Application Note

Flash Multi Bank Feature in MSPM0 Family

Hao Mengzhen

ABSTRACT

With the growing demand of applications such as automotive systems which have strict real-time needs, accessing critical information about the system functionality is necessary before the completion of a Flash memory erase or program operation. Applications need a firmware upgrade, especially when the system power loss occurs during the update process. This can result in many problems such as a transmission error or an information loss. For these reasons, TI offers MSPM0 MCUs that embed dual bank Flash memories designed to respond to the above needs. The dual bank Flash memory allows a code to be executed in one bank, while another bank is being erased or programmed. This avoids a CPU stalling during programming operations and protects the system from power failures or other errors. This application note gives an overview of the MSPM0L, MSPM0G Series Flash memory dual bank capabilities, and shows how to implement Bank Swap function on customer project.

Table of Contents

Trademarks

1 Nonvolatile Memory (NVM) Basic Introduction

The nonvolatile memory system provides on-chip programmable flash memory for storing executable code and data, the device boot configuration, and parameters which are preprogrammed by TI from the factory. The NVM is organized into one or more banks, and the memory in each bank is further mapped into one or more logical memory regions and system address space for use by the application. Key flash bank terms are defined in Table 1-1 to be used as a reference for the rest of this application note.

Table 1-1. NVM System Terminology

Term	Definition	Size
Flash word	Basic data size for program and read operations on the flash memory (also the read bus width to the system)	64 data bits (72 bits with ECC)
Word line	Group of flash words within a sector, with maximum program operation limit before sector erase	16 flash words (128 data bytes, optionally 16 ECC bytes)
Sector	Group of word lines that are erased together (minimum erase resolution of the flash memory)	8 word lines (1024 data bytes, optionally 128 ECC bytes)
Bank	Group of sectors that can be mass erased in one operation. Only one read, program, erase, or verify operation can run concurrently on a given bank.	Variable

The NVM system on MSPM0 device provides support for up to 5 flash memory banks (enumerated as BANK0 through BANK4). The number of flash banks present is device dependent. To determine the bank scheme of a particular device, review the detailed description section of the specific device data sheet Flash Memory chapter.

Within a given flash bank, an ongoing program or erase operation stalls all read requests to that bank until the operation has completed and the flash controller has released control of the bank. On devices with more than one flash bank, read request to other flash banks are unaffected by operations in a different flash bank. As such, the presence of multiple banks improves performance in application cases such as:

- Live firmware updates - an application can continue to execute code functions out of one flash bank while a new image is written to a separate flash bank.
- Data Logging and EEPROM emulation: an application can continue to execute while writing data to a separate bank.
- The memory within each bank is mapped to one or more logical regions based upon the functions that the memory in each bank supports. There are three regions: FACTORY, NONMAIN (Configuration NVM) and MAIN. Some devices also have a independent DATA bank which can be used as data saving or EEPROM emulation. The DATA bank is further detailed in Section 5.

Table 1-2. Flash Memory Regions

Flash Memory Region	Region Contents	Executable	Used by	Programmed by
FACTORY	Device ID and other parameters	No	Application	TI only (not modifiable)
NONMAIN (Configuration NVM)	Device boot configuration (BCR and BSL)	No	Boot ROM	TI, User
MAIN (Flash Memory)	Application code and data	Yes	Application	User

Devices with one bank implement the FACTORY, NONMAIN, and MAIN regions on BANK0 (the only bank present), and the DATA bank is not available. Devices with multiple banks also implement FACTORY, NONMAIN, and MAIN regions on BANK0, but include additional banks (BANK1 through BANK4) that can implement MAIN or DATA bank.

Note

Operating on a NVM

Any interruption during the erase or re-program operations (that is, unplugging device, removing SWD jumpers, hitting reset by accident, canceling the code download, an IDE crash, and so forth) can brick the device permanently. Improper configuration of NONMAIN can also lead to permanent locking of the device.

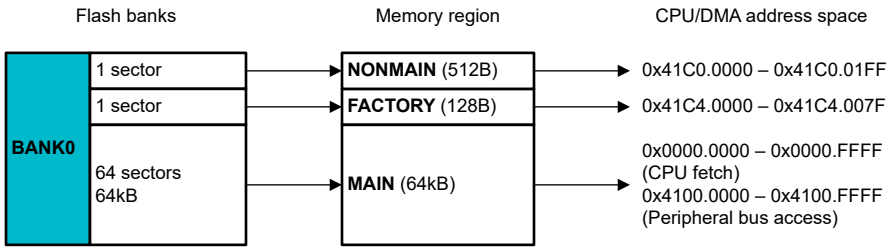

Figure 1-1. Memory Organization Example - Single Bank Configuration

Figure 1-2 is an example of a three bank configuration with a 512KB MAIN region split across BANK0 and BANK1, with a 16KB DATA bank provided in BANK2. Like the single bank example, NONMAIN and FACTORY regions are included in BANK0. This example supports EEPROM emulation in the DATA bank without stalling fetches to MAIN, and also supports dual-image applications where BANK0 main can be written to without stalling fetches to BANK1 main (and the reverse). Most devices with a main region ≥256KB in size implement some form of multibank configuration.

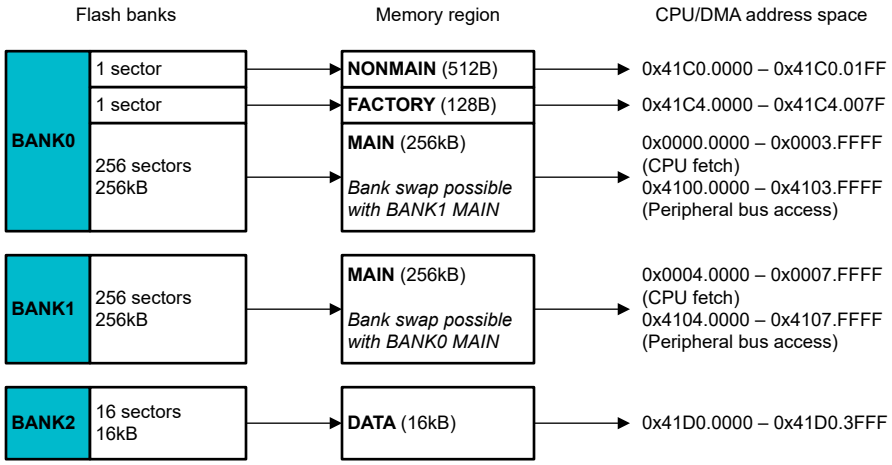

Figure 1-2. Memory Organization Example - Multiple Bank Configuration

For multi bank devices, the BANK0 and BANK1 are considered as Physical Bank 0 (PB0) and Physical Bank 1 (PB1). These banks can switch addresses depending on the bank swap configuration, but upon a BOOTRST the entry point is always PB0.

For example, the Logical Bank 0 (LB0) always maps to address 0x0000.0000 - 0x0003.FFFF, and Logical Bank 1 (LB1) maps to 0x0004.0000 - 0x0007.FFFF. The two logical banks can map to either PB0 or PB1.

Further sections focus on how to take advantage of multiple banks features to achieve application requirements.

1.1 Flash Memory Protection

To meet diverse security requirements, various types of flash memory protection mechanisms are provided.

Table 1-3. Memory Protection Mechanisms on MSPM0 Dual Bank Devices

Memory Protection	Description
Bank Swapping	In dual-bank or quad-bank devices, based on which bank (or pair) is executable, that bank (pair) gets readexecute privileges and loses write or erase privileges. The other bank (or pair) is readable, and writeable but not executable. This mechanism enforces the policy that any firmware update can only be saved in the writeable bank in the current session but can never be executed.
Write Protection	1. Write-protection that is enforced by TI boot-code (NONMAIN configuration). 2. Write-protection that is enforced by CSC to further protect data that is allowed to update but that must not be modified by the application. 3. Write-protection in the context of bank swap (covered in Section 3).
Read-Execute Protection	A region of flash memory can be configured for read-execute protection; read and instruction fetch accesses to this region returns an error. CPU, DMA and debugger accesses are all treated the same way.
IP Protection	A region of flash memory can be configured for read protection; read accesses to this region returns an error while instruction fetch accesses are allowed. CPU, DMA and debugger accesses are all treated the same way.
Data Bank Protection	A region of flash DATA bank can be configured for read-write protection - either reads or writes or both types of accesses can be blocked. CPU, DMA and debugger accesses are all treated the same way.

Note

Please find detailed operation description for each flash protection method in TRM chapter Security.

2 Customer Secure Code (CSC) Introduction

The Customer Secure Code (CSC) is customer-owned software that configures additional advanced security settings after a BOOTRST and SYSRST. This is available on MSPM0 families with advanced security features such as device family MSPM0Gx51x and MSPM0Lx22x and so forth. TI provides a reference implementation in the SDK based on publicly available MCUboot that showcases how to use many of these additional features. This attribute controls whether a second level of security and trusted flash-based code is provisioned or not. When paired with an example such as the customer_secure_image_with_bootloader example in SDK, this represents a full design for updates and verification of new images on the device.

Note

The CSC is not required to put images on the device. In the reference implementation, this is done by the application, which allows for the process to remain updated.

CSC can applied on capable devices that can have any number of banks. The full set of features and execution flow varies depending on the specific device used and features present on the device. Typically, customer owned secure code executes and implements additional security capabilities:

- Bank swap decision
- Secure firmware update
- Secure key storage
- Flash read-execute firewall
- Flash IP firewall
- SRAM write-execute mutual exclusion

2.1 CSC Execution Overview

This section details a high-level understanding of the execution flow and an abstracted memory map showing how the two banks of flash are used. The execution of the device is divided into two distinct phases. A *Privileged* and *Unprivileged* flow. The following flow shows this distinction with the line separating the upper (privileged) and lower (unprivileged) phases.

- The issuing of INITDONE is the official transition from the privileged state to the unprivileged, enabling security.
- The Customer Secure Code is run both times, thus the same body of code contains two execution paths, depending on the state.
- The privileged state must happen before the unprivileged state, and this is not possible to transition back to the privileged state without a BOOTRST.
- This satisfies a one-time Trusted Execution Environment (TEE) as described by some Secure Boot documentation.
- Unprivileged mode has additional activated features, such as firewalls and the Read or Execute bank policy.
- The application is only ever run in the unprivileged mode.
- The state can be determined by the Customer Secure Code by reading whether INITDONE has been issued.
- A SYSRST does not enter the privileged state, and security policies are retained.

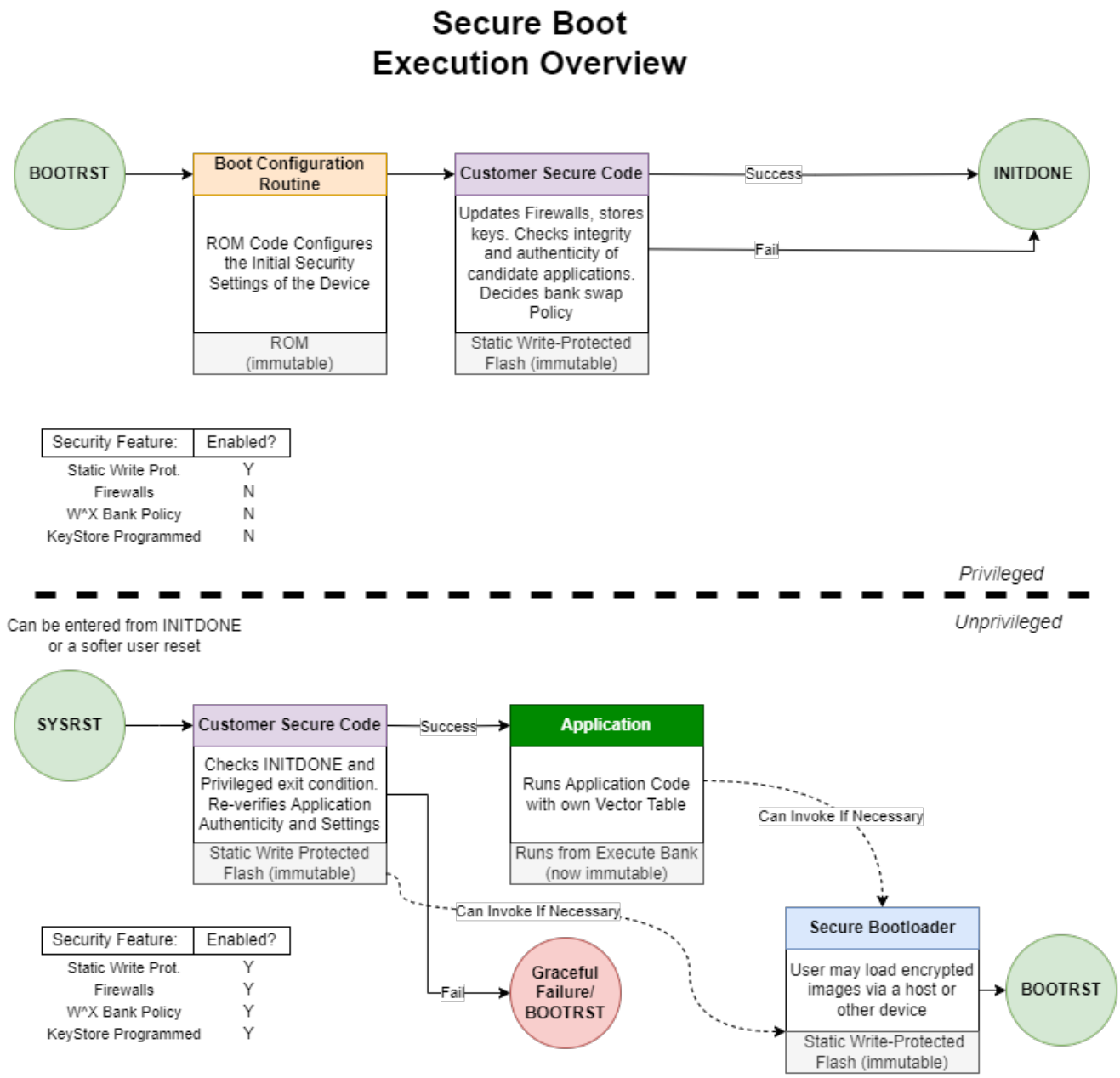

Figure 2-1. CSC Execution Overview

2.2 CSC Memory Maps

A memory map of the components with a single image slot per bank is shown below. The following illustrates the bank swap policy.

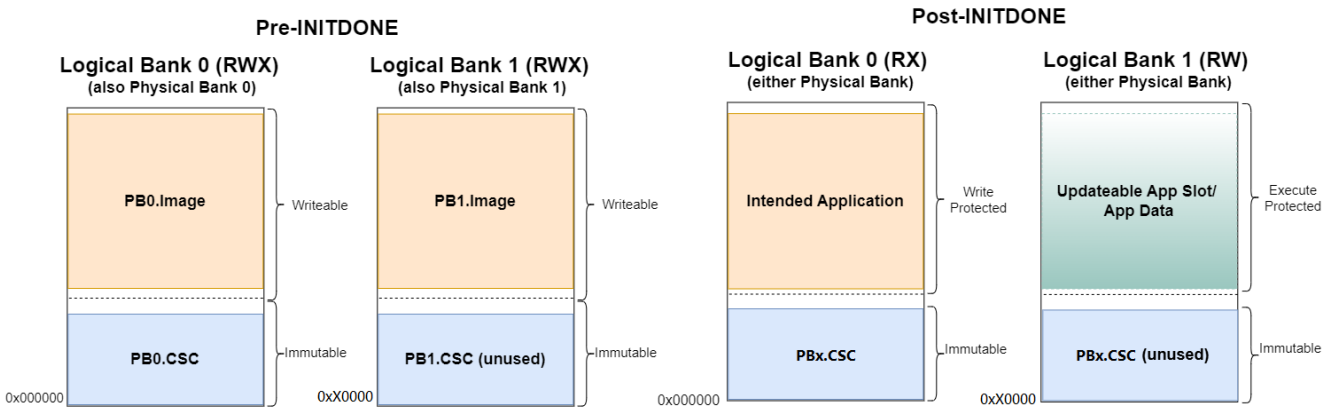

Figure 2-2. CSC Memory Map

The memory map shows the basic behavior of the Bank Swap, and how the CSC is executed.

Prior to INITDONE, the customer secure code always runs from PB0. Post INITDONE, the customer secure code is executed from either PB0 or PB1, depending on the bank swap execution. And, if the bank swap function executed, then the PB1 is re-mapped to address 0x0.

The customer secure code shown in blue is identical across both banks. This means that both CSCs are compiled to run with respect to 0x0000.0000 and are identical, with the symbols being duplicated. At either state, only the code in LB0 needs to be run. Thus, no matter which physical bank is in this region, the CSC runs as expected.

2.3 CSC Execution Process

Figure 2-3 illustrates the boot and startup sequence in security enabled applications. At BOOTRST, TI boot-code execution commences. After successful boot, boot-code issues BOOTDONE. At this point, SYSCTL issues a SYSRST to the device to trigger execution from flash memory. Depending on the boot configuration record, this leads either to the start of the main application (if CSC does not exist in this configuration) or to the start of the CSC (if CSC is configured). CSC is responsible for determining execution bank, memory region protections, secure key initialization into the keystore, and so forth. When the customer secure code issues INITDONE (by writing to SYSCTL.SECCFG.INITDONE MMR), then SYSCTL issues a second SYSRST. The device again starts execution from 0x0 mapped to flash, and the CSC executes a second time. This time, the CSC finds that INITDONE has already been issued previously (this is determined by reading the SYSCTL.SECCFG.SECSTATUS.INITDONE bit) and directly calls the main application.

Note

Please refer to the *Security* chapter in the *MSPM0 L-Series 32MHz Microcontrollers Technical Reference Manual* for further details of register configuration.

The secure execution flow is the path where CSC_EXISTS = YES. In this case, an observation is that after BOOTRST, two SYSRSTs are issued before the main application is launched. After first SYSRST, the customer startup code gets to execute. This configures security and issues INITDONE. At this point, the security configuration is locked and enforced. A second SYSRST is issued at this point, restarting startup code execution. At the second SYSRST, since INITDONE is YES, the main application is launched.

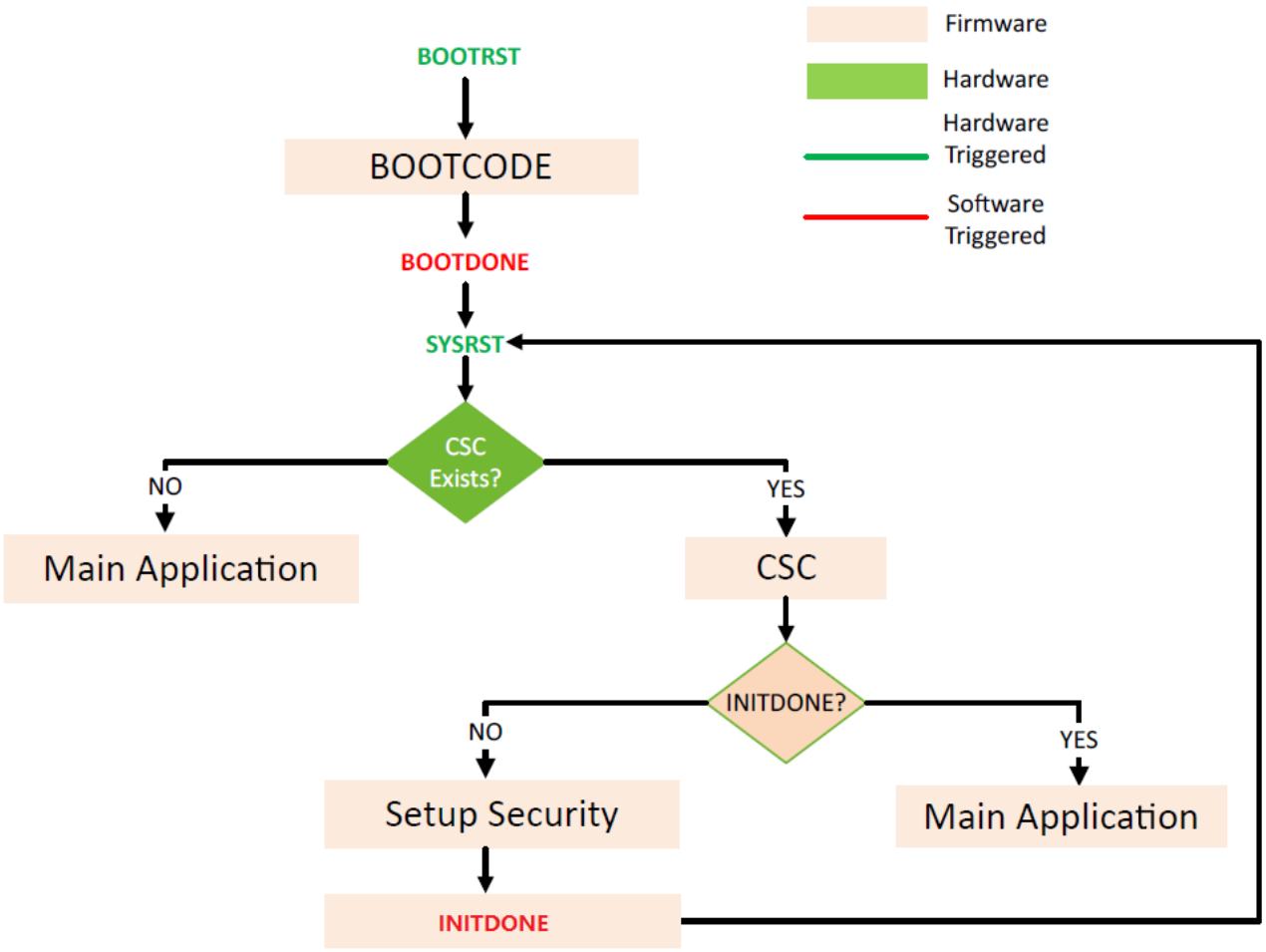

Figure 2-3. Secure Boot and Start Sequence

Use bank swap function for example, CSC specifies which bank holds the more recent authenticated application image. If that bank is physical bank 0 (same bank as where CSC is executing from), then bank 1 is read-write only and does not have execute privilege. If the correct application image is determined to be on physical bank 1, then the CSC must issue a bank-swap request. Besides image authentication, the CSC sets up additional security configurations that are described later in this document. The CSC indicates the end of CSC execution by writing to the SYSCTL.SECCFG.INITDONE register with a PASS value (0x1) along with a KEY value of 0x9d. Successfully writing to the INITDONE register results in a second SYSRST operation during which the bank-swap takes effect, as well as any additional security configurations. The next section describes how to implement bank swap function in details.

3 Bank Swap Example Implementation

In dual-bank or quad-bank devices, based on which bank (or pair) is executable, that bank (pair) gets readexecute privileges and loses write or erase privileges. The other bank (or pair) is readable, and writeable but not executable. This mechanism enforces the policy that any firmware update can only be saved in the writeable bank in the current session but can never be executed. Upon a subsequent BOOTRST, the CSC runs and needs to authenticate the update and decide if the updated image needs to be made executable. If the updated image exists in Upper bank, then configure USEUPPER to 1. If the updated image does not exist in Upper bank, then no other action is needed. When the CSC issues INITDONE, this bank-swap takes effect, swapping the execution or write privileges of the upper and lower banks and re-mapping the upper bank (or pair) to the lower half of the flash memory address space and re-mapping the upper bank (or pair) to the upper half of the flash memory address space.

Dual bank swap function is part of CSC function and follows CSC execution sequence describes in former chapter. On the application side, customers usually need to prepare the projects below to achieve bank swap function.

1. CSC code project.
2. Application code project in Bank0/1.

3.1 CSC Code Project Preparation

There are two parts for the CSC code project.

- NONMAIN configuration
- CSC application code

3.1.1 Enable CSC in NONMAIN

Start with an empty project. Open Sysconfig and check the boxes for *Enable CSC Policy* and *Enable Flash Bank Swap Policy* in Configuration NVM (NONMAIN).

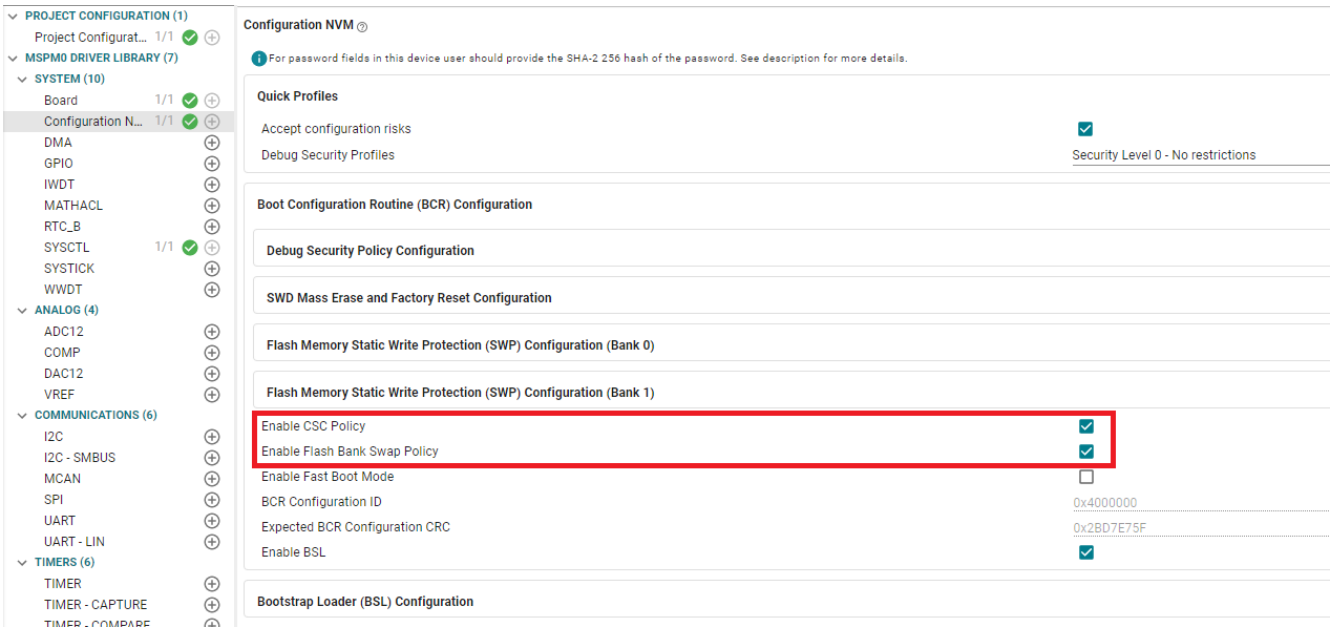

Figure 3-1. NONMAIN Configuration To Enable CSC Policy

Save the NONMAIN configuration and program the MCU with new NONMAIN information. TI recommends to program the NONMAIN alone. For security purposes, MSPM0 devices require a valid Configuration NVM (NONMAIN) at all times. When updating Configuration NVM, the old Configuration NVM configuration is erased, and the new configuration is programmed. Any interruption during the erase or re-program operations (that is, unplugging device, removing SWD jumpers, hitting reset by accident, canceling the code download, an IDE

crash, and so forth) can brick the device permanently. Improper configuration of Configuration NVM can also lead to permanent locking of the device.

Change the erase configuration in project -> Properties -> Debug -> MSPM0 Flash Settings -> Erase method choose *Erase MAIN and NONMAIN memory* (see warning above). This setting allows customers to erase the NONMAIN and program with new configurations. This takes similar steps for other IDE or programmer tools to enable erase NONMAIN setting to program the NONMAIN configuration to the chip.

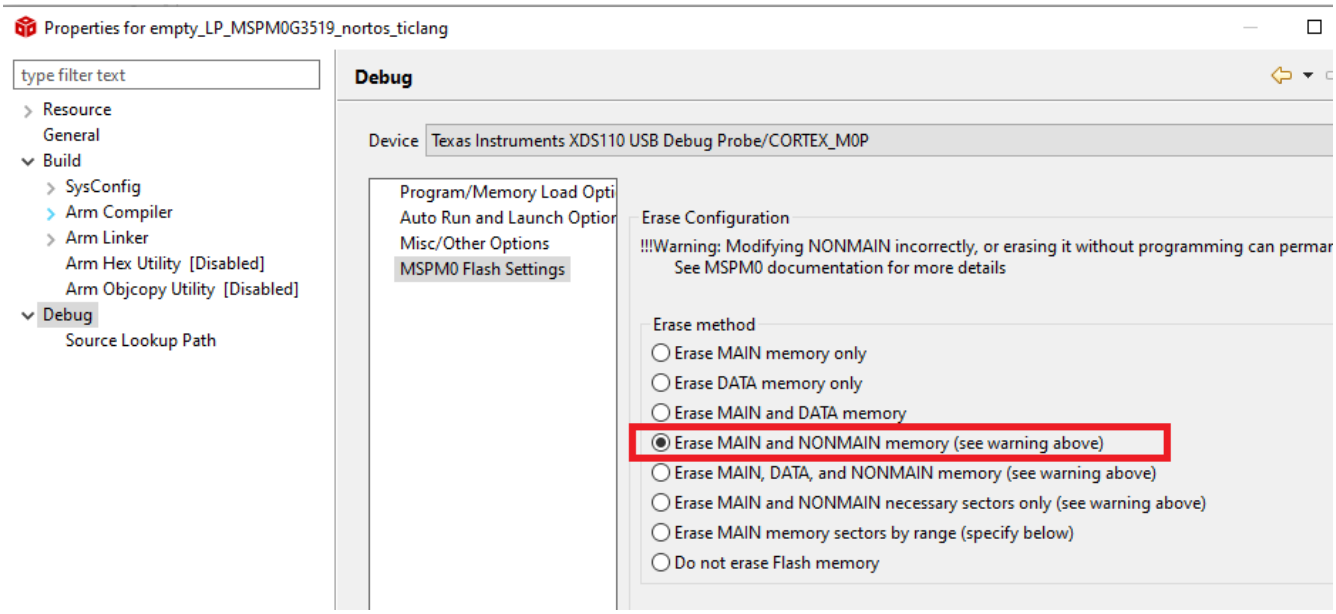

Figure 3-2. Erase Method Configuration

3.1.2 Implementation of CSC Application Code - Bank Swap Feature

The CSC needs to know the location of application image for some security function. So, the first thing needs to be done is to arrange flash memory for CSC code and application code. Change the linker file accordingly. An example of linker file configuration shows below. Split the FLASH memory into two sections FLASH_CSC and FLASH_APP. Set the SECTIONS.intvecs to the start of FLASH_CSC. And assign related sections to FLASH_CSC as well.

Note

In bank-swappable configuration, the flash memory protections are automatically mirrored to both banks. This requests that CSC code must be identical across both BANK0 and BANK1 and the application code starts at the same offset address.

```
-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
}
```

An example is provided below on how to implement the bank swap function based on CSC start up sequence. The software checks the INITDONE bit first. If INITDONE has not been issued, then set up the security code here. In this case, the bank swap function is configured by calling the DL_SYSCTL_executeFromUpperFlashBank() function first. Then, delay for a proper time and issue INITDONE bit by calling DL_SYSCTL_issueINITDONE() function. Calling the DL_SYSCTL_issueINITDONE() function triggers a SYSRST. After the SYSRST, the MCU starts the run at BANK1. Customers can also add bank swap conditions in the code. If the condition matches, then the software calls the bank swap function and runs the BANK1 application project. Customers can add other CSC functions in the code but do not forget to issue INITDONE after all the CSC functions are executed. After system reset, if the bank swap function is not enabled, then the software can run to the BANK0 application project by calling start_app() function.

```
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
    }
}
```

An example of start_app() function is provided below. The input parameter is the start address of the application. The memory map in the linker file configuration has been changed. In this case, the application code in BANK0 starts at 0x1800. The software resets the SP value and Reset Vector to the vector table of the application code. Then, set the PC to the Reset Handler address of the application code. This procedure lets the MCU start to run the application code in BANK0.

```
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 Application Code Project Preparation

The only need to do here is to arrange the flash memory in the linker file. This is required to avoid address overlap with the application code and CSC code in the same bank. Please note that application code on both banks must also start at the same address due to the CSC code policy. An example how to configure the linker file for the application code is provided below.

```
-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
}
```

Customers can add other functions in the application code project according to the project requirements. When the application code project is ready for programming, remember to change the erase method in the IDE or programmer since the application project does not include NONMAIN configuration. TI recommends to use *Erase MAIN memory only*.

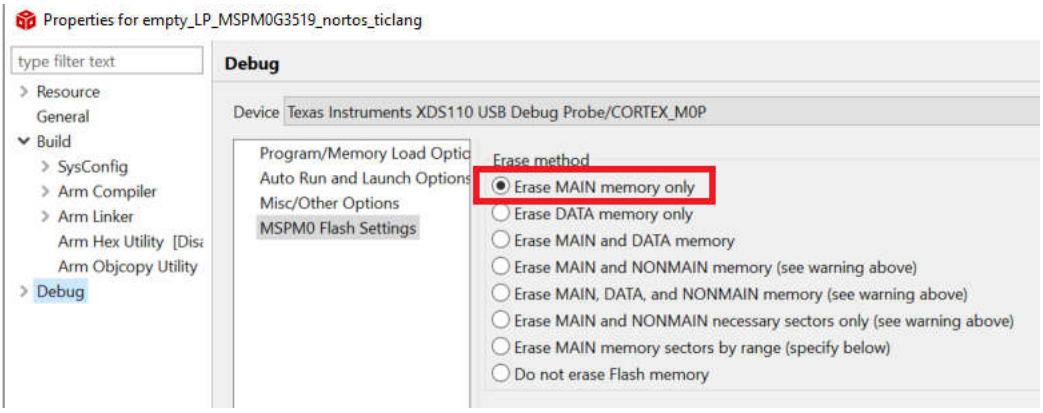

Figure 3-3. Erase Main Memory Only

4 Common Use Case Introduction

The dual bank memory can be configured and used as a single large NVM block with continuous addressing (with few exceptions, not covered in this document). There are significant advantages when the NVM is configured to serve as two parallel blocks, the most important is the possibility to write on one bank without interrupting reading (and fetching instructions) from the other bank. This is the most important prerequisite to perform the updates without breaking the execution of the code from the program NVM.

When designing an application that uses a dual bank device, there are several choices to make on how to utilize the second half of the program memory.

Also referred to as live field upgrade, this is a process that allows the user to modify the code and configuration without disturbing the normal operation of the device. There are more advantages compared to the simple boot loader design:

- Boot loader code back up: Boot code can be updated when using the dual bank
- Application code back up: The original application code is still functional if the boot loading fails
- EEPROM emulation: An application can execute code out of one flash bank while a second flash bank is used for writing data without stalling the application execution

Table 4-1 compares the differences between single bank and multi bank devices related to flash operations (Erase and Program).

Table 4-1. Flash Operation Differences on Same Bank and Different Bank

	On Same Bank	On Different Bank
Flash Operation (Erase and Program)	Flash operation command executed in SRAM.	When operates on the same bank of application code. Same as Single Bank device.
		When operates on the different bank of application code. Flash operation command executed in flash.
Interrupts during flash operation (Erase and Program)	TI recommends to disable interrupts or move important interrupt routine into SRAM before flash operation. Since an ongoing program or erase operation stalls all read requests to the flash memory until the operation has completed and the flash controller has released control of the bank.	When operates on the same bank of application code. Same as Single Bank device.
		When operates on the different bank of application code. Interrupts are responded to on time.

5 Data Bank Introduction

In some MSPM0 devices such as MSPM0Gx51x series, there is an independent 16KB DATA bank provided in BANK2. Read accesses to the DATA bank are processed through the peripheral bus, independently of BANK0 and Bank1.

5.1 Data Bank Protection

A region of flash DATA bank can be configured for read-write protection - either reads or writes or both types of accesses can be blocked. CPU, DMA and debugger accesses are all treated the same way. This is configured by writing to the SYSCTL.SECCFG.FWPROTMAINDATA register. Only the first 4KB of the DATA bank can be protected at a sector (1KB) granularity. Each sector can be configured as below:

- 0b00: Both Read/Write allowed.
- 0b01: Read Only.
- 0b10: No Read No Write.
- 0b11: No Read No Write - Not Used.

Table 5-1. FWPROTMAINDATA Field Descriptions

Bit	Field	Type	Reset	Description
31-8	RESERVED	R	0h	
7-6	DATA	R/W	0h	Sector 3 protection configuration
5-4	DATA	R/W	0h	Sector 2 protection configuration
3-2	DATA	R/W	0h	Sector 1 protection configuration
1-0	DATA	R/W	0h	Sector 0 protection configuration

5.2 Data Bank Erase Write Operation

The erase and write operation on DATA bank are identical to the MAIN region, and can be called from Flash. Erase can happen at set or bank granularity. These feature makes the DATA bank as an excellent design for data logging. Please find a more detailed description in the device technical reference manual. An example how to use the DATA bank is shown below.

```
/* 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 Summary

This application note introduces the multi-bank feature in the MSPM0 family, such as NVM introduction, CSC introduction, and database introduction. In particular, this application note demonstrates a bank swap implementation method to help customers develop this function in applications.

7 References

- Texas Instruments, MSPM0L222x, MSPM0L122x Mixed-Signal Microcontrollers , data sheet
- Texas Instruments, MSPM0 L-Series 32MHz Microcontrollers Technical Reference Manual

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265