 MAIN 存储器*”功能。

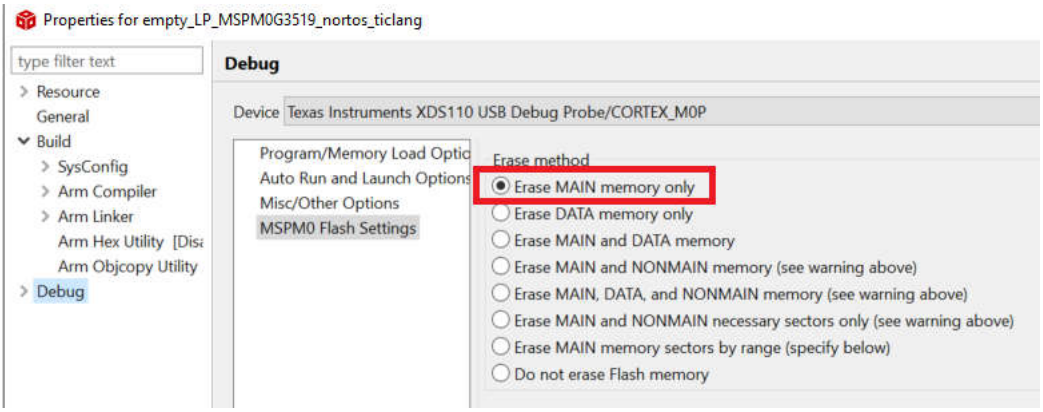

图 3-3. 仅擦除 MAIN 存储器

4 常见用例介绍

双组存储器可以配置并用作具有连续寻址功能的单个大型 NVM 块（只有少数例外情况不在本文档的讨论范围内）。当 NVM 配置为两个并行块时具有显著的优势，最重要的优势是可以在一个存储体中写入而不中断从另一个存储体读取（以及获取指令）。这是在不中断程序 NVM 中代码执行的情况下进行更新的最重要前提。

设计使用双组器件的应用时，有几种选择来决定如何利用程序存储器的第二部分。

该过程也称为实时现场升级，允许用户在不干扰器件正常运行的情况下修改代码和配置。与简单的引导加载器设计相比，这种方式具有更多优势：

- 引导加载器代码备份：使用双组时，可以更新引导代码
- 应用代码备份：如果引导加载失败，原始应用代码仍然起作用
- EEPROM 仿真：应用可以在一个闪存存储体中执行代码，同时使用第二个闪存存储体写入数据，而不会使应用执行停滞

表 4-1 比较了单组和多组器件在闪存操作（擦除和编程）方面的差异。

表 4-1. 同一存储体与不同存储体的闪存操作差异

	在同一存储体上	在不同存储体上
闪存操作（擦除和编程）	在 SRAM 中执行的闪存操作命令。	当在应用代码的同一存储体上运行时。与单组器件相同。
		当在应用代码的不同存储体上运行时。在闪存中执行的闪存操作命令。
闪存操作期间的中断（擦除和编程）	TI 建议在进行闪存操作之前禁用中断或将重要的中断例程移至 SRAM 中。正在进行的编程或擦除操作会暂停对该闪存存储器的所有读取请求，直到操作完成并且闪存控制器已经释放了对存储体的控制。	当在应用代码的同一存储体上运行时。与单组器件相同。
		当在应用代码的不同存储体上运行时。对中断及时做出响应。

5 DATA 存储体简介

在某些 MSPM0 器件（例如 MSPM0Gx51x 系列）中，BANK2 提供了独立的 16KB DATA 存储体。对 DATA 存储体的读取访问通过外设总线进行处理，独立于 BANK0 和 BANK1。

5.1 数据存储体保护

可将闪存 DATA 存储体的一个区域配置为读写保护，以阻止读取或写入，或同时阻止这两种类型的访问。CPU、DMA 和调试器访问均以相同的方式处理。此配置是通过向 SYSCTL.SECCFG.FWPROTMAINDATA 寄存器写入值来实现的。只能以扇区 (1KB) 粒度保护 DATA 存储体的前 4KB。每个扇区可按如下方式配置：

- 0b00：读取/写入均允许。
- 0b01：只读。
- 0b10：禁止读取和写入。
- 0b11：禁止读取和写入 - 未使用。

表 5-1. FWPROTMAINDATA 字段说明

位	字段	类型	复位	说明
31-8	RESERVED	R	0h	
7-6	DATA	R/W	0h	扇区 3 保护配置
5-4	DATA	R/W	0h	扇区 2 保护配置
3-2	DATA	R/W	0h	扇区 1 保护配置
1-0	DATA	R/W	0h	扇区 0 保护配置

5.2 DATA 存储体擦除写入操作

DATA 存储体的擦除和写入操作与 MAIN 区域的操作相同，可以从闪存调用。擦除操作可以在设定的粒度或存储体粒度上进行。这些功能使 DATA 存储体成为数据记录方面的优秀解决方案。有关更详细的说明，请参阅器件技术参考手册。以下示例展示了如何使用 DATA 存储体。

```
/* Address in DATA memory to write to */
#define DATA_BASE_ADDRESS (0x41D00000)
bool status = false;
uint8_t gData8 = 0x11;

    DL_FlashCTL_unprotectSector(
            FLASHCTL, DATA_BASE_ADDRESS, DL_FLASHCTL_REGION_SELECT_MAIN);
    DL_FlashCTL_eraseMemory(
            FLASHCTL, DATA_BASE_ADDRESS, DL_FLASHCTL_COMMAND_SIZE_SECTOR);
    status = DL_FlashCTL_waitForCmdDone(FLASHCTL);
    while(status == false){};

    DL_FlashCTL_unprotectSector(
            FLASHCTL, DATA_BASE_ADDRESS, DL_FLASHCTL_REGION_SELECT_MAIN);
    DL_FlashCTL_programMemory8WithECCGenerated(
            FLASHCTL, DATA_BASE_ADDRESS, &gData8);
```

6 总结

本《应用手册》介绍了 MSPM0 系列中的多组功能，内容包括非易失性存储器简介、客户安全代码简介和数据库简介。值得一提的是，本《应用手册》展示了一种存储体交换的实现方法，旨在帮助客户在其应用中开发此功能。

7 参考资料

- 德州仪器 (TI)，MSPM0L222x、MSPM0L122x 混合信号微控制器数据表
- 德州仪器 (TI)，MSPM0 L 系列 32MHz 微控制器技术参考手册

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265