Application Report

Secure BOOT on C2000 Device

Pramod Prabhakara, Karthik Rajakumar, and Christopher Chiarella

ABSTRACT

This application report discusses how the secure flash boot feature present on TMS320F2838x devices can be used to perform an application boot from flash with an additional security layer of boot code authentication before the actual code execution.

Table of Contents

List of Figures

List of Tables

Trademarks

C2000™ is a trademark of Texas Instruments.
Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
All other trademarks are the property of their respective owners.

1 Introduction

The TMS320F2838x is a powerful 32-bit floating-point Real-Time microcontroller designed for advanced closed-loop control applications such as industrial motor drives; solar inverters and digital power; electrical vehicles and transportation; sensing and signal processing. The device supports dual-core C28x architecture along with a new Connectivity Manager (CM) that offloads critical communication tasks, significantly boosting system performance. While the hardware efficiency of these Real-Time microcontrollers enables powerful applications, it is the code that creates the specific and unique application. The Dual Code Security Module (DCSM) built into this device to help you create a secure solution, contains a Secure Boot feature that enhances your ability to prevent unauthorized updates from running your application.

The DCSM implements a dual security zone concept, where the security configuration that is programmed in the One Time Programmable (OTP) region of the flash decides Zone1/Zone2/Unsecure allotment of securable resources (Flash sectors and RAMs). In addition to this, a Zone1/Zone2 resource can also be configured as EXEONLY, which allows only code execution from that region.

For more details, see the *DCSM* chapter of the *TMS320F2838x Technical Reference Manual* [1].

2 Secure Flash Boot Overview

One of the DCSM features related to the application flash boot is the ability to authenticate the user application code in flash before execution. This ascertains the integrity of the application code by ensuring that it has not been tampered with, after getting programmed into the Flash memory. When applied to a Zone1 EXEONLY Flash Sector, this feature acts as an additional layer of security for the critical user application code. The secure flash boot feature provides a set of additional boot options alongside the traditional flash boot options.

The secure flash boot is realized using the 128-bit AES-CMAC Authentication algorithm that is run on the application code contents returning a pass/fail status and proceeds to execute the application code only if the authentication succeeds. Table 2-1 gives an overview of this feature on the different subsystems of the device. The BootROM of each CPU subsystem initiates the authentication of the first 16KB of the application code of that subsystem, which is referred to as Primary Secure Boot. The authentication of the application code beyond the first 16KB of each CPU subsystem is referred to as the Extended Secure Boot. This can be optionally initiated by the pre-authenticated application code.

Due to the execution of the CMAC authentication algorithm during secure flash boot, the boot up sequence requires additional time to reach the user application compared to normal (non-secure) flash boot. Note that the device CM core secure flash boot requires less time compared to the CPU1 or CPU2 secure flash boot implementations since the CM makes use of a hardware AES accelerator.

Table 2-1. Secure Flash Boot Overview Across the Device

Subsystem (Core)	Secure Boot Feature	CMAC Algorithm Implementation	Additional Time Taken to Authenticate First 16KB of Flash Boot Code
CPU1 SS (C28_1)	Yes	Software (Secure ROM utility) + AES ROM tables	~400 ms (Running on INTOSC at 10 MHz)
CPU2 SS (C28_2)	Yes	Software (Secure ROM utility) + AES ROM tables	~20 ms (Running on PLL at 200 MHz)
CMSS (CM4)	Yes	Software (Secure ROM utility) + Hardware AES accelerator	~6 ms (Running on PLL at 125 MHz)

3 CMAC Authentication

Cipher-based Message Authentication Code (CMAC) is an AES-based authentication algorithm that constructs an authentication tag from a block of input data. The input data block is fed 128 bits at a time, into the crypto engine/software (based on the CPU subsystem), along with a 128-bit CMAC key. The key resides in locations 0x78018-0x7801F of the CPU1 USER OTP and is used by the CMAC authentication algorithm on all cores of the device. The crypto engine/software runs the CMAC algorithm on the entire block of input data and generates the final 128-bit authentication tag.

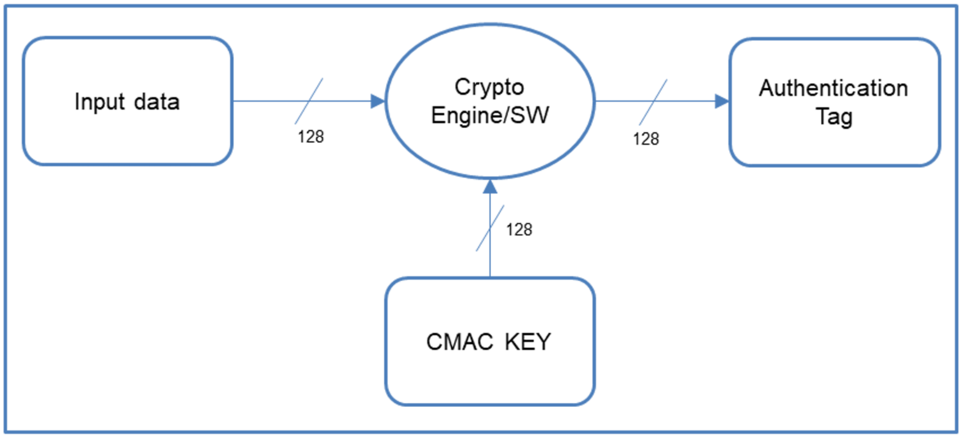


Figure 3-1. CMAC Operation

4 Secure Flash Boot Options

Table 4-1 shows the different Primary Secure Flash Boot options available on the three cores and their corresponding configurations. The Extended Secure Boot which is initiated by the application code, is explained in Section 7. For more details, see the *ROM Code and Peripheral Booting* chapter of the *TMS320F2838x Technical Reference Manual* [1].

Table 4-1. Secure Flash Boot Mode Configuration Details

Secure Boot Option [(1)]	BOOTDEFx/ BOOTMODE Value [(2)]	Flash Entry Point [(3)]	CPU1/CPU2 Entry Address	CPU1/CPU2 128-Bit Golden CMAC Tag Location	CM Entry Address	CM 128-Bit Golden CMAC Tag Location
Option 0	0x0A	Sector 0	0x00080000	0x00080002	0x00200000	0x00200004
Option 1	0x2A	Sector 4	0x00088000	0x00088002	0x00210000	0x00210004
Option 2	0x4A	Sector 8	0x000A8000	0x000A8002	0x00250000	0x00250004
Option 3	0x6A	Sector 13	0x000BE000	0x000BE002	0x0027C000	0x0027C004

(1) The secure boot options can be independently chosen for CPU1/CPU2/CM.

(2) For CPU1, BOOTDEFx field is part of the Zx-BOOTDEF-LOW/ Zx-BOOTDEF-HIGH CPU1 USER OTP memory locations. For CPU2/CM, BOOTMODE field is part of the CPU1TOCPU2IPCBOOTMODE/CPU1TOCMIPCBOOTMODE registers respectively populated by CPU1 application code.

(3) The secure boot feature is applicable only on Zone1 and hence the chosen flash sector(s) have to be configured as Zone1-EXEONLY. Also, irrespective of the sector size, the Primary Secure Flash boot operates on only the first 16KB of the selected sector. For example, Sector 4 and Sector 8 are 64KB each, but only the first 16KB is considered.

5 Secure Flash Boot Flow

Implementation of secure flash boot on device is a two-step process:

1. **Generation of the authentication tag** – this happens outside the device during image creation.
   a. The C2000™ or Arm®, hex utility runs the CMAC algorithm on the flash boot code image using the input CMACKEY and the CMAC application data structures that preserve the memory space for the golden CMAC authentication tag. For more details on the hex utility, see [3] and [4].
   b. The generated golden CMAC tag is embedded in the hex file at the location specified in Table 4-1.
   c. The hex image (now containing the golden CMAC tag) is programmed into the corresponding sector of the flash.
   d. The appropriate secure flash boot mode is chosen as per Table 4-1 and programmed in the CPU1 USER OTP.
2. **Authentication of the application boot code in flash** – this happens inside the device as part of the Secure Flash Boot execution
   a. The BOOTDEFx/BOOTPINCONFIG fields are configured to select the Secure Flash Boot option according to Table 4-1 and upon a reset, the device boots and execute the CMAC algorithm on the specified flash sector.
   b. The tag generated by the CMAC algorithm is compared with the Golden CMAC tag residing at the preprogrammed location.
   c. Upon a successful tag match, the boot process branches to the authenticated flash code and begins execution.
   d. Upon a tag match failure, different actions are taken on CPU1/CPU2/CM :
      i. In the case of CPU1, the device is reset (the code remains in a loop and XRSn is issued automatically on the Watchdog expiry).
      ii. In the case of CPU2, the secure boot failure flag is set in the CPU2TOCPU1IPCBOOTSTS register, IPC command is sent to CPU1 with secure flash CMAC error code, and the CPU2 boot code waits in a loop for CPU1 to take necessary action. A copy of the CPU2TOCPU1IPCBOOTSTS register is also captured in the 0x0000 0002 address location of CPU2.
      iii. In the case of CM, the secure boot failure flag is set in the CMTOCPU1IPCBOOTSTS register, IPC command is sent to CPU1 with secure flash CMAC error code and the CM boot code waits in a loop for CPU1 to take necessary action. A copy of the CMTOCPU1IPCBOOTSTS register is also captured in the 0x2000 0000 address location of CM.

Note

The CMAC algorithm, while calculating the authentication tag on the image and also while authenticating the image, treats the memory addresses containing the golden tag as all ones.

6 C2000Ware Example Details

In C2000Ware [2], an example is provided to show an application setup for secure flash boot. The example includes secure flash boot application projects for each core. The example additionally details how to authenticate flash code beyond the secure flash boot entry address + 16KB. More details on this custom flash range authentication functionality are explained in the Section 7. This example assumes that the flash sectors to be authenticated are pre-configured as Zone 1 EXEONLY and uses the default CMACKEY for authentication. For details on programming a custom CMACKEY and other DCSM settings in CPU1 USER OTP, see [1] and [2].

C2000Ware Location: <C2000Ware_Install_Directory>/driverlib/f2838x/examples/c28x/boot

Project Names:

- boot_ex1_cpu1_cpu2_cm_secure_flash_cpu1
- boot_ex1_cpu1_cpu2_cm_secure_flash_cpu2
- boot_ex1_cpu1_cpu2_cm_secure_flash_cm

Files Included:

- Source files – Includes main application code
  - Example: boot_ex1_cpu1_cpu2_cm_secure_flash_cpu1.c
- HEX Linker Command files – Provides details of the entire length of flash memory to the c2000 or arm hex utility
  - Example: boot_ex1_flash_hex_lnk_cpu1.cmd
- CMAC key text file – Provides the user CMAC key to the c2000 or arm hex utility
  - Example: boot_ex1_user_cmac_key.txt
  - For more details on the cmac_key format, see [3] and [4].

How to run the example:

1. Load application into CPU1 flash (as well as CPU2 and CM applications).
   a. Load the *.hex file, not the *.out file
2. Disconnect and reconnect to only CPU1.
3. To configure the device to perform secure flash boot upon boot up:
   a. Emulation boot (recommended for example/development)
      i. In CCS memory window, set BOOTPINCONFIG location (0x0D00) to 0x5AFFFFFF and BOOTDEF location (0x0D04) to 0x0000000A
   b. Standalone boot (recommended for deployment)
      i. Program CPU1 USER OTP locations corresponding to BOOTPINCONFIG and BOOTDEF. To learn more, see [1].
4. Reset CPU1 via CCS and click resume.
5. Observe the LEDs on the controlCARD for indicators of success.
   a. When all three cores secure boot successfully and authenticate their full flash memory contents, then three LEDs (one for each core) will be blinking.
6. For cases where the controlCARD isn't used, look out for the following GPIO toggles based on the CPU subsystem:
   a. CPU1 – GPIO31
   b. CPU2 – GPIO34
   c. CM – GPIO145

Application code requirements for golden CMAC Tag generation:

CPU1/CPU2 Golden CMAC Tag Memory Allocation for Secure Flash Boot Option 0

```
// Implementation for CPU1/CPU2
#pragma RETAIN(cmac_sb_1)
#pragma LOCATION (cmac_sb_1, 0x080002)
const char cmac_sb_1[8] = {0};
```

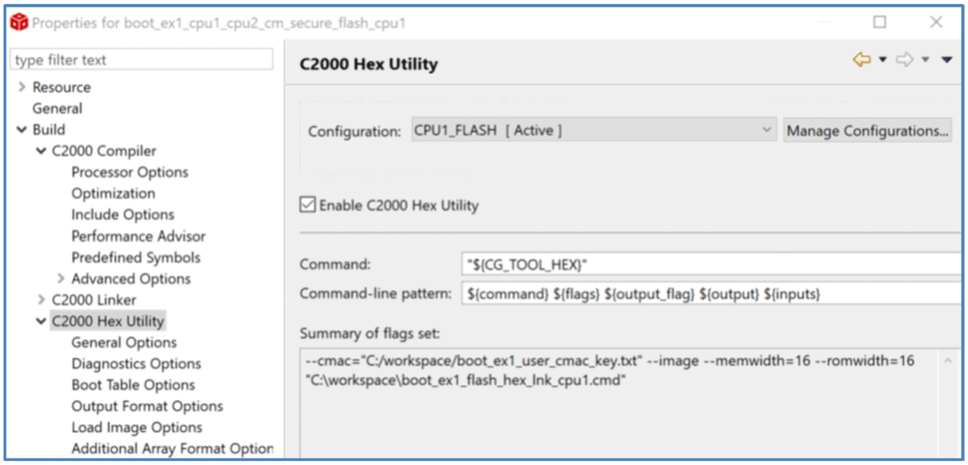
CM Golden CMAC Tag Memory Allocation for Secure Flash Boot Option 0

```
#pragma RETAIN(cmac_sb_1)
#pragma LOCATION (cmac_sb_1, 0x00200004)
const uint8_t cmac_sb_1[16] = {0};
```

Constant char/unsigned integer definitions allocate memory for golden CMAC tags. For more information, see the examples in Section 6:

- Variable naming must be one of the following: cmac_sb_1, cmac_sb_2, cmac_sb_3, cmac_sb_4
  - Further details for C28x can be found in the *TMS320C28x Assembly Language Tools User's Guide* [3]
  - Further details for CM can be found in the *ARM Assembly Language Tools User's Guide* [4].
- Use the LOCATION pragma to specify the address within the intended authentication range for the CMAC golden tag. For CPU1/CPU2, this address must be the entry address + 2 and for CM, this address must be the entry address + 4.
- Leave the variable initialized to zero.

Application setup for using HEX Utility:



Configuration: CPU1_FLASH [Active]

☑ Enable C2000 Hex Utility

Command: "${CG_TOOL_HEX}"

Command-line pattern: ${command} ${flags} ${output_flag} ${output} ${inputs}

Summary of flags set:

```
--cmac="C:/workspace/boot_ex1_user_cmac_key.txt" --image --memwidth=16 --romwidth=16
"C:\workspace\boot_ex1_flash_hex_lnk_cpu1.cmd"
```

Figure 6-1. CPU1 Example Properties With Hex Utility Enabled for CMAC

Each core project enables the hex utility to generate the golden CMAC tag (see Figure 6-1). Flags include:

- “--cmac” provides the path to the user CMAC key text file
- “--image”, “--memwidth”, and “--romwidth” where mem/rom width is set. This should be set to 16 for CPU1/CPU2 and set to 8 for CM
- The paths to the flash HEX linker command file for the corresponding core.

7 Authenticating Flash Code Beyond 16 KB

Thesecure flash boot only authenticates the first 16 KB of the flash sector from the entry address. In order to authenticate other sectors of flash, the user application must call the secure flash boot CMAC API directly.

Using the hex utility, it supports generation of golden CMAC tags for each of the four flash entry addresses + 16KB and 1 custom flash range. The custom flash range is configurable to be able to perform CMAC authentication over a custom address range. This could be the length of all the flash sectors if desired.

CPU1/CPU2 Application CMAC Structure for Custom Flash Range Authentication

```
struct CMAC_TAG
{
   char tag[8];
   uint32_t start;
   uint32_t end;
}
```

CPU1/CPU2 Golden CMAC Tag Memory Allocation for Full Flash Range Authentication

```
#pragma RETAIN(cmac_all)
#pragma LOCATION (cmac_all, 0x087002)
const struct OMAC_TAG cmac_all = {{0}, 0x0, 0x0};
```

CM Application CMAC Structure for Custom Flash Range Authentication

```
struct CMAC_TAG
{
   uint8_t tag[16];
   uint32_t start;
   uint32_t end;
}
```

CM Golden CMAC Tag Memory Allocation for Full Flash Range Authentication

```
#pragma RETAIN(cmac_all)
#pragma LOCATION (cmac_all, 0x00204004)
const struct OMAC_TAG cmac_all = {{0}, 0x0, 0x0};
```

Create a structure called “cmac_all” for creating a custom CMAC authentication range.

- Use the LOCATION pragma to specify any address within the intended authentication range for the CMAC golden tag. (The closer the CMAC golden tag is to the authentication start address, the faster the CMAC algorithm will execute).
- Leave the “tag” struct element to always initialize to zero.
- Initialize the “start” and “end” struct elements to set a custom range. If both are zero, then the full length of the device core flash memory will be authenticated including the 16KB memory range of Primary Secure Boot.
- For additional details on the application CMAC variables/structs, see [3] and [4].

CPU1 Secure Flash CMAC Authentication API

```
applicationCMACStatus = CPU1BROM_calculateCMAC (CMAC_AUTH_START_ADDRESS,
                                                 CMAC_AUTH_END_ADDRESS,
                                                 CMAC_AUTH_TAG_ADDRESS);
```

In the application, include F2838x secure zone code symbols library to your project (Located in C2000Ware [2] at <C2000Ware_Install_Directory>/libraries/boot_rom/f2838x) and call the secure flash boot CMAC API for the applicable core. An example API call on CPU1 is shown in the example above.

- The CMAC_AUTH_TAG_ADDRESS should match what was provided to the LOCATION pragma
- The CMAC_AUTH_START_ADDRESS and CMAC_AUTH_END_ADDRESS should match what was provided to the “cmac_all” struct. Additionally, it is important to note that the start and end address should align to 128 bits.

- For example, the start and end address for authenticating the full length of F2838x CPU1 flash memory would be:
  - Start: 0x00080000
  - End: 0x000C0000
- The status returned from the secure flash boot CMAC API should then be checked and handled accordingly in the user application.
- For additional details on the Secure Flash CMAC Authentication APIs, see the *ROM Code and Peripheral Booting* chapter of [1].

Note

Even though the end address in the above example is actually 0x000BFFFF, it must be provided as 0x000C0000 so that the end address is aligned to 128 bits.

8 Debug Resources

Table 8-1. Secure Flash Boot Debug Scenarios

Scenario	Behavior
Failed Secure boot in CPU1	Standalone Boot: Device is reset. Emulation Boot: CPU1 halts inside the address range 0x3FB13C – 0x3FB142
Failed Secure boot in CPU2	Bit21 of the CPU2TOCPU1IPCBOOTSTS [1] register will be set. CPU2 sends IPC command to CPU1 with secure flash CMAC error code.
Failed Secure boot in CM	Bit21 of the CMTOCPU1IPCBOOTSTS [2] register will be set. CM sendsIPC command to CPU1 with secure flash CMAC error code.
Has a successful Secure boot run for CPU1?	Bits7:0 of the CPU1 BootROM status residing in the 0x0000 0002address location will reflect 0x3.
Has a successful Secure boot run for CPU2?	Bits7:0 of the CPU2TOCPU1IPCBOOTSTS [1] register will reflect 0x3.
Has a successful Secure boot run for CM?	Bits7:0 of the CMTOCPU1IPCBOOTSTS [2] register will reflect 0x3.

1. Same information is also captured in the 0x0000 0002 address location of CPU2.
2. Same information is also captured in the 0x2000 0000 address location of CM.

9 Additional Information and Points to Consider

- Although not recommended, if secure flash boot is performed on CPU2/CM and not on CPU1, then a dummy load must be performed from CPU1 for the Z1 OTP CMACKEY before releasing CPU2/CM out of reset. The dummy load is done by reading the 0x78018-0x7801F locations of CPU1 USER OTP.
- Similarly, if using the CMAC Authentication APIs without running the secure flash boot mode on CPU1, then a dummy load must be performed from CPU1 for the Z1 OTP CMACKEY before calling the APIs.
- While using Secure Flash Boot on CPU1, it is recommended not to have a normal (non-secure) Flash Boot mode to the same sector configured in the BOOTDEF table.
- For authenticating flash code beyond 16 KB:
  - The 128-bit golden CMAC tag must be stored inside of the memory address range that the calculation is performed on.
  - The starting address of the golden CMAC tag must align to a 32-bit boundary.
- As the boot mode settings reside in the One Time Programmable (OTP), it is recommended to make use of the emulation boot mode for trials before freezing the boot related configuration. More information regarding the Emulation boot is provided in the *ROM Code and Peripheral Booting* chapter of the *TMS320F2838x Technical Reference Manual* [1].

10 References

1. Texas Instruments: *TMS320F2838x Real-Time Microcontrollers Technical Reference Manual*
2. C2000Ware for C2000 Real-Time MCUs
3. Texas Instruments: *TMS320C28x Assembly Language Tools v20.2.0.LTS User's Guide*
4. Texas Instruments: *Arm Assembly Language Tools v20.2.0.LTS User's Guide*
5. TMS320F28338D Product Page

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, 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 (www.ti.com/legal/termsofsale.html) 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.

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