Secured bootloading implementation in PSOC™ 4 HVPA-SPM

About this document

Scope and purpose

This document explains how to implement secured bootloader and secured boot in PSOC™ 4 High Voltage Precision Analog Smart Battery Pack Monitor (HVPA-SPM) 1.0 devices, using device security features.

Intended audience

The intended audience for this document is engineers who want to develop secured bootloading in PSOC™ 4 HVPA-SPM 1.0 device.

This is an intermediate-level application note and assumes that you have basic familiarity with PSOC™ HV devices and embedded security concepts.

Key points

- Explains the implementation of secured bootloading and secured boot using PSOC™ 4 HVPA-SPM device security features
- Describes a boot manager, bootloader, and application architecture with memory partitioning and flash protection mechanisms
- Demonstrates host authentication using challenge-response mechanism with AES-CMAC and TRNG
- Describes verification of application integrity using CMAC
- Provides code examples and workflows for integrating cryptography and flash operations to implement secured boot and secured bootloader

About this product family

Product family

Infineon's PSOC™ 4 High Voltage Precision Analog (HVPA) is a cutting-edge device that has been fully integrated into a programmable embedded system, providing outstanding battery monitoring and management capabilities for automotive applications (AEC-Q100 qualified). This device boasts an Arm® Cortex® M0+ processor, high precision analog subsystems, and high voltage subsystems, which ensure precise and accurate battery monitoring

Target applications

- Automotive battery management system (BMS)

Table of contents

Table of contents

1 Introduction

In automotive systems, devices often receive software updates to fix bugs, add features, etc. Without proper safeguards, these updates are vulnerable to cybersecurity attacks, such as tampering or unauthorized access. This document will explore two mechanisms using the device's security features:

- **Secured bootloading:** Verifies that software updates are received from an authenticated source and have not been corrupted or manipulated
- **Secured boot:** Verifies the integrity of the application before execution, preventing modified or malicious code from executing on the device

1.1 Software over-the-air (SOTA) updates

Initial programming of software into a product is typically implemented through the device's programming interface (such as Serial Wire Debug (SWD) or Joint Test Action Group (JTAG)). However, these interfaces in the field are often not accessible as it can be difficult and expensive to open up the product and directly access the PCB.

SOTA updates enable remote software updates without physical access to the device. This document will focus on the implementation of SOTA updates.

Although SOTA updates often imply wireless transfer, many systems use a hybrid architecture: a central host electronic control unit (ECU) that receives the update wirelessly and distributes it to target ECUs over wired in-vehicle networks (for example, CAN or LIN). In battery management systems (BMS), this central host is commonly known as the battery management unit (BMU), as shown in Figure 1.

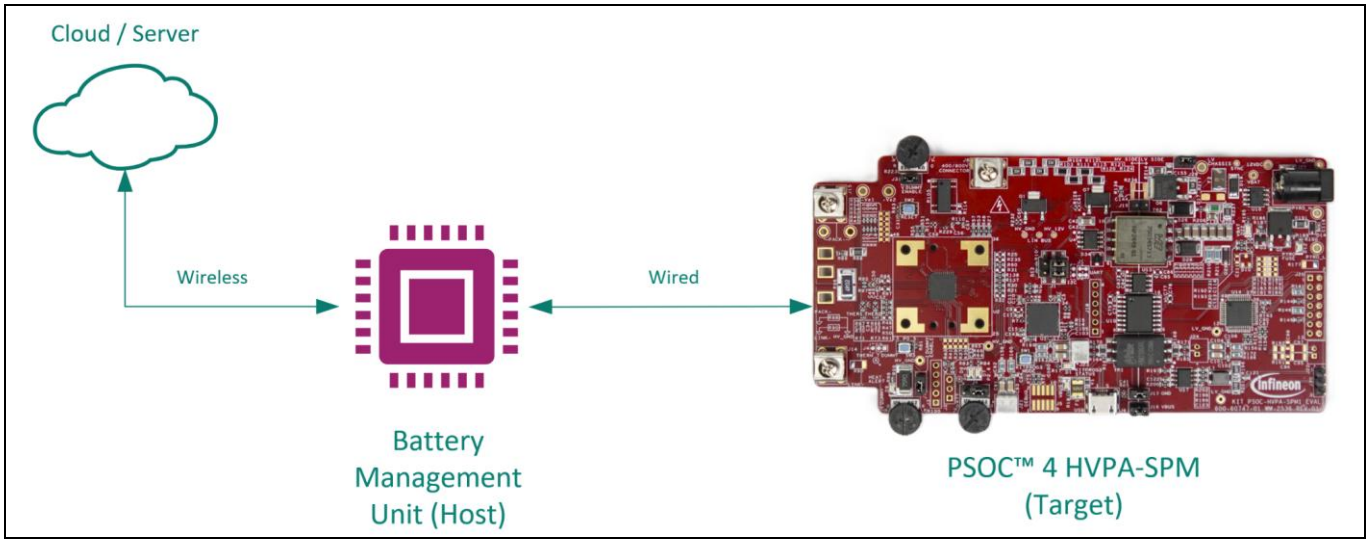

Figure 1 SOTA update

1.2 Bootloader

A bootloader is a small program that makes it possible for a product's software to be updated in the field. It is an integral software component of SOTA update. The bootloader receives an image (of the software update) over a communication interface (for example, CAN or LIN), performs integrity/authenticity checks as defined by the security design, and programs the image into flash memory. The bootloader removes the need for direct use of debug/programming interfaces for software updates in the field.

A few terms that will be used in the context of bootloaders are defined as follows:

- **Target:** The device that receives and programs the software update
- **Host:** The entity that sends the software update to the target
- **Bootloadable:** The software image sent by the host to the target during a bootloading operation. It is also referred to as the application

1.3 Why should boot and bootloader be secured?

The communication channel between the host and the target is a potential attack surface. An attacker could impersonate the host and push malicious software to the target (spoofing) or manipulate the update. For these reasons, the bootloading process should be secured.

After a software update, the stored application could be compromised. Unauthorized flash modification is possible if the Debug Access Port (DAP) is not disabled or a communication error could interrupt the bootloading process, leaving the application in an inconsistent state. Secured boot supports authenticated booting of the user application software. This is implemented by checking the integrity and/or authenticity of the application.

A few of the security goals for secured bootloading are as follows:

- Protect the update mechanism so only authorized entities can deliver software updates to the target
- Check if the application is not corrupted during the bootloading process
- Check if the application is not corrupted during the boot

2 Device features

This section briefly describes the security features of the PSOC™ 4 HVPA-SPM 1.0 that are used in the implementation of secured bootloading. For complete details, see the architecture reference manual.

2.1 Device modes

After boot, the device operates in one of the following modes: OPEN, PROTECTED, or KILL. Each mode provides specific capabilities for the CPU software and debug over DAP. Figure 2 shows the possible transitions between the modes.

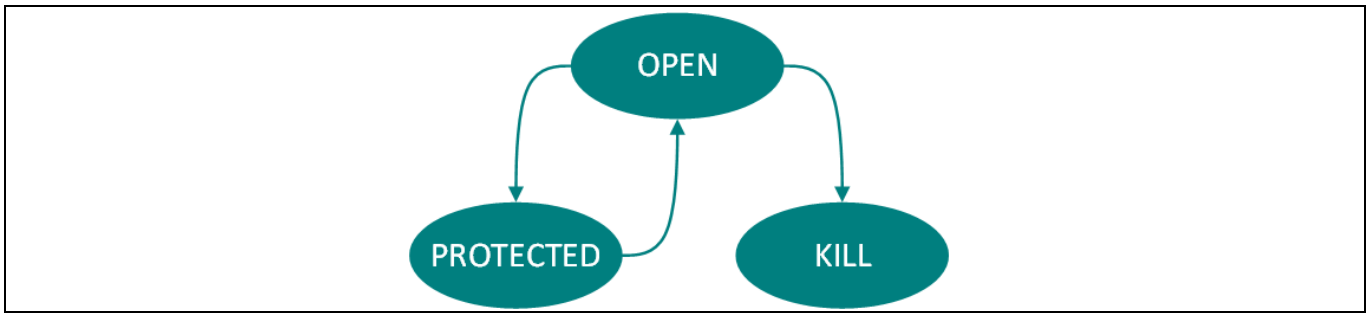

Figure 2 Transitions between device modes

The transitions between the modes are achieved by using the system call '*Write Protection*' (see "Systems calls" section in the architecture reference manual for more details). The transition of device mode takes effect after a reset.

Note: The transition from OPEN to KILL is irreversible.

2.2 Flash row protection

Flash memory is organized into rows. The flash row protection prevents internal (via CPU) or external (via DAP) write operations. This feature is implemented by the system calls '*Load Flash Bytes*' and '*Write Protection*'. Table 1 summarizes the row protection capabilities.

Table 1 Flash row protection

Row protection state	Allowed by CPU	Allowed by DAP
Unprotected	Read, write	Read, write
Full protection	Read	Read

In addition to flash row protection, transitioning the device to PROTECTED prevents read access to the flash over DAP. In PROTECTED mode, only a few registers are accessible (such as registers to execute a system call). This mode also disables debug access to user code or memory.

The row protection bits can be programmed when the device transitions from 'OPEN' mode only. When the device transitions from 'PROTECTED' to 'OPEN', the row protection bits are cleared (to '0').

2.2.1 Protection bit layout for SFlash

The code flash organization for PSOC™ 4 HVPA-SPM 1.0 is as follows:

- Code flash size: 128 KB
- Number of code flash macros: 2
- Macro size: 64 KB
- Flash row size: 128 bytes
- Rows per macro: 512
- Total rows in code flash: 1024

As each row maps to one protection bit, there are 512 protection bits per macro (64 bytes).

The flash row-level protection settings are programmed separately for each flash macro in the device. The pointers to the protection data location are stored in SFlash at address 0x0FFFE020.

In the case of PSOC™ 4 HVPA-SPM 1.0, the pointer value is 0xE0C0E080. As the SFlash base address is 0x0FFE0000, the offsets for row protection are 0xE080 and 0xE0C0 for each macro in the code flash.

Table 2 shows the address ranges in SFlash that store the row protection data for each macro.

Table 2 Flash row protection, address ranges in SFlash

SFlash	Code flash	
Location of row protection data	**Macro**	**Rows**
0x0FFFE080 to 0x0FFFE0BF	0	0 to 511
0x0FFFE0C0 to 0x0FFFE0FF	1	512 to 1023

Table 3 shows examples on the values that needs to be set in the SFlash (for full protection row state):

Table 3 Flash row protection examples

Code flash		SFlash	
Row to be protected	**Macro**	**Address**	**Value**
0	0	0x0FFFE080	0x01
1	0	0x0FFFE080	0x02
511	0	0x0FFFE0BF	0x80
512	1	0x0FFFE0C0	0x01
1023	1	0x0FFFE0FF	0x80

Note: Flash row protection does not support data flash.

2.2.2 Implementation of flash row protection

To implement flash row protection, follow these steps:

- Initialize the clock for flash operations (charge pump clock - clk_pump)
- Execute '*Load Flash Bytes*' system call to load row-protection data into page latch buffer
- Execute '*Write Protection*' system call to program the protection bits from the page latch buffer to the SFlash

Table 4 shows an example of the values set for Load Flash Bytes system call.

Table 4 Flash row protection examples

SRAM address/parameters	Values	Notes
SRAM address + 0x0000		
Key1	0xB6	–
Key2	0xD7	–
Byte address	0x00	–
Flash macro select	0 (Macro 0)	As the row protection data of both the macros of the code flash reside in the SFlash (Macro 0), the value is always '0'
Flash controller number	0 (Flash controller 0)	
SRAM address + 0x0004		
Load size	0x3F	Size of the row protection data per macro
SRAM address + 0x0008		
Data byte [0]	...	Row protection data per macro
...	...	
Data byte [63]	...	

Table 5 shows an example of the values set for Write Protection system call:

Table 5 Flash row protection - examples

CPUSS_SYSARG parameters	Values	Notes
Key1	0xB6	–
Key2	0XE0	–
Device protection byte	0x01 (OPEN) 0x02 (PROTECTED) 0X04 (KILL)	–
Flash macro select	0 (Macro 0) 1 (Macro 1)	Select the code flash macro whose protection bytes will be loaded
Flash controller number	0 (Flash controller 0)	Controller of the code flash

Note: Flash row protection does not support data flash.

2.3 Cryptography block

PSOC™ 4 HVPA-SPM 1.0 includes a hardware cryptography block with the following primitives:

- Advanced Encryption Standard (AES) per FIPS 197 standard
- Secure Hash Algorithm (SHA) per FIPS 180-4 standard
  - SHA1
  - SHA224
  - SHA256
- Cyclic redundancy check (CRC)
- Pseudo random number generator (PRNG)
- True random number generator (TRNG)

Additionally, AutoPDL driver provides a library for AES-CMAC operations.

The cryptography block is an enabler for secured bootloading and secured boot because it allows security operations to be executed in hardware rather than in software. This reduces CPU overhead, improves execution time, and helps implement cryptographic operations, such as encryption, authentication, and integrity verification.

For more information on cryptographic capabilities, see the “Cryptography block” section in the architecture reference manual.

3 Secured bootloading concept

For the bootloader implementation, the following demonstration setup is used as shown in Figure 3:

- Target: PSOC™ 4 HVPA-SPM 1.0
- Host: Computer instead of BMU
- Communication channel: UART

Note: This concept can be adapted to other communication interfaces such as CAN or LIN.

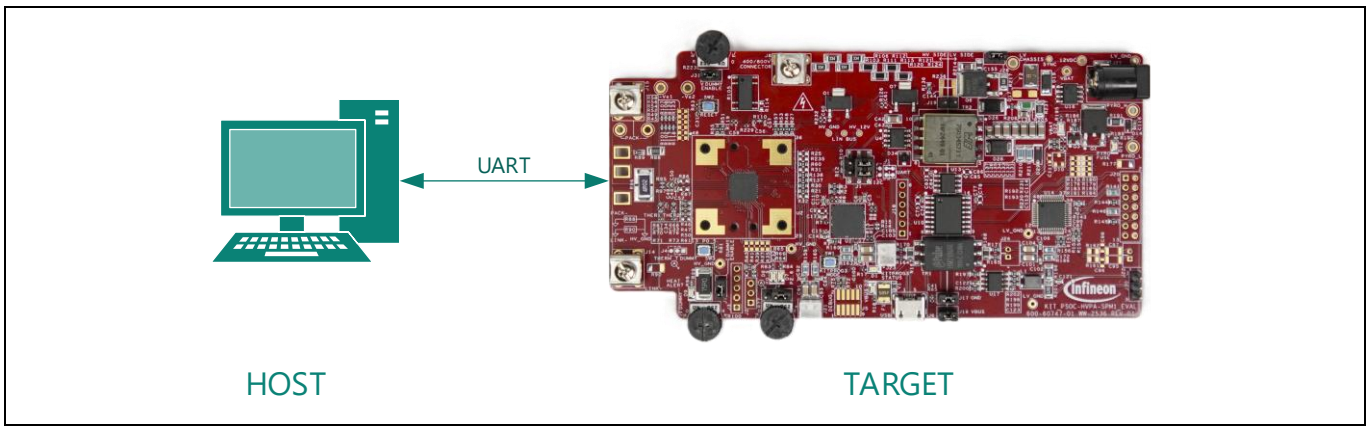

Figure 3 Bootloading demonstration setup

A typical application for SOTA updates or bootloading consists of three primary software components:

- **Boot manager:** Selects whether to launch the bootloader or the application after reset
- **Bootloader:** Responsible for receiving, validating, and programming new application image into code flash memory
- **Application:** Program that executes the intended system functionality

The boot manager is the primary user software component that executes after reset. It determines whether to launch the bootloader or execute the application. It is up to the integrator to implement mechanisms that control the execution flow (such as stay in the bootloader or execute the application). The boot manager can use a hardware signal, such as a GPIO input, or a command through a communication interface to decide whether to execute the bootloader.

A better design of the boot manager involves checking the integrity of the application before executing the application. In some bootloader designs, even if the application is being executed, it will periodically check if there is a request for a bootloader mode.

In this application note, the boot manager makes the decision to execute the bootloader based on a UART command. After every reset, the boot manager will also check the integrity of the application before executing it.

3 Secured bootloading concept

3.1 Memory partitioning

The software components must be organized in flash memory so that the boot manager and bootloader are protected from modification, while the application area remains writable for updates. The memory is allocated based on the software component size.

In PSOC™ 4 HVPA-SPM 1.0 device, the code flash consists of two macros, each 64 KB. For ease, the software components (boot manager and bootloader) that should be protected from write operations can be placed in Macro 0 and the application can be placed in macro 1, as shown in Figure 4. There are two software programs:

- Secured software
  - Includes the boot manager and bootloader
  - Program is placed in Macro 0
- Application software
  - User application program
  - Program is placed in Macro 1

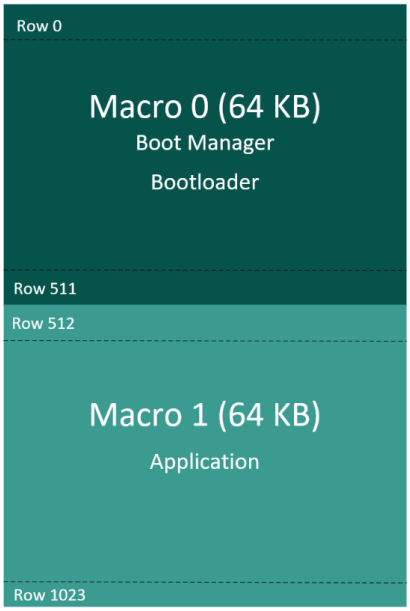

Figure 4 Code flash organization for the software components

3.2 Boot manager

The boot manager is placed in Macro 0 of the code flash, and it is the first part of the user software that executes after reset. It performs the following functions:

1. Performs required system initializations
2. Receives a command from the host to execute the bootloader or the application
3. Authenticates the host before starting the bootloader (host authentication)
4. Verifies the integrity of the application before executing it (secured boot)

Figure 5 shows the high-level program flow of the boot manager.

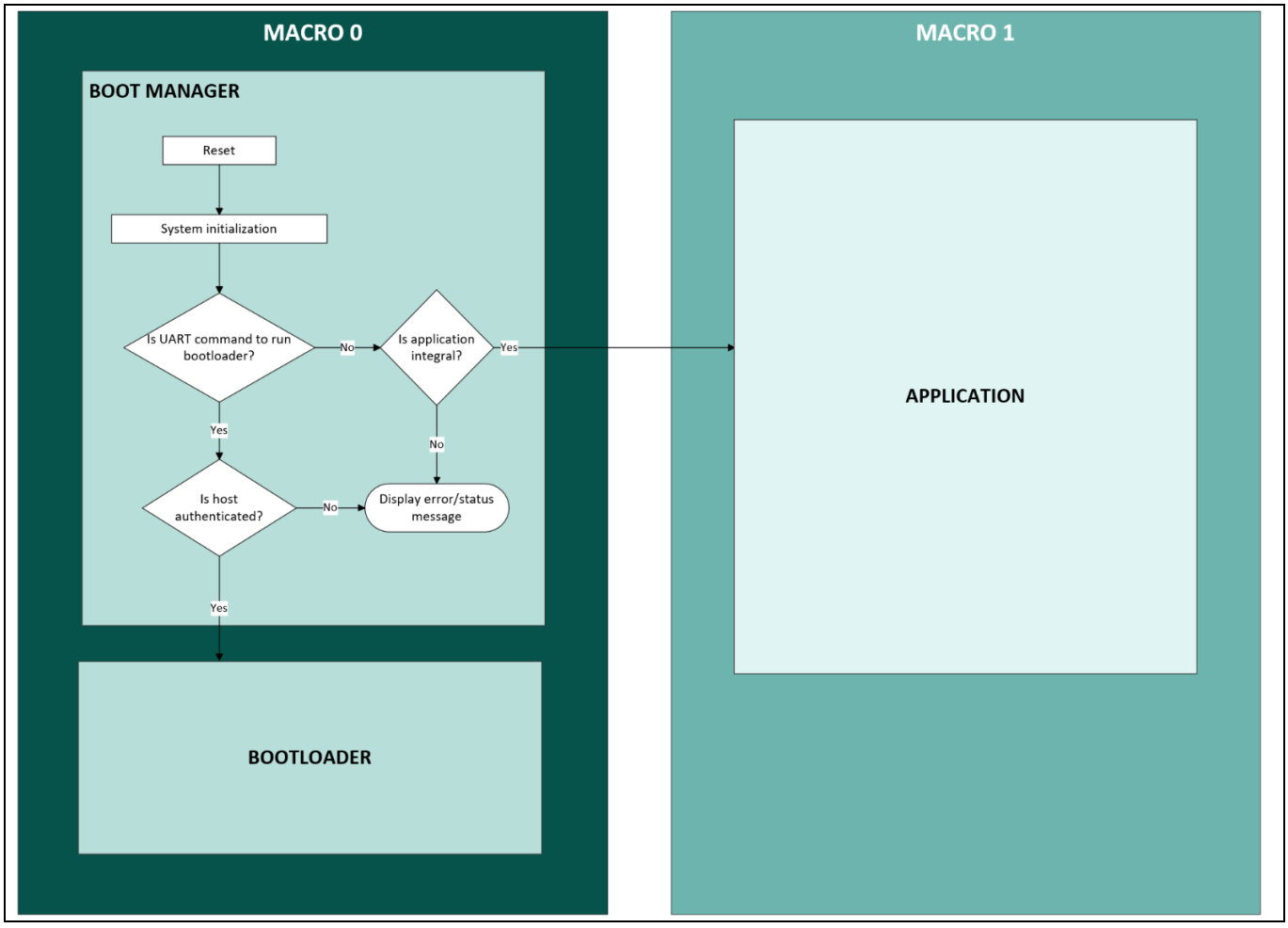

Figure 5 Boot manager: High-level program flow

3.2.1 Host authentication

Host authentication supports restricting the device to accept only authorized software updates. In a simple password-based approach, the host authenticates by sending a password to the target before the bootloader starts. However, this method is not sufficient from a security perspective, if the password is compromised.

Challenge-response authentication is another method to authenticate the host. In this method, the host sends a challenge, and the target responds with an answer known only to both parties. The following is considered for challenge-response authentication in this document and is shown in Figure 6:

- The communication between host and target is through UART
- **Challenge:** The target sends a random number generated by the TRNG to the host. The target computes the CMAC of the random number
- **Response:** The host is expected to compute the CMAC of the random number. The host sends the CMAC to the target to verify
- If the CMACs match, the host is authenticated successfully, otherwise the host is assumed not to be trustworthy
- It is expected that both the target and the host share the same private key for AES-CMAC operation

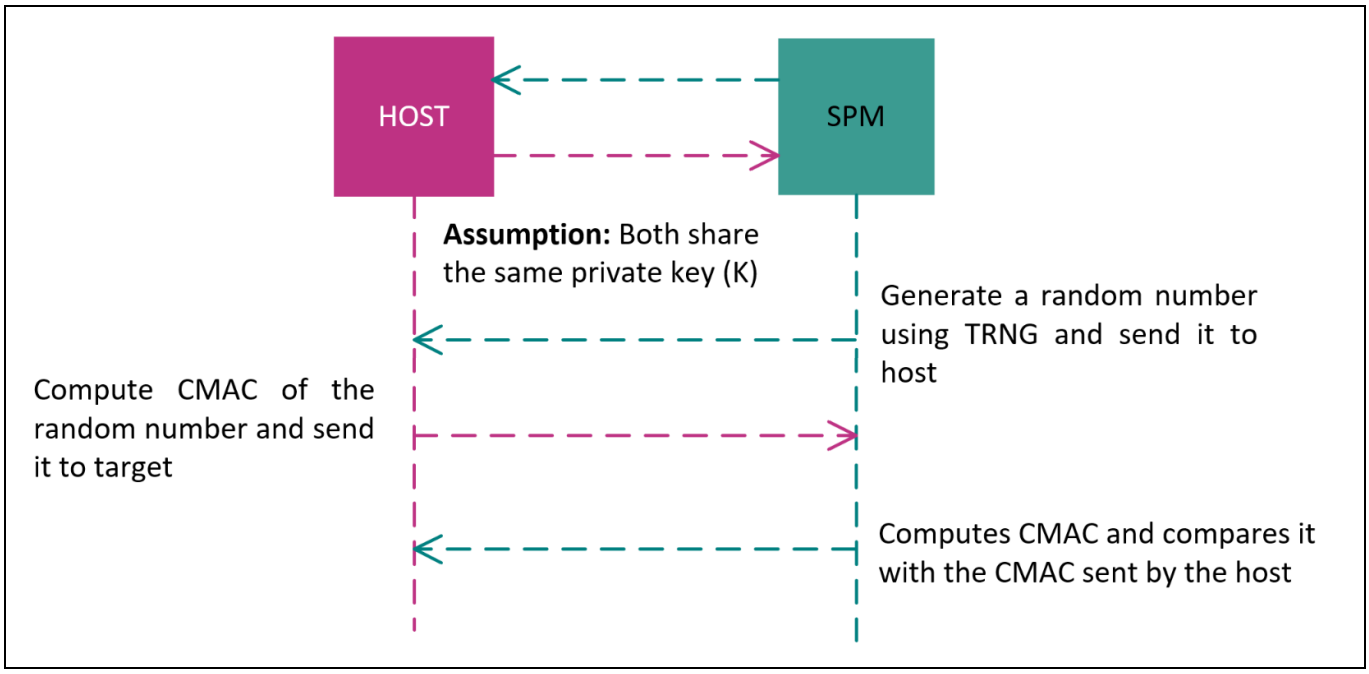

Figure 6 Host authentication flow

As the challenge is always a random number, it would be difficult for an attacker to predict or replay a previous challenge.

3.2.2 Integrity check of application

On every reset, the boot manager verifies the integrity of the application before executing it. This is implemented as follows:

- The boot manager computes the CMAC of the application stored in Macro 1
- It compares the computed CMAC with a reference CMAC stored in a designated flash location (e.g., row 1023 of the code flash)
- If the CMACs match, the application is considered authentic and is launched
- If the CMACs do not match, the application is considered corrupted and the application is not launched.

The boot manager can be designed to handle failure with integrity checks in various ways depending on the application requirements. For example, the target can request a fresh update from the host or log the failure and communicate it with the BMU to take actions.

In this application note, the implementation is simple, if the integrity check fails, the application does not execute.

3.3 Bootloader

Once the host is authenticated, the boot manager launches the bootloader. The bootloader is responsible for receiving the new application from the host and programming it into flash. The process is as follows:

- The bootloader receives the bootloadable (application binary) from the host over UART
- The received data is programmed row-by-row into Macro 1 of the code flash
- After the complete image is programmed, the bootloader computes the CMAC of the application in Macro 1 and compares it with the CMAC sent by the host
- If the CMACs match, the application is considered integral. The bootloader then stores the CMAC in a designated flash location (e.g., row 1023) so the boot manager can use it for the integrity check at the next boot (see Section 3.2.2)
- If the CMACs do not match, the application is considered corrupted. You can design the bootloader to handle this, for example, by restarting the bootload process

The process is shown in Figure 7.

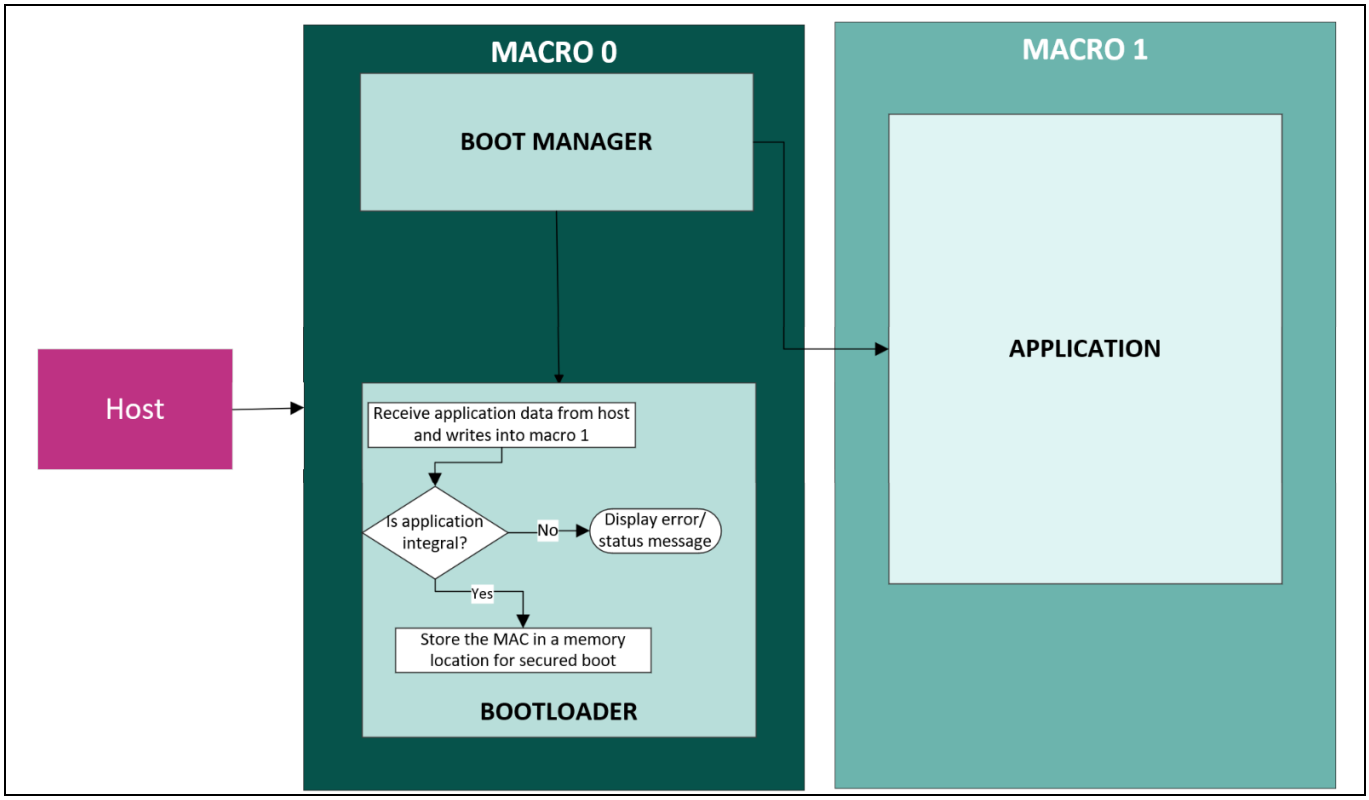

Figure 7 Bootloader: High-level program flow

The integrity check here verifies that the image received was not corrupted during transmission. This is distinct from the secured boot integrity check (Section 3.2.2), which verifies the stored application after every reset.

3.4 Application

The application is the user program that implements the intended device functionality. This is the software component that gets updated during SOTA updates. By design in this document, the application is placed in Macro 1. In general, the application is stored in a memory location based on the application size. The binary of the application can be sent as a bootloadable.

3.4.1 Reset exception

The vector table in an Arm® Cortex®-M0+ stores the addresses of exception and interrupt service routines. The vector table contains pointers (addresses) to exception and interrupt handlers. The table is generally placed at the base memory address.

One of the exceptions is Reset. When reset is asserted, the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table.

Therefore, the secured software in Macro 0 (boot manager + bootloader) must have a reset address in the vector table pointing to the reset handler located within Macro 0's memory space. Similarly, an application in Macro 1 must have its reset vector configured to point to the reset handler within Macro 1's memory space.

3.4.2 Generating binary of application

Because the application resides in Macro 1 instead of the default code flash base address of Macro 0 (0x00000000), the Arm® Cortex®-M0+ vector table must be placed at the Macro 1 base address, which requires a linker configuration change.

The example codes mentioned in this document are developed using IAR Embedded Workbench. There is an IAR Linker Configuration File (ICF) file that defines how the memory and sections of an embedded application are mapped and managed during the linking process. The ICF file gives the linker instructions on where to place program code and other data in the memory layout of a microcontroller.

In this application, symbols can be placed to program the code into the flash or the SRAM. As the application should be stored in Macro 1, the symbol of code flash base address (code_flash_base_address) should be changed to 0x00010000 (starting address of Macro 1). This links the built binary to the address range of Macro 1.

3.5 Additional device security integrations

The boot manager and bootloader (secured software) in Macro 0 form the trusted startup control flow. Therefore, this region should be protected from modifications. As the memory allocation for secured software is Macro 0, enable flash row protection for all rows in Macro 0 (see Section 2.2). This implies that rows 0 to 511 have a flash protection state of ‘full protection’. You can also design to protect the rows that the secured software (boot manager + bootloader) occupies and not the entire macro.

The device still allows read access. To avoid this, transition the device to PROTECTED mode.

Therefore, combined use of flash row protection and PROTECTED mode hardens the security by:

- Preventing unauthorized overwrite of boot manager/bootloader code (secured software)
- Preventing access to resources from debugger
- Preventing read access of the code flash

4 Code example: Secured boot

This example demonstrates secured boot using the Crypto hardware module for verifying the software integrity. For the code example, see AutoPDL code example package.

AES-CMAC is used to verify the integrity of application software stored in flash Macro 1 (starting at flash address 0x00010000) before executing it.

The code example is loaded into Macro 0 while the application is pre-programmed into Macro 1. The application toggles User LED 1. After successful verification, the application is executed.

To demonstrate both a successful and a failed integrity check, the code example allows you to simulate a corrupted application via UART input:

- '1' or `USER_CHOICE_CORRECT_DATA`: Writes valid data to the last row of Macro 1
- '2' or `USER_CHOICE_INCORRECT_DATA`: Writes invalid data to the last row of Macro 1

Before executing the code example, the application must be loaded to the designated code flash location. In this case, the application is loaded to the starting address of flash macro 1 (0x00010000).

The application's binary is included with the code example. The 'AutoFlashUtility' tool can be used to program the application at a specified address. See AutoFlashUtility User Guide (contact Infineon support to obtain the programming tool) and the code example's README to get the commands to program the device using a binary file.

4.1 Implementation

This section describes the main components of the code example. Figure 8 shows the program flow.

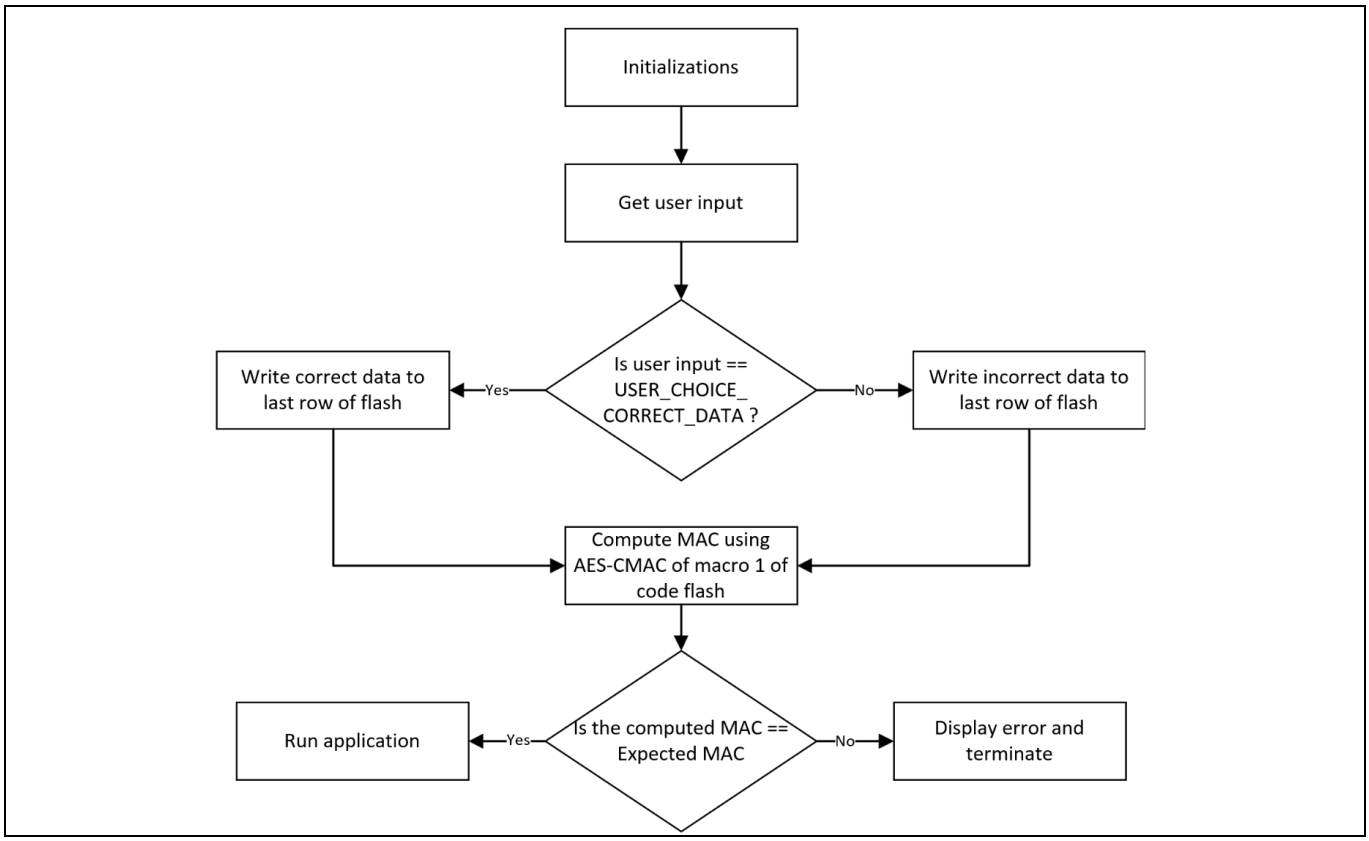

Figure 8 Program flow of secured boot

4.1.1 Initialization

The following are initialized in the main function:

- **GPIO:** To turn on/off the LEDs for error indication
- **UART:** To display status on UART terminal and to receive input from user via UART terminal
- **Flash:** To initialize code flash for write operations
- **Clock:** To enable pump clock for flash operations

4.1.2 Flash operations

Based on the user input (`rxBuffer[0]`) received via the UART terminal, the program writes data to the last row of the code flash. This is to test two cases:

- Application is not corrupted: Correct data is written (`USER_CHOICE_CORRECT_DATA`)
- Application is corrupted: Incorrect data is written (`USER_CHOICE_INCORRECT_DATA`)

Code Listing 1 Flash write operation based on user input

```
uint32 returnValue;
uint32 savedIntrStatus;

/* UART receive buffer for host communication */
static uint8 rxBuffer[16] = {0x00};

/* Buffer containing correct row data for flash write */
static uint8 correctRowData[128]   = {0x00};
/* Buffer containing incorrect row data for flash write */
static uint8 incorrectRowData[128] = {0xFF};


if (USER_CHOICE_CORRECT_DATA == rxBuffer[0])
{
    savedIntrStatus = Cy_SysLib_EnterCriticalSection();
    returnValue = Cy_Flash_WriteRow(CY_FLASH_TYPE_CODE, 1023,
correctRowData);
    Cy_SysLib_ExitCriticalSection(savedIntrStatus);
}
else if (USER_CHOICE_INCORRECT_DATA == rxBuffer[0])
{
   savedIntrStatus = Cy_SysLib_EnterCriticalSection();
   returnValue = Cy_Flash_WriteRow(CY_FLASH_TYPE_CODE, 1023,
incorrectRowData);
    Cy_SysLib_ExitCriticalSection(savedIntrStatus);
}
```

4.1.3 CMAC computation

After the flash write operations are executed successfully, the CMAC of the entire data in Macro 1 of code flash is computed.

Code Listing 2 Computing CMAC using AES-CMAC of code flash Macro 1

```
cy_en_crypto_return_t ret = CY_CRYPTO_RET_ERROR_NO_DATA;
/* AES-CMAC key */
const uint8 AES_KEY[16] = {0x2b, 0x7e,...};
cy_en_crypto_aes_key_length_t keyLength= CY_CRYPTO_KEY_AES_128;
uint8* srcMsg = (uint8*) (0x00010000);
uint32 srcMsgSize = (CY_FLASH_SIZE / 2U);
uint8 outputCmac[16] = {0};
uint32 outputCmacSize = 16;

/* Compute CMAC of the message */
ret = Cy_Crypto_AesCmac(CY_CRYPTO, AES_KEY, keyLength, srcMsg, srcMsgSize,
outputCmac, outputCmacSize);
```

4.1.4 Integrity verification

Once the CMAC is computed as shown in Section 4.1.3, it is compared to the expected CMAC (`EXPECTED_CMAC`). If the CMACs match, the application is not corrupted and the application is executed. If the CMACs do not match, the application is corrupted, and the application is not executed.

After the integrity check passes, the program launches the application by reading the reset vector from the application's vector table at the base address of Macro 1 (0x00010000). The vector table stores the reset handler address at offset 0x04.

The boot manager loads the pointer and jumps to the reset handler to transfer execution to the application. Using a function pointer, the program can be directed to execute the reset handler of the application. An alias for the function pointer is created using typedef as follows:

```
typedef void (*funcPtr)(void);
```

Code Listing 3 Integrity verification by comparing MACs

```
/* Expected CMAC value */
const uint8 EXPECTED_CMAC[16] = {0xa4, 0x37,...};

uint32 resetVectorAddress = 0x00010004;

/* Function pointer that points to the address stored in reset vector */
    funcPtr applicationPtr = (funcPtr) (*((uint32*) resetVectorAddress));
```

```
if (0 == memcmp(outputCmac, EXPECTED_CMAC, outputCmacSize))
{
    /* Run the application */
    (*applicationPtr)();
}
else
{
    /* Display error and terminate */
}
```

4.2 Results

The following sections show the output displayed on the UART terminal on executing the code example from AutoPDL.

Note: *Tera Term is used to monitor UART communication.*

4.2.1 Test 1: Integrity verification passes

- Enter '1' via the UART terminal to write the correct data
- UART terminal displays CMAC values and success message
- Application is executed on successful verification (toggling of User LED 1)

Figure 9 shows the terminal output for test 1.

```
************************************************************
                    SECURED BOOT
************************************************************

Press 1 to write correct data
Press 2 to write incorrect data
1
Correct row data written to last row of code flash.

Expected CMAC: a437bbb96b3c1f232e8bce405472acb9
Computed CMAC: a437bbb96b3c1f232e8bce405472acb9

[OK] Application integrity verified. Starting application...
```

Figure 9 Terminal output: Integrity verification passes

4.2.2 Test 2: Integrity verification fails

- Enter ‘2’ via the UART terminal to write the incorrect data
- UART terminal displays CMAC values and failure message
- User LED 2 turned on to indicate integrity verification failure

Figure 10 shows the terminal output for test 2.

```
*********************************************************
                  SECURED BOOT
*********************************************************

Press 1 to write correct data
Press 2 to write incorrect data

Incorrect row data written to last row of code flash.

Expected CMAC: a437bbb96b3c1f232e8bce405472acb9
Computed CMAC: 1b44b4af66a922abf7bf1062e5f3c864

[ERROR] Application integrity verification failed. Application will not be started.
```

Figure 10 Terminal output: Integrity verification fails

5 Code example: Secured bootloader

This code example demonstrates secured bootloader using the Crypto hardware block of the PSOC™ 4 HVPA-SPM. The code example is available in AutoPDL. The bootloader implements two main features:

- **Challenge-response authentication:** Uses TRNG to generate a random challenge for the host to compute AES-CMAC before permitting bootloading operations
- **Data integrity verification**: Uses AES-CMAC to verify the integrity of the bootloadable/application image and detect any corruption during bootloading

5.1 Prerequisites

A terminal emulation software that supports data in hexadecimal format is required (e.g., CoolTerm) to execute this code example. See the code example's README file for more information.

5.2 Implementation

Relevant sections of the code example are described in this section. **Error! Reference source not found.** shows the program flow of secured bootloader implemented in the code example.

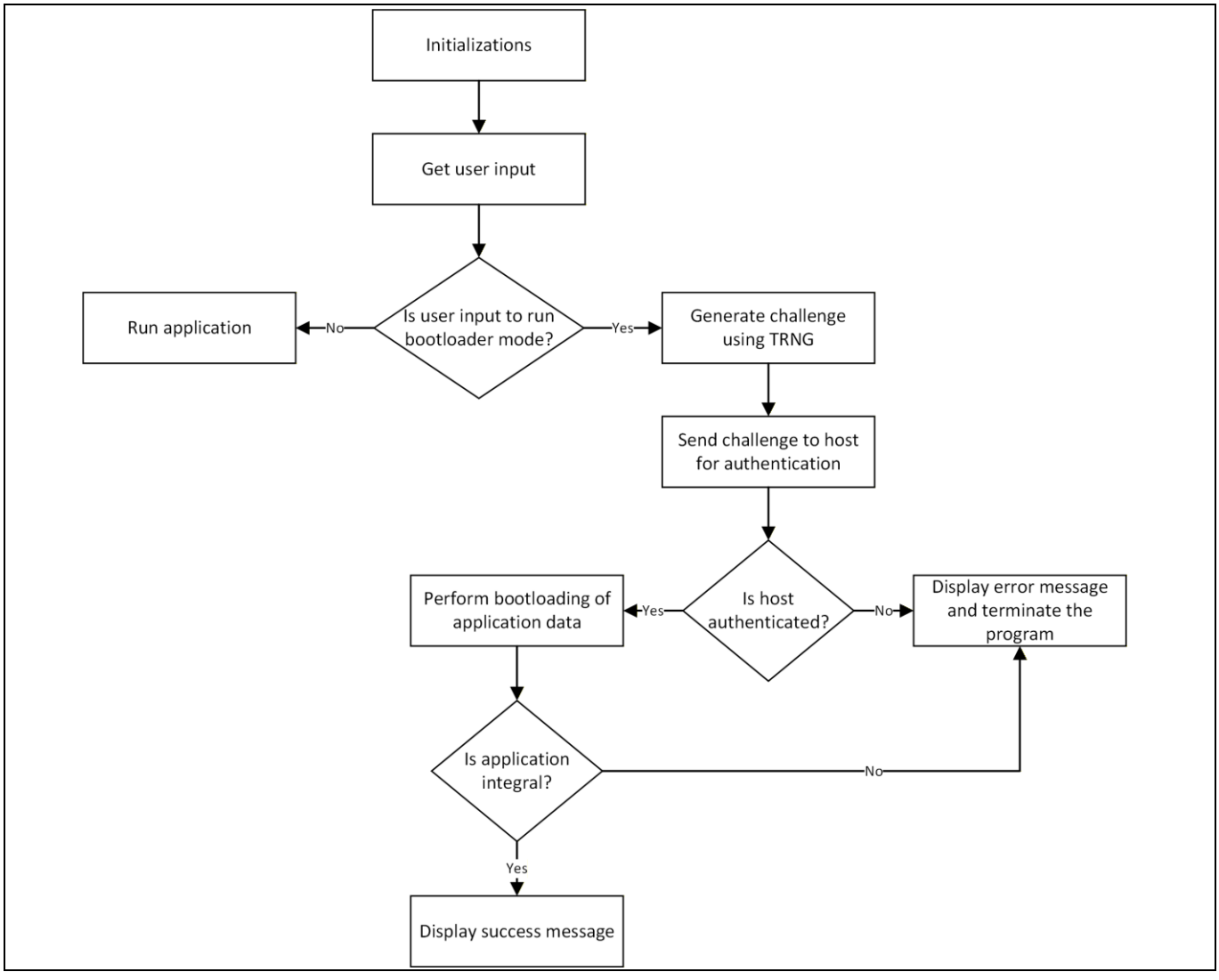

Figure 11 Program flow of secured bootloader

5.2.1 Initialization

The following are initialized in the code example:

- **GPIO:** To turn on/off the LEDs for error indication
- **UART:** To display status and to receive input from user via UART terminal

Based on the user input, either the application or the bootloader can be executed.

5.2.2 Execute application

In this code example, the application is a simple demonstration function that toggles User LED 1. To execute an actual application stored in macro 1, the reset vector method described in section 3.4 can be used (as demonstrated in the secured boot code example in Section 4).

5.2.3 Challenge-response authentication

PSOC™ 4 HVPA-SPM is the target device, and the computer is the host. The challenge in this code example is to send a random number to the host by utilizing the TRNG. This random number is sent via UART to the host.

The authentication flow is as follows:

- The program generates a random number using the TRNG and sends it to host via UART
- The program computes the CMAC of the random number using the provisioned key and displays the expected CMAC on the terminal
- The user (host) sends a CMAC back to the device via the terminal, either the correct CMAC (copied from the terminal display) or an incorrect one to simulate a failed authentication
- If the received CMAC matches the computed CMAC, the host is authenticated and bootloading proceeds
- If the CMACs do not match, User LED 2 turns on and the terminal displays an authentication error

Note: *In this demo, the expected CMAC is displayed on the terminal for convenience. In a production system, the host would independently compute the CMAC using its own copy of the shared secret key. See Code Listing 2 for AES-CMAC operations.*

5.2.4 Integrity of the bootloadable data

After successful authentication, the bootloader receives the bootloadable data from the host via UART. The host also sends the expected CMAC of the bootloadable.

The program computes the AES-CMAC of the received data/bootloadable and compares it with the CMAC from the host:

- If the CMACs match, the data is not manipulated and the bootloading is considered successful
- If the CMACs do not match, User LED 2 turns on, and the terminal displays an integrity check error

Note: *In this code example, the bootloadable data is 5 bytes for simplicity. In a real application, the bootloadable data would typically be the entire application image.*

5.3 Results

The following sections shows the output displayed on the UART terminal on executing the code example from AutoPDL.

5.3.1 Test 1: Application mode

- Enter '1' via UART terminal to execute the application mode.
- Terminal displays message and User LED 1 toggles to indicate the application is executing.

Figure 12 shows the terminal output for test 1.

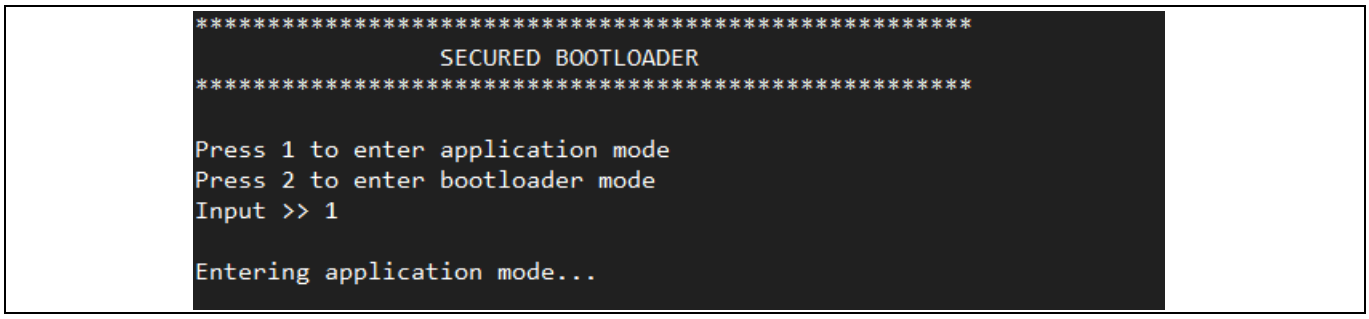

Figure 12 Terminal output of test 1

5.3.2 Test 2: Bootloader mode (successful authentication and integrity check)

- Enter '2' via UART terminal to execute the bootloader mode
- Challenge-response authentication is performed
- Upon successful authentication, the user sends the bootloadable
- The integrity of the bootloadable is checked

Figure 13 shows the terminal output for test 2.

```
*****************************************************
                  SECURED BOOTLOADER
*****************************************************

Press 1 to enter application mode
Press 2 to enter bootloader mode
Input >> 2

Entering bootloader mode...

----- Generate challenge using TRNG -----
Sending 10 byte challenge to host in hex...
46 93 b3 49 c4 6c 81 7d 79 75

---------- Host authentication ----------

CMAC of challenge is:
91 66 11 f5 29 33 ae 1b 14 5c 90 47 78 4d 40 c5

Enter CMAC to complete host authentication:
Input (hex) >> 91 66 11 f5 29 33 ae 1b 14 5c 90 47 78 4d 40 c5

[OK] Host is successfully authenticated!

----- Bootloading and integrity check -----

Enter 5 byte bootloadable/application data
Input (hex) >> 41 42 43 44 45

CMAC of application data is:
f0 7f 5c 21 d8 5e 56 61 c7 e0 b6 be c9 86 f3 9a

Enter CMAC to verify integrity of the bootloadable data:
Input (hex) >> f0 7f 5c 21 d8 5e 56 61 c7 e0 b6 be c9 86 f3 9a

[OK] Application data integrity verified successfully.
Reset device to enter application mode.
```

Figure 13 Terminal output of test 2

5.3.3 Test 3: Bootloader mode (failed authentication)

- Enter '2' via UART terminal to execute the bootloader mode
- Challenge-response authentication is performed and user sends incorrect CMAC
- UART terminal displays authentication failure message
- User LED 2 turns on and device exits bootloader mode

Figure 14 shows the terminal output for test 3.

```
*****************************************************
                  SECURED BOOTLOADER
*****************************************************

Press 1 to enter application mode
Press 2 to enter bootloader mode
Input >> 2

Entering bootloader mode...

----- Generate challenge using TRNG -----
Sending 10 byte challenge to host in hex...
45 27 38 9b 6e f7 29 d8 24 41

---------- Host authentication ----------

CMAC of challenge is:
10 cb e7 8d dd 59 63 86 8e 1b 61 62 bd 2f 84 e9

Enter CMAC to complete host authentication:
Input (hex) >> f0 7f 5c 21 d8 5e 56 61 c7 e0 b6 be c9 86 f3 9a

[ERROR] Host is not authenticated. Exiting bootloader mode...
```

Figure 14 Terminal output of test 3

5.3.4 Test 4 – Bootloader mode (failed integrity verification)

- Enter '2' via UART terminal to execute the bootloader mode
- Challenge-response authentication is performed and user sends correct CMAC
- Upon successful authentication, the user sends the bootloadable
- User sends incorrect CMAC for the integrity check of the bootloadable
- User LED 2 turns on and device exits bootloader mode

Figure 15 shows the terminal output for test 4.

```
**************************************************
                SECURED BOOTLOADER
**************************************************

Press 1 to enter application mode
Press 2 to enter bootloader mode
Input >> 2

Entering bootloader mode...

----- Generate challenge using TRNG -----
Sending 10 byte challenge to host in hex...
f3 25 13 34 45 c3 a8 f9 5e 08

---------- Host authentication ----------

CMAC of challenge is:
9b 54 9c 9e 31 d4 20 c0 a6 85 30 19 ae 8c 5b 27

Enter CMAC to complete host authentication:
Input (hex) >> 9b 54 9c 9e 31 d4 20 c0 a6 85 30 19 ae 8c 5b 27

[OK] Host is successfully authenticated!

----- Bootloading and integrity check -----

Enter 5 byte bootloadable/application data
Input (hex) >> 61 62 63 64 65

CMAC of application data is:
99 53 81 d9 23 f7 9b ce b1 9b 4e f6 10 de 43 15

Enter CMAC to verify integrity of the bootloadable data:
Input (hex) >> 9b 54 9c 9e 31 d4 20 c0 a6 85 30 19 ae 8c 5b 27

[ERROR] Error in verifying integrity of application data. Exiting bootloading mode...
```

Figure 15 Terminal output of test 4

6 Related resources

- Product family webpage: PSOC™ 4 HVPA-SPM 1.0
- Infineon Developer Community: PSOC™ Automotive forum

References

[1] Infineon Technologies AG: *PSOC™ 4 HV Family AutoPDL and middleware*; Available online

[2] Infineon Technologies AG: *PSOC™ 4 HVPA SPM 1.0 MCU architecture reference manual*; Available online

[3] Infineon Technologies AG: *Auto Flash Utility*; Available online (contact Infineon support to obtain the programming tool)

[4] Infineon Technologies AG: *PSOC™ 4 HVPA-SPM: How to set flash row protection*; Available online

Abbreviations

Abbreviation	Expansion
AES	Advanced Encryption Standard
BMU	Battery management unit
BMS	Battery management system
CAN	Controller Area Network
CRC	Cyclic redundancy check
DAP	Debug Access Port
ECU	Electronic control unit
GPIO	General-purpose input/output
HVPA	High Voltage Precision Analog
HVPA-SPM	High Voltage Precision Analog Smart Battery Pack Monitor
ICF	IAR linker configuration file
JTAG	Joint Test Action Group
PRNG	Pseudo random number generator
SHA	Secure Hash Algorithm
SFlash	Supervisory flash
SOTA	Software over-the-air
SWD	Serial Wire Debug
TRNG	True random number generator

Revision history

Document revision	Date	Description of changes
V 1.0	2026-08-06	Initial release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.
PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Edition: 2026-08-06

Published by
Infineon Technologies AG
Am Campeon 1-15
85579 Neubiberg, Germany

Do you have a question about this document?
Email: erratum@infineon.com

Document reference:
AN0073

Important Notice
Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.
For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability.
Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.
If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.
The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.
Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury. However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer.
Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG).
If the Product includes security features:
Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.
If this document includes or references software:
The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved.
Therefore, you may use the software only as provided in the software license agreement accompanying the software.
If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.
For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit https://www.infineon.com.