Application Note

Evaluate Customer Secure Code on MSPM33 MCU

Yuhao Zhao

ABSTRACT

This document shows how to evaluate Customer Secure Code on an MSPM33C32 device (referred to as C32 in this document) that's in HS-FS (High Security Factory State) lifecycle state.

Table of Contents

Trademarks

1 Introduction

Customer secure code (CSC) is an implementation of the publicly available MCUboot for devices in the MSP family with advanced security features, such as the device family MSPM33C321x, to allow customers to have additional and more advanced secure booting features in the development and deployment. When paired with an application image, this represents a full solution for updates and verification of new images on the device. However, the CSC only refers to the verification of the image and configuring security settings, not loading the image onto the device. The full set of features and execution flow varies depending on the specific device used and features present on the device.

The CSC has the following features:

- Verify one or more application images using Elliptic Curve Digital Signature Algorithm (ECDSA) with NIST P-256 curve and hashing algorithm SHA-256
- Verify one or more application images using ML-DSA 87 and hashing algorithm SHA-256
- MCUboot is open source and can be modified to fit the needs of the end developer. Additionally, the CSC has a set of security features and is easily configurable using a header file or SysConfig to tailor the design to the end needs of the user
- Set up security features on the device such as write, read, and IP protect firewalls using global security controller (GSC)
- Set up the KEYSTORE and the bank-swapping policy, and dynamically manage and revoke keys as the user intends
- Rollback protection
- Able to be made immutable and act as part of the root of trust (RoT) chain during a boot.
- Easy feature configuration through the SysConfig tool, such that users can quickly build a tailored CSC

1.1 Key Concepts

NONMAIN	A dedicated flash memory region which configures device boot related parameters. Please refer to *MSPM33 NONMAIN Flash Memory Configuration Guide* for the NONMAIN configuration.
Secure Boot	The process of verifying and validating the integrity and authenticity of updateable firmware and software components as a prerequisite to the execution.
INITDONE	INITDONE is a register in MSPM33 devices that is used to isolate CSC and application. INITDONE is triggered at the end of the CSC and all non-static security policies configured in CSC takes effect during INITDONE
Customer Secure Code (CSC)	A secure boot design provided in MSPM33 SDK for the devices with INITDONE mechanism. The CSC works as part of RoT and keeps immutable after production and achieves application image integrity and authenticity verification as well as other security policy configuration. CSC can also represent a MSPM33 hardware feature which means a MSPM33 device supports INITDONE mechanism.
Root of Trust (RoT)	Especially refers to immutable Root of Trust, the most trusted security component on the device. RoT is inherently trusted because RoT cannot be modified following manufacture. There is no software at a deeper level that can verify that RoT is authentic and unmodified. Including ROM-boot code and CSC with static write protection in CSC option.
Keystore	Secure storage for AES key. Only CSC can configure keys into Keystore and the main application can configure the crypto engine (AES) to use one of the stored keys but can never access any stored keys.
Firewall	A dynamic protection mechanism for some specific region of Flash memory, implemented by global Security Controller(GSC)
Bank Swap	A mechanism to configure flash bank address mapping on MSPM33 dual-bank devices. Bank swap is configured in CSC and takes effects after INITDONE.
Static Write Protection	The static write protection mechanism enabled by NONMAIN configuration. The protected region can not be modified after ROM-boot code finished unless the NONMAIN configuration is changed for enabling writing again.

SHA2-256	The hashing algorithm which takes an entire message and condenses the message into a fixed-length (256bit) digest. This algorithm is used for verifying message integrity. Some MSPM33 devices offer hardware accelerators for SHA.
ECDSA P256	Elliptic Curve Digital Signature Algorithm is an asymmetric algorithm to verify message authenticity. Some MSPM33 devices offer hardware accelerators for ECDSA.
MLDSA-87	Module-Lattice-Based Digital Signature Algorithm is an asymmetric algorithm to verify message authenticity.
AES	Advanced Encryption Standard. Some MSPM33 devices offer hardware accelerators for AES.
TRNG	True Random Number Generator. Some MSPM33 devices offer hardware accelerators for TRNG.

2 Customer Secure Code Overview

This section provides an overview of the CSC process and user-specified policies which can be set to enable a wide variety of use cases.

2.1 Boot and Startup Sequence

Figure 2-1 shows the CSC boot and startup sequence. At BOOTRST, TI ROM 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 MAIN flash memory. For HSFS state device, a MAIN flash program always starts after boot code finishes from physical address 0x0004 vector (Reset Handler). Depending on the CSCEXISTS configuration in NONMAIN flash BCR, there are two execution flow after BOOTDONE:

- CSCEXISTS set: CSC boot sequence is enabled and MAIN flash program starts with INITDONE in clear state. Users need to place the CSC firmware into MAIN flash 0x0000 address in this case. Users can set static write protection policy in the NONMAIN BCR configuration to protect the CSC firmware.
- CSCEXISTS clear: CSC boot sequence is not allowed and MAIN flash program starts with INITDONE in set state. Any security related policy is not configurable and users need to place the application firmware into MAIN flash 0x0000 address in this case.

Note

Bank swap policy resets during BOOTRST, so the MAIN flash program always starts without bank swap after BOOTDONE. Bank swap only takes effects after INITDONE when both CSCEXISTS and FLASHBANKSWAPPOLICY enabled in NONMAIN configuration.

For the CSC existing case, CSC is responsible for determining execution bank, secure key initialization into the KEYSTORE, take application program integrity and authenticity verification, and others. The device is working in a privileged state with permission configuring those security policies. The INITDONE is issued (by writing to SYSCTL.SECCFG.INITDONE, see the device-specific technical reference manual for the register definition) at the end of CSC, then SYSCTL issues a second SYSRST and all the security policies listed below take effect during INITDONE and cannot be modified until the next BOOTRST:

- Bank swap policy
- Keystore protection

During CSC, the global security controller (GSC) can be configured for security policies. The vector table offset register (VTOR) from GSC also updates for the vector table address of the main application.

After INITDONE, a SYSRST is triggered, and the device starts execution from address in VTOR and directly jumps to the main application.

For more details on boot and startup sequence, plrese refer to the *SECURITY* chapter of the *MSPM33 C3-Series 160MHz Microcontrollers* technical reference manual.

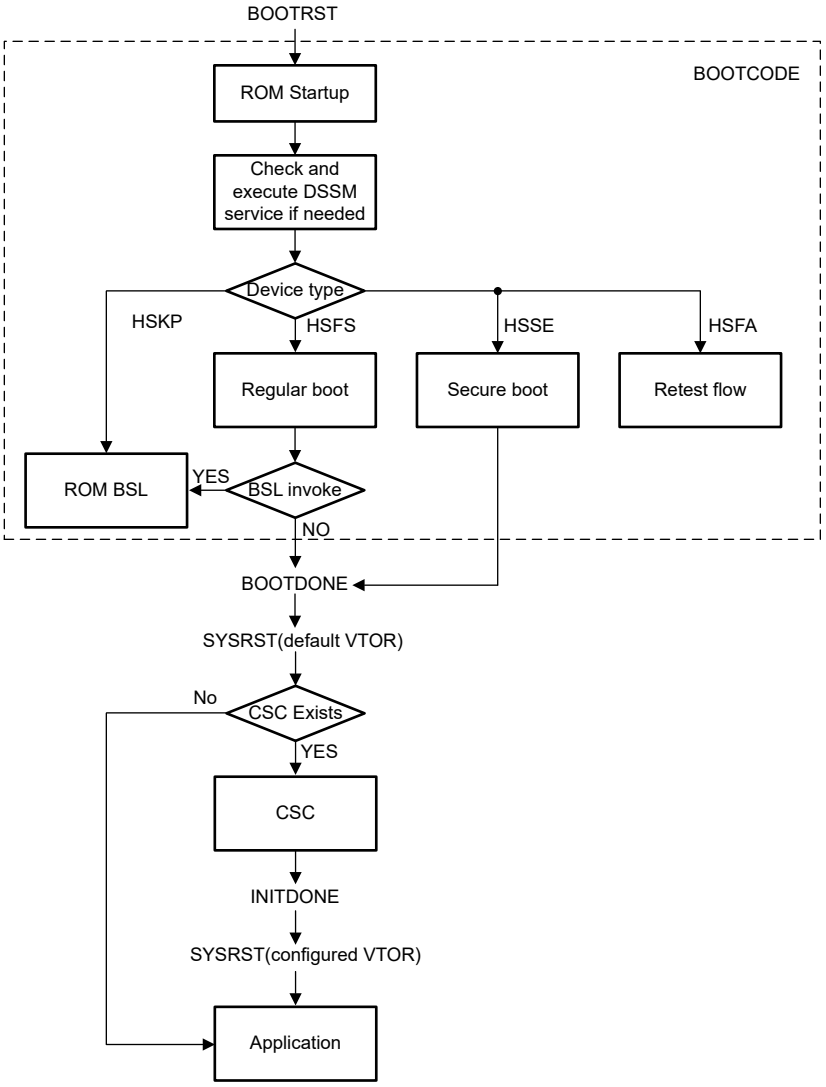


Figure 2-1. Boot and Startup Sequence

2.2 CSC Flow

This section explains the detailed boot flow in the CSC design based on MSPM33 SDK CSC example(MSPM33 SDK 1.04.00.00), shown in Figure 2-2. The whole execution flow is mostly compatible with the flow chart shown in Figure 2-1.

When ROM boot code execution is done, program goes to CSC firmware and the INITDONE (SYSCTL.SECCFG.SECSTATUS.INITDONE) is in clear state. CSC searches for the highest version image from both flash banks, checks version rollback, and then verifies the application image authority and integrity by asymmetric approach (SHA256+ECDSA_p256 and MLDSA87). After the verification is passed, CSC updates rollback counter, SECRET keys and KEYSTORE. Then the CSC configures the firewall in SECRET flash region and Lockable flash region, and determines bank swap policy. CSC also configures VTOR in GSC to set up the vector table offset for application image. CSC issues an INITDONE to trigger a SYSRST and device jumps to application image to start application.

There are some key notes related to the CSC example execution flow:

- The CSCEXISTS and FLASHBANKSWAPPOLICY filed in NONMAIN flash need to be enabled for enabling the whole CSC sequence. Please refer to *MSPM33 NONMAIN Flash Memory Configuration Guide* for the configuration.
- PB0 represents Physical Bank0. As bank swap policy does not take effect before INITDONE is set, the flash address used in CSC always refer to the physical address.

- If two images in PB0 and PB1 are with the same version, the PB0 image is verified and executed in a higher priority.
- If the highest version image does not pass the SHA256+ECDSA or MLDSA verification, then the image in the other bank (if exist) is verified immediately after.
- In the case of asymmetric authentication, the secure hash (SHA2-256) digest of the application code is computed first, then ECDSA and MLDSA verifies the image signature based on public key in firmware.
- For M33 device, SYSRST does not clear VTOR in GSC. When CSC issues an INITDONE, an SYSRST is triggered together. With configured VTOR, device starts from the application.

SECRET flash region is a user specified region which stores secret information and is read-execute protected by a firewall. Lockable flash region is a user specified region which stores unmodified information and is write-protected by a firewall. For more details, see FLASH Memory Map.

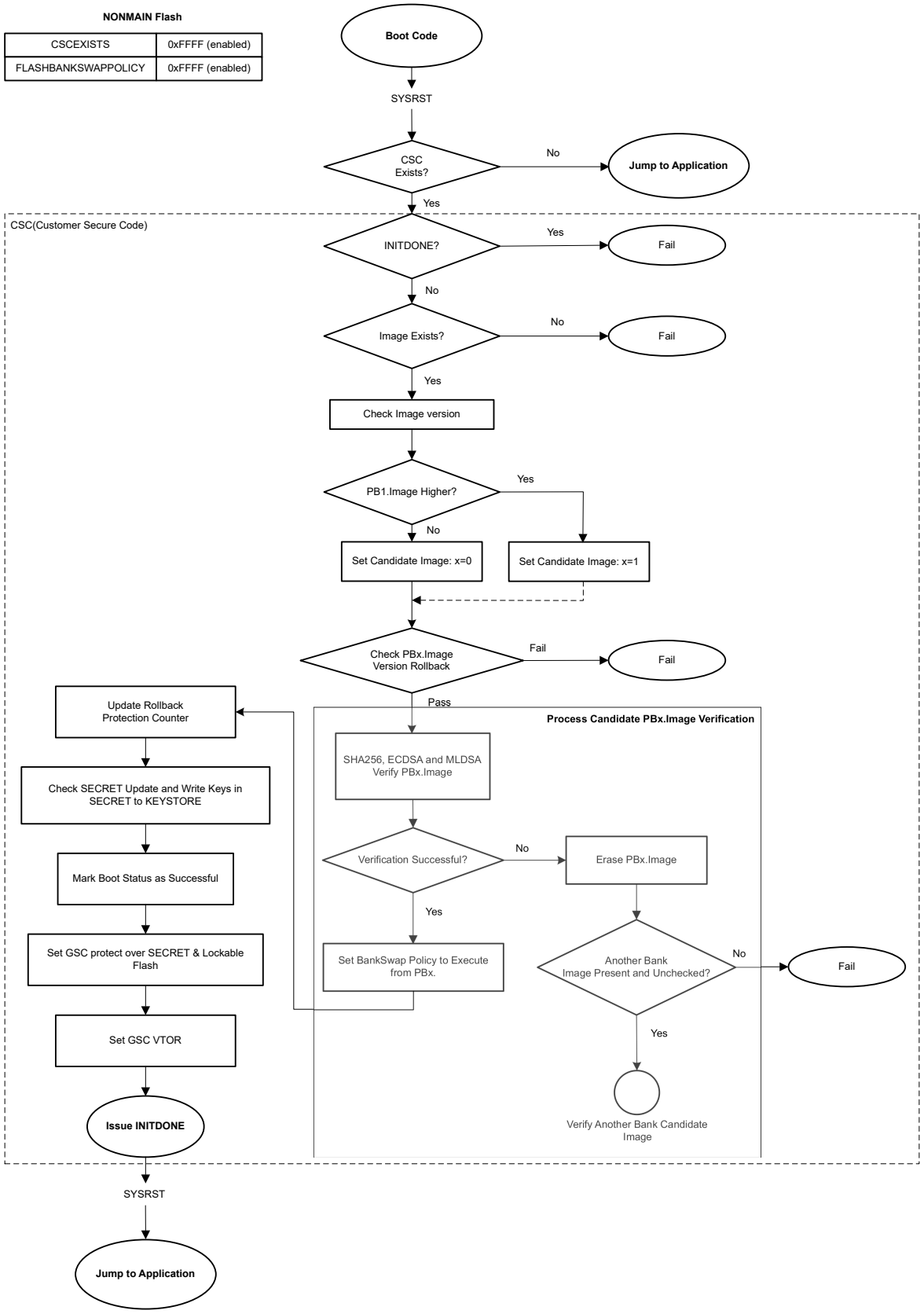


Figure 2-2. MSPM33 SDK CSC Execution Flow

2.3 FLASH Memory Map

Figure 2-3 shows the detailed flash memory map in the CSC secure boot. The following are the explanations of sections in CSC:

- SECRET: The SECRET is visible to the CSC but is protected by a read-protect firewall, thus rendering the SECRET invisible to the application. The SECRET region can store non-volatile keys that are loaded into KEYSTORE at runtime. Thus, the application can use these keys without having read access. Customize the SECRET to include additional information such as tag and key.
- Lockable Flash: The lockable flash provides dynamic write protection to key information that needs to be written by the CSC and be read but unmodified by the application. The security counter (rollback protection), the keystore hash table, and the image hash are typically stored in the lockable flash. Notice that the lockable content is programmed to CSC region in PB0 only, to make sure both bank application programs can access this region.
- CSC Interrupt Vectors: These are the interrupt vectors for the customer secure code. This interrupt vector table is the first thing run from flash in the event of a BOOTRST or a SYSRST with default VTOR. In CSC, VTOR is updated with address of the application interrupt vectors, then SYSRST with configured VROT leads the device to start from application.

  Note

  SYSRST does not reset VTOR, meaning that after CSC is executed, the device does not enter CSC again until next BOOTRST.
- CSC Code: The main code and security primitives is the bulk of the Customer Secure Code. Typically, the CSC code and the interrupt vectors only need to be programmed to Physical Bank 0 because CSC is executed before bank swap take effect. During a bank swap, after the SYSRST triggered by INITDONE, the Program always runs from the application in Logical Bank 0 (logic address). FLASHCTL maps the address to PB0 address or PB1 address according to bank swap policy configuration.

The following are sections of the application image:

- Image Header, Image TLV and Image Trailer: generated by signing tool imgtool, which is provided by MCUBOOT (located with python scripts in <mspm33_sdk_path>\source\third_party\mcuboot\scripts). These parts are generated and merged to a compiled application image in the CCS post-build step. There is a customer_secure_sample_image example in MSPM33 SDK which shows how a signed image is built in CCS. The following are the explanations of those image parts:
  - Image Header: The header information of application image, including the header magic (0x96F3B83D), image size and image version. The image header is located at the address 0x200 bytes (by default) before the application interrupt vectors.
  - Image TLV: MCUBOOT defines Type-length-value records (TLV) containing image metadata which are placed after the end of the image. The TLVs defined in MSPM33 CSC includes: TLV magic (0x6907), image hash, ECDSA public key hash, ECDSA signature, MLDSA public key hash and MLDSA signature. For more details, see mcuboot/docs/design.md at main · mcu-tools/mcuboot · GitHub.

    Note

    A SHA256 is only executed for the content including image header, application interrupt vectors and application image.
  - Image Trailer: A 16-bytes magic content which is located at the end of image flash areas.
- Application Interrupt Vectors: These are separate interrupt vectors that the application uses. During the CSC jumping to the application, the vector table offset register (VTOR) points to this position in memory, and thus all future interrupts occur without a reset(BOOTRST or higher reset) links to this set of interrupt vectors. The start address is 32-bytes aligned.

  Note

  For ARM Cortex-M33, there is a requirement on the VTOR address alignment for 0x200. Please refer to E2E thread for more information.
- Application code: The original application code.

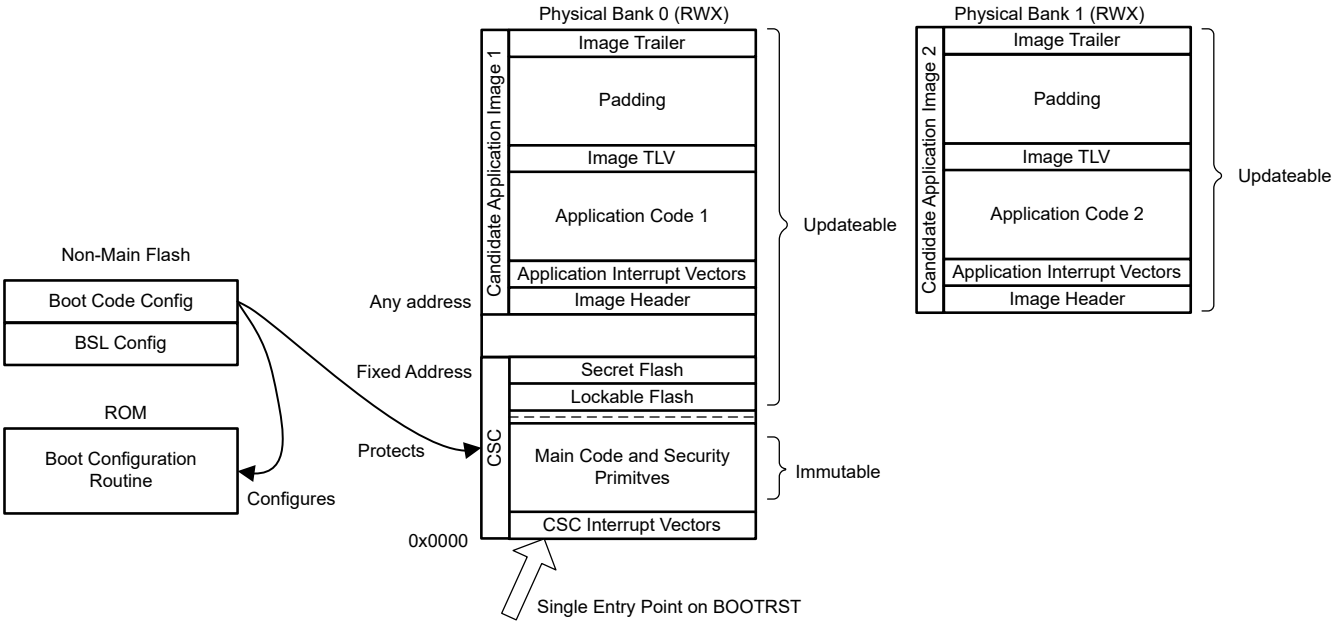


Figure 2-3. CSC Flash Map

2.4 Bank Swap

PB = Physical bank. There is a physical bank 0 (PB0) and physical bank 1 (PB1). These banks can switch addresses depending on the bank swap configuration.

LB = Logical bank. These are the two logical banks that map to physical bank 0 or physical bank 1. Before INITDONE Logical Bank 0 always maps to address PB0, and Logical Bank 1 maps to PB1. However, depending on the bank swap execution, post INITDONE LB0 can map to PB0 or PB1. Logical Bank 0 post INITDONE is the execute bank, and Logical Bank 1 is the data bank.

Note

Since code only executes from logical bank 0, compile all code so that the code runs from logical bank 0. Thus, if the CSC is mapped from 0x0000 - 0x47FF and one application image mapped from 0x4800 onward, then compile each application image with respect to 0x4900 (0x100 trailer, then application interrupt vectors placed at 0x4800 + 0x100 = 0x4900), even if the application image is written to logical bank 1 at effective physical address.

In bank swap method, the actual firmware does not move between physical locations - instead, the hardware is configured to execute from the appropriate bank.

The bank swap mechanism in CSC is implemented using the DL_SYSCTL_executeFromUpperFlashBank() and DL_SYSCTL_executeFromLowerFlashBank() functions, which are called based on the necessity.

Note

When users need to configure security attribution unit (SAU) for security attribution, configured address is based on logical address. In bank-swappable configuration, please carefully configure SAU based on logical address.

Note

When users need to configure global security controller(GSC) for security privilege attribution, FLASH hide protection, static write protection or others, related registers in GSC are based on physical banks. In bank-swappable configuration, please carefully configure GSC based on physical address.

2.5 CSC Feature

2.5.1 Asymmetric Verification

The integrity and authenticity of a new image are verified through cryptographic algorithms. Only images that have been verified are considered secure and can be executed.

2.5.1.1 SHA-256

The Secure Hash Algorithm 256 (SHA-256) is a widely used cryptographic hash function that generates a fixed-length, 256-bit digest from any input message. The algorithm is sensitive to input variations meaning that SHA-256 is designed so that even a minor change in the input message results a different output hash.

One of the core features is collision resistance, which means that the algorithm is extremely unlikely to produce the same hash value for two different messages. This property makes SHA-256 highly reliable for verifying the integrity of data, as any modification can be easily detected.

In practice, SHA-256 is commonly used in digital signatures and data integrity checks. By condensing an entire image into a unique digest, SHA-256 provides a robust foundation for the next of ECDSA algorithm.

2.5.1.2 ECDSA

Elliptic curve digital signature algorithm (ECDSA) is a cryptographic algorithm used for digital signatures, based on elliptic curve mathematics. ECDSA offers high security with much shorter key lengths compared to traditional algorithms like RSA, making this algorithm efficient and designed for resource-constrained environments.

ECDSA is one supported authentication method for MCUboot. ECDSA is an asymmetric algorithm, meaning there is a separate public and private key. The public key is stored in the device flash, and the developer maintains the private key. CSC does not provide secure private key management.

ECDSA uses the hash of the image and the public key to verify a digital signature (r, s), checking the authenticity of the data. Compared to symmetric cryptographic options, this option does not present a key vulnerability on the device.

Note

To keep the boot flow for final deployment, keep the private key secure and managed so that the key is not easily accessible to sign images. Keeping the key on a local share drive is not a secure location! TI does not currently provide secure private key management.

2.5.1.3 MLDSA

Module-lattice-based digital signature algorithm (ML-DSA) is a post-quantum cryptographic algorithm designed to protect against quantum computing threats. ML-DSA is based on the cryptographic suite for algebraic lattices (CRYSTALS) family of PQC algorithms and is particularly effective against chosen-message attacks.

MLDSA is not the supported authentication method for MCUboot. Some code modifications are implemented to integrate MLDSA.

- Define TLV types -
  - #define IMAGE_TLV_MLDSA87 0xCE /* Custom TLV for MLDSA87 signature */
- Key management - In case of hybrid approach where we need more than one type of algorithm. The bootutil_keys array needs to include another key
- Modify the image validation code - Update the bootutil_img_validate function to use MLDSA verification function
- Create the MLDSA verification function - Create a new file (for example, image_mldsa87.c) that implements the MLDSA verification function

MLDSA is an asymmetric algorithm, meaning there is a separate public and private key. The public key is stored in the device flash and the private is maintained by the developer. The CSC does not provide secure private key management.

2.5.2 Keystore

KEYSTORE is a protected SRAM memory which can securely store AES key, the key is configured in CSC before INITDONE, and application can trigger the key transfer from KESTORE to AES engine but not directly access (read or write) these keys after INITDONE.

Only CSC can configure keys into KEYSTORE before INITDONE. Subsequently, the main application can configure the crypto engine to use one of the stored keys but can never access (read or write) any stored keys. The key transfer from KEYSTORE to crypto engine is performed securely and is not visible to the application code.

KEYSTORE contents will be wiped every BOOTRST and the store is unlocked for writing. KEYSTORE contents are unaffected by SYSRST and lower order resets.

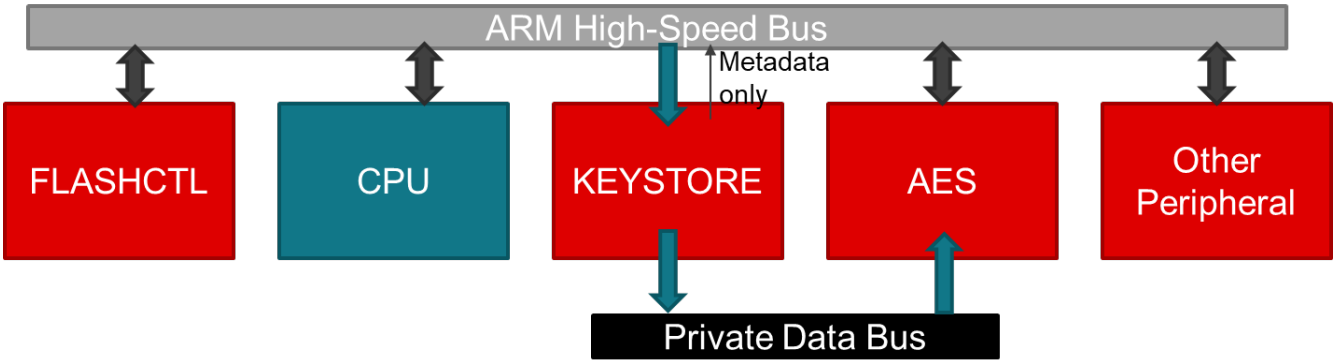


Figure 2-4. KEYSTORE Works Process

2.6 Protection on CSC

MSPM33 provides various types of memory protection mechanisms on FLASH and SRAM to apply protection on CSC.

2.6.1 Write Erase Protection

MSPM33 MCUs implement a static write protection scheme to lock out user defined sectors in the flash region from any program or erase operations at runtime. The desired static write protection scheme is configured in global security controller(GSC) and user can apply write protection on MAIN flash, DATA flash or NONMAIN flash. Please refer to the *GSC* chapter of the *MSPM33 C3-Series 160MHz Microcontrollers* technical reference manual.

Static write protection enables placement of a fixed, user-defined region in the flash memory that once programmed and locked, the region is not modifiable by the application code or ROM bootloader.

- If placed at the beginning of the MAIN flash memory, the application in the region is the first code that executes when the ROM boot configuration routine transfers execution to the user application, which is one requirement to implement a secure boot image manager.
- If placed at the NONMAIN flash, configuration memory in the NONMAIN flash is locked then and can prevent any unauthorized modification of the security policies, bootstrap loader policies, and static write protection policies by either the bootstrap loader or the application code.

Any region that is configured in the GSC to be write-locked is functionally immutable when the boot configuration routine transfers execution to either the bootstrap loader or the user application code in MAIN flash. Any attempt to program or erase a statically protected region by the application code or the bootstrap loader results in a hardware flash operation error, and the sector is not modified.

Note

Access type of GSC regsters for write protection is 'R/W1S'. 'W1S' is that register bit can only be set from 0 to 1, and register can only be reset by BOOTRST or higher reset type, which means the applied write protection can not be directly removed.

MSPM33 also supports to configure write protection policy in the NONMAIN flash. During ROM boot-up, boot configures GSC according to NONMAIN flash. This configuration takes effect before entering application code, which means application code cannot remove write protection unless modifying the NONMAIN flash again. Please refer to *MSPM33 NONMAIN Flash Memory Configuration Guide* for the configuration.

While static write protection prevents any modification by application code or the bootloader, a mass erase or factory reset command sent through the SWD interface is honored. If this behavior is not desired, the mass erase or factory reset SWD commands can be protected with unique passwords or disabled. To completely remove any means of modifying statically write protected MAIN flash sectors, please follow the instructions below.

- Set write protection to MAIN flash in the NOMAIN.
- Disable mass erase and factory reset commands (or the SW-DP) in the NONMAIN.
- Set write protection to NONMAIN flash in the NONMAIN to prevent application code from changing the underling write protection scheme by modifying the contents NONMAIN region.

Please refer to *MSPM33 NONMAIN Flash Memory Configuration Guide* for the above instructions.

Note

By default, the NONMAIN configuration memory (which contains the user-specified boot security policies and bootstrap loader policies) is not write protected. The user can erase the NONMAIN during provisioning and re-programmed with the user-specified policies to use in mass production.

2.6.2 FLASH Hide Protection

MSPM33 supports hide protection (HDP) function, allowing an application to execute a section of code one time, and prevent subsequent read or execute access to the code. As an example, this is useful feature especially when running a second stage authentication code allowing main application to be validated before execution and prevent access to any stored symmetric or private keys. The desired HDP scheme is configured in global security controller(GSC) and user can apply HDP on MAIN flash, DATA flash or NONMAIN flash. Please refer to the *GSC* chapter of the *MSPM33 C3-Series 160-MHz Microcontrollers* technical reference manual.

After HDP is correctly configured and enabled, when CPU is executing from the configured HDP region, the HDP is armed. Any function call or jump outside of the configured HDP region automatically enables the mechanism, preventing the application code from re-entering the HDP region. An access, execute or read, generates a non-maskable interrupt (NMI).

- If HDP is enabled on main Flash bank, then read, write, execute and erase operations are blocked.
- If HDP is enabled on non-main Flash bank, then read and execute operations are blocked. To enable write and erase protection, write-erase protection need to be configured additionally.

Note

Access type of GSC regsters for HDP is 'R/W1S' or 'WOnce'. 'W1S' is that register bit can only be set from 0 to 1. 'WOnce' is that register bit can only be set once before reset. These registers can only be reset by BOOTRST or higher reset type, which means the applied HDP can not be directly removed.

Note

When executing code in HDP region followed by branch to SRAM, the SRAM code can access HDP region. Once all SRAM code execution is completed, the execution must branch back and exit HDP region with continued execution in flash to enable the HDP mechanism. If application does not want SRAM to access HDP region, then the code must implement a trampoline function to enable HDP before branching to SRAM code.

2.6.3 Data Integrity Verification

The BCR supports checking the data integrity of a user-specified address range in the MAIN flash memory before transferring execution from the BCR (in ROM) to the user application (in MAIN flash memory). Please refer to *MSPM33 NONMAIN Flash Memory Configuration Guide* for the configuration.

The integrity check can be used as an additional step to verify that the code which runs first after the boot ROM (usually the secure boot image manager) has a CRC/SHA256 digest that matches the expected value. This integrity check reduces the likelihood that any unexpected corruption of critical code in the flash memory (which can be responsible for authenticating the remaining user application software image) can create a security vulnerability.

A start address, length, and ISO-3309 CRC-32 or SHA2-256 digest can be provisioned into the NONMAIN configuration memory. During the boot process, the BCR computes the CRC-32 digest of the specified range in the MAIN flash memory, and verify the computed digest against the provisioned (expected) digest. If the values match, the user application is started. If the values do not match, the user application is not started and the result is a catastrophic boot error.

2.6.4 Secure & Privilege Protection

Some MSPM33 supports secure protection with trustzone articheture, and privilege protection with PMU. Please refer to data sheet to check the support status.

In general, CSC is set to secure and privileged. Any access from non-secure or non-privilege region leads to security/privilege violation. To configure secure or privilege protection on MSPM33, please refer to *Enable Trustzone for Secure Application in MSPM33 MCUs*.

2.7 CSC Performance

The MSPM33 CSC code size is related to the compiler and optimization level. The default CSC code size information is listed below:

- CSC Region Size: 30KB
- CSC Main Code Size: 28KB
- SECRET Size: 1KB
- Lock Storage Size: 1KB

The CSC example can be customized to modify the main code and SECRET or Lock Storage size. If there is a requirement to change CSC region size, also change the start address of application firmware. Refer to Section 4 for details.

The following is the CSC timing Performance for reference. Test APP image is about 25.9KB.

Table 2-1. CSC Timing Performance

ECDSA Verify	MLDSA Verify	SHA256
130.3ms at 160MHz	104.9ms at 160MHz	361.5us at 160MHz

3 Evaluate CSC-based Secure Boot

3.1 Environment Setup

This document follows the dependencies below.

1. Software and tools:
   a. MSPM33 SDK: mspm33_sdk_1_04_00_01
   b. CCS 20.5
   c. Python 3.11 or newer for MCUBoot's Image Tool (imgtool.py)
2. Hardware:
   a. MSPM33C321A samples

In the *MSPM33 SDK*, the following two projects are provided.

1. CSC example - customer_secure_code
   a. CSC example provides the ability to verify the application image with either ML-DSA-87, ECDSA-256, or both.
   b. This example is located in the folder *./examples/nortos/LP_MSPM33C321A/boot_manager/customer_secure_code*.
2. Application example - customer_secure_code_sample_image
   a. This example is provided as a companion to CSC example to demonstrate the use of CSC
   b. This example is located in the folder *./examples/nortos/LP_MSPM33C321A/boot_manager/customer_secure_code_sample_image*

TI recommends importing both these projects into CCS. For the CSC example usage, user can also refer to *MSPM33 Customer Secure Code and Bootloader (CSC) User's Guide* in the SDK.

To run the initial setup, make sure Python 3.11 or newer is installed with the latest pip package and take below steps to download the necessary requirements.

1. Open a command line window, and run below comment to check whether Python is installed in your environment:

```
python --version
```

2. Go into MSPM33 SDK install path (such as C:\ti\mspm33_sdk_1_03_00_01\), and run below command in command line for necessary requirements:

```
python -m pip install --user -r source/third_party/mcuboot/scripts/requirements.txt
```

3. Python scripts, located under /source/third_party/mcuboot/scripts, apply the python libraries when signing the application image.

3.2 Program CSC

3.2.1 General

When building and flashing the CSC onto the device, the non-main flash must be configured too (configuring BootCFG2 to enable CSC). CSC example has included modified nonmain FLASH default.

The following is the cmd file for nonmain.

```
MEMORY
{
    FLASH          (RX)  : origin = 0x10000000, length = FLASH_SIZE, fill = 0xFFFFFFFF,
    SECRET         (RX)  : origin = end(FLASH), length = CSC_SECRET_SIZE, fill = 0xFFFFFFFF,
    LOCK_STORAGE   (RX)  : origin = end(SECRET), length = CSC_LOCK_STORAGE_SIZE, fill = 0xFFFFFFFF,
    BCR_CONFIG     (R)   : origin = 0x80101800, length = 0x00000400,
    BSL_CONFIG     (R)   : origin = 0x80101C00, length = 0x00000400,
    SRAM           (RWX) : origin = 0x30000000, length = 0x00040000,

}
```

```
SECTIONS
{
    .intvecs:  > 0x10000000
    .text    : palign(16) {} > FLASH
    .const   : palign(16) {} > FLASH
    .cinit   : palign(16) {} > FLASH
    .pinit   : palign(16) {} > FLASH
    .rodata  : palign(16) {} > FLASH

    .ARM.exidx    :  palign(16)  {} > FLASH
    .init_array   :  palign(16)  {} > FLASH
    .binit        : palign(16) {} > FLASH
    .TI.ramfunc   : load = FLASH, palign(16), run=SRAM, table(BINIT)

    .secret  : palign(16) {} > SECRET
    .lockStg : (NOINIT) palign(16) {} > LOCK_STORAGE

    .vtable :   > SRAM
    .data   :   > SRAM
    .bss    :   > SRAM
    .sysmem :   > SRAM
    .stack  :   > SRAM (HIGH)

    .BCRConfig  : {} > BCR_CONFIG
    .BSLConfig  : {} > BSL_CONFIG

}
```

Figure 3-1. Memory Allocation for Nonmain FLASH

Sysconfig has the default configuration on Nonmain FLASH. After build, sysconfig will generate boot_config.c/h for nonmain:

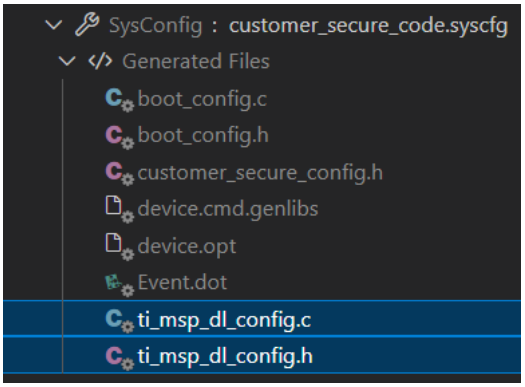


Figure 3-2. boot_config.c/h Generated by SysConfig

CSC is targeted to validate application with sample keys present in *customer_secure_code_LP_MSPM33C321A_nortos_ticlang\boot_keys.c*. These keys are the public keys for verification(the corresponding private keys have been used for signing the application images).

3.2.2 Step 1 - Building

To generate the corresponding CSC Binary, import the CSC example: <MSPM33_SDK>\ examples\nortos\LP_MSPM33C321A\boot_manager\customer_secure_code.

Now build the project and find the generated binaries in the debug folder. The *customer_secure_code_LP_MSPM33C321A_nortos_ticlang.out* shall be flashed onto the device. This binary has the ECDSA-256 public key, ML-DSA87 public key, CSC application and nonmain configuration.

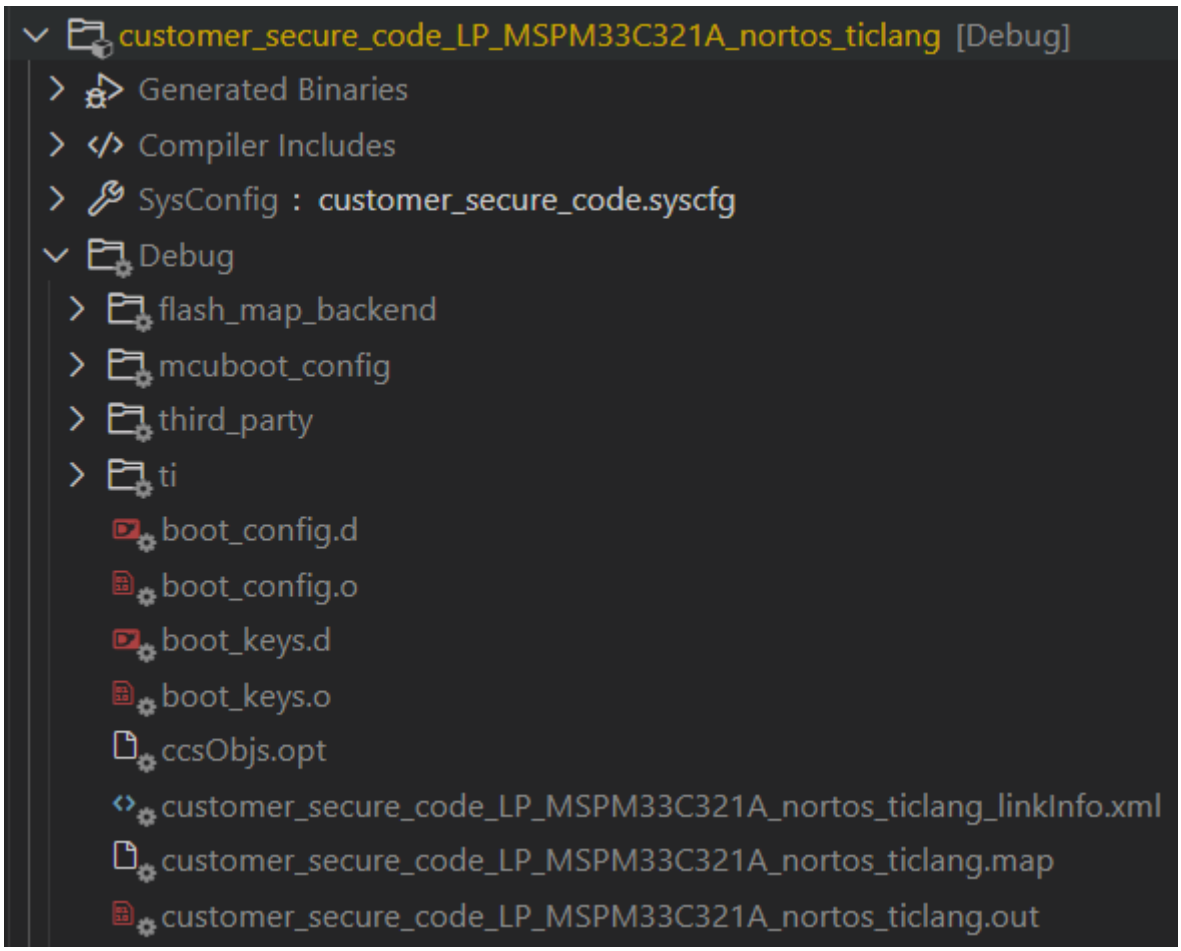


Figure 3-3. Building the Project to Generate Binaries

3.2.3 Step 2 - Programming

To load the image in CCS, do the following steps:

1. Connect to the device by starting a project-less debug using the CCXML file found in the debug folder in the CSC project. To do this right click on the CCXML file and click "start project-less debug".
2. Configure the debugger to **"Erase MAIN and NONMAIN memory"** in the debugger settings. Go to run->Debugger Properties->MSPM33 Flash Settings.
3. Configure the erase configuration to use Erase MAIN and NONMAIN memory as seen in the image below

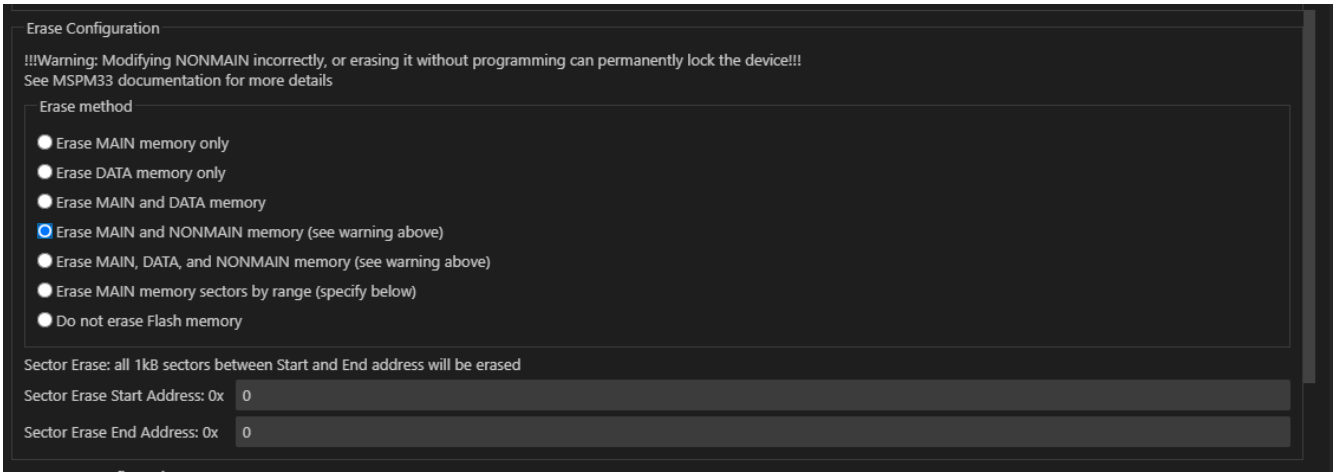


Figure 3-4. Configure 'Erase MAIN and NONMAIN memory'

4. Load the CSC binary. To do this select run-> load->load program. Then select the generated out file.

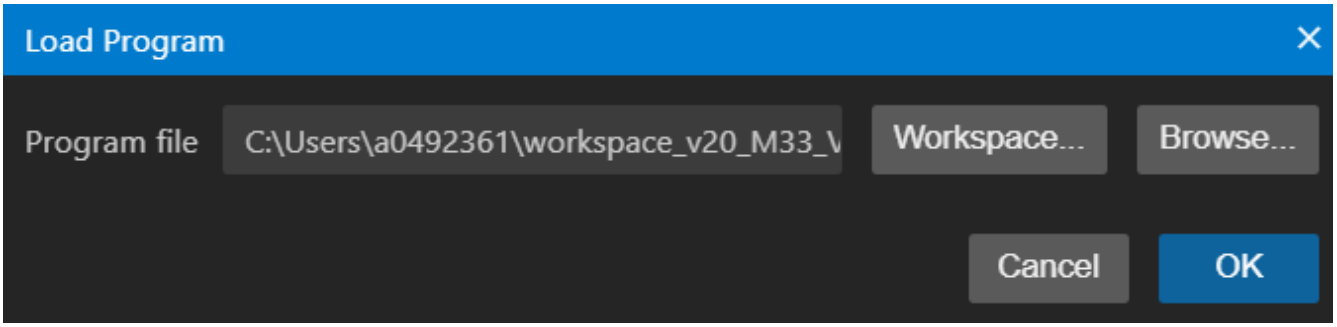


Figure 3-5. Load Program with Generated CSC Binary

5. To confirm non-main configuration is performed as required, please use the memory viewer (found in view -> memory) to see if the value at location 0x400B2048(SYSCTL_SECSTATUS). If the most significant bit is 4 the CSC has been loaded correctly. Press the NRST button and reconnect to the core. See image below for an example

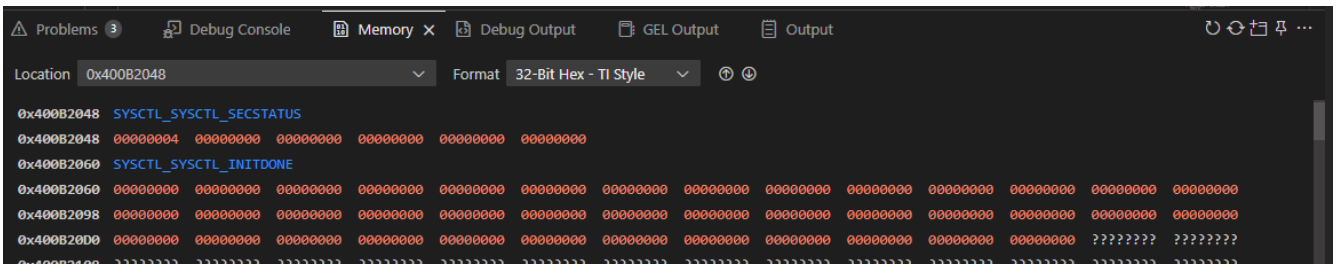


Figure 3-6. Load Program with Generated CSC Binary

Note

After doing the steps above change the Erase configuration back to only Erase MAIN. Incorrectly writing to non-main can make the device unrecoverable.

3.3 Program Application

3.3.1 General

To create the application images that can be verified by the CSC, the programmer needs to sign the application image and put the corresponding hash in the application code. This is calculated using the Image Tool that is provided by MCUBOOT located in the SDK at *<MSPM33_SDK>\source\third_party\mcuboot\scripts*.

This document utilizes Image Tool (*imgtool.py)*from MCUBoot to sign binaries. Image Tool is present in the MSPM33 SDK installation, in the folder *./source/third_party/mcuboot/*. Below is a reference command for signing binaries using Key0.pem (private key).

```
python scripts/imgtool.py sign --pad-header -k key0.pem -v 1.0.0 -H 0x70 -S 0x100000 --align 16 -F
0x10000080 <output_image.bin> <input_image.bin>
```

Note

1. The instructions in this document are based on using python on windows system. If a linux system is used, replace the invocation of "python" with "python3".
2. Image Tool does not support signing of .out files. Convert the .out files to .bin files. This is covered in next sections.
3. Modify the CCS project settings to integrate the Image tool and .bin file generation into the post build steps to automate the signed binary file generation.

For a list of commands for the Image Tool, please refer to *Imgtool ReadMe*.

3.3.2 Step 1 - Application Image Generation – Image 1 (version 1.0.0, Located at Bank 0 – 0x10000)

To generate the application image, you can use any project but for this document, we focus on the CSC sample application. We create two separate versions of the binaries and flash both onto the device. Each binary has a corresponding version number, and the CSC boots the latest version image. This is used to highlight the update functionality of the device and the version differentiation capabilities.

1. To start, import the CSC_sample found here: *<MSPM33_SDK>\examples\nortos\LP_MSPM33C321A\boot_manager\customer_secure_sample_image*. For the first application, build the project by setting the start location of flash to be 0x10200. Find the location in the .cmd file:

```
51  MEMORY
52  {
53    FLASH         (RX)  : origin = 0x00010200, length = FLASH_SIZE, fill = 0xFF,
54    SRAM            (RWX) : origin = 0x30000000, length = 0x0000FFFF
55
56  }
57
58  SECTIONS
59  {
60      .intvecs:  > 0x00010200
61      .text    : palign(16) {} > FLASH
62      .const   : palign(16) {} > FLASH
63      .cinit   : palign(16) {} > FLASH
64      .pinit   : palign(16) {} > FLASH
65      .rodata  : palign(16) {} > FLASH
```

Figure 3-7. Start Location of Flash for Application Image 1

2. Add these post-steps to the Properties-Build-Steps:

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input1.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.0.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input1.bin ${PROJECT_BUILD_DIR}/dual_signed_output1.bin
```

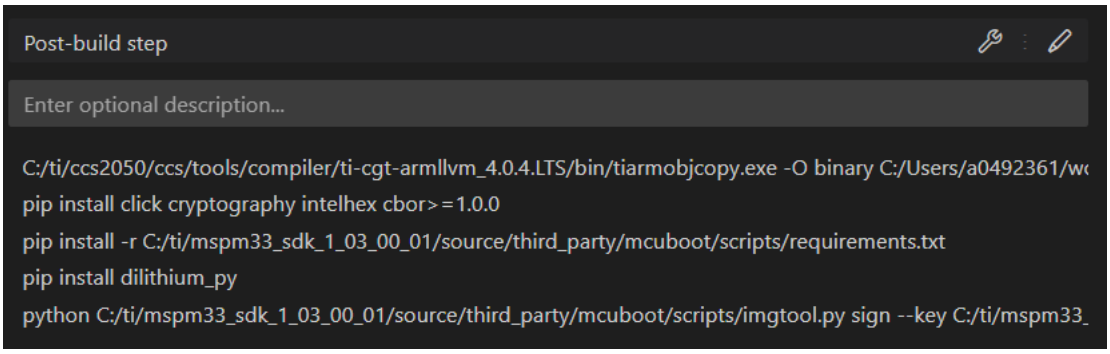


Figure 3-8. Post-steps in the Properties

3. Build the project, and you will see the "dual_signed_out1.bin" file in the projects debug folder.

3.3.3 Step 2 - Application Image Generation – Image 2 (version 2.0.0, Located at 0x20000)

1. With the bank swap feature, the second application uses the same linker file as first application, such that the start location of the application to be 0x10200.
   a. No modification needed here as the logical start address for 1st app and 2nd app are same.

```
51  MEMORY
52  {
53    FLASH         (RX)  : origin = 0x00010200, length = FLASH_SIZE, fill = 0xFF,
54    SRAM            (RWX) : origin = 0x30000000, length = 0x0000FFFF
55
56  }
57
58  SECTIONS
59  {
60      .intvecs:  > 0x00010200
61      .text    : palign(16) {} > FLASH
62      .const   : palign(16) {} > FLASH
63      .cinit   : palign(16) {} > FLASH
64      .pinit   : palign(16) {} > FLASH
65      .rodata  : palign(16) {} > FLASH
```

Figure 3-9. Start Location of Flash for Application Image 2

2. change the post-steps to include the following commands:

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 2.0.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

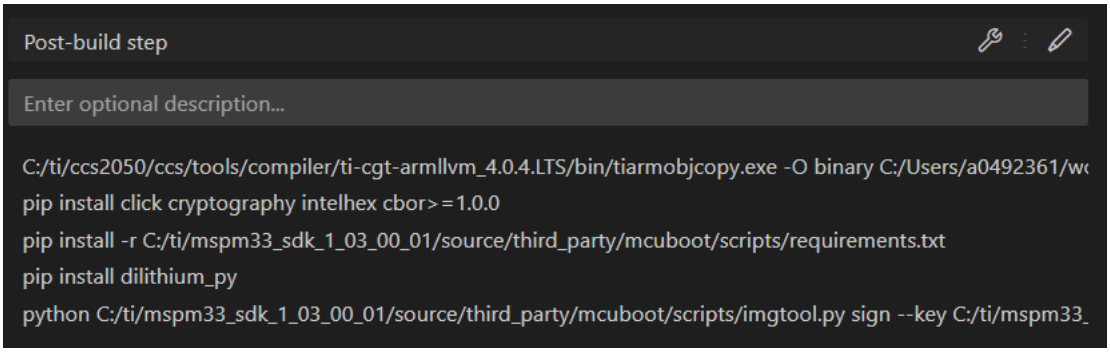


Figure 3-10. Post-steps in the Properties

3. Now build the project again. This will generate the "dual_signed_output2.bin". Figure 3-11 shows what the debug folder looks like after building the two binaries.

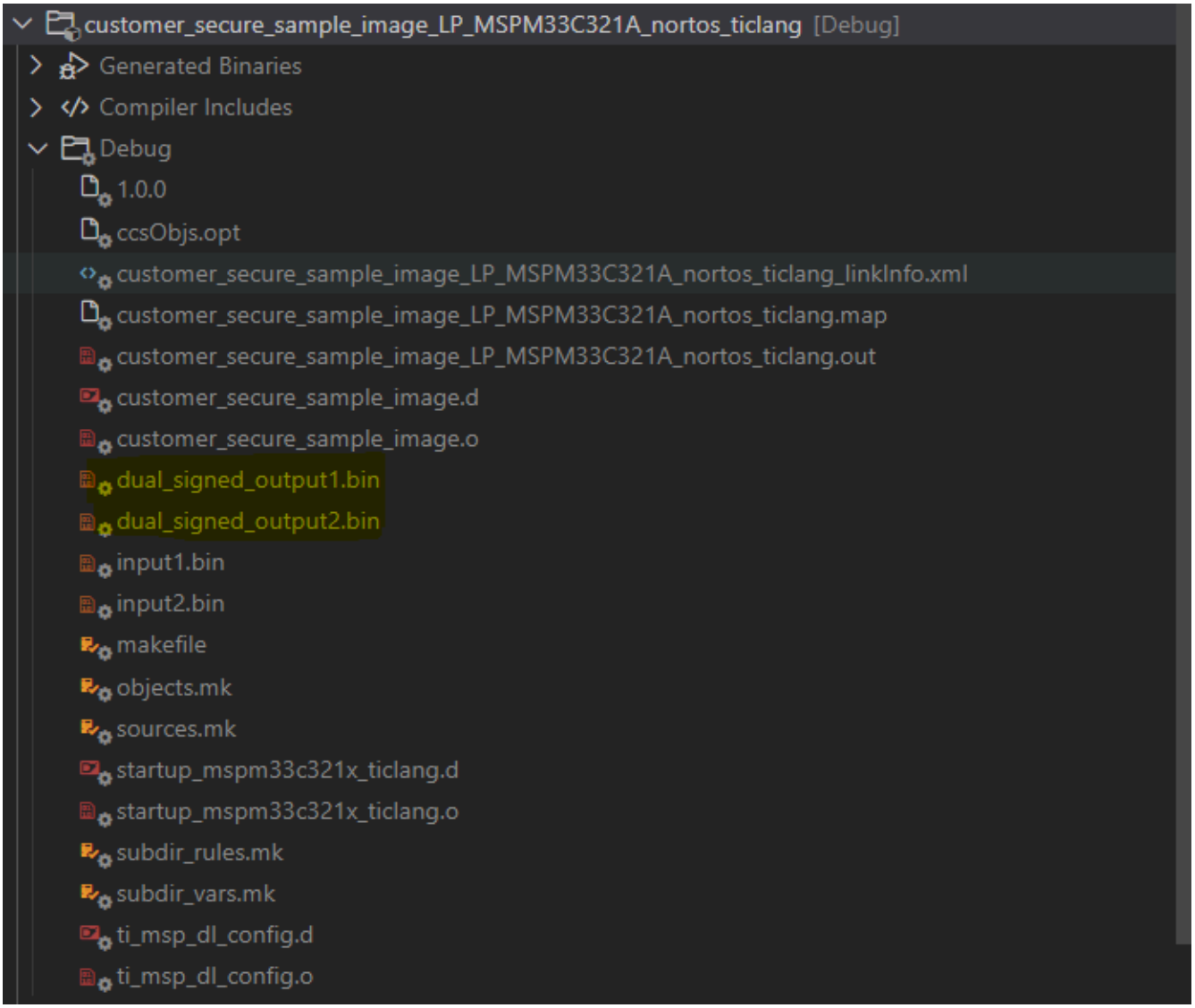


Figure 3-11. Debug Folder in the Project

3.3.4 Step 3 - Programming

Both application images with different versions are built with same address allocation (location 0x10000). Flash one application image at 0x10000 (bank0). Flash the second application image at 0x90000 (bank1).

1. Before flashing the CSC image, set the erase configuration to *Do not erase Flash memory* .

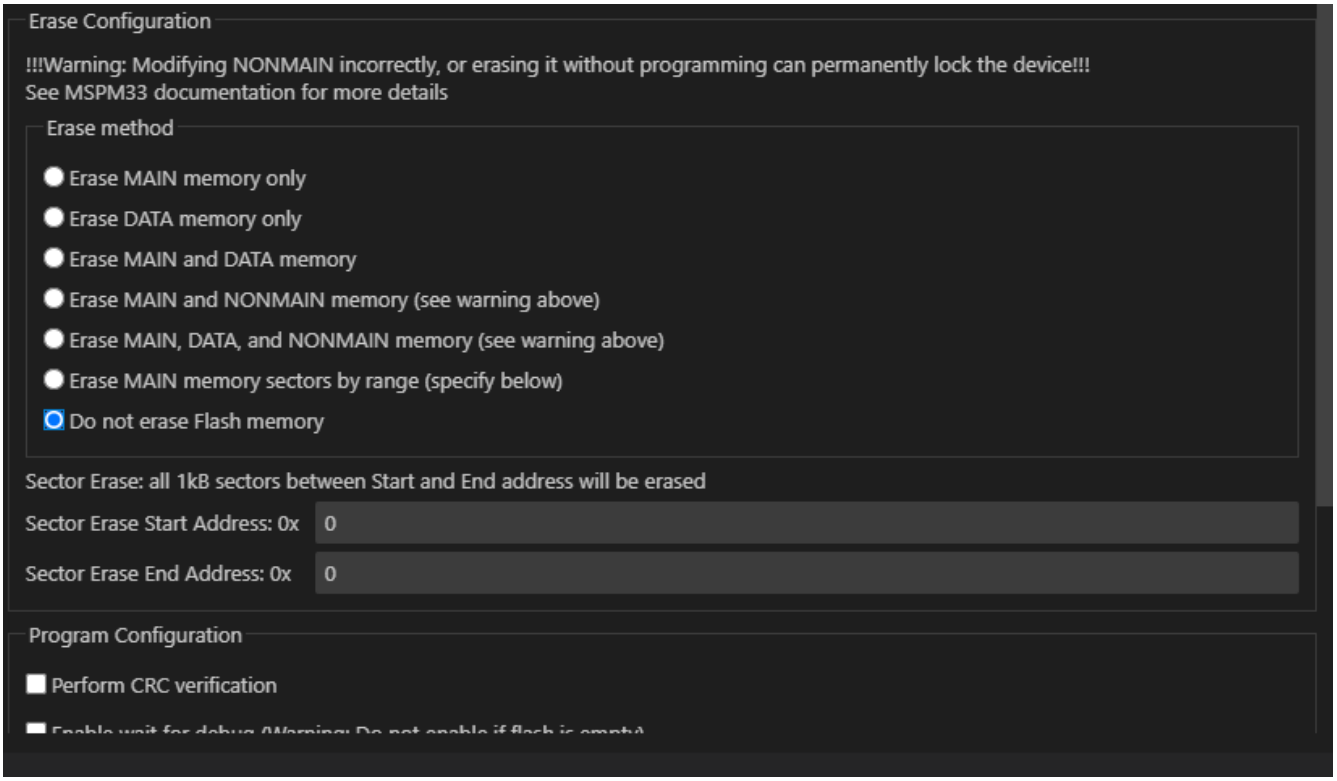


Figure 3-12. Configure '*Do not erase Flash memory*'

2. To Flash the "dual_signed_output1.bin" into the device, open the memory view and then import the file. The location of "dual_signed_output1.bin" is 0x10000. For reference use the images below for "dual_signed_output1.bin"

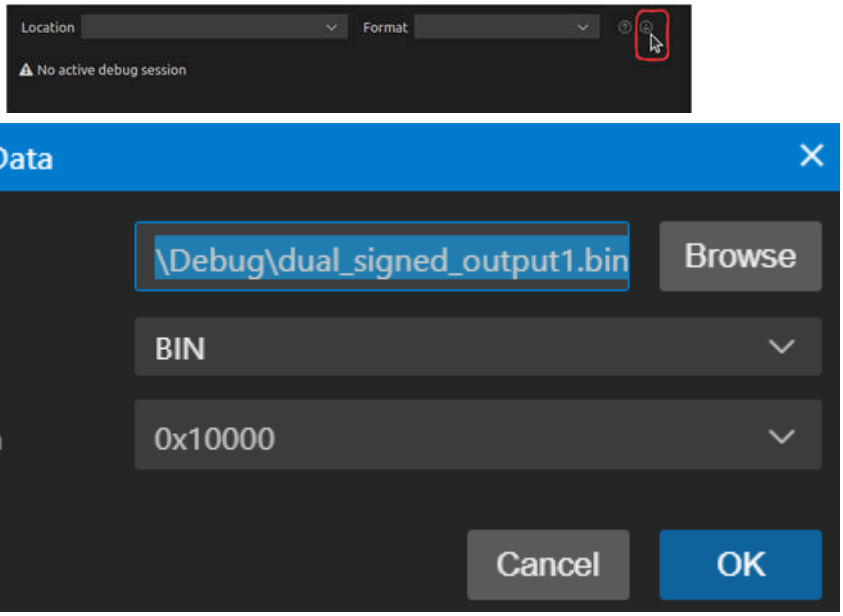


Figure 3-13. Import the Image in Memory View

3. To Flash the "dual_signed_output2.bin" into the project, open the memory view. The location of "dual_signed_output2.bin" is 0x90000.
4. After programming, find data in 0x0000 (Figure 3-14), 0x10000 (Figure 3-15) and 0x90000 (Figure 3-16) in memory view.

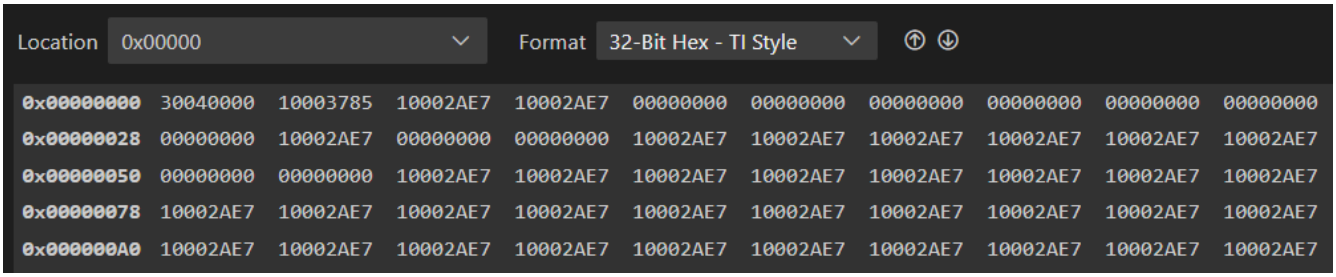


Figure 3-14. Memory View at 0x00000

Figure 3-15. Memory View at 0x10000

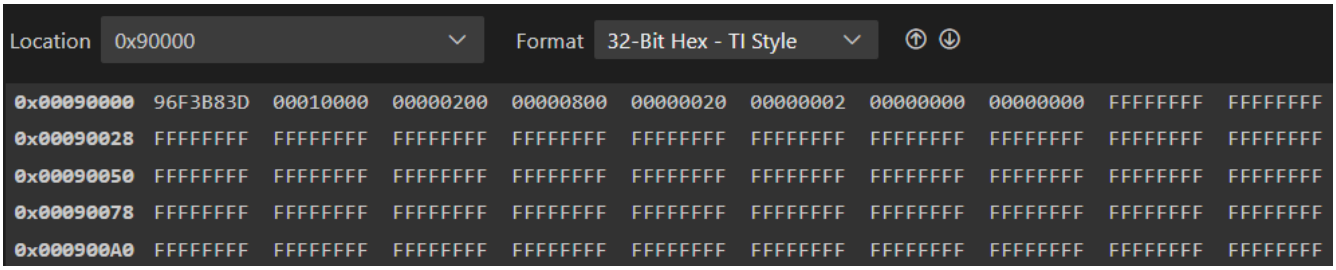


Figure 3-16. Memory View at 0x90000

3.4 Running the CSC Application

After loading the CSC, image 1 and image 2 and performing a boot reset, the CSC example is able to correctly authenticate the signed image 1 and image 2 using the keys present in boot_keys.c. Upon successful verification, the CSC jumps to this application where a blue LED is being set. On successful flashing and running of example, user can see a blue LED ON.

After the reboot, load the symbols of application that is being booted. **To load symbols into the application, navigate to run, then load, and select Symbols**. In the sample application, PA2 is being set, which can be observed after the whole CSC and sample images are flashed and the board is re-booted.

The two images we generated have different versions to show the update capabilities of this device. The versions were clarified in the last step of the post-build steps. "dual_signed_output1.bin" was given version 1.0.0 and "dual_signed_output2.bin" version 2.0.0, meaning that "dual_signed_output2.bin" in bank1 is running instead.

1. To check whether MCU enter APP already, please check your "call stack" menu and see if the device is running in the 0x10000 location. See the image below for an example.

Figure 3-17. Call Stack

2. To check the bank swap, After running the example, you can see the version information that was different at two banks (one at address 0x10000 and another at 0x90000) getting swapped.
   a. Swapping occurs after INIT_DONE is issued from CSC application.
   b. Before start of CSC code execution, version numbers can be seen:

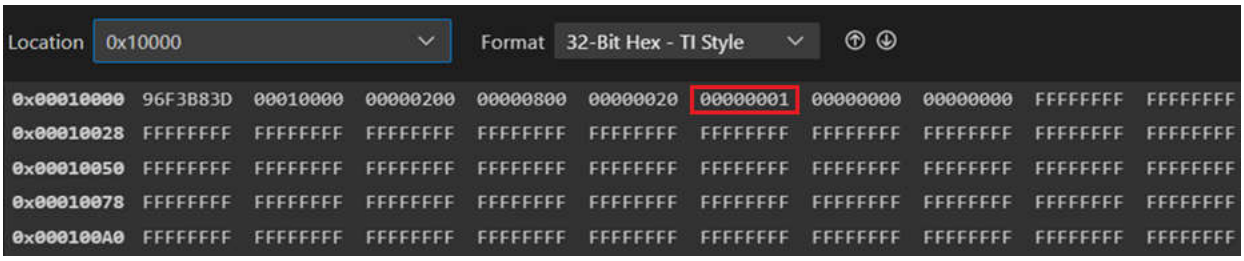


Figure 3-18. Memory View at 0x10000

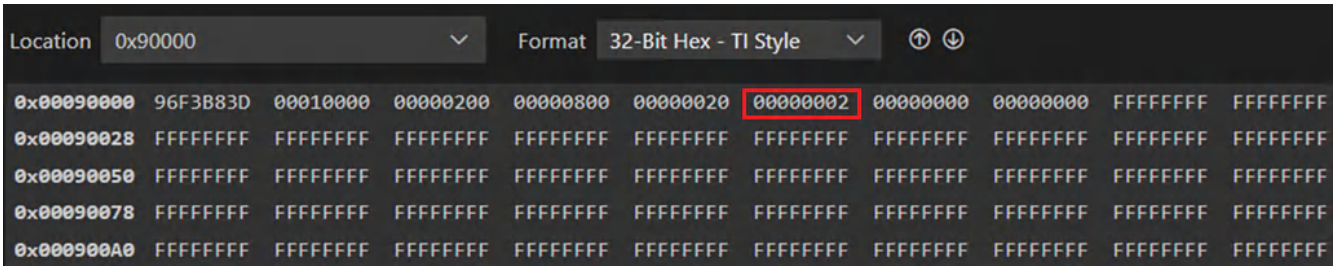


Figure 3-19. Memory View at 0x90000

   c. After CSC code execution, bank swapping is done:

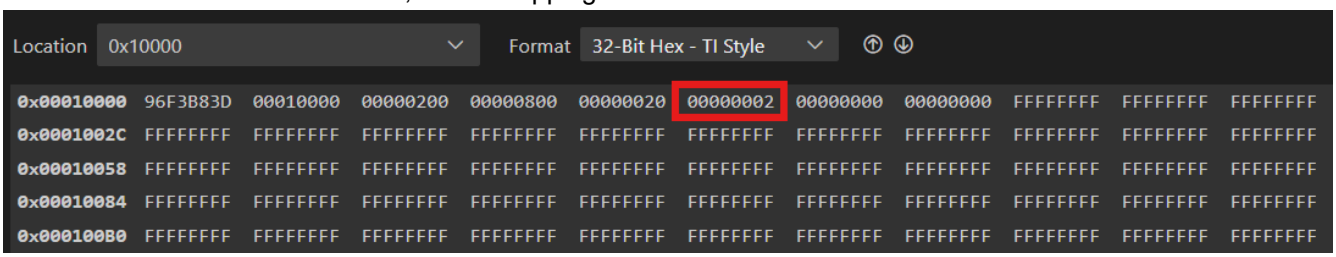


Figure 3-20. Memory View at 0x10000

Figure 3-21. Memory View at 0x90000

4 Q&A

4.1 How to Modify the Application Image Version

Users can modify the app version for the application update. Please follow the steps below.

4.1.1 Application Image

1. Modify the command for sign

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.0.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

2. For example, if the update version is v 1.1.0, then modify v 1.0.0 to v 1.1.0

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.1.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

4.2 How to Modify the Application Image Address

Users can modify the app image address for the custom application. Please follow the steps below

4.2.1 Application Image

1. Modify the .cmd file of the application image.

```
51  MEMORY
52  {
53    FLASH         (RX)  : origin = 0x00010200, length = FLASH_SIZE, fill = 0xFF,
54    SRAM            (RWX) : origin = 0x30000000, length = 0x0000FFFF
55
56  }
57
58  SECTIONS
59  {
60      .intvecs:  > 0x00010200
61      .text   : palign(16) {} > FLASH
62      .const  : palign(16) {} > FLASH
63      .cinit  : palign(16) {} > FLASH
64      .pinit  : palign(16) {} > FLASH
65      .rodata : palign(16) {} > FLASH
```

2. For example, if the update address is 0x30000. Modify 0x10200 to 0x30200

```
MEMORY
{
  FLASH         (RX)  : origin = 0x00030200, length = FLASH_SIZE, fill = 0xFF,
  SRAM            (RWX) : origin = 0x30000000, length = 0x0000FFFF

}

SECTIONS
{
    .intvecs:  > 0x00030200
    .text    : palign(16) {} > FLASH
    .const   : palign(16) {} > FLASH
    .cinit   : palign(16) {} > FLASH
    .pinit   : palign(16) {} > FLASH
    .rodata  : palign(16) {} > FLASH
```

4.2.2 Application Image Sign Command

1. Modify the command for sign.

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.0.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

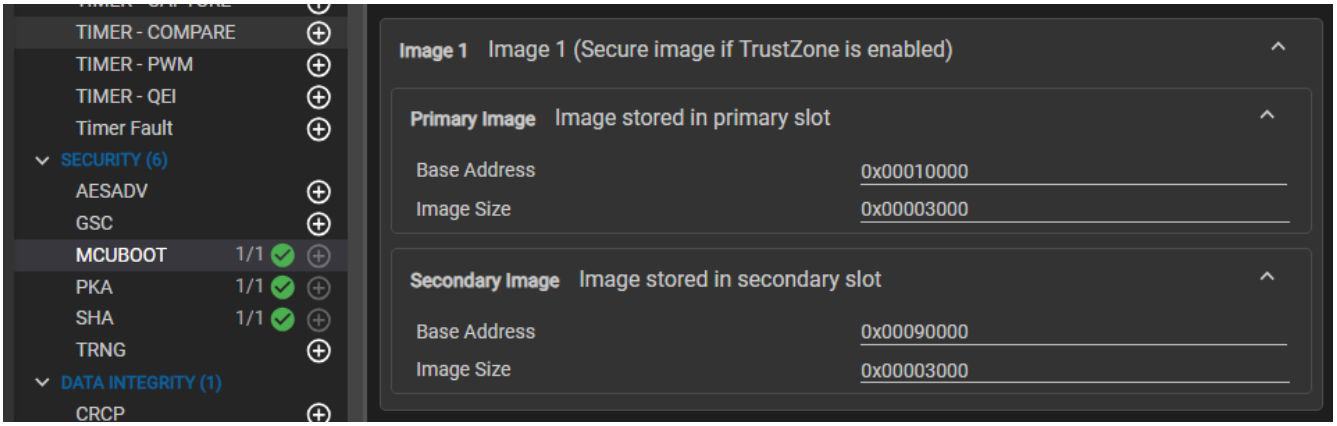
2. For example, if the update address is 0x30000. Modify 0x10000 to 0x30000

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.0.0 --slot-size 0x3000 --load-addr 0x30000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

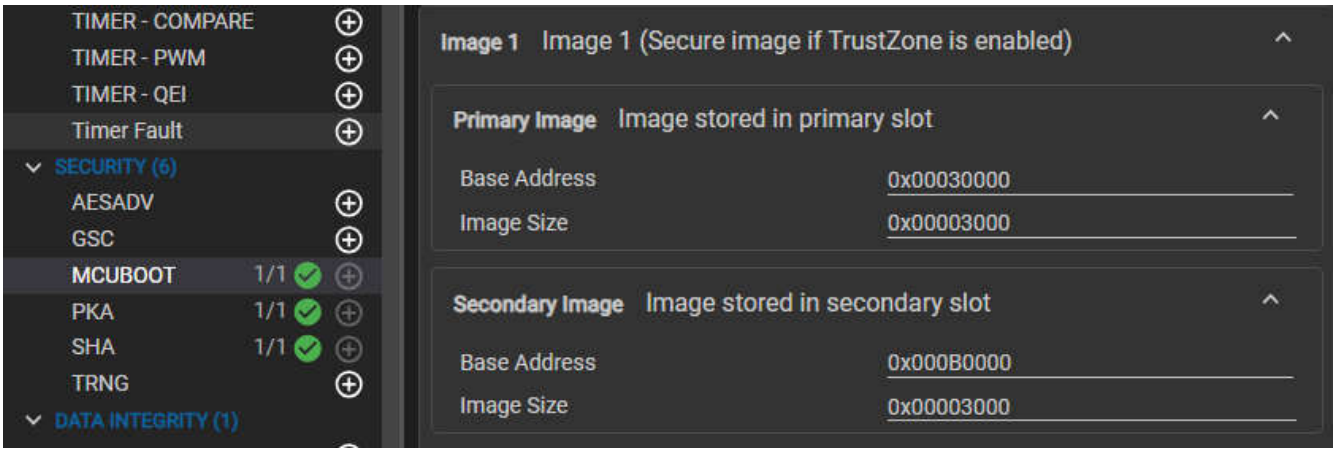
4.2.3 CSC Example Code

1. Modify address in customer_secure_config.h. Now user can configure the address using SysConfig.

```
#define CSC_APPLICATION_IMAGE1_BASE_ADDR                                (0x00010000)
#define CSC_APPLICATION_IMAGE1_SIZE                                     (0x00003000)
#define CSC_APPLICATION_IMAGE2_BASE_ADDR                                (0x00090000)
#define CSC_APPLICATION_IMAGE2_SIZE                                     (0x00003000)
```



2. For example, if the update address is 0x30000. Modify address of primary image from 0x10000 to 0x30000. Modify address of secondary image from 0x90000 to 0xB0000.
   a. Both application images with different versions are built for location 0x30000) but one needs to be flashed at bank0 and another application needs to be flashed at bank1
   b. Bank 0 – 0x00000 + 0x30000 = 0x30000
   c. Bank 1 – 0x80000 + 0x30000 = 0xB0000



4.3 How to Modify the Slot Size

The user may need a larger slot size for the app image.

4.3.1 Application Image

1. Modify the command for sign

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/
input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/
requirements.txt
pip install dilithium_py
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --
pad-header -v 1.0.0 --slot-size 0x3000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

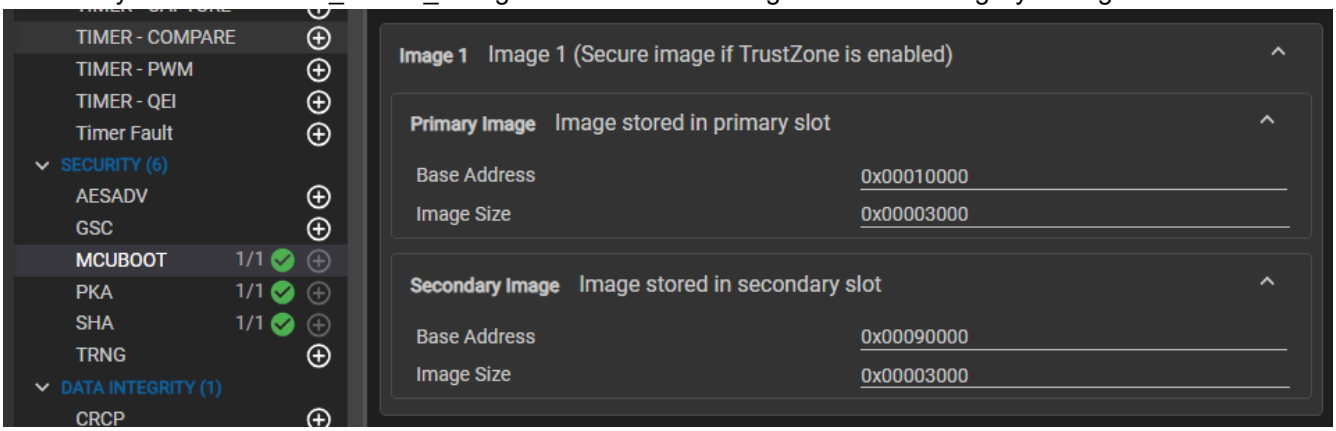
2. For example, if the update slot size is 0x8000. Modify 0x3000 to 0x8000

```
${CG_TOOL_OBJCOPY} -O binary ${PROJECT_BUILD_DIR}/${ProjName}.out ${PROJECT_BUILD_DIR}/input2.bin
pip install click cryptography intelhex cbor>=1.0.0
pip install -r ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/requirements.txt
pip install dilithium_py
```

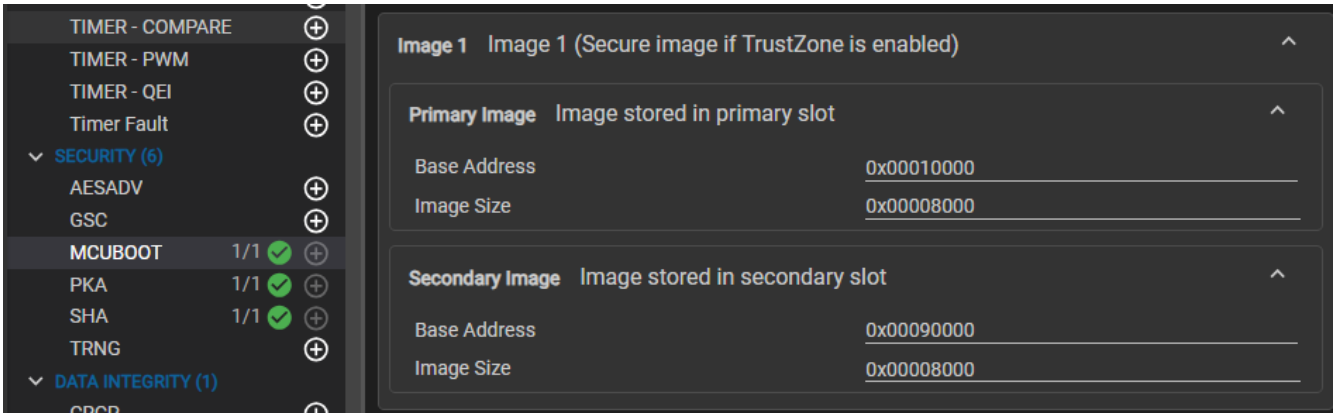
```
python ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/scripts/imgtool.py sign --
key ${COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/ecdsa-p256.pem --second-key $
{COM_TI_MSPM33_SDK_INSTALL_DIR}/source/third_party/mcuboot/mldsa_key.pem --header-size 0x200 --pad-
header -v 1.0.0 --slot-size 0x8000 --load-addr 0x10000 --align 8 --pad ${PROJECT_BUILD_DIR}/
input2.bin ${PROJECT_BUILD_DIR}/dual_signed_output2.bin
```

4.3.2 CSC Example Code

1. Modify size in customer_secure_config.h. The user can configure the size using Sysconfig.



2. For example, if the update size is 0x8000, modify 0x3000 to 0x8000.



4.4 How to Modify the Application Image Size

Users can configure the size of app image.

4.4.1 Application Image

1. Modify FLASH_SIZE in cmd file, which will affect the size of input_bin

```
48  /* Region of Flash that is read-protected after INITDONE */
49  //#define FLASH_SIZE    0x0800
50  #define FLASH_SIZE    0x60000
```

4.4.2 Modification on Slot Size

To modify the slot size, see How to Modify the Slot Size. Verify that the slot size is larger than the size of input_bin.

4.5 Doing More Customization on Application Image or CSC

For more instructions, please refer to *<MSPM33_SDK>\docs\english\middleware\boot_manager\Secure_Booting_Users_Guide* or Secure Booting User's Guide

5 References

- Texas Instruments, *MSPM33 C3-Series 160MHz Microcontrollers* technical reference manual
- Texas Instruments, *MSPM33C Security* user's guide
- Texas Instruments, *MSPM33 NONMAIN Flash Memory Configuration Guide*
- Texas Instruments, *Enable Trustzone for Secure Application in MSPM33 MCUs*
- *Texas Instruments, MSPM33 Customer Secure Code and Bootloader (CSC) User's Guide*
- *Image tool ReadMe, Imgtool.md*

6 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

DATE	REVISION	NOTES
August 2026	*	Initial Release

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025