

# Getting started with PSoC™ 6 security

## About this document

This guide helps you to understand the basic security features implemented on PSoC™ 6 (61/62/63) devices, and gets you started with security on PSoC™ 6 MCUs out of the box. It contains guidance on how to enable secured boot, generate RSA and ECC keys, sign and program the application, and implement a Chain of Trust (CoT) along with debug port configurations.

### Scope and purpose

This document demonstrates how to enable secured boot on a simple Hello World application on PSoC™ 6 MCU, and provides a basic description of secured boot flow, key generation, and image signing and validation.

### Intended audience

This document is intended for customers who want to start with enabling secured boot and Chain of Trust implementation on PSoC™ 6 devices.

## Table of contents

---

## Table of contents

	<b>About this document</b> .....	1
	<b>Table of contents</b> .....	2
<b>1</b>	<b>Introduction</b> .....	3
1.1	PSoC™ 6 boot sequence .....	3
1.2	Basic definitions .....	4
<b>2</b>	<b>Security features</b> .....	8
<b>3</b>	<b>Secured boot</b> .....	9
3.1	Requirements .....	9
3.2	Enabling secured boot .....	11
3.2.1	Prerequisites .....	12
3.2.2	Create MTB application (Hello World App) .....	12
3.2.3	Adding TOC2 .....	14
3.2.4	Add application header and signature .....	17
3.2.5	Generating the RSA key pair .....	20
3.2.6	Signing the application .....	28
3.2.7	Programming the image .....	30
<b>4</b>	<b>Device lifecycle stages (LCS)</b> .....	32
<b>5</b>	<b>Debug port access configuration (DAP)</b> .....	34
<b>6</b>	<b>Device CoT implementation</b> .....	36
6.1	Introduction to MCUboot .....	36
6.1.1	Application structure .....	36
6.2	Enabling secured boot (with MCUboot basic bootloader CE) .....	37
	<b>Revision history</b> .....	41
	<b>Disclaimer</b> .....	42

## 1 Introduction

### 1 Introduction

PSoC™ 6 MCU is an ultra-low-power, programmable embedded system-on-chip with a dual-core architecture, tailored for smart homes, IoT gateways, etc., and available in multiple variants:

- **PSoC™ 61:** Single-core microcontroller: Arm® Cortex® -M4 (CM4)
- **PSoC™ 62/63:** Dual-core microcontroller: Arm® Cortex® -M4 (CM4) and Cortex® -M0+ (CM0+)

There are two generations of PSoC™ 6 devices with minor differences in their operation and configuration: see the following table. This document will refer to these parts as 1st and 2nd Generation PSoC™ 6 devices.

**Table 1** PSoC™ 6 device generations

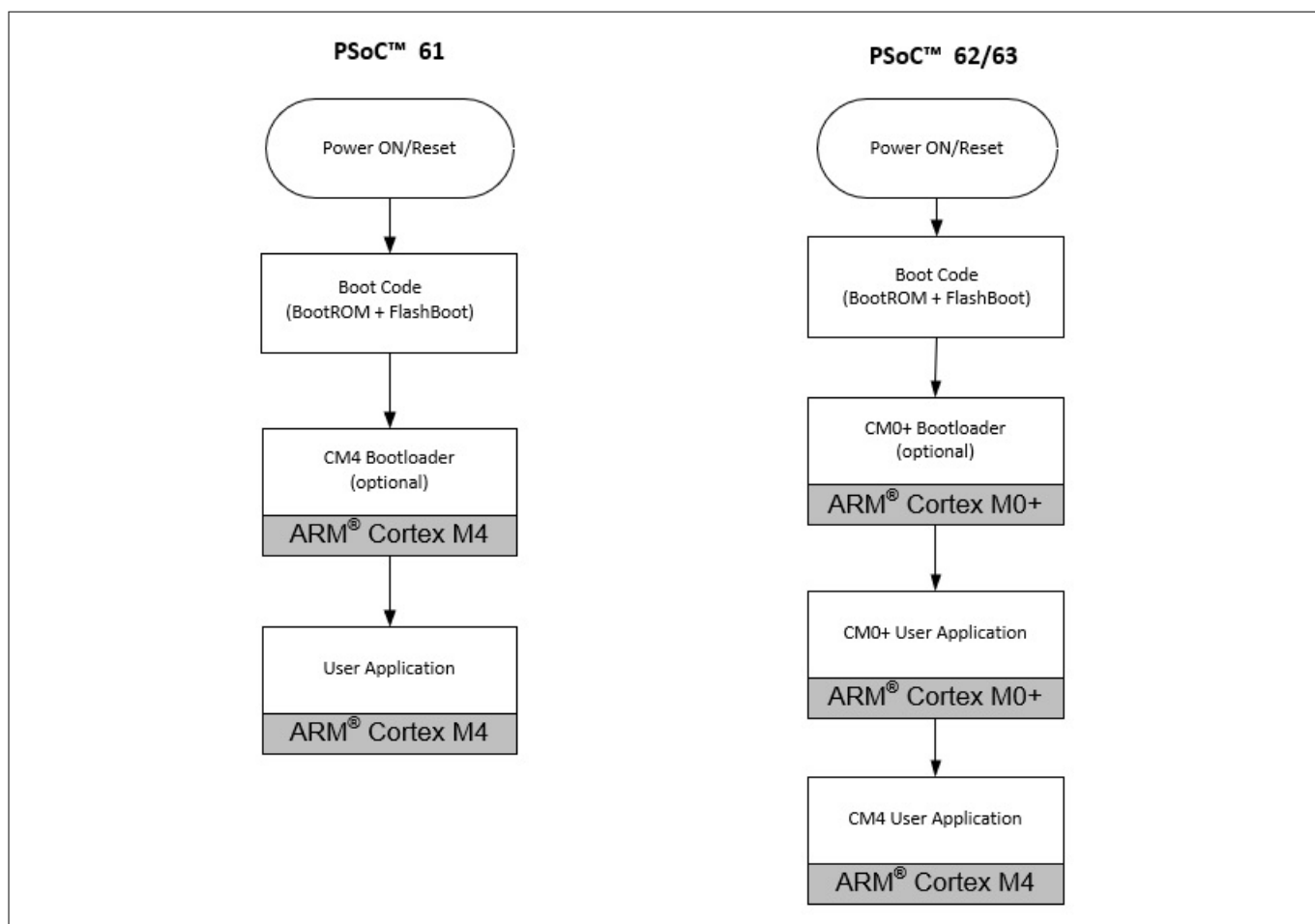
1st Generation PSoC™ devices	CY8C61x6, CY8C62x6, CY8C63x6, CY8C61x7, CY8C62x7, CY8C63x7
2nd Generation PSoC™ devices	CY8C61x4, CY8C62x4, CY8C61x5, CY8C62x5, CY8C61x8, CY8C62x8, CY8C61xA, CY8C62xA

This application note helps you get started with security features available in PSoC™ 6 by enabling secured boot and implementing a secured Chain of Trust.

#### 1.1 PSoC™ 6 boot sequence

When PSoC™ 6 device is powered ON, the Cortex® -M0+ core comes out of reset and executes the boot code that performs basic housekeeping tasks and hands over control to the user application on CM0+. It is up to the user code in Cortex® -M0+ to enable the Cortex® -M4 after the boot sequence if the default Cortex® -M0+ startup code is not used. In case of PSoC™ 61, Cortex® -M0 still handles the boot sequence up to the user application and the entire user application executes only on Cortex® -M4. With PSoC™ 62/63 devices, the user application includes both a user Cortex® -M0 project and a Cortex® -M4 project. The following diagram demonstrates the basic boot flow on a PSoC™ 6 device.

## 1 Introduction



**Figure 1** PSoC™ 6 boot flow

## 1.2 Basic definitions

**Table 2** Abbreviations and definitions

Term	Description
Chain of Trust (CoT)	Chain of Trust is established by validating the blocks of software starting from the root of trust located in the ROM. The root of trust begins with the Infineon code residing in the ROM that cannot be altered.
Code signing	Process of calculating a hash of the code binary and encrypting the hash with a private key and appending this to the code binary.
Dead access restrictions (DAR)	Determines what resources are accessible via the debug port when in DEAD mode. DEAD mode occurs if an error is found during the boot sequence. The DAR are stored in eFuse.

(table continues...)

## 1 Introduction

**Table 2** (continued) Abbreviations and definitions

Term	Description
Debug access port (DAP)	Interface between an external debugger/programmer and PSoC™ 6 MCU for programming and debugging. This allows connection to one of three access ports (AP), CM0_AP, CM4_AP, and System_AP (Sys_AP). System_AP can access only the SRAM, flash, and MMIOs, not the CPU.
Digest	The output of a cryptographic hash function is often called a "message digest" or "digest". This digest is then encrypted with a private key to form a digital signature.
Digital signature	Encrypting of the digest (hash of a data set). For example, the encrypted hash of the user application.
Elliptic-curve cryptography (ECC)	ECC is an asymmetric encryption system that uses two keys. One key is private and should not be shared, and the other is public and can be read without loss of security. ECC is a more modern method than RSA and requires a smaller key than RSA for the same level of security.
eFuse	One-time programmable (OTP) memory that by default is 0 and can be changed only from 0 to 1. eFuse bits may be programmed individually and cannot be erased.
Flash boot	Part of the boot system that performs two basic tasks: 1. Sets up the debug port based on the LCS and TOC. 2. Validates the user application before executing it. Flash boot executes after ROM boot.
Flash (User)	Flash memory that is used to store your application code. It is non-volatile and can be reprogrammed.
Lifecycle stage (LCS)	The LCS is the security mode in which the device is operating. To the user, it has only four stages of interest: NORMAL, SECURE, SECURE_WITH_DEBUG, and RMA.
Normal Access Restrictions (NAR)	NAR determines which debug ports are disabled/enabled. NAR controls the debug ports and what memory is accessible via SYS_AP if it is left open and SYS_AP MPU is enabled. NAR is stored in SFlash.
Public-key cryptography (PKC)	Also known as "asymmetrical cryptography". Public-key cryptography is an encryption technique that uses a paired public and private key (or asymmetric key) algorithm for secure data. It is used to secure a message or block of data. The private key is used to encrypt data and must be kept secured, and the public key is used to decrypt but can be disseminated widely.

(table continues...)

## 1 Introduction

**Table 2** (continued) Abbreviations and definitions

Term	Description
Public key	When using asymmetrical cryptography such as RSA or ECC, a public key is used to validate the firmware that was signed by the private key. It can be shared, but it should be authenticated or secured so it cannot be modified.
Private key	When using asymmetrical cryptography such as RSA or ECC, the private key is used to sign (encrypt the hash) of the firmware after it is built but prior to being loaded into the device. It must be kept in a secure location, so it cannot be viewed or stolen.
RMA	Return Merchandise Authorization
ROM Boot	After a reset, CM0+ starts executing code that has been programmed into the ROM. This code cannot be altered.
Secure Access Restrictions (SAR)	SAR determines which debug ports are disabled/enabled and determines what memory is accessible via SYS_AP, if it is left open and SYS_AP MPU is enabled. SAR is stored in eFuse.
Secure hash	Calculated hash (SHA-256) of the system trim values, Flash boot, TOCs, and user public key. This hash is generated during the transition to SECURE LCS and stored in eFuse. This insures that the OEM's public key has not been corrupted maliciously or accidentally.
Supervisory Flash (SFlash)	Supervisor flash memory. This memory partition in flash contains several areas that include system trim values, flash boot executable code, public key storage, etc. After the device transitions into SECURE mode, it can no longer be modified.
SHA-256	SHA-256 is a common cryptographic hash algorithm used to create a signature for a block of data or code. This hash algorithm produces a 256-bit unique signature of the data no matter the size of the data block.

(table continues...)

---

## 1 Introduction

**Table 2** (continued) Abbreviations and definitions

Term	Description
Table of Contents2 (TOC2)	An area in SFlash of the PSoC™ 6 MCU that is used to store parameters and pointers to objects used for secure boot. Locations of two application pointers (Application1 and Application2) are stored here, but the second one is optional. The first pointer (Application1) must point to the first executable user code, which may be the bootloader or just the application. The table of contents also contains some boot parameters that are settable by the system designer. A duplicate of TOC2 is written in the adjacent page of flash for redundancy. This duplicate is called “RTOC2”. If TOC2 is found to be invalid for any reason, RTOC2 is evaluated. Both these structures are protected with a CRC, and are part of the secure hash.

## 2 Security features

## 2 Security features

PSoC™ 6 MCUs have several security features that are used to build a custom secured system as shown below. It is also important to understand that in all but the simplest secure system, all these security features work together:

**eFuse block:** eFuse is a memory consists of a set of eFuse bits. There are a total of 1024 eFuse bits or first-generation parts; 512 bits of them are available for custom purposes and the rest are available to the user to store system hash values, security attributes and LCS states. For the second-generation parts, 376 eFuse bits are available.

**Device lifecycle stage (LCS):** The LCS dictates how the device boots up and which security attributes to enable, such as SAR, NAR, and DAR.

**Chain of Trust:** This dictates that the validation of the user application code is linked all the way back to the device ROM. The code verification is accomplished by using the user's public key stored in SFlash. A hash is calculated for the SFlash area and stored in eFuse. Any changes in the eFuse, public key, or user code will be detected and boot will fail.

**Protection units:** These protect or isolate the code and data from different bus masters. They are also used to secure the hardware such as flash and eFuse programming. Users can also protect other hardware such as GPIOs or communications ports.

**Debug port:** The debug port consists of three ports (CM0\_AP, CM4\_AP, and SYS\_AP) and are initially used for programming and debugging of the user application, but should be disabled after the device is in SECURE lifecycle stage (e.g., Production stage)

In a PSoC™ 62/63 dual-core device with a CM0+ and a CM4 CPU, CM0+ is usually considered as the secured processor and is the one that performs all system calls and runs the secured boot sequence. The “main()” function in the user project enables CM4 only after CM0+ has run the boot process. CM4 is usually considered as the application processor because it can run faster and is more powerful than CM0+. CM0+ is the logical choice to configure your secure operations because it executes before CM4 and allows for a secured logical isolation between the two cores,

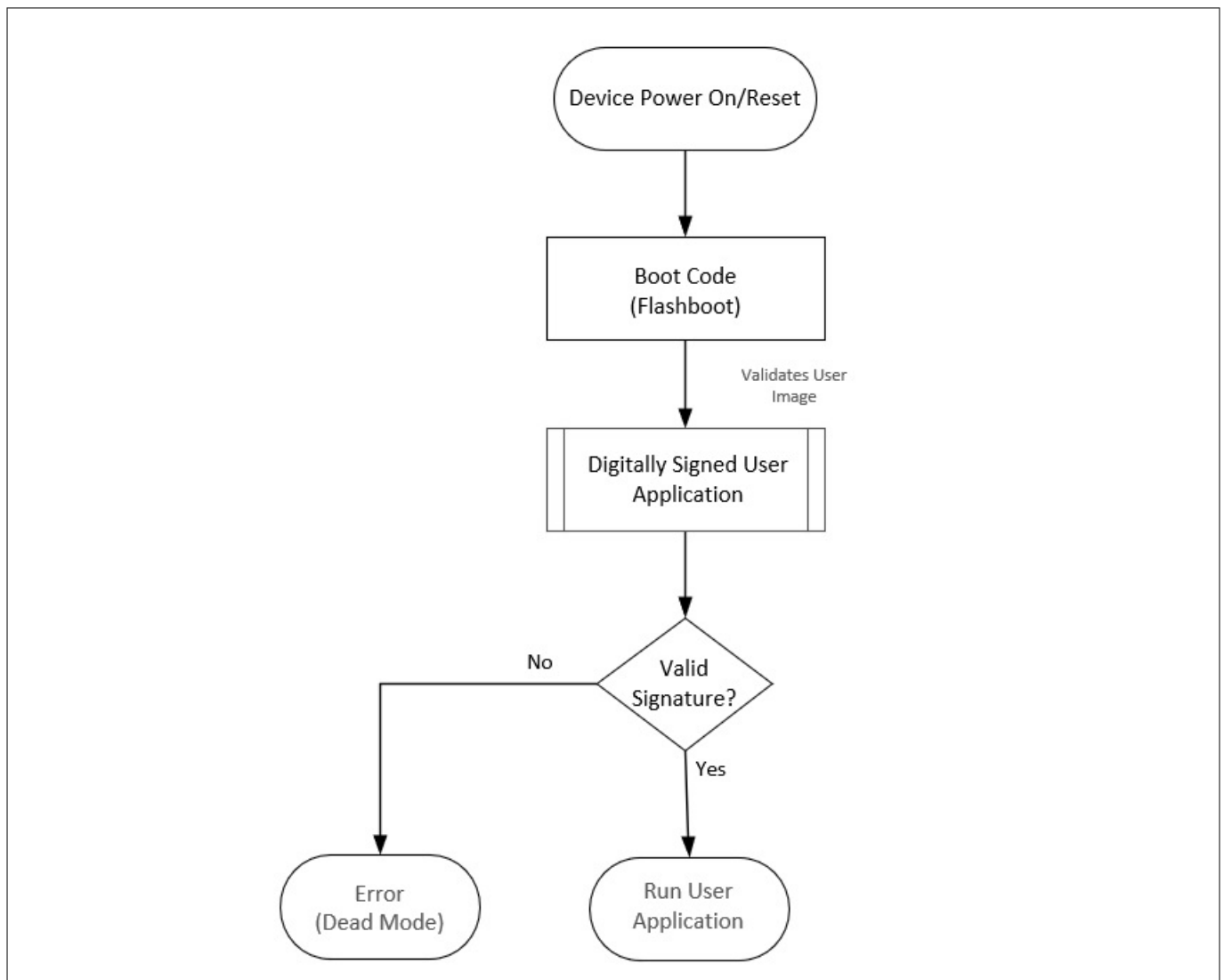
This document covers only how to enable the secured boot sequence and implement the Chain of Trust for implementing a simple secured system. To implement a fully secured system with all the security features enabled, refer to [AN21111 - PSoC™ 6 MCU designing a custom secured system](#).



### 3 Secured boot

## 3 Secured boot

Secured boot is an important security feature to prevent any malicious and unsigned code executing on the device. Infineon provides an immutable boot code programmed in factory that acts as a Root of Trust (RoT), runs on the CM0+ core that is regarded as a secure core; the user application can run either on the CM0+ or CM4 core. In secured boot functionality, the authenticity of the user application is always validated by CM0+ flash boot before handing over control to the user application. The boot code verifies the signature of the user application that should run on the device before launching it. The device fails to boot in case of failed authentication. Refer to [AN21111 - PSoC™ 6 MCU designing a custom secured system](#) for more information on the secured boot process.



**Figure 2 Basic secured boot flow**

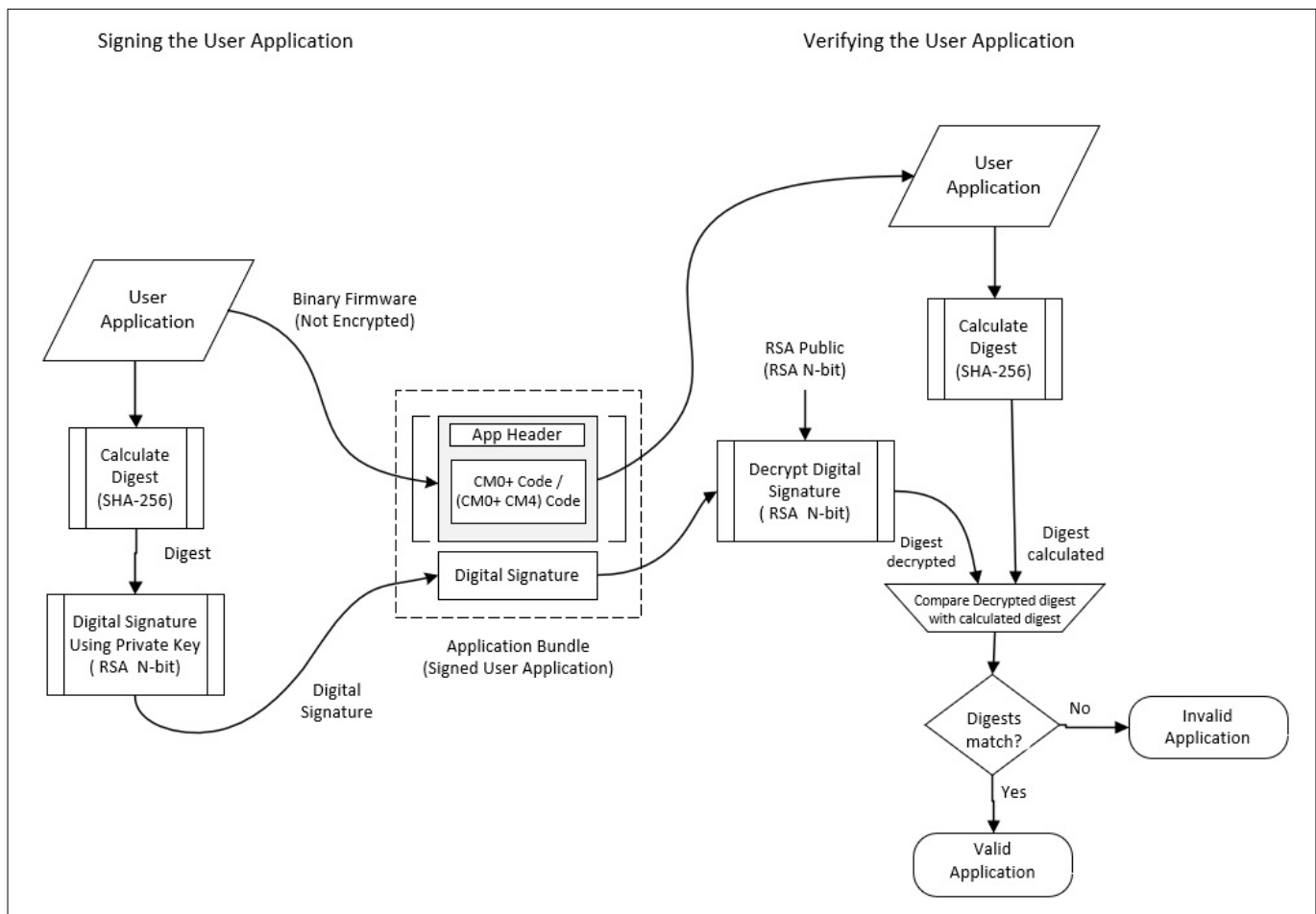
### 3.1 Requirements

To enable the secured boot functionality on a PSoC™ 6 device, ensure the following basic requirements:

- 1. TOC2** - The TOC2 (Table Of Contents 2) is a structure that has all the details required by flash boot to locate and validate the user application. This structure is flashed into the device SFlash region. For example, TOC2 contains the address of the user application to be validated, address of the OEM public key required for validating the application, the application format, etc. For more information on TOC2, refer to the Project configuration section of [AN21111 - PSoC™ 6 MCU designing a custom secured system](#)

## 3 Secured boot

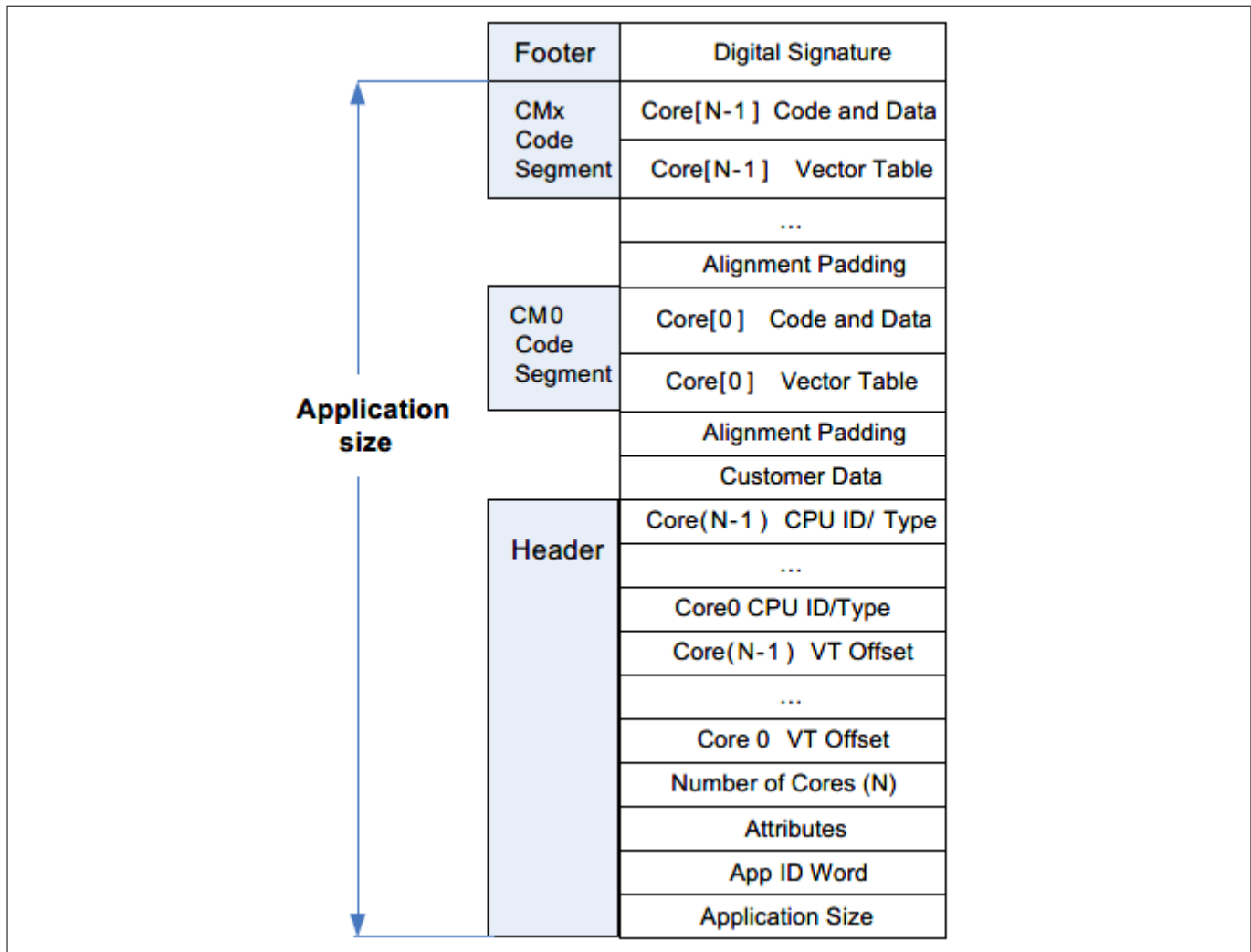
2. **RSA key pair** - The RSA key pair is required to sign the user application with the generated private key and store the public key into the device SFlash. During device boot, flash boot uses the public key stored in SFlash to validate the application signature and proceeds to boot the application if the signature is verified
3. **Application signing** - To verify the user application, a digital signature is created and appended to the end of the project binary. The code itself is not encrypted but the digital signature is the encrypted digest. The digital signature is generated by encrypting the digest with the OEM RSA private key. The digest is generated by running the user project binary through a SHA-256 hash function. This method guarantees that a third-party without access to the OEM's private key cannot properly sign an untrusted executable image; this ensures that only trusted OEM images can execute. The signed image that is executed after flash boot is validated with the RSA public key



**Figure 3 PSoC™ 6 application sign & verify**

During validation, flash boot requires some information about the application that is to be validated, such as the application size, vector table offset, and the core ID. All this information is put in a header and appended to the application in the CYPRESS™ standard application format (CYSAF) as shown in the figure below. This allows flash boot to perform the validation during the boot process before the application is executed. The application format encapsulates the application binary, application metadata, and an encrypted digital signature.

### 3 Secured boot



**Figure 4 CYPRESS application format**

Once the public-private key pair is generated and the image is signed with the private key, the signed image (secured image) along with the TOC2 and public key are programmed into the device in the respective flash regions using a programming tool. After the signed image is flashed and the device is reset, flash boot reads the TOC2 contents for the signed image address, public key address, etc. and starts to validate the image. The validation process by flash boot consists of the following steps:

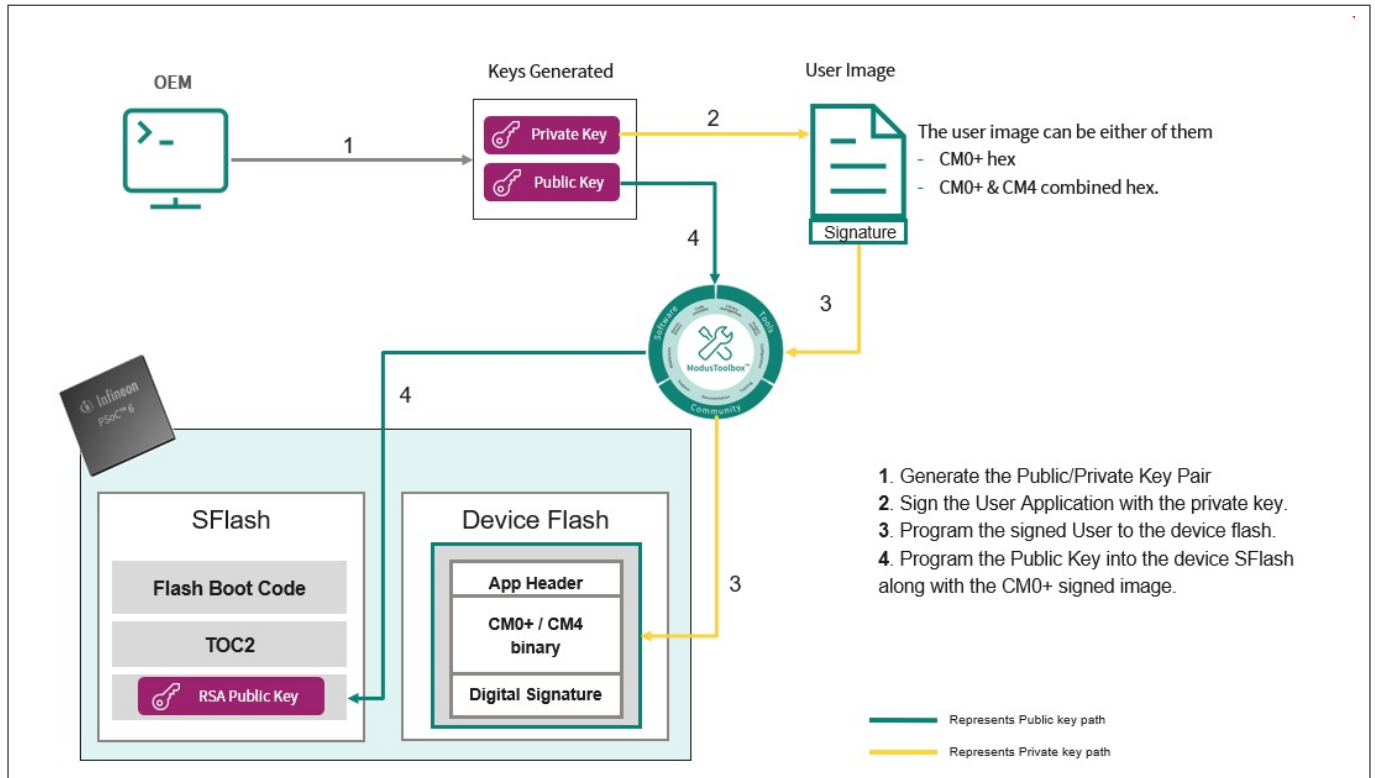
1. Read the image header.
2. Generate the hash (SHA-256) of the image.
3. Decrypt the signature of the image using the public key stored in SFlash and read the existing hash data.
4. Compare the generated hash from Step 2 with the extracted hash from Step 3.
5. If both the hashes match, flash boot passes control to the OEM project, else the boot fails.

### 3.2 Enabling secured boot

This section provides a step-by-step procedure to create a basic secured boot project, program the device, and execute the project without dwelling deep into the security concepts of PSoC™ 6 offerings.

The following figure shows the series of steps required to enable secured boot.

## 3 Secured boot



**Figure 5** Process to enable PSoC™ 6 secured boot

### 3.2.1 Prerequisites

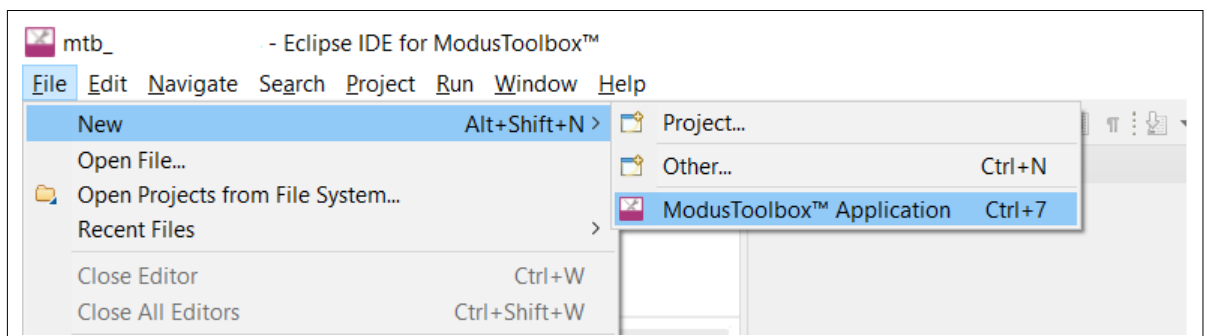
Before you start, ensure that you have the following tools are installed:

- ModusToolbox™ software, version 3.1 or later. Refer to the [ModusToolbox™ Installation Guide](#)

### 3.2.2 Create MTB application (Hello World App)

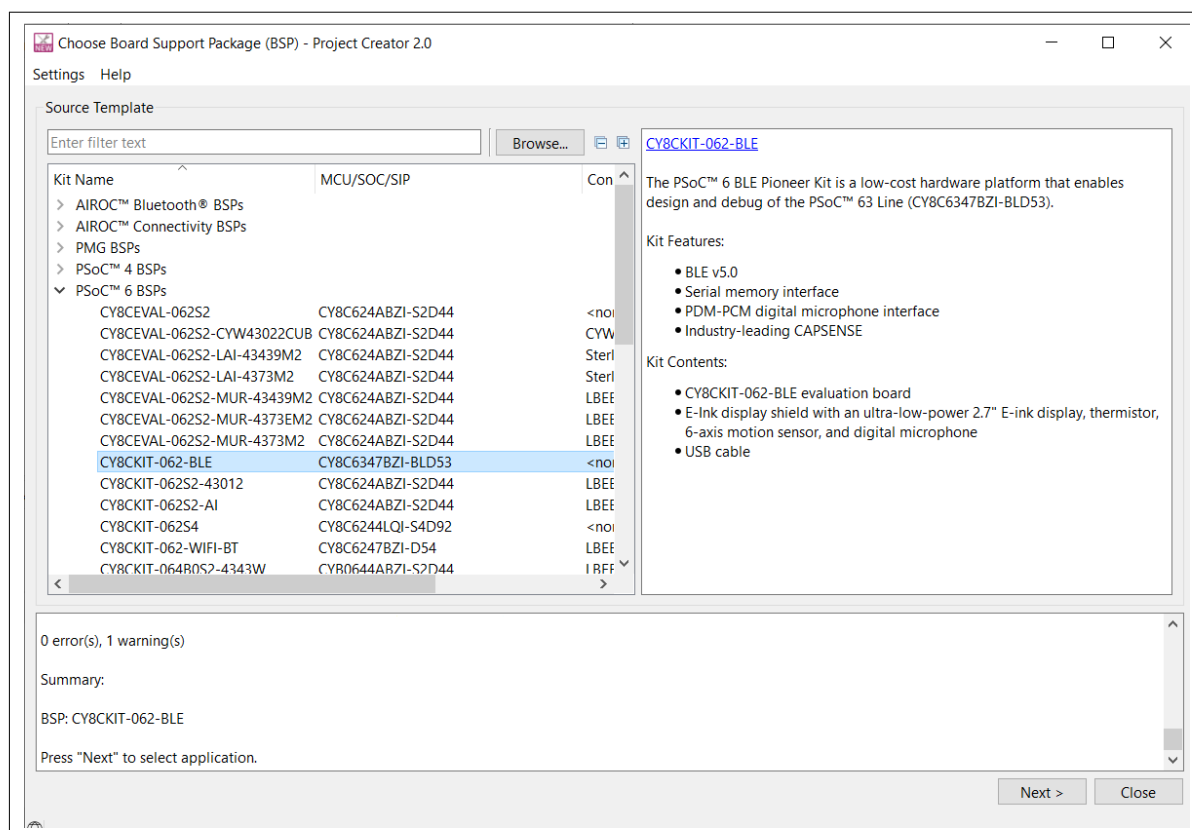
Do the following to create a new project in a new or existing workspace.

1. Launch Eclipse IDE for ModusToolbox™.
2. Select **File > New > ModusToolbox™ Application**
3. In the Project Creator, select the appropriate kit (All PSoC™ 6 kits are supported; this example uses CY8KIT-062-BLE) and click **Next > Create Hello World App**.



**Figure 6** Creating the "Hello World" app

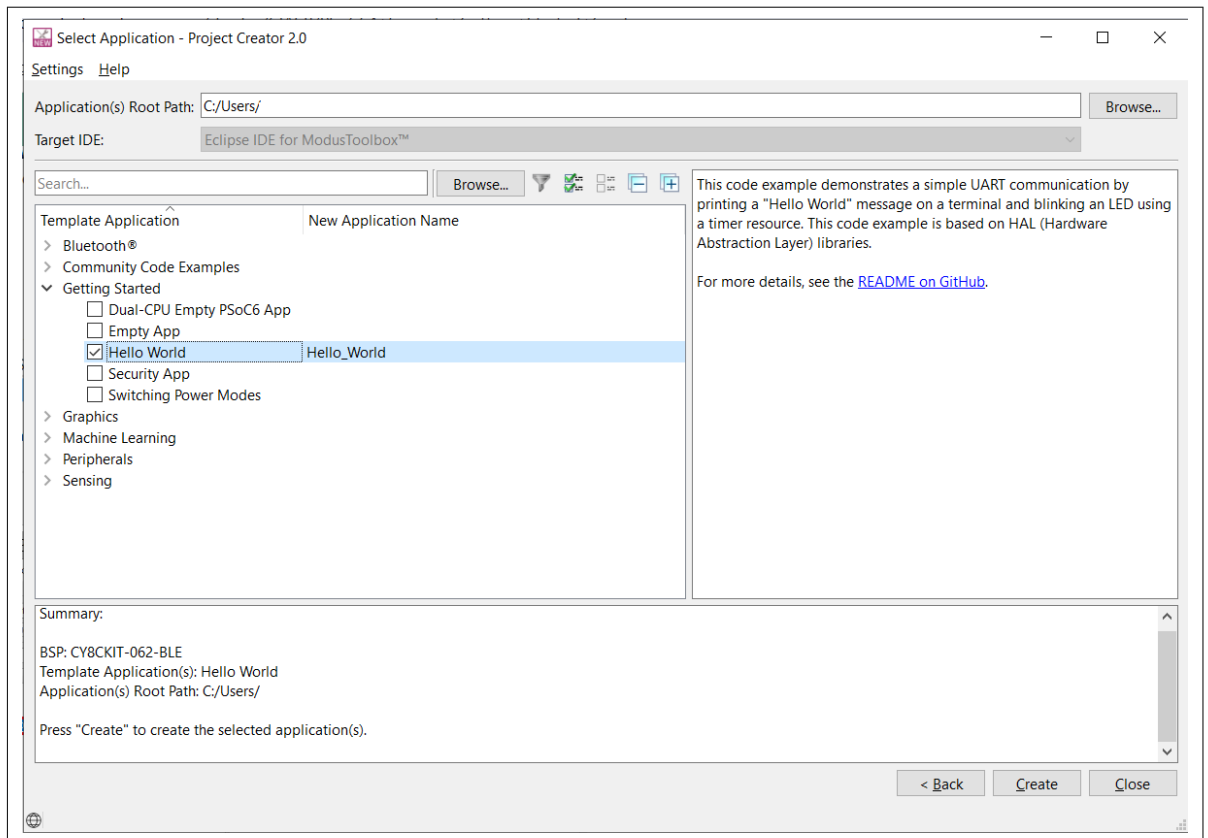
## 3 Secured boot



**Figure 7 Choose appropriate BSP**

4. Select the **Hello World** application under **Getting Started** and click **Create**. This may take a while as it pulls all required sources from respective repositories.

### 3 Secured boot



**Figure 8 Select Hello World Application**

The above steps create a Hello World application that executes on the CM4 core. The CM0+ core hosts a prebuilt image (<https://github.com/Infineon/psoc6cm0p/blob/master/RELEASE.md>). To demonstrate the secured boot, the prebuilt binary will be bypassed to run the "Hello World" application on the CM0+ core instead of the CM4 core with a few configuration settings discussed in later sections.

#### 3.2.3 Adding TOC2

There are two instances of the TOC2: TOC2 and RTOC2, with RTOC2 being the redundant copy of TOC2, each with its own CRC for validation. Each copy of TOC2 is on a separate flash page so that if one is corrupted during writing, it is unlikely that the other is corrupted. Flash boot uses the first one with a valid CRC. If both fail CRC validation, flash boot will remain in a loop, ready to be reprogrammed.

### 3 Secured boot

Once the "Hello World" application is created, declare the TOC2 structure either in the main file or create a header for it.

```

/*****
*
*           Structs
*****

/* Table of Content structure */
typedef struct{
    volatile uint32_t objSize;           /* Object size (Bytes) */
    volatile uint32_t magicNum;          /* TOC ID (magic number) */
    volatile uint32_t userKeyAddr;       /* Secure key address in user Flash */
    volatile uint32_t smifCfgAddr;       /* SMIF configuration structure */
    volatile uint32_t appAddr1;          /* First user application object address */
    volatile uint32_t appFormat1;        /* First user application format */
    volatile uint32_t appAddr2;          /* Second user application object address */
    volatile uint32_t appFormat2;        /* Second user application format */
    volatile uint32_t shashObj;          /* Number of additional objects to be verified (S-HASH) */
    volatile uint32_t sigKeyAddr;        /* Signature verification key address */
    volatile uint32_t addObj[116];       /* Additional objects to include in S-HASH */
    volatile uint32_t tocFlags;          /* Flags in TOC to control Flash boot options */
    volatile uint32_t crc;               /* CRC16-CCITT */
}cy_stc_ps_toc_t;

```

### 3 Secured boot

Now, add the TOC2 and RTOC2 definitions in main.c file before *main()*

```
#define CY_START_OF_FLASH    0x10000000
#define CY_BOOT_BOOTLOADER_SIZE 0x20000

/* Flashboot parameters stored in TOC2 for PSOC6ABLE2 devices */
#if defined(CY_DEVICE_PSOC6ABLE2)
#define CY_PS_FLASHBOOT_FLAGS ((CY_PS_FLASHBOOT_VALIDATE_YES <<
CY_PS_TOC_FLAGS_APP_VERIFY_POS) \
    | (CY_PS_FLASHBOOT_WAIT_20MS << CY_PS_TOC_FLAGS_DELAY_POS) \
    | (CY_PS_FLASHBOOT_CLK_25MHZ << CY_PS_TOC_FLAGS_CLOCKS_POS))
#endif

/* Flashboot parameters stored in TOC2 for PSOC6A2M / PSoC6A512K devices */
#if defined(CY_DEVICE_PSOC6A2M) || defined(CY_DEVICE_PSOC6A512K)
#define CY_PS_FLASHBOOT_FLAGS ((CY_PS_FLASHBOOT_VALIDATE_YES <<
CY_PS_TOC_FLAGS_APP_VERIFY_POS) \
    | (CY_PS_FLASHBOOT_WAIT_20MS << CY_PS_TOC_FLAGS_DELAY_POS) \
    | (CY_PS_FLASHBOOT_CLK_25MHZ << CY_PS_TOC_FLAGS_CLOCKS_POS) \
    | (CY_PS_FLASHBOOT_SWJ_PINS_ENABLE << CY_PS_TOC_FLAGS_SWJ_ENABLE_POS))
#endif

/* TOC2 in SFlash */
CY_SECTION(".cy_toc_part2") __USED static const cy_stc_ps_toc_t cy_toc2 =
{
    .objSize      = sizeof(cy_stc_ps_toc_t) - sizeof(uint32_t), /* Object Size (Bytes)
excluding CRC */
    .magicNum     = CY_PS_TOC2_MAGICNUMBER,                    /* TOC2 ID (magic number) */
    .userKeyAddr  = (uint32_t)&CySecureKeyStorage,             /* User key storage address
*/
    .smifCfgAddr  = 0UL,                                        /* SMIF config list pointer
*/
    .appAddr1     = CY_START_OF_FLASH,                          /* App1 (MCUBoot) start
address */
    .appFormat1   = CY_PS_APP_FORMAT_CYPRESS,                  /* App1 Format */
    .appAddr2     = 0,                                          /* App2 (User App) start
address */
    .appFormat2   = 0,                                          /* App2 Format */
    .shashObj     = 1UL,                                        /* Include public key in
the SECURE HASH */
    .sigKeyAddr   = (uint32_t)&SFLASH->PUBLIC_KEY,             /* Address of signature
verification key */
    .tocFlags     = CY_PS_FLASHBOOT_FLAGS,                     /* Flashboot flags stored
in TOC2 */
    .crc          = 0UL                                         /* CRC populated by
cymcuelftool */
};

/* RTOC2 in SFlash, this is a duplicate of TOC2 for redundancy */
CY_SECTION(".cy_rtoc_part2") __USED static const cy_stc_ps_toc_t cy_rtoc2 =
{
    .objSize      = sizeof(cy_stc_ps_toc_t) - sizeof(uint32_t), /* Object Size (Bytes)
excluding CRC */
```



### 3 Secured boot

```

        .magicNum      = CY_PS_TOC2_MAGICNUMBER,           /* TOC2 ID (magic number) */
        .userKeyAddr   = (uint32_t)&CySecureKeyStorage,     /* User key storage address
    */
        .smifCfgAddr   = 0UL,                               /* SMIF config list pointer
    */
        .appAddr1      = CY_START_OF_FLASH,                /* App1 (MCUBoot) start
address */
        .appFormat1    = CY_PS_APP_FORMAT_CYPRESS,         /* App1 Format */
        .appAddr2      = 0,                                /* App2 (User App) start
address */
        .appFormat2    = 0,                                /* App2 Format */
        .shashObj       = 1UL,                             /* Include public key in
the SECURE HASH */
        .sigKeyAddr    = (uint32_t)&SFLASH->PUBLIC_KEY,     /* Address of signature
verification key */
        .tocFlags      = CY_PS_FLASHBOOT_FLAGS,           /* Flashboot flags stored
in TOC2 */
        .crc           = 0UL                               /* CRC populated by
cymcuelftool */
    };

```

**Note:** The `CY_BOOT_BOOTLOADER_SIZE` can be found in CM0+ linker file, defined as flash region in the **MEMORY** section.

Example: flash (rx): ORIGIN = 0x10000000, LENGTH = **0x20000**

The `CY_SECTION(".cy_toc_part2")` before the structure definition mentions that this structure must be placed in the `.cy_toc_part2` section of the memory, which is the SFlash.

The **.tocFlags** under the TOC2 structure is a combination of various flash boot parameters that can be set. These parameters vary slightly for the 1st and 2nd generation parts. Refer to [AN21111 - PSoC™ 6 MCU designing a custom secured system](#) Section 4.1 - Table of Contents2 (TOC2) to understand more about TOC2 and flash boot parameters.

#### 3.2.4 Add application header and signature

As mentioned in Section 3.1, the initial project that is executed after flash boot must be validated with a public RSA crypto key. To do this, the application must use the CYPRESS™ standard application format (CYSAF) that includes a digital signature and the application header.

Declare the application header structure in the header file created in the previous section.

```

/* Application header in Cypress format */
typedef struct{
    volatile uint32_t objSize;           /* Object size (Bytes) */
    volatile uint32_t appId;             /* Application ID/version */
    volatile uint32_t appAttributes;     /* Attributes (reserved for future use) */
    volatile uint32_t numCores;          /* Number of cores */
    volatile uint32_t core0Vt;          /* (CM0+)VT offset - offset to the vector table from that
entry */
    volatile uint32_t core0Id;           /* CM0+ core ID */
}cy_stc_ps_appheader_t;

```

Now, define the header and signature in `main.c` immediately after the RTOC2 definition.

### 3 Secured boot

#### Application header and signature

```

/*****
 *   Application header and signature
 *****/
#define CY_PS_VT_OFFSET      ((uint32_t)(amp_Vectors[0]) - CY_START_OF_FLASH \
    - offsetof(cy_stc_ps_appheader_t, core0Vt)) /* CM0+ VT Offset */
#define CY_PS_CPUID          (0xC600000UL)      /* CM0+ ARM CPUID[15:4] Reg shifted to
[31:20] */
#define CY_PS_CORE_IDX      (0UL)              /* Index ID of the CM0+ core */

/** Secure Application header */
CY_SECTION(".cy_app_header") __USED
static const cy_stc_ps_appheader_t cy_ps_appHeader = {
    .objSize      = CY_BOOT_BOOTLOADER_SIZE - CY_PS_SECURE_DIGSIG_SIZE,
    .appId        = (CY_PS_APP_VERSION | CY_PS_APP_ID_SECUREIMG),
    .appAttributes = 0UL,                      /* Reserved */
    .numCores     = 1UL,                      /* Only CM0+ */
    .core0Vt      = CY_PS_VT_OFFSET,          /* CM0+ VT offset */
    .core0Id      = CY_PS_CPUID | CY_PS_CORE_IDX, /* CM0+ core ID */
};

/* Secure Digital signature (Populated by cymcuelftool) */
CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
static const uint8_t cy_ps_appSignature[CY_PS_SECURE_DIGSIG_SIZE] = {0u};

```

### 3 Secured boot

#### Add supporting macros and constants for TOC2 and header

You can add the following supporting macros and constants either in *main.c* or to the existing header file created in earlier sections.

```

/*****
*
*           Macros
*****/

/* TOC2 Flag values for PSoC6ABLE2 devices */
#if defined(CY_DEVICE_PSO6ABLE2)
/* Clock selection for Flash boot execution. */
#define CY_PS_FLASHBOOT_CLK_25MHZ      (0x00UL)          /* 25 MHz clock selection for
Flashboot (default) */

/* Debugger wait window selection for Flash boot execution. */
#define CY_PS_FLASHBOOT_WAIT_20MS      (0x00UL)          /* 20ms debugger wait window for
Flashboot (default) */

/* Flash boot validation selection in chip NORMAL mode. */
#define CY_PS_FLASHBOOT_VALIDATE_YES    (0x01UL)          /* Validate app1 (MCUBoot) in
NORMAL mode (default) */
#define CY_PS_FLASHBOOT_VALIDATE_NO     (0x00UL)          /* Do not validate app1 (MCUBoot)
in NORMAL mode */
#endif

/* TOC2 Flag values for PSoC6A2M/PSoC6A512K devices */
#if defined(CY_DEVICE_PSO6A2M) || defined(CY_DEVICE_PSO6A512K)
/* Clock selection for Flash boot execution. */
#define CY_PS_FLASHBOOT_CLK_25MHZ      (0x01UL)          /* 25 MHz clock selection for
Flashboot */

/* Debugger wait window selection for Flash boot execution. */
#define CY_PS_FLASHBOOT_WAIT_20MS      (0x00UL)          /* 20ms debugger wait window for
Flashboot (default) */

/* Determines if SWJ pins are configured by flash boot */
#define CY_PS_FLASHBOOT_SWJ_PINS_ENABLE (0x02)           /* Enable Debug pins (default) */

/* Flash boot validation selection in chip NORMAL mode. */
#define CY_PS_FLASHBOOT_VALIDATE_NO     (0x01UL)          /* Do not validate app1 (MCUBoot)
in NORMAL mode */
#define CY_PS_FLASHBOOT_VALIDATE_YES    (0x02UL)          /* Validate app1 (MCUBoot) in
NORMAL mode (recommended) */
#endif

/* Application format selection for secure boot. */
#define CY_PS_APP_FORMAT_CYPRESS        (1UL)            /* Cypress application format
(Cypress header) - Recommended */

/* Application type selection for secure boot. */
#define CY_PS_APP_ID_SECUREIMG          (0x8002UL)       /* Secure image ID Type */

```

### 3 Secured boot

```

#define CY_PS_SECURE_DIGSIG_SIZE      (256u) /* Size (in Bytes) of the digital signature */

#define CY_PS_VERSION_MAJOR           1UL  /* Major version */
#define CY_PS_VERSION_MINOR          20UL /* Minor version */
#define CY_PS_APP_VERSION              ((CY_PS_VERSION_MAJOR << 24u) | (CY_PS_VERSION_MINOR <<
16u)) /* App Version */

/*****
 *          Constants
 *****/

/* TOC2 Flag offsets and masks for PSoC6ABLE2 devices */
#if defined(CY_DEVICE_PSO6ABLE2)
#define CY_PS_TOC_FLAGS_CLOCKS_POS    (0UL) /* Bit position of Flashboot clock
selection */
#define CY_PS_TOC_FLAGS_DELAY_POS     (2UL) /* Bit position of Flashboot wait
window selection */
#define CY_PS_TOC_FLAGS_APP_VERIFY_POS (31UL) /* Bit position of Flashboot NORMAL
mode app1 validation */
#endif

/* TOC2 Flag offsets and masks for PSoC6A2M/PSoCA512K devices */
#if defined(CY_DEVICE_PSO6A2M) || defined(CY_DEVICE_PSOCA512K)
#define CY_PS_TOC_FLAGS_CLOCKS_POS    (0UL) /* Bit position of Flashboot clock
selection */
#define CY_PS_TOC_FLAGS_DELAY_POS     (2UL) /* Bit position of Flashboot wait
window selection */
#define CY_PS_TOC_FLAGS_SWJ_ENABLE_POS (5UL) /* Bit position of Flashboot SWJ
pins control */
#define CY_PS_TOC_FLAGS_APP_VERIFY_POS (7UL) /* Bit position of Flashboot NORMAL
mode app1 validation */
#endif

#define CY_PS_TOC2_MAGICNUMBER        (0x01211220UL) /* TOC2 identifier */

```

#### 3.2.5 Generating the RSA key pair

Now that you have made the required changes in main.c to enable secure boot, it is time to create the keys. The ModusToolbox™ installation includes the tools required to generate and format the private/public key pairs, including OpenSSL and Python 3.7+. The latest ModusToolbox™ versions do not include Python. Refer to the ModusToolbox™ installation guide to install Python. To access these tools without updating any system paths, use the “modus-shell” command-line shell.

The following steps will generate a custom RSA-2048 public-private key pair in the /keys/ directory and generate a C-compatible public key, which must be manually copied to the cy\_ps\_keystorage.c file.

Do the following to generate a new custom key pair:

1. Start the “modus-shell” application that is installed along with ModusToolbox™ software.
2. Navigate to Hello\_World/ application directory.
3. Copy the [rsa\\_to\\_c.py](#) (The latest ModusToolbox™ versions do not include Python. Refer to the ModusToolbox™ installation guide to install Python) from [mtb-example-psoc6-security/proj\\_btldr\\_cm0p/](#)

3 Secured boot

scripts to Hello\_World/scripts. This script is required to convert the RSA public key in .pub format to a C array

- a. The public key is stored in the device SFlash region in a binary format, not the ASCII format generated by OpenSSL. The modulus, exponent, and three coefficients are pre-calculated to speed up the validation. It has the following format:

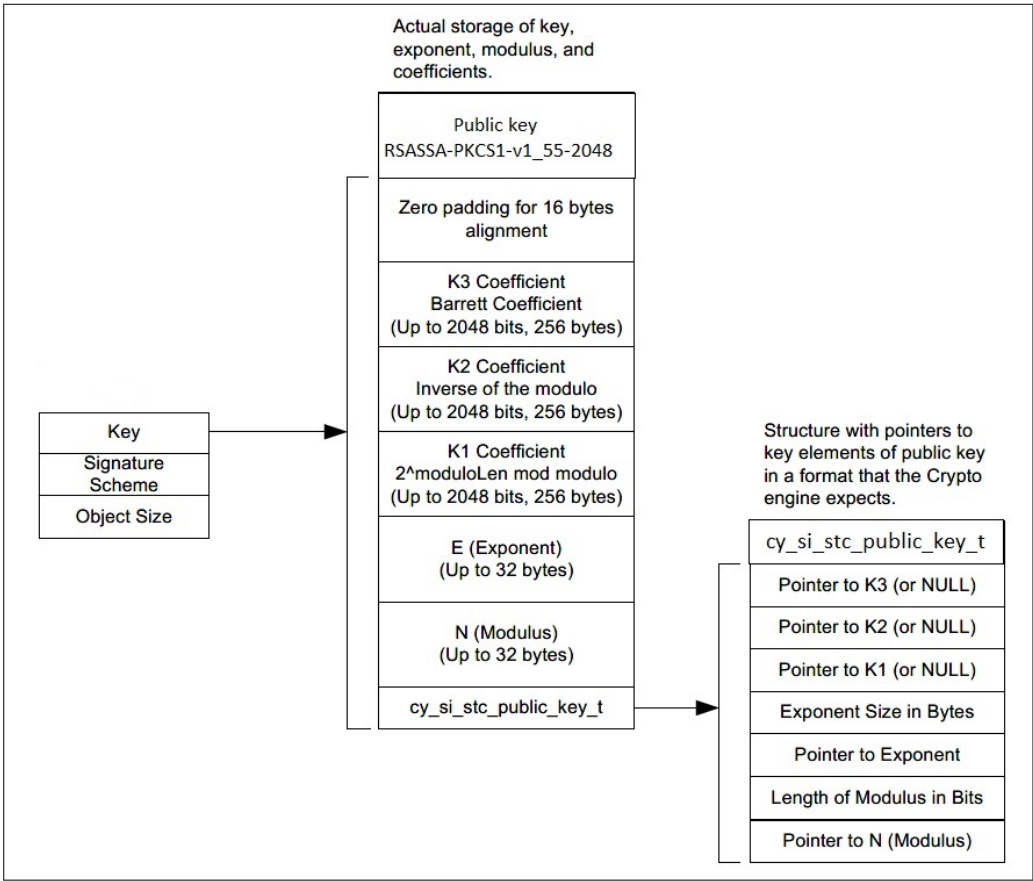


Figure 9 Crypto key structures

- b. The key is stored in three structures:
  - 1. The first structure “Key” is stored as an object. The "Signature Scheme" defines the structure of the key. This example uses RSASSA-PKCS1-v1\_5-2048. The “Object Size” contains the full size of the public key object, which contains the entire three structures.
  - 2. The second structure contains the individual pieces of the public key: coefficients (K1, K2, K3), exponent (E), and modulus (N). These values must be stored in a little-endian list of bytes. OpenSSL generates these values in a big-endian format.
  - 3. The third structure is a list of pointers to each piece of the public key, which is the format required for a call to the Crypto driver. The key is stored in this expanded to speed up the code verification process.
- c. The rsa\_to\_c.py script converts the output from OpenSSL to a format compatible with C and the structures (as mentioned earlier) used by the secure image to store the public key. The output of the rsa\_to\_c.py script is placed in the rsa\_to\_c\_generated.txt file; it consists of the coefficients, exponent and modulus in C array format.

## 3 Secured boot

4. Run the following commands in `modus-shell` under `Hello_world/`:

```
1. mkdir keys
2. openssl genrsa -out ./keys/rsa_private_generated.txt 2048
3. openssl rsa -in ./keys/rsa_private_generated.txt -outform PEM -pubout -out ./keys/
   rsa_public_generated.txt
4. python.exe ./scripts/rsa_to_c.py ./keys/rsa_public_generated.txt > ./keys/
   rsa_to_c_generated.txt
5. cp ./keys/rsa_private_generated.txt ./keys/cypress-test-rsa-2048.pem
```

5. Note that the following files are created in the `/keys/` directory:

```
./keys
- cypress-test-rsa-2048.pem --> Private key used by the signing tool to sign the
  application
- rsa_private_generated.txt
- rsa_public_generated.txt
- rsa_to_c_generated.txt --> consists the public key in array format to be flashed to
  SFlash
```

6. Copy the following arrays from the `rsa_to_c_generated.txt` file and fill into the public key structures that are to be placed in SFlash:

- `.moduleData[ ]`
- `.expData[ ]`
- `.barrettData[ ]`
- `.inverseModuleData[ ]`
- `.rBarData[ ]`

For better readability, two new files (`cy_ps_keystorage.h` and `cy_ps_keystorage.c`) for definition and declaration of the public key structures are created instead of populating them in `main.c`. Because the public key must be stored in SFlash in a structured format as expected by the Crypto driver, you can use these two files as is and replace the array data in `cy_ps_keystorage.c` with the one generated in `rsa_to_c_generated.txt` as mentioned in Step 6 and include the header file in `main.c`.

### 3 Secured boot

#### cy\_ps\_keystorage.h

```
#include <stdint.h>
#include "cy_syslib.h"

/*****
 *
 *      Macros
 *****/

/* Macros used to define the user-defined key array */
#define CY_PS_SECURE_KEY_LENGTH      (256u) /* Key length (Bytes) */
#define CY_PS_SECURE_KEY_ARRAY_SIZE  (4u)   /* Number of Keys */

/* Macros used to define the Public key. */
#define CY_PS_PUBLIC_KEY_RSA_2048    (0UL) /* RSASSA-PKCS1-v1_5-2048 signature scheme */
#define CY_PS_PUBLIC_KEY_RSA_1024    (1UL) /* RSASSA-PKCS1-v1_5-1024 signature scheme */
#define CY_PS_PUBLIC_KEY_STRUCT_OFFSET (8UL) /* Offset to public key struct in number of bytes */
#define CY_PS_PUBLIC_KEY_MODULOLENGTH (256UL) /* Modulus length of the RSA key */
#define CY_PS_PUBLIC_KEY_EXPLENGTH    (32UL) /* Exponent length of the RSA key */
#define CY_PS_PUBLIC_KEY_SIZEOF_BYTE  (8UL) /* Size of Byte in number of bits */

/*****
 *
 *      Structs
 *****/

/* Public key definition structure as expected by the Crypto driver */
typedef struct
{
    uint32_t moduloAddr; /* Address of the public key modulus */
    uint32_t moduloSize; /* Size (bits) of the modulus part of the public key */
    uint32_t expAddr; /* Address of the public key exponent */
    uint32_t expSize; /* Size (bits) of the exponent part of the public key */
    uint32_t barrettAddr; /* Address of the Barret coefficient */
    uint32_t inverseModuloAddr; /* Address of the binary inverse modulo */
    uint32_t rBarAddr; /* Address of the (2^moduloLength mod modulo) */
} cy_ps_stc_crypto_public_key_t;

/* Public key structure */
typedef struct
{
    uint32_t objSize; /* Public key Object size */
    uint32_t signatureScheme; /* Signature scheme */
    cy_ps_stc_crypto_public_key_t publicKeyStruct; /* Public key definition struct */
} /*
    uint8_t moduloData[CY_PS_PUBLIC_KEY_MODULOLENGTH]; /* Modulo data */
    uint8_t expData[CY_PS_PUBLIC_KEY_EXPLENGTH]; /* Exponent data */
    uint8_t barrettData[CY_PS_PUBLIC_KEY_MODULOLENGTH + 4UL]; /* Barret coefficient data */
    uint8_t inverseModuloData[CY_PS_PUBLIC_KEY_MODULOLENGTH]; /* Binary inverse modulo data */
    uint8_t rBarData[CY_PS_PUBLIC_KEY_MODULOLENGTH]; /* 2^moduloLength mod modulo
data */
} cy_ps_stc_public_key_t;

/*****
```

### 3 Secured boot

```
*           Globals
*****/
/* Secure Key Storage (Note: Ensure that the alignment matches the Protection unit
configuration) */
extern const uint8_t CySecureKeyStorage[CY_PS_SECURE_KEY_ARRAY_SIZE][CY_PS_SECURE_KEY_LENGTH];

/* Public key in SFlash */
extern const cy_ps_stc_public_key_t cy_publicKey;

/* [] END OF FILE */
```

**Note:** *The array data shown in the file below must be replaced with your key data. If not, the device will not boot due to incorrect public key.*



### 3 Secured boot

**cy\_ps\_keystorage.c** (The data to be replaced with the generated key data as shown in bold)

```
#include <cy_ps_keystorage.h>

/* Secure Key Storage (Note: Ensure that the alignment matches the Protection unit
configuration) */
CY_ALIGN(1024) __USED const uint8_t CySecureKeyStorage[CY_PS_SECURE_KEY_ARRAY_SIZE]
[CY_PS_SECURE_KEY_LENGTH] = {
    {0x00u}, /* Insert user key #1 values */
    {0x00u}, /* Insert user key #2 values */
    {0x00u}, /* Insert user key #3 values */
    {0x00u} /* Insert user key #4 values */
};

/* Public key in SFlash */
CY_SECTION(".cy_sflash_public_key") __USED const cy_ps_stc_public_key_t cy_publicKey =
{
    .objSize = sizeof(cy_ps_stc_public_key_t),
    .signatureScheme = CY_PS_PUBLIC_KEY_RSA_2048,
    .publicKeyStruct =
    {
        .moduloAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, moduloData),
        .moduloSize = CY_PS_PUBLIC_KEY_SIZEOF_BYTE * CY_PS_PUBLIC_KEY_MODULOLENGTH,
        .expAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, expData),
        .expSize = CY_PS_PUBLIC_KEY_SIZEOF_BYTE * CY_PS_PUBLIC_KEY_EXPLENGTH,
        .barrettAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, barrettData),
        .inverseModuloAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, inverseModuloData),
        .rBarAddr = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_ps_stc_public_key_t, rBarData),
    },
    .moduloData =
    {
        0x17u, 0xA0u, 0xE6u, 0x98u, 0xC9u, 0xD0u, 0x8Cu, 0xB9u,
0x9Cu, 0xF0u, 0xCCu, 0x7Eu, 0x5Cu, 0xB6u, 0x63u, 0x92u,
0xA6u, 0x0Au, 0x8Au, 0xE0u, 0xA0u, 0xEEu, 0xF3u, 0x9Cu,
0xDDu, 0x23u, 0xD8u, 0xA5u, 0xF3u, 0x84u, 0xA9u, 0x67u,
0xC4u, 0xEEu, 0x71u, 0x72u, 0xE7u, 0x98u, 0x64u, 0xB9u,
0xC1u, 0xEFu, 0xC7u, 0xEFu, 0x36u, 0x9Bu, 0x5Bu, 0xE5u,
0x7Cu, 0x26u, 0xD6u, 0x3Du, 0xA4u, 0x25u, 0x03u, 0x7Du,
0x68u, 0x2Au, 0x9Eu, 0x51u, 0x34u, 0xD9u, 0xBCu, 0xA8u,
0xDBu, 0x0Au, 0xA8u, 0xDCu, 0x20u, 0x43u, 0xD1u, 0xC8u,
0x6Du, 0xBDu, 0xD4u, 0x0Eu, 0xEFu, 0x55u, 0x50u, 0x72u,
0x64u, 0x2Bu, 0xA5u, 0xAFu, 0xC1u, 0x1Fu, 0x74u, 0x5Eu,
0xD6u, 0x12u, 0xB6u, 0xD9u, 0x84u, 0x34u, 0xEAu, 0x30u,
0xF2u, 0xBFu, 0xD8u, 0x10u, 0x50u, 0x91u, 0xC2u, 0x54u,
0xF4u, 0x3Bu, 0x09u, 0xA6u, 0x1Bu, 0x01u, 0xCAu, 0x94u,
0xAAu, 0x6Bu, 0xF0u, 0x96u, 0x82u, 0x8Au, 0x42u, 0x5Eu,
0x52u, 0xA3u, 0xEBu, 0x36u, 0xF0u, 0xD9u, 0xD4u, 0x37u,
0x12u, 0xCCu, 0x03u, 0xA0u, 0x06u, 0x78u, 0xD4u, 0x94u,
    }
};
```

### 3 Secured boot

```

    0xC6u, 0x2Cu, 0x99u, 0xCDu, 0x00u, 0xFEu, 0x2Cu, 0xB9u,
    0xE6u, 0x76u, 0x48u, 0xE8u, 0x08u, 0xDAu, 0x86u, 0x2Bu,
    0x59u, 0x72u, 0xC1u, 0x66u, 0x92u, 0xB9u, 0x6Eu, 0x49u,
    0x07u, 0x46u, 0x06u, 0x0Cu, 0x88u, 0x7Du, 0x90u, 0x94u,
    0x43u, 0x35u, 0x89u, 0x18u, 0x75u, 0x6Du, 0xD7u, 0x5Eu,
    0x7Eu, 0x57u, 0xAAu, 0xD8u, 0x5Fu, 0xD4u, 0x24u, 0x69u,
    0x70u, 0xD0u, 0x77u, 0xB0u, 0x8Bu, 0x34u, 0x6Eu, 0xA1u,
    0xD9u, 0x7Fu, 0x12u, 0xF2u, 0x37u, 0x42u, 0x02u, 0x3Fu,
    0x77u, 0xD7u, 0x15u, 0x45u, 0xB3u, 0x0Cu, 0x69u, 0xB6u,
    0x00u, 0x8Bu, 0xE6u, 0x0Au, 0x16u, 0x4Bu, 0x13u, 0x4Au,
    0x11u, 0x42u, 0xBAu, 0x07u, 0xDBu, 0xDBu, 0xADu, 0xAFu,
    0x59u, 0x38u, 0xCDu, 0x9Au, 0x30u, 0x7Bu, 0x43u, 0xD4u,
    0xFBu, 0x12u, 0xA9u, 0x3Fu, 0x3Bu, 0xB0u, 0x0Cu, 0xC8u,
    0x4Au, 0x9Bu, 0x36u, 0x37u, 0xC0u, 0x52u, 0xC6u, 0x4Cu,
    0x17u, 0x14u, 0x0Fu, 0x89u, 0x54u, 0x47u, 0xBFu, 0xB1u,
},
.expData =
{
    0x01u, 0x00u, 0x01u, 0x00u,
},
.barrettData =
{
    0x2Au, 0xA2u, 0x13u, 0x54u, 0xE6u, 0xDCu, 0x4Cu, 0xD1u,
    0xDCu, 0x75u, 0x12u, 0xAAu, 0x36u, 0xC1u, 0x11u, 0xFAu,
    0xFAu, 0x82u, 0x6Cu, 0xCAu, 0xE9u, 0xC1u, 0xA5u, 0xCCu,
    0x8Cu, 0xFDu, 0x57u, 0xFFu, 0xAFu, 0x30u, 0x45u, 0x44u,
    0x59u, 0x25u, 0x98u, 0x09u, 0x0Du, 0xFFu, 0xB8u, 0xC1u,
    0x15u, 0xDBu, 0x07u, 0x45u, 0x16u, 0xE3u, 0xEFu, 0xB3u,
    0x5Bu, 0x34u, 0x3Au, 0x6Cu, 0xCEu, 0x11u, 0x38u, 0x69u,
    0x4Bu, 0xB0u, 0xCDu, 0x20u, 0x8Du, 0x09u, 0xA4u, 0xCDu,
    0x64u, 0x52u, 0x86u, 0x71u, 0x7Au, 0x10u, 0x3Bu, 0x61u,
    0xCCu, 0xC5u, 0xCFu, 0xFCu, 0x7Du, 0xF4u, 0xC4u, 0x5Au,
    0x0Fu, 0x38u, 0x9Au, 0x08u, 0xD2u, 0xD0u, 0x9Au, 0x87u,
    0xDFu, 0x6Cu, 0x4Eu, 0x85u, 0xB2u, 0x7Au, 0x22u, 0x58u,
    0x3Fu, 0x61u, 0x87u, 0x69u, 0xA2u, 0xD0u, 0x41u, 0x34u,
    0x1Bu, 0xEAu, 0ABu, 0xCDu, 0xE8u, 0xA0u, 0x52u, 0x78u,
    0x22u, 0x86u, 0xF9u, 0x3Du, 0x35u, 0xA6u, 0xEDu, 0x14u,
    0xF7u, 0xBCu, 0x45u, 0x36u, 0x47u, 0x17u, 0xE5u, 0xF3u,
    0x05u, 0xEDu, 0x1Du, 0xA5u, 0x47u, 0xB1u, 0x14u, 0x31u,
    0x08u, 0xEFu, 0x1Eu, 0x89u, 0x77u, 0x56u, 0x07u, 0x76u,
    0xE1u, 0xD9u, 0x32u, 0xC2u, 0xE0u, 0ABu, 0x24u, 0x5Cu,
    0x5Au, 0xD8u, 0x54u, 0xFFu, 0x97u, 0x38u, 0x4Au, 0x9Du,
    0x01u, 0xBEu, 0xB7u, 0x1Au, 0x1Fu, 0xE4u, 0x8Cu, 0ABu,
    0x22u, 0x80u, 0x7Du, 0x5Eu, 0xF5u, 0x76u, 0xE1u, 0xFAu,
    0x01u, 0x75u, 0xCBu, 0x86u, 0x66u, 0x82u, 0xC9u, 0x53u,
    0x02u, 0x0Au, 0x4Eu, 0x71u, 0xF8u, 0x64u, 0x35u, 0xEBu,
    0x68u, 0xF6u, 0xDFu, 0xE4u, 0xFBu, 0x09u, 0x43u, 0x37u,
    0xA2u, 0xE7u, 0xE8u, 0x50u, 0x40u, 0x7Du, 0x9Au, 0xF7u,
    0BDu, 0xC8u, 0xC9u, 0xB6u, 0xB9u, 0x2Cu, 0BBu, 0xB8u,
    0x5Du, 0x1Cu, 0xFFu, 0xD4u, 0x1Eu, 0xC3u, 0xF9u, 0x96u,
    0x62u, 0x55u, 0xF0u, 0x42u, 0x64u, 0xB8u, 0x05u, 0x7Fu,
    0x61u, 0xF7u, 0x83u, 0xCFu, 0xFDu, 0x69u, 0xE8u, 0x85u,
    0x47u, 0x0Bu, 0x04u, 0x29u, 0x93u, 0xCCu, 0x4Bu, 0xA7u,

```

### 3 Secured boot

```

    0x8Au, 0xCFu, 0xF3u, 0x7Fu, 0x8Fu, 0x15u, 0xB4u, 0x70u,
    0x01u, 0x00u, 0x00u, 0x00u,
},
.inverseModuloData =
{
    0x59u, 0x68u, 0x1Fu, 0xDCu, 0xEDu, 0xCFu, 0x53u, 0x57u,
    0x93u, 0x43u, 0x56u, 0x22u, 0xE4u, 0x4Cu, 0xFAu, 0x5Du,
    0xADu, 0xB8u, 0x32u, 0x27u, 0x75u, 0x12u, 0x02u, 0xFEu,
    0xAEu, 0xE0u, 0xB7u, 0xDBu, 0xCBu, 0x07u, 0xE2u, 0x1Du,
    0x6Eu, 0x21u, 0xD3u, 0x70u, 0x10u, 0x08u, 0x36u, 0x62u,
    0x11u, 0xC3u, 0x97u, 0x2Eu, 0x32u, 0xDBu, 0x2Au, 0xA3u,
    0x12u, 0x29u, 0x65u, 0x47u, 0xEFu, 0x3Cu, 0xC8u, 0xD0u,
    0xD6u, 0x47u, 0x8Eu, 0xD3u, 0xAAu, 0x33u, 0x83u, 0xC3u,
    0x08u, 0x2Cu, 0x7Bu, 0xD5u, 0x41u, 0xD2u, 0x83u, 0x57u,
    0xF1u, 0x31u, 0xB2u, 0x78u, 0x63u, 0xC5u, 0xD8u, 0xC0u,
    0xECu, 0x14u, 0x79u, 0x20u, 0xCEu, 0x6Bu, 0x5Au, 0x2Eu,
    0x31u, 0xB7u, 0x8Bu, 0xCCu, 0x43u, 0x18u, 0x59u, 0xB2u,
    0x92u, 0x6Cu, 0x09u, 0xACu, 0x67u, 0x89u, 0x87u, 0x1Cu,
    0x2Bu, 0x08u, 0xF1u, 0xECu, 0xB8u, 0x16u, 0x6Bu, 0xB8u,
    0x9Bu, 0xE6u, 0x43u, 0xCAu, 0xE2u, 0xD7u, 0xC0u, 0xEDu,
    0xDBu, 0x12u, 0xF5u, 0x95u, 0xF0u, 0xE6u, 0x10u, 0xA1u,
    0x46u, 0x9Cu, 0x97u, 0x9Cu, 0x25u, 0x86u, 0xD6u, 0xD6u,
    0x3Eu, 0x00u, 0x7Bu, 0x19u, 0xFAu, 0x13u, 0xEAu, 0x3Au,
    0x34u, 0xDFu, 0xCBu, 0xACu, 0x42u, 0x67u, 0xFFu, 0x51u,
    0xDEu, 0xC3u, 0xBAu, 0xA1u, 0xF4u, 0x53u, 0x6Eu, 0xC9u,
    0x06u, 0x63u, 0x41u, 0xA7u, 0x5Bu, 0xD7u, 0x28u, 0x17u,
    0x62u, 0x23u, 0x83u, 0x45u, 0x44u, 0x96u, 0x4Bu, 0xC7u,
    0x5Au, 0x83u, 0xB8u, 0xF5u, 0x0Bu, 0x4Cu, 0x3Eu, 0x6Du,
    0x48u, 0xA5u, 0xF1u, 0x85u, 0x52u, 0xCDu, 0x30u, 0xC3u,
    0xCDu, 0xF0u, 0x13u, 0x10u, 0x82u, 0x5Fu, 0x9Eu, 0xEBu,
    0x9Du, 0x0Eu, 0xF7u, 0x27u, 0x5Du, 0xEBu, 0x46u, 0x27u,
    0x56u, 0xB4u, 0x1Cu, 0xA2u, 0x11u, 0x30u, 0xADu, 0x91u,
    0x07u, 0x30u, 0x48u, 0xBDu, 0xE1u, 0xA7u, 0x3Bu, 0x79u,
    0x78u, 0xB2u, 0x89u, 0x9Au, 0x5Bu, 0xA3u, 0x3Bu, 0x8Cu,
    0xC7u, 0xA1u, 0x98u, 0x18u, 0xD9u, 0x6Cu, 0x8Au, 0x9Eu,
    0x96u, 0x71u, 0x09u, 0x08u, 0x39u, 0xA8u, 0xBAu, 0x14u,
    0x6Fu, 0x34u, 0x4Du, 0x62u, 0xD3u, 0xE8u, 0x84u, 0xEDu,
},
.rBarData =
{
    0xE9u, 0x5Fu, 0x19u, 0x67u, 0x36u, 0x2Fu, 0x73u, 0x46u,
    0x63u, 0x0Fu, 0x33u, 0x81u, 0xA3u, 0x49u, 0x9Cu, 0x6Du,
    0x59u, 0xF5u, 0x45u, 0x1Fu, 0x5Fu, 0x11u, 0x0Cu, 0x63u,
    0x22u, 0xDCu, 0x27u, 0x5Au, 0x0Cu, 0x7Bu, 0x56u, 0x98u,
    0x3Bu, 0x11u, 0x8Eu, 0x8Du, 0x18u, 0x67u, 0x9Bu, 0x46u,
    0x3Eu, 0x10u, 0x38u, 0x10u, 0xC9u, 0x64u, 0xA4u, 0x1Au,
    0x83u, 0xD9u, 0x29u, 0xC2u, 0x5Bu, 0xDAu, 0xFCu, 0x82u,
    0x97u, 0xD5u, 0x61u, 0xAEu, 0xCBu, 0x26u, 0x43u, 0x57u,
    0x24u, 0xF5u, 0x57u, 0x23u, 0xDFu, 0xBCu, 0x2Eu, 0x37u,
    0x92u, 0x42u, 0x2Bu, 0xF1u, 0x10u, 0xAAu, 0xAFu, 0x8Du,
    0x9Bu, 0xD4u, 0x5Au, 0x50u, 0x3Eu, 0xE0u, 0x8Bu, 0xA1u,
    0x29u, 0xEDu, 0x49u, 0x26u, 0x7Bu, 0xCBu, 0x15u, 0xCFu,
    0x0Du, 0x40u, 0x27u, 0xEFu, 0xAFu, 0x6Eu, 0x3Du, 0xABu,

```

### 3 Secured boot

```

0x0Bu, 0xC4u, 0xF6u, 0x59u, 0xE4u, 0xFEu, 0x35u, 0x6Bu,
0x55u, 0x94u, 0x0Fu, 0x69u, 0x7Du, 0x75u, 0xBDu, 0xA1u,
0xADu, 0x5Cu, 0x14u, 0xC9u, 0x0Fu, 0x26u, 0x2Bu, 0xC8u,
0xEDu, 0x33u, 0xFCu, 0x5Fu, 0xF9u, 0x87u, 0x2Bu, 0x6Bu,
0x39u, 0xD3u, 0x66u, 0x32u, 0xFFu, 0x01u, 0xD3u, 0x46u,
0x19u, 0x89u, 0xB7u, 0x17u, 0xF7u, 0x25u, 0x79u, 0xD4u,
0xA6u, 0x8Du, 0x3Eu, 0x99u, 0x6Du, 0x46u, 0x91u, 0xB6u,
0xF8u, 0xB9u, 0xF9u, 0xF3u, 0x77u, 0x82u, 0x6Fu, 0x6Bu,
0xBCu, 0xCAu, 0x76u, 0xE7u, 0x8Au, 0x92u, 0x28u, 0xA1u,
0x81u, 0xA8u, 0x55u, 0x27u, 0xA0u, 0x2Bu, 0xDBu, 0x96u,
0x8Fu, 0x2Fu, 0x88u, 0x4Fu, 0x74u, 0xCB u, 0x91u, 0x5Eu,
0x26u, 0x80u, 0xEDu, 0x0Du, 0xC8u, 0xBDu, 0xFDu, 0xC0u,
0x88u, 0x28u, 0xEAu, 0xBAu, 0x4Cu, 0xF3u, 0x96u, 0x49u,
0xFFu, 0x74u, 0x19u, 0xF5u, 0xE9u, 0xB4u, 0xECu, 0xB5u,
0xEEu, 0xBDu, 0x45u, 0xF8u, 0x24u, 0x24u, 0x52u, 0x50u,
0xA6u, 0xC7u, 0x32u, 0x65u, 0xCFu, 0x84u, 0xBCu, 0x2Bu,
0x04u, 0xEDu, 0x56u, 0xC0u, 0xC4u, 0x4Fu, 0xF3u, 0x37u,
0xB5u, 0x64u, 0xC9u, 0xC8u, 0x3Fu, 0xADu, 0x39u, 0xB3u,
0xE8u, 0xEBu, 0xF0u, 0x76u, 0xABu, 0xB8u, 0x40u, 0x4Eu,
},
};

/* [] END OF FILE */

```

#### 3.2.6 Signing the application

After generating the keys as described in the previous section, you should sign the application (Hello\_World) with the generated RSA private key. As mentioned in [Requirements](#), the signed application image should be of CYPRESS standard application format (CYSAF) with a header at the start and the signature at the end of the image. Before you sign the application, you must create the header and signature sections to store the respective data as part of the signed image. These sections are created by adding their entries in the CM0+ linker script as shown below.

Open \Hello\_World\bsps\TARGET\_APP\_CY8CKIT-062-BLE\COMPONENT\_CM0P\TOOLCHAIN\_GCC\_ARM\linker.ld file and add the following header section before the .text section of CM0+ as the header is always placed before the start of the image.

**Note:** This document uses TOOLCHAIN\_GCC\_ARM as the default compiler.

```

SECTIONS
{
++   .cy_app_header :
++   {
++       KEEP(*(.cy_app_header))
++   } > flash

/* Cortex-M0+ application flash area */
.text :
{

```

### 3 Secured boot

Similarly, add a section for the application signature in the linker script after the `.heap` section because the signature is placed at the end of the image.

```
/* Used for the digital signature of the secure application and the Bootloader SDK application.
 * The size of the section depends on the required data size. */
.cy_app_signature ORIGIN(flash) + LENGTH(flash) - 256 :
{
    KEEP(*(.cy_app_signature))
} > flash
```

While signing the application, the signing tool (mcuelftool) requires the signature size, image size, and start address to generate the signature. Therefore, add these symbols at the end of the linker script along with other existing symbols required by the signing tool to generate the signature.

```
/* The size of the application signature is 256 for RSA 2048. */
__cy_boot_signature_size = 256;

__cy_app_verify_start = ORIGIN(flash);
__cy_app_verify_length = LENGTH(flash) - __cy_boot_signature_size;
```

Once the header and signature sections are added to the linker script, add the signing command in the Makefile. The signing is performed on the `Hello world` application as part of post-build commands during build.

### 3 Secured boot

Add the following lines to the Hello World application Makefile after the PREBUILD command.

```
# Application Signing using MCUELFTOOL

$(info Tools Directory: $(CY_TOOLS_DIR))
CY_BUILD_LOCATION=./build
BINARY_OUT_PATH=$(CY_BUILD_LOCATION)/$(TARGET)/$(CONFIG)/$(APPNAME)
CY_MCUELFTOOL_DIR=$(wildcard $(CY_TOOLS_DIR)/cymcuelftool-*)
MCUELFTOOL_LOC=$(CY_MCUELFTOOL_DIR)/bin/cymcuelftool
SREC_CAT_LOC=$(CY_TOOLS_DIR)/srecord/bin/srec_cat

# Custom post-build commands to run.
POSTBUILD+=\
cp -f $(BINARY_OUT_PATH).hex $(BINARY_OUT_PATH)_unsigned.hex;\
cp -f $(BINARY_OUT_PATH).elf $(BINARY_OUT_PATH)_unsigned.elf;\
rm -f $(BINARY_OUT_PATH).elf;\
$(MCUELFTOOL_LOC) --sign $(BINARY_OUT_PATH)_unsigned.elf SHA256 --encrypt RSASSA-PKCS \
--key ./keys/cypress-test-rsa-2048.pem --output $(BINARY_OUT_PATH).elf \
--hex $(BINARY_OUT_PATH).hex;

METADATA_START_ADDR=0x90300000
METADATA_END_ADDR=0x90700000

POSTBUILD+=\
$(SREC_CAT_LOC) $(BINARY_OUT_PATH).hex -intel -exclude $(METADATA_START_ADDR) \
$(METADATA_END_ADDR) -o $(BINARY_OUT_PATH)_stripped.hex -intel --Output_Block_Size 16 ; \
cp -f $(BINARY_OUT_PATH)_stripped.hex $(BINARY_OUT_PATH).hex; \
rm -f $(BINARY_OUT_PATH)_stripped.hex;
```

The above POSTBUILD script calculates the hash digest of the hex file generated, and encrypts the hash digest with the RSA-2048 private key to generate a digital signature. This signature is placed at the end of the image through **mcueelftool**, thereby generating a signed image. The **mcueelftool** also generates the CRC for TOC2 and RTOC2 and installs it.

By default, the Hello World application is configured to run on the CM4 core. Because this document is intended to demonstrate secured boot on the CM0+ core, change the core name in the Makefile to enable the application execution on CM0+ by adding the following lines in the Makefile after APPNAME.

```
# CPU to target; CM4 is the default CPU when this variable is not present.
CORE=CM0P
CORE_NAME=CM0P_0
```

The following sections will demonstrate how to add a CM4 project along with CM0 project.

This ends the change list required to enable secured boot with the Hello World example. Although secure boot is enabled, it doesn't guarantee that the device is secured unless it is transitioned into the SECURE LCS.

#### 3.2.7 Programming the image

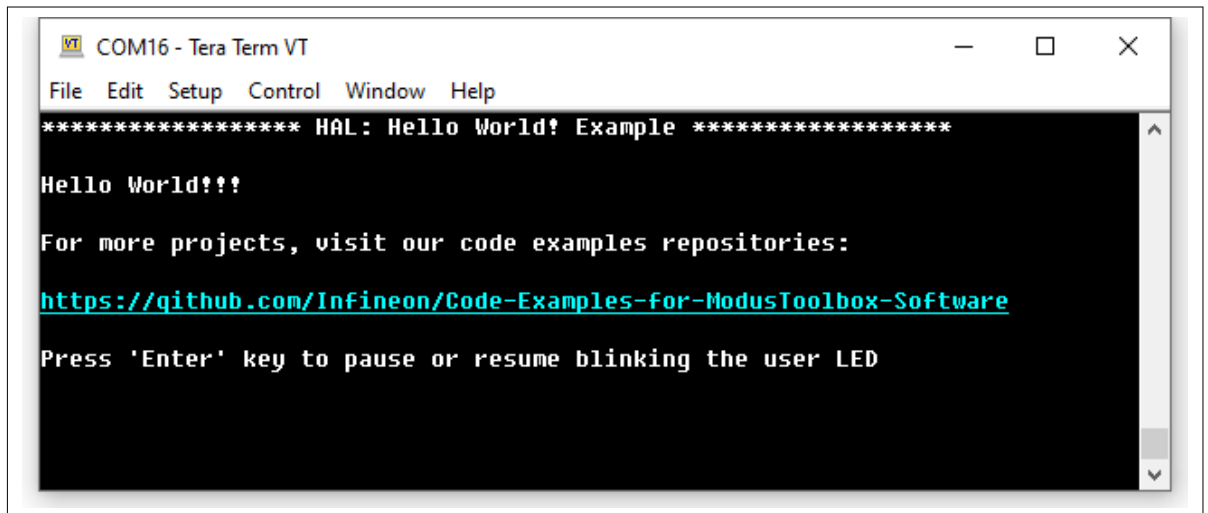
After adding all the necessary changes to the code, build it on ModusToolbox™. Once built, two hex files are generated in the build folder:

## 3 Secured boot

- mtb-example-hal-hello-world.hex - This is the signed application
- mtb-example-hal-hello-world\_unsigned.hex - This is an unsigned application

To demonstrate secured boot, program the signed application as follows:

1. Connect the board to your PC using the provided USB cable through the KitProg3 USB connector
2. Open a terminal program and select the KitProg3 COM port. Set the serial port parameters to 8N1 and 115200 baud
3. Before programming the image, erase the entire Sflash using mtb-programmer tool by selecting **Erase/Program entire Sflash allowed** option under **Probe Settings > Sflash Restrictions**
4. Program the board using Eclipse IDE (you can use other supporting IDEs or MTB Programmer)
  - a. Select the application project in Project Explorer.
  - b. In the **Quick Panel**, scroll down, and click **<Application Name> Program (KitProg3\_MiniProg4)**
5. After programming the application, confirm that "HAL: Hello World! Example" is displayed on the UART terminal. This means that the application is validated successfully by flash boot and executed



**Figure 10** Application execution with secured boot enabled

6. Confirm that the kit LED blinks at approximately 1 Hz

## 4 Device lifecycle stages (LCS)

### 4 Device lifecycle stages (LCS)

The device lifecycle is a key aspect of the PSoC™ 6 MCU's security. In the lifecycle stages, the device goes through a sequence of stages and the progression is dictated by blowing the eFuse/OTP (changing values from '0' to '1') in the device. The LCS scheme follows a strict irreversible progression to protect the internal device data and code at the level required by the customer.

The PSoC™ 6 device has the following lifecycle stages:

1. **VIRGIN** - This is the initial lifecycle stage of the device when manufactured. During this stage, trim values and Flash boot are written into SFlash. Parts in this stage leave the factor only after transitioned to NORMAL LCS.
2. **NORMAL** - This is the lifecycle stage in which the parts are sent to customers. By default, users have full debug access and may program all user flash including certain areas in SFlash such as user SFlash, TOC2, and public key
3. **SECURE** - This is the lifecycle stage of a secured device. After an MCU is in the SECURE lifecycle stage, there is no going back. The debug ports may be disabled, programming restrictions in the user flash, certain areas in SFlash such as user SFlash, TOC2, and public key. Code should be tested in NORMAL or SECURE\_WITH\_DEBUG lifecycle stages before moving to the SECURE lifecycle stage. This is to prevent a configuration error that could cause the part to be no longer accessible for device programming and therefore unusable
4. **SECURE\_WITH\_DEBUG** - This is the same as the SECURE lifecycle stage, except with NAR (Normal Access Restrictions) applied to enable debugging. When in the SECURE\_WITH\_DEBUG lifecycle stage, the access restrictions are taken from the NAR located in SFlash. Parts put in this stage cannot be changed back to either SECURE or NORMAL stage; they are most likely discarded or destroyed after testing
5. **RMA** - The RMA lifecycle stage is a mechanism to allow customers to return parts that are in the SECURE lifecycle stage back to Infineon to evaluate if a defect in the device is suspected

For more information on LCS, refer to [AN21111 - PSoC™ 6 MCU designing a custom secured system](#)

There are two types of device boot flows in NORMAL LCS:

- NORMAL (no validation, no code security)
- NORMAL with Validate (no code security)

When the part is sent to the customer from the factory, it boots in NORMAL (no validation, no code security) LCS. As the customer enables the code validation (secured boot), it boots in NORMAL with validate.

**NORMAL with validate** is the NORMAL lifecycle stage with the option to validate the first code. It operates just as the SECURE lifecycle stage, but skips secure hash validation, uses NAR instead of SAR, and configures the debug GPIOs for communication with the debug hardware before executing the user code. This lifecycle stage is useful to verify that your key generation and code signing process is working properly. If you find a problem, it is easy to debug, and you can erase the entire device and start over if a problem is found.

To change the LCS state from NORMAL to SECURE/SECURE\_WITH\_DEBUG, you should program the eFuse memory. Refer to [mtb-example-psoc6-security](#) template example and [AN21111 - PSoC™ 6 MCU designing a custom secured system](#) Section 3.2. for eFuse programming and LCS transition. For development purposes, it is recommended for the device to be in the NORMAL LCS with no access restrictions.



---

### 4 Device lifecycle stages (LCS)

You can read the eFuse register to know the present LCS of the device using the API listed below. The PSoC™ 6 MCU has a total of 1024 eFuse bits. Of them, one byte is reserved for the lifecycle stage starting at 43 offset. Refer to [PSoC™ 6 TRM Chapter 15: eFuse Memory](#) and [PSoC™ Peripheral Driver Library - eFuse](#)

```
/* eFuse Register Offsets */
#define LIFECYCLE_EFUSE_OFFSET      0x02B // 43 offset

/* Get lifecycle states */
Cy_EFUSE_GetEfuseByte(LIFECYCLE_EFUSE_OFFSET, &lifecycle);

printf("\r\nDevice lifecycle=0x%02x,\r\n", lifecycle);
```

## 5 Debug port access configuration (DAP)

### 5 Debug port access configuration (DAP)

There are three debug ports on the PSoC™ 62/63, CM4 CPU access port (CM4-AP), CM0+ CPU access port (CM0+\_AP), and the system access port (SYS\_AP). CM4-AP and CM0+\_AP are primarily used for debugging the individual CPU. SYS-AP does not have access to the CPU debug registers, but may access any memory or registers in the memory map. Any combination of these debug ports may be enabled or disabled.

There are three different access port restriction settings:

- Secure access restrictions (SAR)
- Normal access restrictions (NAR)
- Dead access restrictions (DAR)

NAR and SAR are used to set up debug port restrictions for NORMAL and SECURE lifecycle stages respectively. DAR are used in the SECURE lifecycle stage if there is an error during the secure boot process. This could occur if the memory has been corrupted, an incorrect public key is used, the code is signed incorrectly, or TOC2 is corrupted. SAR and DAR are stored in the eFuse, which cannot be erased once set. These restrictions must be set before entering the SECURE lifecycle stage. Once in SECURE lifecycle stage, SAR and DAR cannot be altered.

NARs are stored in the SFlash and are not recommended for a truly secure system, but can be sufficient for the user to test the access restrictions because it is stored in SFlash which can be altered unlike the eFuse which cannot be altered once written.

The table below shows the memory location for each of the three type of debug access restrictions (SECURE, DEAD, and NORMAL).

**Table 3** Debug access restrictions

Access restriction	ACCESS_RESTRICT0 (Addr)	ACCESS_RESTRICT1 (Addr)	Storage area
SECURE	0x402C_0829*	0x402C_082A*	eFuse
DEAD	0x402C_0827*	0x402C_0828*	eFuse
NORMAL	0x1600_1A00	0x1600_1A01	SFlash

The format for all three types of access restrictions (SECURE, DEAD, NORMAL) is the same, although stored in different locations.

The figure below defines the location and definition of each of those parameter in a 2-byte access restriction structure that can be configured to disable DAP and SYS AP. If the bit is set to '0', it is enabled; if '1', it is disabled. The default state of all debug access restrictions is '0', which means that all debug ports are open for full access.

ACCESS_RESTRICT0									
Bit	7	6	5	4	3	2	1	0	
Field	MMIO_Allowed [7:6] (SYS_AP)		SFlash_Allowed [5:4] (SYS_AP)		SYS_AP_MPU Enable	SYS_AP Disable	CM4 Disable	CM0 Disable	

ACCESS_RESTRICT1									
Bit	7	6	5	4	3	2	1	0	
Field	DIRECT_EXECUTE (SYS_AP)		SMIF_XIP (SYS_AP)	SRAM_ALLOWED[5:3] (SYS_AP)		FLASH_ALLOWED[2:0] (SYS_AP)			

**Figure 11** ACCESS\_RESTRICT register

---

### 5 Debug port access configuration (DAP)

For more information, refer to Chapter 9 - Debug port access settings ([AN21111 - Debug port access settings](#))

To disable and re-enable the debug ports in the NORMAL lifecycle stage for PSoC™ 61/62/63 devices, refer to AN235938 - PSoC™ 6 MCU debug restriction for normal access. It shows how to set the NAR register bits to disable the debug access port (DAP), and how to clear the NAR register bits to re-enable the DAP for debugging and reprogramming. Using NAR can prevent unwanted access to proprietary code through the debug port and also provides an option to unlock the debug port later. Infineon provides a code example to configure the NAR values to SFlash. Contact your FAE for the application note and code example.

**Note:** *In NORMAL LCS, the configurations are done in SFlash, thus giving the user flexibility to re-program the device later. However, while configuring the debug ports and lifecycle stages for SECURE LCS, you must exercise caution because the configuration is required to be programmed in eFuses. The eFuse bits are one-time programmable and once set, cannot be erased. It means that if debug ports are disabled, there is no way to enable them again.*

Refer to [mtb-example-psoc6-security](#) template example for eFuse programming and LCS transition.

## 6 Device CoT implementation

### 6 Device CoT implementation

The basis of the chain of trust (CoT) relies on memory that cannot be changed (internal ROM). This is the root of trust (RoT). After power-up, the device boots from the code in the ROM, which verifies the flash boot code. Flash boot starts the execution and verifies the CM0+ application. Once the CM0+ application is validated successfully, it starts execution and validates the CM4 application (if available) before passing control to it. Thus, forming a chain of validations and establishing security with no code executing without being signed and validated by a trusted key.

Chain of Trust is an extension of secured boot with multiple images, each validated by its preceder. The chain breaks when the validation fails at any stage. This section demonstrates the CoT using an MCUboot code example.

#### 6.1 Introduction to MCUboot

[MCUboot](#) is a library that helps to implement secured bootloader applications for 32-bit MCUs. It works by dividing the flash into two slots per image – primary and secondary. The first version of the application is programmed into the primary slot during production. A firmware update application running in the device receives the upgrade image over a wired or wireless (over-the-air or OTA) communication interface and places it in the secondary slot. This slot-based partition helps in read-/write-protecting the primary slot from a less privileged application. Typically, a bootloader application executes in secured mode and is privileged to access the primary slot while a less-privileged application such as an OTA application cannot access the primary slot, but it can access the secondary slot.

MCUboot always boots from the primary slot and copies the image from the secondary slot into the primary slot when an upgrade is requested. The upgrade can be either overwrite-based or swap-based. In an overwrite-based upgrade, the image in the primary slot is lost and there is no way to roll back if the new image has an issue. In a swap-based upgrade, the images are swapped between the two slots and a rollback is possible. In this case, MCUboot makes use of an additional area in the flash called *scratch area* for reliable swapping. MCUboot for PSoC™ 6 MCU supports both swap-based and overwrite-based upgrades. For more information refer to [MCUboot Basic Bootloader](#) Readme and [MCUboot](#) GitHub.

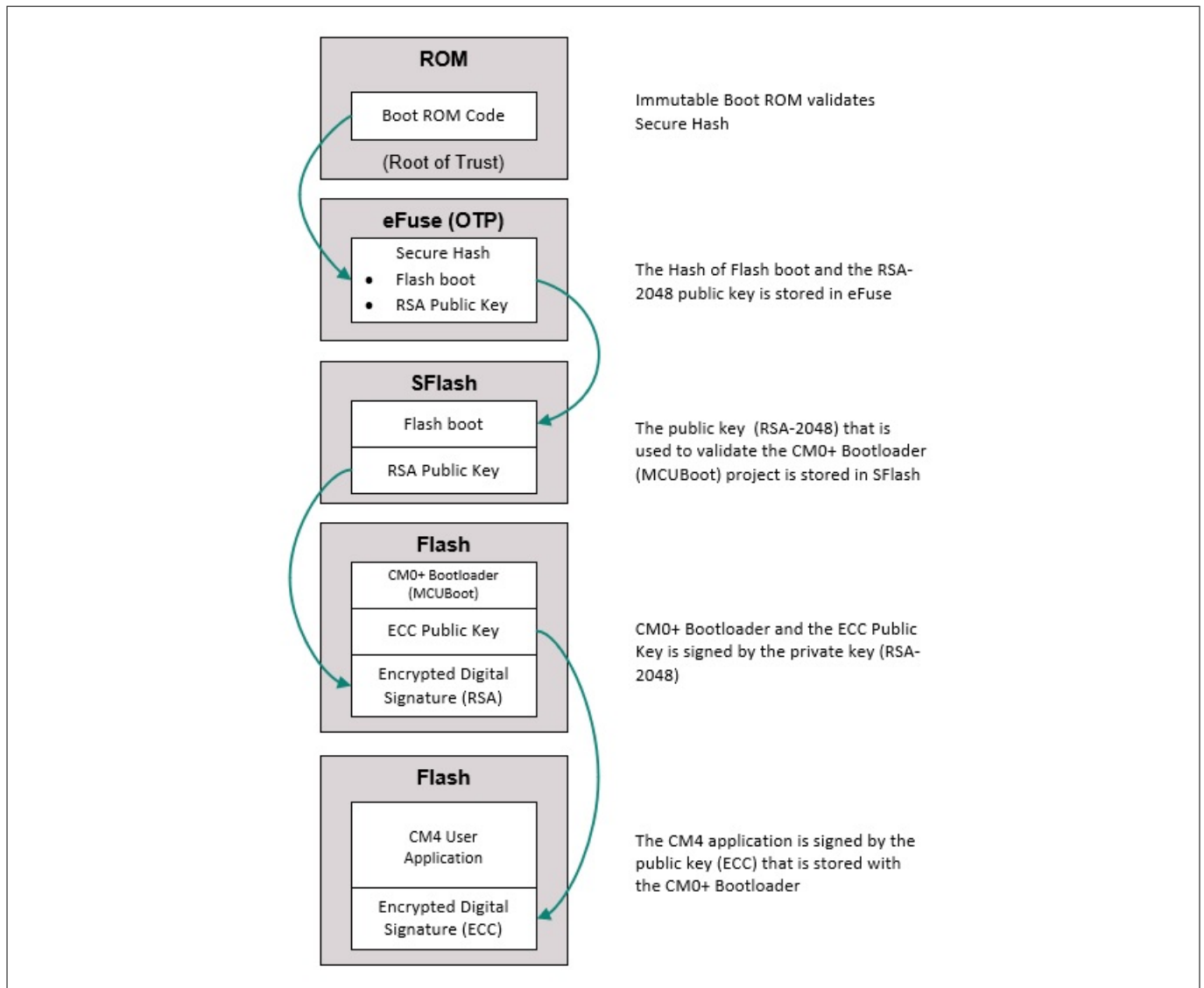
##### 6.1.1 Application structure

This chapter demonstrates the secured boot CoT functionality on an existing [MCUboot-based basic bootloader](#) application. This example bundles two projects:

- **CM0+ Bootloader app:** Implements an MCUboot-based basic bootloader application run on CM0+. The bootloader handles image authentication and upgrades. When the image is valid, the bootloader lets the CM4 CPU boot or run the image by passing the starting address of the image to it.
- **CM4 Blinky app (User app):** Implements a simple LED blinky application run by CM4. The application toggles the user LED.

The following diagram illustrates how the secured boot Chain of Trust (CoT) is implemented in this application:

## 6 Device CoT implementation



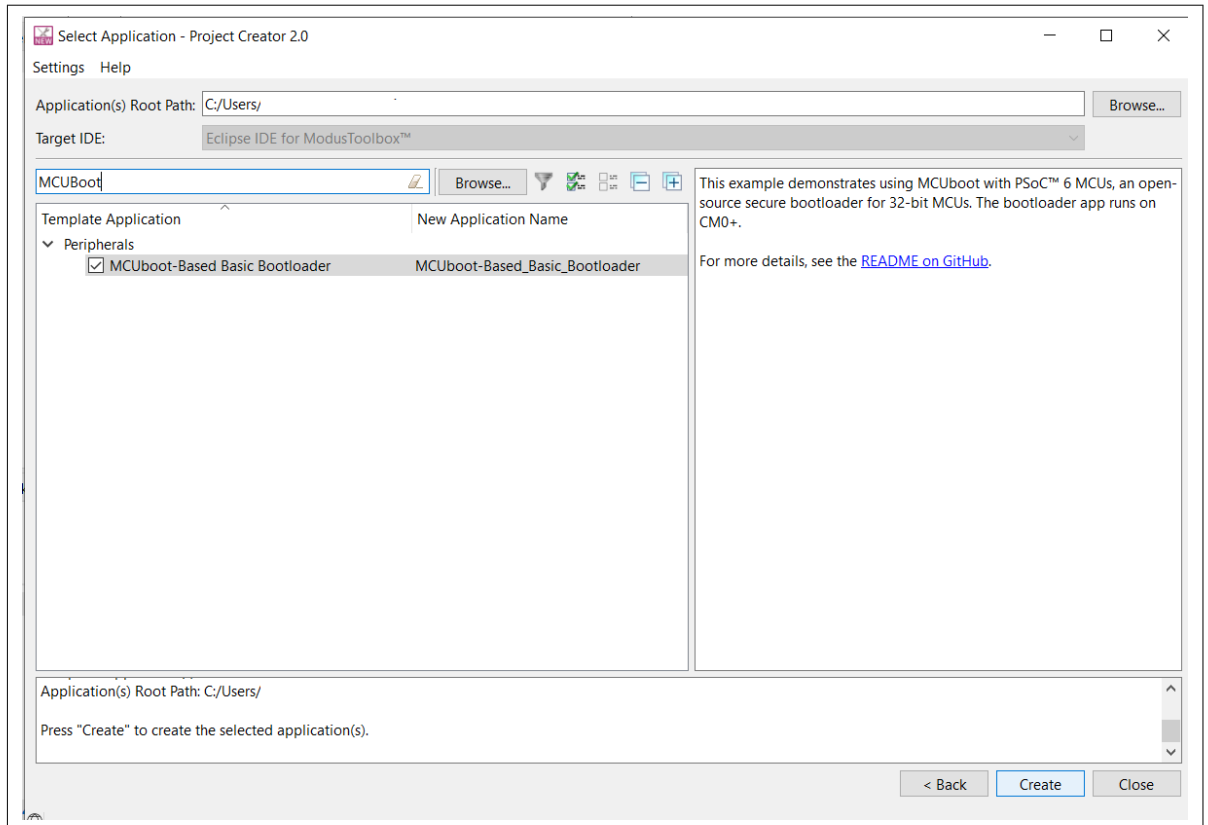
**Figure 12** PSoC™ 6 application chain of trust (CoT)

### 6.2 Enabling secured boot (with MCUboot basic bootloader CE)

Enabling the secured boot feature on an MCUboot bootloader application is similar to the Hello\_World application described in [Enabling secured boot](#). The only difference is the CM0+ MCUboot bootloader uses ECC 256 public key to validate the CM4 user application.

- Create MCUboot-basic bootloader application** in ModusToolbox™ as mentioned in [Create MTB application \(Hello World App\)](#). This application template is found under **Peripherals** as shown in the figure below:

## 6 Device CoT implementation



**Figure 13** MCUboot-basic bootloader application

2. **Add ToC2 contents** in `.\mtb-example-psoc6-mcuboot-basic\bootloader_cm0p\source\main.c` file as mentioned in [Adding TOC2](#)
3. **Add application header and signature** to `bootloader_cm0p\source\main.c` as mentioned in [Add application header and signature](#)
4. Set **CY\_BOOT\_BOOTLOADER\_SIZE** to `0x18000` in `main.c`. This is the size of the MCUboot bootloader
5. **Generate RSA key pair** in `/bootloader_cm0p/keys` as mentioned in [Generating the RSA key pair](#). This key is used for validating MCUboot by flash boot

## 6 Device CoT implementation

6. **Generate ECC keys** in /bootloader\_cm0p/keys as shown below. This key is used for validating the CM4 user application by MCUboot

Generating the ECC key pair required for the bootloader (MCUboot) is similar to the RSA key pair generation but simpler.

1. Start the modus-shell application.
2. Navigate to the mtb-psoc6-example-security/bootloader\_cm0p directory.
3. Execute the following commands in the shell.

```
> python.exe ../mtb_shared/mcuboot/v1.8.1-cypress/scripts/imgtool.py keygen -k ./keys/
cypress-test-ec-p256.pem -t ecdsa-p256
> python.exe ../mtb_shared/mcuboot/v1.8.1-cypress/scripts/imgtool.py getpub -k ./keys/
cypress-test-ec-p256.pem >> ./keys/cypress-test-ec-p256.pub
> python.exe ../mtb_shared/mcuboot/v1.8.1-cypress/scripts/imgtool.py getpub -k ./keys/
cypress-test-ec-p256.pem >> ./keys/ecc-public-key-p256.h
```

This will create the following files in the/keys folder:

- cypress-test-ec-p256.pem (private key)
- cypress-test-ec-p256.pub (public key in a C-like array)
- ecc-public-key-p256.h (public-key header file in C-like array)

The public key will automatically be incorporated into the MCUboot build, no need to edit the files

## 7. Sign the application

- a. Sign the CM0+ bootloader app (MCUBoot) . See [Signing the application](#)
- b. CM4 Blinky app signing is taken care during the build time by the imgtool Python module that is part of the post-build scripts in the Makefile of the Blinky application. Refer to [MCUboot basic CE Readme](#) for more information

**Note:** *If user wants to have two applications each executing on CM0+ and CM4, the both images can be combined into a single hex and this combined hex is signed by the imgtool. For more information on this implementation, refer to the CM4 Makefile of [mtb-example-psoc-security](#)*

## 8. Program the image

There are two ways to build and program the image. This document uses CLI to build and program because it demonstrates the step by step validation process of MCUboot and Blinky applications.

- a. **Using Eclipse IDE for ModusToolbox™ software**
  1. Select the 'bootloader\_cm0p' application in Project Explorer
  2. In the **Quick Panel**, scroll down, and click **<Application Name> Program (KitProg3)**. It programs both CM0 and CM4 applications

- b. **Using CLI (recommended)**

From the terminal, go to < application >/bootloader\_cm0p and execute the make program\_proj command to build and program the application using the default toolchain to the default target.

## 6 Device CoT implementation

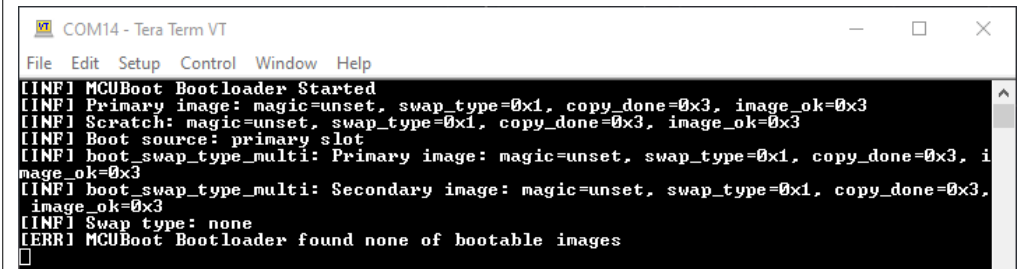
The default toolchain and target are specified in the application's Makefile, but the values can be overridden manually:

```
make program_proj TOOLCHAIN=<toolchain>
```

Example:

```
make program_proj TOOLCHAIN=GCC_ARM
```

After programming, the MCUboot bootloader starts automatically on CM0 after being validated by flash boot. Confirm that the UART terminal displays a message as shown follows:



```
COM14 - Tera Term VT
File Edit Setup Control Window Help
[INF] MCUboot Bootloader Started
[INF] Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: primary slot
[INF] boot_swap_type_multi: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, i
image_ok=0x3
[INF] boot_swap_type_multi: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3,
image_ok=0x3
[INF] Swap type: none
[ERR] MCUboot Bootloader found none of bootable images
```

**Figure 14** MCUboot with no bootable image

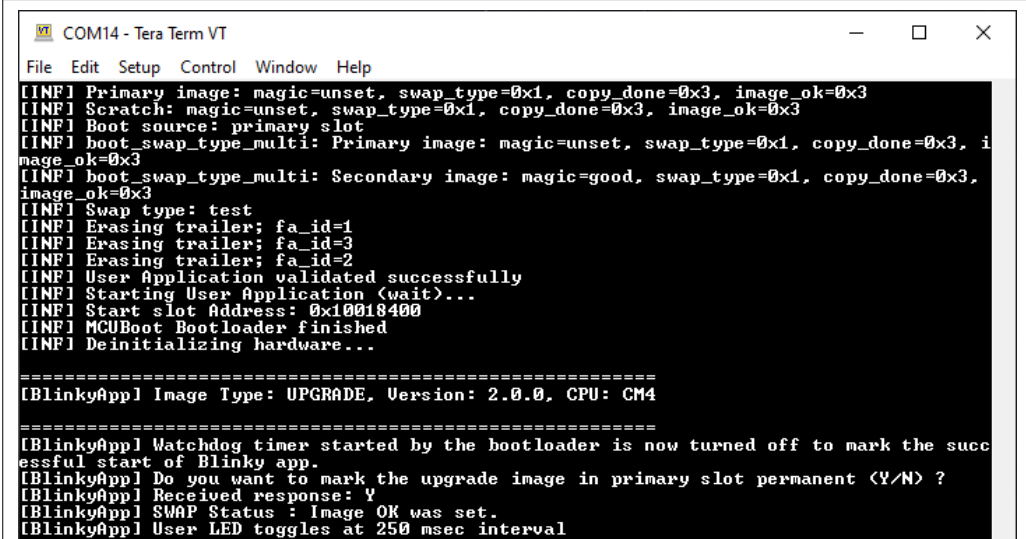
Now, build the Blinky application on CM4. From the terminal, go to the `<application>/blinky_cm4` directory and execute the `make program_proj` command to build and program the application using the default toolchain to the default target:

```
make program_proj TOOLCHAIN=<toolchain>
```

Example:

```
make program_proj TOOLCHAIN=GCC_ARM
```

After programming the Blinky app, the MCUboot bootloader verifies the Blinky application successfully and lets the CM4 core run the Blinky app. Confirm that the user LED toggles at a one second interval and the UART terminal displays a message as follows:



```
COM14 - Tera Term VT
File Edit Setup Control Window Help
[INF] Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: primary slot
[INF] boot_swap_type_multi: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, i
image_ok=0x3
[INF] boot_swap_type_multi: Secondary image: magic=good, swap_type=0x1, copy_done=0x3,
image_ok=0x3
[INF] Swap type: test
[INF] Erasing trailer; fa_id=1
[INF] Erasing trailer; fa_id=3
[INF] Erasing trailer; fa_id=2
[INF] User Application validated successfully
[INF] Starting User Application (wait)...
[INF] Start slot Address: 0x10018400
[INF] MCUboot Bootloader finished
[INF] Deinitializing hardware...

=====
[BlinkyApp] Image Type: UPGRADE, Version: 2.0.0, CPU: CM4
=====

[BlinkyApp] Watchdog timer started by the bootloader is now turned off to mark the succ
essful start of Blinky app.
[BlinkyApp] Do you want to mark the upgrade image in primary slot permanent (Y/N) ?
[BlinkyApp] Received response: Y
[BlinkyApp] SWAP Status : Image OK was set.
[BlinkyApp] User LED toggles at 250 msec interval
```

**Figure 15** MCUboot-with-blinky-in-upgrade-mode





Revision history

Revision history

Document version	Date of release	Description of changes
**	2024-03-22	<ul style="list-style-type: none"><li>Initial release</li></ul>

## Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

**Edition 2024-03-22**

**Published by**

**Infineon Technologies AG**  
**81726 Munich, Germany**

**© 2024 Infineon Technologies AG**  
**All Rights Reserved.**

**Do you have a question about any aspect of this document?**

**Email:** [erratum@infineon.com](mailto:erratum@infineon.com)

**Document reference**  
**IFX-ufr1709274718505**

## Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

## Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.