# Getting started with PSOC™ Control C3M/P8, C3M/P7 MCUs security

## About this document

This application note provides information on how to get started with security on the PSC3M/P8, PSC3M/P7 MCUs out of the box. It contains guidance on how to provision and program a secure application. It introduces the security architecture of the PSOC™ Control MCU, consisting of an Arm® TrustZone®-enabled Cortex®-M33. This document provides a brief overview of the security features available on the device.

### Intended audience

This document is intended for customers who want to start with enabling secured boot and explore security feature implementation on PSC3M/P8, PSC3M/P7 MCU devices.

# Table of contents

**About this document** . . . 1

**Table of contents** . . . 2

**1 Overview of PSC3M/P8, PSC3M/P7 MCUs** . . . 4

1.1 Hardware security features . . . 4

1.2 Lifecycle security and controlled debug access . . . 5

1.3 Secure boot, code authentication, and firmware update . . . 5

1.4 Development tooling . . . 6

**2 Ease of development** . . . 8

2.1 Device lifecycle stages (LCS) . . . 8

2.1.1 Development LCS . . . 9

2.1.2 Production LCS . . . 9

2.1.3 RMA LCS . . . 10

2.2 Development flow . . . 10

2.2.1 Transfer of ownership . . . 10

2.2.2 Getting started . . . 10

2.2.2.1 Initialize the application directory . . . 11

2.2.2.2 Key generation . . . 12

2.2.2.3 OEM certificate creation . . . 13

2.2.2.4 Provisioning and the policy file . . . 15

2.2.3 Enable secure boot . . . 16

2.2.4 Signing user application . . . 17

2.2.4.1 Postbuild signing using ModusToolbox™ application . . . 18

2.2.5 Transition to Production LCS . . . 20

2.2.6 Re-provisioning . . . 20

**3 Application structure** . . . 21

3.1 Basic application structure . . . 21

3.2 Application with Edge Protect Bootloader . . . 22

**4 Boot flow** . . . 23

4.1 Boot sequence . . . 23

4.2 SIMPLE_APP . . . 27

4.3 SECURE_APP . . . 27

4.4 DUAL_BANK_SIMPLE_APP . . . 28

4.5 DUAL_BANK_SECURE_APP . . . 29

**5 Code signing and verification** . . . 30

5.1 Code signing . . . 30

5.2 Code validation . . . 31

**6 Edge Protect Tools** . . . 32

## Table of contents


6.1 Overview . . . 32
6.2 Device communication . . . 33
6.3 Packaged assets . . . 34
6.4 Edge Protect Tools package for PSC3M/P8, PSC3M/P7 MCUs . . . 34

**7 Processing, memory, and peripheral isolation . . . 36**
7.1 Memory isolation . . . 37
7.2 Peripheral isolation . . . 38
7.2.1 Security aware peripheral protection controller (PPC) regions . . . 38

**8 Firmware upgrade . . . 40**
8.1 BootROM DFU serial interface . . . 40
8.2 User application (DFU MW) serial interface . . . 41

**9 Debug token . . . 43**
9.1 Debug token use cases . . . 43
9.2 Generating a debug certificate . . . 44
9.3 Generating RMA certificate . . . 46

**10 Edge Protect Bootloader . . . 48**
10.1 Boot flow . . . 48
10.2 Architecture . . . 48
10.3 Edge Protect Bootloader features . . . 49
10.3.1 Power fail-safe update . . . 50
10.3.2 Multi image authentication . . . 50
10.3.3 Upgrade by overwrite . . . 54
10.3.4 Upgrade by swap . . . 55
10.3.5 Secure boot . . . 56
10.3.6 Post-quantum cryptography signature schemes . . . 57
10.3.7 Secure firmware update . . . 58
10.3.8 Image encryption . . . 58
10.3.9 Downgrade prevention . . . 60
10.3.10 Dependency check . . . 61
10.3.11 Fault injection hardening . . . 61
10.3.12 Serial logging . . . 62

**11 Policy file . . . 63**

**A Default OEM provisioning policy . . . 65**

**References . . . 72**

**Glossary . . . 73**

**Revision history . . . 75**

**Trademarks . . . 76**

**Disclaimer . . . 77**

# 1 Overview of PSC3M/P8, PSC3M/P7 MCUs

The PSC3M/P8, PSC3M/P7 MCUs integrates three Arm® Cortex®-M33 cores: a main core with Arm® TrustZone® enabled, and two additional power cores programmable power control accelerator (PPCA) based on Cortex®-M33 without TrustZone® support. The device has SONOS flash, a range of peripherals, including high-performance programmable analog subsystem (HPPASS), coordinate rotation digital computer (CORDIC), ADC, UART, SPI, I2C, CAN FD, crypto accelerators, and several other blocks. The CM33 main core provides for both a secure processing environment (SPE) and a non-secure processing environment (NSPE). Although a single CPU is used, individual SPE and NSPE projects can be created. The BootROM is responsible for the initial secure boot and the root of trust (RoT). The main function of the BootROM is to configure the system information, authenticate the application, and transfer control to the application. Figure 1 shows the basic hardware cores and boot flow of the PSC3M/P8, PSC3M/P7 MCUs.

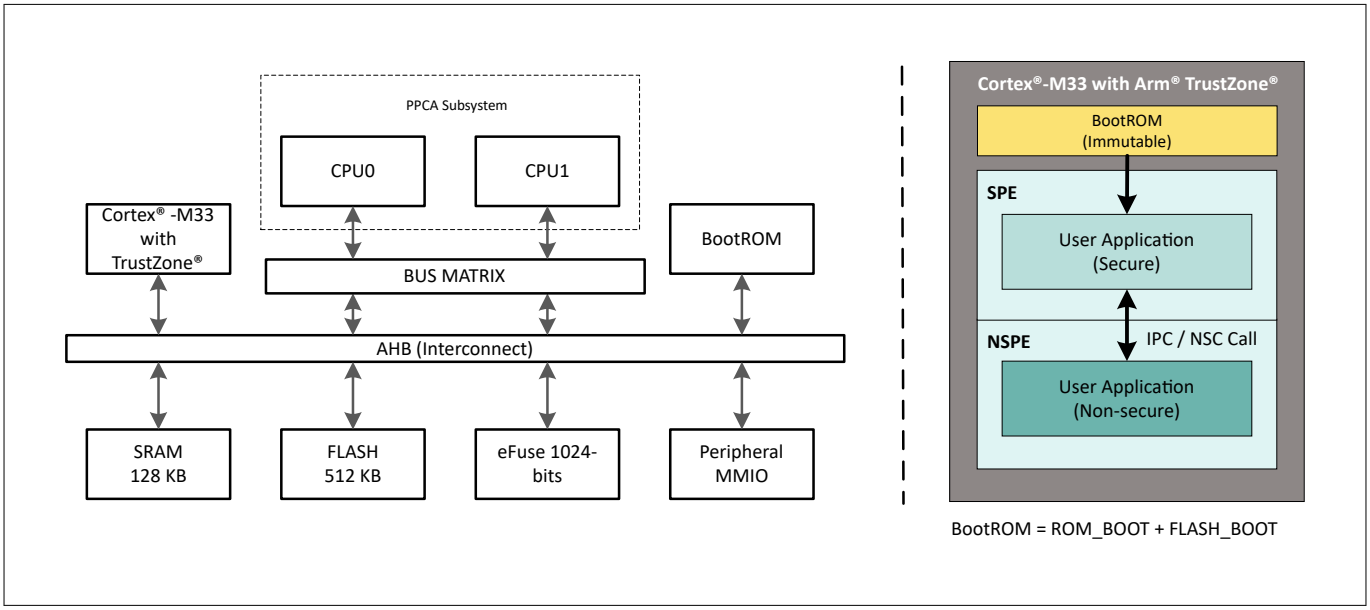



**Figure 1 Basic hardware blocks and boot flow**

The PSC3M/P8, PSC3M/P7 MCUs are certified to Platform Security Architecture Level 3 (PSA L3), reflecting a comprehensive, layered security architecture that spans hardware security primitives, a hardware-anchored secure boot framework, and controlled device lifecycle management. The security features described in this document are available across all variants of the family: PSC3M/P8, PSC3M/P7.

## 1.1 Hardware security features

**Table 1 Hardware security features**

| Feature | Description |
|---|---|
| Arm® TrustZone® for Cortex®-M33 | Hardware-enforced isolation between Secure and Non-Secure worlds, providing the hardware root of trust (HWRoT) |
| Cryptography Accelerators | Dedicated hardware engines for AES-128/256 (encryption/decryption), SHA-256/384 (hashing), RSA, and ECC (asymmetric key operations) |
| True Random Number Generator (TRNG) | Hardware-based entropy source for cryptographic key generation and nonce production |

**(table continues...)**

**Table 1 (continued) Hardware security features**

| Feature | Description |
|---|---|
| Memory Protection Units (MPU/SMPU) | Secure and Non-Secure MPU instances enforce fine-grained read/write/execute access control on memory regions |
| Memory Protection Controller (MPC) | Security-aware controller that restricts memory access based on the security configuration (Secure/Non-Secure) to clearly define memory isolation and access control |
| Peripheral Protection Controller (PPC) | Security-aware controller that restricts peripheral register access based on the security state (Secure/Non-Secure/Privileged) of the calling code |
| Fault Injection Hardening (Hardware) | Hardware countermeasures against voltage-glitch and clock-manipulation physical attacks |

See the architecture reference manual [5] of the device for more information.

## 1.2 Lifecycle security and controlled debug access

Device lifecycle stages (LCS) define the security posture of the device at each phase of its operational life:

- **Development LCS**: Full debug access is available to support software development and testing
- **Production LCS**: Device policy assets including OEM keys are made permanent. This is the LCS in which devices must be deployed in the field. Only the assets defined under re-provisioning policy can be updated
- **RMA LCS**: Devices are transitioned to this stage when returned to Infineon for failure analysis. Only Infineon can open up the device in this LCS

LCS transitions are not reversible as they are locked in the OTP memory. For details on provisioning and lifecycle transitions, see the Ease of development section.

The debug access ports (DAP) support configurable enable/disable control and are governed by the device lifecycle stage. Where debug access is required on a production device, a cryptographically signed debug token can be used to grant selective, time-limited access without transitioning the device out of its current lifecycle state. See the Debug token use cases section for more information.

## 1.3 Secure boot, code authentication, and firmware update

The secure boot and firmware update capabilities listed below are provided entirely by the BootROM. However, these features need to be activated by provisioning the device with a policy file that explicitly enables them— a device in its factory state does not enforce image authentication and boots any image.

**Table 2 Secure boot and update features**

| Feature | Description |
|---|---|
| Hardware Root of Trust (HWRoT) | BootROM is the immutable, OTP-protected anchor of the chain of trust. When enabled, all subsequent boot stages are authenticated before execution is transferred. |

**(table continues...)**

**Table 2 (continued) Secure boot and update features**

| Feature | Description |
|---|---|
| Secure Boot – ECDSA image verification | BootROM can verify the cryptographic signature of the next boot stage (EPB or application/s) using ECDSA (P-256, P-384, or P-521) before transferring execution. Images that fail verification are rejected. |
| Multi-Image Authentication (up to 3 images) | BootROM can authenticate up to three application images. Each image is verified independently before any image is executed, ensuring the integrity of the entire multi-image application. |
| Firmware Update — Authenticated Overwrite | BootROM can authenticate a new image staged in the shared secondary flash slot using ECDSA and then copy to the corresponding primary slot by overwrite. Only images that pass signature verification are promoted, ensuring that only authenticated firmware is applied. |
| DFU over serial interface | The BootROM includes a built-in DFU capability over a serial interface (UART), allowing a host to transfer a new firmware image to the device's flash without an external programmer.<br>Image can be downloaded to primary slots (no authentication enforced) or to the shared secondary slot, in which case the BootROM authenticates the image on the next reset before promoting it to the primary slot. |

***Note**: Device also supports Post-Quantum Cryptography (PQC). For this feature, Edge Protect Bootloader (EPB) must be used as first OEM application in the boot chain. Refer to* Edge Protect Bootloader *for more information.*

## 1.4 Development tooling

**Table 3 Development tooling**

| Feature | Description |
|---|---|
| Edge Protect Tools | Host-side tool for provisioning, key generation, certificate creation, policy file management, and firmware signing (including PQC schemes). |
| Edge Protect Signing Service | The Infineon Edge Protect Signing Service (EPSS) is a cloud-based signing service that can be used to obtain OEM Owner Certificate. This certificate cryptographically binds the OEM's root of trust public key to the device's unique identity, establishing verifiable ownership and authorizing the OEM to provision and manage the device throughout its lifecycle. |

**(table continues...)**

**Table 3 (continued) Development tooling**

| Feature | Description |
|---|---|
| Device Configurator | A graphical configuration tool integrated into ModusToolbox™ that enables developers to configure device peripherals, memory regions, and security boundaries — including the Memory Protection Unit (MPU), Memory Protection Controller (MPC), Security Attribution Unit (SAU), and Peripheral Protection Controller (PPC) — without manual register programming. Configuration is exported as generated initialization code consumed directly by the application.<br>***Note**: The application layout and flash MPC configuration declared in the Policy File must be consistent with the corresponding settings applied in Device Configurator. The BootROM treats the Policy File as the authoritative source of truth at runtime — any divergence between the two will result in boot failures or incorrectly enforced security boundaries.* |

# 2 Ease of development

## 2.1 Device lifecycle stages (LCS)

Devices are shipped in the development stage, which allows the device to be provisioned and programmed as many times as needed to develop the final application. Devices are shipped with a default policy that defines all debug ports as open and allow application execution without code authorization. This allows you to program, evaluate, and debug the parts prior to any knowledge about security or code signing.

During the development stage, the OEM can take ownership of the part, allowing it to replace the default policy file with a custom policy. All aspects of the production stage can be simulated, including locking down the CPU access ports, which will lock out any attempt to connect the debugger to the part. The only difference between the development and production stages from a code operation point of view is that the SYS_AP (system access port) cannot be disabled in the development stage. This ensures that the developer cannot be locked out of the part and can, at any time, recover from a device configuration error and even return the device to its original state. When the firmware is ready for production, the device can be advanced to the production stage.

The production stage locks down the device owner, meaning that the key used to authenticate the firmware can no longer be changed from what was programmed into the device. The debug ports, including SYS_AP, can all be disabled or left enabled as per your requirement. Devices can be advanced to the production stage after receiving them from Infineon. This is the typical model once the OEM starts to ship their product. There is no need to transfer ownership in the development stage; instead, you can take ownership and advance the device to the production stage in a single step as part of provisioning the device.

The RMA stage is only used in the event that a problem is found in the hardware and the device must be shipped back to Infineon for failure analysis (FA). Figure 2 shows the flow from development to RMA.

***Note**: Blowing eFuses to change LCS requires VDDIO0 to be 2.5 V. See the device datasheet for more details.*

BootROM execution flow depends on the lifecycle stage at which the MCU is running. LCS is dictated by blowing eFuse bit values in to the device. During provisioning, the LCS can be moved for the device. Infineon signed RAM applications are used to provision the assets into the device and change the device LCS. RAM applications and the associated input parameters are loaded into the device SRAM through a SWD or DFU serial interface. RAM applications are authenticated by BootROM before it execute from device SRAM. The LCS is sequential and cannot be reverted back to the previous stage, since the LCS is maintained in the eFuse memory. Device DAP access restriction shall be based on a deployed policy, which will be applicable for both production and development LCS. Figure 2 shows the LCS transition in the PSC3M/P8, PSC3M/P7 MCU devices.

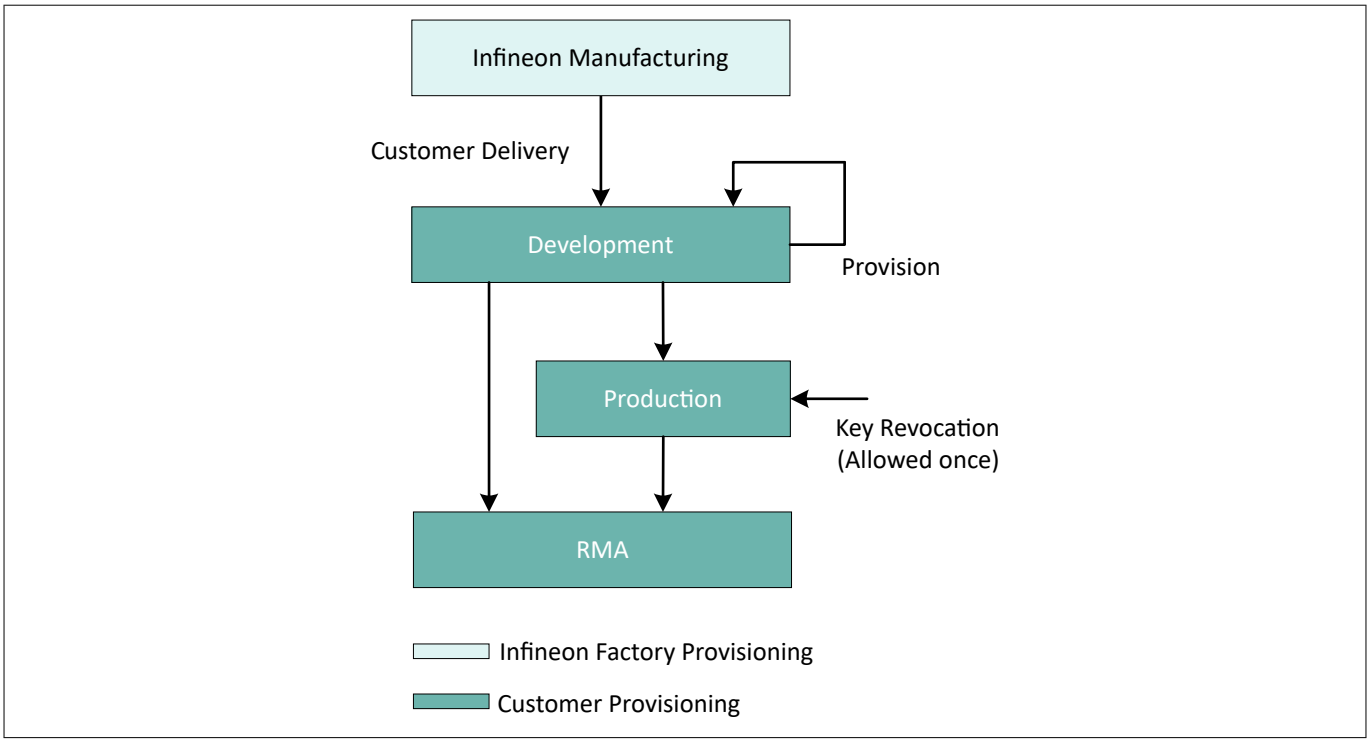



**Figure 2 LCS transition**

### 2.1.1 Development LCS

- **Initial device configuration**: Devices are shipped in Development Lifecycle Stage (LCS) by Infineon. In this LCS, default policy assets are provisioned and configured by Infineon
- **OEM provisioning**: OEMs can optionally customize the policy options and obtain the device ownership through provisioning. Device can be provisioned as many times as need in this LCS. Even if the keys are lost after the ownership option, device can be safely recovered without the fear of loosing the devices
- **Secure boot and debug**: Secure boot and secure debug can be optionally enabled through provisioning
- **LCS transitions**: From this stage, the device can be transitioned to either the Production LCS or the Return Merchandise Authorization (RMA) LCS. Once the LCS is advanced, it cannot be reversed as LCS advancement involves burning an eFuse

### 2.1.2 Production LCS

**Prerequisite**

All necessary OEM policy assets, including policies and OEM_KEY, must be fully set up before moving to this stage.

- **Secure Boot**: The device will only run the customer's application, either with or without secure boot enabled. However, Infineon advises enabling secure boot for added security in this configuration
- **Asset Locking**: This configuration locks the device's settings, safeguarding the application and sensitive data from unauthorized access or modifications, whether physical or software-based
- **LCS Transition**: To maintain a secure chain of trust, the only allowed transition from this state is to the RMA (Return Merchandise Authorization) configuration
- **Re-provisioning**: While in this configuration, certain policy settings can be reconfigured, but only those specified under the re-provisioning policy are eligible for changes

### 2.1.3 RMA LCS

- **Failure Analysis (FA)**: The RMA LCS enables Infineon to perform failure analysis (FA) on devices
- **Transitioning to RMA LCS**: To initiate FA, device must transition to RMA LCS and shipped to Infineon

  Following assets and secrets installed on the device will be automatically wiped when device advances to this LCS

  - Hardware Unique Key (HUK)
  - Initial Attestation Key (IAK)
  - MPN_KEY and MPN_ROT_KEY_01_HASH
  - OEM policies
  - All user flash contents
  - PROT_FW policy
- **Infineon policies Device Behavior After RMA Transition**: Once a device has been transitioned to the RMA state, it will no longer boot the OEM application. This ensures that the device is properly secured and prepared for failure analysis and potential refurbishment or replacement

## 2.2 Development flow

The typical development flow for a PSC3M/P8, PSC3M/P7 devices has the same work flow as most other systems.

1. Provision the device
2. Design, develop, and modify the project code
3. Compile/link
4. Program the device
5. Run/debug on the device

### 2.2.1 Transfer of ownership

Ownership of the device belongs to the one who owns the private/public keys that are used to sign and authenticate the application code and the policy file. When the PSOC™ Control parts leave Infineon, the default owner is Infineon. The default policy file configures the debug ports to be open and does not require code signing, so anyone can download and debug code without performing any additional action. Once ownership has been 'transferred', the policy file must be signed by the owner of the device, or the device will not boot. If you want to change something in the policy file, such as enabling code signing or changing the location of the first code, then you must also 'transfer ownership' to yourself because the policy file must be signed by the owner. The process of transferring ownership from Infineon to the OEM (user) only needs to occur once, but there is no limit to how many times ownership is transferred while the device is in the 'development'/ NORMAL_PROVISIONED lifecycle stage (LCS). This allows a device or development kit to be evaluated by several different people without having to remember or share security keys. Note that the device leaves Infineon in the development.

### 2.2.2 Getting started

Infineon ships the device with development LCS and provisioned with SIMPLE_APP configuration, which the BootROM can expect the first user application to be in raw format (simple hex files without any overhead of special image header/trailers) and launches the application without authentication. Even without transferring ownership, you can still evaluate the device.

General application development instructions are available in the "Application development instructions" section of the AN241270 - Getting started with PSC3M/P8, PSC3M/P7 MCUs on ModusToolbox™ software document.

For getting started with security, it is recommended to use the PSC3M/P8, PSC3M/P7 MCU: Basic Secure Application code example. You can start using this code example out of the box without applying any additional security, as the default OEM policy disables the 'secure boot' feature of the device. See the next sections for instructions on how to start using security.

The first step towards making any changes to the policy file is to transfer the ownership.

Once you are ready to transfer ownership to yourself, there are a few things that you need to do:

- Initialize application directory for Edge Protect Tools
- Generate OEM key pair (private/public)
- (Optional) Enable secure boot in the policy
- Create OEM_CERT certificate and have it signed by Infineon
- Sign the application
- Provision the device
- Transition to prod LCS - Reprovision - RMA

Infineon's Edge Protect Tools is a set of command-line tools used to perform the needed functions for key signing, key generation, provisioning the device, and so on. Edge Protect Tools is available as part of the Edge Protect Security Suite which must be installed in your system along with the ModusToolbox™ installation package.

Pre-compiled executable binary file for the tools is available for easy use and can be located in the `C:/Users/ /Infineon/Tools/ModusToolbox-Edge-Protect-Security-Suite/tools/edgeprotecttools/bin` directory. Add the path to this executable to the system environment PATH variable of the host PC and move it to the top of the list to use this tool from the shell command line. It is recommended to use 'modus-shell' that is installed along with ModusToolbox™. Modus-shell is installed in the `ModusToolbox/tools_x.y` directory.

***Note**: Ensure that the KitProg3/MiniProg4 on your PSC3M/P8, PSC3M/P7 MCU devices are updated to firmware version 2.60.1412 or later. For more information about the KitProg3, see the AN241270 - Getting started with PSOC™ Control C3M/P8, C3M/P7 MCUs on ModusToolbox™ software document.*

### 2.2.2.1 Initialize the application directory

When initializing the application folder for using Infineon's Edge Protect Tools, several folders will be created in your application directory. The following is a list of these folders and their contents.

- **keys:** Any keys that you create with the Edge Protect Tools will be placed in this directory. If you prefer to create your own keys, you must place the keys in this directory
- **logs:** Each time you run the Edge Protect Tools, creates a time and date log files. These files can assist in debugging any problems with the tools that can occur. This directory is created for the first time that you execute an Edge Protect Tools command after initialization
- **packets:** Some commands generate JSON-formatted packets that are downloaded into the device or will be used by another command. These files are placed in this folder
- **policy:** Contains some sample policy files that can be modified or copied as required. Eventually, each one will have a custom policy file that fits their application

Open modus-shell and navigate to the top level of your ModusToolbox™ application directory. This will be the same level as the application project directories.

```
$ cd <application_directory>
```

The following command will initialize the application directory and prepare it for all the following Edge Protect Tools commands, it will be executed once per application. Additionally, execute this command whenever a new version of Edge Protect Tools is installed.

```
edgeprotecttools -t psoc_c3x8 init
```

Execute the following command once per session to configure the connection between the tool and a MiniProg4 or kit SWD port.

```
edgeprotecttools set-ocd --name openocd --path <openocd_path>
```

***Note**: If you are using modus-shell on a macOS system, the recommended usage for Edge Protect Tool is as follows:*

```
arch -arm64 edgeprotecttools <command>
```

### 2.2.2.2 Key generation

You need to create two key pairs that are private and public keys. These keys are used during secure boot enablement to sign the images to be validated by the BootROM. The default key used is the primary key pair (key_0). The secondary key pair (key_1) is useful if you need to revoke the primary key incase of a private key security breach. The following two commands generate two keys that will be placed in the key's directory. The key types supported are ECDSA-P256, ECDSA-P384, and ECDSA-P512.

```
edgeprotecttools create-key --key-type ECDSA-P521 --output keys/oem_private_key_0.pem keys/
oem_public_key_0.pem
```

```
edgeprotecttools create-key --key-type ECDSA-P521 --output keys/oem_private_key_1.pem keys/
oem_public_key_1.pem
```

The keys generated are ECDSA 521-bit key (P521) keys and the output is in the .pem format. This is a common ASCII format compatible with many public-domain tools. To get the supported key list from the edgeprotecttools, try the following command edgeprotecttools create-key help.

***Note**: User can create either one or two OEM RoT key pairs to be used for provisioning. For production lifecycle, user must provision two pairs of OEM RoT keys into the device.*

**Table 4 Command-line parameters**

| Command option | Example parameter | Description |
|---|---|---|
| --key-type | ECDSA-P521 | Key type of the generated key |
| -o, --output | keys/oem_private_key_0.pem | Path where generated key is stored |

### 2.2.2.3 OEM certificate creation

This step creates a certificate signing request (CSR). A CSR contains information about the OEM-certificate such as the certificate type (development or production), the OEM ROT public keys, the OEM name, and project information. The CSR should be signed by the OEM ROT private key. The OEM-signed CSR must then be submitted to Infineon to generate the Infineon signed OEM certificate. This step is required every time when the OEM provisions a new set of OEM ROT keys to the device. You can continue to use them in the Development LCS for as long as needed, and across multiple devices, without having to regenerate them. The certificates remain valid and can be reused multiple times, as long as you do not need to change the underlying keys. You will only need to regenerate the certificates in two scenarios:

1. When you want to update or change the keys associated with the certificates
2. When you are ready to transition from the Development LCS to the Production LCS

If you want to use a common key across multiple devices or a batch of devices, you can create the certificates once and reuse them as needed. It is essential to note that CSR/OEM certificates are not tied to a specific device and can be used across multiple devices, if desired. This flexibility allows you to manage your certificates efficiently and streamline your development and production processes.

The following command creates an OEM signed CSR for development LCS that can be submitted to Infineon to create an Infineon signed OEM certificate (OEM_CERT).

```
$ edgeprotecttools -t psoc_c3x8 oem-csr --public-key-0
keys/oem_dev_pub_key.pem --public-key-1 keys/oem_dev_pub_key.pem--sign-
key-0 keys/oem_dev_priv_key.pem --sign-key-1 keys/oem_dev_priv_key.pem --
oem "Company Name" --project "Project Name" --project-number 12345678 --
cert-type development --output oem_csr_development.bin
```

**Table 5 Command-line parameters for oem-csr**

| Command option | Example parameter | Description |
|---|---|---|
| `-o` | packets/apps/prov_oem/oem_csr.bin | Path where output CSR is stored |
| `--certificate-name` | "oem-cert" | Name of the certificate |
| `--oem` | "Dummy OEM" | Name of the OEM |
| `--project` | "Dummy Project" | Name of the project associated with the certificate |
| --project-number | "1234" | Project number associated with the certificate |
| `--public-key-0` | keys/oem_public_key_0.pem | OEM ROT Public Key |
| `--sign-key-0` | keys/oem_private_key_0.pem | Private key associated with the `oem_public_key_0` |
| `--cert-type` | development/production | The certificate type determines the LCS of the device. It can be either development or production LCS |

Once the CSR is created, it must be signed by Infineon to create a valid OEM certificate. Follow these steps outlined to generate an Infineon signed OEM certificate.

1. Prior to creating a certificate, you must sign up for an Infineon online software tools and services (OSTS) account. Any developer may create an OSTS account by registering at osts.infineon.com

2. Once you have registered, login to your OSTS account and click on Edge Protect Signing Service. This will take you to a page where you can upload your Certificate Signing Request (CSR) that you created in the previous step. Click on the Upload New Certificate Request button. This will take you to a window where you can upload your CSR, enter a name for the certificate, and enter a description in the previous step. Click on the Upload New Certificate Request button. This will take you to a window where you can upload your CSR, enter a name for the certificate, and enter a description

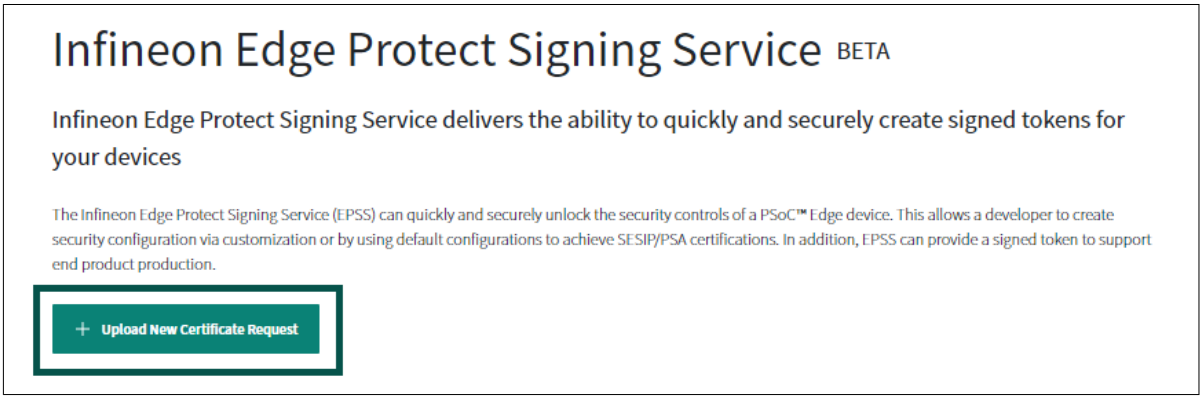



**Figure 3 Upload new certificate request**

3. Enter the certificate name without any spaces or special characters, if you enter the name as “oem”, the generated certificate will be named “oem_cert.bin”. Select the silicon revision as PSOCC3X8. Next, enter the description for this certificate in the “Description” field. This description will be in the list of certs that you own, so you can easily identify one cert from another if you have more than one.
4. Click the Drop file here or click to upload button to upload the CSR and navigate to the CSR that you created in the previous step instead of dropping the file in this area. In the previous steps, the path was `packets/apps/prov_oem/oem_csr_X.bin`

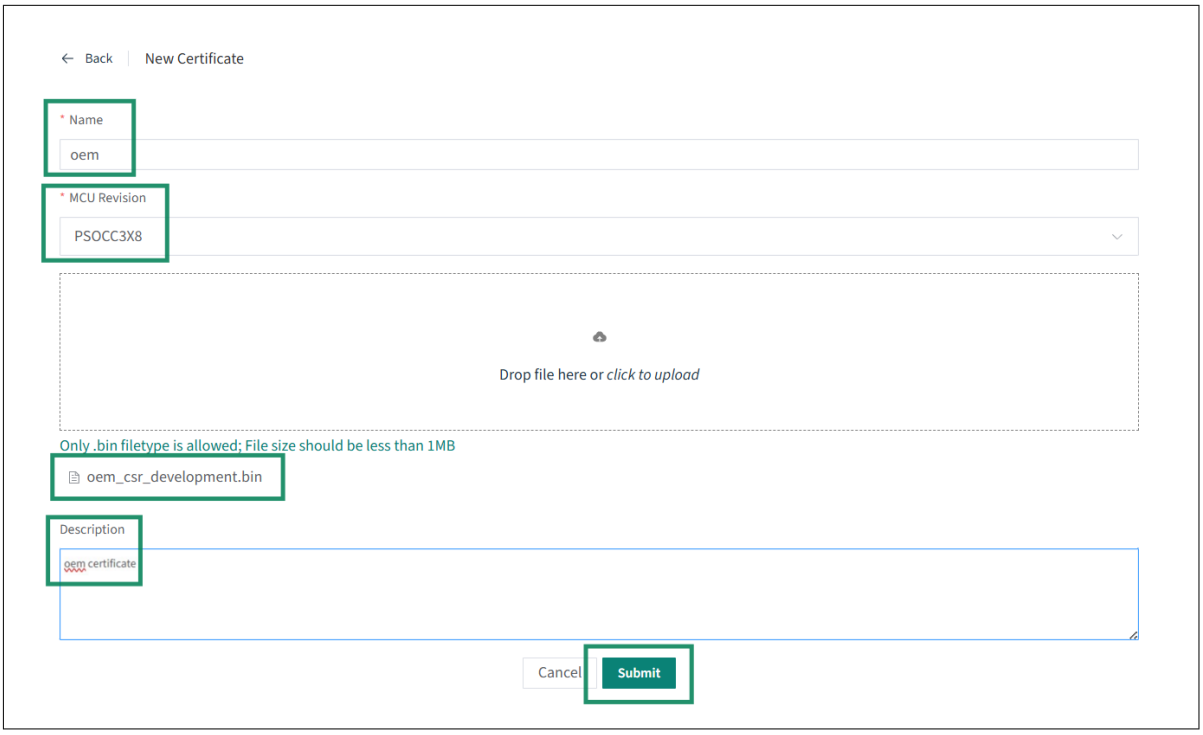



**Figure 4 Uploading CSR: Provide this image**

5. Once the name and description have been entered and the CSR has been uploaded, click the Submit button. This should take you back to the original page with a list of certificates under Manage Certificates. If the list does not show the most recent certificate generated, click on the Refresh List. You should now see the signed certificate ready for you to download. Click on the Download button on

the line that contains the certificate you want to download. This will download the signed certificate to the location on your computer where the files are downloaded. Copy the certificate to the path `packets/apps/prov_oem/oem_cert.bin`

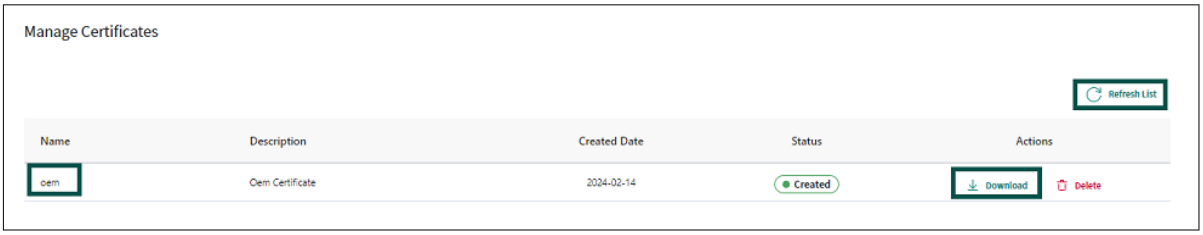



**Figure 5 Manage certificates**

You can revisit this website at any time and download any of the certificates that have been signed in the past. During development, you only need to perform these steps once, but you can generate multiple certificates if needed.

### 2.2.2.4 Provisioning and the policy file

The default `OEM_Policy` file and `OEM_ROT_PUBLIC_KEY` are stored in the device before leaving the Infineon factory. The `OEM_Policy` file is configured such that all debug ports are open and the OEM code does not need to be signed. This allows the user to program and debug the device without restrictions. These default files are intended to be replaced by the OEM during development. These files must be replaced to eventually advance to the production stage.

Provisioning by the OEM occurs any time when the user takes ownership of the device and/or updates the policy file. The policy file is used to set boot parameters and define keys that the OEM wants to store in the device prior to shipment. During the development stage, the policy file may be updated as frequently as needed. The policy file is a text-editable file in JSON format. It is recommended to reuse the original policy file the first time you take ownership of the device to verify that you are still able to program the device in case of an error. The user-settable parameters in the policy file include the following:

- Flag to authenticate the first OEM code or not
- Location of OEM application(s)
- CM33 CPU boot clock speed
- Configuration of debug ports
- Keys to be loaded into the device

The default policy file `policy_oem_provisioning.json` (OEM_Policy) is stored in the `<application directory>/policy` directory that was created when the application directory was initialized with `edgeprotecttools`. In order for the provisioning process to install a new OEM_Policy, it must be signed by the OEM_ROT_PRIVATE_KEY. Even during the Development stage, the OEM_Policy must be signed. More detail on the structure of the policy file and a description of the parameters are provided in the Policy file.

Any time ownership is transferred, the policy file must be signed by the new OEM_ROT_KEY (private). The following command changes ownership and updates the policy file.

```
$ edgeprotecttools -t psoc_c3x8 provision-device -p policy/policy_oem_provisioning.json --key
keys/oem_private_key_0.pem --ifx-oem-cert packets/apps/prov_oem/oem_cert.bin
```

If ownership has already been transferred and you only want to update the policy file, you can leave the --ifx-oem-cert option, as shown below.

```
$ edgeprotecttools -t psoc_c3x8 provision-device -p policy/policy_oem_provisioning.json --key
keys/oem_private_key_0.pem
```

Table 6 provides the usage of each of the command-line parameters.

**Table 6 Command-line parameters for provision-device**

| Command option | Example parameter | Description |
|---|---|---|
| `-p` | policy/policy_oem_provisioning.json | This is the policy that is being used to provision with. You may start with the default and edit as needed |
| `--key` | keys/oem_private_key_0.pem | This should be the initial OEM private key. It is used to sign the policy file. It must be the private key associated with the initial public key that was added to the oem-cert. |
| `--ifx-oem-cert` | packets/apps/prov_oem/oem_cert.bin | This is the Infineon signed certificate that was created in the previous step. This option is not required if ownership has already been transferred and only the policy file is being updated |

See AN242919 - PSOC™ Control C3M/P8, C3M/P7 MCUs provisioning specification.

See the Appendix A section of this document for more details on the policy fields.

### 2.2.3 Enable secure boot

Secure boot of the first OEM application can be enabled through the OEM policy file configuration. When the `SECURE_APP` flag in the policy is enabled, Extended Boot will always verify the signature of the application that it boots, which is the first OEM code. The default policy in the PSOC™ Edge device disables the `secure_boot` that means that for out-of-the-box use, BootROM does not verify the signature of the first OEM code.

To enable or disable the "secure_boot" in policy, you have to edit the OEM policy, which is a JSON file that can be located in the `<application directory>/policy/policy_oem_provisioning.json` path once you have initialized the Edge Protect Tools. See Initialize the application directory section for more details.

Set the `boot_cfg_id` value to `SECURE_APP` in the boot policy.

```
"boot_cfg_id": {
"description": "A behavior for BOOT_APP_LAYOUT (BOOT_SIMPLE_APP applicable to
NORMAL_PROVISIONED only)",
"applicable_conf": "SIMPLE_APP, SECURE_APP, DUAL_BANK_SIMPLE_APP, DUAL_BANK_SECURE_APP,
PROT_FW",
"value": "SECURE_APP"
},
```

**Table 7 Application configuration**

| boot_cfg_id | Description |
|---|---|
| SIMPLE_APP | A simple application without any overhead of image headers and signature. |
| SECURE_APP | A simple application with overhead of image headers and signature, that is authenticated by the BootROM during boot. |
| DUAL_BANK_SIMPLE_APP | A simple dual back application without any overhead of image headers and signature. |
| DUAL_BANK_SECURE_APP | A simple dual bank application with overhead of image headers and signature, that is authenticated by the BootROM during boot. |
| PROT_FW | A simple application with overhead of image headers and signature and enhanced protection that is authenticated by the BootROM during boot. |

After making the required changes to the policy file, provision the device with the updated policy.

```
$ edgeprotecttools -t psoc_c3x8 provision-device -p policy/policy_oem_provisioning.json
```

***Note**: As a prerequisite, follow the steps mentioned in the Getting started section to generate your keys and certificate and provision the device to transfer ownership.*

### 2.2.4 Signing user application

The first OEM application must always be in MCUboot format (If any one of these features enabled: secure_app, dual_bank_secure_app, prot_fw) to be successfully launched by the BootROM software. When using the default OEM policy, which has 'SIMPLE_APP' flag set.

Once the device has been provisioned with an OEM policy that enables 'SECURE_APP' flag, the signature of the first OEM application will be authenticated by the BootROM before launching the application.

The following command can be used to sign an application image. This command adds both the MCUboot header and signature.

```
$ edgeprotecttools sign-image --image .\user_app.hex --output .\user_app_signed.hex --key-
path .\keys\oem_private_key_0.pem --overwrite-only --header-size 0x400 --slot-size 0x1000 --min-
erase-size 0x200 --pad --hex-addr 0x32000000
```

***Note**: Use "`edgeprotecttools sign-image --help`" for command options.*

**Table 8 Command-line parameters for sign-image**

| Name | Type | Optional/required | Description |
|---|---|---|---|
| file | string | Required | Path to the file to be signed or converted into MCUboot format. |

**(table continues...)**

**Table 8 (continued) Command-line parameters for sign-image**

| Name | Type | Optional/required | Description |
|---|---|---|---|
| signing-key | string | Optional | ECDSA or RSA private key used to sign the image. |
| sha | string | Optional (required for ML-DSA signature) | The SHA algorithm used for image hashing. Accepted values: `sha256`, `sha384`, `sha512`, or `auto`. When set to `auto`, SHA-256 is used if no cryptographic signature is applied, or the default algorithm for the selected signature type is used. Default : auto |
| header-size | string, integer | Required | MCUboot header size. |
| slot-size | string, integer | Required | Maximum slot size. |
| min-erase-size | string, integer | Optional | Minimum erase size. Default: 0x8000. |
| align | string, integer | Optional | Flash alignment. Default: 8. Available values: 1, 2, 4, 8. |
| pubkey-format | string | Optional | Public key format in the image TLV: full key or hash of the key. Available values: hash or full. Default: hash. |
| pubkey-encoding | string | Optional | Public key encoding in the image TLV. Applicable values: der, or raw. Default: der. |
| signature-encoding | string | Optional | Image signature encoding. Applicable values: asn1, or raw. Default: asn1. |
| pad | boolean | Optional | Adds padding to the image trailer. Pads the image from the end of the TLV area up to the slot size. boot_magic is always at the very end after the padding. |
| overwrite-only | boolean | Optional | Use overwrite mode instead of swap. |
| hex-address | string, integer | Optional | Adjust the address in the hex output file. |

#### 2.2.4.1 Postbuild signing using ModusToolbox™ application

If you are working on a PSC3M/P8, PSC3M/P7 ModusToolbox™ code example, such as the secure boot application, then the first user application is the CM33 SPE project (main_cm33_s).

This code example comes with the ‘secure_boot’ disabled policy. When the 'SECURE_APP (secure boot)' is enabled in the OEM_Policy, then the CM33 SPE application needs to be signed.

## 2 Ease of development

The user can enable signing of the application by selecting the suitable combiner-signer JSON file from the /configs folder. For example, in the /configs/ directory, the JSON file "signer_combiner.json" enables signing of the proj_cm33_s. This JSON file contains the path to the key (see Key generation) to be used for signing the image along with other parameters.

***Note**: The key pair (private key) used for signing the image should be the same key pair(public key) that is provisioned into the device during transfer of ownership. See Transfer of ownership.*

Ensure that the `<Basic Secure App>/common.mk` points to the correct JSON file before building the application.

```
COMBINE_SIGN_JSON?=configs/singer_combiner.json
```

In other ModusToolbox™ code examples, there is no pre-configured combiner signer JSON file that can sign image, it needs to be manually modified to enable signing. Navigate to the <application-directory>/configs directory and open the `singer_combiner.json` file. Edit the JSON file to add the following line 'signing-key" :"./../keys/oem_dev_priv_key'. This provides the path to the OEM private key to sign the proj_cm33_s image. Save this configuration and build and program the application as instructed in the code example `README.md` file.

```
"commands" :
            [
                {
                    "command" : "sign",
                    "inputs" :
                    [
                        {
                            "description": "Secure application",
                            "file" : "../build/project_hex/main_cm33_s.hex",
                            "signing-key": "../keys/oem_dev_priv_key.pem",
                            "header-size": "0x400",
                            "slot-size": "0x10000",
                            "fill-value": "0x00",
                            "min-erase-size": "0x200",
                            "signature-encoding": "raw",
                            "pubkey-encoding": "raw",
                            "pubkey-format": "full",
                            "hex-address": "0x32000000",
                            "overwrite-only": true,
                            "align": "1",
                            "remove-tlv": ["0x01", "0x02", "0x10", "0x11", "0x12"]

                        }
                    ],
                    "outputs":
                    [
                        {
                            "description": "Signed secured application",
                            "file" : "../build/project_hex/main_cm33_s_signed.hex",
                            "format": "ihex"
                        }
                    ]
```

### 2.2.5 Transition to Production LCS

The device can be transitioned to production LCS by using a production type of certificate for provisioning.

***Note**: Once the device is transitioned to production LCS, it cannot be transitioned back to development LCS.*

To transition the device to production LCS, the following steps are required.

1. Generate the OEM ROT public-private key pair
2. Generate a new CSR and get it signed by Infineon to create a valid OEM certificate via Infineon Online Software Tools and Services (OSTS) account. Any developer may create an account by registering at OSTS
3. Provision the device with the ROT public keys and the signed CSR certificate to transfer the ownership
4. Optionally enable the “SECURE_APP” option in the BOOT_CFG_ID (if secure boot is required)

For more information, see PSOC™ Control C3M8 MCU: Secure boot application.

### 2.2.6 Re-provisioning

Re-provisioning is allowed only in the Production LCS. Only those policies that are allowed to be re-provisioned can be modified using the re-provisioning process. For details on the re-provisioning policy refer to AN242919 - PSOC™ Control C3M/P8, C3M/P7 MCUs provisioning specification.

Device can be re-provisioned using the following Edge Protect Tools command after it has been transitioned to Production LCS

```
$ edgeprotecttools -t psc3x8 reprovision-device -p
policy/policy_oem_provisioning.json --key keys/oem_rot_private_key_0.pem
```

***Note**: If the system access port is closed during the transition to Production LCS, then re-provisioning is not possible through the SWD interface. Alternatively, you may close the ports using the re-provisioning step.*

# 3 Application structure

When using a multi-CPU system, your application will usually contain at least one project per CPU. The PSC3M/P8, PSC3M/P7 MCU devices are no different in this way. With the addition of a bootloader and TrustZone®, you can have two or three separate projects for a single CPU included in your application. This section will show a few typical configurations, from simple to more complex, to give you an idea of the possibilities. In this document, the term 'application' refers to one or more projects that run on different CPUs. Each CPU will have one or more projects in a single application.

Although PSC3M/P8, PSC3M/P7 are multicore devices, CM33 is a main CPU core, which includes TrustZone®, so there will be two projects associated with it: a Secure Processing Environment (SPE) and a non-secure processing environment (NSPE) project. It is optional to support just the NSPE project if needed.

Each application may contain four additional folders that are generated when the Edge Protect Tools is used to initialize the application for provisioning. These folders include policy, keys, packets, and logs. They contain the default policy files, the generated or stored keys, packets generated by the tools, and log files generated each time you execute the Edge Protect Tools. See the Getting started section for more details about Edge Protect Tools and these folders.

## 3.1 Basic application structure

In the default application structure, only one SPE project will be created and presented, this default structure can be used if you need only an SPE-based project, see Figure 6. Additionally, you can add an NSPE project if needed, see Figure 6.

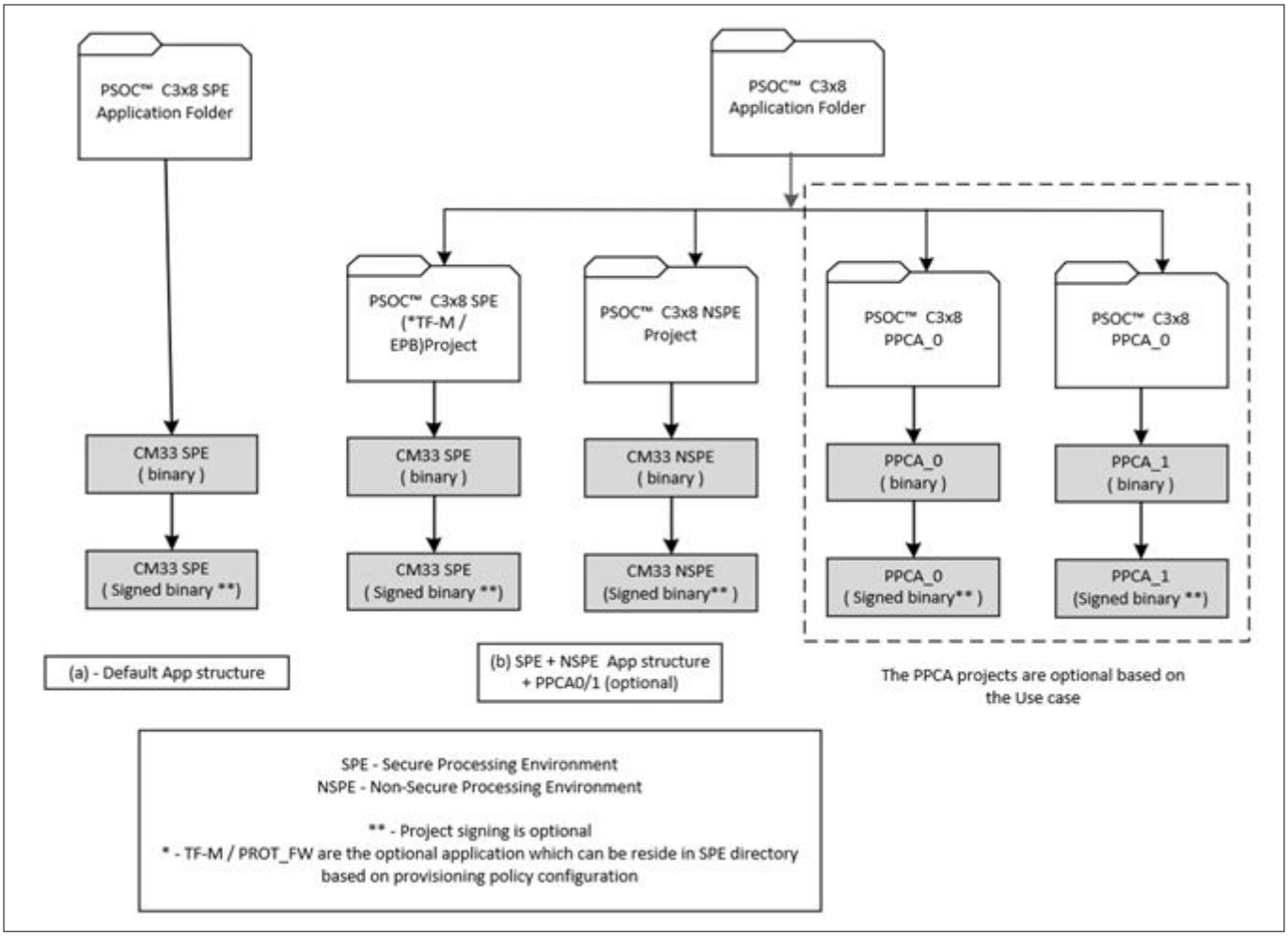



**Figure 6 Project structure**

In a simple application, the BootROM just launches the SPE project and the SPE project launches the NSPE if present. In case of secure application, the BootROM validates all the available applications (for example, SPE, NSPE, PPCA0, PPCA1). The number of applications to be validated is set in the "boot_app_layout" section of the policy. (For more details, refer to SECURE_APP section) during the provisioning of the device. Refer to Secure Non-Secure template PSOC™ Control C3M8 MCU: Secure-Non-Secure template.

## 3.2 Application with Edge Protect Bootloader

If the Edge Protect Bootloader (EPB) is used, it will be viewed as a separate application within ModusToolbox™ and it will be compiled, linked, and programmed separately from the main application, see Figure 7. The main application will contain the CM33 SPE and CM33 NSPE project. Note that in this configuration, only the EPB application directory includes the four folders required when provisioning the device. This is because when the device boots up, the secure boot sequence only extends the chain of trust to the next application. In this case, the next application is the EPB. EPB is responsible for extending the chain of trust to the next application by authenticating the code. Provisioning and the policy file only directly affect the first OEM application that is the EPB. Keys used to authenticate the main application can have been loaded into the device during provisioning or can be stored in the EPB; this is up to you and the application requirements. See PSOC™ Control C3M8 MCU: Edge Protect Bootloader code example.

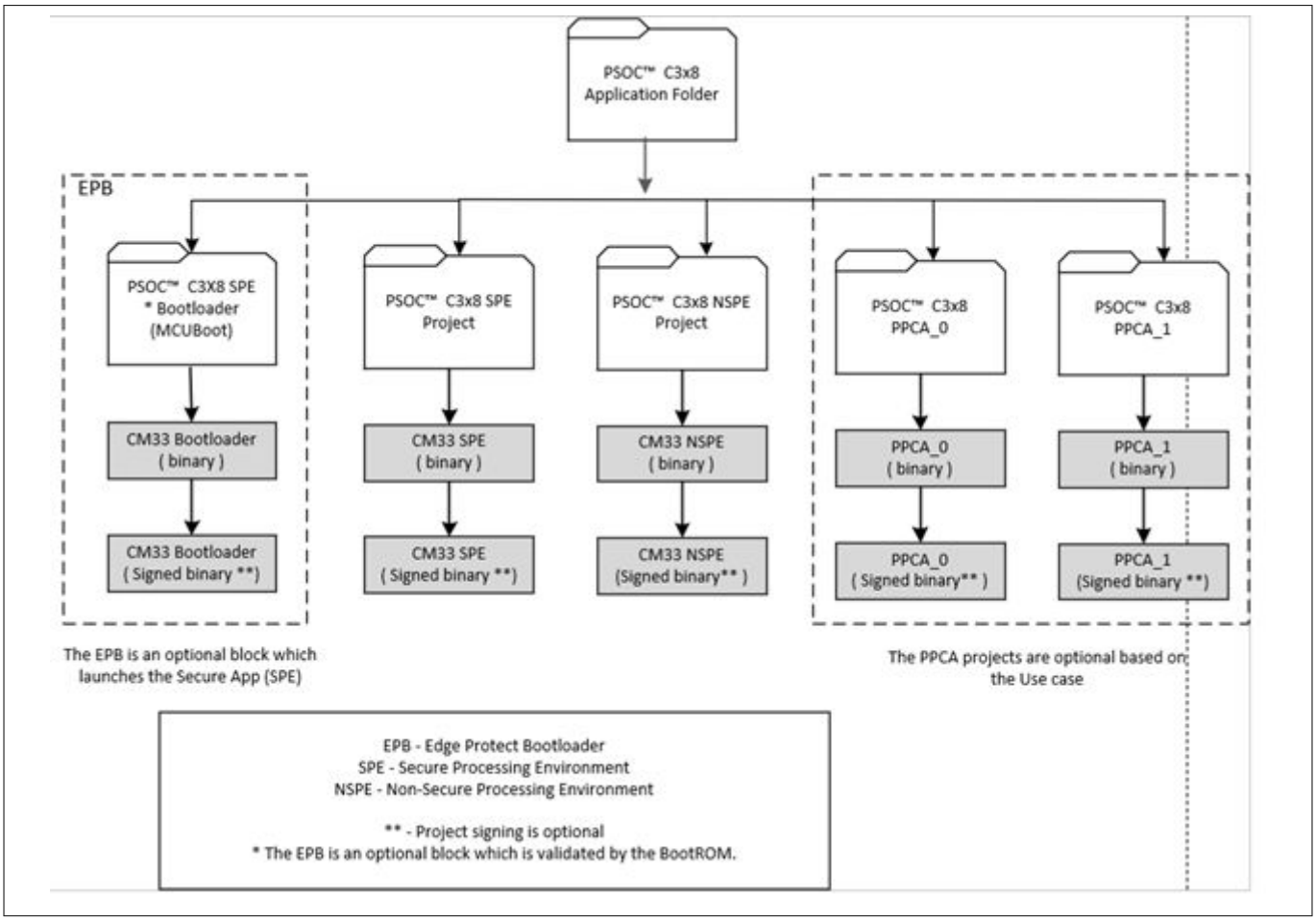



**Figure 7 Bootloader Structure**

# 4 Boot flow

BootROM executes the code from the beginning of power-on reset (POR) until the launch of an OEM application. The main functionality of BootROM is to launch the next application. It may authenticate the image, if the device is provisioned to enable secure boot. BootROM consists of ROM_BOOT and FLASH_BOOT.

## 4.1 Boot sequence

- After reset, the ROM_BOOT applies basic trim, configures clocks, and validates the SFlash
- ROM_BOOT execution jumps to FLASH_BOOT on successful validation
- FLASH_BOOT authenticates the next stage application (OEM)
- FLASH_BOOT prepares to launch the application on successful authentication
- If authentication fails, the device will go to the DEAD branch

Figure 8 shows the steps involved in the BootROM root of trust.

## 4 Boot flow

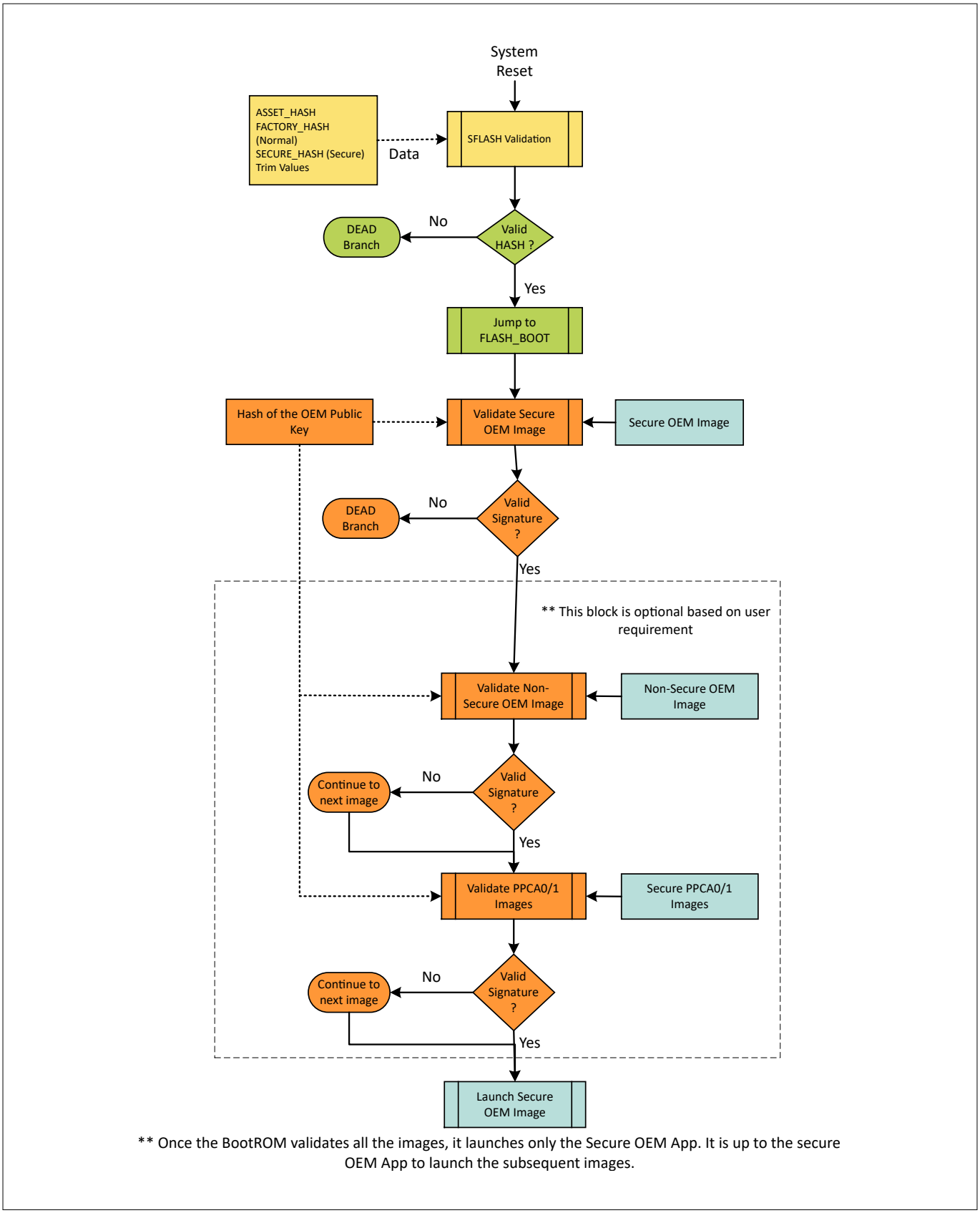



**Figure 8 Root of trust**

Table 9 shows the elements used to validate each stage of BootROM validation.

**Table 9 Elements used for each stage of validation**

| Step | Element | Checked by | Using |
|---|---|---|---|
| 1 | ROM | N/A immutable | N/A immutable |
| 2 | ASSET_HASH | ROM_BOOT | eFUSE zeroes |
| 3 | SFLASH_BOOT code | ROM_BOOT | ASSET_HASH |
| 4 | OEM_ROT_KEY | SFLASH_BOOT code | eFUSE_HASH |
| 5 | OEM Application | SFLASH_BOOT code | OEM_ROT_KEY |

The BootROM validation can be extended to multiple images. In this case, The BootROM validates all the applications and further does the following:

- BootROM launches the Secure Application after validating all the images based on the images entries made in "boot_app_layout" field of the policy. Failure to validate the Secure application will push the device to dead state. If validation fails for any image other than the Secure Application, BootROM still boots the Secure Application; however, it records the failed image's status in the boot status registers
- The secure application executes to launch either Non-Secure Application or the PPCA0/1 applications based on the OEM use case

## 4 Boot flow

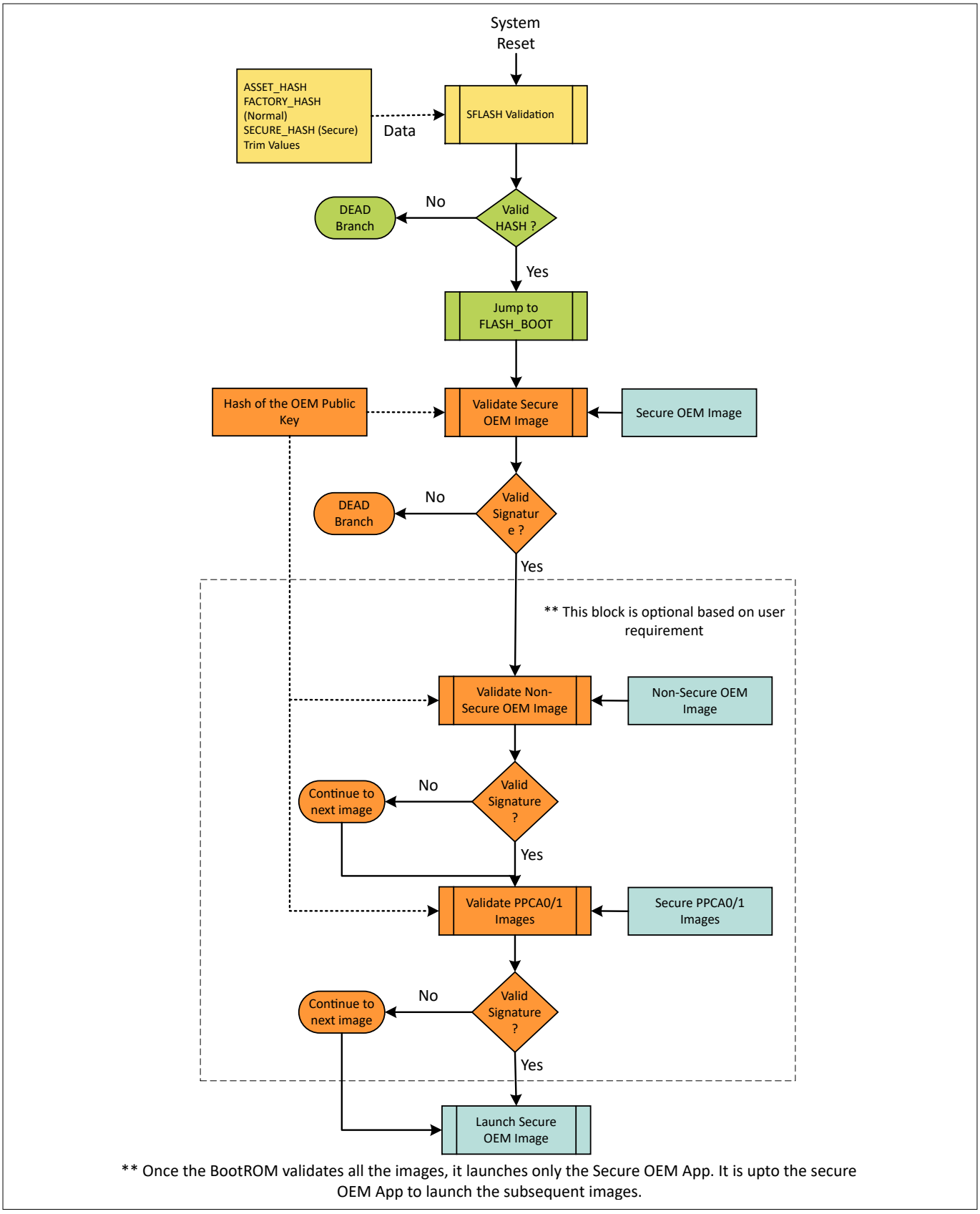



**Figure 9 Root of Trust (Multiple Images)**

To enable the secure boot (image validation) in the device, a **BOOT_CFG_ID** parameter must be set in the boot policy along with the **boot_app_layout** parameter as mentioned in the earlier sections which notifies BootROM the number of images to be validated. These values will be configured during device provisioning.

## 4.2 SIMPLE_APP

- SIMPLE_APP is a default BootROM configuration where the BootROM launches the OEM Application(Secure) without authenticating the signature of the image
- The OEM application will be present in the user flash region with basic application format (interrupt table and code, no MCUboot format)
- OEM application start address is the start of internal flash

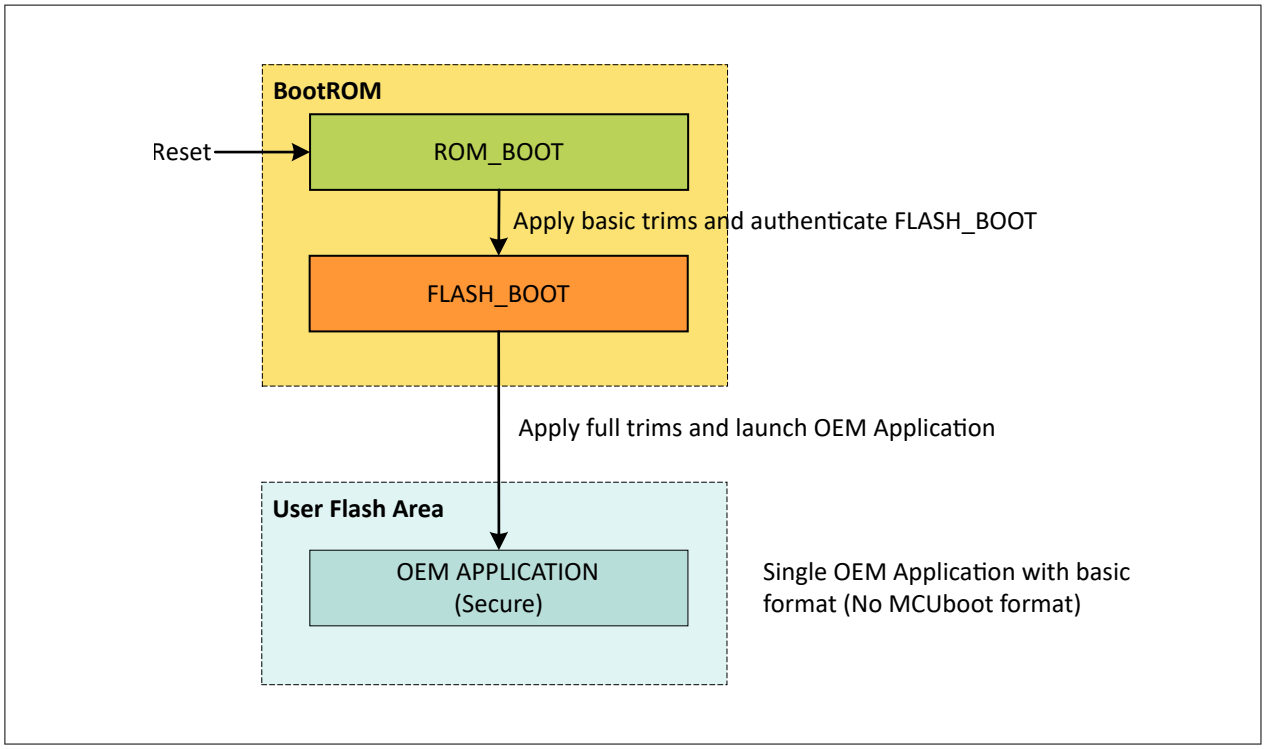



**Figure 10 Boot simple app**

- The OEM Application can launch the Non-Secure OEM application or the PPCA0/1 Application based on the OEM use case
- The BOOT_CFG_ID field in the policy has to be set to SIMPLE_APP to enable simple boot

## 4.3 SECURE_APP

The process of verifying and validating the integrity and authenticity of firmware as a prerequisite to their execution. At each stage of the boot process checks that code is authorized to run before execution.

The BootROM authenticates user application based on boot app layout values provided in the oem policy. The sample policy showing the enablement of the secure boot(image validation by BootROM) and the number of images to be authenticated. The address and size parameter can be set to zero to skip validation of a particular image. In this sample policy, we are validating four images OEM App(Secure and Non-Secure) and PPCA0/1 Apps.

***Note**: With SECURE_APP enabled, at least one entry (Secure Application) in the boot_app_layout is required. Else, the device fails to boot.*

```
"boot": {
      "boot_cfg_id": {
        "description": "A behavior for BOOT_APP_LAYOUT (BOOT_SIMPLE_APP applicable to
NORMAL_PROVISIONED only)",
        "applicable_conf": "SIMPLE_APP, SECURE_APP, DUAL_BANK_SIMPLE_APP, DUAL_BANK_SECURE_APP,
PROT_FW",
        "value": "SECURE_APP"
      },

"boot_app_layout": {
        "description": "The memory layout for the applications defined by BOOT_CFG_ID.
0x32000000 - 0x33FFFFFF for secure addresses; 0x22000000 - 0x23FFFFFF for non-secure addresses",
        "value": [
          {
            "address": "0x32000000",       ** Secure Application
            "size": "0x40000"
          },
          {
            "address": "0x22010000",         ** Non-Secure Application
            "size": "0x10000"
          },
          {
            "address": "0x22030000",        ** PPCA0 Application
            "size": "0x8000"
          },
          {
            "address": "0x22038000",        ** PPCA1 Application
            "size": "0x8000"
          },
          {
            "address": "0x00000000",        ** Shared Secondary Slot
            "size": "0x00"
          }
        ]
      },
```

***Note**: Maximum number of images a BootROM can validate are 5.*

## 4.4 DUAL_BANK_SIMPLE_APP

- The user flash regions can be split into two equal regions. Both regions will have dual bank counter offset and an OEM application
- BootROM can read the dual bank counter offset value and identify the active bank region
- BootROM will then authenticate the active bank region and execute the application

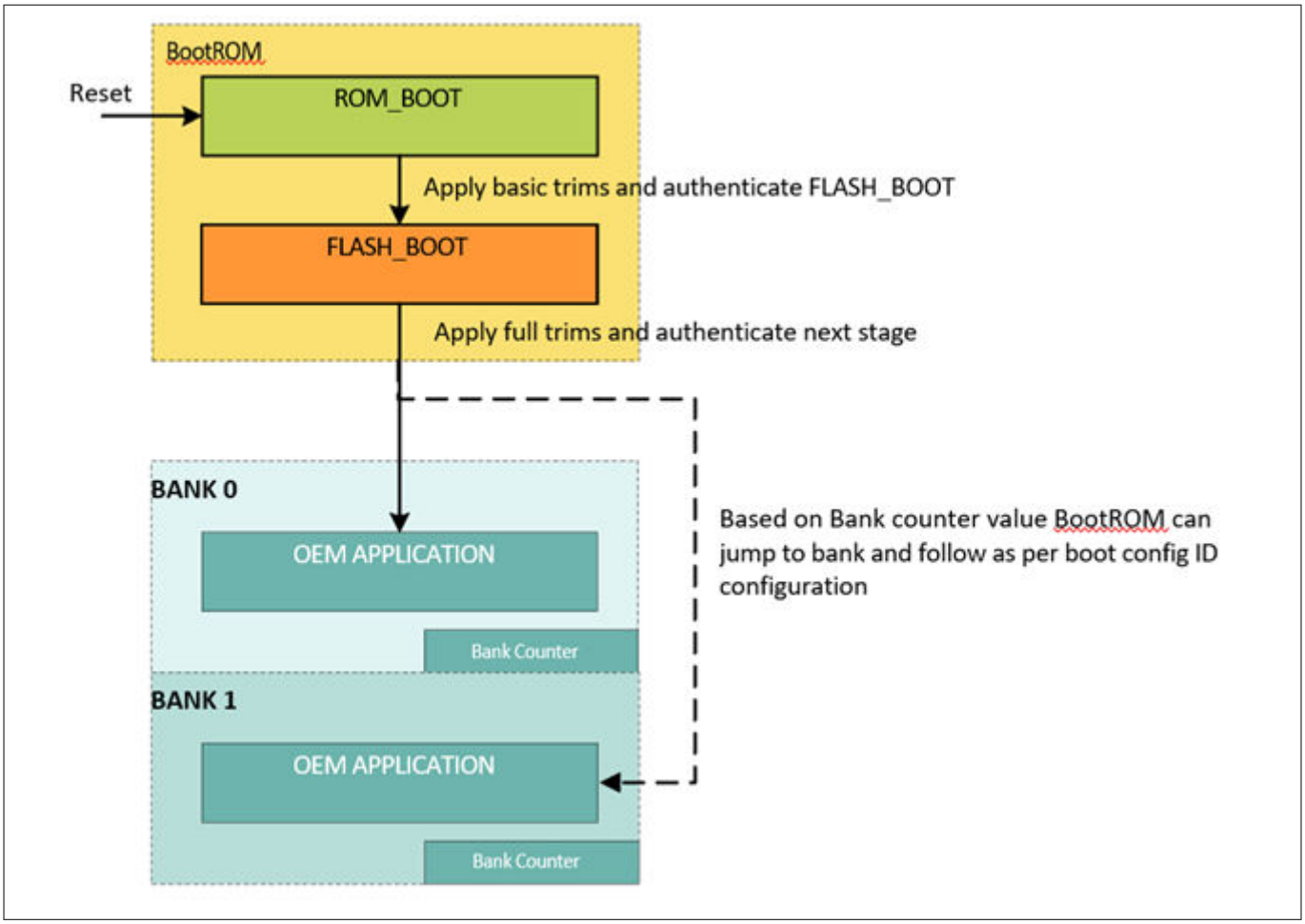



**Figure 11 Dual bank configuration**

***Note**: The optional PPCA images that are launched by the OEM applications are also part of the BANK 0/1. In this case, each bank will have an OEM Application along with the PPCA0 and PPCA1 applications as well.*

When Dual bank is enabled, particular image (CM33, PPCA0, PPCA1) update is not recommended. Rather suggest to update the complete bank. The reason being rest of the image version will be older.

## 4.5 DUAL_BANK_SECURE_APP

This is same as the DUAL_BANK_SIMPLE_APP but with OEM applications being authenticated by the BootROM before the launch from the appropriate bank.

# 5 Code signing and verification

Code signing and verification must for secure boot configuration. In other words, the user application must be signed for SECURE_APP and DUAL_BANK_SECURE_APP configurations and the number of images to be signed and validated depends on the image entries in the boot app layout in the oem policy.

## 5.1 Code signing

To verify the user application, create a digital signature and append it to the end of the project binary during build time. The user code is not encrypted, but the digital signature is the encrypted digest one. The digital signature is generated by encrypting the digest with the ECDSA private key. In this case, it is the OEM_ROT_PRIVATE_KEY/OEM_DEV_PRIV_KEY. The digest is generated by running the user project binary through an SHA-256 hash function. This method guarantees that a third party without access to the OEM's private key cannot properly sign the application code, which is therefore only created and signed by the OEM.

***Note**: The MCUboot-based Project Header consists of image attributes and Project Trailer consists of image trailer TLV information.*

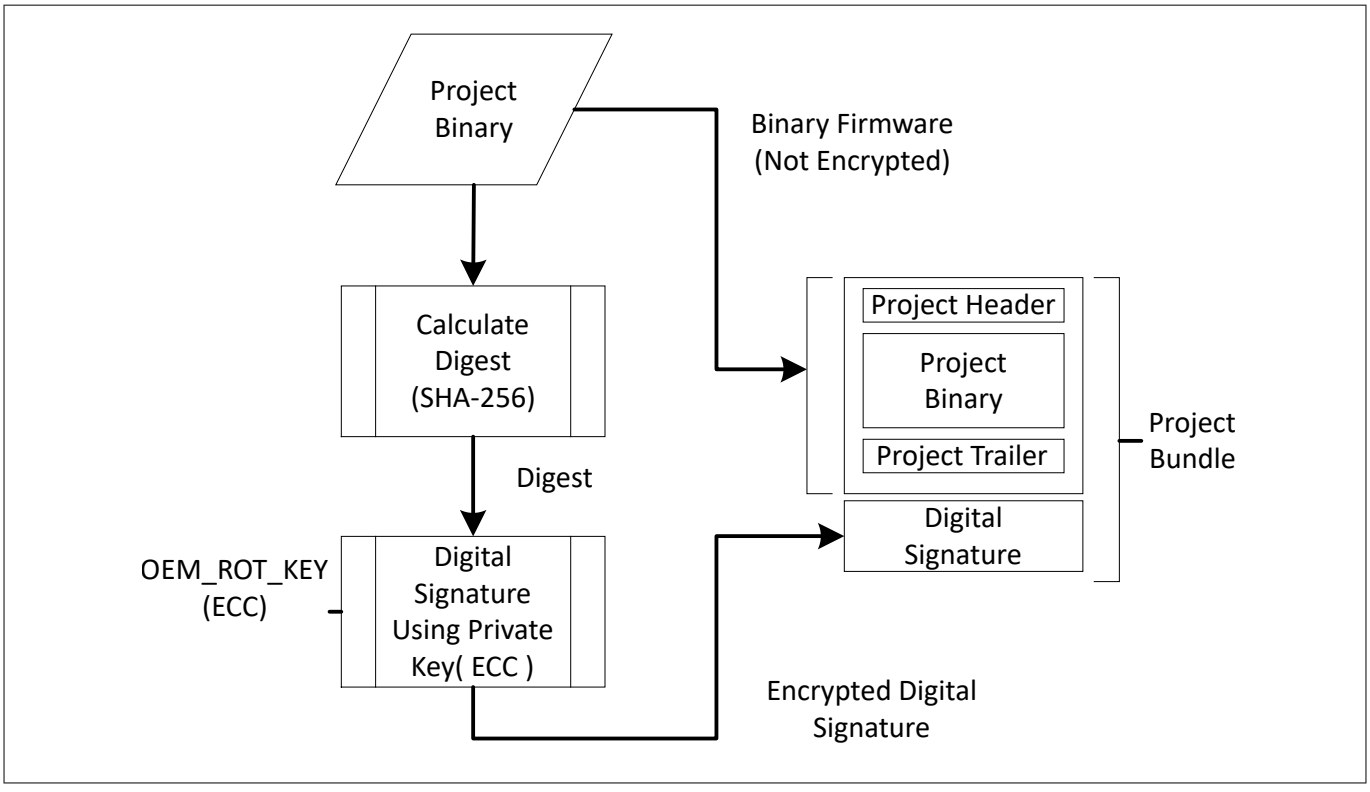



**Figure 12 Code signing**

There are two types of keys used in generating the digital signature of the Application.

1. **OEM Development private key (OEM_DEV_PRIV_KEY)** – This key is used to sign the application in development_LCS
2. **OEM ROT private key (OEM_ROT_PRIV_KEY)** – This key is used to sign the application in production LCS

Separate keys allows Customer to create multiple signed development versions of firmware with OEM_DEV_PRIV_KEY, while release versions will be signed by OEM_ROT_PRIV_KEY.

## 5.2 Code validation

When secured boot is enabled through provisioning, whether the device is in production or development LCS, the BootROM must verify the OEM code prior to execution. The project binary image must be signed by the OEM_ROT_PRIVATE_KEY before loaded into the device. BootROM verifies the OEM code by first calculating the hash of the project binary image. It then uses the calculated hash, the project digital signature that is appended to the end of the code, and the OEM_ROT_PUBLIC_KEY/OEM_DEV_PUB_KEY to verify if the OEM code is authentic. If the code fails verification, the BootROM will not jump to the user code but will wait in an endless loop until the device has been reset. Figure 13 shows a graphical representation of this process.

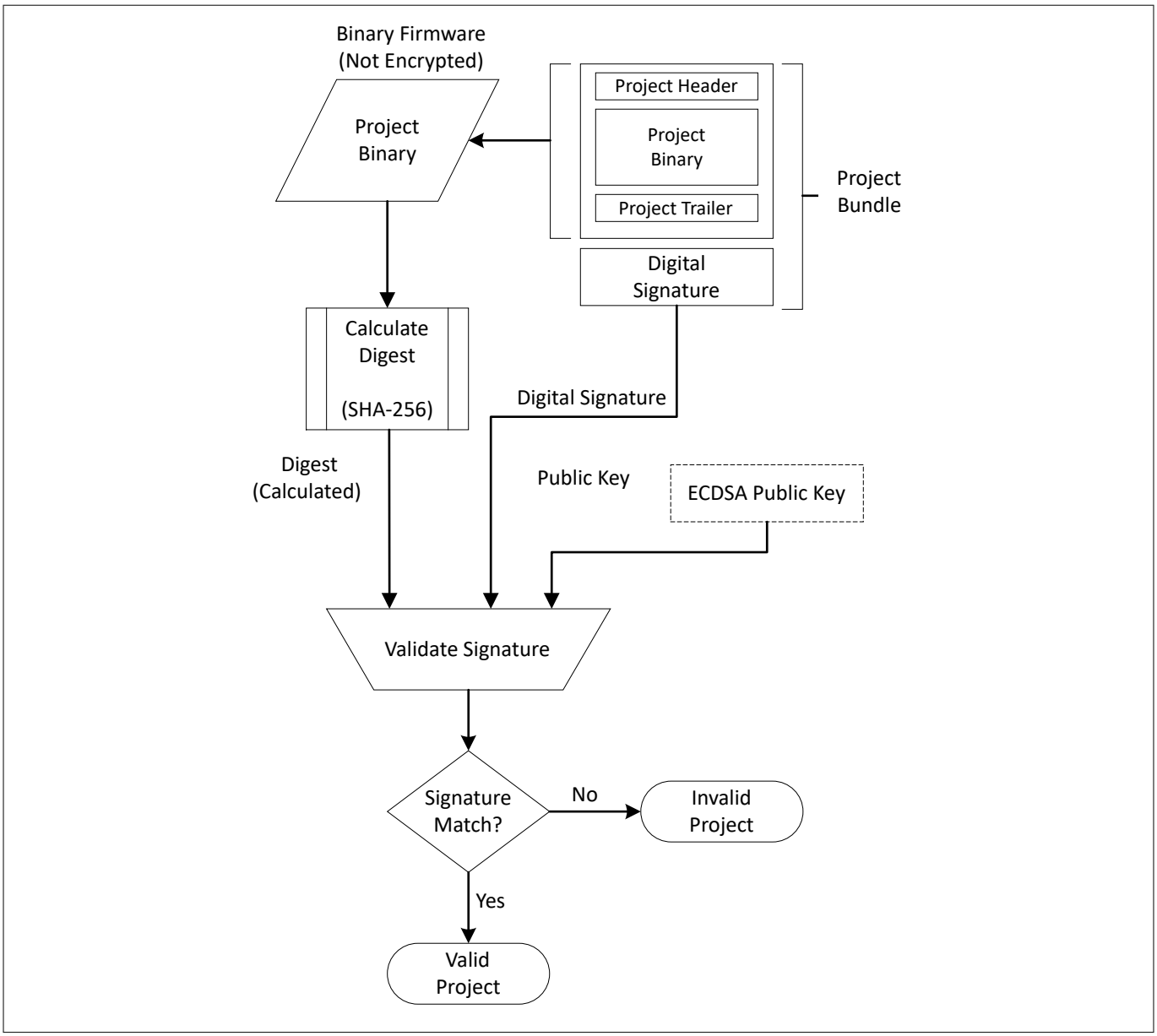



**Figure 13 ECDSA code verification**

# 6 Edge Protect Tools

## 6.1 Overview

Edge Protect Tools is a comprehensive security tools package built on Python, offering a command-line interface to interact with MCU for various purposes. These purposes include: provisioning, device integrity checks, RMA transition, and more. The tools are developed using the Python programming language and leverage several third-party Python modules to handle different tasks. For instance, these modules are used to manage configuration files in JSON format, object files in ELF format, and perform cryptographic operations.

Use any shell command-line tool to interact with Edge Protect Tools. However, Infineon recommends using modus-shell, which comes bundled with the ModusToolbox™ installation package for a seamless interaction experience. When executing commands that require interaction with the MCU, such as provisioning, the tools prepare the necessary provisioning packets and communicate with the device using programming tools like OpenOCD via the SWD interface or the DFU Host Tool via serial interfaces like I2C, SPI, and UART. Figure 14 shows the provisioning flow between OEM and MCU through Edge Protect Tool (EPT).

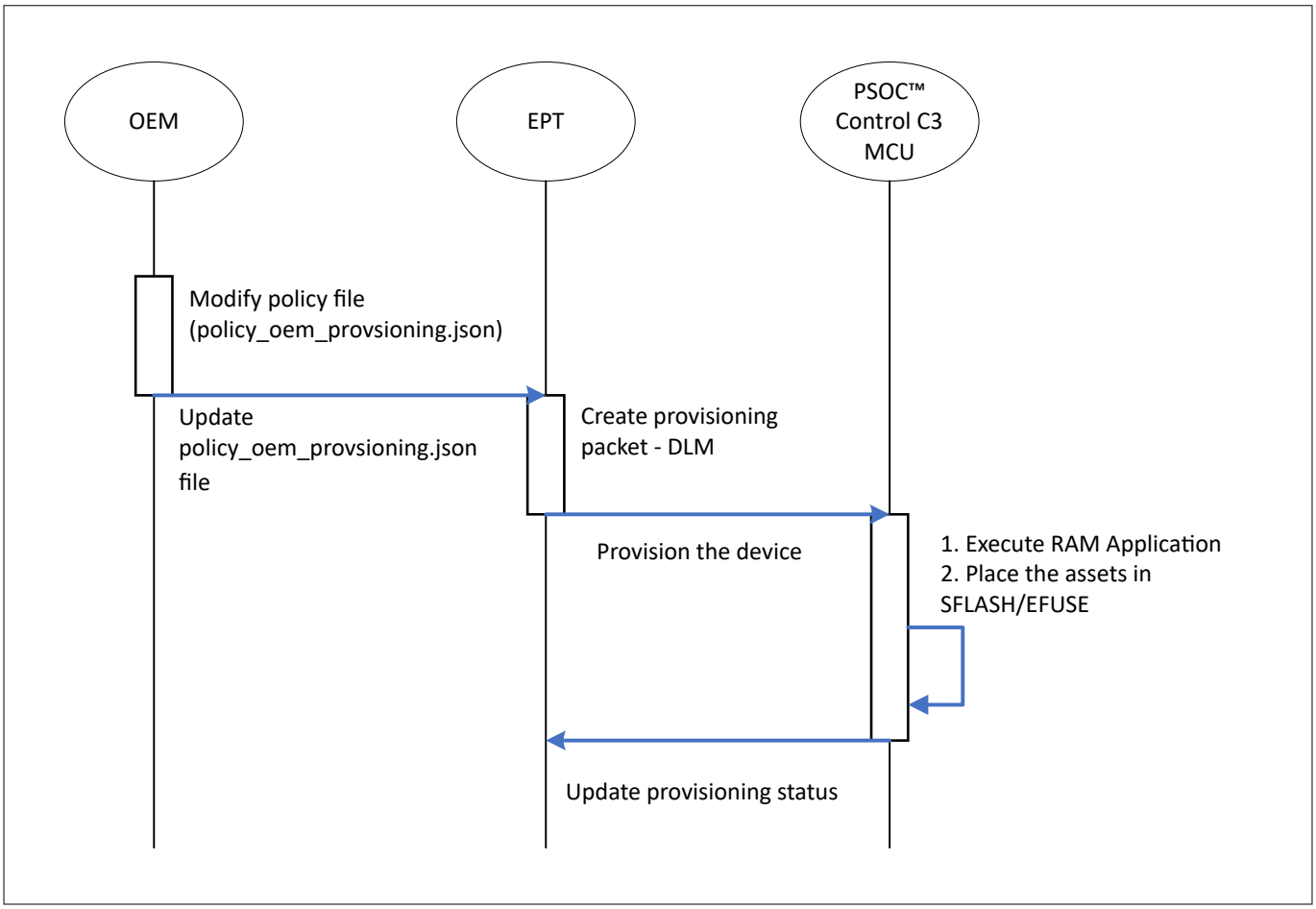



**Figure 14 PSC3M/P8, PSC3M/P7 MCUs provisioning flow**

The Edge Protect Tools package supports a range of operations that cater to various security-related needs. Some of the key operations supported by the tools include:

- Provisioning, which involves injecting keys and policy into the device, ensuring a secure setup
- Re-provisioning allows for modifying certain configurations post-provisioning and providing flexibility in device management
- Enables a Device Integrity Exam, which verifies the authenticity of the device and ensures it is in a known state, thereby preventing any potential security breaches

- In addition, the Edge Protect Tools facilitate key pair generation, creating key pairs that can be utilized for provisioning, and image signing purposes. Certificate generation is another supported operation, which generates certificates that can be used for debugging and RMA transition
- The tools support signing images with the OEM key, ensuring the integrity and authenticity of the images used in the device

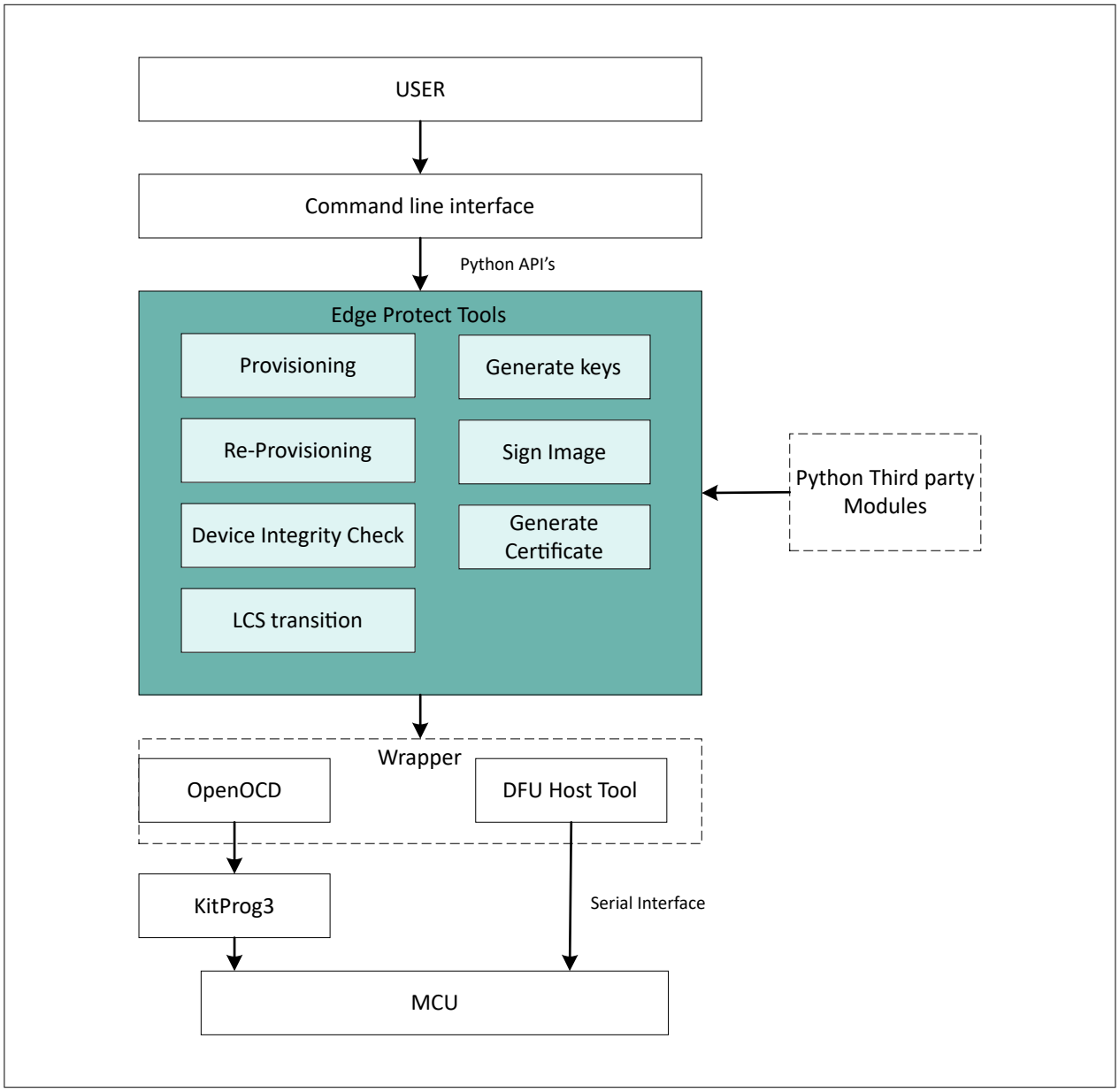



**Figure 15 Edge Protect Tools design flow**

## 6.2 Device communication

The Edge Protect Tools package supports communication with the device using programming tools such as the OpenOCD tool via the SWD interface on the device, or by utilizing the DFU Host Tool via serial interfaces like I2C, SPI, UART, or USB. The Edge Protect Tools package does not directly call the programming tool commands, but instead utilizes a wrapper mechanism to interface with the programming tools. This model has the potential to be further extendible to other programming tools, although this is currently outside the scope of the tool's current version.

## 6.3 Packaged assets

OEM policy - The OEM policy (`policy_oem_provisioning.json`) file can be used to configure the device policy configurations, which is located in the policy folder as part of the Edge Protect Tools package.

## 6.4 Edge Protect Tools package for PSC3M/P8, PSC3M/P7 MCUs

The Edge Protect Tools is packaged as a part of the Edge Protect Security Suite for the PSC3M/P8, PSC3M/P7 devices. Follow these steps to setup Edge Protect Security Suite.

1. Download and install the Infineon Developer Center Launcher
2. Login using your Infineon credentials
3. Download and install the “ModusToolbox™ Edge Protect Security Suite” from Developer Center Launcher

***Note**: The default installation directory of the Edge Protect Security Suite in Windows operating system is C:/Users/<USER>/Infineon/Tools. After installing the Edge Protect Security Suite, add the Edge protect tools executable to system PATH variable (/ModusToolbox-Edge-Protect-Security-Suite-<version>/tools/edgeprotecttools/bin)*

To utilize the executable file, find the executable file in the `./bin` directory, which is compatible with Windows, Linux, and macOS. To use it directly, add the executable path to your system's environment variable PATH.

***Note**: You need to install Python separately. The Edge Protect Tools support Python versions ranging from 3.8 to 3.12.*

Once you have installed the Edge Protect Tools package, initialize it for a specific target device. For example, to initialize it for the PSC3M/P8, PSC3M/P7 MCU devices, navigate to your working directory and execute the following command in a shell CLI:

```
edgeprotecttools.exe -t psoc_c3x8 init
```

After that, set up the OpenOCD path to be used by the tools. Run the following command:

```
edgeprotecttools.exe set-ocd --name openocd --path <openocd path>
```

replacing <openocd path> with the actual path to the OpenOCD tool on your system. After initialization, a specific folder structure will be created in your working directory.

To provision the device, execute a specific command using the Edge Protect Tools utility. The basic command is as follows:

```
edgeprotecttools -t psoc_c3x8 provision-device -p policy/policy_oem_provisioning.json
```

However, if you are using an external MiniProg, specify the probe ID in the command. To find the probe ID, run the following command:

```
~/ModusToolbox/tools_3.3/fw-loader/bin/fw-loader.exe --device-list
```

This will provide you with a connected device, including their respective probe IDs. If you have multiple MiniProgs connected to your system, specify the correct probe ID in the provision-device command. For example, m

```
edgeprotecttools -t psoc_c3x8 provision-device -p policy/policy_oem_provisioning.json --probe-
id 061003B803260400
```

If you want to lock in the policy parameters and prevent any further modifications to the configuration, you can set the `prov_oem_complete` parameter to `true` in the `policy_oem_provisioning.json` file and provision the device. Once this parameter is set to true, the device is essentially "locked" and cannot be provisioned again in normal provisioned LCS. This ensures that the configured policy parameters remain unchanged and secure.

**Code Listing 6**

```
"prov_oem_complete": {
    "description": "Indicates OEM assets are finalized in SFLASH",
    "warning": "Once the 'prov_oem_complete' parameter value is set to true, it cannot be
changed! No further updates to SFLASH are allowed",
    "value": true
},
```

To transition a device to the production LCS, set the `target_lcs` parameter to `SECURE` and provision the device. Once the device is moved to the secure LCS, it is locked into this stage and cannot be provisioned again. Additionally, it is mandatory that the user application be in MCUboot format, as the device will only launch authenticated applications during the secure LCS. This ensures that only trusted and verified applications can run on the device.

**Code Listing 7**

```
"device_policy": {
    "target_lcs": {
    "description": "The LCS to which the transition should be done",
    "applicable_conf": "NORMAL_PROVISIONED, SECURE",
    "value": "SECURE"
    },
```

***Note**:* *The terms NORMAL_PROVISIONED LCS and DEVELOPMENT LCS are interchangeable, and similarly, SECURE LCS is equivalent to PRODUCTION LCS. The key distinction between DEVELOPMENT LCS and PRODUCTION LCS remain in the requirement for secure boot. In DEVELOPMENT LCS, secure boot is not mandatory, whereas in PRODUCTION LCS, secure boot is a requirement.*

# 7 Processing, memory, and peripheral isolation

The Cortex® M33 CPU of PSC3M/P8, PSC3M/P7 MCUs has the Arm® TrustZone®-M security extension enabled. The extension allows partitioning of the system's resources into two worlds: Secure and Non-secure. Additionally, the extension introduces Secure and Non-secure states to the CPU. In the Secure state, the CPU has access to all the resources on the MCU while in the Non-secure state the CPU has access only to the Non-secure resources. The TrustZone® extension adds new access restrictions which isolates resources into Secure and Non-secure worlds at the hardware level.

The TrustZone®-M security extension enables you to create an isolated execution environment and execute Secure and Non-secure software on the same M33 CPU. Since the Secure and Non-secure worlds are isolated at a hardware level, there is no information leakage from Secure to Non-secure software. Typically the software, such as bootloader, cryptography service, secure storage service, etc. are executed in the Secure world. The Non-secure software handle communication with the external world and other user tasks. If the Non-secure software requires some service from Secure software, the TrustZone® allows it to create Secure entry points to call Secure software from Non-secure software.

The isolation provided by TrustZone® is address space based. The memory and peripherals are aliased into two regions in the address space, i.e., same memory/peripheral is mapped in multiple regions in the 4 GB address space. The aliasing is done using the bit 28 of the address. If bit 28 = 0, the address is Non-secure and if bit 28 = 1, the address is Secure. The CPU in Secure state must use the Secure alias for code execution and peripheral access, while in Non-secure state, the CPU must use the Non-secure alias. Table 10 shows how the 256 KB flash, 64 KB RAM, and peripherals of the MCU are mapped to multiple address ranges.

**Table 10 PSC3M/P8, PSC3M/P7 MCUs memory and peripheral address mapping**

| Memory/peripheral | Address range | Alias |
|---|---|---|
| Flash | 0x0200_0000 -0x0207_FFFF / 0x2200_0000 - 0x2207_FFFF | Non-secure |
| | 0x1200_0000 - 0x1207_FFFF / 0x3200_0000 - 0x3207_FFFF | Secure |
| RAM | 0x0400_0000 - 0x0401_FFFF / 0x2400_0000 - 0x2401_FFFF | Non-secure |
| | 0x1400_0000 - 0x1401_FFFF / 0x3400_0000 - 0x3401_FFFF | Secure |
| Peripherals | 0x4200_0000 - 0x43FF_FFFF | Non-secure |
| | 0x5200_0000 - 0x53FF_FFFF | Secure |

On top of TrustZone®, protection context (PC) is implemented in PSC3M/P8, PSC3M/P7 MCUs for additional security. Protection context is a protection mechanism, which provides the ability to restrict memory and peripheral resources between multiple masters or from the same master executing multiple applications. Protection context adds an additional attribute to bus masters on top of the Non-Secure (NS) and Privilege attribute. This additional attribute is used for isolating resources by memory protection controller (MPC) and peripheral protection controller (PPC).

After reset, the M33 CPU starts in Secure state and the BootROM executes in Secure state. The BootROM later transfers the control to the user application. The BootROM does not change the CPU state. The CPU continues to run in a Secure state, therefore, it is required that the first user application must be a Secure application. Depending on the use case, you can build the entire application as Secure software or it can be split into Secure and Non-secure software. If the application is split into Secure and Non-secure software, then ensure that the memory and peripherals are partitioned such that the Non-secure software has access to its resources. The next section describes how the memory is partitioned to enable Non-secure software execution.

***Note**: The majority of the code example for PSC3M/P8, PSC3M/P7 MCUs are demonstrated as "secure software".*

See the "TrustZone®" and "Protection units" sections from PSOC™ Control C3 PSC3M/P8, PSC3M/P7 MCUs architecture reference manual for more information about TrustZone® and protection context and its isolation capabilities.

## 7.1 Memory isolation

Memory isolation (partitioning) in PSOC™ C3 MCU is controlled with implementation-defined attribution unit (IDAU), secure attribution unit (SAU), and memory protection controller (MPC). The memory can be partitioned into Secure, Non-secure, or Non-secure callable (NSC: A special type of Secure region is used for calling Secure software from Non-secure software) regions by configuring these protection units.

**IDAU:** It provides a fixed (non-configurable) implementation to indicate whether an address is Secure or Non-secure. It indicates an address as Secure if the 28th bit of address is '1' else, it is marked as Non-secure. The security attribute indication by IDAU is aligned with the address aliasing scheme shown in Table 10.

**SAU:** It has eight programmable regions to indicate certain address ranges as Secure, Non-secure, or Non-secure callable. CMSIS provides the necessary assets (APIs and device headers) required to program SAU. SAU and IDAU are implemented as part of TrustZone® extension and the attribution units work together to indicate whether an address is Secure or Non-secure, the higher security among the two is used as indication.

**MPC:** It divides the flash and SRAM on to smaller blocks of 2 KB and configures Non-secure (NS), Read (R), and Write (W) attributes of each memory block for each protection context. If a memory block is marked as Secure, then it can be accessed by Secure alias address and if a memory block is marked as Non-secure, it can be accessed using Non-secure alias address. MPC configuration is stored in the MCU during the device provisioning process through the OEM policy.

The BootROM reads the OEM policy during device initialization and configures the SAU and MPC. After device initialization, the MPC configuration cannot be updated in the run time, the device must be re-provisioned to change it. SAU can be reconfigured in run time by Secure privileged software. Ensure that SAU and MPC configuration are aligned, that is they mark the same set of address range as Secure or Non-secure. In case of conflicting configuration, the program execution can end up in fault.

Depending on the application's needs, you can choose the isolation level required between Secure and Non-secure software. You can either rely only on TrustZone® or use the additional isolation capability of protection context on top of TrustZone®. If the context switching latency between Non-secure and Secure is critical, then it is recommended to use TrustZone® isolation, as the protection context switching adds an additional overhead, which cannot be desired in certain applications. If security is the utmost priority, then use the additional protection context isolation.

Taking an example to demonstrate how the device memory can be partitioned by configuring SAU and MPC. Assume that a use case requires both Secure (PC2) and Non-secure (PC3) projects and the memory requirements for this application are as shown in Figure 16.

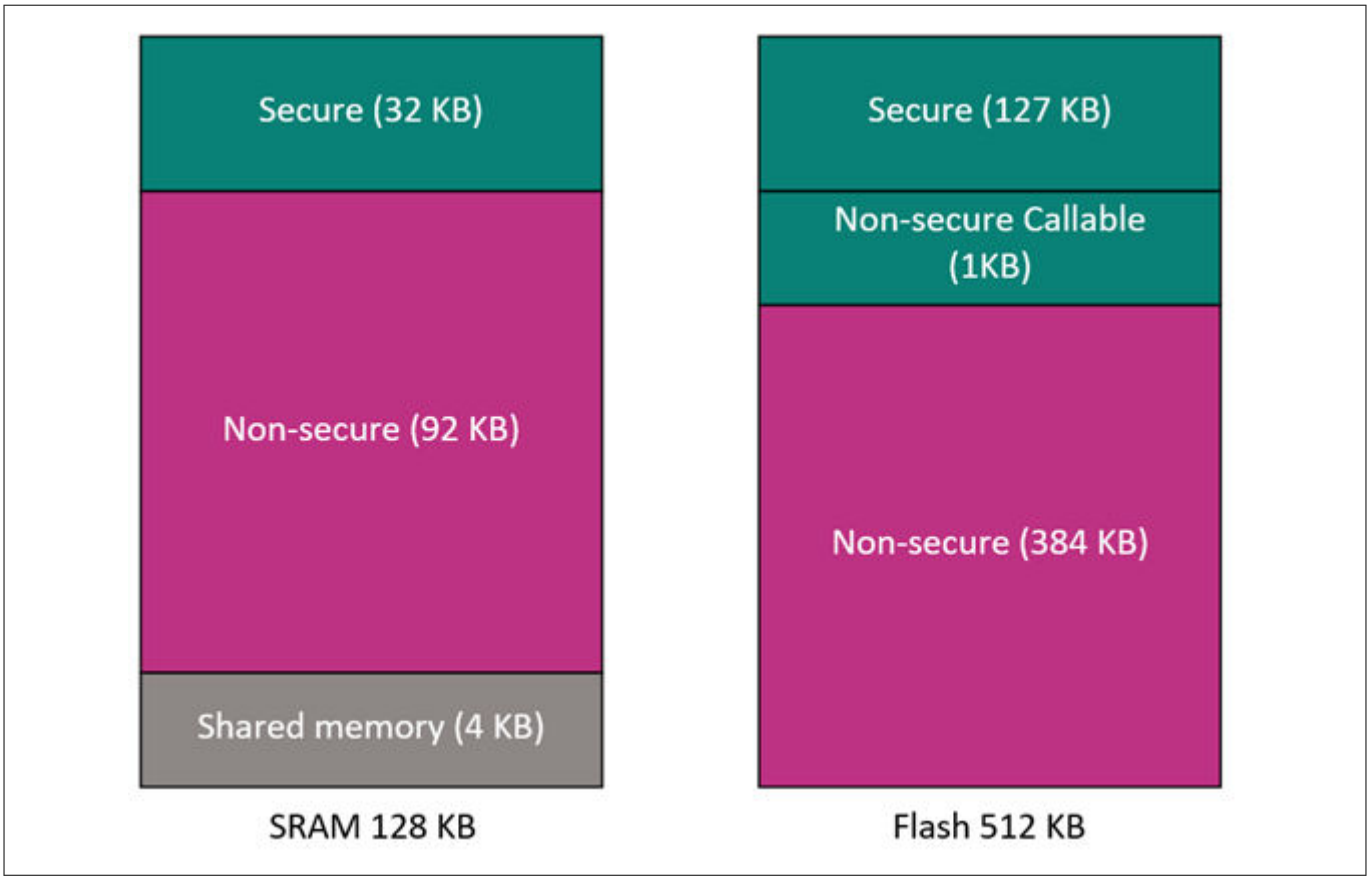



**Figure 16 Example memory partitioning**

By default, TrustZone® marks the entire memory as secure. SAU registers must be programmed with CMSIS to configure Non-secure and Non-secure callable regions. On Arm® TrustZone® technology, see the Armv8-M architecture reference manual and see how SAU registers are configured to create Non-secure and Non-secure callable regions.

## 7.2 Peripheral isolation

The peripherals of the PSC3M/P8, PSC3M/P7 MCUs are mapped and aliased into the 0x4200_0000 - 0x43FF_FFFF/0x5200_0000 - 0x53FF_FFFF address space. Similar to memory, the peripherals can be configured for Secure or Non-secure access. Partitioning using TrustZone® alone is not enough because of limited SAU entries. For fine grain access control, PPC is available in the MCU. The PPC divides the peripheral address space into fixed size smaller regions. A PPC region protects the MMIO registers of a peripheral. PPC is configurable at run time by Secure privileged software. For each PPC region, configure the NS attribute, PC_MASK (which PCs can access the region), and privileged/unprivileged access restrictions. Secure peripherals can be accessed by Secure software using Secure alias address and Non-secure peripherals can be accessed by both Secure and Non-secure software using Non-secure alias.

### 7.2.1 Security aware peripheral protection controller (PPC) regions

There are certain special PPC region whose access control rules differ from the normal PPC region. If the special, aka security aware, PPC region's NS attribute is configured as Non-secure, then the PPC allows the peripheral to be accessed from both Secure and Non-secure alias. The security aware PPC regions are implemented for peripherals, which either understand the Non-secure access attribute or shared between Secure and Non-secure software.

## 7 Processing, memory, and peripheral isolation

The I/O subsystem of the PSC3M/P8, PSC3M/P7 MCUs are security aware peripheral. The I/O subsystem (IOSS) can differentiate between Secure and Non-secure access and can take the decision whether to respond to access. Each I/O pin of the device can be configured as Secure or Non-secure. The behavior of IOSS is as follows:

- If a pin is configured as Secure, it can be accessed only with a Secure alias address
- If a pin is configured as Non-secure, it can be accessed only with a Non-secure alias
- If a Secure pin is accessed with Non-secure alias address or a Non-secure pin is accessed with Secure alias address, the pin's response is Read Zero Write Ignore (RZWI)

This behavior ensures that if Secure and Non-secure pins are configured on the same GPIO port, then a Secure access to the port does not affect the Non-secure pin and a Non-secure access does not affect the Secure pins. The PSOC™ Control MCU: Security aware GPIO code example demonstrates how the PPC can be configured for IOSS to configure a Secure and Non-secure pin on the same port. The example also demonstrates how to configure interrupts for Secure and Non-secure pins.

Security aware PPC regions are also available for Inter-Processor Communication (IPC) peripheral, which can be used for communication between Secure and Non-secure software. In such cases, an IPC channel will be shared between Secure and Non-secure worlds and it should be accessible to both the worlds.

To know more about protection units, see the "Protection units" section of the PSOC™ Control C3 (PSC3P7xx, PSC3M7xx, PSC3P8xx, PSC3M8xx) MCUs architecture reference manual.

# 8 Firmware upgrade

The PSC3M/P8, PSC3M/P7 MCUs offer firmware upgrade capabilities through its Device Firmware Update (DFU) serial interface, which can be accessed via two distinct methods.

1. The first method involves utilizing the BootROM DFU serial interface, which provides a direct pathway for upgrading the firmware
2. The second method uses the user application (DFU MW: device firmware upgrade middleware) serial interface, offering an alternative route for firmware upgrades

## 8.1 BootROM DFU serial interface

For factory programming and upgrading images with reset, the BootROM-based method is highly recommended. This approach leverages the immutable BootROM, which has built-in support for Infineon's custom serial DFU protocol, enabling firmware upgrades. The DFU serial interface only supports the UART transport and the firmware can be sent to the device over DFU via DFU Host Tool. This is a graphical interface tool that runs on the host PC.

Once the firmware is loaded through the DFU Host Tool, the BootROM optionally authenticates (if provisioned for SECURE_APP) it based on the boot configuration and successfully launches the user application. Figure 17 shows the firmware upgrade process through the BootROM DFU flow.

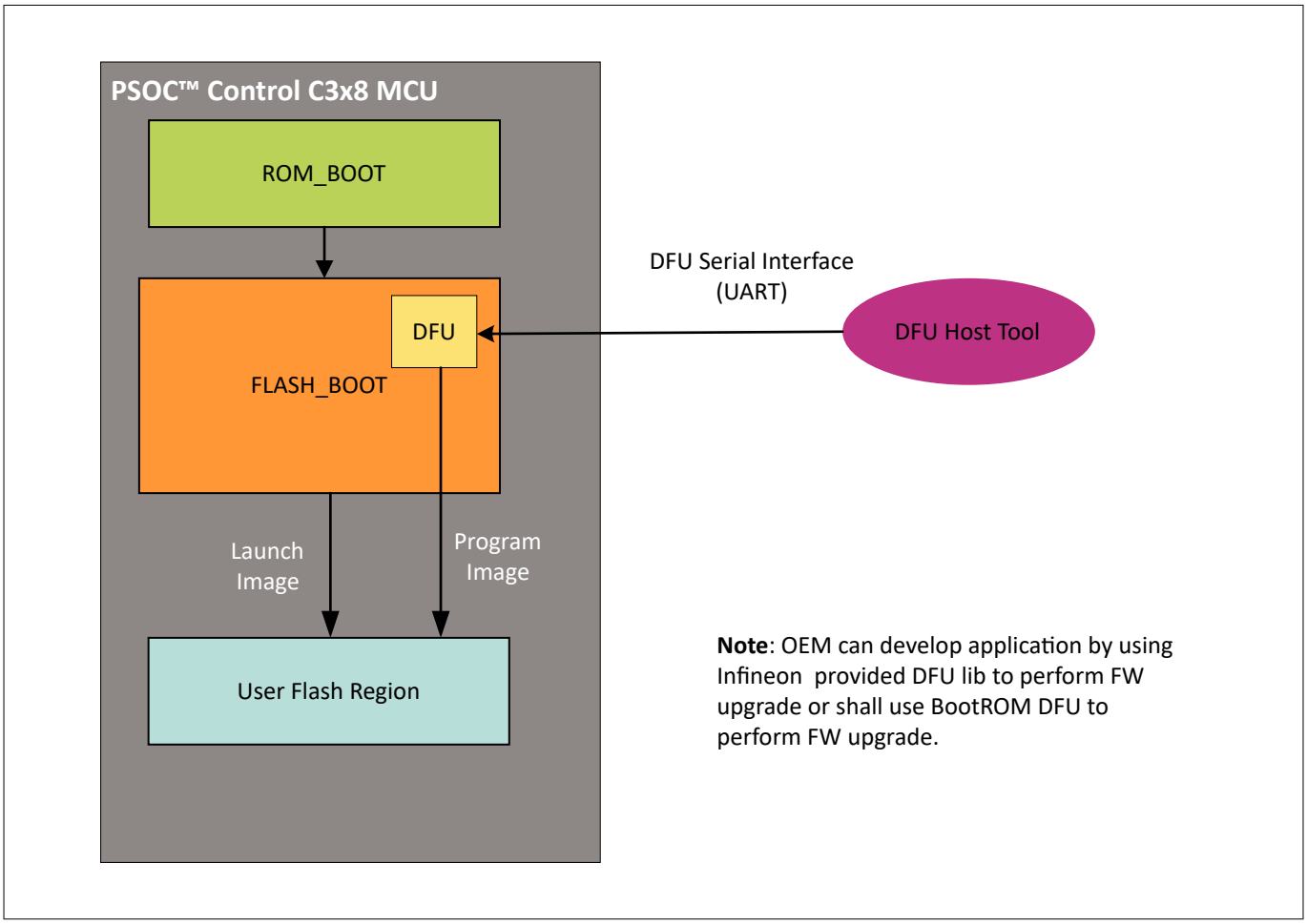



**Figure 17 Firmware upgrade through BootROM DFU**

## 8.2 User application (DFU MW) serial interface

For field firmware upgrades, the user application (DFU MW) method is highly recommended. This approach allows to develop the end application using the DFU MW library, which is provided by Infineon as part of the ModusToolbox™ Library manager. Alternatively, you can create your own custom protocol to perform firmware upgrades. Figure 18 shows the firmware upgrade process through the user application using the DFU MW protocol library. Notably, the DFU MW serial interface supports a range of ports, including UART, I2C, SPI, PMBus and CAN FD.

One of the key benefits of using the DFU MW library is that it abstracts the underlying complexity of the firmware upgrade process, providing a higher-level interface to interact with. This makes it easier to integrate firmware upgrade capabilities into the user application, without requiring extensive knowledge of the underlying hardware or firmware upgrade protocols.

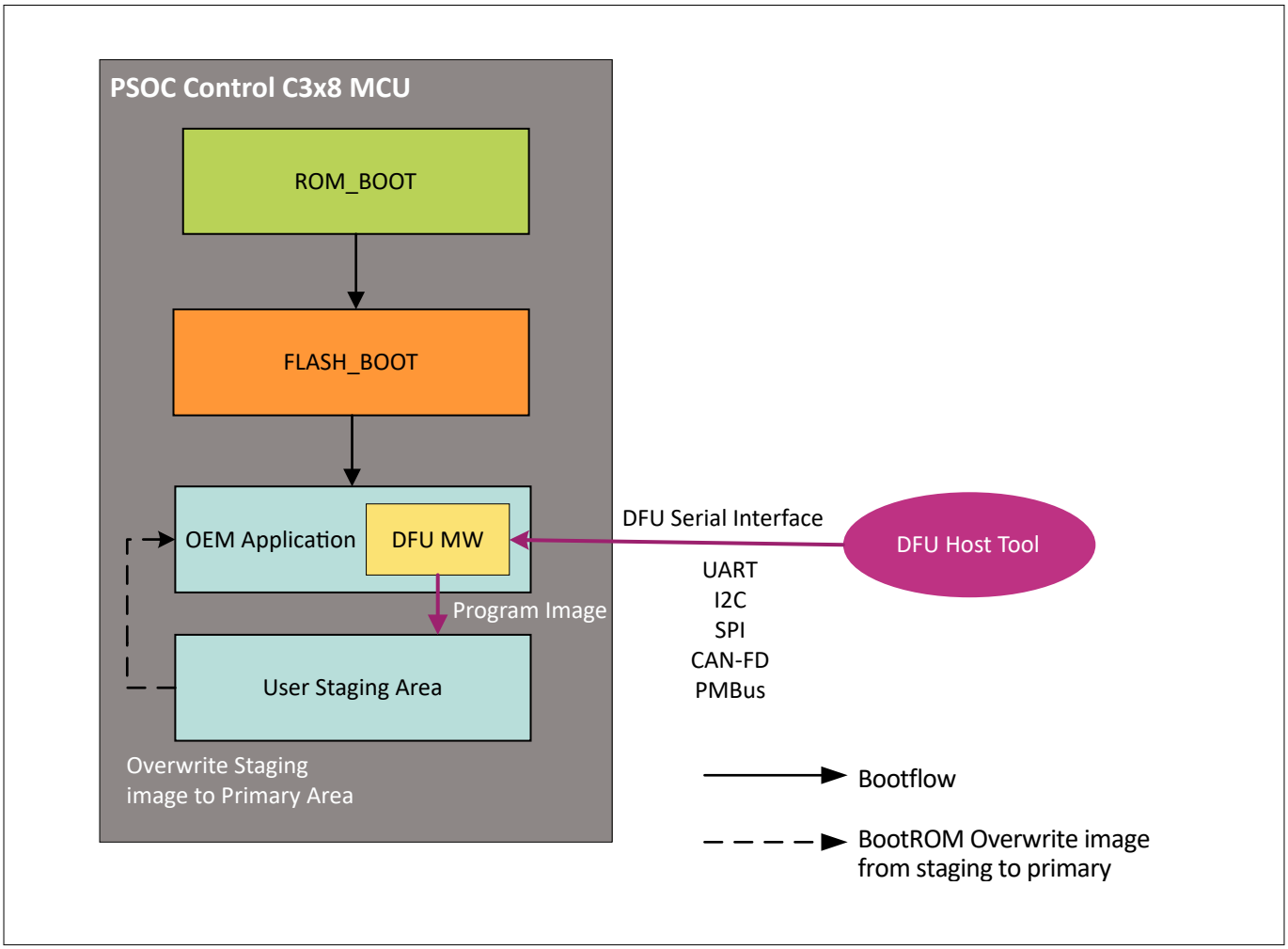



**Figure 18 Firmware upgrade through DFU middleware**

In both the BootROM-based and user application (DFU MW) methods, the Infineon provided PC-based DFU Host tool supports transferring the firmware image to the device. The tool uses the DFU protocol format to transfer the image packet by packet, ensuring a reliable and efficient upgrade process.

The boot configuration of the device determines whether the new image should be signed or unsigned. There are two possible boot configurations: SIMPLE_APP and SECURE_APP.

For the SIMPLE_APP configuration, the BootROM does not perform authentication of the image, and the image is expected to be in raw format. This means that the image is not encrypted or signed, and the BootROM will load it directly into the device.

In contrast, for the SECURE_APP configurations, the image must be signed with the MCUboot format. This ensures that the image is authenticated and verified before it is loaded into the device. The Infineon provided

## 8 Firmware upgrade

EdgeProtectTool is a useful utility that supports signing the image using the MCUboot format, making it easier to create secure and authenticated firmware images.

Figure 19 shows the MCUboot image structure.

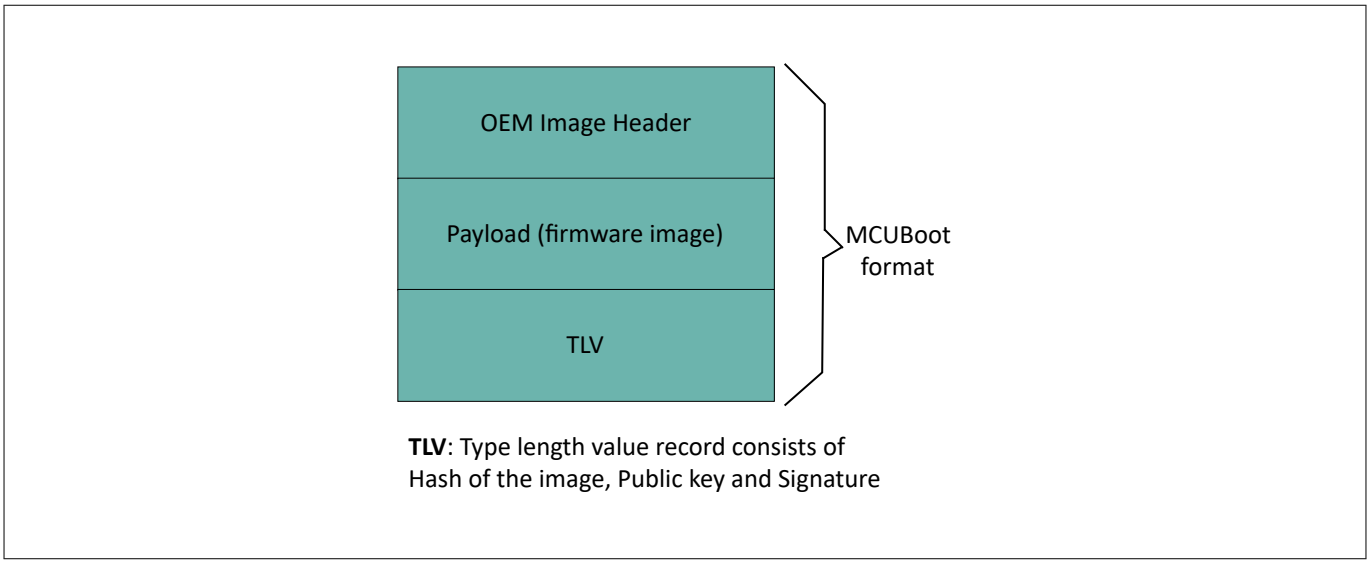



**Figure 19 MCUboot format**

Once the firmware image is prepared, it can be loaded onto the device using the DFU Host Tool via a serial interface. This interface enables the transfer of the image to the device, where it can be authenticated and loaded into memory.

For more information on the image signing process, it is recommended to see the PSOC™ Control MCU: Basic secure application code example and the Edge Protect Tools section. These resources provide detailed guidance on how to sign the image using the MCUboot format, ensuring that the firmware is secure and authenticated.

The PSOC™ Control MCU: Basic secure application code example offers a comprehensive overview of the secure boot process, including the steps involved in signing and verifying the firmware image. The Edge Protect Tools section, on the other hand, provides a detailed guide on how to use the Edge Protect Tool to sign and encrypt the firmware image.

# 9 Debug token

A debug token is a signed package stored in RAM that contains values in the control word, which are used to modify debug port settings in AP_CTL. However, note that a debug token cannot override AP that has been permanently disabled by the debug policy.

The debug token serves as a versatile tool, enabling the device to transition into RMA mode, open the Device Access Port (DAP), or facilitate debugging of RAM, or user applications. Notably, the format of the debug token remains consistent across various use cases, including debugging RAM applications, user applications, transitioning to RMA, and opening DAPs during the RMA lifecycle.

Ensure the authenticity of these tokens by signing them with the appropriate keys. Specifically, both the debug token and the transit to RMA token must be signed with the OEM key, whereas the open RMA token requires signing with the Infineon vendor (IC vendor) key.

The debug certificate format is a critical aspect of this process. The debug token must include the public key, which allows the BootROM to verify the digital signature by matching it with the stored hash. The boot firmware determines which public key hash to use for verification based on the flow, whether it is the OEM or ICV public key embedded in the debug token.

Debug tokens provide access to RAM and MMIO registers through Sys-AP, while CM33-AP offers additional access to CM33 internals, including registers and Arm® MMIO.

## 9.1 Debug token use cases

There are several use cases for debug tokens, including:

- **OEM debug token:** Signed by the OEM_ROT_KEY, used for OEM-specific debugging
- **To RMA token:** A debug token for transitioning to RMA LCS, signed by the OEM_ROT_KEY, and used as an input parameter for cyapp_to_rma
- **OpenRMA token:** A debug token to open access to the system in RMA LCS, signed by the ICV_ROT_KEY, and authenticates that only ICV has full access to the system in RMA LCS

The BootROM detects whether a certificate should be signed by the OEM_ROT_KEY based on the BOOT_DLM_CTL.REQUEST value, with two indicating an OEM debug token signed by OEM_ROT_KEY.

**Code Listing 8: Debug token template**

```
{
    "version": "1.0.0.0",
    "device_id": {
        "silicon_id": "0xFFFF",
        "family_id": "0xFFF",
        "revision_id": "0x01"
    },
    "control_word": {
        "cm33_ap": "Enable",
        "sys_ap": "Enable",
        "rma": "Enable"
    },
    "die_id": {
        "max": {
            "day": 255,
            "lot": 16777215,
            "month": 255,
            "wafer": 255,
            "xpos": 255,
            "year": 255,
            "ypos": 255,
            "sort": 255
        },
        "min": {
            "day": 0,
            "lot": 0,
            "month": 0,
            "wafer": 0,
            "xpos": 0,
            "year": 0,
            "ypos": 0,
            "sort": 0
        }
    }
}
```

## 9.2 Generating a debug certificate

The template information is available in the `project/packets/debug_token.json`. Before generating a debug certificate, complete the following steps:

1. Manually enter the device_id data, which can be obtained by executing the device-info command

```
$edgeprotecttools -t psoc_c3 device-info
```

2. Configure the control_word section according to your specific requirements
3. Manually provide the die_id data, which can be retrieved by running the die-id command

```
$ edgeprotecttools -t psoc_c3 read-die-id
```

4. After these steps are complete, you can use the CLI or API to create the signed debug certificate.
   a. Create a non-signed token

```
$edgeprotecttools -t psoc_c3 debug-token --template packets/debug_token.json --
output debug_token_unsigned.bin
```

   b. Get the signature using HSM
   c. Add a signature returned by the HSM to the token.

```
edgeprotecttools merge-bin --image debug_token_unsigned.bin --image
signature_hsm.bin --output debug_token_signed.bin
```

   d. If you need to generate a key and token for development purposes, utilize the Edge Protect Tools and execute the following command, which will facilitate the generation of the required key and token.

```
edgeprotecttools -t psoc_c3 debug-token –template packets/debug_token.json -o
packets/debug_token.bin --key keys/oem_rot_priv_key_0.pem
```

5. Configuring the policy
   a. By default, the CM33_DAP and SYS_DAP debug ports are open, which poses a security risk. To secure these ports, configure the debug section of the policy accordingly. To enable authenticated debug, modify the debug policy for CM33_DAP and SYS_DAP as follows:
   b. In this configuration, set CM33_DAP to 'Enable', which allows CM33 debugging. However, the 'Enable' field is then set to 'Disable', indicating that a valid debug certificate is required to access the debug ports. Note that if you set the field to 'Permanently Disable', even a valid certificate will not grant access to the CM33 port
   c. Table 11 summarizes the debug port status based on the debug policy parameter configuration:

**Table 11 Debug port status and debug port policy**

| Use case | ap_cm33 | system ap | Debug port status |
|---|---|---|---|
| 1 | Enable | Enable | Port open, user debug the device |
| 2 | Disable | Enable | Port closed, the user cannot debug the device (but can open with a valid debug token/certificate) |
| 3 | Permanently Disable | Enable | Port closed, the user cannot debug the device (even with a valid debug token/certificate) |
| 4 | Enable/ Disable/ Permanently Disable | Disable | Port closed, the user cannot debug the device (with or without a valid debug token/certificate) |

   d. Figure 20 displays the error log in the ModusToolbox™ console for the scenario where the debug port is closed (use case 2) and no valid debug certificate is provided

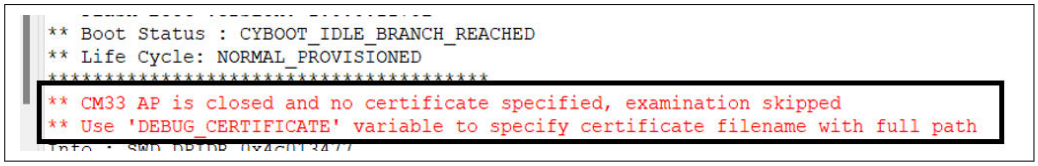



**Figure 20 Console log for debug port closed**

6. Place the generated file in the ModusToolbox™ debug configuration (Go to **Project** > **Debug Configurations** > **Debugger**) as shown in Figure 21

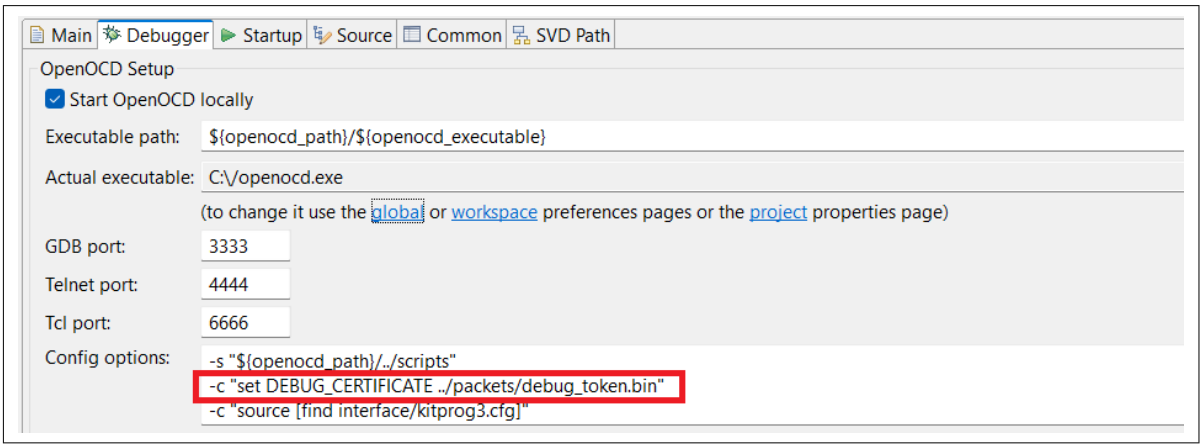



**Figure 21 Debug certificate navigation in ModusToolbox™**

7. Figure 22 displays the log in the ModusToolbox™ console for the scenario where the debug port is open (use case 2) and a valid debug certificate is provided

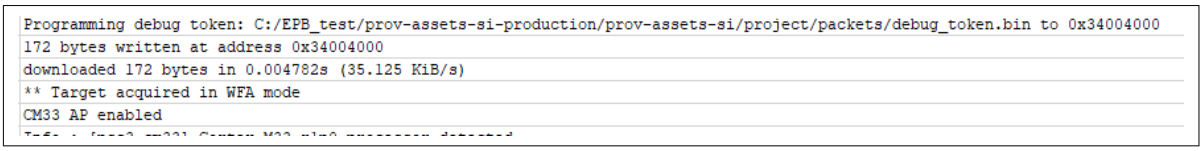



**Figure 22 Console log for CM33 AP port enabled**

## 9.3 Generating RMA certificate

1. To create an RMA token
   a. Ensure the "rma" field in the *packet/debug_token.json* template is *Enable*
   b. From the **project** directory, run the following command

```
edgeprotecttools -t psoc_c3 debug-token --template packets/debug_token.json --key
keys/oem_rot_priv_key_0.pem -o packets/debug_token.bin
```

2. To transition the device to RMA LCS
   a. Ensure that the device is in NORMAL_PROVISIONED or SECURE LCS
   b. For transition into RMA LCS, use the token generated in the previous step. From the **project** directory, run the following command

```
edgeprotecttools -t psoc_c3 transit-to-rma --token packets/debug_token.bin
```

3. Default `debug_token.json` file

## 9 Debug token

### Code Listing 9

```
{
  "version": "1.0.0.0",
  "silicon_id": "0xEE40",
  "family_id": "0x0118",
  "si_revision_id": "0x11",
  "control_word": {
    "cm33_ap": "Enable",
    "sys_ap": "Enable",
    "cm33_dbg": "Enable",
    "cm33_nid": "Enable",
    "cm33_spid": "Enable",
    "cm33_spnid": "Enable",
    "cm33_secure": "Enable",
    "sys_secure": "Enable",
    "rma": "Enable"
  },
  "die_id": {
    "min": {
      "lot": 0,
      "wafer": 0,
      "xpos": 0,
      "ypos": 0,
      "sort": 0,
      "minor": 0,
      "day": 0,
      "month": 0,
      "year": 0
    },
    "max": {
      "lot": 16777215,
      "wafer": 255,
      "xpos": 255,
      "ypos": 255,
      "sort": 255,
      "minor": 255,
      "day": 255,
      "month": 255,
      "year": 255
    }
  }
}
```

# 10 Edge Protect Bootloader

The Edge Protect Bootloader (EPB) is a secure bootloader designed to provide a robust and reliable solution for the PSC3M/P8, PSC3M/P7 MCU. The key benefits of Edge Protect Bootloader is that that Infineon provides it with its source code, giving you the flexibility to choose and customize the bootloader according to your end-product requirements. This bootloader offers a range of advanced features, including seamless software upgrades, image encryption, serial logging, and upgrade by swap capabilities.

## 10.1 Boot flow

OEM will use the Edge Protect Bootloader as an optional bootloader. Its primary functions include: initializing the minimal required hardware, authenticating the next stage application, and launching the application.

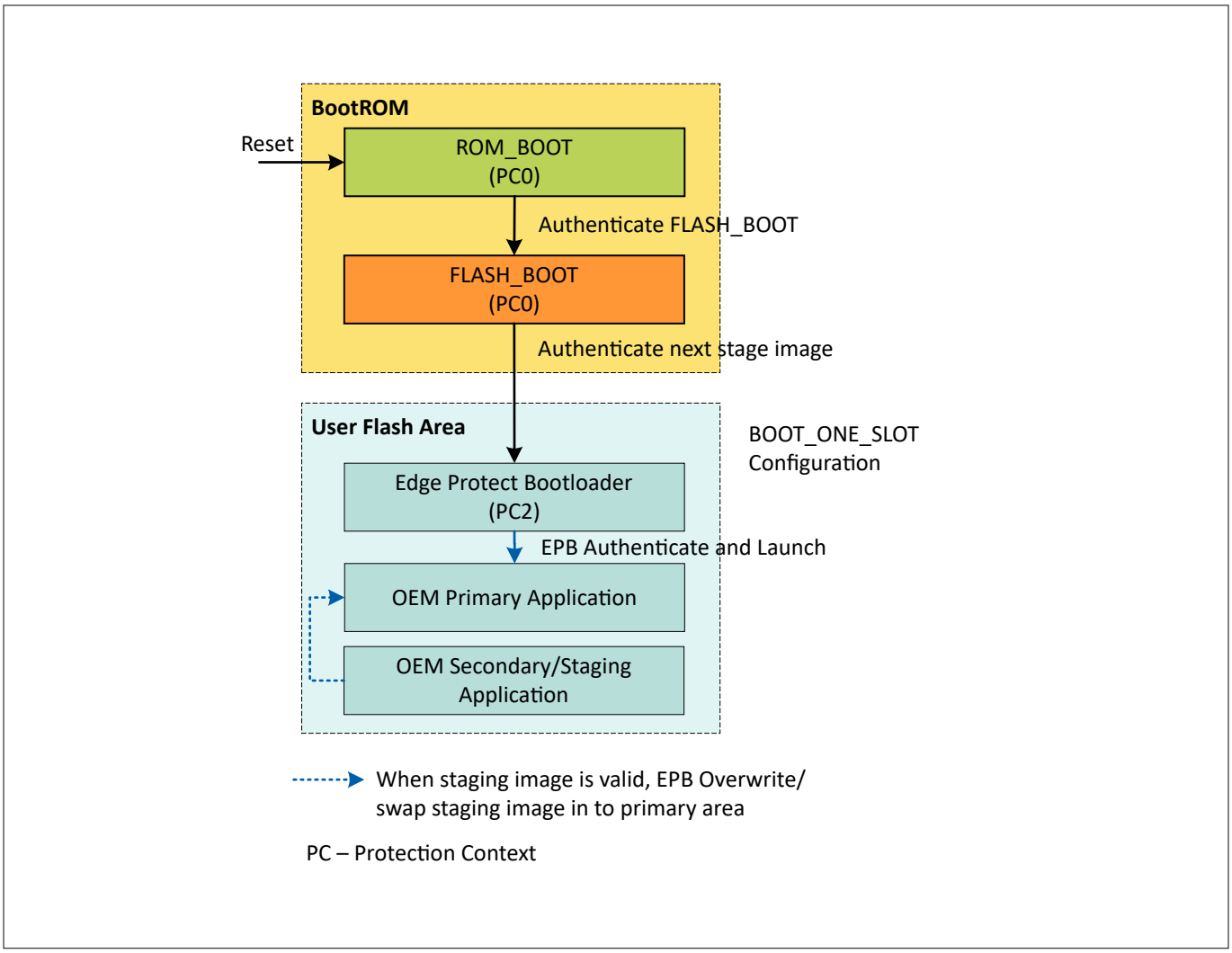



**Figure 23 Edge Protect Bootloader with boot one slot configuration**

## 10.2 Architecture

Edge Protect Bootloader is based on an open-source MCUboot bootloader library as its foundation and integrates with a range of other components. These include Mbed TLS for secure communication, cryptography hardware driver for cryptographic operations, retarget-io for input/output management, and Peripheral Driver Library (PDL) for peripheral control.

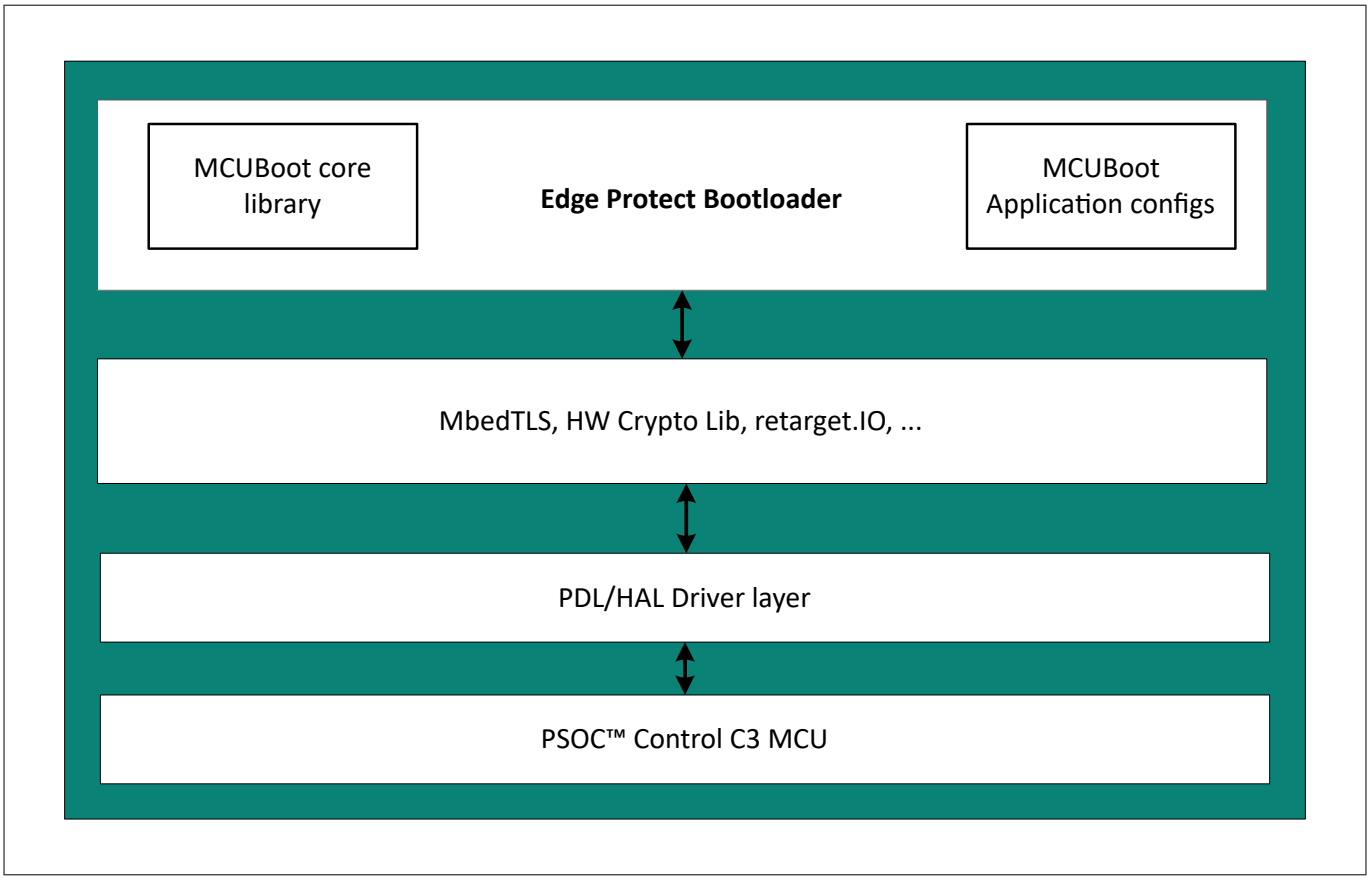



**Figure 24 Edge Protect Bootloader architecture**

The Edge Protect Bootloader always boots the application from the primary slot and overwrite/swap the image from the secondary slot into the primary slot when an upgrade is requested.

## 10.3 Edge Protect Bootloader features

The Edge Protect Bootloader offers customizable features that can be tailored to meet specific user requirements through the use of JSON configuration files. These JSON files are located in the `platforms\PSC3\memory_maps` directory and are denoted by the `*.json` extension. By modifying these files, you can configure the Edge Protect Bootloader's features to suit your needs.

After the feature set has been modified, it is necessary to rebuild the application, and reload it onto the device to apply the changes. This ensures that the updated configuration takes effect and the Edge Protect Bootloader operates according to the new settings.

**Table 12 Feature configuration through JSON file**

| File name | Description |
|---|---|
| `feature_config.json` | Features can be configured by 'set ' or 'false' |
| `platform.json` | Edge Protect Bootloader memory attributes can be configured in this file |
| `overwrite_single_flash.json` | Single image firmware upgrade by overwrite feature can be configured in this file |
| `overwrite_multi2_flash.json` | Multi image firmware upgrade overwrite feature can be configured in this file |
| `swap_single_flash.json` | Single image firmware upgrade by swap feature can be configured in this file |
| `swap_multi2_flash.json` | Multi image firmware upgrade by swap feature can be configured in this file |

### 10.3.1 Power fail-safe update

If the Edge Protect Bootloader resets in the middle of a swap operation, the two images may be discontiguous in user flash memory. Bootutil recovers from this condition by using the image trailers to determine how the image parts are distributed in flash. The first step is to determine where the relevant swap status region is located. Because this region is embedded within the image slots, its location in the flash changes during a swap operation, see the reset recovery. After the swap operation has been complete, the Edge Protect Bootloader proceeds as though it had just been started.

### 10.3.2 Multi image authentication

The Edge Protect Bootloader offers enabling authentication and upgrade of multiple images before launching the primary application. This process can be configured to either overwrite or swap images, depending on the specified settings. These settings are defined through a JSON file, which provides a flexible and customizable approach to image management.

- In the case of the multi-image overwrite method, users are required to configure the `overwrite_multi2_flash.json` file

## 10 Edge Protect Bootloader

### Code listing 10

```
{
    "bootloader":
    {
        "bootloader_area":
        {
            "address"           : "0x32000000",
            "size"              : "0x18000"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
        },

        "shared_data":
        {
            "address"           : "0x3400F000",
            "size"              : "0x1000"
        }
    },

    "application_1":
    {
        "slots":
        {
            "boot"              : "0x32018000",
            "upgrade"           : "0x32020000",
            "size"              : "0x8000"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
        }
    },

    "application_2":
    {
        "slots":
        {
            "boot"              : "0x32028000",
            "upgrade"           : "0x32030000",
            "size"              : "0x8000"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
```

```
            }
        }
    }
```

- In the case of the multi-image swap method, configure the `swap_multi2_flash.json` file. These files serves as a crucial component in the image upgrade process, allowing you to tailor the overwrite or swap behavior to the specific needs

## 10 Edge Protect Bootloader

### Code Listing 11

```
{
    "bootloader":
    {
        "bootloader_area":
        {
            "address"           : "0x32000000",
            "size"              : "0x18000"
        },

        "scratch_area":
        {
            "address"           : "0x32038000",
            "size"              : "0x2000"
        },

        "status_area":
        {
            "address"           : "0x3203A000",
            "size"              : "0x2800"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
        },

        "shared_data":
        {
            "address"           : "0x3400F000",
            "size"              : "0x1000"
        }
    },

    "application_1":
    {
        "slots":
        {
            "boot"              : "0x32018000",
            "upgrade"           : "0x32020000",
            "size"              : "0x8000"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
        }
    },

    "application_2":
```

```
    {
        "slots":
        {
            "boot"              : "0x32028000",
            "upgrade"           : "0x32030000",
            "size"              : "0x8000"
        },

        "ram":
        {
            "address"           : "0x34000000",
            "size"              : "0xF000"
        }
    }
}
```

To ensure a secure boot process with Edge Protect Bootloader, the device must be provisioned with the BOOT_ONE_SLOT configuration. This setup enables the BootROM to authenticate the first user image, which in this scenario is the Edge Protect Bootloader, and then the Edge Protect Bootloader is responsible for authenticating further additional images that are loaded. This hierarchical approach provides an added layer of security and ensures that only trusted images are executed on the device.

### 10.3.3 Upgrade by overwrite

The Edge Protect Bootloader always boot from the primary slot and when an upgrade is requested, it copies the new image from the secondary slot into the primary slot. There are two possible upgrade modes: overwrite-based and swap-based.

In overwrite-based mode, the existing image in the primary slot is replaced and if the new image has issues, there is no possibility of rolling back to the previous version. In swap-based mode, the Edge Protect Bootloader preserves the original image, allowing for a potential rollback if needed.

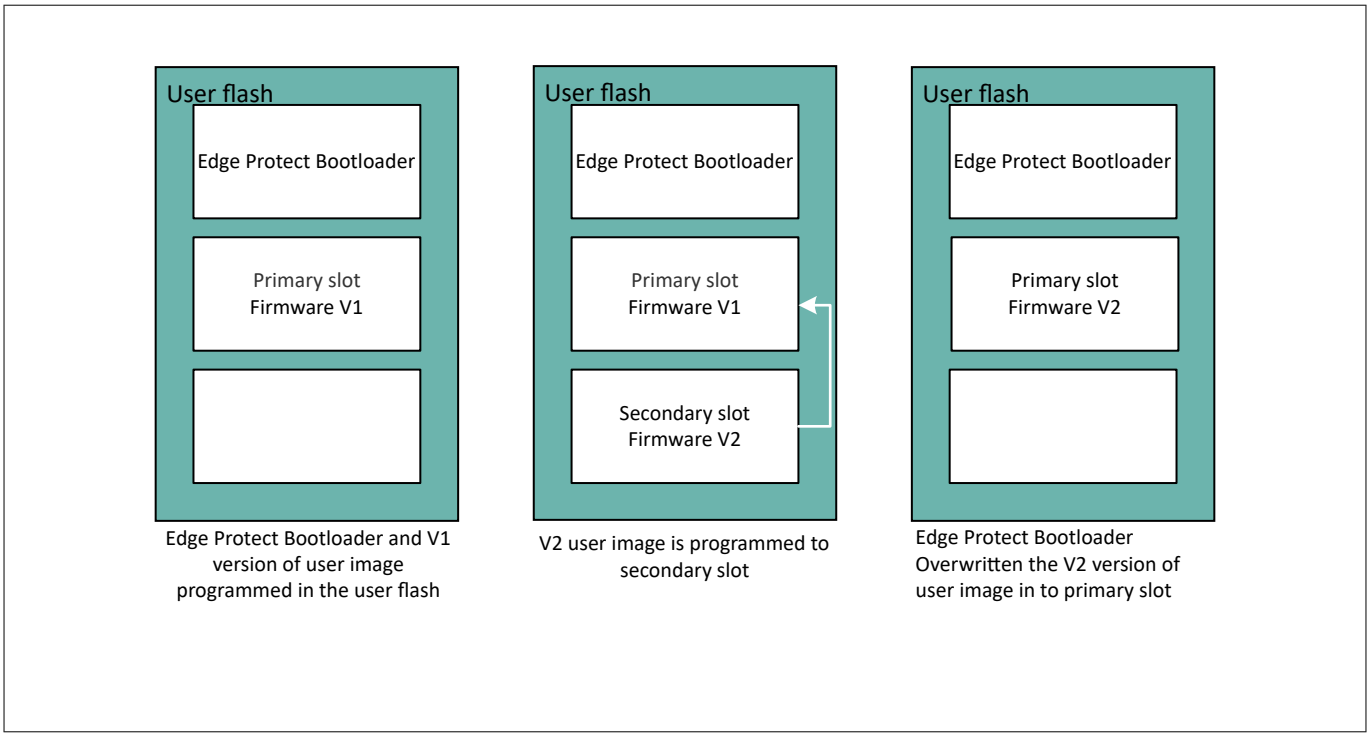



**Figure 25 Upgrade by overwrite**

### 10.3.4 Upgrade by swap

When a swap-based upgrade is enabled, the Edge Protect Bootloader exchanges the images between the primary and secondary slots, allowing for a seamless rollback to the previous version if the upgraded image proves to be faulty. This process involves simultaneously copying the content of the primary slot to the secondary slot and vice versa, effectively preserving both versions of the application on the device.

Once the upgraded firmware is swapped to the primary slot, the Edge Protect Bootloader verifies its operability. If the upgraded image fails to boot, the Edge Protect Bootloader initiates a revert procedure, rolling back to the previous firmware in the primary slot, ensuring a safe and reliable upgrade process.

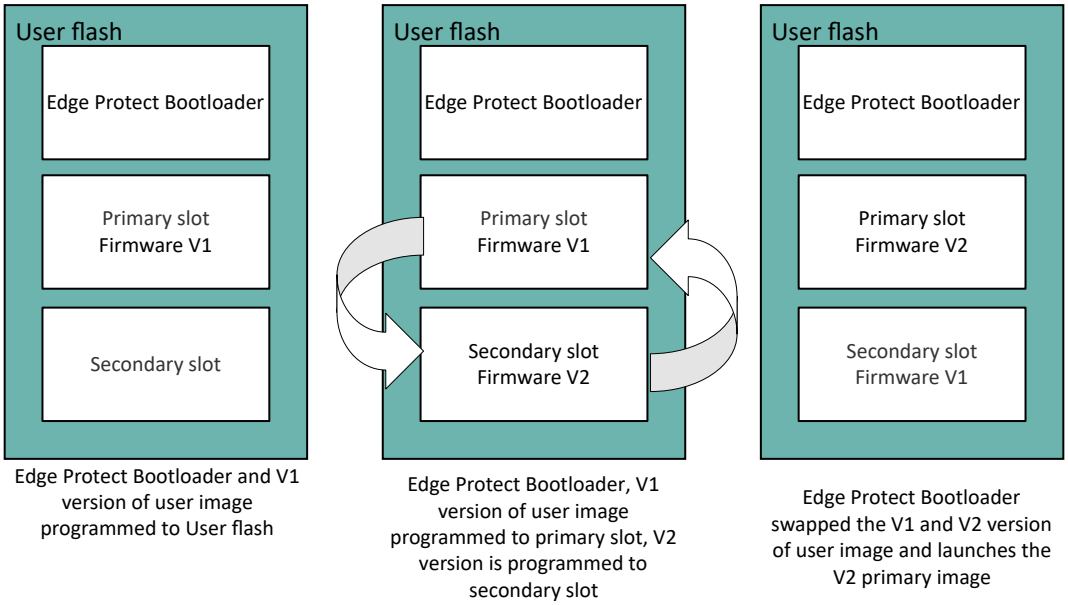



**Figure 26 Upgrade by swap**

### 10.3.5 Secure boot

Secure boot functionality ensures a trusted boot process through a combination of immutable boot code and digital signatures. The immutable BootROM code serves as the root of trust, launching the Edge Protect Bootloader only after verifying its digital signature. This verified Edge Protect Bootloader, in turn, checks the digital signature of the user application before allowing it to run. If any authentication fails, the device will not boot.

To maintain the integrity of the boot process, the signature verification takes place every time the device boots. This verification occurs in the primary slot. Additionally, the Edge Protect Bootloader verifies the signature of the image in the secondary slot before copying it to the primary slot, ensuring that only authenticated images are used.

Secure boot can be configured through `feature_config.json` file present in the `<CE workspace\platforms\PSC3>`

**Code Listing 12**

```
"validation_key": {
    "description": "Key for image validation",
    "value" : "Path to key, TBD"
},
"validate_boot": {
    "description": "Image validation during boot",
    "value": true
},
"validate_upgrade": {
    "description": "Image validation during upgrade",
    "value": true
},
```

During the Edge Protect Bootloader build, it is necessary to provide the private key that will be used to digitally sign the Edge Protect Bootloader. This private key is essential for ensuring the authenticity and integrity of the Edge Protect Bootloader.

```
make build_proj BOOT_MODE=signed OEM_KEY_FILE=./oem_rot_priv_key_0.pem
```

For a more in-depth look at the Edge Protect Bootloader implementation for the PSC3M/P8, PSC3M/P7 MCU devices, see the Edge Protect Bootloader code example.

***Note**: When secure boot is enabled, it is essential to configure the boot process accordingly. Specifically, the boot configuration should not be set to BOOT_SIMPLE_APP, as this would compromise the security features of the secure boot process.*

### 10.3.6 Post-quantum cryptography signature schemes

The Edge Protect Bootloader supports Post-Quantum Cryptography (PQC) signature schemes for image authentication, in addition to the classical ECDSA schemes. PQC signing ensures that the firmware authentication chain remains secure against adversaries equipped with quantum computing capabilities.

The following PQC schemes are supported:

**Table 13 PQC schemes**

| Scheme | Type | Parameter set | Standard |
|---|---|---|---|
| ML-DSA | Lattice-based (stateless) | ML-DSA-87 | FIPS 204 |
| LMS | Hash-based (stateful) | LMS_SHA256_M32_H10 with LMOTS_SHA256_N32_W8 | RFC 8554 |
| XMSS | Hash-based (stateful) | XMSS_SHA2_10_256 | RFC 8391 |

***Note**: LMS and XMSS are stateful signature schemes — each private key can only sign a finite number of messages. The signing state must be tracked and persisted carefully on the host. Reusing a private key state compromises the security of all signatures produced from that state. ML-DSA is stateless and does not have this constraint.*

Image authentication scheme can be configured through `feature_config.json` file located in the `<CEworkspace\platforms\PSC3_P8`.

**Code Listing 13**

```
"validation_key_type": {
"description": "Type of the image validation key. Possible values:
'ECDSA-256', 'ECDSA-384', 'ECDSA-521',
'ML-DSA-87', 'LMS_SHA256_M32_H10', 'XMSS_SHA2_10_256'",
"value": "ML-DSA-87"
},
"validation_key": {
"description": "Key for image validation",
"value": "../keys/ml_dsa_87_pub_key.der"
},
```

The OEM public validation key is embedded directly into the Edge Protect Bootloader binary and resides in the device's internal flash. An open Debug Access Port (DAP) on a production device allows an attacker to inspect flash and RAM contents, set breakpoints in the boot sequence, and potentially bypass or manipulate the signature verification flow through fault injection or memory patching. Therefore, closing all debug access ports before shipping a device to production is recommended. For the full debug access policy and how to enforce it through provisioning, refer to [2].

For details on signing application images, see Signing user application section and refer to [7].

### 10.3.7 Secure firmware update

The Edge Protect Bootloader is important in authenticating firmware during device startup and facilitating firmware switching during the update process. However, it does not have the capability to update the firmware independently. Instead, it relies on external middleware to download new firmware images. Infineon provides two options for this purpose: Device Firmware Update (DFU) middleware for wired updates (via UART, I2C, SPI, or CAN FD) and firmware upgrade via BootROM DFU serial interface (UART, I2C, and SPI).

These middleware solutions can be integrated into end application, enabling communication with the host and downloading of MCUboot-compatible hex files for new firmware images. When using a DFU middleware-based user application, the new image is downloaded to the secondary slot, and control is transferred to the Edge Protect Bootloader. The bootloader then verifies the image version, performs additional validation checks, and depending on the configuration, either swaps or overwrites the image in the secondary slot with the primary boot slot if the secondary slot is valid.

After transferring the image to the primary slot, the Edge Protect Bootloader verifies the primary image and boots to it if verified. Code examples demonstrating firmware upgrade support in end applications for the Edge Protect Bootloader over wired connections on PSC3M/P8, PSC3M/P7 MCU devices are available, including:

- PSOC™ Control MCU: Firmware upgrade through DFU MW
- PSOC™ Control MCU: Factory programming through BootROM DFU

For more information on incorporating DFU middleware into your application or Edge Protect Bootloader, see the AN236282 - Device Firmware Update (DFU) middleware (MW) for ModusToolbox™ application note.

### 10.3.8 Image encryption

To ensure the confidentiality of image data during transportation to a device, Edge Protect Bootloader provides a feature to encrypt and decrypt images in real-time during the upgrade process. To utilize this feature, the image header must be flagged as ENCRYPTED (indicated by the value 0x04) and include a Type-Length-Value (TLV) containing the encryption key.

When upgrading an image from the secondary slot to the primary slot, Edge Protect Bootloader automatically decrypts the image after validation. Additionally, if swap upgrades are enabled, Edge Protect Bootloader re-encrypts the image located in the primary slot, which must also have the ENCRYPTED flag set and the TLV present, when swapping it to the secondary slot.

Image encryption can be enabled through `feature_config.json` file present in the `<CE workspace\platforms\PSC3>`.

**Code Listing 14**

```
"image_encryption": {
    "description": "Activate/deactivate image encryption",
    "key" : "Path to key, TBD",
    "value": true
},
```

## 10 Edge Protect Bootloader

When encrypting an image, only the payload (FW) is encrypted. The header TLVs are still sent as plain data.

**Encrypting the image**

- Generate a new key pair EC-P256. User can skip this step if they want to use existing keys.

```
edgeprotecttools create-key --format PEM --key-type ECDSA-P256 --output enc_privkey.pem
enc_pubkey.pem
```

- Generate C code for public key.

```
edgeprotecttools convert-key --fmt C_ARRAY -k enc_privkey.pem -o enc_key.c
```

- Open `<user project>/bootloader/enc_key.c` file and `ecdsa_pub_key` to `enc_priv_key` Generate C code for public key. For example,

**Code Listing 15**

```
/* Autogenerated by imgtool.py, do not edit. */
const unsigned char enc_priv_key[] = {
    0x30, 0x81, 0x87, 0x02, 0x01, 0x00, 0x30, 0x13,
    0x06, 0x07, 0x2a, 0x86, 0x48, 0xce, 0x3d, 0x02,
    0x01, 0x06, 0x08, 0x2a, 0x86, 0x48, 0xce, 0x3d,
    0x03, 0x01, 0x07, 0x04, 0x6d, 0x30, 0x6b, 0x02,
    0x01, 0x01, 0x04, 0x20, 0xd5, 0xea, 0xc4, 0x0e,
    0x9d, 0xb7, 0x62, 0x7c, 0xb9, 0x40, 0x4f, 0x44,
    0x66, 0x7c, 0xa9, 0x4c, 0xae, 0x16, 0x58, 0xd5,
    0x2a, 0x43, 0x4d, 0x8b, 0x28, 0xed, 0x6d, 0xad,
    0xe5, 0xa9, 0xea, 0x88, 0xa1, 0x44, 0x03, 0x42,
    0x00, 0x04, 0x68, 0x18, 0x8f, 0xbf, 0x94, 0xea,
    0x58, 0xe5, 0xe0, 0x55, 0x2e, 0x7c, 0x0b, 0x6f,
    0x55, 0x36, 0x9e, 0x89, 0x59, 0x3f, 0xee, 0x6c,
    0x28, 0xb2, 0xe2, 0x82, 0x0d, 0x01, 0xeb, 0x18,
    0x46, 0x15, 0xc8, 0xed, 0x03, 0x35, 0xcb, 0xc5,
    0xc3, 0x74, 0x2a, 0xb1, 0xbc, 0x42, 0x7e, 0x65,
    0x36, 0xc9, 0x8d, 0xbc, 0x92, 0x33, 0x14, 0xc4,
    0xa1, 0xf7, 0xca, 0x26, 0xe2, 0xcd, 0xf8, 0xfb,
    0x15, 0x56
};
const unsigned int enc_priv_key_len = 138;
```

- Build the Edge Protect Bootloader project in the `ModusShell` with keys to sign the Edge Protect Bootloader

```
make build_proj BOOT_MODE=signed OEM_KEY_FILE=./oem_rot_priv_key_0.pem
```

- Build and sign the user application with `--encrypt` command using Edge Protect Tools as follows:

**Code Listing 16**

```
ADDITIONAL_ARGS+=--public-key-format full               \
                 --pubkey-encoding raw           \
                 --signature-encoding raw        \
                 --overwrite-only                \
                                 -- encrypt

POSTBUILD+=$(EPT_PATH) sign-image                \
             --image $(INPUT_IMAGE)             \
             --output $(OUTPUT_IMAGE)         \
             --header-size $(HDR_OFFSET)        \
             --slot-size $(SLOT_SIZE)        \
             --key $(KEY_PATH)            \
             --hex-addr $(APP_START_ADDR)         \
             --align $(FLASH_ALIGNMENT)         \
             --image-version $(IMAGE_VERSION)    \
             --erased-val $(ERASED_VAL)         \
             --min-erase-size $(MIN_ERASE_SIZE)     \
             -s $(RB_COUNTER)            \
             $(ADDITIONAL_ARGS)            \
             --boot-record secureBoot
```

- After programming both Edge Protect Bootloader and user application, Edge Protect Bootloader uses its internal `enc_priv_key`to decrypt the TLV containing the key. Given that no errors are found, it will then start the validation process, decrypting the blocks before check. A good image being determined, the upgrade consists in reading the blocks from the `secondary slot`, decrypting and writing to the `primary slot` and Edge Protect Bootloader launches the user application

### 10.3.9 Downgrade prevention

Downgrade prevention is a security mechanism that ensures a device's firmware can only be updated with a newer version that has a higher security counter number than the existing one. This safeguard prevents malicious attempts to rollback the device to an earlier version, potentially vulnerable firmware version, thereby maintaining the integrity of the device's security.

The software-based downgrade prevention mechanism involves a comparison of image version numbers to prevent the installation of older firmware versions. This feature can be enabled by setting the `sw_downgrade_prev` parameter present in the `<CE workspace\platforms\PSC3feature_config.json>` file. However, it is essential to note that this feature is only functional when the overwrite-based image update strategy is employed, which requires the default `MEMORY_MAP` should be predefined for each platform inside the`common.mk` file as follows:

```
MEMORY_MAP ?= ../platforms/$(PLATFORM)/memory_maps/overwrite_single_flash.json
```

Download prevention can be configured through `feature_config.json` file located in the `<CE workspace\platforms\PSC3>`.

**Code Listing 17**

```
"sw_downgrade_prev": {
    "description": "Use software downgrade prevention (for 'overwrite' method only!)",
    "value": true
}
```

### 10.3.10 Dependency check

The Edge Protect Bootloader has the capability to manage multiple firmware images. While it is possible to update these images independently, additional Type-Length-Value (TLV) entries can be added to the protected TLV area following each image.

Dependency check can be configured through `feature_config.json` file located in the `<CE workspace\platforms\PSC3>`.

**Code Listing 18**

```
    "dependency_check": {
        "description": "To prevent upgrading the non-compatible images for multiple-images",
        "value": true
    },
```

During the dependency check phase, the Edge Protect Bootloader verifies whether all image dependencies are satisfied. If any image dependency is unfulfilled, modify the swap type of that image accordingly, and restart the dependency check. This process continues until the number of unsatisfied dependencies decreases or remains the same. There is always at least one valid configuration. In the worst-case scenario, the system reverts to its initial state after the dependency check.

A dependency can be specified as an additional parameter while generating the MCUboot format image using the Edge Protect Tool.

For example, `-d "(image_id,image_version)"`, where the `image_id` is the number of the image which the current image depends on and the `image_version` is the minimum version of that image to satisfy compliance. For example, `-d "(1, 1.2.3+0)"` means that this image depends on Image 1, which version has to be at least 1.2.3+0.

### 10.3.11 Fault injection hardening

To counteract hardware-based fault injection attacks, such as power glitching and EM pulses, which aim to execute unauthorized code, the Edge Protect Bootloader incorporates software-based countermeasures. These attacks use induced changes in CPU behavior, including skipped instructions, modified registers, corrupted memory reads, and altered instruction decoding to bypass security mechanisms.

While these software countermeasures enhance the resistance to fault injection attacks, they do not provide complete protection. The level of protection can be controlled using the `MCUBOOT_FIH_PROFILE_*` macros, which offer four levels of protection: none, moderate, medium, and maximum. These countermeasures are primarily integrated into the core MCUboot codebase, focusing on critical code paths that are susceptible to fault injection. Additionally, independent cryptographic libraries, such as mbed-crypto or tinycrypt can offer further protections, but this requires separate assessment.

The key hardening code is located in the following files:

- `boot/bootutil/include/bootutil/fault_injection_hardening.h`
- `boot/bootutil/src/fault_injection_hardening.c`

The fault injection mitigation library provides support for various measures, which can be enabled or disabled individually or by defining one of the MCUBOOT_FIH_PROFILEs. For more information on the measures supported by each profile, see the `fault_injection_hardening.h` file and `boot\cypress\MCUBootApp\MCUBoot.mk` file.

### 10.3.12 Serial logging

During the boot and upgrade process, the Edge Protect Bootloader is designed to log its status and reports, transmitting this information over a serial port. The level of detail in these logs is determined by the compiled log level, which sets the maximum amount of information that can be recorded. By default, these logs are printed over a UART port. Notably, this feature can be enabled or disabled during the compilation process, allowing for customization of the logging behavior.

There are five log-levels with a unique value defined in the Edge Protect Bootloader in the `\boot\cypress\MCUBootApp\config\mcuboot_config\mcuboot_logging.h` file.

```
#define MCUBOOT_LOG_LEVEL_OFF      0
#define MCUBOOT_LOG_LEVEL_ERROR    1
#define MCUBOOT_LOG_LEVEL_WARNING  2
#define MCUBOOT_LOG_LEVEL_INFO     3
#define MCUBOOT_LOG_LEVEL_DEBUG    4
```

You can define the global logging level by defining any of the above-mentioned log-levels in the `application>/bootloader_app/Makefile`.

MCUBOOT_LOG_LEVEL sets the minimum level that will be logged. The function priority is:

```
MCUBOOT_LOG_ERR > MCUBOOT_LOG_WRN > MCUBOOT_LOG_INF > MCUBOOT_LOG_DBG
```

# 11 Policy file

The policy file controls aspects of the boot procedure that requires to be assigned during provisioning before running the first OEM code. Table 14 describes the common parameters that can be set by the OEM.

**Table 14 Policy file**

| Parameter | Application values | Default value | Description |
|---|---|---|---|
| **Device policy** | | | |
| target_lcs | NORMAL_PROVISIONED, SECURE | NORMAL_PROVISIONED | The device will be shipped to an OEM in this LCS |
| prov_oem_complete | false, true | FALSE | Indicates that all OEM assets are finalized in SFlash and no further update is allowed. After this, the flag is set and provisioned. OEM can do only re-provisioning and RMA if needed |
| **debug** | | | |
| ap_cm33 | Enable, Disable, Permanently Disable | Enable | Configures the CM33 debug access port availability in the SECURE and NORMAL_PROVISIONED LCS |
| listen_window | 100 ms, 20 ms, 1 ms, 0 ms | 100 ms | Configures the listen window time to acquire the CM33 debug access port in the SECURE LCS |
| cm33_dbg | Enable, Disable, Permanently Disable | Enable | Invasive Debug. It implies that non-invasive is also enabled |
| cm33_nid | Enable, Disable, Permanently Disable | Enable | Non-Invasive Debug (monitoring without control) |
| cm33_spid | Enable, Disable, Permanently Disable | Enable | Secure Privileged Invasive Debug. The control field is not applicable in the development LCS |
| cm33_spnid | Enable, Disable, Permanently Disable | Enable | Secure Privileged Non-Invasive Debug. The control field is not applicable in the development LCS |
| cm33_Secure | Enable, Disable, Permanently Disable | Enable | Secure Privileged Debug. The control field is not applicable in the development LCS |
| **system** | | | |
| ap | Enable, Disable, Permanently Disable | Enable | System access port (SYS-AP) restriction |
| secure | Enable, Disable, Permanently Disable | Enable | Enables the CM33 secure application interface |
| **boot** | | | |
| boot_cfg_id | SIMPLE_APP, SECURE_APP, DUAL_BANK_SIMPLE_APP, DUAL_BANK_SECUR | SIMPLE_APP | Boot configuration |
| boot_dual_bank_enable | false, true | false | Flag to enable flash dual bank feature |

**(table continues...)**

**Table 14 (continued) Policy file**

| Parameter | Application values | Default value | Description |
|---|---|---|---|
| boot_bank_ctr_addr | 0xC39D5455 | – | An address of a 32-bit counter for flash dual bank switching |
| bbot_auth_images | NONE, ONE, ALL, ALL_MOVE | ALL_MOVE | What to be authenticated, or moved from a Shared Secondary Slot after a DFU update |
| boot_app_layout | – | – | The memory layout for the applications defined by the BOOT_CFG_ID |
| **dfu** | | | |
| dfu_enable | false, true | false | Flag to enable the BootROM DFU serial interface |
| boot_od_clk | 48 MHz, 100 MHz, 180 MHz | 180 MHz | CM33 boot frequency options |
| pre_build | – | – | – |
| **keys** | | | |
| oem_rot_public_key_0 | – | – | Key path for the first application OEM rot key 0 |
| oem_rot_public_key_1 | – | – | Key path for the first application OEM rot key 1 |
| oem_debug_public_key | - | - | Key path to the OEM debug public key |
| oem_dev_public_key | - | - | Key path to the OEM rot development public key |

# A Default OEM provisioning policy

The default OEM policy parameters are preset on a device before it is shipped to the OEM.

**Code Listing 19**

```
{
  "policy": {
    "platform": "psoc_c3p8",
    "version": 1.0,
    "type": "oem_provisioning"
  },
  "device_policy": {
    "target_lcs": {
      "description": "The LCS to which the transition should be done",
      "applicable_conf": "NORMAL_PROVISIONED, SECURE",
      "value": "NORMAL_PROVISIONED"
    },
    "program_sflash_assets": {
      "description": "Indicates whether to program the assets which are placed into SFLASH",
      "value": true
    },
    "program_iak": {
      "description": "Indicates whether to generate and program Initial Attestation Key",
      "value": false
    },
    "program_hash": {
      "description": "Enables ASSET_HASH and FACTORY_HASH validation in NORMAL_PROVISIONED LCS",
      "value": false
    },
    "nv_counter_a1": {
      "description": "NV counter slot 1. 0, if no update",
      "value": 0
    },
    "nv_counter_a2": {
      "description": "NV counter slot 2. 0, if no update",
      "value": 0
    },
    "nv_counter_a3": {
      "description": "NV counter slot 3. 0, if no update",
      "value": 0
    },
    "nv_counter_a4": {
      "description": "NV counter slot 4. 0, if no update",
      "value": 0
    },
    "failed_signature_counter": {
      "comment_1": "If a counter value is 0, a signature check is not performed, but a
signature is treated as failed.",
      "comment_2": "If a counter value is 1 to 65534, a signature is checked, and if a
signature check fails then decrease a counter",
      "comment_3": "If a counter value is 65535, the counter is ignored, but a signature is
checked each time",
      "ram_app": {
```

## A Default OEM provisioning policy


```
        "description": "Failed signature counter for RAM application",
        "value": 65535
      },
      "boot_app_layout_0": {
        "description": "Failed signature counter for application in BOOT_APP_LAYOUT[0] slot",
        "value": 65535
      },
      "boot_app_layout_1": {
        "description": "Failed signature counter for application in BOOT_APP_LAYOUT[1] slot",
        "value": 65535
      },
      "boot_app_layout_2": {
        "description": "Failed signature counter for application in BOOT_APP_LAYOUT[2] slot",
        "value": 65535
      },
      "boot_app_layout_3": {
        "description": "Failed signature counter for application in BOOT_APP_LAYOUT[3] slot",
        "value": 65535
      }
    },
    "iak_cfg": {
      "description": "IAK generation config",
      "applicable_conf": "NONE, AES256, ECC256, ECC384, ECC521",
      "value": "NONE"
    },
    "prov_oem_complete": {
      "description": "Indicates OEM assets are finalized in SFLASH",
      "warning": "Once the 'prov_oem_complete' parameter value is set to true, it cannot be
changed! No further updates to SFLASH are allowed",
      "value": false
    },
    "debug": {
      "cpu": {
        "ap_cm33": {
          "description": "Configures the CM33 debug access port availability in the SECURE and
NORMAL_PROVISIONED LCS",
          "applicable_conf": "Enable, Disable, Permanently Disable",
          "value": "Enable"
        },
        "listen_window": {
          "description": "Configures the listen window time to acquire the CM33 debug access
port in the SECURE LCS",
          "applicable_conf": "100 ms, 20 ms, 1 ms, 0 ms",
          "value": "100 ms"
        },
        "cm33_dbg": {
          "description": "Configures the invasive debug of CM33 core",
          "applicable_conf": "Enable, Disable, Permanently Disable",
          "value": "Enable"
        },
        "cm33_nid": {
          "description": "Configures the non-invasive debug of CM33 core",
          "applicable_conf": "Enable, Disable, Permanently Disable",
```

## A Default OEM provisioning policy


```
            "value": "Enable"
          },
          "cm33_spid": {
            "description": "Configures the secure privileged invasive debug of CM33 core",
            "applicable_conf": "Enable, Disable, Permanently Disable",
            "value": "Enable"
          },
          "cm33_spnid": {
            "description": "Configures the secure privileged non-invasive debug of CM33 core",
            "applicable_conf": "Enable, Disable, Permanently Disable",
            "value": "Enable"
          },
          "cm33_secure": {
            "description": "Enables the CM33 secure AP interface",
            "applicable_conf": "Enable, Disable, Permanently Disable",
            "value": "Enable"
          }
        },
        "system": {
          "ap": {
            "description": "Configures the System debug access port availability in the SECURE
and NORMAL_PROVISIONED LCS",
            "applicable_conf": "Enable, Disable, Permanently Disable",
            "value": "Enable"
          },
          "secure": {
            "description": "Enables the system secure AP interface",
            "applicable_conf": "Enable, Disable, Permanently Disable",
            "value": "Enable"
          },
          "mmio_restrictions": {
            "description": "MMIO restrictions for System AP",
            "applicable_conf": "No restrictions, SRSS_GENERAL2 only, No access",
            "value": "No restrictions"
          },
          "debug_pins": {
            "description": "Configures debug pins and interface",
            "comment_1": "0000 - 2 pins (TMS, TCLK), JTAG is disabled",
            "comment_2": "0011 - 2 pins (TMS, TCLK), JTAG is enabled",
            "comment_3": "0101 - 4 pins (TMS, TCLK, TDO, TDI)",
            "comment_4": "0110 - 5 pins (TMS, TCLK, TDO, TDI, TRSTN)",
            "comment_5": "1001 - SWJ pins disabled, JTAG is enabled",
            "comment_6": "1010 - SWJ pins disabled, JTAG is disabled",
            "applicable_conf": "0000, 0011, 0101, 0110, 1001, 1010",
            "value": "0000"
          }
        }
      },
      "boot": {
        "boot_cfg_id": {
          "description": "A behavior for BOOT_APP_LAYOUT (BOOT_SIMPLE_APP applicable to
NORMAL_PROVISIONED only)",
          "applicable_conf": "SIMPLE_APP, SECURE_APP, DUAL_BANK_SIMPLE_APP, DUAL_BANK_SECURE_APP,
```

## A Default OEM provisioning policy


```
PROT_FW",
        "value": "SECURE_APP"
      },
      "boot_dual_bank_enable": {
        "description": "Indicates whether to enable flash dual bank feature",
        "value": false
      },
      "boot_bank_ctr_offset": {
        "description": "An offset from the start of each flash bank where a 32-bit counter for
flash dual bank switching is placed",
        "value": "0xC39D5455"
      },
      "boot_hash_locking": {
        "description": "Enable hash locking, fast authentication (compare a hash) instead of
ECDSA",
        "value": true
      },
      "boot_auth_images": {
        "description": "Configures the images authentication and moving from a SSS after DFU
update",
        "applicable_conf": "NONE, ONE, ALL, ALL_MOVE",
        "comment_1": "NONE - Disable all authentications",
        "comment_2": "ONE - Authenticate the first image, ignore the other images",
        "comment_3": "ALL - Authenticate all images",
        "comment_4": "ALL_MOVE - Authenticate all images, move from SSS on a successful DFU
update",
        "value": "ALL_MOVE"
      },
      "boot_auth_signature": {
        "description": "Signature type for Boot Authentication",
        "applicable_conf": "ES256, ES384, ES512",
        "value": "ES512"
      },
      "boot_app_layout": {
        "description": "The memory layout for the applications defined by BOOT_CFG_ID.
0x32000000 - 0x33FFFFFF for secure addresses; 0x22000000 - 0x23FFFFFF for non-secure addresses",
        "value": [
          {
            "address": "0x32000000",
            "size": "0x40000"
          },
          {
            "address": "0x22010000",
            "size": "0x10000"
          },
          {
            "address": "0x22030000",
            "size": "0x8000"
          },
          {
            "address": "0x22038000",
            "size": "0x8000"
          },
```

## A Default OEM provisioning policy


```
            {
              "address": "0x00000000",
              "size": "0x00"
            }
          ]
        },
        "boot_od_clk": {
          "description": "CM33 boot frequency",
          "applicable_conf": "48 MHz, 100 MHz, 180 MHz",
          "value": "180 MHz"
        }
      },
      "dfu": {
        "dfu_enable": {
          "description": "Indicates whether Device Firmware Update is enabled",
          "value": false
        }
      },
      "reprovisioning": {
        "revoke_ifx_rot_key_0": {
          "description": "Indicates whether to revoke IFX_ROT_KEY_0",
          "value": false
        },
        "revoke_ifx_rma_master_key_0": {
          "description": "Indicates whether to revoke IFX_RMA_MASTER_KEY_0",
          "value": false
        },
        "revoke_oem_rot_key_0": {
          "description": "Indicates whether to revoke OEM_ROT_KEY_0",
          "value": false
        },
        "nonce": {
          "description": "Hex string or path to a binary file containing nonce for revocation of
IFX_ROT_KEY_0 or IFX_RMA_MASTER_KEY_0",
          "value": ""
        },
        "ifx_revocation_signature": {
          "description": "Hex string or binary file containing IFX_REVOCATION_SIGNATURE for
revocation of IFX_ROT_KEY_0 or IFX_RMA_MASTER_KEY_0",
          "value": ""
        }
      },
      "pc1_ctl_valid": {
        "description": "True when 'pc1_handler' is valid",
        "value": false
      },
      "pc2_ctl_valid": {
        "description": "True when 'pc2_handler' is valid",
        "value": false
      },
      "pc3_ctl_valid": {
        "description": "True when 'pc3_handler' is valid",
        "value": false
```

## A Default OEM provisioning policy


```
    },
    "pc2_handler": {
      "description": "Used to switch to PC2 when it is a 'Special' PC",
      "value": "0xFFFFFFFF"
    },
    "pc3_handler": {
      "description": "Used to switch to PC3 when it is a 'Special' PC",
      "value": "0xFFFFFFFF"
    },
    "oem_device_cert_addr": {
      "description": "OEM_DEVICE_CERT address",
      "value": "0x00000000"
    },
    "oem_device_cert_blocks": {
      "description": "OEM_DEVICE_CERT size in blocks",
      "value": "0x00"
    },
    "boundary_scan": {
      "description": "Indicates whether Boundary Scan is enabled",
      "value": true
    },
    "mpc": {
      "mpc_struct_flash": {
        "description": "Flash MPC structures. The total number of RAM and Flash MPC structures
must be less than 32",
        "value": [
          {
            "description": "Offset and size in KB. The values must be multiple of 2 (e.g. 0 KB,
2 KB, 4 KB, 8 KB, ...)",
            "offset": "2 KB",
            "size": "2 KB",
            "attr": {
              "description": "The MPC attributes for this memory region",
              "comment_1": "WRN allow read, write, and NS only access",
              "comment_2": "-RN allow read, NS-only access",
              "comment_3": "--- and --N, no access",
              "comment_4": "WR- read and write access, but NS=0 only",
              "value": {
                "pc0": "WR-",
                "pc1": "WR-",
                "pc2": "WR-",
                "pc3": "WR-",
                "pc4": "WR-",
                "pc5": "WR-",
                "pc6": "WR-",
                "pc7": "WR-"
              }
            }
          }
        ]
      }
    }
  },
```

## A Default OEM provisioning policy


```
    "pre_build": {
      "keys": {
        "oem_rot_public_key_0": {
          "description": "Path to the OEM RoT public key 0",
          "value": "../keys/oem_rot_pub_key_0.pem"
        },
        "oem_rot_public_key_1": {
          "description": "Path to the OEM RoT public key 1",
          "value": "../keys/oem_rot_pub_key_1.pem"
        },
        "oem_debug_public_key": {
          "description": "Path to the OEM debug public key",
          "value": "../keys/oem_debug_pub_key.pem"
        },
        "oem_dev_public_key": {
          "description": "Path to the OEM RoT dev public key",
          "value": "../keys/oem_dev_pub_key.pem"
        }
      }
    },
    "raw_data_pc012": {
      "description": "Path to a binary file containing custom data accessible in PC0, PC1, and
PC2. Up to 84 bytes",
      "value": ""
    }
  }
```

# References

### Application notes

**[1]** Infineon Technologies AG: *AN241270 – Getting started with PSOC™ Control C3M/P8, C3M/P7 MCUs on ModusToolbox™ software*; Available online

**[2]** Infineon Technologies AG: *AN242919 – PSOC™ C3M/P8, C3M/P7 MCUs provisioning specification*

### User guide

**[3]** Infineon Technologies AG: *002-39923: ModusToolbox™ Edge Protect Configurator user guide*; Available online

### Datasheet

**[4]** Infineon Technologies AG: *002-39793: PSOC™ Control C3 - PSC3P7xx, PSC3M7xx, PSC3P8xx, PSC3M8xx*; Available online

### Reference manuals

**[5]** Infineon Technologies AG: *002-42141: PSOC™ Control C3 (PSC3P7xx, PSC3M7xx, PSC3P8xx, PSC3M8xx) architecture reference manual*; Available online

**[6]** Infineon Technologies AG: *002-41257: PSOC™ Control C3M/P8, C3M/P7 registers reference manual*

### Code examples

**[7]** Infineon Technologies AG: *PSOC™ C3M8 MCU: Secure boot and update with EPB*; Available online

**[8]** Infineon Technologies AG: *PSOC™ C3M8 MCU: Secure-non-secure template*; Available online

***Note**: For more documents related to PSC3M/P8, PSC3M/P7 devices, see PSOC™ Control C3 Performance Line webpage.*

# Glossary

**Concise binary object representation (CBOR)**

This format is used to store the owner's policy in the device's NVM

**Debug access port (DAP)**

There may be multiple DAPs in this device accessible to the customer; system, CM33

**Development LCS**

This is the stage that the device is in during development. In this stage, no permanent changes are made to the device, and it can be reverted back to its original state that was shipped from Infineon. The device can be provisioned as many times as needed while in this stage

**Device firmware upgrade (DFU)**

DFU is the act of transferring data from the host to the target flash either using a BootROM DFU or customer DFU interface and booting to newly downloaded firmware

**Hash**

A crypto algorithm that generates a repeatable but unique signature for a given block of data. This function is not reversible

**Infineon's customers OEM_Policy**

This file contains the boot-up parameters used

**IPC**

Inter-processor communication. Hardware used to facilitate communication between secure and non-secure environment

**Lifecycle stage (LCS)**

The device is shipped from Infineon in the development LCS and then will be advanced to the production LCS when the customer ships the product

**MCUboot**

An open-source secure bootloader library for 32-bit MCUs aiming to define a common infrastructure for the bootloader and system flash layout on microcontroller systems and to provide a secure bootloader that enables easy software upgrade

**Memory protection unit (MPU)**

Hardware blocks used to allow or deny access to sections of memory or peripherals to bus masters

**Non-volatile memory (NVM)**

A type of memory that can retain stored information even after power is removed

**OEM**

Original Equipment Manufacturer

**OEM_ROT_KEY**

This refers to the OEM_ROT_PUBLIC_KEY and OEM_ROT_PRIVATE_KEY pairs

**OEM_ROT_PRIVATE_KEY**

This key is used to sign the first OEM code in the system when secured boot is enabled. This key can also be used to sign other projects or firmware updates that are installed on the device

## Glossary

**OEM_ROT_PUBLIC_KEY**

This is the key used to validate the first OEM code when secured boot is enabled

**Primary code space**

This is the location in memory where the code to be executed is stored in non-volatile memory

**Production LCS**

Once a device is transitioned to this stage, it cannot revert back to the development LCS

**Provision**

This is the process by which the OEM loads their custom software, keys, and boot parameters into the device. A textual, JSON-formatted file (OEM_POLICY) is used to define a set of security-related parameters

**RAM APP**

Executable code that is loaded into SRAM. RAM APP can be written and loaded by the OEM and is executed by the CM33

**Return merchandise authorization (RMA)**

This LCS is used if a device is suspected to be defective. The OEM places the device in this stage prior to shipment back to Infineon

**Root of trust (RoT)**

In the PSOC™ Control C3 devices, the BootROM code is the root of trust

**Secondary code space**

This is the area where code updates are stored until they have been copied into the primary

**Secured boot**

In the context of this document, this refers to when the first OEM code must be signed and then authenticated by the boot code

**Serial Wire Debug (SWD)**

A method used to interface debug and programming tools to the device

**TLV**

Tag-length-value is a way of storing data in a project header that is easily parsed by code

# Revision history

| Document revision | Date | Description of changes |
|---|---|---|
| ** | 2026-03-27 | Initial release |
| *A | 2026-04-29 | Updated the following sections:<br>• Overview of PSC3M/P8, PSC3M/P7 MCUs<br>• Signing user application<br>Added the following sections:<br>• Hardware security features<br>• Lifecycle security and controlled debug access<br>• Secure boot, code authentication, and firmware update<br>• Development tooling<br>• Post-quantum cryptography signature schemes |
| *B | 2026-09-18 | Release to web |

# Trademarks

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

**Trademarks**

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

**Edition 2026-09-18**

**Published by**

**Infineon Technologies AG**

**81726 Munich, Germany**

**© 2026 Infineon Technologies AG**

**All Rights Reserved.**

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

**Email: erratum@infineon.com**

**Document reference**
**IFX-bog1773066940746**

**Important notice**

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.