PSoC 6 MCU: Creating a Secure System

Author: Mark Hastings

Associated Part Family: PSoC® 6 MCU

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC video library here.

This document teaches you what is required to create a secure system, from the boot process all the way to your application execution.

Contents

1 Introduction

This application note teaches you how to build a secure embedded system with a Cypress PSoC 6 MCU. You will learn about the boot process and how it pertains to a secure system. It also introduces you to how PKC (Public Key Cryptography) is used to validate your application in an embedded system. An example project is provided as a template to give you a head start to creating your own secure system.

This is an advanced application note and assumes that you are familiar with the basic PSoC 6 architecture which can be found in the device datasheet or the Technical Reference Manual (TRM). This document includes the topics listed below.

- What is a secure system
- Chain of trust
- PSoC 6 MCU boot up sequence
- PSoC 6 MCU security hardware
- How Public Key Cryptography works
- How to generate a secure system

2 System Security

There are many reasons that a system must be secure: maybe manufacturers want to protect their IP to maintain a barrier to entry in that market, or to protect the end user from dangerous operation caused by a malicious attack from a third party. For whatever reason, every project team should assess its security requirements.

In recent years, we have all heard stories of thousands of Wi-Fi lightbulbs taken over by a malicious third-party or security cameras used for denial of service attacks. In the past, only PCs were smart enough to be a target for attack. In the last couple years, every new toys or pieces of electronics have more power and networking abilities than the first PCs, so it’s time to start being more vigilant when securing firmware.

There are three main ways products can be hacked.

- Direct access to the debug port. With the use of common debug dongles, accessing or reprogramming firmware or examining internal data is easy. Most common CPUs are based on just a few architectures, so hacking or reverse engineering a product is easy if the device is left unsecured.
- Direct connection to a communication port such as SPI, I^{2}C, or a UART. Depending on the firmware, this connection may allow firmware to be read or updated with non-sanctioned software. If one person decodes the protocol and posts it on the internet, you can have many hackers gaining access.
- Wireless connection such as Bluetooth or Wi-Fi. This has become the standard method of hacking because it doesn’t require physical contact. The perpetrator can be out on the street or half way around the world via the internet.

2.1 What is a Secure System

The definition of a secure system can be different depending on the application. Some systems require that all access to the device is blocked, but others just need to verify the firmware has not been corrupted. PSoC 6 MCU allows you to define the security level required for the project. There is no one perfect method because every project has different requirements. The following is a list of projects with different security goals:

- **Trusted firmware updates only with a hardware debugger:** This is usually not thought of as a secure system, but if the hardware is installed such that third parties cannot get direct access, then it may be secure. Flash write commands from firmware can be disabled so that any internal hack could not change or replace the application. The device can be put in a secure mode with the debug port open, which will force the firmware to be authenticated with a public key each time the device comes out of reset.
- **No access to debug port, support firmware updates:** The debug port provides access to all memory and a method for the device to be reprogrammed. In most cases, a real secure system requires that the debug port is disabled. Then the only way to update the firmware is for the application to provide a way to download new program data with some type of communication port such as UART, I^{2}C, or SPI. How secure this communication port must be will be up to the designer.

- **Lock down firmware, no updates:** This means that the debug port is disabled and there is no provision to allow for bootloading. This may be the most secure, but there is no way to perform bug fixes or add future enhancements.
- **Trusted firmware updates, protect IP.** To fully protect IP (Intellectual Property), the debug port must be disabled. Because the debug port is disabled, the user must provide a method to load new firmware with a bootloader. This is typically implemented with a serial port such as UART, SPI, or I^{2}C. Because the IP must be protected, the bootloader must encrypt the data transferred. PSoC 6 MCU includes a crypto block to help accelerate encryption and decryption. Security keys installed in the device at the factory can be used to authenticate the code and decrypt the data transferred during the bootload process.

2.2 Basic Definitions

Before continuing, you must understand some terms that will be used throughout this document. Many of these terms will be discussed in more detail in the following sections of this application note.

- **Chain of Trust (CoT)**: The root of trust begins with the Cypress code residing in ROM, which cannot be altered. Chain of Trust is established by validating the blocks of software starting from the root of trust located in ROM.
- **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. The System_AP can only access SRAM, Flash, and MMIOs, not the CPUs.
- **Digital Digest/Signature**: The signature generated by an SHA-256 function that operates on a block of data.
- **eFuse**: One Time Programmable (OTP) memory that by default is 0 and can be changed only from 0 to 1. The eFuse bits can be programmed individually, but cannot be erased.
- **Flash (Boot)**: This is part of the boot system that performs two basic tasks:
  - Sets up the debug port based on the lifecycle stage.
  - Validates the user application before executing it.
- **Flash (User)**: This is the flash memory that is used to store your application code.
- **Hash**: A crypto algorithm that generates a repeatable but unique signature for a given block of data. This function is non-reversible.
- **IP**: Intellectual Property. This can be both code and data stored in a device.
- **IPC**: Inter-Processor Communication hardware used to facilitate communication between the two CPU cores.
- **Lifecycle**: This is the security mode in which the device is operating. To the user, it has only two states of interest: Normal and Secure.
- **Memory Protection Unit (MPU)**: Used to isolate memory sections from different bus masters.
- **MMIO:** Memory-Mapped Input/Output, usually refers to registers that control the hardware I/O.
- **PC**: Protection Context. Although PC most often refers to a Program Counter, in this document, it refers to the Protection Context state.
- **Peripheral Protection Unit (PPU)**: PPUs are used to restrict access to a peripheral or set of peripherals to only one or a specific set of bus masters.
- **Protection State**: This is the current state of the Debug Access Port. There are three possible states, Normal, Secure, and Dead. Each of these states may be configured by the user. The Normal protection state is stored in SFlash, but Secure and Dead state configurations are stored in the one-time programmable eFuse.
- **Protection Units**: These hardware blocks are used to limit bus master access to memory (SRAM, ROM, flash) or hardware (peripheral) registers. They include MPU, PPU, and Shared Memory Protection Unit (SMPU).
- **Public-Key Cryptography (PKC)**: Otherwise 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 communication. It is used to decode 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.
  - **Public Key**: The public key can be shared, but it should be authenticated or secured so it cannot be modified.
  - **Private Key**: The private key must be kept in a secure location so it cannot be viewed or stolen. It is used to encrypt a block of data that will be decrypted using an associated public key.
- **RMA:** Return Merchandise Authorization

- **ROM**: Read Only Memory that is programmed as part of the fabrication process and cannot be reprogrammed.
- **RSA-nnnn**: 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. The encryption/decryption is controlled by a key that is commonly 1024, 2048, or 4096 bits in length (RSA-1024, RSA-2048, or RSA-4096).
- **Secure Image**: An application template that is used to set up the security features of PSoC 6 MCU. It can be modified by the programmer to implement a specific security policy.
- **Security Policy**: This is the set of rules that the designer imposes to determine what resources are protected from outside tampering or between the internal CPUs.
- **Serial Memory Interface (SMIF)**: A SPI (Serial Peripheral Interface) communication interface to serial memory devices, including NOR Flash, SRAM, and non-volatile SRAM.
- **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 a secure mode, it can no longer be changed.
- **SHA-256**: A 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.
- **Shared Memory Protection Unit (SMPU)**: SMPUs are used to allow access to a specific memory space (flash, SRAM, or registers) to only one or a specific set of bus masters.
- **System Calls**: Functions such as flash write functions that are executed by the Arm® Cortex® M0+ CPU (CM0+) from ROM.
- **Table of Contents 1**: (TOC1) This is an area in SFlash that is used to store pointers to the trim values, flash boot entry points, etc. It is used only by boot code in ROM and is not editable by the designer.
- **Table of Contents 2**: (TOC2) This is an area in SFlash that is used to store pointers to two applications blocks: User Application1 and User Application2. It also contains some boot parameters that are settable by the system designer. There is also a duplicate of the TOC2 that is written in the adjacent page of flash. This duplicate is referenced as RTOC2. If the TOC2 is found to be invalid for any reason, the RTOC2 is evaluated.

2.3 Security Policy

This application note discusses how to make sure that the system executes code only from a trusted source. If the system's firmware is never updated after it leaves the factory, this can be a simple solution. But if the firmware may need to be updated in the field, this becomes more complicated. The designer must decide on a security policy for the system. This requires several questions to be answered before the overall design is started.

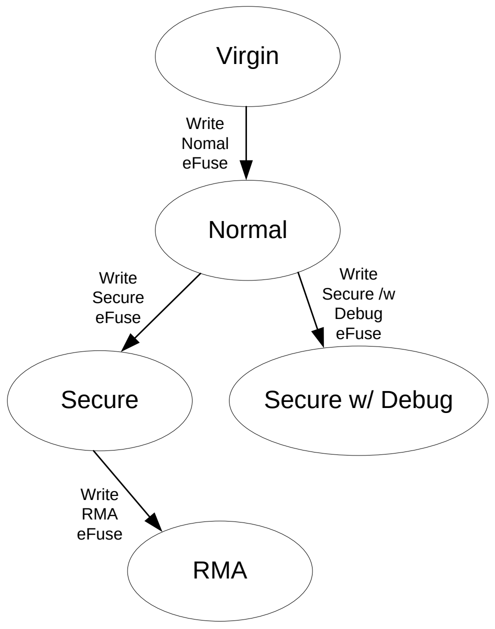
- Will firmware be upgradable after the device leaves the factory?
- If firmware is upgradable, will a bootloader be needed?
- Which features will be implemented on the Arm® Cortex® M4 CPU (CM4) and the CM0+?
- What will the SRAM and flash partition look like between CM4 and CM0+ CPUs?
- What memory and hardware must be isolated between CPUs and/or tasks?

Answering these questions helps to determine the system's security policy. This application note does not discuss bootloaders or the security requirements of a secure bootloader.

2.4 Device Lifecycle

To help understand how PSoC 6 MCUs provide security, the device lifecycle must be understood. The device lifecycle is a key aspect of the PSoC 6 MCU's security. Lifecycle stages follow a strict irreversible progression dictated by blowing eFuses (changing a fuse's value from '0' to '1'). This system is used to protect the internal device data and code at the level required by the customer. Lifecycle stages are governed by the LIFECYCLE_STAGE eFuses. PSoC 6 MCUs support the lifecycle stages shown in Figure 1.

Figure 1. Device Lifecycle



2.4.1 Virgin

This stage is used by Cypress during assembly and testing. During this cycle, trim values and flash boot are written into SFlash. Parts in this cycle never leave the factory. In this stage, the boot ROM assumes that no other eFuse data or flash data is valid. Devices are transitioned to the Normal stage before they leave the factory.

2.4.2 Normal

This is the lifecycle stage of a device after trimming and testing is complete in the factory. All configuration and trimming information is complete. Valid flash boot code has been programmed in the SFlash. To allow the OEM to check the data integrity of trims, flash boot, and other objects from the factory, a hash (SHA-256 truncated to 128 bits) of these objects is stored in eFuse. This hash is referred to as Factory_HASH. In Normal mode by default, there is full debug access to the CPUs and System. The device may be erased and programmed as needed.

2.4.3 Secure

This is the lifecycle stage of a secure device. Prior to transitioning to this stage, the Secure_HASH must have been blown in eFuse and valid application code must have been programmed in the main flash. In this stage, the protection state is set to Secure and Secure DAP access restrictions are deployed. A Secure device will boot only when authentication of its flash boot and application code succeeds.

After an MCU is in the Secure life cycle stage, there is no going back. The debug ports may be disabled depending on your preferences, which means that there is no way to reprogram or erase the device with a hardware programmer/debugger. The only way to update the firmware at this point is to provide a bootloader as part of device firmware and provide a way to invoke it.

Access restrictions in the SECURE state can be controlled by writing to eFuse's SECURE_ACESS_RESTRICT0 and SECURE_ACESS_RESTRICT1.

Code should be tested in Normal and Secure with Debug stages before the move to Secure mode. This is to prevent a configuration error that could cause the part to be no longer accessible for device programming and therefore unusable.

2.4.4 Secure with Debug

This is the same as Secure lifecycle stage, except with Normal access restrictions applied to enable debugging. Parts put in the Secure with Debug mode cannot be changed to either Secure or back to Normal mode. After Parts in this mode will most likely be discarded after testing. Devices should not be shipped in this mode since it is not secure.

2.4.5 RMA Mode

The customer transitions the part to RMA lifecycle stage (from Secure) when the customer wants Cypress to perform failure analysis on the part. The customer erases all their sensitive data (may include firmware) prior to invoking the system call that transitions the part to RMA. After the part has been transitioned to RMA, a certificate that is signed by the customer using his private key must be provided for the part to be evaluated.

When invoking the system call to transition to RMA, the customer must provide a certificate that authorizes Cypress to transition the part with a specific Unique ID to RMA lifecycle stage. The certificate will be signed by the customer using the same private key that is used in signing the user application image. The verification of the signature uses exactly the same algorithm used by flash boot in authenticating the user application. The same public key (injected by the OEM) stored in SFlash is used for the verification.

When a part is reset in the RMA lifecycle stage, the boot will set the access restrictions such that the DAP has access only to System AP, IPC MMIO registers for making system calls, and adequate RAM for communication. It will then wait for the system call “Open for RMA” from the DAP along with the certificate of authorization. This is the same certificate used in transitioning the device to RMA. The boot process will not initiate any firmware until it successfully executes the “Open for RMA” command. After the command is successful, the device will behave as though it were in the Virgin lifecycle stage, but it cannot be transitioned to Virgin or any other lifecycle stage. The lifecycle stage stored in eFuse cannot be changed from RMA. Every time the part is reset, it must execute the “Open for RMA” command successfully before the part can be used. See Appendix C, Transition to RMA for more details on supporting transition to RMA.

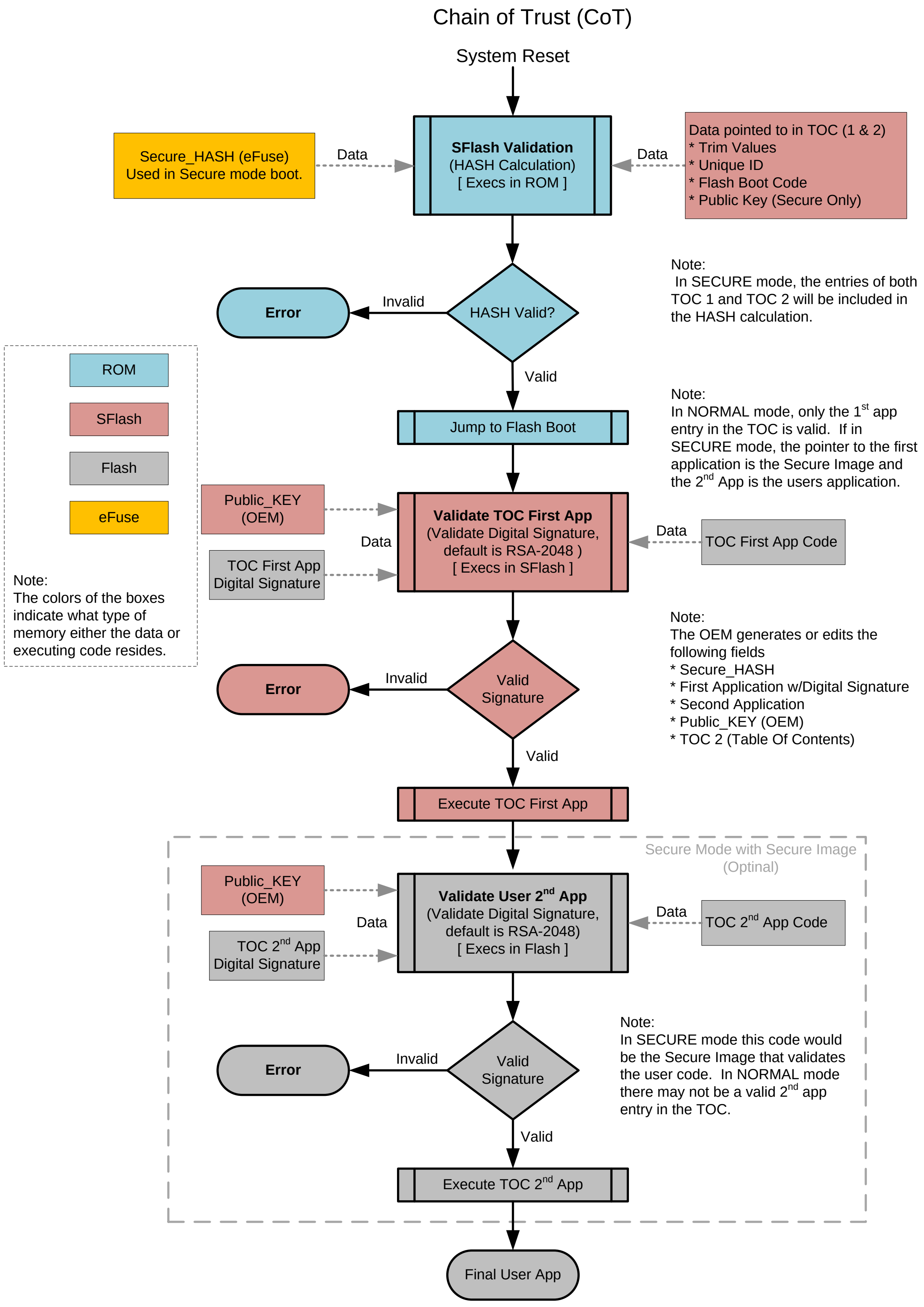
3 Boot Sequence and Chain of Trust

The CoT is inherently part of the boot sequence. It begins at the root of trust, which is the initial boot code stored in ROM, and which cannot be changed by users or Cypress.

3.1 Boot Sequence

The boot sequence and the validation sequence are, for the most part, one and the same. Figure 2 shows how the CM0+ operation starts from reset. After reset, the CM0+ starts executing from ROM. It first configures some basic hardware then validates SFlash so that it can read data required to trim (calibrate) some of the aspects of the memory and clocking systems. After trim/calibration is complete, execution jumps to Flash Boot and configures the Debug Access Port (DAP) as needed by the lifecycle stage. Notice the color coding that depicts the memory type where the data and code resides.

Figure 2. PSoC 6 MCU Boot with CoT


Chain of Trust (CoT)
System Reset
Secure_HASH (eFuse)
Used in Secure mode boot.
Data
SFlash Validation
(HASH Calculation)
[Execs in ROM]
Data
Data pointed to in TOC (1 & 2)
* Trim Values
* Unique ID
* Flash Boot Code
* Public Key (Secure Only)
Error
Invalid
HASH Valid?
Note:
In SECURE mode, the entries of both TOC 1 and TOC 2 will be included in the HASH calculation.
Valid
ROM
SFlash
Flash
eFuse
Note:
The colors of the boxes indicate what type of memory either the data or executing code resides.
Jump to Flash Boot
Note:
In NORMAL mode, only the 1st app entry in the TOC is valid. If in SECURE mode, the pointer to the first application is the Secure Image and the 2nd App is the users application.
Public_KEY (OEM)
TOC First App Digital Signature
Data
Validate TOC First App
(Validate Digital Signature, default is RSA-2048)
[Execs in SFlash]
Data
TOC First App Code
Error
Invalid
Valid Signature
Note:
The OEM generates or edits the following fields
* Secure_HASH
* First Application w/Digital Signature
* Second Application
* Public_KEY (OEM)
* TOC 2 (Table Of Contents)
Valid
Execute TOC First App
Secure Mode with Secure Image (Optinal)
Public_KEY (OEM)
TOC 2nd App Digital Signature
Data
Validate User 2nd App
(Validate Digital Signature, default is RSA-2048)
[Execs in Flash]
Data
TOC 2nd App Code
Error
Invalid
Valid Signature
Note:
In SECURE mode this code would be the Secure Image that validates the user code. In NORMAL mode there may not be a valid 2nd app entry in the TOC.
Valid
Execute TOC 2nd App
Final User App

Flash Boot then validates the first application listed in the Table of Contents (TOC2) and jumps to its entry point if validated. In a non-secure system, there may be only one user application listed in the TOC2, which is the user application. In the secure system defined in this application note, the first user application is the Secure Image. After the Secure Image configures the hardware to secure the system, it will point to User Application 2 from the TOC2. This second user application is the main user application. The Secure Image validates the user application and jumps to the entry point if the application is validated.

If either application (secure image or user) are found to be invalid or corrupted, the CPU will jump to a secure point and stay in an idle loop until the device is reset.

3.2 Chain of Trust (CoT)

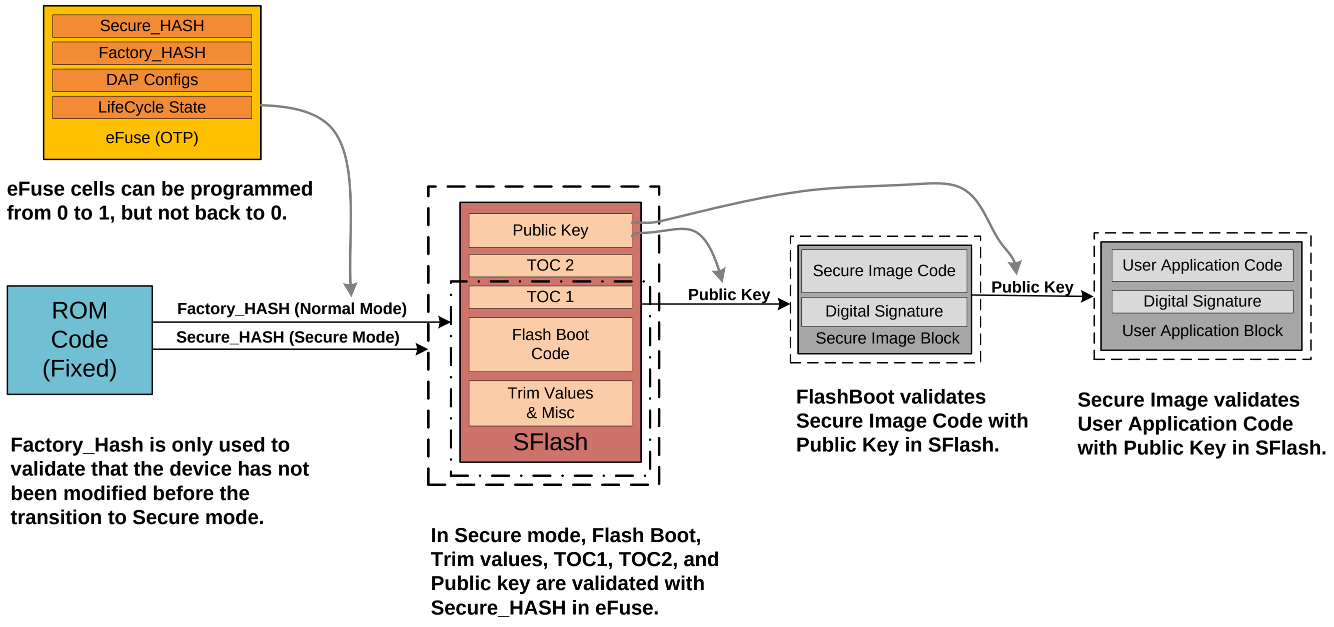
The basis of the chain of trust relies on memory that cannot be changed, such as ROM. The rest of the chain is dependent on this fact.

Flash boot, trim constants and the Table of Contents1 (TOC1) are located in SFlash (Supervisory Flash) which is restricted from being reprogrammed in either Normal or Secure modes. This area is validated with a Factory_HASH value stored in eFuse. The Factory_HASH code is not used during the boot sequence in Normal or Secure mode. The Factory_HASH ensures that the device can be validated prior to the transition to Secure mode. This ensures that the flash boot code, trim values, and the TOC1 has not been tampered with after the MCU has left Cypress. If the Factory_HASH has been corrupted, Cypress should be contacted immediately.

PSoC Programmer will automatically validate the Factory_HASH before writing the Secure_HASH prior to switching to Secure mode. Two sections of SFlash can be reprogrammed in Normal mode: Table of Contents 2 (TOC2) and an area reserved for a user public key.

After the transition from Normal to Secure mode, all the blocks in the SFlash, including the Public Key area and the TOC2, are validated with another hash value referred to as the Secure_HASH. It is also stored in eFuse and cannot be changed without detection. The entire SFlash block is validated with the Secure_HASH each time the device wakes from reset in the Secure mode. If an error is found while validating the SFlash, the device will no longer complete the boot sequence and enter a Dead state. Figure 3 shows the CoT from the perspective of data and code validation.

Figure 3. Basic Chain of Trust


Secure_HASH
Factory_HASH
DAP Configs
LifeCycle State
eFuse (OTP)
eFuse cells can be programmed from 0 to 1, but not back to 0.
ROM Code (Fixed)
Factory_HASH (Normal Mode)
Secure_HASH (Secure Mode)
Factory_Hash is only used to validate that the device has not been modified before the transition to Secure mode.
Public Key
TOC 2
TOC 1
Flash Boot Code
Trim Values & Misc
SFlash
In Secure mode, Flash Boot, Trim values, TOC1, TOC2, and Public key are validated with Secure_HASH in eFuse.
Public Key
Secure Image Code
Digital Signature
Secure Image Block
FlashBoot validates Secure Image Code with Public Key in SFlash.
Public Key
User Application Code
Digital Signature
User Application Block
Secure Image validates User Application Code with Public Key in SFlash.

At this point, the entire SFlash is now trusted because its validation is based on memory (eFuse) that cannot be modified without detection during SFlash validation in ROM.

The public key, which is locked into the SFlash, is secure and cannot be changed without being detected as well. It is used by flash boot to validate the next step in the boot process. Flash boot validates the code in the Secure Image block, which includes a digital signature at the end of the code block. Flash boot uses the SHA-256 hash function to calculate the digital signature of the Secure Image Block. The digital signature attached to the Secure Image block is encrypted using a private key that is associated with the Public Key in SFlash using RSA 2048-bit encryption. More details on how this is done are discussed in the next section. The calculated and the stored encrypted digital signatures are then checked to see whether they match. If they match, the Secure Image Block has been validated. The same process is used by the Secure Image to validate the User Application Block.

4 Code Signing and Validation

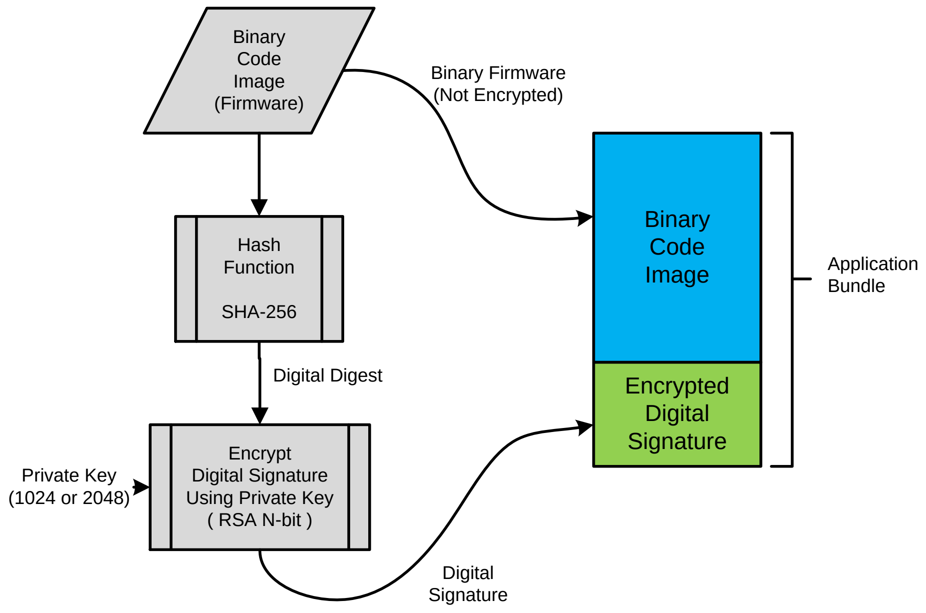
In the CoT section above, code validation was mentioned but with not much detail. In this section, we will dive into the process of signing a block of code so that it can be validated during runtime.

The encryption method used is PKC (Public Key Cryptography) that uses a private and public key. Care must be taken to keep the private key at a secure location, so that it never gets into the public domain. If the private key is exposed, it will endanger your system's security. Companies must create a method in which very limited access to the private key is allowed. The public key, on the other hand, can be viewed by anyone. The only requirement is that the public key must be validated or locked in such a way that it can't be changed, or so that any modification to the public key can be detected. In this example, the public key is stored in SFlash and validated with the Secure_HASH key as defined in the CoT section above. These private and public keys can be generated with common encryption libraries such as OpenSSL. More details of generating and using the private and public keys is discussed in Appendix A.

4.1 Code Signing

To validate code, such as the user applications during runtime, a digital signature must be created and bundled with the code during build time. The code itself is not stored in flash in an encrypted format but the digital signature is encrypted. The digital signature is generated with the SHA-256 hash function, then encrypted using a private key with RSA-2048-bit encryption. The reason the digital signature is encrypted is so that a third party, without access to the private key, cannot create a valid code/signature bundle. See Figure 4.

Figure 4. Generation of Encrypted Digital Signature


Binary Code Image (Firmware)
Binary Firmware (Not Encrypted)
Hash Function SHA-256
Digital Digest
Private Key (1024 or 2048)
Encrypt Digital Signature Using Private Key (RSA N-bit)
Digital Signature
Binary Code Image
Encrypted Digital Signature
Application Bundle

4.2 Code Validation

A secure system must be able to detect code that was not created by the original manufacturer or trusted source. If non-trusted code is detected, execution must take a known path to a safe state. This also validates that the firmware was not corrupted intentionally or accidently.

4.2.1 Parts of Code Validation

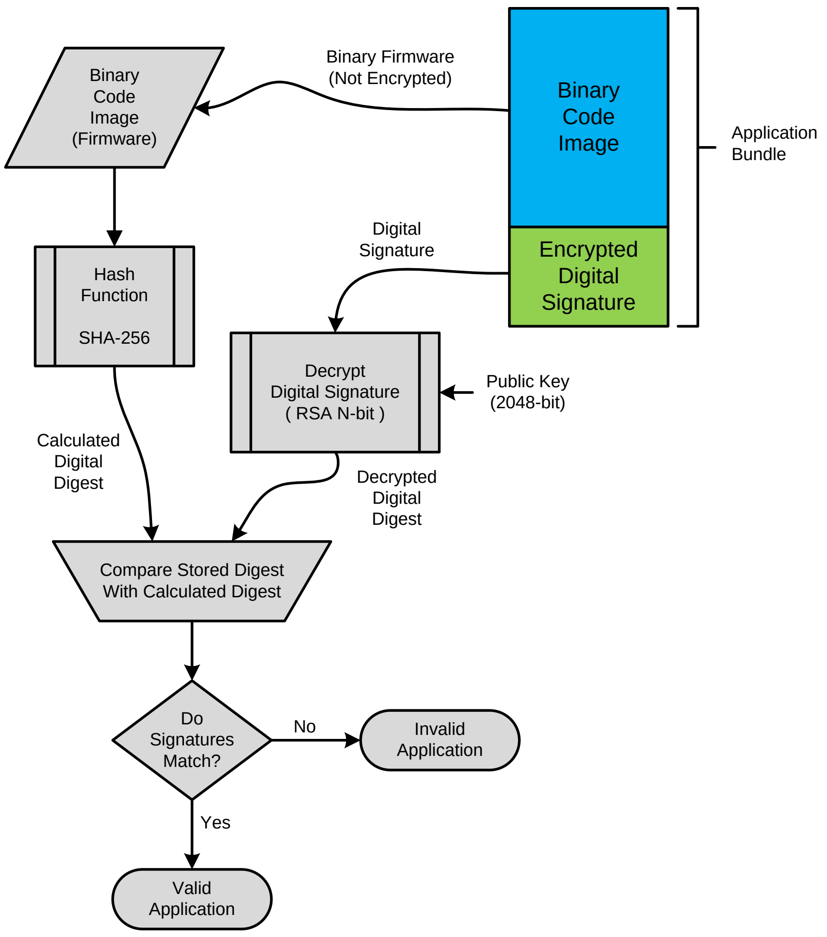
Validation requires three pieces: data, signature, and crypto key. The data and the signature come as a pair; the key is stored in a location that cannot be changed.

- **Data:** This includes both executable code and constants that makes up the firmware in an embedded system. This data usually resides in flash memory that usually can be modified at one time or another. Depending on the system, you may modify or update it. Therefore, you must be able to determine whether this data is from a known source and has not been corrupted either by accident or by a malicious event.
- **Digital Signature:** A digital signature is the hash of a block of data. The hash algorithm used in this case is SHA-256. The digital signature alone can be used to validate that the data is intact. By encrypting the digital signature with a crypto key, you can determine whether the data is from a trusted source, as well as intact.
- **Crypto Key:** This can be either a public or private key. In the system described in this application note, the public key is stored on the device and the private key is secured by the developer. The public key must be secured in one of two ways:
  - A method to validate the source of the key. This can be accomplished with some type of communication with a known source or server. This is not practical for devices that cannot easily communicate with a known server when required.
    The other and most likely option for an embedded system, is to have the key data itself validated by using a key that is stored in memory that cannot be modified. In Cypress PSoC 6 MCUs, a hash is calculated from the areas containing the public key, flash boot code, and trim values. This hash is then stored in one-time programmable eFuse and referred to as Secure_HASH.

4.2.2 Code Validation Used by PSoC 6 MCUs

A user application binary code block (Application Bundle) includes an encrypted digital signature that was created during build time. The Secure Image application also uses this format. To validate the block of code, a hash function (SHA-256) is applied to the binary code image, which creates a calculated digital signature, or digital digest. Next, the Encrypted Digital Signature is decrypted using the stored public key, to reveal the decrypted digital signature. The calculated digital digest and the decrypted digital digest (signature) are then checked for an exact match. If they are an exact match, the code is validated. See Figure 5.

Figure 5. Application Validation


Binary Code Image (Firmware)
Binary Firmware (Not Encrypted)
Binary Code Image
Application Bundle
Digital Signature
Encrypted Digital Signature
Hash Function SHA-256
Decrypt Digital Signature (RSA N-bit)
Public Key (2048-bit)
Calculated Digital Digest
Decrypted Digital Digest
Compare Stored Digest With Calculated Digest
Do Signatures Match?
No
Invalid Application
Yes
Valid Application

5 Resource Protection

Resource protection means that during runtime, only the bus master or task that should have access to a memory space or register space can access it. This can be a combination of read, write, or execute. PSoC 6 MCUs have blocks called protection units to add this functionality. These protection units can be configured to create multiple protection zones that include flash, SRAM, peripherals, and I/Os. These zones can then be restricted by CPU, tasks, or both. There are three main types of protection units: SMPU, MPU, and PPU.

Each CPU and the crypto block has its own Memory Protection Unit (MPU). The MPU is different from the protection structure of the other protection units. MPU protection structures cater only to protection attributes pertaining to a single master. MPU protection structures do not have a Protection Context parameter associated with them. Protection attributes for an MPU are user/privilege, read/write/execute, and secure/non-secure. MPUs are specific for each bus master and provide resource protection from its various threads or tasks. MPUs are most likely to be used with a real-time operating system (RTOS) and modified as the task execution changes.

Shared Memory Protection Units (SMPUs) protect the memory regions that are used by multiple masters. These SMPUs have all the attributes of the MPU and the protection context attribute.

The Secure Image uses SMPUs to restrict access to secure sections of the memory from the non-secure application. Registers used for flash write operations are restricted so that only the SROM code may access those operations. This eliminates any accidental writing or erasure of the flash memory. Also, SRAM and registers used for Crypto operations are protected to keep operations secure.

Peripheral Protection Units (PPUs) are designed specifically to protect peripheral registers. A PPU is similar to the SMPU. Some of the PPUs are fixed-function because they are hardwired to protect a specific peripheral memory region. Therefore, their address and size parameters in the protection structure are not configurable, but the protection attributes can be configured.

The Secure Image template sets up several SMPU and PPU structures by default. You can leave these as provided or add or remove as needed.

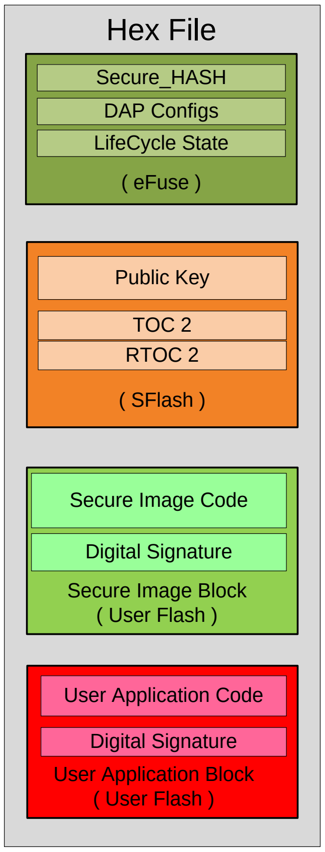
For more information on protection units and how to use them, see Appendix D, Protection Units and the Peripheral Driver Library manual. The manual can be found under Cypress Peripheral Driver Library/API References/Protection Unit.

6 Building a Secure System

As you see, building a fully secure system with a CoT is more complicated than generating a simple application. Instead of just the user code, the hex file now must contain several other pieces normally not required in a simple (non-secure) system. The following are the memory sections that will need to be programmed when creating a secure system:

- Secure_HASH (eFuse)
- DAP Configs (eFuse)
- Lifecycle Stage (eFuse)
- Public Key (SFlash)
- TOC2 and RTOC2 (SFlash)
- Secure Image Block (user flash)
- User Application Block (user flash)

Figure 6. Secure System Hex File Configuration



This section will take you through each of the pieces of a secure system and describe how to generate them. A template that contains a workspace with two projects “secure_image” and “user_app0”, is provided to simplify the process. The “user_app0” project in this workspace may be replaced with your project. This template can be found at *C:/Program Files (x86)/Cypress/PDL/3.0.1/security*. The first step is to make a copy of the security folder and place it in your work directory, because you may be editing some of the files. The example workspace is in */security /secure_image/example/Secure_Image_Example_workspace.cywrk*.

The building of the TOC2 (Table of Contents 2) block is dependent on both the User Application Block (user_app0) and the Secure Image Block (secure_image). Both projects must be built before TOC2 can be configured completely. The Secure Image Block does not need to know about the User Application Block. Instead, it refers to TOC2 to find the address of the starting location of the CM0+ project in the User Application Block. Note that the Secure Image and user projects can still be developed independently. The two projects are very independent of each other and as long as TOC2 is filled in (programmed) at the end, it will work properly.

We will take a step-by-step approach to generate each project.

6.1 Secure Image Project (secure_image)

This section covers the functionality of each file and the modifications that you are expected to make to the Secure Image project template. This assumes that the template described in the section above has been copied to a user directory.

The Secure Image project is packaged in the Cypress Standard User Application Format. This project contains only code for CM0+. This is because CM4 is not enabled during the Secure Image phase. Its main purpose is to set up the System Call filtering and configure several protection units to isolate secure operations from CM4 and CM0+ user code, stack space, and SRAM. This ensures that non-secure code cannot accidently or purposely get direct access to sensitive operations such as flash writes and crypto functions without going through approved library functions.

Normally, System Calls are handled directly with the embedded ROM code by CM0+, through an inter-processor communication (IPC) channel. The Secure Image instead intercepts the interrupt, and filters the flash write routines to make sure that only the flash that should be written, can be written. A similar operation is done with the crypto library functions as well.

The Secure Image project is broken into several files to make it easy to maintain. Table 1 shows the files that are important to the user.

Table 1. Main Source Files in Secure Image Project

File	Description
cy_si_main.c	Contains the application main(), ISR handler for System Calls, and ISR handler for Crypto.
cy_si_config.c	This file contains the memory blacklist setup, protection unit configuration, application validation, Syscall and Crypto interrupt configuration, and memory range checking. The memory blacklist and range checking are to limit the memory that can be written to or accessed by the Crypto function.
cy_si_keystorage.c	Holds the public key that resides in SFlash and an empty array that provides a place to add additional secure keys into. You are responsible for making the additional keys secure.

6.1.1 cy_si_main.c

This file contains the main() function and interrupt handlers for the crypto function call (`Cy_SI_CryptoTask`) and System Calls (`Cy_SI_SysCallTask`) handlers. Both System Calls, such as Flash Write APIs, and the crypto library calls interface between the two CPUs with an IPC channel. Most of the source code in this file **will not need to be modified**. The only cases where you may want to make any change would be to disable a system or crypto call or to add a custom system or crypto function call.

The *cy_si_main.c* file contains a couple of structures that are very important in generating the proper hex file, but need little or no modification by the designer. These structures are used to generate the application header for any project and a Table Of Contents (TOC2) that flash boot needs to read to know where the Secure Image is located in memory. It also contains the pointer to the user application so that the Secure Image can validate it and then execute it.

6.1.1.1 Table of Contents2 (Optional Reading)

TOC2 is used to point to the location of the first and second executable applications. For a Secure System, the Secure Image application (CM0+) is the first application; the second is your application. TOC2 has a redundant copy, with 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 if in Normal mode, or in a Dead state if in Secure mode. See Table 2 for the elements of TOC2. These structures are placed here to make sure they are populated properly and are written into the device when the rest of the device is programmed.

There are two parameters that you may want to change for a faster boot sequence, at offset 0x1F8 (See Table 2 below): the boot clock frequency parameter “IMO/FLL clock frequency” and the debug parameter “Wait Window Time”. You may change these in TOC2 to the values defined in the table.

Table 2. Elements of Table of Contents2

Offset	Purpose
0x00	Size in bytes used to calculate the CRC. This should be multiples of 4 and start at offset 0x00. CRC is always at the end of the table.
0x04	Magic number used to validate TOC2 (0x01211220)
0x08	Address of additional Key Storage Flash Blocks. Note that this also marks the top of the flash available to the user. This key storage area is in the main flash and is accessed with the Secure Image API. This is an optional area, and should be set to zero if not used.
0x0C	Address of SMIF Configuration Table. This points to a null terminated array of SMIF structures. This is not a Secure Image feature. Placing security details in external memory through SMIF is strongly discouraged.
0x10	Address of the first user application object
0x14	Object format of the first application (four bytes). 0 means Basic; 1 means Cypress Standard; and 2 means Simplified
0x18	Address of the second user application object (0s if none)
0x1C	Object format of the second application (four bytes).
0x20	Number of additional objects (in addition to objects covered by Factory_HASH) starting from offset 0x24 to be verified for Secure_HASH
0x24	Address of the signature verification key (0 if none). The object is specific to the signature scheme. If it is an RSA signature, it is the public key.
0x28-0x1F4	32-bit pointer (uint32_t) to additional objects. The first four bytes of any object must be the length of that object including the length bytes. This space should be all zeros if not used.
0x1F8	Flash boot parameters: Bits [1:0] flag indicate the CM0+ CPU clock frequency configuration. The clock will remain at this setting after flash boot execution. Default frequency is 25 MHz. Value [1:0] — IMO/FLL clock frequency 0x00 — 25 MHz (FLL) (Default) 0x01 — 8 MHz (IMO) 0x02 — 50 MHz (FFL) 0x03 — Reserved Bits [4:2] Flags to determine the wait window to allow sufficient time to acquire the debug port. Value [4:2] — Wait Window Time 0x00 — 20 ms (Default) 0x01 — 10 ms 0x02 — 1 ms 0x03 — 0 ms (No wait window) 0x04 — 100 ms 0x05-0x07 — Reserved Bit [31] Determines whether application #1 should be validated when in Normal mode. Value [31] — 0 – Do not validate App#1 in Normal mode. 1 – Validate App#1 in Normal mode.
0x1FC	CRC16-CCITT (the upper two bytes contain the CRC and the lower two bytes are 0) CRC calculation must be based on CRC16-CCITT with the CRC poly=0x1021 and initial value=0xFFF

```
/** Flash Boot parameters */
    #define CY_SI_TOC_FLAGS ((CY_SI_FLASHBOOT_VALIDATE_YES << CY_SI_TOC_FLAGS_APP_VERIFY_POS) \
                           | (CY_SI_FLASHBOOT_WAIT_20MS << CY_SI_TOC_FLAGS_DELAY_POS) \
                           | (CY_SI_FLASHBOOT_CLK_25MHZ << CY_SI_TOC_FLAGS_CLOCKS_POS))
/** TOC2 in SFlash */
    CY_SECTION(".cy_toc_part2") __USED static const cy_stc_si_toc_t cy_toc2 =
    {
        .objSize     = sizeof(cy_stc_si_toc_t) - sizeof(uint32_t),  /**< Object Size (Bytes) excluding CRC */
        .magicNum    = CY_SI_TOC2_MAGICNUMBER,                      /**< TOC2 ID (magic number) */
        .userKeyAddr = (uint32_t)&CySecureKeyStorage,               /**< User key storage address */
        .smifCfgAddr = 0UL,                                         /**< SMIF config list pointer */
        .appAddr1    = CY_SI_SECURE_FLASH_BEGIN,                    /**< App1 (Secure Image) start address */
        .appFormat1  = CY_SI_APP_FORMAT_CYPRESS,                    /**< App1 Format */
        .appAddr2    = CY_SI_USERAPP_FLASH_BEGIN,                   /**< App2 (user application image) start address */
        .appFormat2  = CY_SI_APP_FORMAT_CYPRESS,                    /**< App2 Format */
        .shashObj    = 1UL,                                         /**< Number of additional objects to add to the
SECURE_CMAC */
        .sigKeyAddr  = (uint32_t)&SFLASH->PUBLIC_KEY,               /**< Address of signature verification key */
        .tocFlags    = CY_SI_TOC_FLAGS,                             /**< Flash Boot flags stored in TOC */
        .crc         = 0UL                                          /**< CRC populated by cymcueltool */
    };

    /** RTOC2 in SFlash */
    CY_SECTION(".cy_rtoc_part2") __USED static const cy_stc_si_toc_t cy_rtoc2 =
    {
        .objSize     = sizeof(cy_stc_si_toc_t) - sizeof(uint32_t),  /**< Object Size (Bytes) excluding CRC */
        .magicNum    = CY_SI_TOC2_MAGICNUMBER,                      /**< TOC2 ID (magic number) */
        .userKeyAddr = (uint32_t)&CySecureKeyStorage,               /**< User key storage address */
        .smifCfgAddr = 0UL,                                         /**< SMIF config list pointer */
        .appAddr1    = CY_SI_SECURE_FLASH_BEGIN,                    /**< App1 (Secure Image) start address */
        .appFormat1  = CY_SI_APP_FORMAT_CYPRESS,                    /**< App1 Format */
        .appAddr2    = CY_SI_USERAPP_FLASH_BEGIN,                   /**< App2 (user application image) start address */
        .appFormat2  = CY_SI_APP_FORMAT_CYPRESS,                    /**< App2 Format */
        .shashObj    = 1UL,                                         /**< Number of additional objects to add to the
SECURE_CMAC */
        .sigKeyAddr  = (uint32_t)&SFLASH->PUBLIC_KEY,               /**< Address of signature verification key */
        .tocFlags    = CY_SI_TOC_FLAGS,                             /**< Flash Boot flags stored in TOC */
        .crc         = 0UL                                          /**< CRC populated bycymcueltool */
            };
```

6.1.1.2 Cypress Standard Application Format (Optional Reading)

Secure applications must be validated with a public Crypto key. To do this, the application must use the Cypress standard application format that includes a digital signature. 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. Although there is a place for both CM0+ and CM4 images in this format, the Secure Image requires only the CM0+ image. The user application includes both images, see Figure 7.

Figure 7. Cypress Standard Application Format

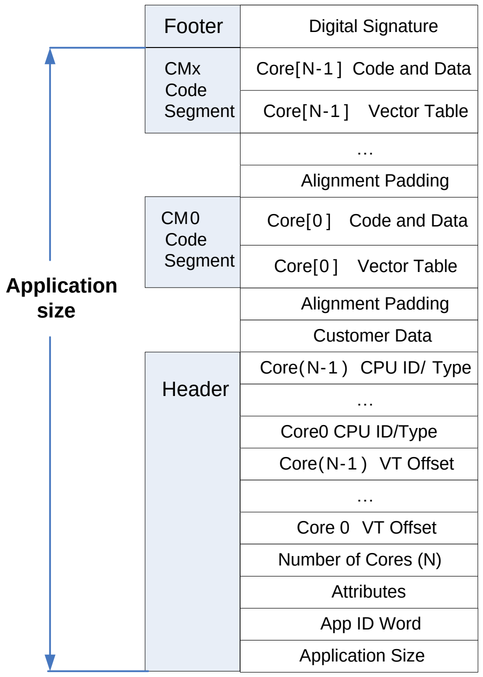


Table 4 provides the details of the header section. It defines the total size, the number of cores, the type of application, and the offset to each core application vector table.

Table 3. Application Header Details

Offset	Size	Item	Description
0x0	4 bytes	Application Size	Flash image size in bytes
0x4	4 bytes	Application ID Word	This value identifies the type of the flash image. Bit 31 – 28: Always 0 Bit 27 – 24: Major Version Bit 23 – 16: Minor Version Bit 15 – 0: Application ID. For example 0x0000 – 07FFF User Application ID 0x8001 – Flash boot 0x8002 – Security Image 0x8003 – Bootloader Values between 0x8004 – 0xFFFF Reserved
0x8	4 bytes	Attribute	Reserved for future use
0xC	4 bytes	Number of Cores(N)	Number of cores used by the application.
0x10 + (4*i)	4 bytes	Core(i) VT offset	Offset to vector table in Core(i) code segment
0x10 + (4*N) + (4*i)	4 bytes	Core(i) CPU ID/Type	Customer assigned CPU ID and Core index. Bit 31 – 20: CPU ID. This is the part number value from the CPUID [15:4] register in an Arm device. Bit 7 – 0: Core Index The core index is used to distinguish between multiple cores of the same type. For example, consider a system consisting of M0+ and two M4s. The M0+ is identified by CPUID=0xC60 and Core Index=0. The first M4 is identified by CPUID=0xC24 and Core Index=0. The second M4 is identified by CPUID=0xC24 and Core Index=1.

To generate proper values for the application header, an instance of the `cy_stc_si_appheader_t` structure must be included in the project. This structure is included in the *cy_si_main.c* source file.

The following is a snippet of code from the *cy_si_main.c* file that is a part of the example "secure_image" project.

```
/** Secure Application header */
CY_SECTION(".cy_app_header") __USED static const cy_stc_si_appheader_t cy_si_appHeader = {
    .objSize        = CY_SI_SECURE_FLASH_END - CY_SI_SECURE_FLASH_BEGIN - CY_SI_SECURE_DIGSIG_SIZE,
    .appId          = (CY_SI_APP_VERSION | CY_SI_APP_ID_SECUREIMG),
    .appAttributes  = 0UL,                          /* Reserved */
    .numCores       = 1UL,                          /* Only CM0+ */
    .core0Vt        = CY_SI_VT_OFFSET,              /* CM0+ VT offset */
    .core0Id        = CY_SI_CPUID | CY_SI_CORE_IDX, /* CM0+ core ID */
};

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

This code can be used as is, but if you would like to change the Secure Image version, change the `CY_SI_VERSION_MAJOR` and `CY_SI_VERSION_MINOR` in the *cy_si_config.h* file in the project. The version can be changed by updating the two following constants:

```
#define CY_SI_VERSION_MAJOR             1UL /**< Major version */
#define CY_SI_VERSION_MINOR             0UL /**< Minor version */
```

6.1.2 cy_si_config.c

This file contains the configuration for all protections units and the memory blackout lists. The section of the file labelled "Memory Blacklist" is used to limit non-secure process access to reading and writing flash and SRAM. This can be done with protection units, but depending on the complexity of your system, you may run out of protection units. The blacklist is divided into flash and SRAM. To add additional protected areas, you add to the arrays that are already provided. The following is a copy of the code that can be modified to add additional SRAM and flash blacklisted areas.

```
/****************************************
*          Memory Blacklists
****************************************/

/** RAM Blacklist: [beginAddr][endAddr] */
const uint32_t CyRamBlacklist[CY_SI_MEM_RAM_BLACKLIST_SIZE][CY_SI_MEM_ADDR_SIZE] = {
    {CY_SI_SECURE_RAM_BEGIN, CY_SI_SECURE_RAM_END}
    /* Expand as necessary */
};

/** Flash Blacklist: [beginAddr][endAddr] */
const uint32_t CyFlashBlacklist[CY_SI_MEM_FLASH_BLACKLIST_SIZE][CY_SI_MEM_ADDR_SIZE] = {
    {CY_SI_SECURE_FLASH_BEGIN,  CY_SI_SECURE_FLASH_END},
    {CY_SI_USERAPP_FLASH_BEGIN, CY_SI_USERAPP_FLASH_END},
    {CY_SFLASH_BASE,            CY_SFLASH_BASE + CY_SFLASH_SIZE},
    {CY_ROM_BASE,               CY_ROM_BASE + CY_ROM_SIZE}
    /* Expand as necessary */
 };
```

The majority of *cy_si_config.c* is used to set up and configure the protection units. The section starting with "SMPU configurations" handles the initialization of more than 20 protection unit (master and slave) structures. Table 4 shows the structure initialization performed in the function, `Cy_SI_Config_Prot()`. This function uses the initialized structures to configure protection units. The majority of these are configured to allow only PC=0 bus masters with "secure" attribute. To add more protected areas, configure an additional protection unit structure and then in `Cy_SI_DonfigProt()`, call the appropriate function to initialize the protection unit hardware. Read the manual page on Protection Units in the Peripheral Driver Library manual to become familiar with the protection unit concept and APIs used.

Table 4. List of Default Protection Units Used

Protection Unit Type	Region	Notes
SMPU	ROM	Protect ROM except for sub-regions 0 and 7.
SMPU	SFlash	The entire SFlash is protected.
SMPU	Secure Keys	User-defined secure key region
SMPU	SRAM	SRAM area used for secure CPU (CM0+) stack and heap
SMPU	CPUSS	`CPUSS AP_CTL`, `PROTECTION`, `CM0_NMI_CTL`, `DP_CTL`, & `MBIST_CTLS` (The important one to protect here is `CM0_NMI_CTL`). See Technical reference manual for more details.
SMPU	CPUSS	CPUSS WOUNDING & CM0_PC0_HANDLER Disallow malicious software from wounding memory regions or changing the PC=0 handler location.
SMPU	FM_CTL	Flash macro should only be accessed by the ROM code. You can protect the entire flash controller if required.
SMPU	EFUSE	
PPU	Crypto MMIO	Crypto registers are accessible only by the Crypto server running on the CM0+ CPU operating at PC=0.
PPU	MS_CTL	MMIO region, only allow PC=0 to modify what PC values bus masters can take.

The last section of this file after protection unit initialization are the utility functions used by the Secure Image. These functions do not require any modification. See the Secure Image section of the Peripheral Driver Library manual or more information on these utility functions.

6.1.3 cy_si_keyStorage.c

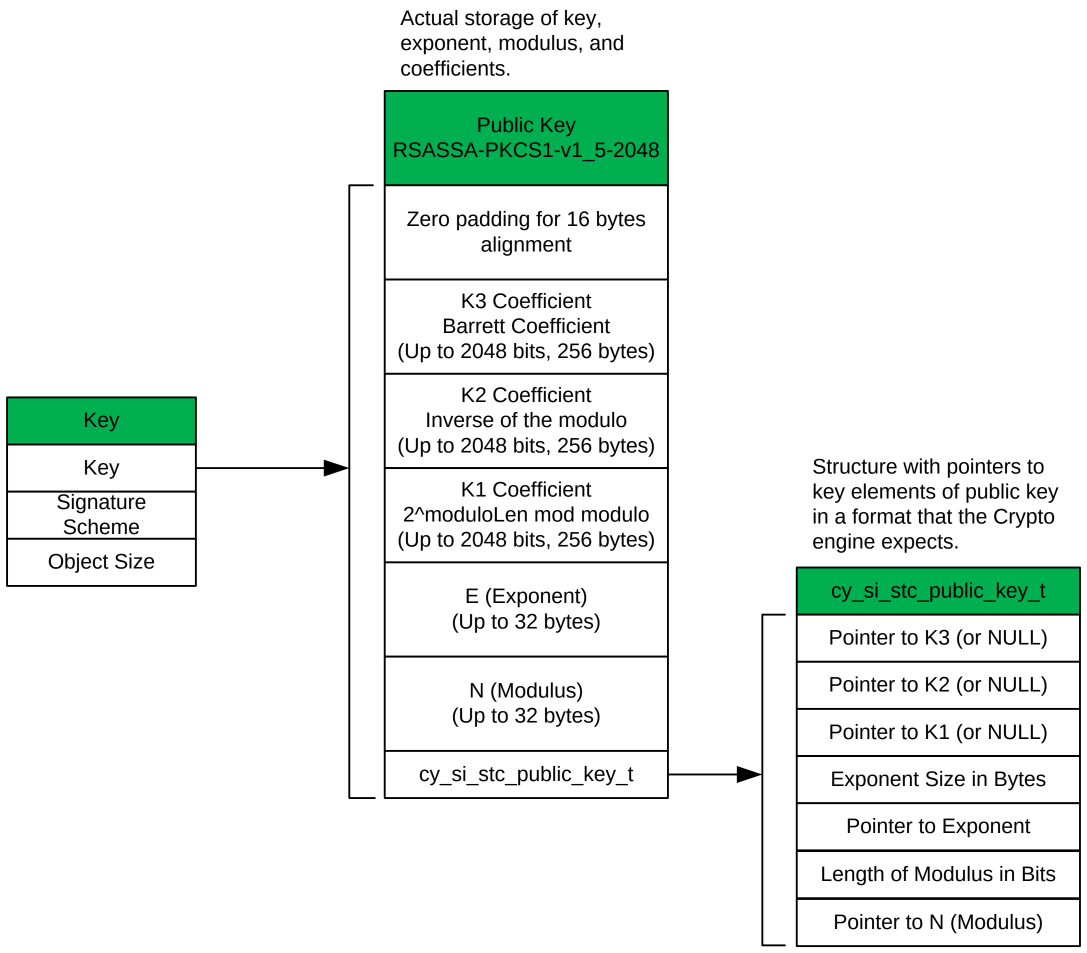
This contains two arrays: one is for the public key (`cy_publicKey`) that resides in SFlash. Replace the key in the “moduloData” element of the structure with one that you generate using the procedure defined in Appendix A “Creating Public and Private Keys.”

The other array (`CySecureKeyStorage`) is optional and resides in user flash. You can add as many keys in user flash as there is free space. The format of these keys is straight-forward: you can see the format just by looking at the source code or reading the next section. More details of how to generate the public and private keys are located in Appendix A “Creating Public and Private Keys.”

6.1.3.1 Cypress Secure Boot RSA Public Key Format (Optional Reading)

The SFlash region stores the public key. It is stored 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. Figure 8 shows the format.

Figure 8. Crypto Key Structures


Actual storage of key, exponent, modulus, and coefficients.
Public Key RSASSA-PKCS1-v1_5-2048
Zero padding for 16 bytes alignment
K3 Coefficient Barrett Coefficient (Up to 2048 bits, 256 bytes)
K2 Coefficient Inverse of the modulo (Up to 2048 bits, 256 bytes)
K1 Coefficient 2^moduloLen mod modulo (Up to 2048 bits, 256 bytes)
E (Exponent) (Up to 32 bytes)
N (Modulus) (Up to 32 bytes)
cy_si_stc_public_key_t
Key
Key
Signature Scheme
Object Size
Structure with pointers to key elements of public key in a format that the Crypto engine expects.
cy_si_stc_public_key_t
Pointer to K3 (or NULL)
Pointer to K2 (or NULL)
Pointer to K1 (or NULL)
Exponent Size in Bytes
Pointer to Exponent
Length of Modulus in Bits
Pointer to N (Modulus)

The key is stored in three structures. The first structure “Key” is stored as an object that can easily be included in the Secure_HASH calculation. The “Signature Scheme” defines the structure of the key. This example uses RSASA-PKCS1-v1_5-2048. The “Object Size” contains the full size of the public key object, which contains the entire three structures.

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. 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. A Python script is provided below to convert the default OpenSSL format to the Cypress Structure format.

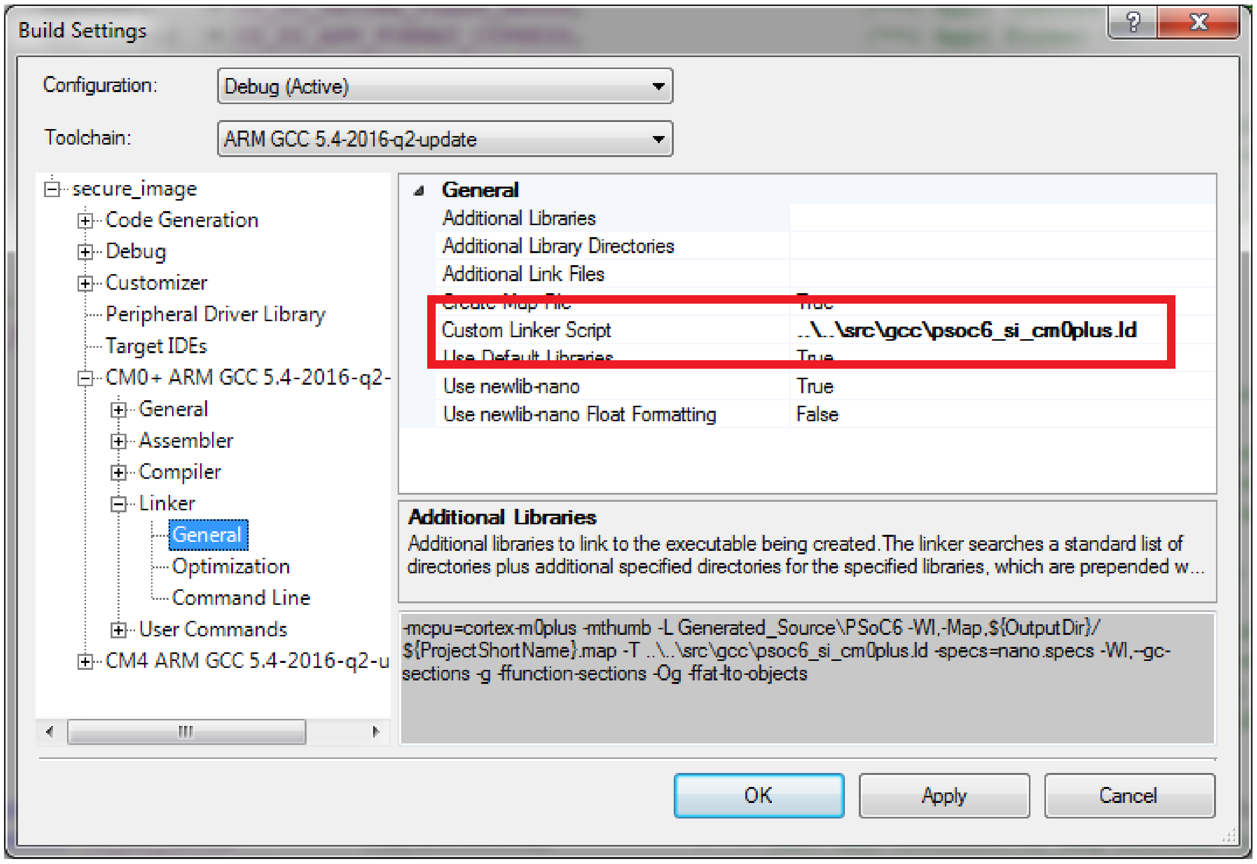
6.1.4 Scripts and Settings

There are a few scripts that must be added to the project to simplify the build process. Adding them to the “Build Settings” for the project in PSoC Creator makes their execution automatic when your project is being built. For the “secure_image” project in the template, they have already been added so there is no additional work that you must do if starting with the template.

6.1.4.1 psoc6_si_cm0plus.ld Script

This is a custom linker file required for specifying the range of SRAM and flash that is needed for your application running on CM0+. This file could change depending on the size of the secure_Image project, but little should change from the default. See Figure 9.

Figure 9. Adding psoc6_si_cm0plus.ld Script



Configuration: Debug (Active)

Toolchain: ARM GCC 5.4-2016-q2-update

General

Additional Libraries	
Additional Library Directories	
Additional Link Files	
Create Map File	True
Custom Linker Script	**..\..\src\gcc\psoc6_si_cm0plus.ld**
Use Default Libraries	True
Use newlib-nano	True
Use newlib-nano Float Formatting	False

Additional Libraries

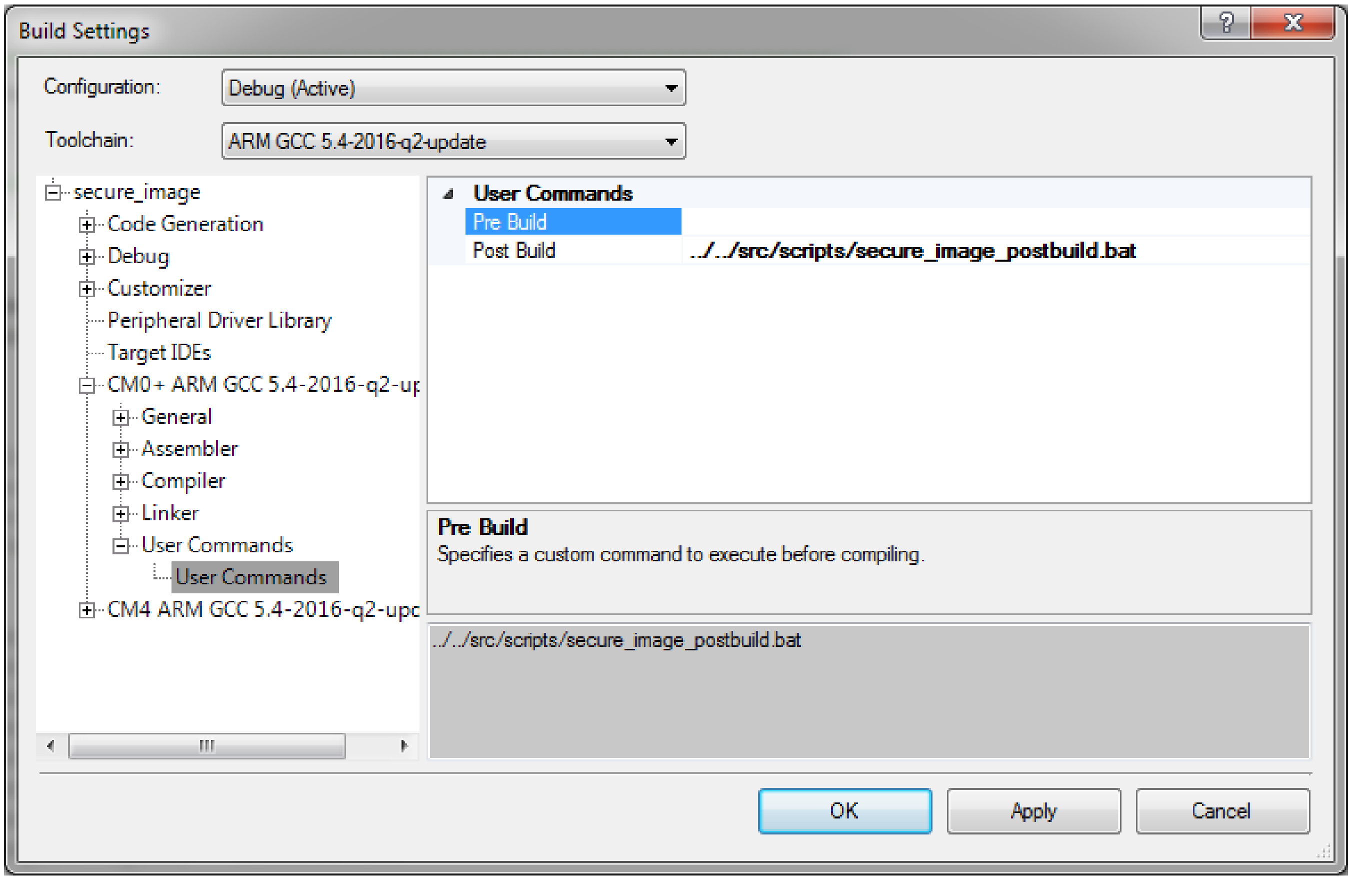
Additional libraries to link to the executable being created. The linker searches a standard list of directories plus additional specified directories for the specified libraries, which are prepended w...

```
-mcpu=cortex-m0plus -mthumb -L Generated_Source\PSoC6 -Wl,-Map,${OutputDir}/
${ProjectShortName}.map -T ..\..\src\gcc\psoc6_si_cm0plus.ld -specs=nano.specs -Wl,--gc-
sections -g -ffunction-sections -Og -ffat-lto-objects
```

6.1.4.2 secure_image_postbuild.bat Script

This script is designed to run after the secure_image CM0+ application link step. The script runs the *cymcuelftool.exe* application to export select symbols as defines in the *cy_si_export.txt* file, to be referenced by the user application. It also uses the *cymcuelftool.exe* application to create a digital signature of the Secure Image, which is used to validate the binary before it is executed during boot time. This script uses the *rsa_private.txt* private key file located at *security/secure_image/src/scripts/keys* to generate the encrypted digital signature that is used to validate the project. The private/public key pair should be generated before compiling the secure image project. See Appendix A "Creating Public and Private Keys" to learn how to generate the private/public key pairs. The source of this script is text and is fully documented with comments. See Figure 10.

Figure 10. Adding secure_image_postbuild.bat Script


Build Settings

Configuration: Debug (Active)

Toolchain: ARM GCC 5.4-2016-q2-update

User Commands

Pre Build	
Post Build	../../src/scripts/secure_image_postbuild.bat

Pre Build
Specifies a custom command to execute before compiling.

../../src/scripts/secure_image_postbuild.bat

The Secure Image should be compiled first so that when the user project is compiled, it will automatically pack the Secure Image hex file along with it.

6.2 User Application Project and Modifications (user_app0)

If starting from the template to create your user application, there is very little to modify. If the user application was generated independently, and later added to the Workspace, this section takes you through those changes.

6.2.1 Main_cm0p.c

This file contains the main() function and must include the application header structure as was done with the Secure Image project. The following is an example of what is provided in the template.

```
/*********************************/
/*    Application Header         */
/*********************************/
#define CY_USERAPP_MAJOR_VERSION    (1UL)           /** App Major version [27:24] */
#define CY_USERAPP_MINOR_VERSION    (0UL)           /** App Minor version [23:16] */
#define CY_USERAPP_VERSION          ((CY_USERAPP_MAJOR_VERSION << 24u) | (CY_USERAPP_MINOR_VERSION <<
16u)) /** App version */
#define CY_USERAPP_ID               (CY_USERAPP_VERSION | CY_SI_APP_ID_USERAPP) /* Application ID */
#define CY_USERAPP_CM0P_CPUID       (0xC6000000UL)  /** CM0+ CPUID[15:4] */
#define CY_USERAPP_CM4_CPUID        (0xC2400000UL)  /** CM4 CPUID[15:4] */

#define CY_USERAPP_CM0P_VT_OFFSET   ((uint32_t)(&__Vectors[0]) - CY_SI_USERAPP_FLASH_BEGIN -
(sizeof(uint32_t) * 4u))  /** CM0+ VT offset */
#define CY_USERAPP_CM4_VT_OFFSET    (CY_SI_USERAPP_CM4_FLASH_BEGIN - CY_SI_USERAPP_FLASH_BEGIN -
(sizeof(uint32_t) * 5u)) /** CM4 VT offset */

/** User application header */
CY_SECTION(".cy_app_header") __USED static const cy_stc_user_appheader_t cy_user_appHeader = {
    .objSize        = CY_SI_USERAPP_FLASH_END - CY_SI_USERAPP_FLASH_BEGIN - CY_SI_USERAPP_DIGSIG_SIZE,
    .appId          = CY_USERAPP_ID,                /* Application ID */
    .appAttributes  = 0UL,                          /* Reserved */
    .numCores       = 2UL,                          /* CM0+ and CM4 */
    .core0Vt        = CY_USERAPP_CM0P_VT_OFFSET,    /* CM0+ Vector table offset */
    .core1Vt        = CY_USERAPP_CM4_VT_OFFSET,     /* CM4 Vector table offset */
    .core0Id        = CY_USERAPP_CM0P_CPUID,        /* CM0+ core ID */
    .core1Id        = CY_USERAPP_CM4_CPUID          /* CM4 core ID */
};
```

You should update the MAJOR and MINOR version constants at the top of the code segment to the required value. The `CY_USERAPP_ID` is up to you; it should be between 0x0000 and 0x7FFF.

The only other code that must be included is to set the stack pointer to the same area as what was used for the Secure Image.

```
int main(void)
{
    /* Set the MSP to the one defined in the Secure Image */
           __set_MSP(*(uint32_t *)CY_SI_SECURE_VT_ADDRESS);
```

6.2.2 Scripts and Settings

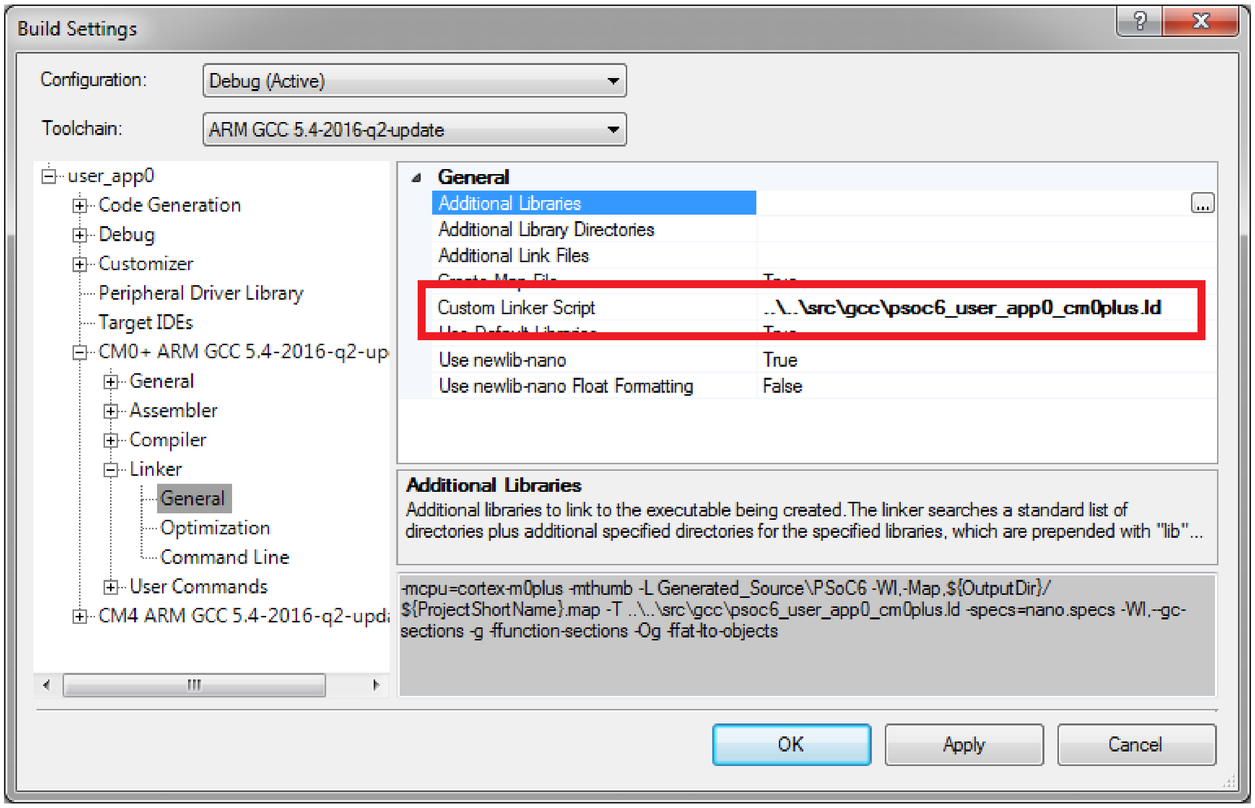
There are a few scripts that must be added to the project to simplify the build process. Adding them to the “Build Settings” for the project in PSoC Creator makes their execution automatic when your project is being built. If the user application was used from the template, the scripts have already been added.

6.2.2.1 Psoc6_user_app0_cm0plus.ld

This is a custom linker file is required to specify the range of SRAM and flash that is required for your application that is running on CM0+. This script may require modification depending on the size of the Secure Image project, but not much should be changed from the default. This script can be found in the *secure_image* folder with the following path: *secure_image/src/gcc/psoc6_user_app0_cm0plus.ld*.

Figure 11 shows how this script is incorporated into the build settings.

Figure 11. Adding Linker Script for CM0+ Memory Requirements



Additional Libraries	
Additional Library Directories	
Additional Link Files	
Create Map File	True
Custom Linker Script	..\..\src\gcc\psoc6_user_app0_cm0plus.ld
Use Default Libraries	True
Use newlib-nano	True
Use newlib-nano Float Formatting	False

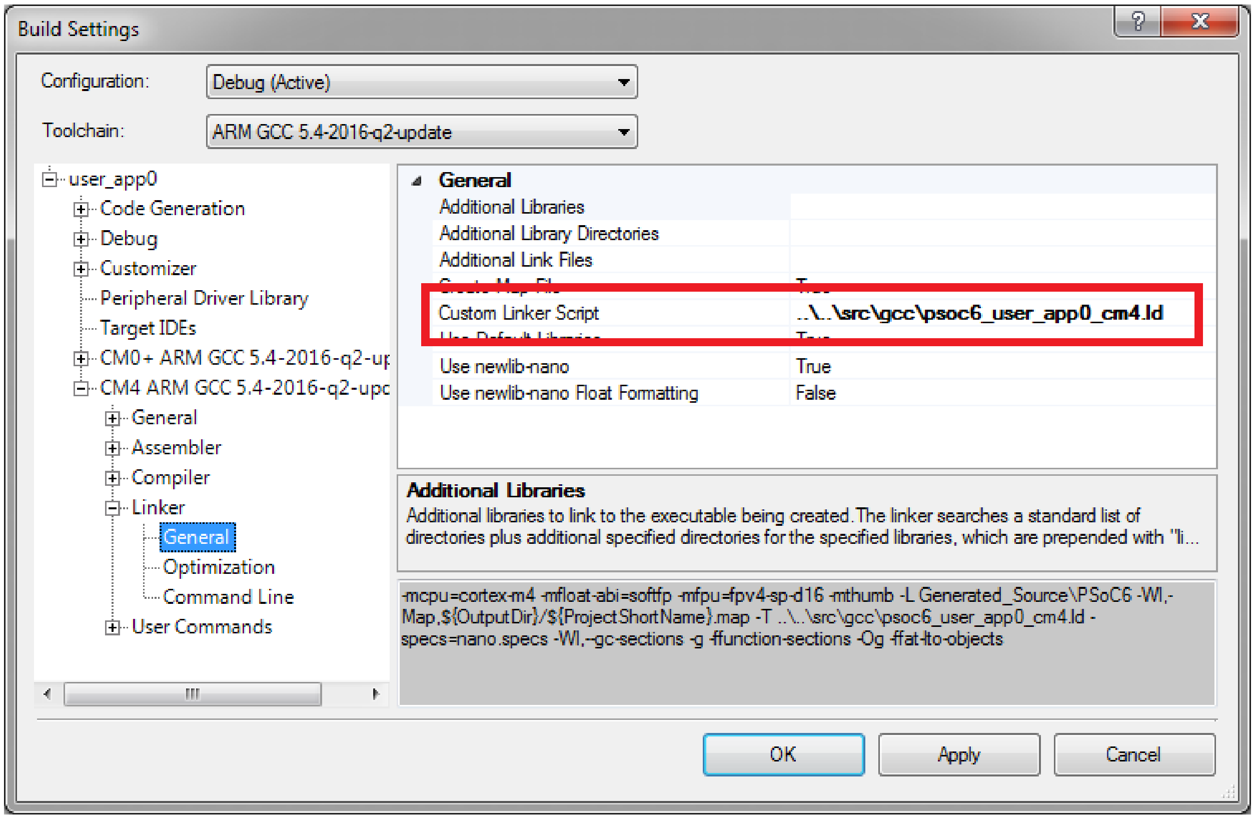
Additional Libraries
Additional libraries to link to the executable being created. The linker searches a standard list of directories plus additional specified directories for the specified libraries, which are prepended with "lib"...

```
-mcpu=cortex-m0plus -mthumb -L Generated_Source\PSoC6 -Wl,-Map,${OutputDir}/${ProjectShortName}.map -T ..\..\src\gcc\psoc6_user_app0_cm0plus.ld -specs=nano.specs -Wl,--gc-sections -g -ffunction-sections -Og -ffat-lto-objects
```

6.2.2.2 Psoc6_user_app0_cm4.ld Script

This is a custom linker file that is required to specify the range of SRAM and flash required for your application running on CM4. This file could change depending on the size of the secure_Image project, but little should need to be changed from the default. This file can be found in the secure_image folder that was copied: *secure_image/src/gcc/psoc6_user_app0_cm4.ld*. Figure 12 shows how this script is incorporated into the build settings.

Figure 12. Adding psoc6_user_app0_cm4.ld Script



Configuration: Debug (Active)

Toolchain: ARM GCC 5.4-2016-q2-update

General	
Additional Libraries	
Additional Library Directories	
Additional Link Files	
Create Map File	True
Custom Linker Script	..\..\src\gcc\psoc6_user_app0_cm4.ld
Use Default Libraries	True
Use newlib-nano	True
Use newlib-nano Float Formatting	False

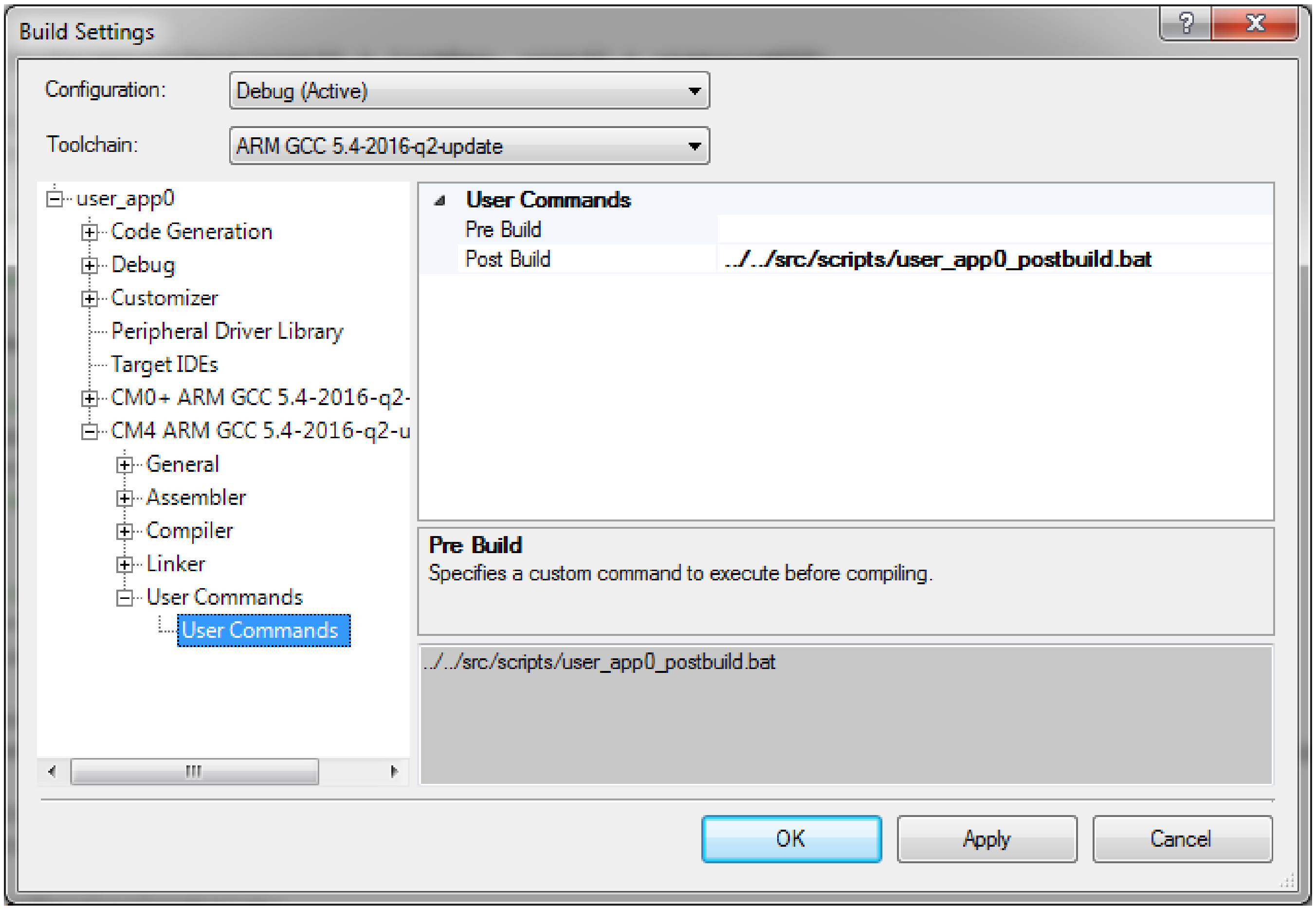
Additional Libraries
Additional libraries to link to the executable being created. The linker searches a standard list of directories plus additional specified directories for the specified libraries, which are prepended with "li...

```
-mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -mthumb -L Generated_Source\PSoC6 -Wl,-Map,${OutputDir}/${ProjectShortName}.map -T ..\..\src\gcc\psoc6_user_app0_cm4.ld -specs=nano.specs -Wl,--gc-sections -g -ffunction-sections -Og -ffat-lto-objects
```

6.2.2.3 User_app0_postbuild.bat Script

This script runs after the CM4 application link step of the user application. It runs the cymcuelftool application to generate the digital signature for the user application image. It also combines the user application with the Secure Image application to create a single elf file and a single hex file. See Figure 13.

Figure 13. Adding user_app0_postbuild.bat Script



Configuration: Debug (Active)

Toolchain: ARM GCC 5.4-2016-q2-update

```
user_app0
  Code Generation
  Debug
  Customizer
  Peripheral Driver Library
  Target IDEs
  CM0+ ARM GCC 5.4-2016-q2-
  CM4 ARM GCC 5.4-2016-q2-u
    General
    Assembler
    Compiler
    Linker
    User Commands
      User Commands
```

User Commands	
Pre Build	
Post Build	../../src/scripts/user_app0_postbuild.bat

Pre Build
Specifies a custom command to execute before compiling.

../../src/scripts/user_app0_postbuild.bat

When using the supplied templates for both the Secure Image and the user application, the hex files will automatically be combined into a single hex file. If you use the template unchanged, this file is located at */secure_image/example/user_app0.cydsn/CortexM4/ARM_GCC_541/Debug/user_app0.hex*.

6.3 Programming eFuse to Change Lifecycle Stage

WARNING: This is the most critical section to perform properly. The eFuse bits can be programmed only from 0 to 1. If you program these bits incorrectly, you will need to remove the device from the board and dispose it off. You should not proceed to this step until you have proven all other code is working properly. Also, the only way to update the code after entering the Secure state is to have a proven bootloader working. Also, the supply voltage V_{DDD} must be set to 2.5 V to program eFuse.

A special project is included in the security folder that you previously copied into the user directory. The project is located at *\security\secure_image\example\efuse_gen.cydsn\efuse_gen.cyprj*. The two files of interest are *cy_si_efuse.h* and *cy_si_efuse.c*. The *cy_si_efuse.c* file contains the structures that compile and create the hex file that is required to program eFuse. By default, this configuration will not generate any eFuse programming, just in case the project is compiled and programmed by accident. To enable programming, a line in *cy_si_efuse.h* must be changed from;

“#define CY_SI_EFUSE_AVAILABLE   0”

to

“#define CY_SI_EFUSE_AVAILABLE   1”.

The rest of the modifications to this project are in the *cy_si_efuse.c* file. There are three sections to this file.

- DEAD ACCESS RESTRICTIONS
- SECURE ACCESS RESTRICTIONS
- LIFECYCLE_STAGE

The “Dead” and “Secure” access restrictions have the same parameters or limits. Both are composed of two structures `_ACCESS_RESTRICT0` and `_ACCESS_RESTRICT1`. The ‘Dead Access’ restrictions set the limits to access the device through the debug port in the event of a failed boot process. This can occur for several reasons such as invalid TOC2 and invalid program block. The ‘Secure Access’ restrictions limit the debug access during runtime. You may block all access or only partially limit access to the device. The debug port should be totally disabled when in the Secure stage.

A byte of data is required to program each bit of the eFuse. The following pattern is used to program, validate, or set as ‘don’t care’ each bit of eFuse.

- 0x00 -> 0 // Validate but do not program
- 0x01 -> 1 // Set eFuse bit to “1”
- 0xFF -> Don’t Care // Do not program or validate area.

The following is an example of the default settings for the Secure mode.

```
.SECURE_ACCESS_RESTRICT0 =
{
    .CM0_DISABLE = CY_EFUSE_STATE_SET,                     /* Disable CM0+ access port */
    .CM4_DISABLE = CY_EFUSE_STATE_SET,                     /* Disable CM4 access port */
    .SYS_DISABLE = CY_EFUSE_STATE_SET,                     /* Disable System access port */
    .SYS_AP_MPU_ENABLE = CY_EFUSE_STATE_SET,               /* Enable the system access port MPU */
    .SFLASH_ALLOWED = CY_EFUSE_SFLASH_ALLOWED_ENTIRE,      /* Allow SYS AP MPU protection of SFlash */
    .MMIO_ALLOWED = CY_EFUSE_MMIO_ALLOWED_ENTIRE,          /* Allow SYS AP MPU protection of MMIO */
},
.SECURE_ACCESS_RESTRICT1 =
{
    .FLASH_ALLOWED = CY_EFUSE_FLASH_ALLOWED_ENTIRE,        /* Allow SYS AP MPU protection of Flash */
    .SRAM_ALLOWED = CY_EFUSE_SRAM_ALLOWED_ENTIRE,          /* Allow SYS AP MPU protection of SRAM */
    .SMIF_XIP_ALLOWED = CY_EFUSE_SMIF_XIP_ALLOWED_ENTIRE,  /* Allow SYS AP MPU protection of SMIF XIP
*/
    .DIRECT_EXECUTE_DISABLE = CY_EFUSE_STATE_SET           /* Disable "direct execute" system call */
 },
```

Parameter	Description
CM0_DISABLE	Disable debug access to CM0+ CPU.
CM4_DISABLE	Disable debug access to CM4
SYS_DISABLE	Disable debug access to System access port
SYS_AP_MPU_ENABLE	Enable access to system access port
SFLASH_ALLOWED	Enable access to SFlash, enabled by default.
MMIO_ALLOWED	Enable access to MMIO registers
FLASH_ALLOWED	Enable access to flash
SRAM_ALLOWED	Enable access to SRAM
SMIF_XIP_ALLOWED	Enable execution from external SMIF memory.
DIRECT_EXECUTE_DISABLE	Disable "direct execute" system call.

The last section `LIFECYCLE_STAGE` is where you set the lifecycle state, and therefore the name. By default, the eFuse project is set to `SECURE_WITH_DEBUG`. Advance to this stage first because it allows the same access restrictions as Normal stage will be enabled to allow full debug. After you advance to Secure, you will not be able to use the debugger or to reprogram without a bootloader.

```
.LIFECYCLE_STAGE =
{
 .NORMAL = CY_EFUSE_STATE_IGNORE,            /* Normal lifecycle already set - ignore */
 .SECURE_WITH_DEBUG = CY_EFUSE_STATE_SET,    /* Transition to "Secure with Debug" lifecycle */
 .SECURE = CY_EFUSE_STATE_IGNORE,            /* Cannot be at both "Secure" and "Secure with
                                                Debug" - ignore */
 .RMA = CY_EFUSE_STATE_IGNORE,               /* Cypress use only - ignore */
 .RESERVED = CY_EFUSE_LIFECYCLE_RESERVED0    /* Reserved bits ignored */
    },
```

When you are ready to advance to the Secure stage, a different part must be used that is still in the Normal stage. This is because it is not possible to move from the SECURE_WITH_DEBUG stage to the SECURE stage, due to possible security issues. Changing the line "`.SECURE = CY_EFUSE_STATE_IGNORE`" to "`.SECURE = CY_FUSE_STATE_SET`" causes the SECURE bit to be set. After the change, rebuild the project and program the part, but remember that the device V_{DDD} must be at 2.5 V.

Important Notes:

1. When using PSoC Programmer to change the Lifecycle mode to either SECURE or SECURE_WITH_DEBUG, it does more than just programming the eFuse bits. The following are the actual steps. Some programmers may not perform all three steps as outlined below.
    a. Validates the SFlash area with the internal Factory_HASH to make sure the part has not been modified after leaving the factory.
    b. Generates Secure_HASH based on the TOC2 entries. By default, Secure_HASH includes all of SFlash. Additional areas from user flash may be included if additional entries are added in TOC2. This hash is written into the Secure_HASH area of eFuse along with the number of zeros in the hash. This guarantees that the hash cannot be modified by simply changing zeros to ones.
    c. Program the eFuse bits for access restrictions and the Lifecycle mode SECURE bit.
2. If using a Cypress MiniProg3 device, it may need to be a revision "*C" or greater. The "*C" revision programmers have a more accurate measurement of V_{DDD} to validate that it is 2.5 V, as required to program eFuse.
3. The default PSoC Programmer settings ("Options/Programmer Options" tab) must be changed to program Secure mode bits in eFuse.
    a. **Chip Lock:** Enable programming of Chip Security data for PSoC 3/4/5/6 devices. Change this option to "Enable" before programming any eFuse. Set it back to "Disable" when complete.
    b. **Partial Program:** Set this to "Enable" when programming eFuse. This is because you must first program your flash before doing this step. If you program eFuse (go into Secure mode), you will not be able to program the flash.

c. **Erase Flash:** Change this option to “Disable” so that the application is not erased before eFuse is programmed. Failure to do this will erase the entire part and move the device to Secure mode, which will not allow reprogramming the part. In addition, any bootloader present in flash will be erased.

7 Summary

This application note has shown how the PSoC 6 MCU hardware and Secure Image template can be used to provide a secure environment for your application by customizing it for your specific application needs. You can also use the Secure Image template as an example to implement a custom secure application to perform a similar task. The following is a summary of steps to create a secure system as defined by this application note.

7.1 Step-by-step Summary

1. Copy the provided Secure Image template from *C:/Program Files (x86)/Cypress/PDL/3.0.1/security* to your own project directory. (See Chapter 6, Building a Secure System)
2. Change the version (if needed) in the “secure_image” application header by updating the `#define` statements in *cy_si_config.h*.
3. Add any additional areas to the blacklist areas. (See 6.1.2 cy_si_config.c)
4. Modify the protection unit configuration section as required. (See 6.1.2 cy_si_config.c)
5. Generate your own private/public key pair and add the public key to the file *cy_si_keystorage.c* file (See Section 6.1.3 cy_si_keyStorage.c and Appendix A)
6. (Optional) Add the provided scripts to the build options (See 6.1.4 Scripts and Settings). This step is not required if using the template.
7. Build the Secure Image project.
8. Change the Major/Minor application version and the application ID as needed in the *main_cm0p.c* file of the “user_app0” application.
9. Add the Application header structure and `#defines` to the *main.c* file in the user project. (See 6.2.1 Main_cm0p.c)
10. Add the scripts to the user project build options. (See 6.2.2 Scripts and Settings)
11. Build the projects and program the device. The included build scripts automatically include the Secure Image code and generate a single hex file. This file is located at */secure_image/example/user_app0.cydsn/CortexM4/ARM_GCC_541/Debug/user_app0.hex*, if you use the unchanged template.
12. Verify that all code is working properly.
13. Program eFuse to SECURE_WITH_DEBUG and validate project. The device **CANNOT** be changed from SECURE_WITH_DEBUG after this step (See 6.3 Programming eFuse to Change Lifecycle Stage).
14. Program device into SECURE mode and validate. (See 6.3 Programming eFuse to Change Lifecycle Stage) Remember that the part that was changed to SECURE_WITH_DEBUG will not be able to be changed to Secure stage. Also, once in the Secure mode, it cannot be changed back to Normal.

7.2 Design Planning

Determining the allocation of both SRAM and flash is important as you develop your secure application. A typical secure system includes at least three or four sections of memory.

- Secure Image (CM0+)
- (Optional) Bootloader (CM0+)
- CM0+ user application
- CM4 user application

Each of these partitions requires some amount of flash and SRAM that you must allocate. You must make the required modifications that are project-specific. Example linker scripts provided with the example and described in this application note are a good place to start. They are well documented so that you can make the required changes.

7.3 More Reading

Read the datasheets on the following driver libraries before attempting modification of the Secure Image template.

- CRYPTO – Cryptographic Accelerator
- PROT – Memory Protection (Protection units, MPU, SMPU, and PPU)
- SYSINT – System Interrupt

The datasheet for these libraries can be found in a single document, Peripheral Driver Library. This manual can be found on the internet or in the PSoC Creator 4.2 or later installation.

- *C:/Program Files (x86)/Cypress/PDL/3.0.1/doc/pdl_user_guide.pdf*. As newer versions of PDL are released the path will change from ../3.0.1/… to a higher revision.
- On the internet, Peripheral Driver Library (PDL).

The Secure Image template can be found at *C:/Program Files (x86)/Cypress/PDL/3.0.1/security*. As mentioned above with the datasheet, the PDL version will increase and you may need to update the path for the latest release.

Appendix A Creating Public and Private Keys

The Secure Image is provided with two files, *rsa_private.txt* and *rsa_public.txt*. These files contain a sample private and public key, which are provided only as place holders to help you get your build system working properly. They should be replaced with your own files before going into production. This section explains how to generate a set of public and private keys, to format them to a ‘C’ format and to update the source file with the new key.

A.1 Additional Tools Required

1. OpenSSL
2. Python 2.7 or Python 3 (Required for one of the provided scripts used to format the public key.)

There are several ways to generate RSA private and public keys. The method documented below requires installing **OpenSSL** on your computer. Source or binaries for OpenSSL may be downloaded from several sources on the internet.

A.2 Provided Scripts

The PDL installation provides two scripts to convert the output from OpenSSL to a format compatible with C and the structures used by the Secure Image to store the public key. These scripts are available in the following path:

C:\Program Files (x86)\Cypress\PDL\3.0.1\security\secure_image\src\scripts

Copy the two scripts *rsa_keygen.bat* and *rsa_to_c.py* to a work directory where the key pairs and a section of C code will be generated. The path of this work directory must not include spaces.

The batch script *rsa_keygen.bat*, calls OpenSSL functions, so as mentioned above, OpenSSL must be installed on your computer. The batch file creates a directory in your work directory called “keys_generated”. This creates two files containing the private and public keys generated with OpenSSL. These files are called “rsa_private_generated.txt” for the private key, and “rsa_public_generated.txt” for the public key.

Next, the batch file will call the Python script, “rsa_to_c.py”. This script extracts the public key coefficients and public key from the generated public key file, and then formats the data to be compatible with C and the Cypress public key format. The output is placed in the file *rsa_to_c_generated.txt* in the *keys_generated* directory along with the public and private key files.

A.3 Running the Scripts

The batch file *rsa_keygen.bat*, may be called from a command line interface or by double-clicking it in a Windows 7 or 10 environment. After the script runs, verify that the following files have been generated in the *keys_generated* directory.

- *rsa_private_generated.txt* (Private RSA key)
- *rsa_public_generated.txt* (Public RSA key)
- *rsa_to_c_generated.txt* (Public key in C format)

An example of what the private key file should look like:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEA6GAUJmZyLhv6lMTYglg7oQkuMQgvixZhFWIG7sbBzVdTnXrH
NA6Jus/cELGmyl46ImhxHkGXXP0Ja9sVIJM+lCIC6urX+tdCJ61mCiofuCwrwKRm
yYyBUV1m0vxUxXh3RgJ/BmzqFVwgCCXj6MAbI0fzG4K6EZVvNXLdVbXkGHt1dAtv
IZCr5WFJbASAnJpBvRNzWA5bDyT6LW8g1HtZC6Uk6gesrgMh5aEXpWXTjtvD9miM
F3K4xCl4VcV/Q68f3a011gGU2C6WlOgsIF2Ugf28YyY2ddppFaoHgtZLieqLkFI0
kY9SgY6ZZeHw/mIMY1wyKEvryY4WZmyGqw/ZmQIDAQABAoIBAG0ygSWObMeINFfm
rMuLxPGmy8MU0qqsqJCQ0Ssx0W66Q7u0L5oh3s0f0o0xvmxInU7//3A1aHi1G9FI
UYROTVD1LqPQGhBxSVA15saOBvvYHfNYDklWMorWtnQdSYTGWF5Y2/JcCiBhe1f4
3kHIzLfmnrJl2yRaMblPMf2ODRMHQ+rejFfGi5KbcoP1jJ4U1RFKbMmC0cT7myQ8
+POSXN62yFUuTU3QilatTB/EDZ6J4L4KIebQQT5ubwJF5fCK6BFtOhuuA44XfVSb
cxI5zOpZSK8bgLL99LoSVrSCQPHrbZpApzLeK42chT4fDdPF50GyHhRzHIS/BPLs
PjEwziUCgYEA9o6LspqHU3nYfnGe2Z1B1/CvRT3je+gNOAu5ttvga+iSxdCg/cA2
K6bqCZbK5pbwj16lessJfz1mSPYttyYZl4UK4wEu4faoTkldEg0TUvRmdj/RFShB
H50dotEmAk4UWVbpw/OlLsLVWnV44xZNAKY5w5JVUr4YRg0BYIhGYbcCgYEA8UZ8
PlDub1g7iRcCFrhxv5+5ZA8l+0rKt5FaVRPI0OHWkivoYzN6L0zXTP2OBX5iy4YG
```

```
LuQ0Sk27qrmZ9TLqToNuw9csqbg+isLSPA0dbPOPBPb72wAN6ebhMORSBMECFPNF
9te92N4rT6iUYSY8q9litMFT9s14LVmZjoMnXy8CgYAj9kCsCVwJsfEA6GOqDATp
kUKPT+qZTQx4i0VIRaPjOWYHEloZCOsdzNfAFE19+rAVyVFLqse01mjP5ZBfcWA0
OISQ/cAv10FPQeYgVuXlqJ41SzOc6WUuTkVfVTA9D9Rp/4JTQXtraaGi8xVx0fPj
T1uHihWF3xI9TXJQ+S+C2wKBgCXS6PNT+K0X5e1t3/Pz4lEqFwQqo8erR/BTJxgj
S94DKYIsw/eZQFRd9XqqXTNbRt2lGx8Kw2/Kk9sF0a7w1m6MiDbHascIjTVvqUmk
vLIx1H/wwDbq4UOD4FWr1XfNUig6owM5exeebKfGQ8yfE+/U2nZ/wEv2lhp16269
/NQdAoGAbJhnafoaso9s5Pv2Z60olxGUkDRWiFFCC3opujZ0alrJAvTfGiD9grLY
S9v9L4bdVn+gf48imqoHp9cTM2qHu43mpXNYPTS6subxp5y1eqiDhB5tyaPZM12C
FxSnygzJnOzG3ePDWw3UzGKrHIYhk/0gPhvvE2F5GP91jxoKSfE=
-----END RSA PRIVATE KEY-----
```

An example of what the public key file should look like:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA6GAUJmZyLhv6lMTYglg7
oQkuMQgvixZhFWIG7sbBzVdTnXrHNA6Jus/cELGmyl46ImhxHkGXXP0Ja9sVIJM+
lCIC6urX+tdCJ61mCiofuCwrwKRmyYyBUV1m0vxUxXh3RgJ/BmzqFVwgCCXj6MAb
I0fzG4K6EZVvNXLdVbXkGHt1dAtvIZCr5WFJbASAnJpBvRNzWA5bDyT6LW8g1HtZ
C6Uk6gesrgMh5aEXpWXTjtvD9miMF3K4xCl4VcV/Q68f3a011gGU2C6WlOgsIF2U
gf28YyY2ddppFaoHgtZLieqLkFI0kY9SgY6ZZeHw/mIMY1wyKEvryY4WZmyGqw/Z
mQIDAQAB
-----END PUBLIC KEY-----
```

A.4 Updating the Keys

The supplied private/public keys supplied with the secure image project should be updated with the new files. These new files will be used by the *secure_image_postbuild.bat* script during the build process to generate the application encrypted digital signature. Perform the following substitutions.

- rsa_private_generated.txt → /security/secure_image/src/scripts/keys/rsa_private.txt
- rsa_public_generated.txt → /security/secure_image/src/scripts/keys/rsa_public.txt

A.5 Installing the Public Key

The final step to updating the public key in the Secure Image is to copy the code in the generated *rsa_to_c_generated.txt* file to the *cy_si_keystorage.c* source file, which is part of the Secure Image project. An example of this file after being updated is shown below. The replacement code from the generated key data is shown below in *green italics*.

```
/***************************************************************************//**
* \file cy_si_keystorage.c
* \version 1.0
*
* \brief
* Secure key storage for the secure image.
*
********************************************************************************
* \copyright
* Copyright 2017, Cypress Semiconductor Corporation. All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*******************************************************************************/

#include "cy_si_keystorage.h"

#if defined(__cplusplus)
extern "C" {
#endif

/** Secure Key Storage (Note: Ensure that the alignment matches the Protection unit configuration)
*/
```

```
CY_ALIGN(1024)                         __USED                        const                      uint8_t
CySecureKeyStorage[CY_SI_SECURE_KEY_ARRAY_SIZE][CY_SI_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_si_stc_public_key_t cy_publicKey =
{
    .objSize = sizeof(cy_si_stc_public_key_t),
    .signatureScheme = CY_SI_PUBLIC_KEY_RSA_2048,
    .publicKeyStruct =
    {
        .moduloAddr          = (uint32_t)&(SFLASH->PUBLIC_KEY) + offsetof(cy_si_stc_public_key_t,
moduloData),
        .moduloSize         = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_MODULOLENGTH,
        .expAddr             = (uint32_t)&(SFLASH->PUBLIC_KEY) + offsetof(cy_si_stc_public_key_t,
expData),
        .expSize            = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_EXPLENGTH,
        .barrettAddr         = (uint32_t)&(SFLASH->PUBLIC_KEY) + offsetof(cy_si_stc_public_key_t,
barrettData),
        .inverseModuloAddr   = (uint32_t)&(SFLASH->PUBLIC_KEY) + offsetof(cy_si_stc_public_key_t,
inverseModuloData),
        .rBarAddr            = (uint32_t)&(SFLASH->PUBLIC_KEY) + offsetof(cy_si_stc_public_key_t,
rBarData),
    },
/* Replaced key data from this point */
.moduloData =
{
    0x99u, 0xD9u, 0x0Fu, 0xABu, 0x86u, 0x6Cu, 0x66u, 0x16u,
    0x8Eu, 0xC9u, 0xEBu, 0x4Bu, 0x28u, 0x32u, 0x5Cu, 0x63u,
    0x0Cu, 0x62u, 0xFEu, 0xF0u, 0xE1u, 0x65u, 0x99u, 0x8Eu,
    0x81u, 0x52u, 0x8Fu, 0x91u, 0x34u, 0x52u, 0x90u, 0x8Bu,
    0xEAu, 0x89u, 0x4Bu, 0xD6u, 0x82u, 0x07u, 0xAAu, 0x15u,
    0x69u, 0xDAu, 0x75u, 0x36u, 0x26u, 0x63u, 0xBCu, 0xFDu,
    0x81u, 0x94u, 0x5Du, 0x20u, 0x2Cu, 0xE8u, 0x94u, 0x96u,
    0x2Eu, 0xD8u, 0x94u, 0x01u, 0xD6u, 0x35u, 0xADu, 0xDDu,
    0x1Fu, 0xAFu, 0x43u, 0x7Fu, 0xC5u, 0x55u, 0x78u, 0x29u,
    0xC4u, 0xB8u, 0x72u, 0x17u, 0x8Cu, 0x68u, 0xF6u, 0xC3u,
    0xDBu, 0x8Eu, 0xD3u, 0x65u, 0xA5u, 0x17u, 0xA1u, 0xE5u,
    0x21u, 0x03u, 0xAEu, 0xACu, 0x07u, 0xEAu, 0x24u, 0xA5u,
    0x0Bu, 0x59u, 0x7Bu, 0xD4u, 0x20u, 0x6Fu, 0x2Du, 0xFAu,
    0x24u, 0x0Fu, 0x5Bu, 0x0Eu, 0x58u, 0x73u, 0x13u, 0xBDu,
    0x41u, 0x9Au, 0x9Cu, 0x80u, 0x04u, 0x6Cu, 0x49u, 0x61u,
    0xE5u, 0xABu, 0x90u, 0x21u, 0x6Fu, 0x0Bu, 0x74u, 0x75u,
    0x7Bu, 0x18u, 0xE4u, 0xB5u, 0x55u, 0xDDu, 0x72u, 0x35u,
    0x6Fu, 0x95u, 0x11u, 0xBAu, 0x82u, 0x1Bu, 0xF3u, 0x47u,
    0x23u, 0x1Bu, 0xC0u, 0xE8u, 0xE3u, 0x25u, 0x08u, 0x20u,
    0x5Cu, 0x15u, 0xEAu, 0x6Cu, 0x06u, 0x7Fu, 0x02u, 0x46u,
    0x77u, 0x78u, 0xC5u, 0x54u, 0xFCu, 0xD2u, 0x66u, 0x5Du,
    0x51u, 0x81u, 0x8Cu, 0xC9u, 0x66u, 0xA4u, 0xC0u, 0x2Bu,
    0x2Cu, 0xB8u, 0x1Fu, 0x2Au, 0x0Au, 0x66u, 0xADu, 0x27u,
    0x42u, 0xD7u, 0xFAu, 0xD7u, 0xEAu, 0xEAu, 0x02u, 0x22u,
    0x94u, 0x3Eu, 0x93u, 0x20u, 0x15u, 0xDBu, 0x6Bu, 0x09u,
    0xFDu, 0x5Cu, 0x97u, 0x41u, 0x1Eu, 0x71u, 0x68u, 0x22u,
    0x3Au, 0x5Eu, 0xCAu, 0xA6u, 0xB1u, 0x10u, 0xDCu, 0xCFu,
    0xBAu, 0x89u, 0x0Eu, 0x34u, 0xC7u, 0x7Au, 0x9Du, 0x53u,
    0x57u, 0xCDu, 0xC1u, 0xC6u, 0xEEu, 0x06u, 0x62u, 0x15u,
    0x61u, 0x16u, 0x8Bu, 0x2Fu, 0x08u, 0x31u, 0x2Eu, 0x09u,
    0xA1u, 0x3Bu, 0x58u, 0x82u, 0xD8u, 0xC4u, 0x94u, 0xFAu,
    0x1Bu, 0x2Eu, 0x72u, 0x66u, 0x26u, 0x14u, 0x60u, 0xE8u,
},
.expData =
{
    0x01u, 0x00u, 0x01u, 0x00u,
},
.barrettData =
{
    0x76u, 0x86u, 0x68u, 0x89u, 0xC2u, 0x0Fu, 0xEAu, 0x4Eu,
    0x01u, 0x49u, 0x94u, 0x83u, 0x00u, 0x6Cu, 0x06u, 0x32u,
    0x5Du, 0x2Bu, 0xC6u, 0x52u, 0x9Bu, 0xABu, 0x02u, 0x72u,
    0x00u, 0x73u, 0xF6u, 0x3Du, 0x89u, 0x6Au, 0x93u, 0x76u,
    0xA9u, 0x72u, 0x77u, 0x15u, 0xDEu, 0x4Cu, 0x99u, 0x4Bu,
    0xE6u, 0x67u, 0x2Du, 0xA9u, 0x80u, 0x85u, 0x78u, 0xB8u,
    0xF1u, 0xD3u, 0x2Au, 0x5Du, 0x22u, 0x7Bu, 0xDFu, 0xB2u,
    0x1Du, 0x2Du, 0xBCu, 0x91u, 0x94u, 0x1Du, 0x92u, 0x5Bu,
    0xF4u, 0x4Cu, 0xA0u, 0x6Fu, 0xC7u, 0x88u, 0x1Fu, 0x8Bu,
    0x2Du, 0x3Eu, 0xF1u, 0xA9u, 0xC6u, 0x14u, 0x12u, 0x3Du,
    0x59u, 0x3Cu, 0x8Cu, 0xF6u, 0x52u, 0xEDu, 0x4Bu, 0x27u,
    0x23u, 0xEDu, 0x20u, 0x10u, 0x25u, 0x54u, 0xAFu, 0x39u,
    0xE3u, 0x97u, 0xA4u, 0xC4u, 0x1Fu, 0x61u, 0xB3u, 0x05u,
    0xDBu, 0x47u, 0x21u, 0x68u, 0xE3u, 0x47u, 0x5Eu, 0xA1u,
    0xA1u, 0x0Au, 0xDDu, 0xD5u, 0x76u, 0x4Eu, 0x6Eu, 0x26u,
    0xEFu, 0xE7u, 0x49u, 0x83u, 0xB0u, 0x21u, 0x0Fu, 0xC6u,
    0x90u, 0xA0u, 0x05u, 0xF1u, 0x97u, 0x60u, 0x3Fu, 0x09u,
    0x05u, 0x6Fu, 0x87u, 0x4Fu, 0x7Eu, 0xE0u, 0xAEu, 0x0Au,
    0x43u, 0x02u, 0xB4u, 0xBFu, 0xD1u, 0x75u, 0xBAu, 0x6Du,
    0xA9u, 0x0Fu, 0x48u, 0xE9u, 0x3Bu, 0x31u, 0xD5u, 0x42u,
    0xD6u, 0xF3u, 0xBEu, 0x88u, 0x9Fu, 0x07u, 0x05u, 0xC5u,
    0xD6u, 0x85u, 0x90u, 0xC8u, 0x37u, 0x9Fu, 0x08u, 0x94u,
    0x03u, 0xBDu, 0xA9u, 0x90u, 0xC1u, 0x99u, 0xC7u, 0xAAu,
    0xAFu, 0x96u, 0xF2u, 0x8Au, 0x11u, 0x9Au, 0x17u, 0x39u,
    0x6Fu, 0x6Eu, 0x7Fu, 0xAEu, 0x8Au, 0x93u, 0xB9u, 0x0Cu,
    0xD7u, 0x29u, 0xA5u, 0xA9u, 0x11u, 0x9Au, 0x96u, 0xBFu,
    0x7Du, 0xD9u, 0xB4u, 0x1Cu, 0x0Fu, 0xD2u, 0xDAu, 0xB4u,
    0x86u, 0x7Eu, 0x49u, 0x13u, 0x1Bu, 0x55u, 0xFBu, 0xECu,
    0x1Eu, 0x70u, 0xD6u, 0xEBu, 0x7Eu, 0x14u, 0x6Eu, 0x73u,
    0x21u, 0xA9u, 0x5Eu, 0x53u, 0xF3u, 0xCCu, 0x33u, 0xE9u,
    0xD2u, 0xBFu, 0xB2u, 0xAAu, 0xF3u, 0x22u, 0xE8u, 0xFEu,
    0xCBu, 0x23u, 0x76u, 0x70u, 0x36u, 0xCAu, 0x06u, 0x1Au,
    0x01u, 0x00u, 0x00u, 0x00u,
},
.inverseModuloData =
{
    0x57u, 0x95u, 0x0Fu, 0x84u, 0xC3u, 0x03u, 0x27u, 0x6Fu,
    0x0Eu, 0xAAu, 0x6Fu, 0xE1u, 0xC4u, 0xADu, 0x95u, 0x66u,
    0xCAu, 0x98u, 0xA5u, 0xABu, 0x23u, 0x4Eu, 0x2Eu, 0xA9u,
    0xDBu, 0x84u, 0xE2u, 0x47u, 0x7Fu, 0x87u, 0xCCu, 0xC9u,
    0x0Cu, 0xD7u, 0xADu, 0xB2u, 0x3Du, 0xB0u, 0xECu, 0xB6u,
    0x2Fu, 0x3Cu, 0xAAu, 0xAFu, 0x77u, 0xB2u, 0xD7u, 0xDDu,
    0x5Fu, 0x83u, 0xADu, 0x7Eu, 0xE8u, 0xB1u, 0x60u, 0x66u,
    0x48u, 0x73u, 0x73u, 0x9Eu, 0xBFu, 0x50u, 0x74u, 0xE2u,
    0xBCu, 0x80u, 0xB9u, 0x5Cu, 0xA4u, 0xEBu, 0xADu, 0xF6u,
    0x32u, 0xCBu, 0x78u, 0x2Eu, 0x28u, 0x14u, 0x3Eu, 0xBCu,
    0x39u, 0x92u, 0x29u, 0x39u, 0xACu, 0xEEu, 0x4Au, 0x6Fu,
    0x06u, 0x3Cu, 0xDBu, 0x29u, 0xFBu, 0xD3u, 0x34u, 0x22u,
    0x14u, 0xD8u, 0xC5u, 0xBFu, 0x03u, 0xF3u, 0x52u, 0xC4u,
    0xBEu, 0xF9u, 0x99u, 0xB7u, 0xB1u, 0x58u, 0x20u, 0x85u,
    0xD8u, 0xD8u, 0xEDu, 0x91u, 0x4Cu, 0x90u, 0x35u, 0xF1u,
    0xDCu, 0x2Au, 0x82u, 0xCFu, 0xACu, 0x2Cu, 0x76u, 0x2Fu,
    0xD3u, 0x7Cu, 0xF0u, 0x57u, 0x35u, 0xBFu, 0x5Eu, 0x80u,
    0xA7u, 0xF0u, 0x22u, 0xE8u, 0xCAu, 0x80u, 0x4Au, 0xF7u,
    0xD2u, 0x72u, 0x1Cu, 0x9Eu, 0xB7u, 0x87u, 0xD9u, 0x80u,
    0x19u, 0x40u, 0xC6u, 0xBFu, 0x3Cu, 0x5Fu, 0x94u, 0x4Cu,
    0xDAu, 0x5Au, 0xD4u, 0x33u, 0xDDu, 0x4Cu, 0x45u, 0x89u,
    0xCFu, 0x35u, 0xB2u, 0xB5u, 0x36u, 0x6Fu, 0x39u, 0xC0u,
    0xC8u, 0x4Au, 0x07u, 0x5Au, 0x9Eu, 0xA1u, 0xBCu, 0xE5u,
    0x5Eu, 0x45u, 0x75u, 0xC4u, 0x16u, 0xABu, 0xD6u, 0x3Du,
    0x4Au, 0xBDu, 0x9Du, 0x2Du, 0x5Fu, 0x12u, 0x70u, 0x55u,
    0xEFu, 0xAAu, 0x16u, 0x90u, 0xC2u, 0x64u, 0xFDu, 0xFBu,
    0x9Fu, 0xD6u, 0x77u, 0xFDu, 0x42u, 0x23u, 0x27u, 0x39u,
    0xDEu, 0xA8u, 0x07u, 0x5Fu, 0x9Fu, 0xBAu, 0x95u, 0x71u,
    0x5Du, 0x3Cu, 0x45u, 0x4Au, 0x49u, 0xC7u, 0x99u, 0x47u,
    0xEEu, 0x09u, 0xC7u, 0x4Eu, 0x12u, 0xD5u, 0x36u, 0x3Bu,
    0x89u, 0xE0u, 0xFCu, 0x8Eu, 0x92u, 0xC7u, 0x52u, 0x8Au,
    0x82u, 0xA7u, 0x91u, 0xAFu, 0x62u, 0x39u, 0xF0u, 0x61u,
},
.rBarData =
{
    0x67u, 0x26u, 0xF0u, 0x54u, 0x79u, 0x93u, 0x99u, 0xE9u,
    0x71u, 0x36u, 0x14u, 0xB4u, 0xD7u, 0xCDu, 0xA3u, 0x9Cu,
    0xF3u, 0x9Du, 0x01u, 0x0Fu, 0x1Eu, 0x9Au, 0x66u, 0x71u,
    0x7Eu, 0xADu, 0x70u, 0x6Eu, 0xCBu, 0xADu, 0x6Fu, 0x74u,
    0x15u, 0x76u, 0xB4u, 0x29u, 0x7Du, 0xF8u, 0x55u, 0xEAu,
    0x96u, 0x25u, 0x8Au, 0xC9u, 0xD9u, 0x9Cu, 0x43u, 0x02u,
    0x7Eu, 0x6Bu, 0xA2u, 0xDFu, 0xD3u, 0x17u, 0x6Bu, 0x69u,
    0xD1u, 0x27u, 0x6Bu, 0xFEu, 0x29u, 0xCAu, 0x52u, 0x22u,
    0xE0u, 0x50u, 0xBCu, 0x80u, 0x3Au, 0xAAu, 0x87u, 0xD6u,
    0x3Bu, 0x47u, 0x8Du, 0xE8u, 0x73u, 0x97u, 0x09u, 0x3Cu,
    0x24u, 0x71u, 0x2Cu, 0x9Au, 0x5Au, 0xE8u, 0x5Eu, 0x1Au,
    0xDEu, 0xFCu, 0x51u, 0x53u, 0xF8u, 0x15u, 0xDBu, 0x5Au,
    0xF4u, 0xA6u, 0x84u, 0x2Bu, 0xDFu, 0x90u, 0xD2u, 0x05u,
    0xDBu, 0xF0u, 0xA4u, 0xF1u, 0xA7u, 0x8Cu, 0xECu, 0x42u,
    0xBEu, 0x65u, 0x63u, 0x7Fu, 0xFBu, 0x93u, 0xB6u, 0x9Eu,
    0x1Au, 0x54u, 0x6Fu, 0xDEu, 0x90u, 0xF4u, 0x8Bu, 0x8Au,
    0x84u, 0xE7u, 0x1Bu, 0x4Au, 0xAAu, 0x22u, 0x8Du, 0xCAu,
    0x90u, 0x6Au, 0xEEu, 0x45u, 0x7Du, 0xE4u, 0x0Cu, 0xB8u,
    0xDCu, 0xE4u, 0x3Fu, 0x17u, 0x1Cu, 0xDAu, 0xF7u, 0xDFu,
    0xA3u, 0xEAu, 0x15u, 0x93u, 0xF9u, 0x80u, 0xFDu, 0xB9u,
    0x88u, 0x87u, 0x3Au, 0xABu, 0x03u, 0x2Du, 0x99u, 0xA2u,
    0xAEu, 0x7Eu, 0x73u, 0x36u, 0x99u, 0x5Bu, 0x3Fu, 0xD4u,
    0xD3u, 0x47u, 0xE0u, 0xD5u, 0xF5u, 0x99u, 0x52u, 0xD8u,
    0xBDu, 0x28u, 0x05u, 0x28u, 0x15u, 0x15u, 0xFDu, 0xDDu,
    0x6Bu, 0xC1u, 0x6Cu, 0xDFu, 0xEAu, 0x24u, 0x94u, 0xF6u,
    0x02u, 0xA3u, 0x68u, 0xBEu, 0xE1u, 0x8Eu, 0x97u, 0xDDu,
    0xC5u, 0xA1u, 0x35u, 0x59u, 0x4Eu, 0xEFu, 0x23u, 0x30u,
    0x45u, 0x76u, 0xF1u, 0xCBu, 0x38u, 0x85u, 0x62u, 0xACu,
    0xA8u, 0x32u, 0x3Eu, 0x39u, 0x11u, 0xF9u, 0x9Du, 0xEAu,
    0x9Eu, 0xE9u, 0x74u, 0xD0u, 0xF7u, 0xCEu, 0xD1u, 0xF6u,
    0x5Eu, 0xC4u, 0xA7u, 0x7Du, 0x27u, 0x3Bu, 0x6Bu, 0x05u,
    0xE4u, 0xD1u, 0x8Du, 0x99u, 0xD9u, 0xEBu, 0x9Fu, 0x17u,
    },
  /* End of key data */
};

#if defined(__cplusplus)
}
#endif

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

Appendix B DAP Settings and Lifecycle State (eFuse)

As mentioned previously, memory stored in eFuse can be changed only from 0 to 1, and never the other way. This provides a secure method to control access to PSoC 6 MCUs without a third party being able to reverse the action. Programming eFuse requires the device supply, V_{DDD} set to 2.5 V. For more information, see the PSoC 6 MCU Programming Specifications.

Table 5 shows the usage of PSoC 6 MCU eFuse bytes. The values at offsets 0x27 through 0x2B are the locations of interest for changing the DAP Settings and Lifecycle State.

Table 5. eFuse Locations

Offset	# of Bytes	Name	Description
0x00	20	Reserved	Reserved for PSoC 6 MCU system usage
0x14	16	Secure_HASH	Secure objects 128-bit hash
0x24	2	Reserved	Reserved
0x26	1	Secure_HASH_ZEROS	Number of zeros in Secure_HASH
0x27	2	DEAD_ACCESS_RESTRICT	Access restrictions in Dead lifecycle stage
0x29	2	SECURE_ACCESS_RESTRICT	Access restrictions in Secure lifecycle stage
0x2B	1	LIFECYCLE_STAGE	Normal, Secure, and Secure with Debug fuse bits
0x2C	16	Factory_HASH	Factory_HASH value
0x40	64	CUSTOMER_DATA	Customer data

B.1 DAP Settings

The SECURE_ACCESS_RESTRICT0 and DEAD_ACCESS_RESTRICT0 registers (all default to 0) are used to set access restrictions for Dead and Secure modes. Table 6 shows the format used to set the DAP settings for these modes.

Table 6. Debug Access Port (DAP) Settings

Bits	Field Name	Description
0	CM0_DISABLE	A ‘1’ indicates that this device does not allow access to the CM0+ debug port.
1	CM4_DISABLE	A ‘1’ indicates that this device does not allow access to the CM4 debug port.
2	SYS_DISABLE	A ‘1’ indicates that this device does not allow access to the system debug port.
3	SYS_AP_MPU_ENABLE	A ‘1’ indicates that the MPU on the system debug port must be programmed and locked according to the settings in the next four fields. The SYS_DISABLE bit must be left at ‘0’ for this setting to matter. If the SYS_DISABLE bit is set to ‘1’, then the next four fields are invalid.
5:4	SFLASH_ALLOWED	This field indicates what portion of the flash supervisory region is accessible through the system debug port. Only a portion of SFLASH starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MPU_ENABLE=1. Encoding is as follows: 0x0: entire region 0x1: 1/2 0x2: 1/4 0x3: nothing
7:6	MMIO_ALLOWED	This field indicates what portion of the MMIO region is accessible through the system debug port. Valid only if SYS_DISABLE=0 and SYS_AP_MPU_ENABLE=1. Encoding is as follows: 0x0: All MMIO registers 0x1: Only IPC 0, 1, and 2 MMIO registers accessible (system calls) 0x2 or 0x3: No MMIO access

Table 7. SECURE_ACCRESS_RESTRICT1 , DEAD_ACCESS_RESTRICT1 (All Default to 0)

Bits	Name	Description
2:0	FLASH_ALLOWED	This field indicates what portion of the flash main region is accessible through the system debug port. Only a portion of flash starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MPU_ENABLE=1. Encoding is as follows: 0x0: entire region 0x1: 7/8 0x2: 3/4 0x3: 1/2 0x4: 1/4 0x5: 1/8 0x6: 1/16 0x7: nothing
5:3	SRAM_ALLOWED	This field indicates what portion of the SRAM region is accessible through the system debug port. Only a portion of SRAM starting at the bottom of the area is exposed. Valid only if SYS_DISABLE=0 and SYS_AP_MPU_ENABLE=1. 0x0: entire region 0x1: 7/8 0x2: 3/4th 0x3: 1/2 0x4: 1/4th 0x5: 1/8th 0x6: 1/16th 0x7: nothing
6	SMIF_XIP_ALLOWED	This field indicates what portion of XIP is accessible through the system access port. 0x0: Entire region 0x1: Nothing
7	DIRECT EXE DISABLE	Disables Direct Execute system call functionality (implemented in software).

B.2 Lifecycle Settings

There are three bits that define the Lifecycle stages NORMAL, SECURE_WITH_DEBUG, and SECURE. From the factory, the NORMAL bit will be set but the other bits will be 0.

Bits	Field Name	Description
0	NORMAL	Used by ROM boot to detect Normal stage
1	SECURE_WITH_DEBUG	Used by ROM boot to detect Secure with Debug stage
2	SECURE	Used by ROM boot to detect Secure stage
3	RMA	Used by ROM boot to detect RMA stage

Appendix C Transition to RMA Mode

A device cannot transition to RMA stage without access to the device unique ID and the customer's private key that is paired with a public key stored and authenticated in SFlash. The customer must implement at least two special commands that can be sent from outside the device via UART, SPI, I^{2}C, and so forth. The first command is to read the internal unique device ID, and the other is to invoke the transition to RMA.

The special commands can be implemented in two ways. The first method would be to include these special commands as part of the existing application. Another, a safer and secure way to do this is to have a special device code image that supports only the required commands. This way when bootloading this special code image, all proprietary and sensitive data can be erased at the same time. Also, a special code image allows an easy implementation of a standard interface such as a UART to implement the communication needed to invoke the commands.

After this infrastructure is in place, do the following to the transition the device to RMA mode.

1. Erase all sensitive or proprietary code stored in the device. This may be performed with a special command or with a special code image described above. Erase the flash at least four times to ensure there is no way to detect any residual code. The public key stored in SFlash must remain because it is used to transition to RMA mode and to open RMA mode later by Cypress.
2. Read the device unique ID stored in the devices SFlash. This can be done by invoking system call 0x1F (Read Unique ID), then sending ID out via the communication interface.
3. Use the unique ID and the customer's private key that is paired with the public key stored internally in SFlash to generate a certificate. This is the same method that is used to sign code as described in 4 Code Signing and Validation above, using the same private/public key pairs. The format of the certificate is shown below.

Figure 14. RMA Certificate Format

Object size in bytes of the certificate 0x00000114 (4 bytes)
Command ID 0x28000000 (4 bytes)
Unique ID (10 bytes)
Zero Padding 0x0000 (2 bytes)
Digital Signature (256 bytes)

4. Send a command to the device that includes the above certificate. The user must implement code to accept this certificate to invoke the transition to RMA system call (0x28) and pass the certificate as its parameter. The device power supply must be at 2.5V before performing this step, since the RMA eFuse will be programmed. (Any programming of eFuse bits requires the device power supply to be 2.5 volts.)
5. After the device is reset or power cycled, it will sit in a mode awaiting a single command from the debug port to open RMA mode (System call 0x29) along with the same certificate that was used to invoke transition to RMA in the first place. It will have all the same access modes as Virgin mode, but a debugger/programmer will need to invoke the system call open RMA every time the device is reset or power cycled. The device in this state is unusable except for failure analysis.

After you have performed the steps described above, the device and certificate can be sent to Cypress to allow failure analysis. Even if this special certificate falls into the wrong hands, it will be valid only for the one part for which it was generated and will be useless for any other part. Because the device has been erased of all sensitive and proprietary data, there is no risk in Cypress or other parties extracting usable code or data once stored in flash.

Appendix D Protection Units

This appendix provides a brief description of how the protection units work and how to use them. For a more detailed description of protections units, read section 8 “Protection Units” in the PSoC 6 TRM (Technical Reference Manual).

Any time one of the bus masters initiates a memory read, write, or code execution, the operation is checked by any protection unit that is enabled. If any bus operation violates the rules set by the protection unit attributes, a bus fault will occur. More information on protection context and the other protection unit attributes is provided in more detail later in this appendix.

D.1 Bus Masters

In PSoC6 MCUs, a bus master is any block that can directly access SRAM or flash without the aid of another bus master. There are at least six bus masters in the PSoC6 family of MCUs, and may be more in future devices. In this document, a mention of bus master includes the CM0+ and CM4 processors. Table 8 shows the bus masters and important configurations registers.

Table 8. List of Bus Masters in the PSoC 6 Family of MCUs

Master #	Bus Master	Master Protection Context Control Register	Master Control Register
0	CM0+ Processor	PROT_SMPU_MS0_CTL	PROT_MPU0_MS_CTL
1	Crypto Block	PROT_SMPU_MS1_CTL	PROT_MPU1_MS_CTL
2	DataWire 0	PROT_SMPU_MS2_CTL	Inherits settings from CPU
3	DataWire 1	PROT_SMPU_MS3_CTL	Inherits settings from CPU
14	CM4 Processor	PROT_SMPU_MS14_CTL	PROT_MPU14_MS_CTL
15	Test Controller	PROT_SMPU_MS15_CTL	PROT_MPU15_MS_CTL

The Master Protection Context Control Registers are key to making protection units function. They are used to configure each bus master with the appropriate attributes that the protection units use to determine access. The protection unit library contains a function Cy_Port_ConfigBusMaster() that can be used to configure this register.

The Master Control Register for each bus master controls the active protection context. The DataWire blocks don’t have an associated register, because they inherit their attributes from the CPU. The protection unit library includes a function Cy_Prot_SetActivePC() to set the active protection context for the bus master.

D.2 Protection Unit Configuration

Cypress provides a library to simplify the programming of protection units. Read the documentation for the protection unit (Prot) API. This document can be accessed through PSoC Creator under the “Help” menu, Help/Documentation/Peripheral Driver Library. In the document, select “Protection Unit (Prot)” under the “API Reference” section.

Protection units can be programmed to allow only specific CPUs and other bus masters to access only predefined memory segments and peripherals. If a bus master attempts to use a section of memory not allowed by the protection unit, a bus fault will occur. Because the two CPUs (CM4 and CM0+) share the same memory space in the PSoC 6 MCUs, the protection units allow you to guarantee that one CPU cannot affect the memory or peripherals allocated to the other. Not all memory or peripherals must be allocated to just one CPU; by default, the entire memory space is shared between the two CPUs and sections can remain that way if desired.

Protection units can also be used to isolate memory for different tasks that run concurrently in an RTOS. Different sections of a boot process may also be another example of securing sections of memory. For example, you may want your bootloader to have access to write flash in the user application area, but deny the user application from writing any flash. Protection units can be configured to do just that.

There are three main types of protection units available in the PSoC 6 MCU family:

- MPU (Memory Protection Unit)
- SMPU (Shared Memory Protection Unit)
- PPU (Peripheral Protection Unit).

There are two types of MPUs:

- The ones that are part of each CPU (CM0+ and CM4)
- The other MPUs in the system that are paired with other bus masters such as the Crypto block.

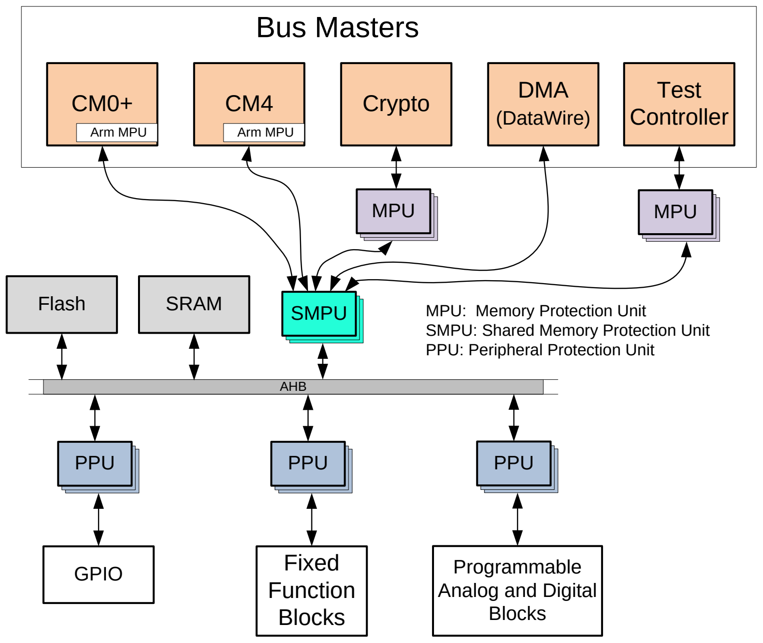
The MPUs are really meant to be dynamic and changed often during runtime such as with an RTOS. They are meant to be changed based on firmware tasks not resource management between two CPUs and not specifically for security purposes. Because of this, this section will concentrate on the SMPU and PPUs.

There are also two types of PPUs:

- Programmable – Programmable PPUs are very similar to the SMPUs but are dedicated to the peripheral address space
- Fixed – Fixed PPUs are dedicated to a specific peripheral block and cannot be used for any other purpose

See Figure 15, which shows how the protection units are virtually connected with respect to the PSoC 6 MCU memory, bus masters, and peripherals.

Figure 15. Protection Unit Functional Block Diagram


Bus Masters
CM0+
Arm MPU
CM4
Arm MPU
Crypto
DMA
(DataWire)
Test Controller
MPU
MPU
Flash
SRAM
SMPU
MPU: Memory Protection Unit
SMPU: Shared Memory Protection Unit
PPU: Peripheral Protection Unit
AHB
PPU
PPU
PPU
GPIO
Fixed Function Blocks
Programmable Analog and Digital Blocks

There are 16 SMPU structures in the SMPU block, so you can allocate several sections of memory as needed. SMPUs can be used for any memory area, even the peripheral memory space, but the PPUs should be used first.

The SMPUs are shared among the bus masters as the name implies. They are mainly used to restrict access of portions of memory to a specific bus master (CPUs) and/or specific tasks within a CPU application. The SMPU block consist of a set of 16 filters (structure) that can be configured individually. Each filter can limit access to one or more memory blocks to a specified group of bus masters and tasks. Although the SMPUs can limit access to peripherals as well as memory, you should use the SMPUs for memory and use the PPUs for peripherals as intended.

D.3 Master and Slave Protection Units

All SMPU and PPU protection units are paired with a master protection unit. The master protection unit is so that the SMPU or PPU can be controlled only by a specific bus master. The most secure way to do this is to configure all the SMPUs and PPUs with the master protection units before enabling the CM4, with the secure CPU (CM0+). This is what the Secure Image example project does that is defined in Chapter 6, Building a Secure System of this document.

D.4 Protection Unit Attributes

The SMPUs and PPUs can be configured by more than just which bus master can access a specified area of memory. There are several other attributes that can be specified. See Table 9 for all the attributes that can be defined.

Table 9. Protection Unit Attributes

Attribute	Full Name	Description
ADDR24[23:0]	Most significant 24 bits of memory region	This attribute sets the 24 most significant bits of the 32-bit address of the memory block that is being protected.
REGION_SIZE[15:0]	Memory Region Size	This attribute defines the size of the protected regions. It defines a region from 256 bytes to 4GB in powers of 2.
SUBREGION_DISABLE[7:0]	Enable/Disable Regions	Breaks the Memory Region into eight equal blocks and allows the user to enable/disable protection to any or all of these blocks.
PC_MASK_15_TO_1[15]	Protection Context Mask	The PC (Protection Context) mask allows the memory systems defined in the protection units to be divided into 16 groups. This specific attribute defines masks for groups 1 to 15.
PC_MASK_0	Protection Context Mask for PC=0	Any bus master with the protection context set to “0” has access to all memory and peripherals.
NS	Non-Secure	0=> Secure, 1=>Non-Secure
PX	Privileged Execute Enabled	0=> Privileged Execution not allowed 1=> Privileged Execution allowed
PW	Privileged Write Enabled	0=> Privileged Write not allowed 1=> Privileged Write allowed
PR	Privileged Read Enabled	0=> Privileged Read not allowed 1=> Privileged Read allowed
UX	User Execution Enabled	0 => User execute NOT allowed 1 => User execute allowed
UW	User Write Enabled	0 => User write NOT allowed 1 => User write allowed
UR	User Read Enabled	0 => User read NOT allowed 1 => User read allowed

The Privileged/User attributes (PX, PW, PR, UX, UW, UR) use the Privileged state of the CM0+ and CM4 to determine if the memory region is protected or not. See the Arm CM0+ and CM4 manuals for more information on how Privileged and User modes operate.

The NS (Non-Secure/Secure) attribute determines which bus masters have access to secure memory and peripherals. In most cases, you will want to select the CM0+ as the Secure processor and the CM4 as the Non-Secure processor.

The PC (Protection Context) allows the designer to divide memory and peripherals space into 15 different regions, plus one super user mode, PC=0. For example, a simple system may divide the memory and peripherals into CM4 application (1), CM0+ application (2), and CM0+ bootloader space (3). Each space may include several regions, such as SRAM, flash, and peripherals. The PC=0 mode is special in that it supersedes all User/Privileged, and Secure/Non-Secure attributes. This mode (PC=0) is meant only for the designated secure CPU, which should be the CM0+.

There is a priority among the protections units. If there are 16 SMPUs 0 to 15, the highest number has the highest priority. For example, if SMPU[15] configured a section such that it was OK for the CM4 to read, but the SMPU[10] was configured so that the CM4 could not be read, SMPU[15] would override SMPU[10] and the CM4 could read that section of memory without a fault.

The Protection Unit (Prot) library includes several functions to configure SMPUs and PPUs. A few of the more important APIs are listed below.

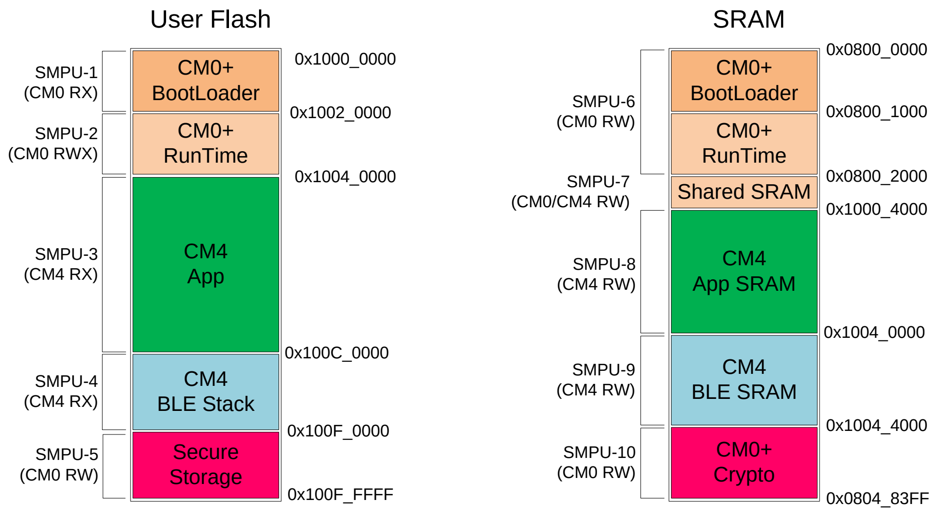
- **Cy_Prot_ConfigSmpuSlaveStruct():** Configure SMPU protection unit (slave)
- **Cy_Prot_ConfigSmpuMasterStruct():** Configure SMPU master, protects the slave SMPU
- **Cy_Prot_ConfigPpuProgSlaveStruct();** Configure PPU protection unit (slave)
- **Cy_Prot_ConfigPpuProgMasterStruct():** Configure PPU master, protects PPU slave

Note: You can block SRAM so that it can be read and written but not executed from. This is useful in blocking buffer overflow attacks. Applications or functions that take data from an external user such as Telnet or HTTP server, could be susceptible to this type of attack if the firmware isn’t rock solid.

D.5 Example Usage of SMPUs

Below is an example of how a designer may allocate the user flash and the SRAM. The bootloader will most likely have access to all flash because it will be tasked with updating the code. For each SMPU, the diagram also shows which CPU has access to that area. For example, SMPU-2 allows the CM0 to Read/Write/Execute (CM0 RWX) the user flash for that area. As far as the SRAM allocations, notice the “Shared SRAM” section. An SMPU is not really needed if you just want to share some SRAM between the two CPUs. But if you wanted to make sure there is no way code could execute from that SRAM, an SMPU may be configured to disallow code being executed in that space. If the CM4 is running an RTOS, an MPU could be used to divide part of the “CM4 App SRAM” into partitions for each task.

Figure 16. Example of Memory Allocation



D.6 Pre-Configured Protection Units

Some protection units are configured during the boot process and should not be reconfigured. These protection units are vital to providing a secure system and providing a reliable access to system call functions. The following table lists the protection units that must not be reconfigured by the user.

Table 10. Protection Units Used by the System

Protection Unit	Usage Description
SMPU 15	Read/write restriction for ROM private stack.
SMPU 14	Read/write restriction for ROM region.
PROG PPU 15	Write restriction for CPUSS AP_CTL, PROTECTION, CM0_NMI_CTL, DP_CTL and MBIST_CTL registers.
PROG PPU 14	Read/write restriction for CPUSS WOUNDING and CM0_PC0_HANDLER registers.
PROG PPU 13	Write restriction for FlashC FM_CTL.BOOKMARK register.
PROG PPU 12	Read/write restriction for eFuse region (excluding CUSTOMER_DATA).
PROG PPU 11	Write restriction for IPC 0, 1 and 2 during system calls.
PROG PPU 10	Read/write restriction for Crypto during system calls that use crypto operations.
PROG PPU 9	Read/write restriction for FM_CTL registers.

Related Documents

Application Notes	
AN210781	Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity
AN218241	PSoC 6 MCU Hardware Design Considerations
AN213924	PSoC 6 MCU Bootloader Software Development Kit (SDK) Guide
PSoC Creator Component Datasheets	
PSoC 6 MCU: PSoC 63 with BLE	Datasheet
PSoC 6 MCU: PSoC 62	Datasheet
Device Documentation	
PSoC 6 MCU: PSoC 63 with BLE	Architecture Technical Reference Manual
PSoC 6 MCU: PSoC 63 with BLE	Register Technical Reference Manual
PSoC 6 MCU: PSoC 62	Architecture Technical Reference Manual
PSoC 6 MCU: PSoC 62	Register Technical Reference Manual
Programming Spec	PSoC 6 MCU Programming Specifications
Development Kit (DVK) Documentation	
CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit	

About the Author and Contributors

Name:	Mark Hastings
Title:	Cypress Fellow
Background:	Mark Hastings has a BSEE from Washington State University. For more than 30 years he has worked with embedded systems and mixed signal designs.

Document History

Document Title: AN221111 - PSoC 6 MCU: Creating a Secure System

Document Number: 002-21111

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	6001412	MEH	05/03/2018	New Application Note.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709