Secure Boot - Bare Metal Practical Guide - RA8M1

Version 1.0

Table of Contents

1 Introduction

1.1 Purpose and Scope

This document provides guidelines for Original Equipment Manufacturers (OEMs) intending to deploy microcontrollers from the Renesas RA8 family in production systems with Secure Boot enabled. The primary objective is to describe a secure and repeatable provisioning process suitable for a manufacturing environment, with a particular focus on protecting cryptographic keys using a Hardware Security Module (HSM).

The document presents a reference provisioning approach based on a custom-developed tool that integrates secure key storage, firmware authentication, and device configuration into a streamlined production workflow. The intent is to illustrate how secure boot can be enabled and managed in an automated and repeatable manner, avoiding manual, file-based key handling and the operational overhead of switching between multiple tools, both of which are generally unsuitable for scalable, auditable production environments.

This document does not claim to define the only valid provisioning strategy for RA8 devices. Instead, it describes one concrete and practical solution that balances security, automation, and flexibility. OEMs are expected to adapt the described approach to their own manufacturing constraints, security policies, and threat models.

1.2 Prerequisite and Intended Audience

This document is intended for engineers and technical architects involved in secure firmware development, device provisioning, and manufacturing workflows for embedded systems. It is written for readers who are expected to interact directly with both firmware and production tooling and to make informed decisions about security configuration in a manufacturing context.

The document is designed to be self-contained. It provides the background information necessary to understand the concepts, mechanisms, and workflows described, without requiring the reader to refer to external documentation. Where relevant, introductory sections are included to explain foundational concepts that the reader must be familiar with in order to follow the rest of the material.

Although this document is written as a self-contained guide and includes the background required to understand the concepts and workflows it describes, readers will benefit most if they have prior experience in several relevant technical area. The material is intended for engineers who are comfortable working across firmware, tooling, and manufacturing environments, and who can make informed decisions about security-related trade-offs. Specifically:

- Experience with embedded software development, including bootloaders and firmware integration
- Practical knowledge of Python, sufficient to understand and modify tooling scripts
- Familiarity with the Renesas RA ecosystem, including e² studio and standard development workflows
- An understanding of manufacturing and device provisioning processes for embedded systems
- A working knowledge of basic cryptography and security concepts, such as public/private keys, hashing, signatures, and chains of trust

While not required to follow this guide, the following Renesas documentation provides additional detail and context on many of the mechanisms discussed, and may be useful for readers seeking deeper background or authoritative reference material:

- **FSBL**: *Application Design using RA8 FSBL* (R11AN0774EU0110)
- **Boot Firmware**: *RA8M1 Boot Firmware* (R01AN7140EU0130)
- **DLM**: *Device Lifecycle Management* (R11AN0785EU0100)
- **Secure Key Injection and Update**: R11AN0496
- **Production Tools**: R01AN7884
- **SKMT User's Manual**

2 Theoretical Background

This chapter introduces the foundational concepts required to understand the secure provisioning approach described later in the document. It is intentionally high-level and avoids implementation-specific details, which are covered in subsequent chapters.

2.1 Secure Boot

Secure boot is a mechanism that ensures only authenticated and authorised firmware is executed on a device. Its main purpose is to establish a hardware-rooted chain of trust, starting from immutable boot code and extending through bootloaders to application firmware.

Key characteristics of secure boot include:

- **Authentication:** Every stage of the boot process verifies the integrity and authenticity of the next stage before execution.
- **Immutability of the root of trust:** The first code executed is stored in read-only memory (ROM) and cannot be modified in the field.
- **Protection against common threats:** Prevents unauthorised firmware replacement, persistent malware, and downgrade attacks.

In production environments, secure boot is a mandatory security foundation that enables higher-level protections, including secure firmware updates, device attestation, and controlled feature enablement.

2.2 Chain of Trust

Secure boot relies on a chain of trust, where each stage verifies the integrity and authenticity of the next stage before transferring execution. The chain typically begins with immutable boot ROM code and extends through the bootloader to the final application firmware.

If any link in the chain is compromised or improperly provisioned, the entire security model collapses. For this reason, key management and provisioning are as critical as the cryptographic algorithms themselves.

2.3 Public Key Cryptography in Secure Boot

Secure boot typically relies on public key cryptography to separate signing and verification responsibilities:

- **Private keys** are used to sign firmware and must be kept secret
- **Public keys** are embedded or injected into the device for verification

Elliptic Curve Cryptography (ECC), specifically the NIST P-256 curve, is widely adopted in embedded secure boot implementations due to its strong security properties and relatively low computational overhead.

In a production environment, the protection of private signing keys is paramount. Exposure of a private key typically results in a complete security failure, requiring device recalls, key revocation mechanisms, or permanent loss of trust.

2.4 Hardware Security Modules (HSMs)

A Hardware Security Module (HSM) is a dedicated device designed to securely generate, store, and use cryptographic keys. Private keys inside an HSM are not directly accessible in plaintext form, significantly reducing the attack surface compared to file-based key storage.

In production provisioning scenarios, HSMs are commonly used to:

- Generate root and intermediate keys

- Perform firmware signing operations
- Enforce access control and auditability
- Prevent manual key handling that introduces human error

HSMs often expose standardised interfaces, such as PKCS#11, allowing tools to perform cryptographic operations without direct access to private key material.

2.5 Secure Provisioning in Production

Provisioning is the process of moving a device from a blank or unconfigured state into a production-ready, secure state. Typical steps include:

- Injecting public keys or wrapped secrets
- Enabling secure boot and setting lifecycle states
- Programming bootloaders and application firmware
- Performing cryptographic signing and verification checks

A secure provisioning workflow should be repeatable, auditable, and scalable. Manual handling of keys or reliance on multiple disparate tools increases the risk of mistakes, reduces throughput, and complicates security audits. Integrating an HSM and automating key handling helps achieve a process suitable for high-volume manufacturing while maintaining the integrity of the chain of trust.

Figure 1 illustrates the common security challenges in current production provisioning environments where a secure area is required for key management and encryption operations. In many deployments, this secure area is physically or logically separated from the production facility, creating potential vulnerabilities during the transfer of sensitive data—such as configuration information, images, and cryptographic keys—between sites.

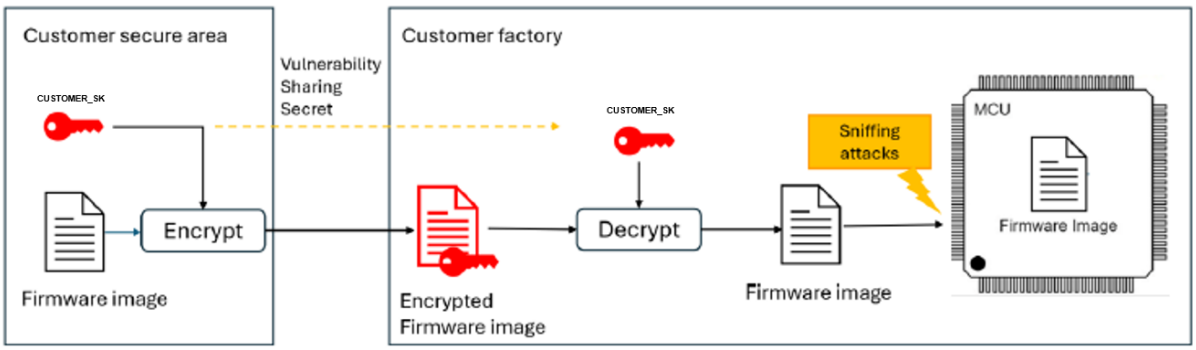


Figure 1 - Typical production provisioning setup

The next section describes a process that eliminates vulnerabilities arising from the need to share the customer's secret key.

3 Secure Boot on RA8M1

This chapter outlines the provisioning process for enabling Secure Boot on the RA8 Series.

It can be divided into three phases: Setup, Signing (Credential Preparation), and Programming.

3.1 Phase 1: Setup

This phase involves establishing the cryptographic infrastructure, obtaining the necessary wrapping materials from Renesas and generating the required keys.

The process of key injection into an RA device requires the user to generate their own secret intermediary key (256-bit) called a User Factory Programming Key (UFPK). Ideally, the generation process should be carried out in a Hardware Security Module to protect the key, but it could also be generated in a secure area. In order to protect both the Renesas MCU Hardware Root Key and the user's application keys, no sensitive key material should be exposed in transfer.

To accomplish this, Renesas have created a key wrapping service (**Error! Reference source not found.**) which is part of the Device Lifecycle Management (DLM) function. The UFPK is PGP-encrypted and sent to the Renesas Key Wrap Service, an automated secure server that wraps the UFPK (creating a W-UFPK), PGP-encrypts it, and returns it to the user.

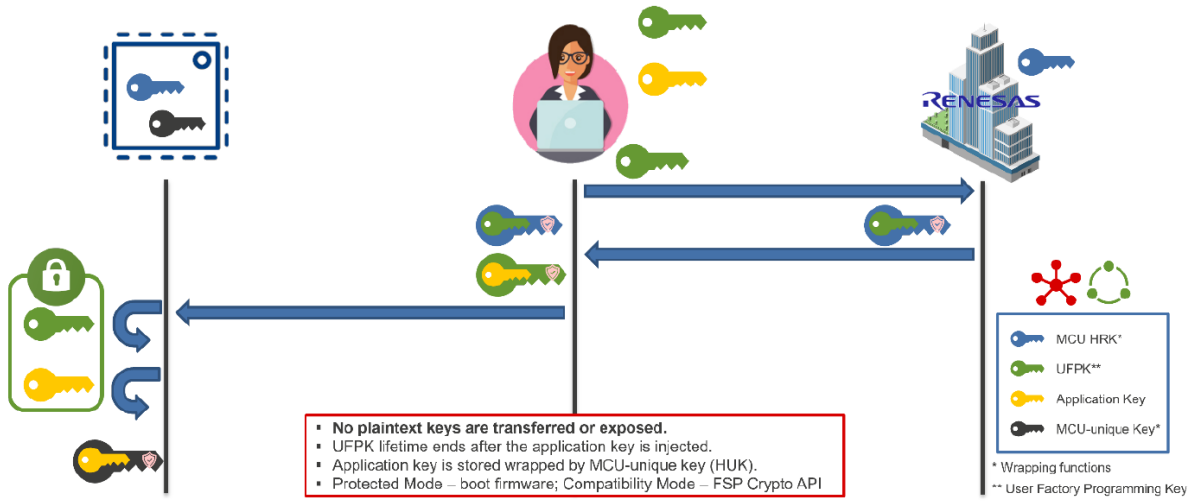
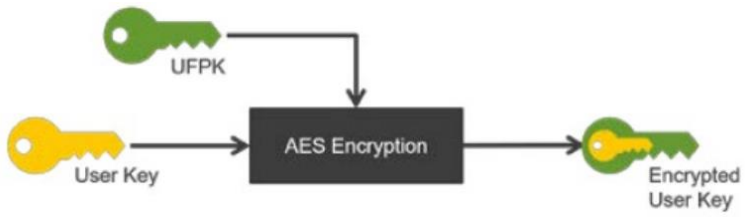


Figure 2 - Renesas Key Wrapping Service

The UFPK from Renesas can now be used to encrypt any keys (**Error! Reference source not found.**) that are required to be injected into the target MCU. This ensures that keys transferred from a user to a Renesas RA device are not accessible by any external entity, i.e. a factory, and can only be unwrapped and accessed internally by a Renesas MCU with the HRK.



Along with the UFPK, two sets of secp256r1 ECC Key Pairs must be generated when using the secure boot functionality, required to establish the chain of trust.

3.1.1 Setup Phase steps

The setup phase can be summarised with the following steps:

1. **Register with Renesas DLM Server:** establish a PGP-encrypted communication channel with the Renesas Device Lifecycle Management (DLM) Server. This is a one-time process involving the exchange of public PGP keys (i.e. obtaining *keywrap-pub.key,* sent by the DLM Server via email) to ensure that sensitive material is never sent in plaintext.
2. **Obtain the Wrapped User Factory Programming Key (W-UFPK)**:
   - Generate a random 256-bit UFPK
   - Encrypt this UFPK using the Renesas PGP public key (*keywrap-pub.key*) and upload it to the DLM Server

- The Renesas Key Wrap Service wraps the UFPK using the Renesas Hardware Root Key (HRK), a unique key mirrored inside the MCU's security engine. The resulting W-UFPK is specific to the RA8 MCU Group and the security engine mode (e.g., Protected Mode).
  This key should be unique per product family.

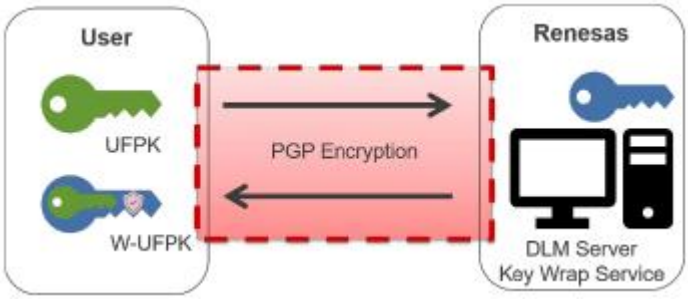


Figure 3 - Wrapping the UFPK

3. **Generate Production ECC Key Pairs:** generate two sets of ECC NIST P-256 (secp256r1) key pairs:
   - OEM_ROOT Key Pair: the anchor for the Chain of Trust
     - OEM_ROOT_PK: the public part of OEM_ROOT key
     - OEM_ROOT_SK: the private (secret) part of OEM_ROOT key
   - OEM_BL Key Pair: used for signing the second-stage bootloader (e.g., MCUboot)
     - OEM_BL_PK: the public part of OEM_BL key
     - OEM_BL_SK: the private (secret) part of OEM_BL key

   These keys should be unique per product family.
4. **Generate AL and RMA Keys:** optionally generate 128-bit secret keys for AL1 (non-secure debug access), AL2 (full debug access), and RMA (Renesas failure analysis)

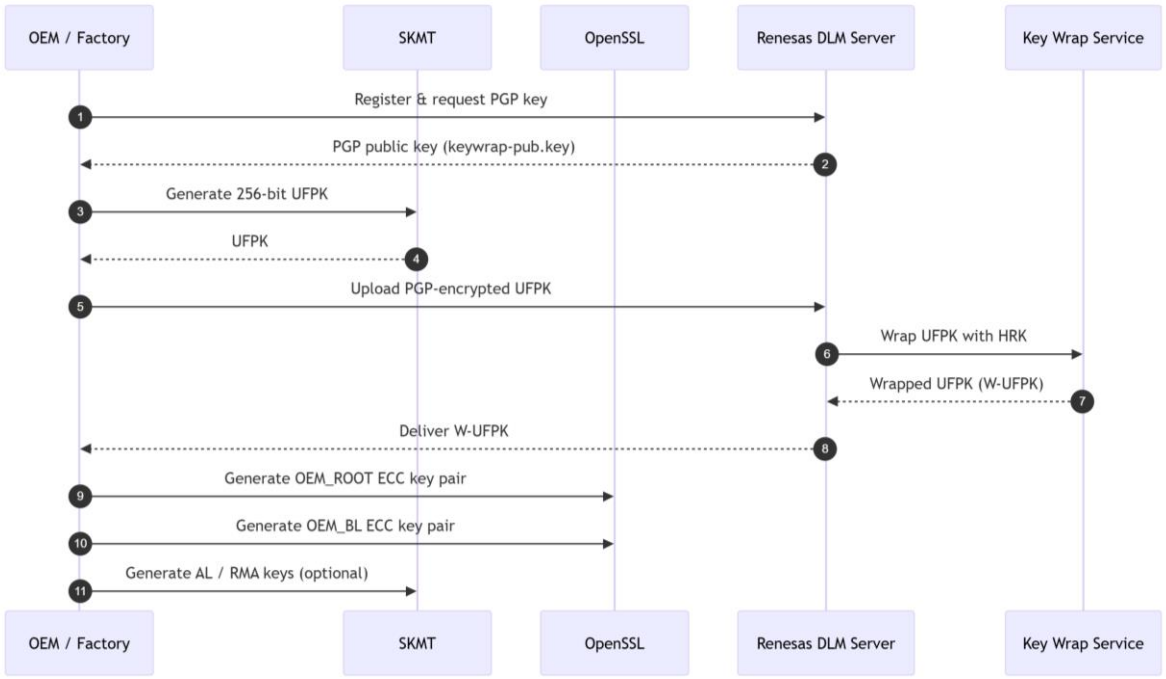


Figure 4 - Setup Phase

3.2 Phase 2: Signing and Credential Preparation

This phase generates the artifacts necessary to establish the Chain of Trust and enable Secure Boot on the device.

In a secure system, an application program must be executed only after confirming that it has not been altered, either maliciously or inadvertently. This confirmation, typically performed by an immutable boot loader, can be a simple integrity check, or it can involve signature verification to ensure authenticity. However, it is important to note that the legitimacy of the boot loader itself must also be guaranteed.

This example utilises an authenticity check which is the most secure. During initial MCU programming, the main application is authenticated using ECDSA with the secp256r1 (NIST P256) ECC curve. Prior to execution, the application is authenticated using HMAC-SHA256 with the MCU's Hardware Unique Key (HUK).

The secure boot sequence is often implemented in stages, starting with an immutable entity to ensure a strong Root of Trust (RoT). The RA8 Series MCUs include an immutable First Stage Bootloader (FSBL), which provides this RoT as a starting point for a secure boot sequence. Typically, the FSBL validates the OEM's bootloader (e.g. MCUBoot), which then validates the remainder main OEM application, forming a "Chain of Trust" for booting the system. Thus, the availability of the FSBL allows for securely updating the OEM bootloader.

In **Error! Reference source not found.** is a high-level representation of the usage of the FSBL in an embedded system in MCU single chip mode.

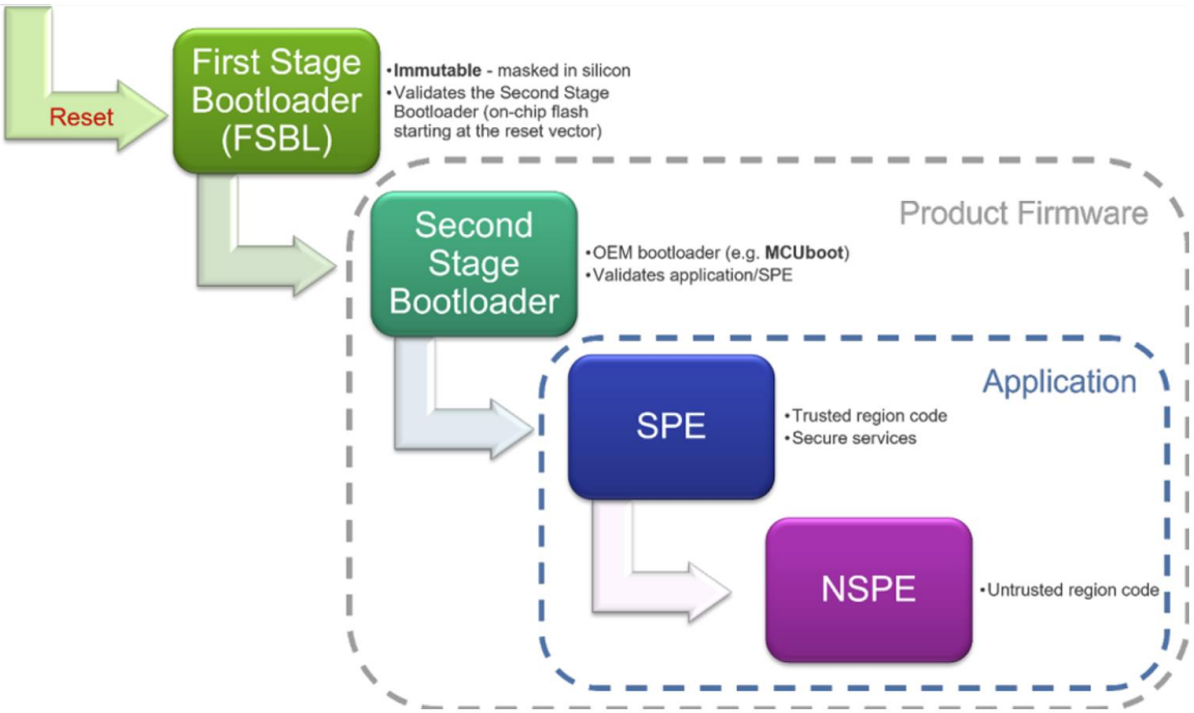


Figure 5 – Single-chip mode MCU with FSBL enabled

3.2.1 Renesas Chain of Trust

In the RA8 Series, the Chain of Trust is rooted in the First Stage Bootloader, an immutable entity masked in ROM that provides a silicon-based Root of Trust. The chain is anchored by the SHA256 hash of the OEM_ROOT_PK, which is securely stored in the MCU's hardware-locked HOEMRTPK register during production.

Upon reset, the FSBL uses this hardware-locked hash to verify the Key Certificate, which is signed by the OEM_ROOT_SK and contains the OEM_BL_PK. Once the Key Certificate is authenticated, the FSBL uses the validated OEM_BL_PK to verify the Code Certificate. This Code Certificate, signed by the OEM_BL_SK, contains the digital signature of the OEM Bootloader (OEM_BL) binary and enforced metadata such as the Image Version for anti-rollback protection. After the FSBL successfully confirms the authenticity of the OEM_BL, it calculates a unique OEM_BL_digest using the device's Hardware Unique Key (HUK) for future high-speed verification and then jumps to the to continue the boot sequence.

Tools such as Renesas' Secure Key Management Tool (SKMT) can be used to generate the Key and Code Certificate.

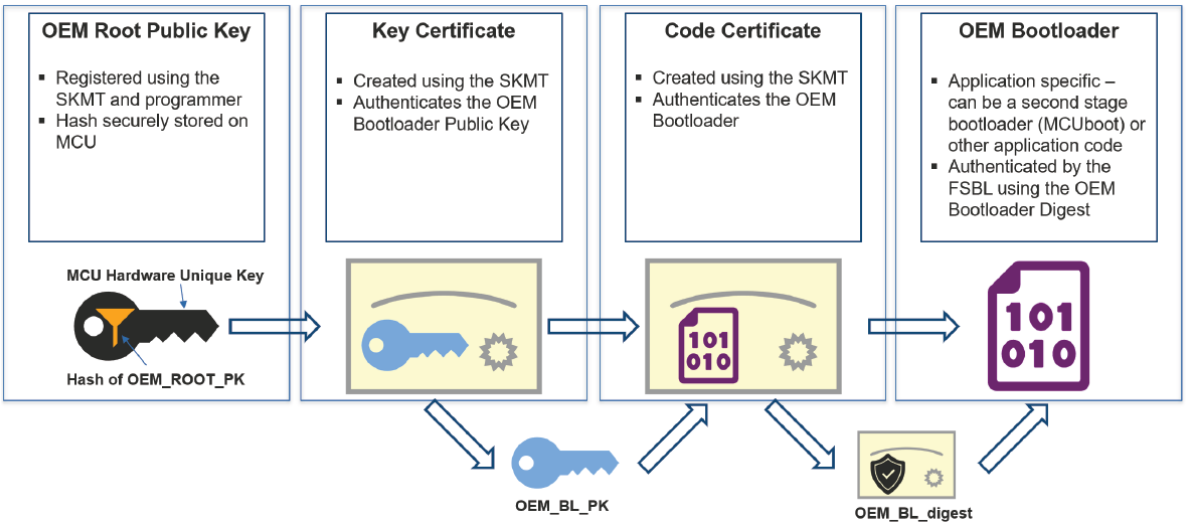


Figure 6 - RA8 Chain of Trust

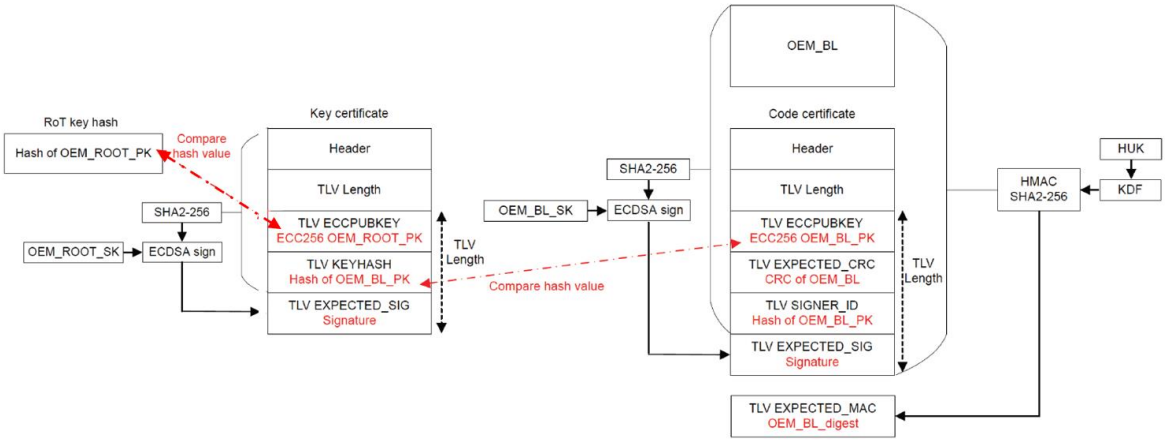


Figure 7 – Verification process with Key and Code Certificate

To complete the Chain of Trust, an additional set of secp256r1 ECC Key Pair is necessary to allow the second stage bootloader (OEM_BL) to verify the application code:

- CUSTOMER_PK: the public part of the Customer Key pair
- CUSTOMER_SK: the private (secret) part of the Customer Key pair

In the case of MCUBoot, *imgtool* is used to sign the application using the CUSTOMER_SK.

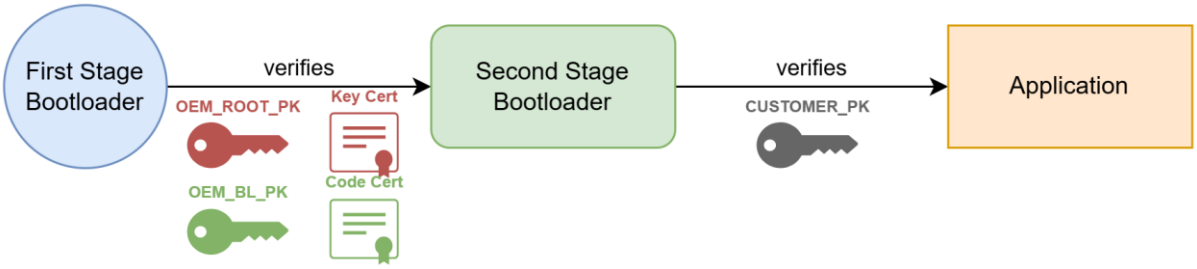


Figure 8 - Verification

The diagram in Figure 9 shows an overview of the key usage during Secure Boot:

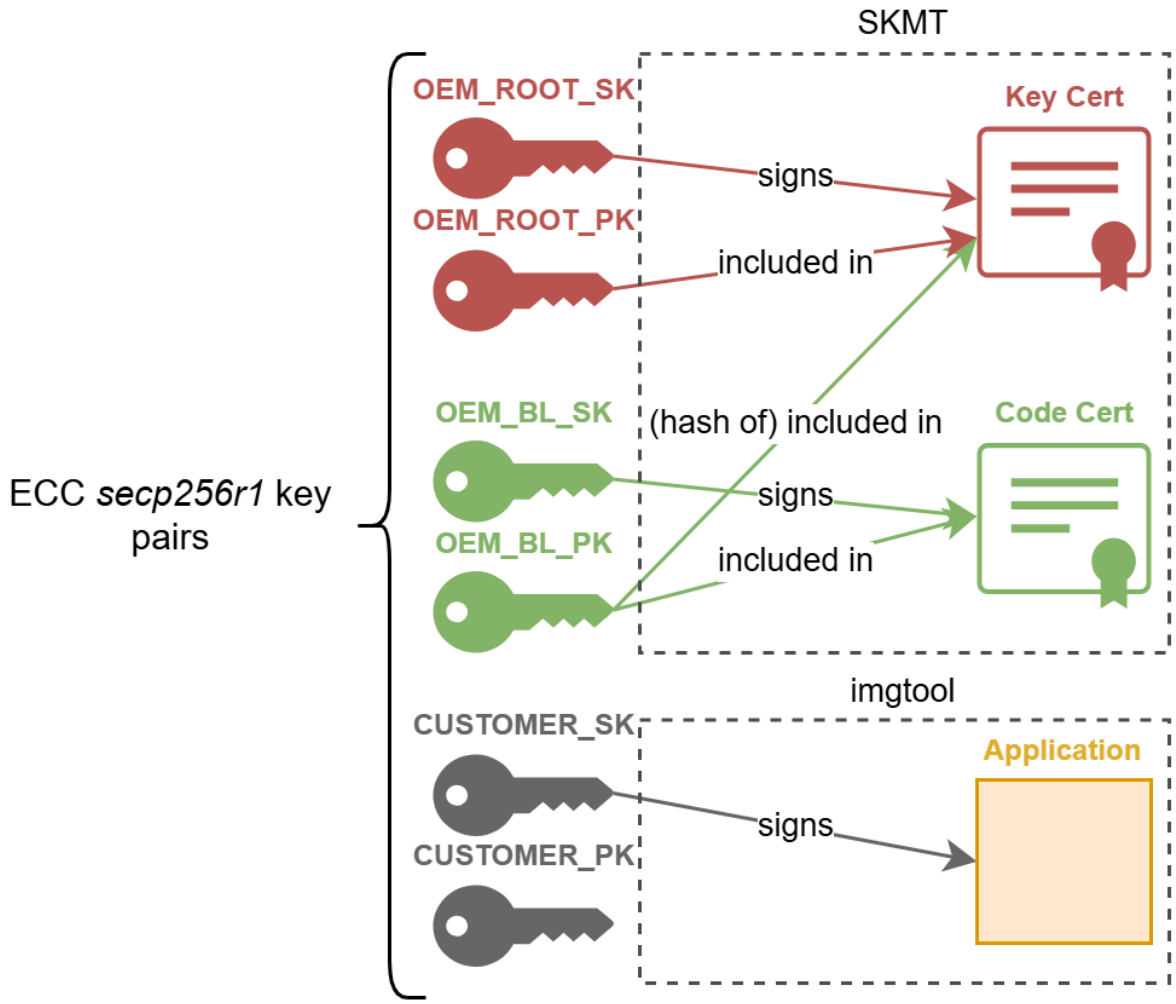


Figure 9 - Keys involved in RA8 Secure Boot

3.2.2 Renesas Security System

The Renesas RA8 family includes a dedicated security engine (RSIP-E51A) which is a cryptographic subsystem that is integrated into the MCU silicon, managed and protected by dedicated control logic. The cryptographic operations are physically isolated from the rest of the chip, with dedicated RAM for holding sensitive material (e.g., plaintext keys) during cryptographic operations. The security engine is further protected with TrustZone isolation technology.

The diagram below shows the On-chip isolation of the security engine:

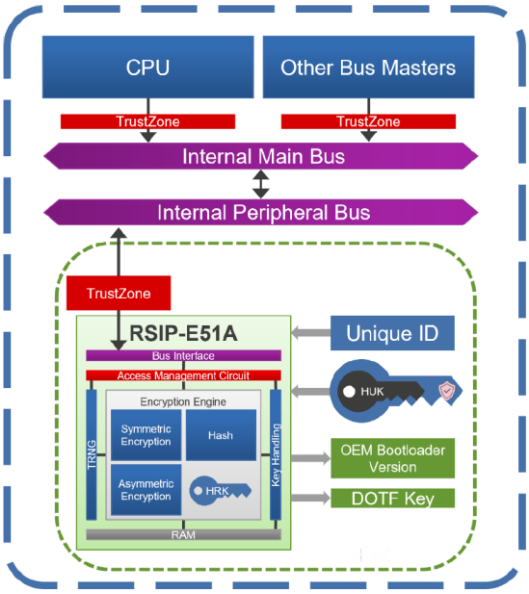


Figure 10 - On-chip isolation with RSIP-E51A

The secure key storage mechanism leverages the chip's Hardware Unique Key (HUK), a 256-bit key that is unique for each chip and is accessible only by the security engine. Application keys that have been wrapped with this key via a key injection or key update process can be stored securely at any location in any memory. This mechanism provides unlimited secure key storage and cloning protection.

3.2.3 Signing Phase steps

This section provides an overview of the steps required to generate all of the artifacts discussed above.

5. **Wrap the OEM Root Public Key:** use the UFPK to wrap the OEM_ROOT_PK into a Renesas-compatible .rkey format. This file also includes the W-UFPK, allowing the MCU to unwrap the root key internally during injection
6. **Wrap the lifecycle keys:** if present, the AL1, AL2, and RMA keys must also be wrapped using the UFPK to create their respective *.rkey* injection files
7. **Create and Sign the OEM_BL and Application:**
   - Develop the OEM_BL (typically MCUboot) and the Application firmware
   - Sign the application image using the imgtool utility provided with MCUboot
   - The public portion of the key used to sign the application is also embedded within the MCUboot binary to continue the Chain of Trust
8. **Generate Key and Code Certificates:** these binary certificates establish the hardware-verified Chain of Trust.
   - Key Certificate: contains the OEM_ROOT_PK and the hash of the OEM_BL_PK. It is signed with the OEM_ROOT_SK and its primary function is to authenticate the OEM_BL_PK by verifying the provided OEM_ROOT_PK against the hardware-locked Root of Trust hash (HOEMRTPK).
   - Code Certificate: contains the OEM_BL_PK, image metadata (load address, size, and Image Version), and a signature of the OEM_BL binary. It is signed with the OEM_BL_SK and it is used by the FSBL to authenticate the bootloader binary and enforce anti-rollback protection.

Anti-Rollback: the Image Version (1–64) included here is used by the FSBL, which will only permit updates where the version number is higher than the currently stored value.

Note that an extra step to configure boundaries is necessary when using TrustZone.

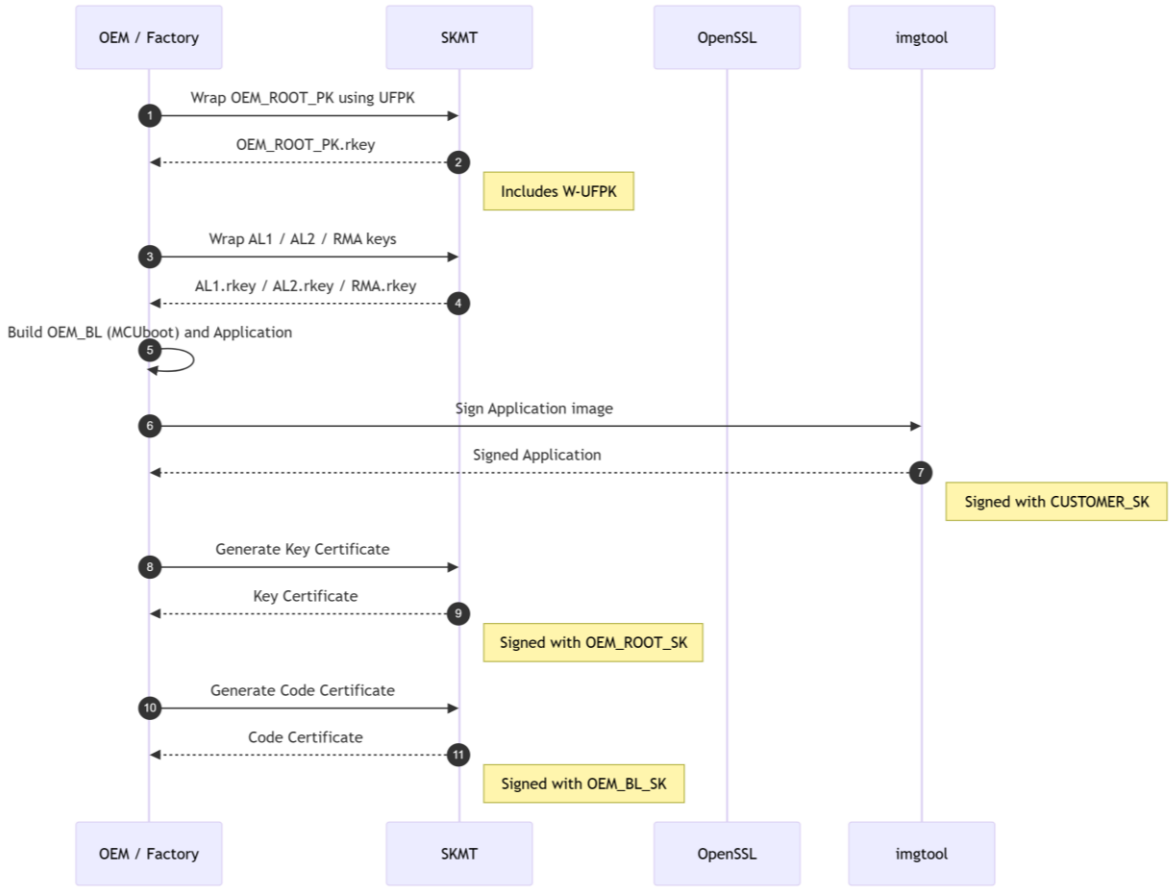


Figure 11 - Signing Phase

3.3 Phase 3: Programming and Lockdown

The final phase executes the physical injection of code and credentials via the RA8 MCU's boot mode.

Programming the RA8M1 is facilitated by the factory boot firmware, an immutable program embedded in the MCU ROM that provides access to on-chip flash memory via SCI/UART, USB, or JTAG/SWD interfaces.

The procedure initiates in boot mode, triggered by pulling the MD pin low during reset, during which the external tool establishes a connection. Once communication is established, the tool issues the *initialize* command to reset the device to the OEM state at Protection Level 2 (PL2), which erases existing User, Data, and Config areas. During the subsequent programming sequence, the boot firmware injects the Root of Trust (the hash of the OEM_ROOT_PK), writes the binary images and their Key and Code Certificates, and performs an ECDSA-based authenticity check. Upon successful verification, the firmware generates a unique OEM_BL_digest using the chip's Hardware Unique Key (HUK) for high-speed hardware booting and transitions the device to a final deployment state, such as PL0 or LCK_BOOT, to secure the onboard intellectual property.

The secure key injection process leverages the MCU's Hardware Root Key (HRK), a key that is contained within the security engine and is common across all chips (and is only known by Renesas). This key is used only for the secure key injection process, and it is not used for secure storage of application keys, nor it is not accessible outside the security engine.

3.3.1 Programming Phase Steps

This sections provides an overview of the steps necessary to provision and lock the device.

9. **Initialise the device:** issue the *initialize* command to transition the device to the OEM state and PL2. This erases all non-locked flash regions and sets the device to a known factory default
10. **Inject keys:** inject the Root of Trust (wrapped OEM_ROOT_PK .rkey file) and the lifecycle keys (AL1, AL2, RMA .rkey files). The MCU's security engine uses its Hardware Root Key (HRK) to internally recover the UFPK from the provided W-UFPK. This plaintext UFPK is then used to unwrap the actual keys, which are stored in secure, non-user-accessible memory. For the Root of Trust, the device specifically stores the SHA256 hash of the public key in the HOEMRTPK register.
11. **Data Programming:** program the combined OEM_BL + Application binary and write the Key and Code Certificates to the addresses defined in the SACCn registers
12. **Verification:** the boot firmware verifies the certificates and the OEM_BL, it then generates an OEM_BL_digest using the MCU's Hardware Unique Key (HUK). This digest provides high-speed, unclonable verification for every subsequent reset
13. **Lock the Device:** once verified, the device can be moved from a "development" to a "deployed" state:
    - Transition PL: move the Protection Level to PL1 (limited debug) or PL0 (no debug). If AL keys were injected, it is still possible to temporarily regress the state for debugging purpose.
    - Disable initialisation: optionally issue a command to permanently disable the *initialize* command. This makes the current Root of Trust and firmware configuration irreversible, preventing any future erasure of the flash.
    - Transition DLM: move to the LCK_BOOT state to disable the boot mode interface entirely. This is a final, irreversible step that fully locks the device.

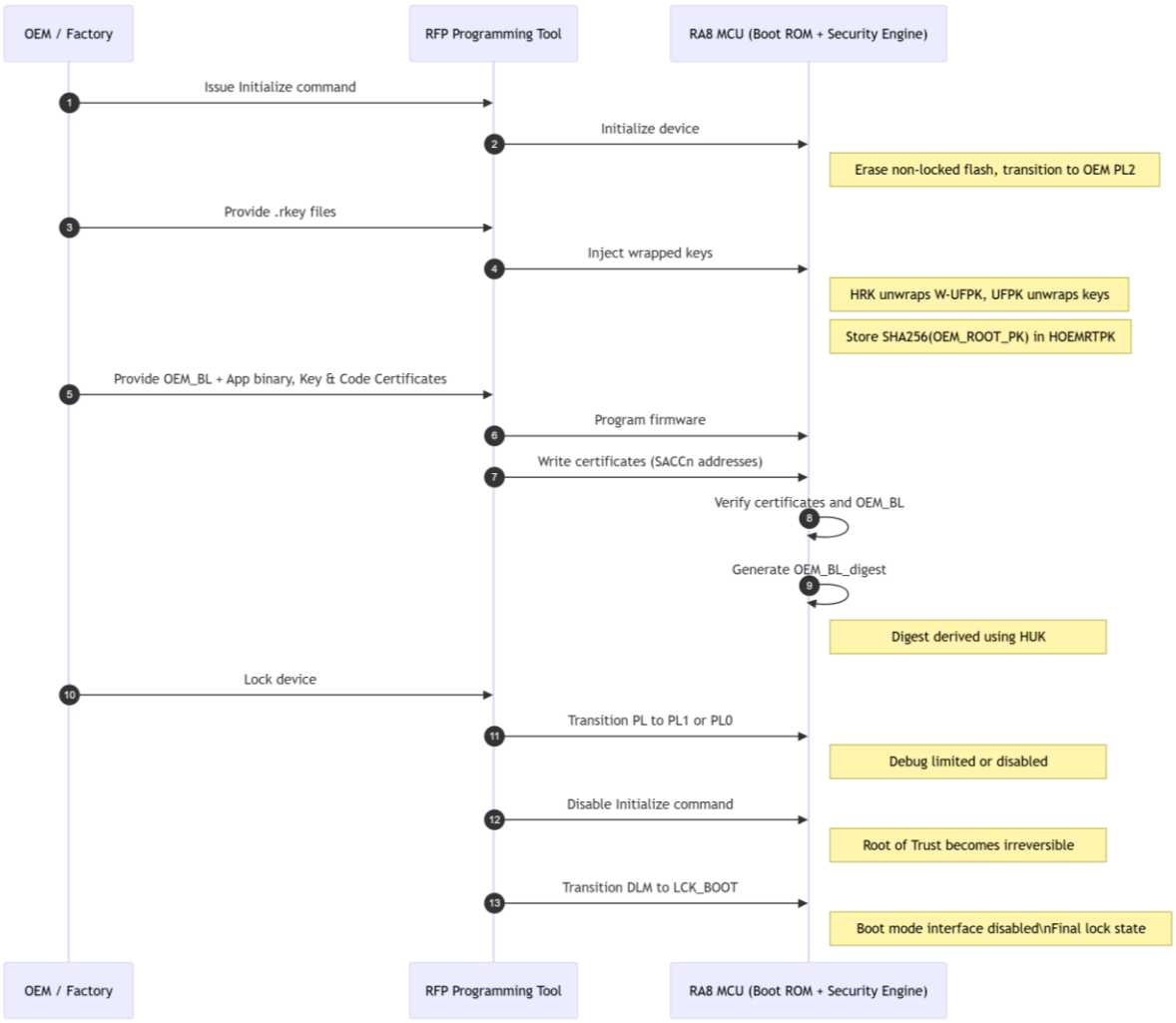


Figure 12 - Programming Phase

A diagrammatic representation of the entire Renesas process is shown below:

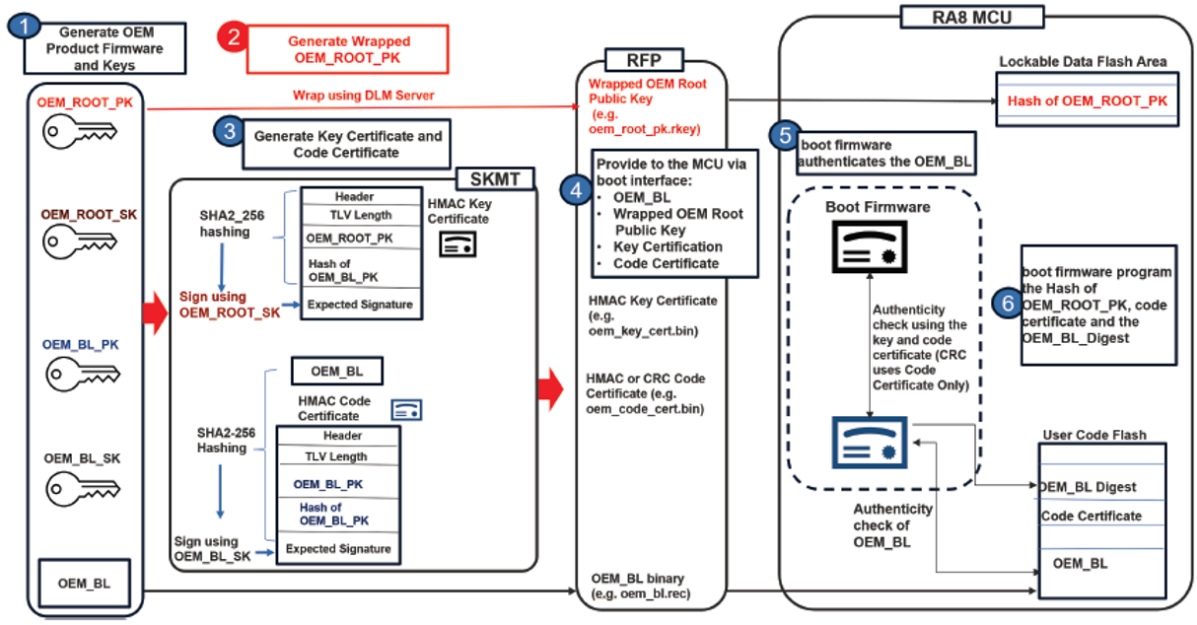


Figure 13 - Complete provisioning process

4 Solution Overview

This chapter provides an overview of the proposed provisioning solution for RA8 microcontrollers. Detailed implementation guidance, including code snippets, is provided in 7.Provisioning Implementation Guide.

The solution presented here serves as a reference implementation for production environments and it is important to note that this is one possible approach rather than the only valid method for securing RA8 devices. Readers should treat this chapter as a guide to understanding the technical details of the provisioning process, which can then be used as a foundation to develop a production-ready provisioning tool tailored to their specific requirements.

The idea is to replace standard manual tools, such as Security Key Management Tool (SKMT) and Renesas Flash Programmer (RFP) with a single unified Python tool, capable of managing the entire lifecycle, from signing artifacts to final device lockdown. Furthermore, the tool introduces seamless integration with HSMs, for a production-level secure key management.

In reference to the flow discussed in 3.Secure Boot on RA8M1, the tool covers Phase 2 and Phase 3 in full.

4.1 Design

To ensure a high level of security and reliability for the RA8 Series, the following decisions have been made as part of the solution proposed:

- **Signature Mode authentication:** the system uses Secure Boot based on ECC NIST P-256 (secp256r1) for all verification stages. This mode validates application authenticity during programming via ECDSA and ensures high-speed hardware-based verification upon reset using HMAC-SHA256 and the MCU's Hardware Unique Key (HUK).
- **MCUBoot as the Second Stage Bootloader (SSBL):** MCUBoot is utilised as the SSBL to provide a standard, fail-safe update mechanism. It manages image slots and handles update strategies such as *swap* or *overwrite,* while the FSBL remains the silicon-based Root of Trust.
- **Hardware Security Module (HSM) Key Management:** all production key pairs, i.e. OEM_ROOT Keys, OEM_BL Keys and the Application Signing keys (CUSTOMER Keys) are generated, stored and utilised within an HSM. This ensures that sensitive cryptographic material (the private keys) is never exposed, mitigating the risk of credential leak.
- **Custom Python tool:** a unified tool developed in Python is proposed to streamline the provisioning phase, replacing existing manual utilities such as the Security Key Management Tool (SKMT) and Renesas Flash Programmer (RFP). The tool enhances security by interfacing directly with the HSM for all the signing operations and implements the Boot Firmware command protocol. This allows to automate the generation of the artifacts, initialisation, key injection and programming of the target MCU.
- **Setup Phase excluded:** the setup phase (3.1) is deliberately excluded from the automation tool, as it consists of one-time administrative tasks (including web registration and manual acceptance of legal agreements).

The diagram in Figure 14 shows the full provisioning flow with the planned solution. It highlights the different phases discussed:

- Setup Phase in blue
- Signing Phase in orange
- Programming Phase in green

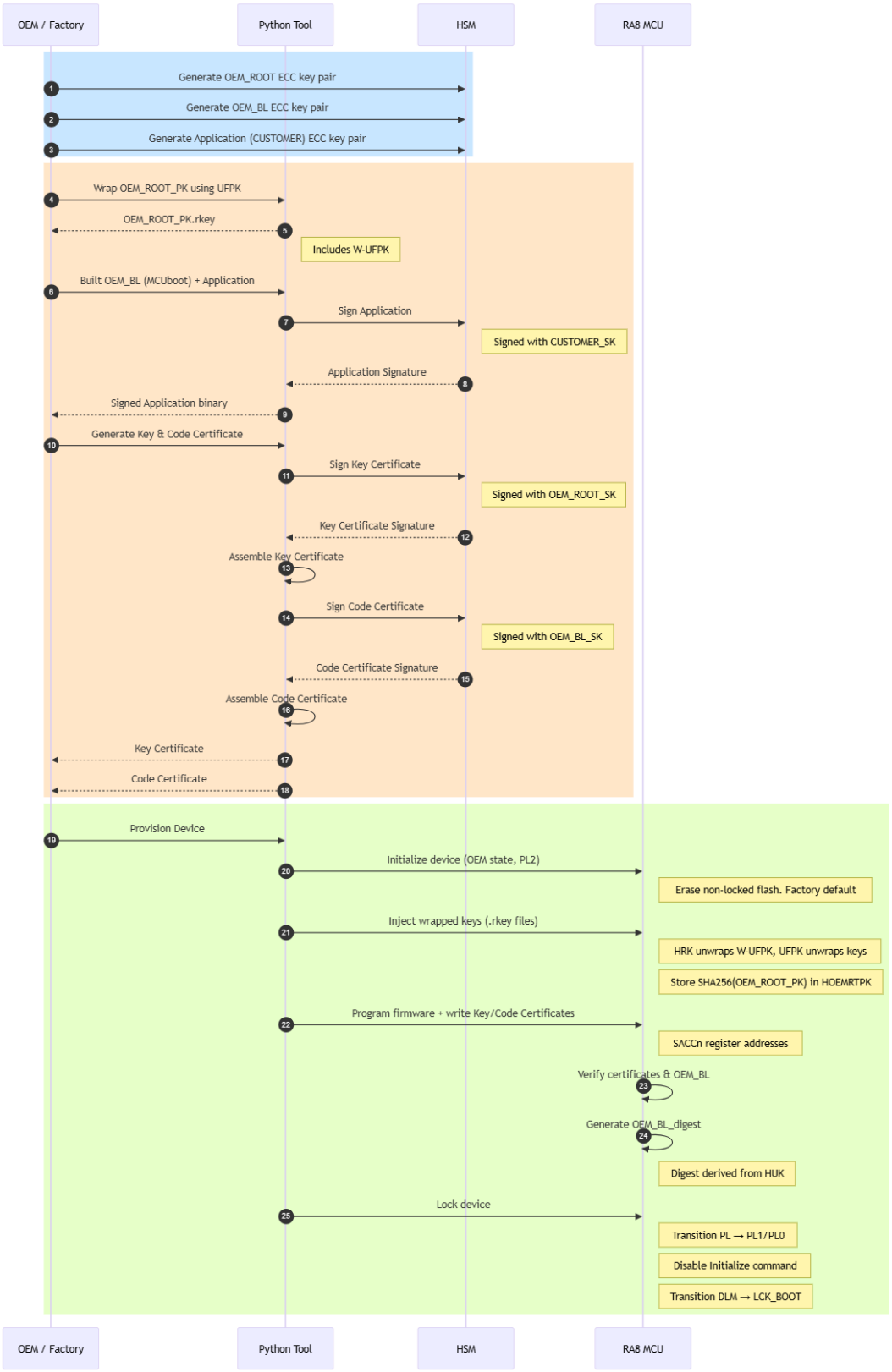


Figure 14 - Full Flow

5 Setup

This chapter covers all the steps necessary to setup the development environment required to implement the tool. The OS used as reference is Windows 10/11.

5.1 Python

Python is the programming language adopted to develop the tool, version 3.10 or higher is required.

Download and install Python 3.10 or later from https://www.python.org/downloads/.

To verify the correct installation, open a terminal window and run

```
python --version
```

Expected output: Python 3.10.x or later.

5.2 AWS CLI

This section provides instructions for using AWS KMS as the HSM, including the necessary setup steps. However, the tool is designed to be HSM-agnostic, with the workflow and commands remaining largely the same, ensuring that integration with other HSM solutions is straightforward.

The reader is assumed to be in possession of an AWS account with permission to manage KMS.

The AWS CLI is required for interacting with the AWS Key Management Service, the HSM of choice for this guide.

Download:

- Visit https://aws.amazon.com/cli/
- Download AWS CLI v2 for Windows (MSI installer)

Installation:

1. Run the MSI installer
2. Complete installation

Configuration:

From a terminal window:

```
aws configure
```

Data required for the configuration:

- AWS Access Key ID
- AWS Secret Access Key
- Default region (e.g., eu-west-2): KMS (and consequently the keys) are region-specific; ensure to configure the correct region
- Output format (json)

Verify installation:

From a terminal window

```
aws --version
```

```
aws kms list-keys
```

5.3 Create Keys in AWS KMS

This section describes how the necessary keys can be created within the AWS Key Management Service.

As discussed in previous chapters, three ECC P-256 keys are required, as reported in Table 1.

Table 1 - HSM Keys

Key Name	Purpose	Usage
OEM_ROOT Key	Root of trust for device	Signs Key Certificate
OEM_BL Key	Bootloader authentication	Signs Code Certificate
CUSTOMER Key	Application signing	Signs application firmware

5.3.1 Create keys via AWS Console

1. Navigate to *KMS* > *Customer managed keys* > *Create key*
2. Select:
    - Key type: ***Asymmetric***
    - Key usage: ***Sign and verify***
    - Key spec: ***ECC_NIST_P256***
3. Set alias (e.g. oem-root-key, oem-bl-key, customer-key)
4. Configure key policy to allow IAM user
5. Copy the Key ID (UUID format: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)

5.3.2 Create keys via AWS CLI

From a windows terminal:

```
# Create OEM Root Key
aws kms create-key --key-spec ECC_NIST_P256 --key-usage SIGN_VERIFY --description "OEM Root Key for RA8M1"

# Create OEM Bootloader Key
aws kms create-key --key-spec ECC_NIST_P256 --key-usage SIGN_VERIFY --description "OEM Bootloader Key for RA8M1"

# Create MCUboot App Key
aws kms create-key --key-spec ECC_NIST_P256 --key-usage SIGN_VERIFY --description "Customer Application Signing Key"
```

5.3.3 Verify AWS Access

From Terminal

```
# List your KMS keys
aws kms list-keys

# Verify specific key access
aws kms describe-key --key-id <KEY_ID>

# Test signing permission
aws kms get-public-key --key-id <KEY_ID>
```

6 Create and Configure Projects

This chapter contains step-by-step instructions to guide the reader through the creation of an MCUBoot bootloader and application project, using Renesas e^{2}studio and Renesas Flexible Software Package (FSP). This only serves as an example and engineers should develop the project according to their requirements.

If an MCUboot-based bootloader and application project are already available, this chapter can be safely skipped.

The instructions contained below aim to create a secure bootloader and application projects for the Renesas RA8M1 microcontroller using MCUboot, with the following configuration:

- Bare metal implementation
- Secure boot process with digital signatures
- Dual bank flash configuration with Direct XIP update strategy

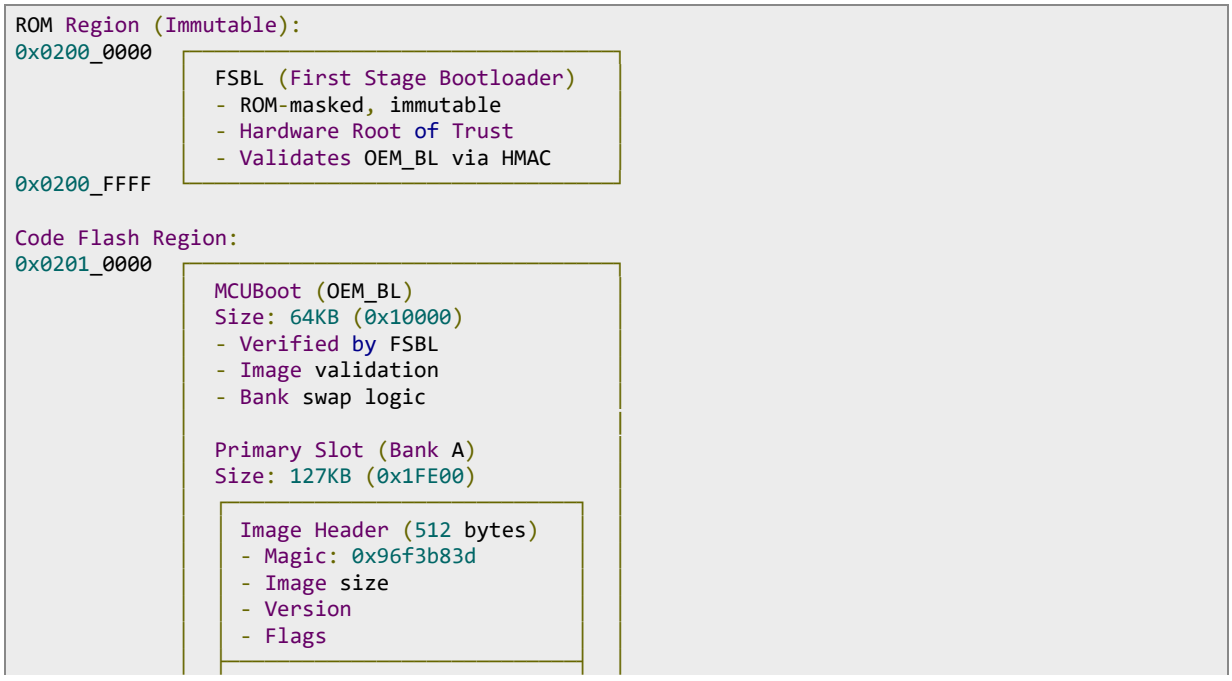
As part of these steps, the following projects will be created within the workspace:

- MCUBoot bootloader project
- Application project, with the bootloader as preceding project
- Solution project, needed for defining the memory configuration across the two projects

6.1 Prerequisites

- e^{2}studio IDE (v2025-12)
- Renesas Flexible Software Package (v6.1.0)
- Python 3.x for image signing tools
- RA8M1 Evaluation Kit
- USB Cable for programming (included in the Evaluation Kit)

6.2 Flash Memory Layout



```
                  Application Firmware
                  - Vector table
                  - Code section
                  - Data section
                  TLV Footer
                  - SHA256 hash
                  - ECDSA P-256 signature
                  - Protected TLVs
0x0203_0000
                  Secondary Slot (Bank B)
                  Size: 127KB (0x1FE00)
                  - OTA staging area
                  - Same structure as Primary
0x0205_0000
                  Reserved
                  Size: depending on uc variant
0x0206_0000
                  Code Certificates

                  - Code certificate for Bank 0
                  - Code certificate for Bank 1
0x0300_A100
                  Config area 0
                  Size: 128bytes
0x0300_A200
                  Config area 1
                  Size: 128bytes
0x2703_0800
                  FSBL configuration
                  Size: variable
                  - OSM registers configuration

Critical Fixed Addresses (Per Renesas Specification):
0x0206_0000      SACC0 - Code Certificate 0 (256 bytes)
0x2703_0380      HOEMRTPK - OEM Root Public Key Hash (32 bytes)
0x2703_0878      ARC_OEMBL0 - Anti-Rollback Counter Bank 0 (4 bytes)
0x2703_087C      ARC_OEMBL1 - Anti-Rollback Counter Bank 1 (4 bytes)

Special Registers (Fixed Hardware Addresses)¶
Security Registers (Memory-Mapped):
0x2703_0380  HOEMRTPK    - OEM Root Public Key Hash (32 bytes)
0x2703_0878  ARC_OEMBL0 - Anti-Rollback Counter Bank 0 (4 bytes)
0x2703_087C  ARC_OEMBL1 - Anti-Rollback Counter Bank 1 (4 bytes)

Code Flash Locations:
0x0206_0000  SACC0       - Code Certificate 0 storage (256 bytes)
```

6.3 Creating the Bootloader Project

6.3.1 Step 1: Launch e²studio

1. Open e²studio from the applications menu
2. Select or create a workspace when prompted
3. Choose a location for workspace (e.g., *C:/Renesas/RA8M1_Projects*)

6.3.2 Step 2: Create New C/C++ Project

1. In the menu bar, click *File* → *New* → *C/C++ Project*
2. In the dialog that appears, expand *Renesas*
3. Select ***Renesas RA C/C++ Project***

4. Click *Next*

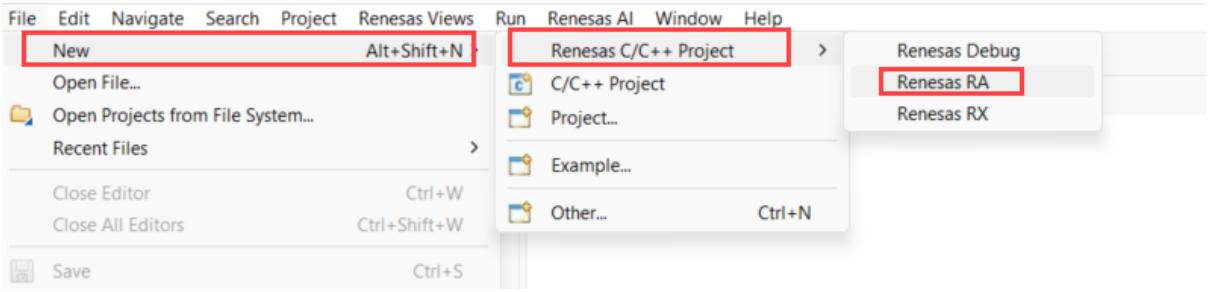


Figure 15 - Create a new Renesas RA C/C++ Project

6.3.3 Step 3: Configure Project Name and Location

1. Enter the project name: e.g. **EK_RA8M1_MCUBOOT_BL**
2. Optionally, change the project location
3. Click *Next*

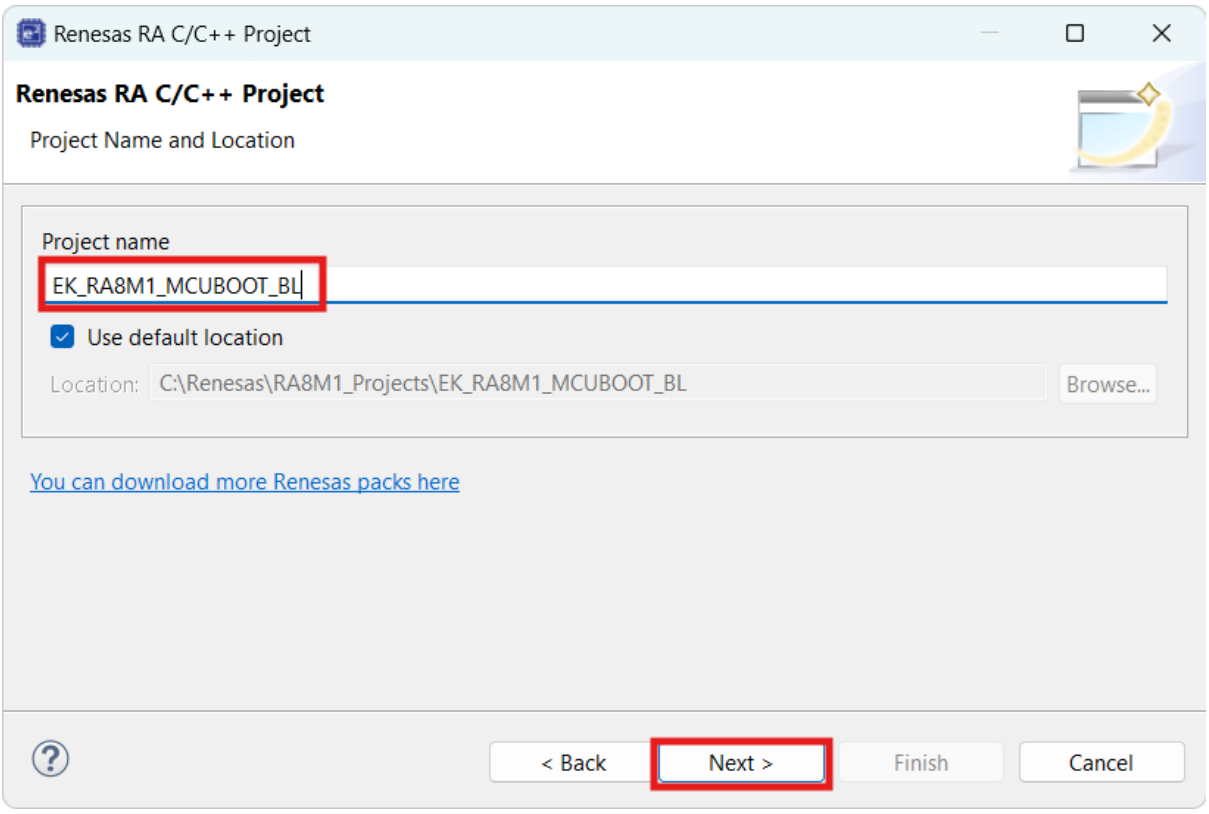


Figure 16 - Set the bootloader project name

6.3.4 Step 4: Select the Board or Device

1. In the *Board* dropdown, select ***EK-RA8M1***
2. Alternatively, select *Device* tab and choose the specific RA8M1 variant
3. Verify the device details match your hardware
4. Click *Next*

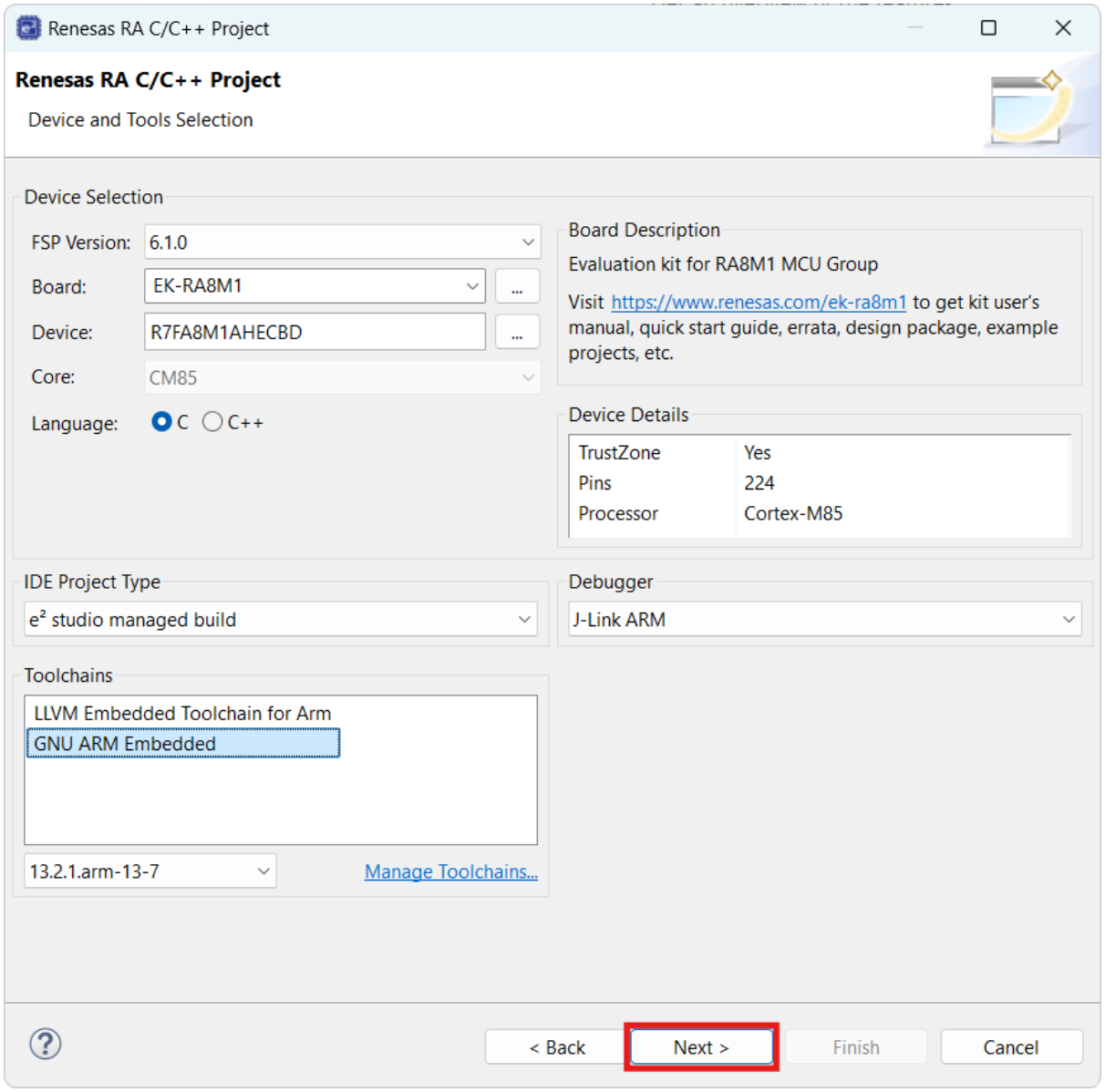


Figure 17 - Select the EK-RA8M1 board

6.3.5 Step 5: Choose Project Type and preceding project

For the RA8M1 bootloader, select the appropriate project type:

1. Select ***Flat (Non-TrustZone) Project***
2. This is suitable for most bootloader implementations
3. Click *Next* to continue.

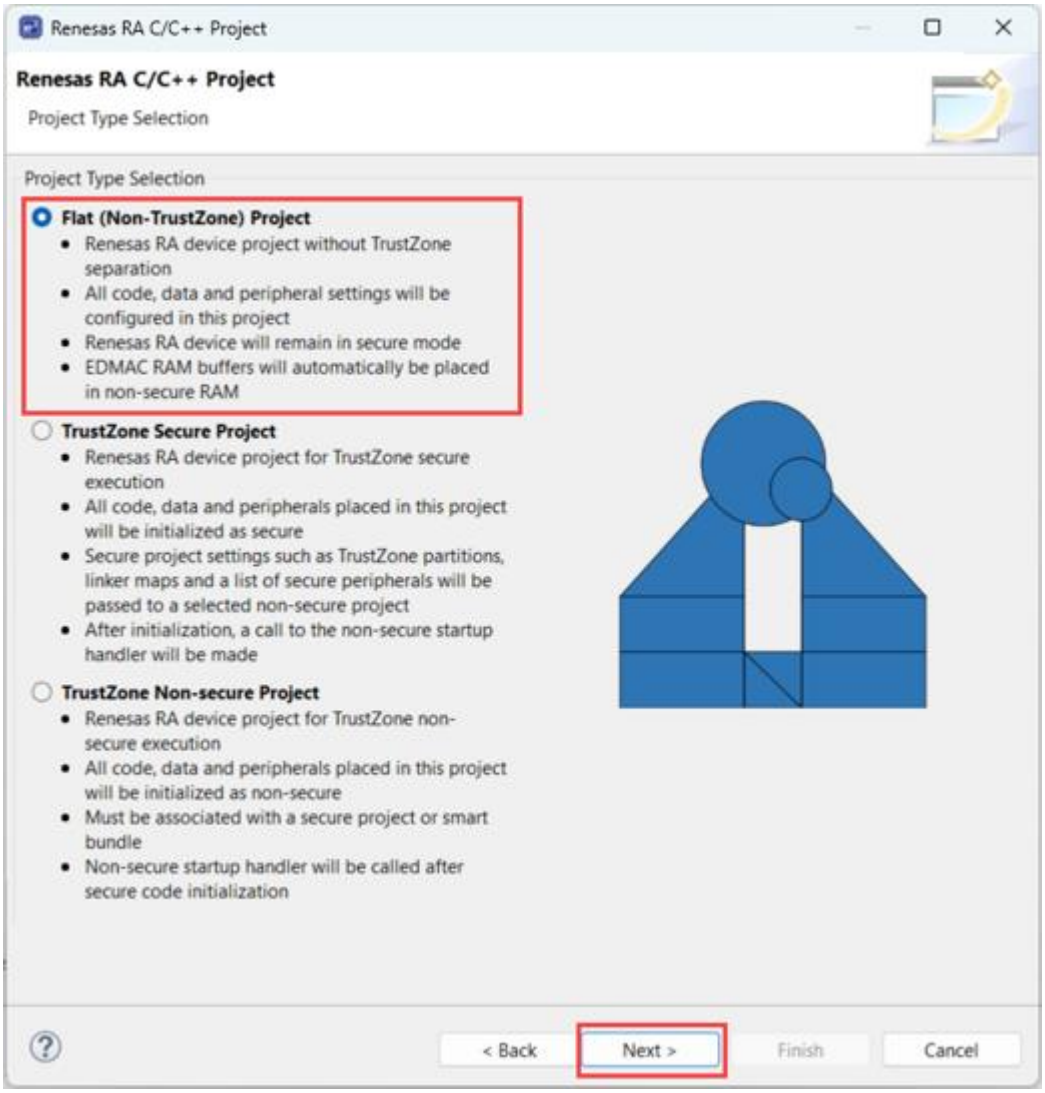


Figure 18 - Select Project type

4. Select ***None*** (no preceding project) and click *Next* to continue

6.3.6 Step 6: Select Build Artifact and RTOS selection

1. Choose ***Build Executable*** (default)
2. Select ***No RTOS*** for the bootloader

   MCUboot operates independently without an RTOS

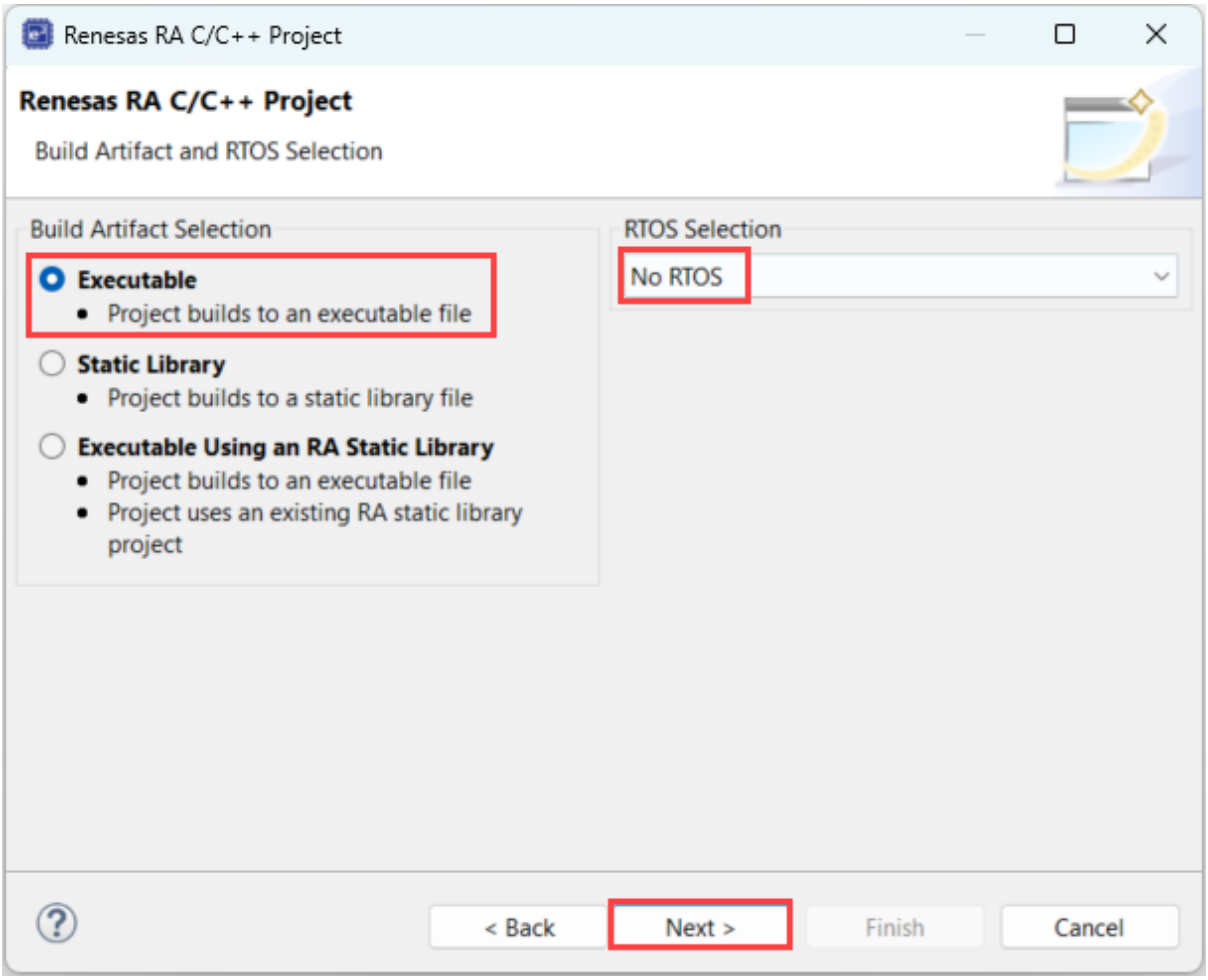


Figure 19 - Build configuration selection

3. Click *Next*

6.3.7 Step 7: Choose Project Template

1. Select ***Bare Metal – Minimal*** from the Project Template list
   This provides a minimal starting point for the bootloader
2. Click *Finish* to create the project

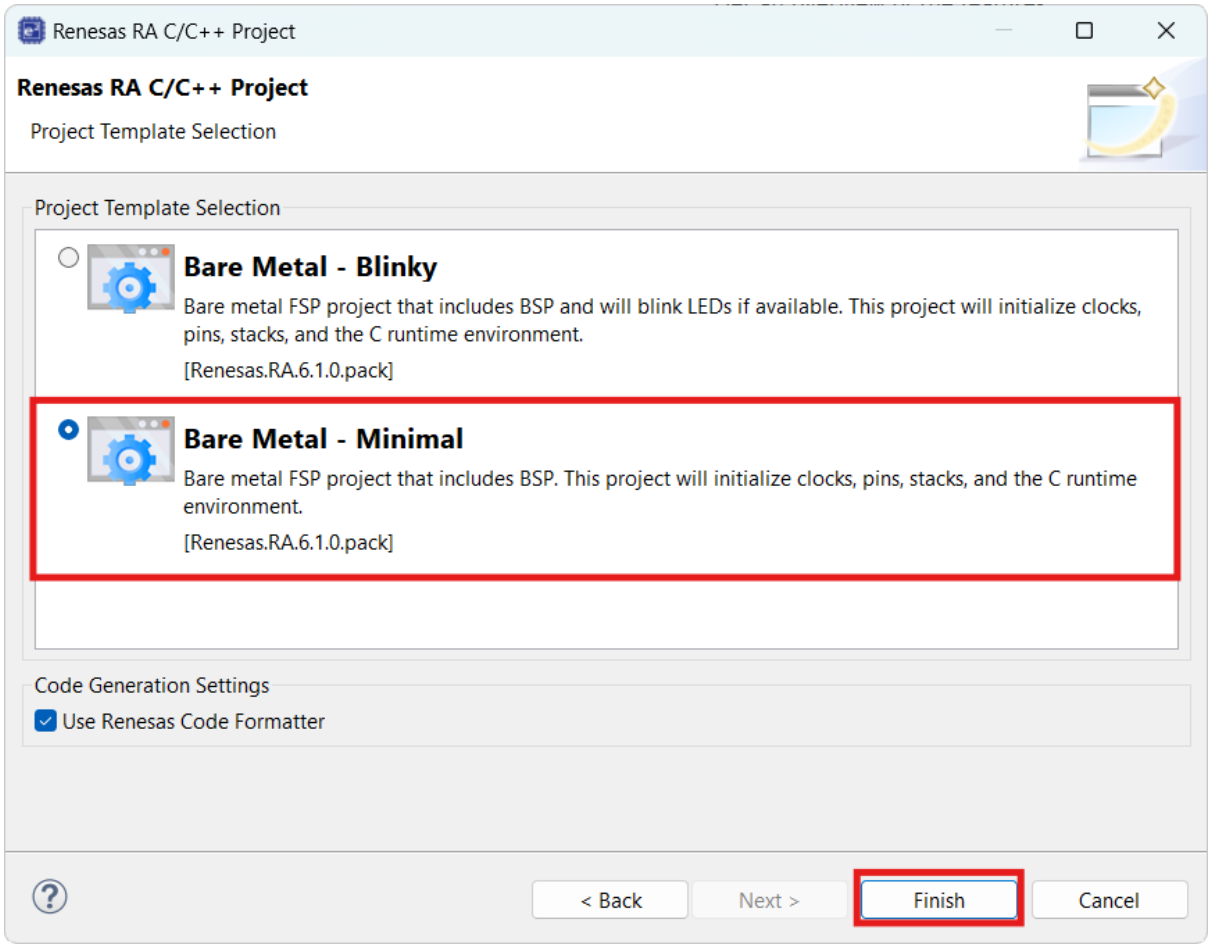


Figure 20 - Select Bare Metal - Minimal template

e² studio will generate the project structure; wait for the process to complete.

6.3.8 Step 8: Add MCUboot Module to the Bootloader

1. In the *Project Explorer*, double-click *configuration.xml* to open the FSP Configuration
2. Click on the *Stacks* tab at the bottom of the configuration window
3. Click *New Stack* button
4. Navigate to *Bootloader → MCUboot*
5. Click *Add* to include the MCUboot module

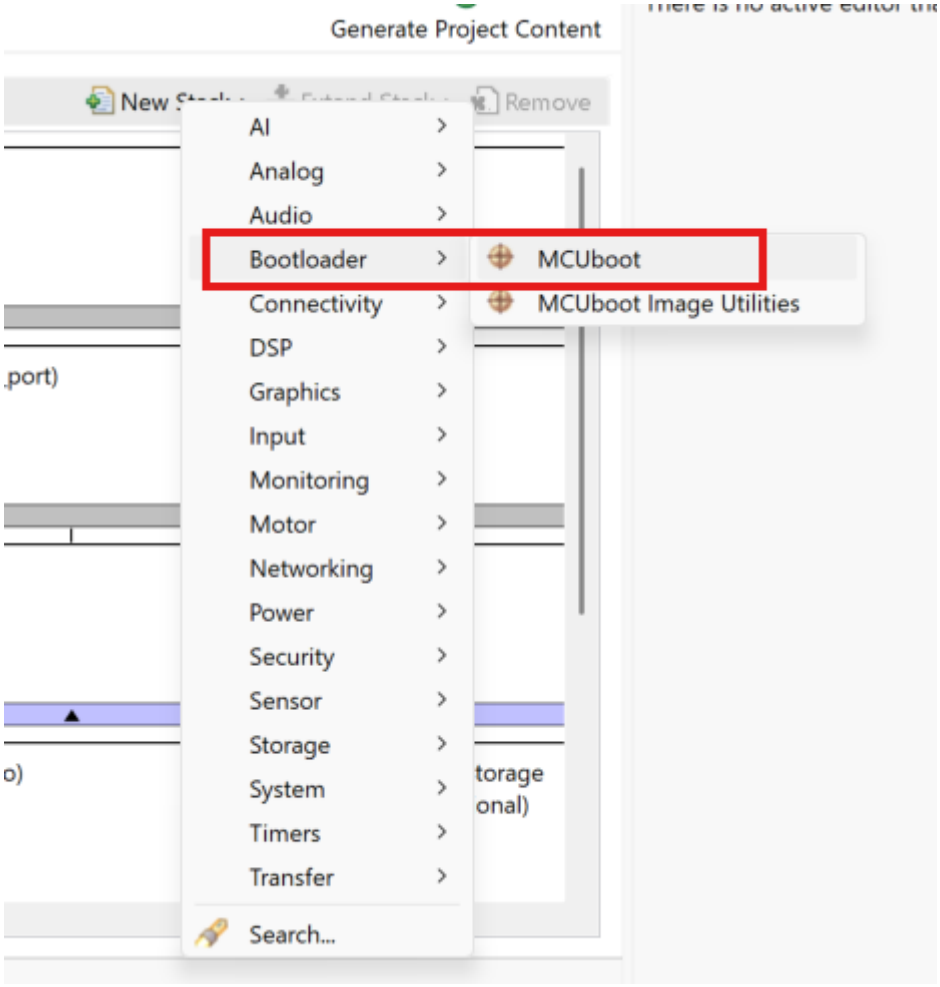


Figure 21 - Add MCUBoot module to the bootloader project

6.3.9 Step 9: Configure MCUboot Properties

In the Properties panel:

1. *Common → General → Upgrade Mode:* select ***Direct XIP*** for Direct XIP mode
2. *Common → Signing and Encryption Options → Signature Type:* ensure it is configured to ***ECDSA P-256***

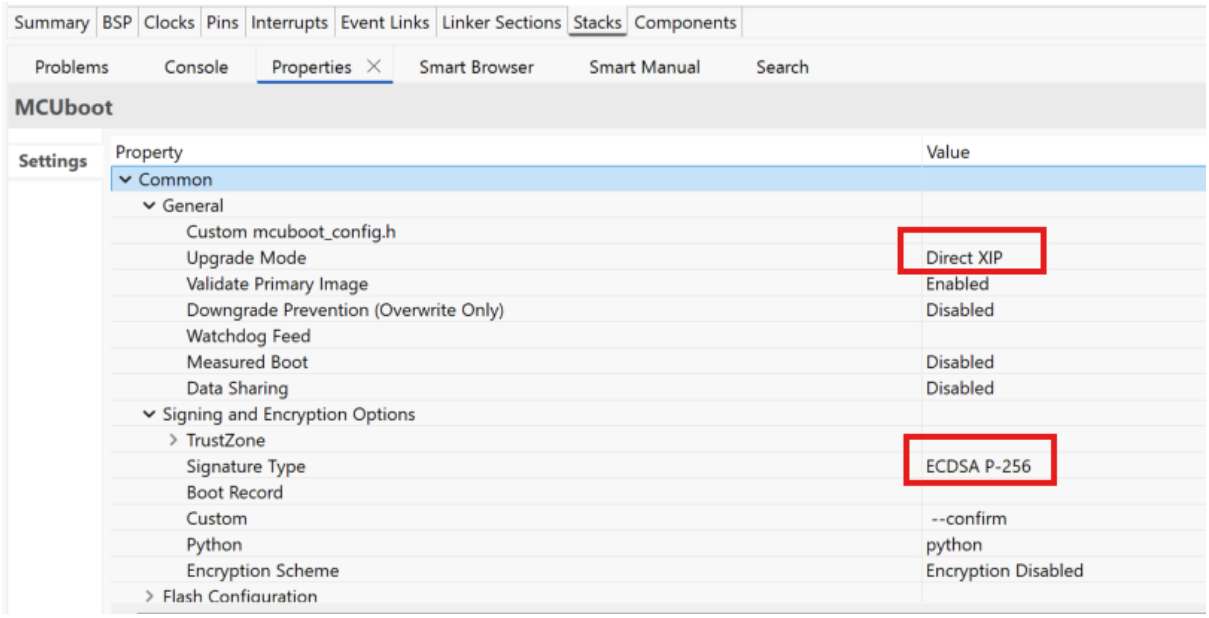


Figure 22 - MCUBoot module properties

6.3.10 Step 10: Add Crypto Module to the Bootloader

1. Go to the *Stacks* tab
2. Click on ***Add Requires a crypto stack -> MbedTLS (Crypto Only)***

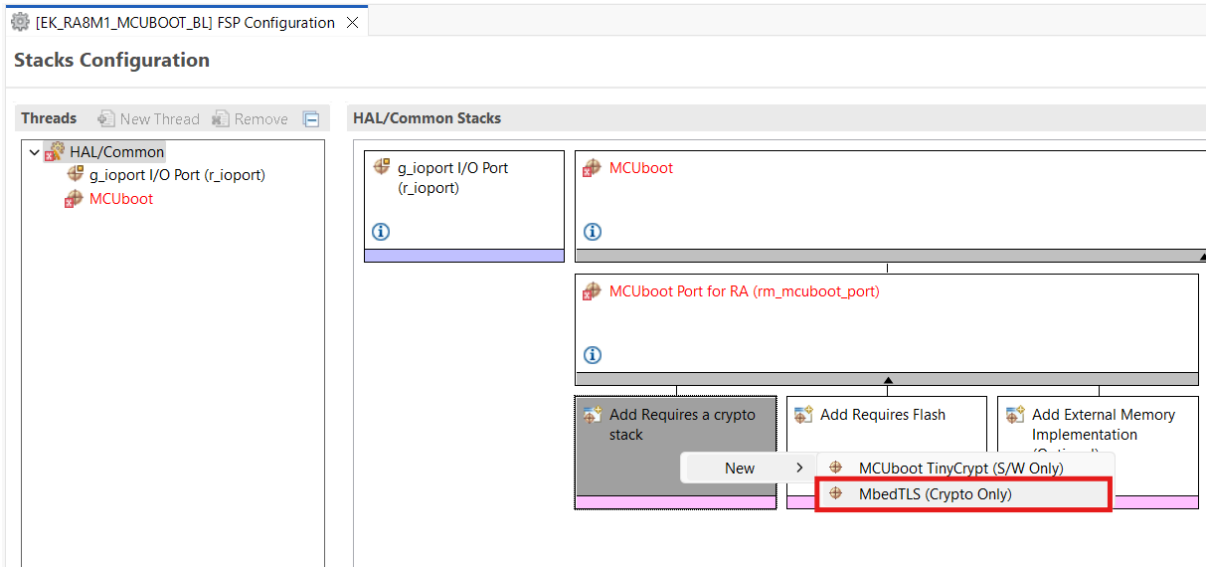


Figure 23 - Add the MbedTLS module to the bootloader project

6.3.11 Step 11: Configure MbedTLS Properties

In the Properties panel:

1. *Common → General*
   - *MBEDTLS_THREADING_ALT*: Select ***Undefine***
   - *MBEDTLS_THREADING_C*: Select ***Undefine***

- *MBEDTLS_MEMORY_BUFFER_ALLOC_C*: Select ***Define***

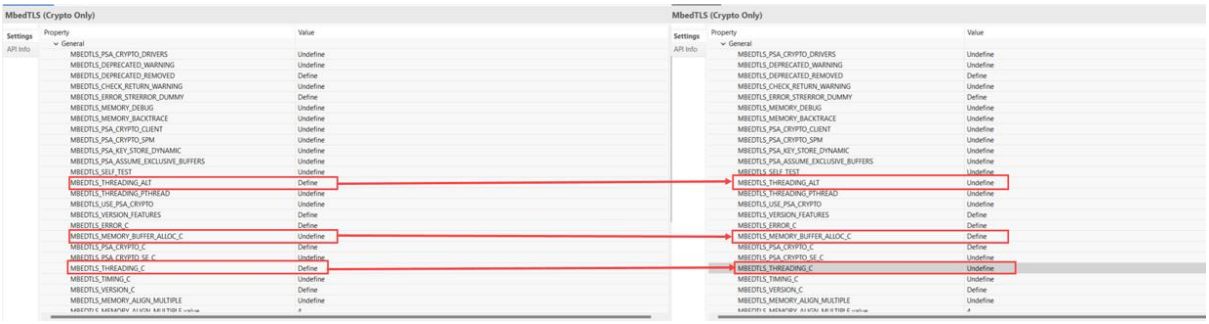


Figure 24 - Configure MbedTLS module properties

6.3.12 Step 12: Add Flash Module to the Bootloader

1. Select *Stacks* tab
2. Click on ***Add Requires Flash -> Flash (r_flash_hp)***

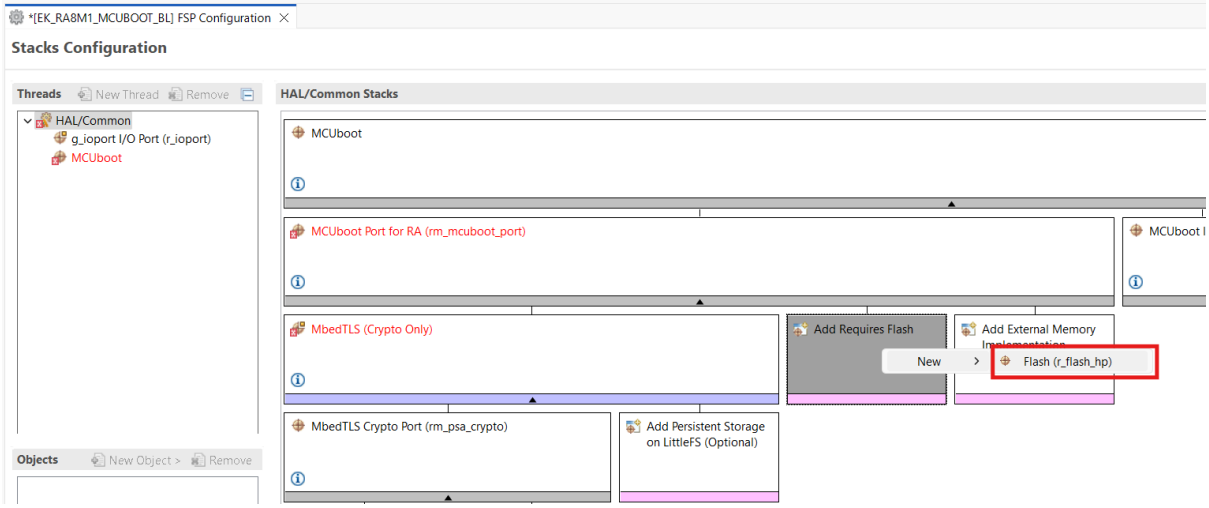


Figure 25 - Add the r_flash_hp module to the bootloader project

6.3.13 Step 13: Configure Flash Properties

In the Properties panel:

1. *Common*
   - *Code Flash Programming Enable*: Select ***Enabled***
   - *Data Flash Programming Enable*: Select ***Disabled***
2. *Module g_flash0 Flash(r_flash_hp)*
   - *Data Flash Background Operation*: Select ***Disabled***

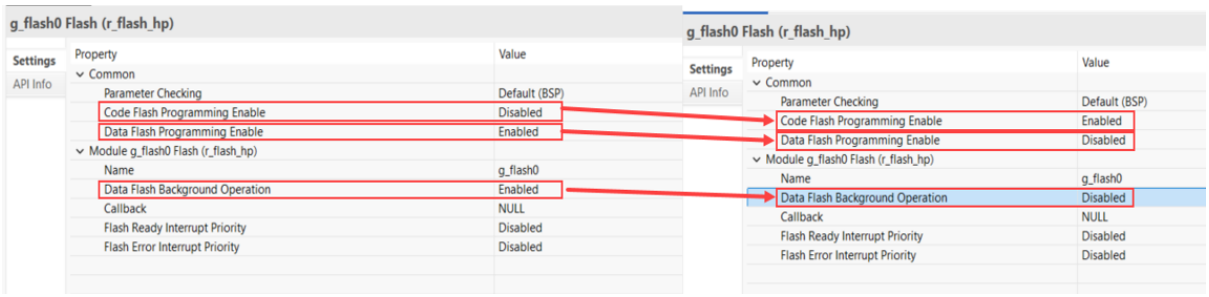


Figure 26 - Configure Flash module properties

6.3.14 Step 14: Configure BSP Properties

1. Select the *BSP* tab
2. Go to *Properties* tab
3. In *RA Common*
   - *Main stack size (bytes)*: Configure to ***0x2000***
   - *Heap size (bytes):* Configure to ***0x2000***
3. In *RA8M1 Device Options -> OFS Registers*
   - *DUALSEL*: Configure to ***Enabled*** and *Bank Mode* to ***Dual***
   - *FSBL:*

Parameter	Description	Value
Code Certificate → SACC0	Start address of code certificate for bootloader of lower bank. Used when BANKSWP[2:0]=111b in dual mode (primary image load success)	***Enabled*** *Addr: 0x02060000*
Code Certificate → SACC1	Start address of code certificate for bootloader of lower bank. Used when BANKSWP[2:0]=000b in dual mode (primary image load success.	***Enabled*** *Addr: 0x02060000*
FSBLCTRL0 → FSBLEN	FSBL Enable - activates First Stage Bootloader secure boot	***Enabled***
FSBLCTRL0 → FSBLSKIPSW	Skip FSBL upon a software reset (faster boot)	***Enabled***
FSBLCTRL0 → FSBLSKIPDS	Skip FSBL upon reset from deep software standby reset (faster boot)	***Enabled***
FSBLCTRL0 → FSBLCLK	FSBL clock frequency selection	***240 MHz*** *(default)*
FSBLCTRL1 → FSBLEXMDFSBLEN	FSBL Execution Mode - selects boot verification method (CRC vs Secure boot) and reporting (logging for debug purposes)	***Secure boot w/o report***
FSBLCTRL2 → PORTPN	Debug port pin selection for FSBL error reporting (dev only)	***PORTn07***
FSBLCTRL2 → PORTGN	Debug port group selection for FSBL error reporting (dev only)	***PORT1m***

IMPORTANT: SACC0/SACC1 addresses (*0x02060000*) are configured outside of flash memory areas designated for bootloader and both primary and secondary images. These registers must be reconfigured if the flash layout changes.

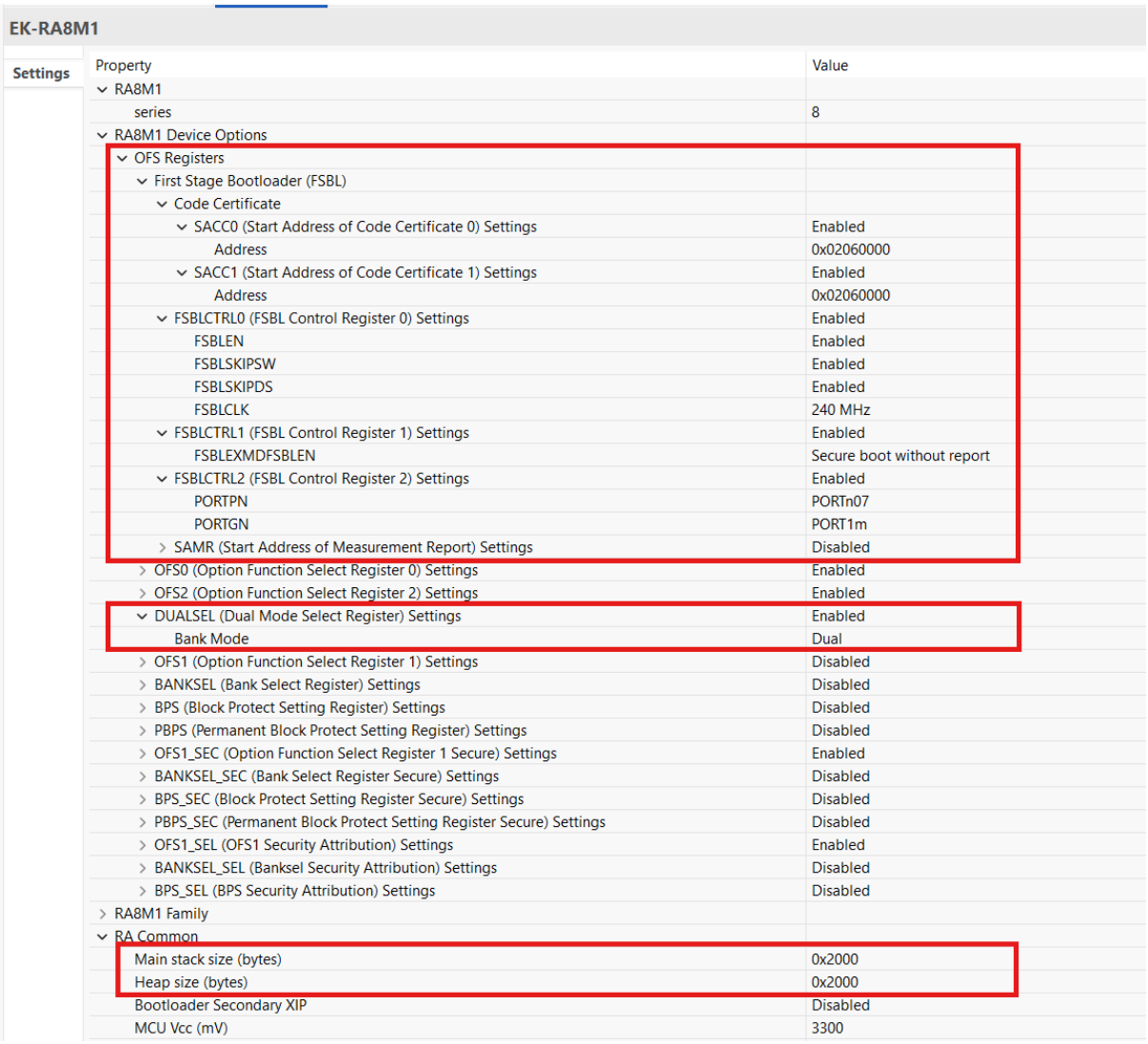


Figure 27 - Configure BSP properties

6.3.15 Step 15: Add example keys to the Bootloader

This is a demo key used by the bootloader to verify application image signature. It needs to be added here so that the project can build successfully, without the need to modify the structure of the MCUBoot project generated via FSP. **This key is replaced by the Application Signing Key (CUSTOMER Key Pair) in the production flow.**

1. Select *Stacks* tab
2. Click on ***Add Example keys -> MCUboot Example Keys (NOT FOR PRODUCTION)***

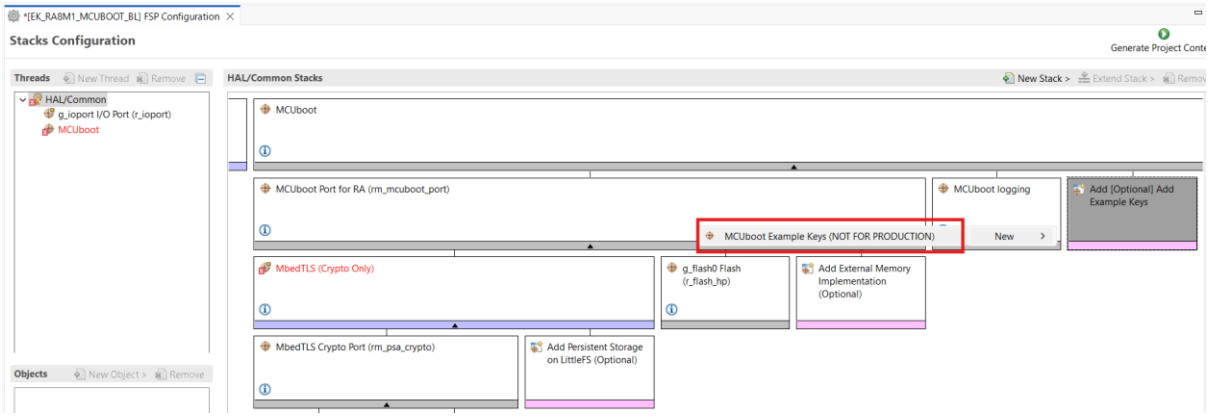


Figure 28 - Add the demo keys to the bootloader project

IMPORTANT: This configuration adds file ***keys.c*** to the *workspace/EK_RA8M1_MCUBOOT_BL/ra/mcu-tools/MCUboot/sim/mcuboot-sys/csupport* folder for compilation. The customer public key is then embedded directly in the bootloader as `root_pub_der[]` (see Figure 29) and stored at a specific address in flash memory.

For this example configuration the address is **0x02009114**, but this is subject to change depending on the bootloader code and configuration. To extract the correct address, open file ***EK_RA8M1_MCUBOOT_BL.map*** in *workspace/EK_RA8M1_MCUBOOT_BL/Debug* and locate the correct offset value for *root_pub_der* (Figure 30).

```
#define HAVE_KEYS
#ifndef MCUBOOT_SIGN_EC384
const unsigned char root_pub_der[] = {
    0x30, 0x59, 0x30, 0x13, 0x06, 0x07, 0x2a, 0x86,
    0x48, 0xce, 0x3d, 0x02, 0x01, 0x06, 0x08, 0x2a,
    0x86, 0x48, 0xce, 0x3d, 0x03, 0x01, 0x07, 0x03,
    0x42, 0x00, 0x04, 0x2a, 0xcb, 0x40, 0x3c, 0xe8,
    0xfe, 0xed, 0x5b, 0xa4, 0x49, 0x95, 0xa1, 0xa9,
    0x1d, 0xae, 0xe8, 0xdb, 0xbe, 0x19, 0x37, 0xcd,
    0x14, 0xfb, 0x2f, 0x24, 0x57, 0x37, 0xe5, 0x95,
    0x39, 0x88, 0xd9, 0x94, 0xb9, 0xd6, 0x5a, 0xeb,
    0xd7, 0xcd, 0xd5, 0x30, 0x8a, 0xd6, 0xfe, 0x48,
    0xb2, 0x4a, 0x6a, 0x81, 0x0e, 0xe5, 0xf0, 0x7d,
    0x8b, 0x68, 0x34, 0xcc, 0x3a, 0x6a, 0xfc, 0x53,
    0x8e, 0xfa, 0xc1, };
const unsigned int root_pub_der_len = 91;
```

Figure 29 - Demo public key data in code

```
2009108  2009108        c     4        ./ra_gen/pin_data.o:(.rodata.g_bsp_pin_cfg)
2009108  2009108        c     1                g_bsp_pin_cfg
2009114  2009114       5b     1        ./ra/mcu-tools/MCUboot/sim/mcuboot-sys/csupport/keys.o:(.rodata.root_pub_der)
2009114  2009114       5b     1                root_pub_der
2009170  2009170        4     4        ./ra/mcu-tools/MCUboot/sim/mcuboot-sys/csupport/keys.o:(.rodata.root_pub_der_len
2009170  2009170        4     1                root_pub_der_len
```

Figure 30 - Locate customer public key address in .map file

6.3.16 Step 16: Add MCUboot Initialisation Code

The next steps cover adding the MCUboot activation code and compiling the bootloader:

1. Open *hal_entry.c*
2. Open *Developer Assistance → HAL/Common → MCUboot → Quick Setup*

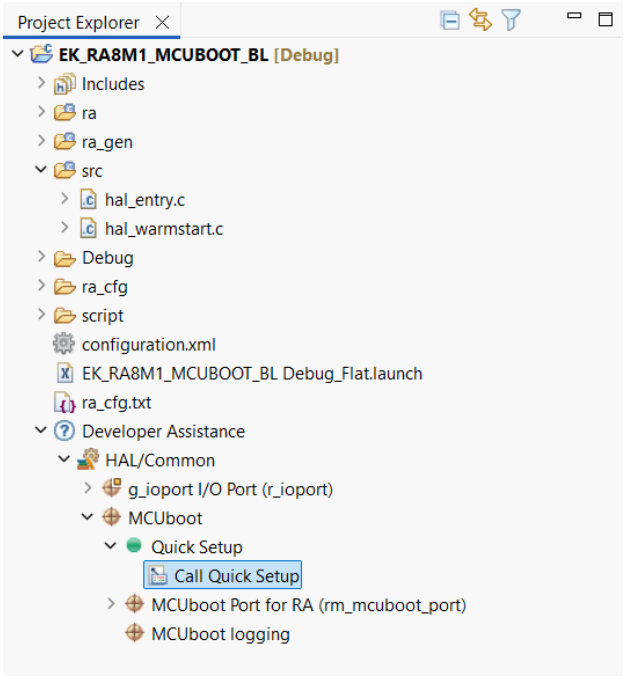


Figure 31 - Call Quick Setup

3. Drag ***Call Quick Setup*** to the top of the *hal_entry.c* file before the `hal_entry()` function call
4. Add a function call to `mcuboot_quick_setup();` at the top of the `hal_entry()` function

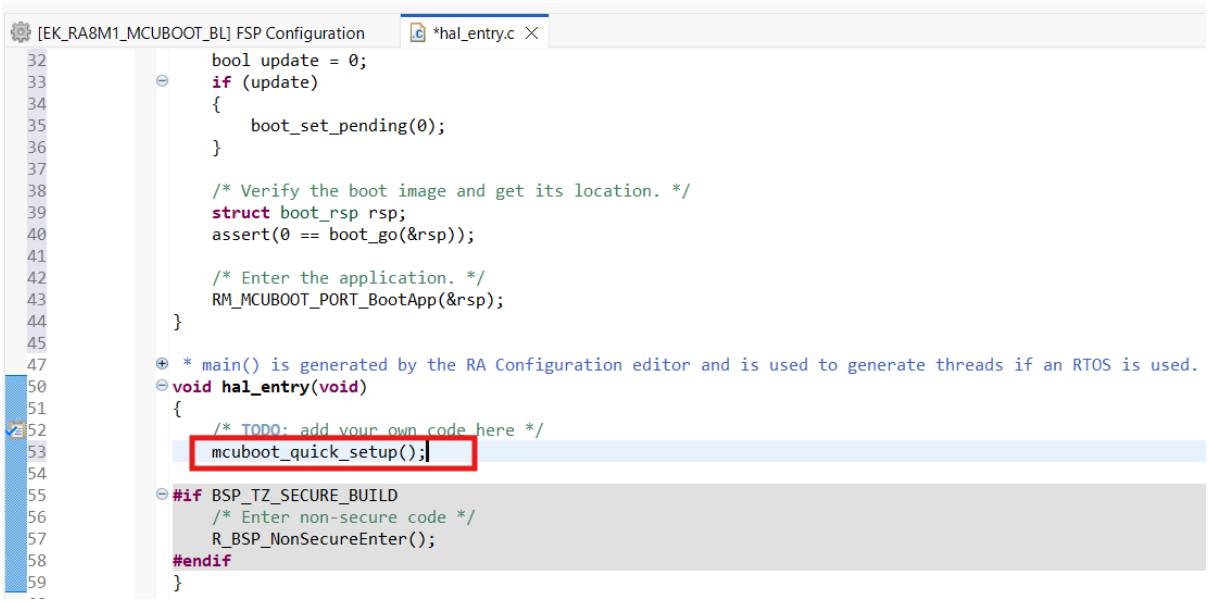


Figure 32 - Add MCUBoot Initialisation Code

5. Save the project (save the source code and the *configuration.xml* file), click ***Generate Project Content*** and then Build the project

Upon successful build, locate the generated files:

- *Debug/EK_RA8M1_MCUBOOT_BL.elf*
- *Debug/EK_RA8M1_MCUBOOT_BL.srec*
- *Debug/EK_RA8M1_MCUBOOT_BL.map*

6.4 Creating the Application Project

6.4.1 Step 1: Create New Project

It follows similar steps as the bootloader project:

1. In the menu bar, click *File → New → C/C++ Project*
2. Select ***Renesas RA C/C++ Project***
3. Name the project: e.g. ***EK_RA8M1_MCUBOOT_APP***
4. Select ***EK-RA8M1*** board
5. Choose the same project type as the bootloader (*Flat (Non-TrustZone) Project*)
6. Select *EK_RA8M1_MCUBOOT_BL* as preceding project

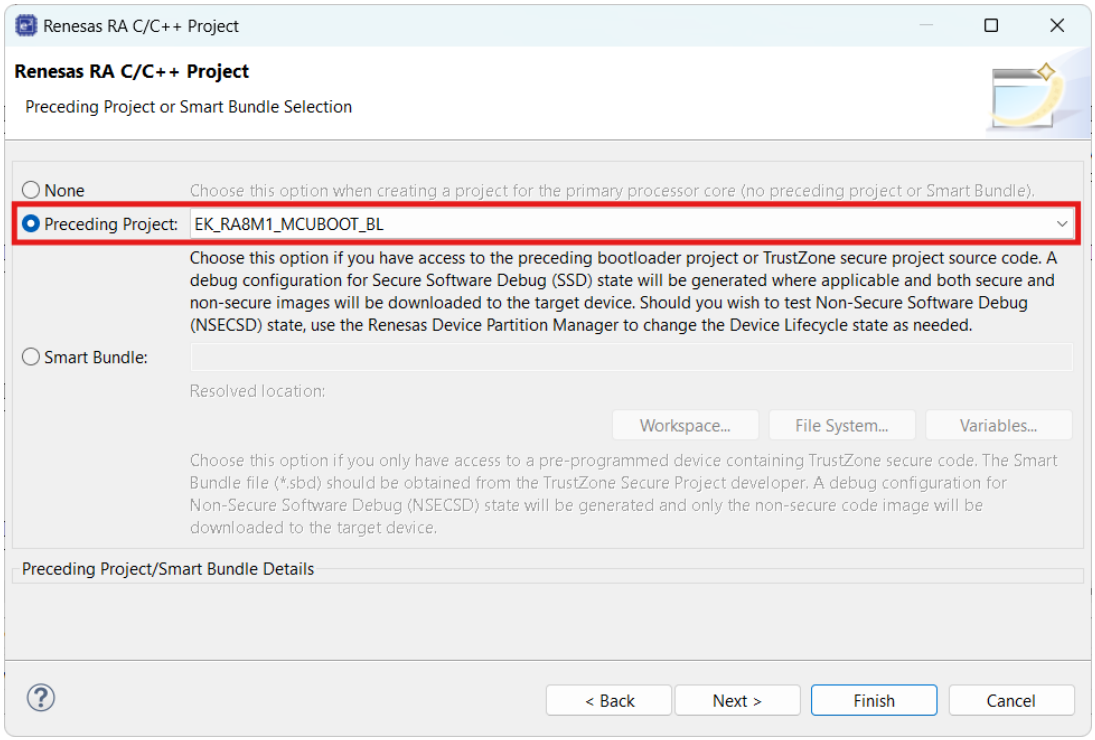


Figure 33 - Preceding Project

7. Set ***Executable*** as Build Artifact Selection and ***No RTOS*** as RTOS Selection
8. Select an appropriate template (***Bare Metal - Blinky***)
9. Click *Finish*

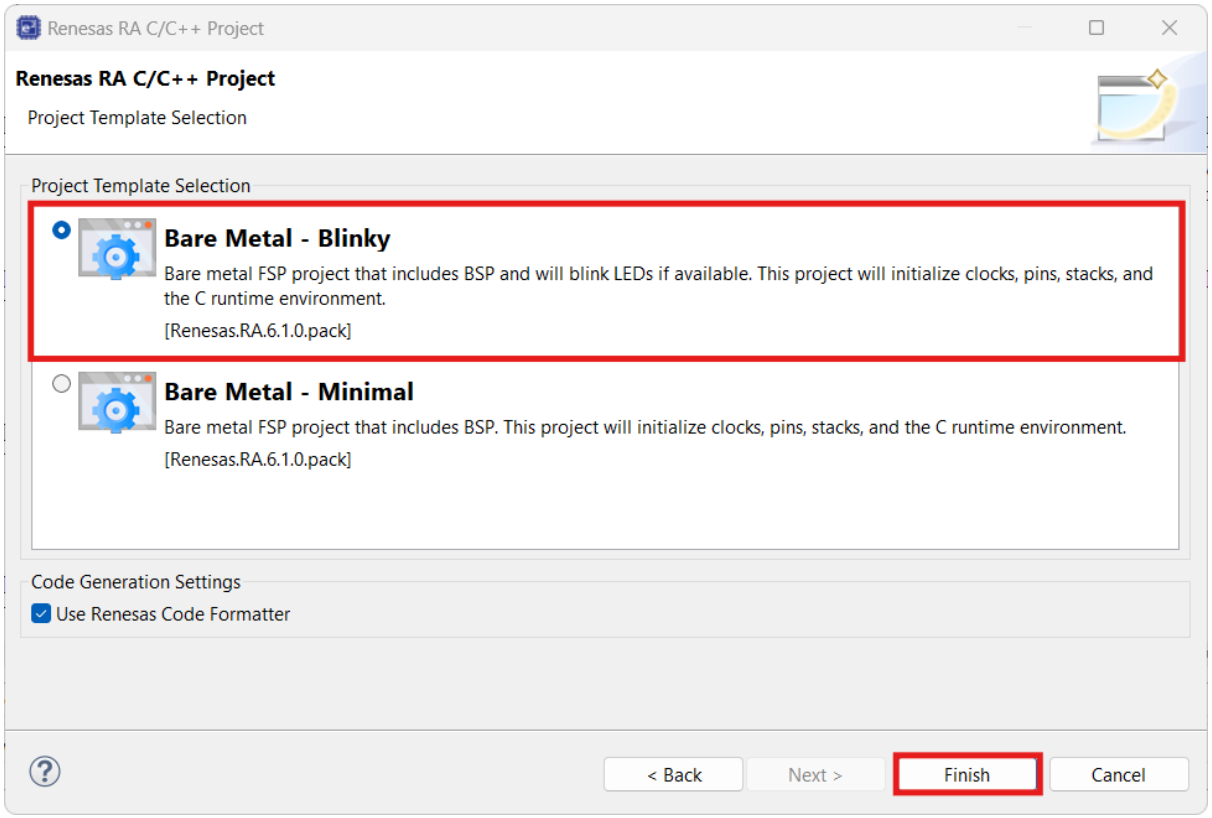


Figure 34 - Create a new application project

6.4.2 Step 2: Set Up Environment Variables

The application image must be signed to work with MCUboot. This step is only necessary to correctly build the application if the reader is interested in running it without going through the whole provisioning flow. Signing as part of the provisioning is taken care by the Python tool, as described in Solution Overview.

Configure environment variables:

1. Right-click the application project
2. Select *Properties*
3. Navigate to *C/C++ Build → Environment*
4. Click *Add…* to add each variable

MCUBOOT_IMAGE_VERSION

- **Value**: 1.0.0 (or desired version)
- **Description**: Semantic version of the application image

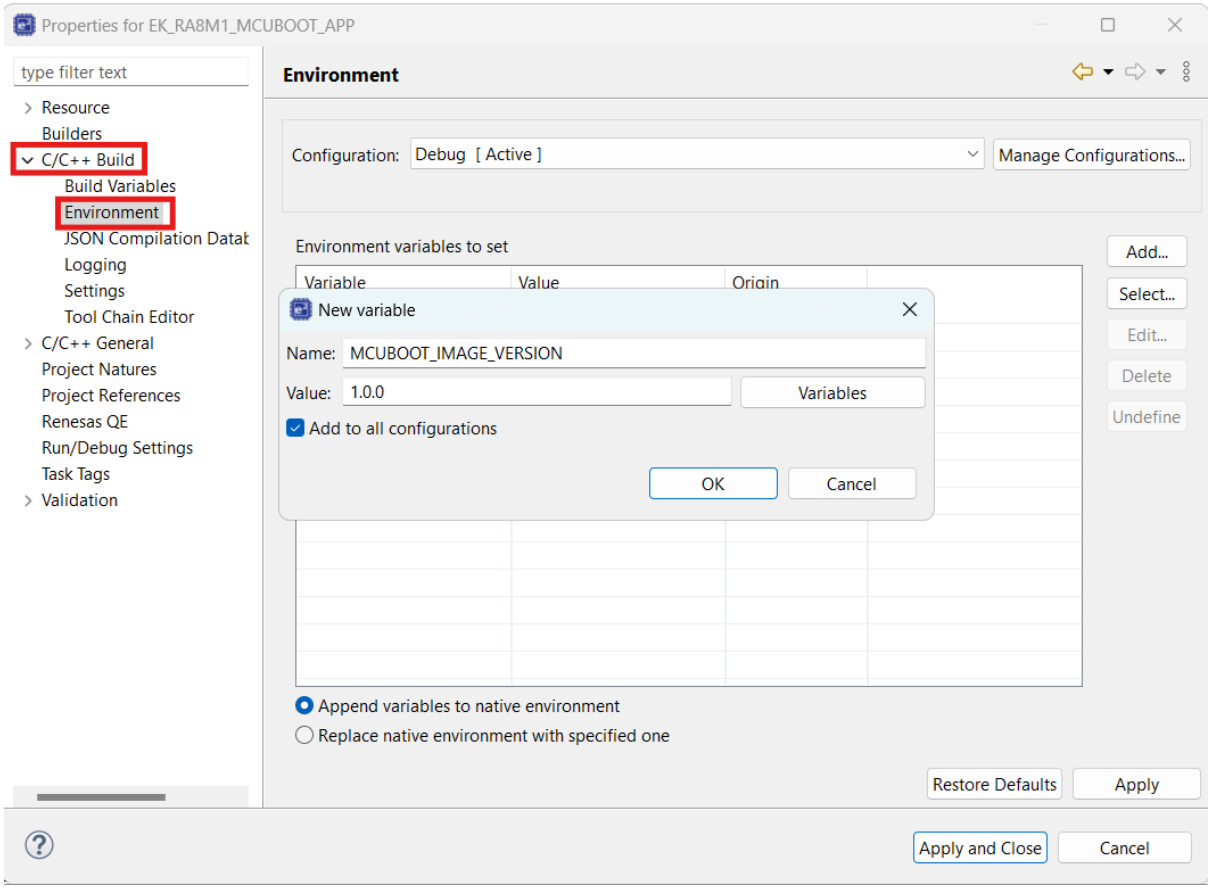


Figure 35 - MCUBOOT_IMAGE_VERSION

MCUBOOT_APP_BIN_CONVERTER (Optional)

- **Value**: Path to *objcopy* tool
- **Example**: *C:\Renesas\RA\e2studio_v2025-07_fsp_v6.1.0\toolchains\gcc_arm\13.2.rel1\arm-none-eabi\bin\objcopy.exe*
- **Description**: Tool to convert ELF to binary (usually auto-detected)

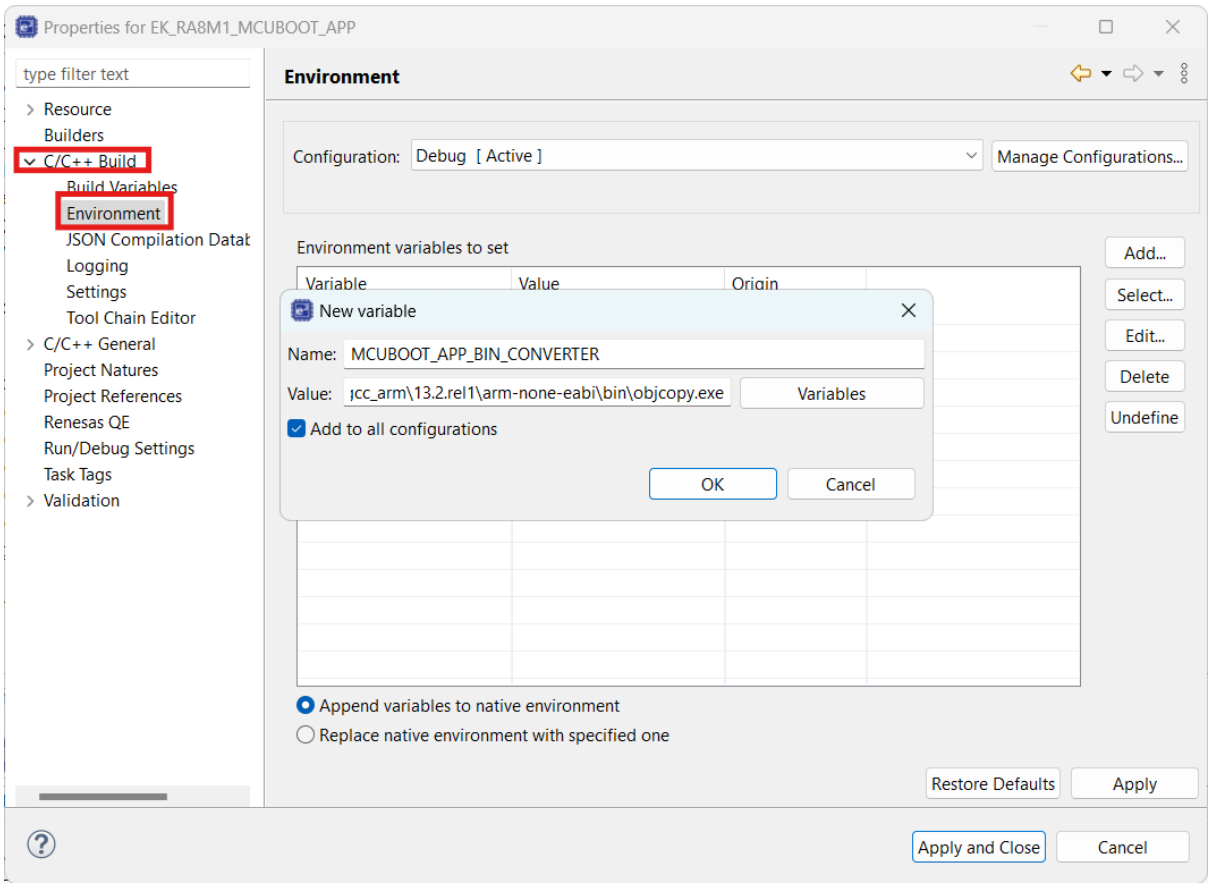


Figure 36 - MCUBOOT_APP_BIN_CONVERTER

Click **Apply** and **Close** after adding all variables.

6.4.3 Step 3: Build the Application Project

1. Ensure the bootloader project has been built successfully
2. Right-click the application project in *Project Explorer*
3. Select *Build Project*
4. Monitor the build output in the *Console*

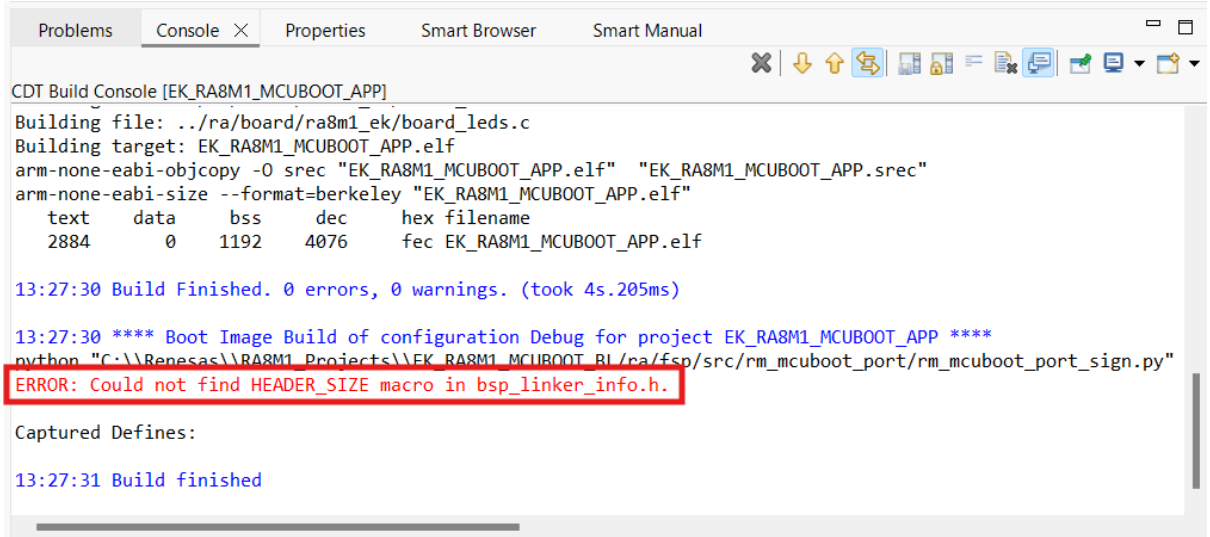


Figure 37 - Build output of application project

Code compilation is successful, but an error will pop-up indicating the application image signing has failed.

This is expected and is addressed by the solution project created in the next step.

6.5 Creating the Solution Project

6.5.1 Step 1: Create New Project

Follow similar steps to create a solution project:

1. Click *File → New → C/C++ Project*
2. Select ***Renesas FSP Solution Project (Advanced)***

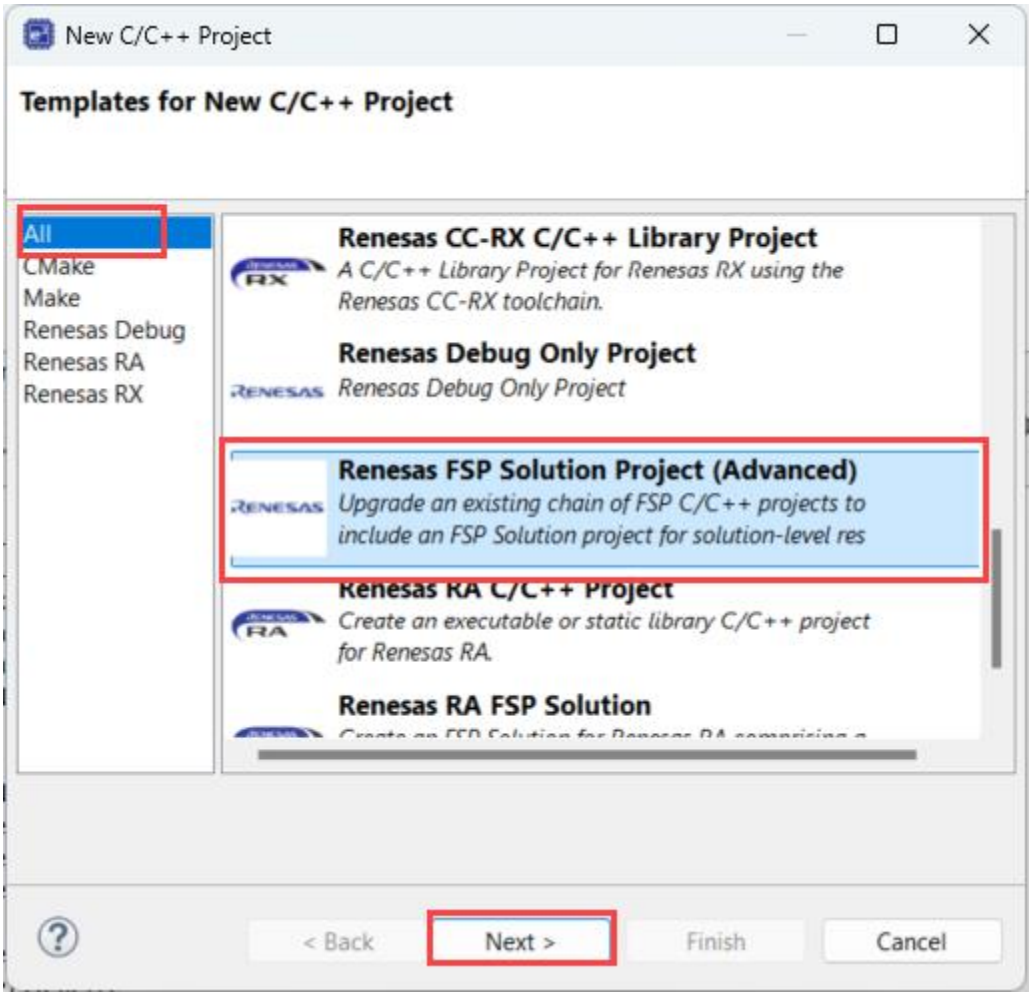


Figure 38 - Create new Solution Project

3. Name the project: e.g ***EK_RA8M1_MCUBOOT_SLN***
4. Select ***EK_RA8M1_MCUBOOT_APP*** as final project in the chain of projects
5. Click *Finish*

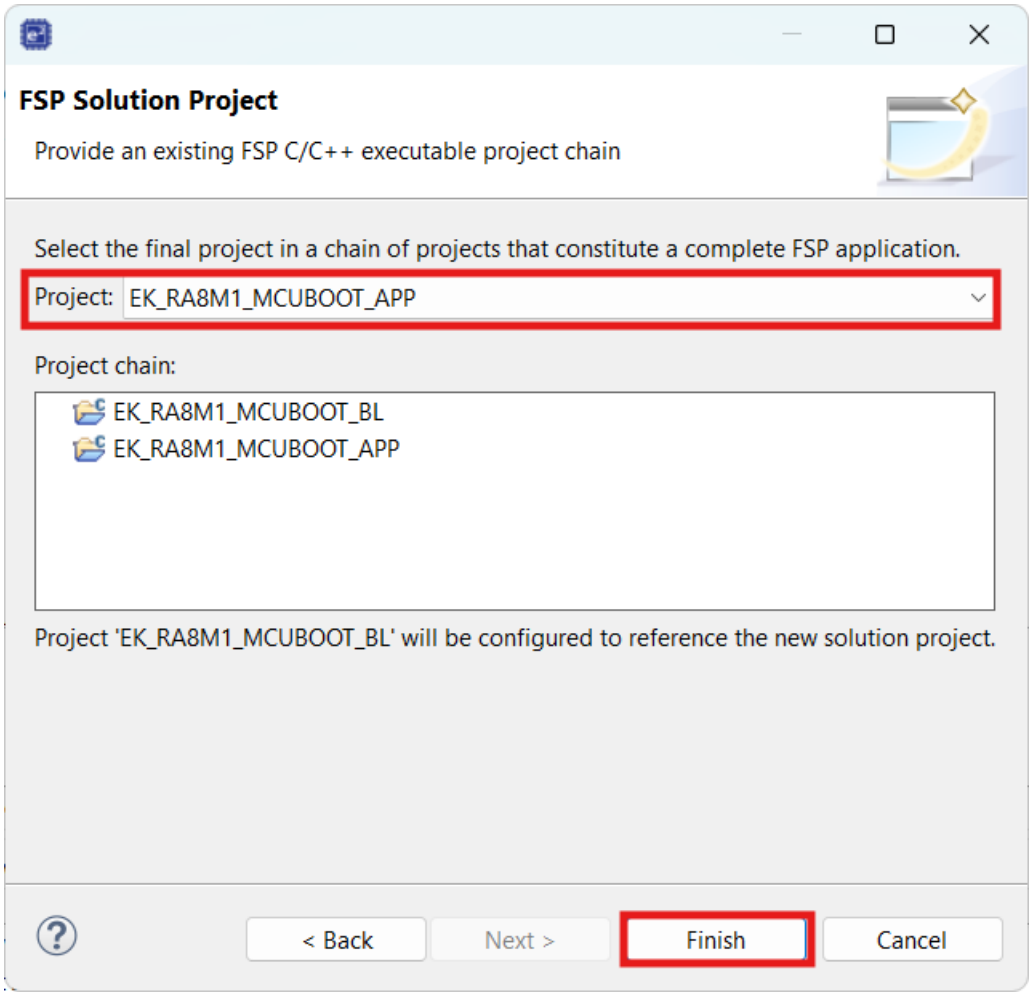


Figure 39 - Select final project in the chain

6.5.2 Step 2: Configure flash layout

1. Open *solution.xml*
2. Go to *Memories* tab
3. Select ***Flash*** and then click on ***Add Partition***
4. The following partitions must be defined:

Table 2 - Flash Partition Layout

Name	Start	Size
FLASH_CM85_B	0x02000000	0x1000
BL_0_P_H	0x02001000	0x200
FLASH_CM85_S	0x02001200	0x1FE00
BL_0_S_H	0x02030000	0x200
BL_1_S_I	0x02030200	0x1FE00

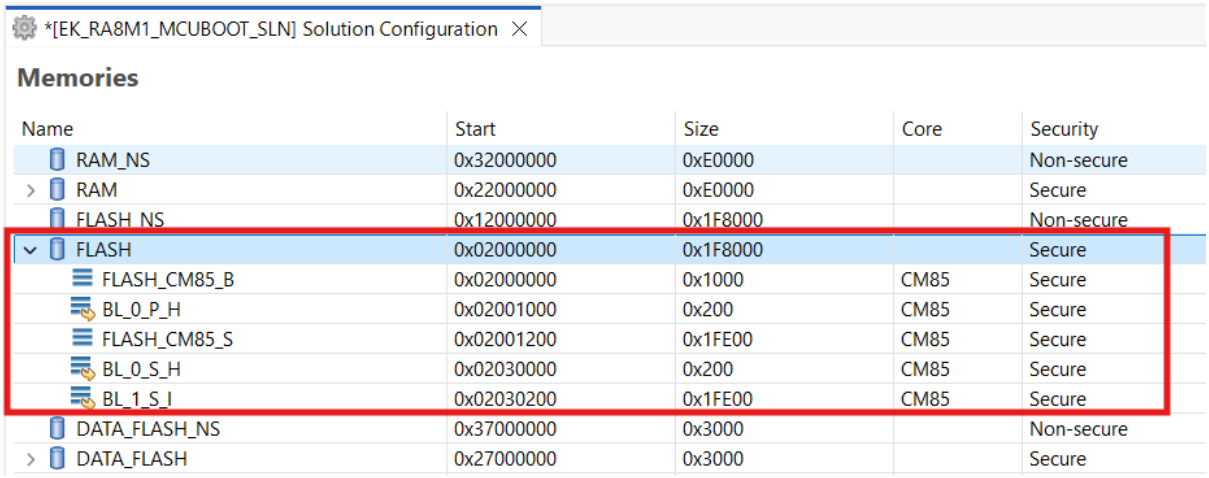


Figure 40 - Memories - Flash Partition

5. Build solution (both bootloader and application projects shall be built)

6.6 Conclusion

The relevant output files after creating the three projects (bootloader, application and solution) are:

- ***EK_RA8M1_MCUBOOT_BL.srec***
- ***EK_RA8M1_MCUBOOT_APP.srec***

These represent the MCUBoot bootloader and application binaries, which will be used by the provisioning tool.

7 Provisioning Implementation Guide

This chapter covers the implementation details necessary to develop the tool outlined in chapter 4.

It is assumed that the production keys (OEM_ROOT, OEM_BL, CUSTOMER) have already been created in the HSM and that MCUBoot and Application projects have been developed. If this is not the case, refer to chapters 5 and 6 for more details.

As previously mentioned, the tool covers all the operations that chapter 3 identified as phase 2 and phase 3.

The code included in this chapter consists of simplified snippets designed to illustrate key concepts and provide a foundation for development. Developers are encouraged to refine these examples by implementing robust structure, thorough error handling, and any additional functionality needed for production use. Refer to chapter 8 for more instructions and a fully working example.

7.1 Phase 2: Signing and Credential Preparation

This section contains guidelines and code snippets that enable the user to implement a secure way to generate all the artifacts necessary for enabling Secure Boot on the RA8 MCUs.

7.1.1 Wrap the OEM Root Public Key

Wrapping the OEM Root public key (OEM_ROOT_PK) requires:

- OEM_ROOT_PK (from the HSM)
- UFPK
- W-UFPK

The objective of this operation is a *.rkey* file that is injected into the device and unwrapped by the MCU's security engine. An rkey is structured as specified by Table 3.

Table 3 - rkey file structure

Field	Type	Size (bytes)	Description
Magic Code	Char[4]	4	"REK1"
Suite Version	Integer	4	Data format version Currently must be 1
Reserved	Byte[7]	7	0
Key Type	Byte	1	OEM_ROOT_PK = 0xFD DLM-AL2 = 0x01 DLM-AL1 = 0x02 DLM-RMA = 0x03
Encrypted Key Size	Integer	4	Size of Encrypted Key (= N bytes)
W-UFPK	Byte[36]	36	Value of the W-UFPK file sent from the Renesas DLM server The first 4 bytes are Shared Key Number The remaining 32 bytes are the WUFPK value
Initialization Vector	Byte[16]	16	Initialization vector value used to Wrap user key
Encrypted Key	Byte[N]	N	User key encrypted with UFPK value + MAC value

Data CRC	Byte[4]	4	Calculated CRC for all data except this CRC data. Initial Value = 0xFFFFFFFF Magic number = 0x04C11DB7

Within the .rkey is enclosed the OEM_ROOT_PK encrypted with the UFPK, according to a vendor-specific algorithm, documented in “Security Key Management Tool User Manual”.

Python code for wrapping the OEM_ROOT_PK and creating the .rkey can be found below:

```
# Wrap user_key with ufp_key
def wrap_user_key(user_key: bytes, ufp_key: bytes, iv: bytes) -> bytes:
    if len(ufp_key) != 32:
        raise ValueError("UFPK must be 32 bytes")
    if len(iv) != 16:
        raise ValueError("IV must be 16 bytes")
    if len(user_key) % 16 != 0:
        raise ValueError("User key must be multiple of 16 bytes")

    cbc_key = ufp_key[:16]
    mac_key = ufp_key[16:]
    mac = bytes(16)
    current_iv = iv
    encrypted_blocks = []

    for i in range(0, len(user_key), 16):
        block = user_key[i:i+16]

        # Update MAC
        mac = aes_ecb_encrypt(mac_key, bytes(a ^ b for a, b in zip(block, mac)))

        # Encrypt block
        ct = aes_ecb_encrypt(cbc_key, bytes(a ^ b for a, b in zip(block, current_iv)))
        encrypted_blocks.append(ct)

        # Update IV to ciphertext for next block
        current_iv = ct

    # Final block: XOR last MAC with last IV, encrypt with CBCKey
    final_block = aes_ecb_encrypt(cbc_key, bytes(a ^ b for a, b in zip(mac, current_iv)))

    return b"".join(encrypted_blocks) + final_block


# AES ECB helper
def aes_ecb_encrypt(key: bytes, block: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())
    encryptor = cipher.encryptor()
    return encryptor.update(block) + encryptor.finalize()
```

```
# Create the .rkey file
def create_rkey(
    user_key: bytes,
    ufp_key: bytes,
    w_ufpk: bytes,
    key_type: int,
    suite_version: int = 1
) -> tuple[str, bytes]:
    """
    Returns a tuple:
      - rkey_text: Base64-wrapped .rkey string
      - raw_binary: binary structure before Base64 encoding
    """
    if len(w_ufpk) != 36:
```

```
        raise ValueError("W-UFPK must be 36 bytes")

    # use a random IV
    iv = os.urandom(16)
    encrypted_key = wrap_user_key(user_key, ufp_key, iv)

    # Assemble binary structure
    binary = bytearray()
    binary += b"REK1"                    # Magic Code
    binary += struct.pack(">I", suite_version)  # Suite Version (big-endian)
    binary += b"\x00" * 7                # Reserved
    binary += struct.pack("B", key_type)  # Key Type
    binary += struct.pack(">I", len(encrypted_key))  # Encrypted Key Size
    binary += w_ufpk                      # Wrapped UFPK (36 bytes)
    binary += iv                          # IV (16 bytes)
    binary += encrypted_key               # Encrypted Key + final block

    # CRC32 over all preceding bytes
    crc = crc32_msb(binary)
    binary += struct.pack(">I", crc)

    # Base64 encode and wrap with header/footer
    b64 = base64.b64encode(binary).decode("ascii")
    lines = [b64[i:i+64] for i in range(0, len(b64), 64)]
    rkey_text = "-----BEGIN RENESAS KEY-----\n" + "\n".join(lines) + "\n-----END RENESAS KEY----
-"

    return rkey_text, bytes(binary)


# CRC32 (non-reflected, MSB-first)
def crc32_msb(data: bytes, poly=0x04C11DB7, init=0xFFFFFFFF) -> int:
    crc = init
    for b in data:
        crc ^= (b << 24)
        for _ in range(8):
            if crc & 0x80000000:
                crc = ((crc << 1) ^ poly) & 0xFFFFFFFF
            else:
                crc = (crc << 1) & 0xFFFFFFFF
    return crc
```

The same wrapping function can be used to wrap AL1, AL2 and RMA keys.

Importantly, the OEM_ROOT_PK used here is in its 64-bytes raw format (Qx||Qy) and needs to be exported from the HSM.

It is encouraged to abstract the HSM layer for a straightforward integration of different HSM services. The following code defines an abstract class `HSMClient` that can function as an interface for all HSM implementations.

```
class HSMClient(ABC):
    """Abstract base class for HSM clients."""

    @abstractmethod
    def connect(self) -> None:
        """Establish connection to HSM."""
        pass

    @abstractmethod
    def disconnect(self) -> None:
        """Close HSM connection."""
        pass

    @abstractmethod
    def sign_digest(self, key_handle: str, data: bytes) -> bytes:
        """
        Sign data with specified key.
```

```
        Args:
            key_handle: Key identifier (ARN for AWS KMS)
            data: Data to sign

        Returns:
            DER-encoded ECDSA signature
        """
        pass

    @abstractmethod
    def get_public_key(self, key_handle: str) -> bytes:
        """
        Export public key from HSM.

        Args:
            key_handle: Key identifier

        Returns:
            Public key in SubjectPublicKeyInfo DER format
        """
        pass

    @abstractmethod
    def verify_signature(
        self,
        public_key: bytes,
        data: bytes,
        signature: bytes
    ) -> bool:
        """Verify signature using public key."""
        pass
```

According to the above, exporting the OEM_ROOT_PK from the HSM so that it can be wrapped looks like this.

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import load_pem_public_key, load_der_public_key
from cryptography.hazmat.backends import default_backend

def extract_raw_public_key_qxqy(key_bytes: bytes) -> bytes:
    """
    Extract raw public key as Qx||Qy (64 bytes) from PEM or DER format.
    Format: Qx (32 bytes big-endian) || Qy (32 bytes big-endian) = 64 bytes total
    Args:
        key_bytes: Public key in PEM or DER format
    Returns:
        Raw public key as Qx||Qy (64 bytes)
    """
    try:
        try:
            pk = load_pem_public_key(key_bytes, backend=default_backend())
        except Exception:
            try:
                pk = load_der_public_key(key_bytes, backend=default_backend())
            except Exception as der_e:
                # raise error

        # Get raw uncompressed point (65 bytes: 0x04 || Qx(32) || Qy(32))
        raw = pk.public_bytes(
            encoding=serialization.Encoding.X962,
            format=serialization.PublicFormat.UncompressedPoint
        )
        # Remove 0x04 prefix to get Qx||Qy (64 bytes)
        if len(raw) == 65 and raw[0] == 0x04:
            qxqy = raw[1:]  # Return Qx||Qy (64 bytes)
        elif len(raw) == 64:
            qxqy = raw  # Already Qx||Qy
```

```
        else:
            # raise error

        if len(qxqy) != 64:
            # raise error

        return qxqy
    except Exception as e:
        # raise error

oem_root_pk_der = hsm_client.get_public_key(oem_root_key_id)
raw_oem_root_pk = extract_raw_public_key_qxqy(oem_root_pk_der)
```

As mentioned, the guide uses AWS KMS as the HSM of choice and the below provides an implementation of the abstract class HSMClient for AWS KMS.

```
class AWSKMSClient(HSMClient):

    def __init__(self, config: dict):
        super().__init__(config)
        self.aws_region = config.get("aws_region", "eu-central-1")
        self.key_prefix = config.get("key_prefix", "RA8M1_")

        try:
            # Initialize KMS client
            self.kms_client = boto3.client(
                "kms",
                region_name=self.aws_region,
                aws_access_key_id=config["aws_access_key_id"],
                aws_secret_access_key=config["aws_secret_access_key"],
            )
        except Exception as e:
            raise HSMError(f"Failed to initialize AWS KMS client: {str(e)}") from e

    def connect(self) -> None:
        try:
            sts_client = boto3.client(
                "sts",
                region_name=self.aws_region,
                aws_access_key_id=self.config["aws_access_key_id"],
                aws_secret_access_key=self.config["aws_secret_access_key"],
            )
            sts_client.get_caller_identity()
            self.initialized = True
        except Exception as e:
            raise HSMError(f"Unexpected error connecting to AWS KMS: {str(e)}") from e

    def disconnect(self) -> None:
        self.initialized = False

    def get_public_key(self, key_handle: str) -> bytes:
        if not self.initialized:
            raise HSMError("AWS KMS not connected")
        try:
            # Get public key from AWS KMS
            response = self.kms_client.get_public_key(KeyId=key_handle)
            public_key_der = response["PublicKey"]
            return public_key_der
        except Exception as e:
            raise HSMError(f"Unexpected error retrieving public key: {str(e)}") from e

    def sign_digest(self, key_handle: str, message_digest: bytes) -> bytes:
        if not self.initialized:
            raise HSMError("AWS KMS not connected")
        try:
            key_info = self.kms_client.describe_key(KeyId=key_handle)
            key_spec = key_info["KeyMetadata"]["KeySpec"]
            algorithm_map = {
```

```
                "ECC_NIST_P256": "ECDSA_SHA_256",
                "ECC_NIST_P384": "ECDSA_SHA_384",
                "ECC_NIST_P521": "ECDSA_SHA_512",
            }
            if key_spec not in algorithm_map:
                raise HSMError(f"Unsupported key spec for signing: {key_spec}")
            signing_algorithm = algorithm_map[key_spec]
            response = self.kms_client.sign(
                KeyId=key_handle,
                Message=message_digest,
                MessageType="DIGEST",
                SigningAlgorithm=signing_algorithm,
            )
            signature = response["Signature"]
            return signature
        except Exception as e:
            raise HSMError(f"Unexpected error signing digest: {str(e)}") from e

    def verify_signature(
        self, public_key: bytes, data: bytes, signature: bytes, key_handle: Optional[str] = None
    ) -> bool:
        if key_handle and self.initialized:
            try:
                key_info = self.kms_client.describe_key(KeyId=key_handle)
                key_spec = key_info["KeyMetadata"]["KeySpec"]
                algorithm_map = {
                    "ECC_NIST_P256": ("ECDSA_SHA_256", hashes.SHA256()),
                    "ECC_NIST_P384": ("ECDSA_SHA_384", hashes.SHA384()),
                    "ECC_NIST_P521": ("ECDSA_SHA_512", hashes.SHA512()),
                }
                if key_spec not in algorithm_map:
                    raise HSMError(f"Unsupported key spec for verification: {key_spec}")

                signing_algorithm, hash_algorithm = algorithm_map[key_spec]
                digest = hashes.Hash(hash_algorithm, backend=default_backend())
                digest.update(data)
                message_digest = digest.finalize()
                response = self.kms_client.verify(
                    KeyId=key_handle,
                    Message=message_digest,
                    MessageType="DIGEST",
                    Signature=signature,
                    SigningAlgorithm=signing_algorithm,
                )
                return response["SignatureValid"]
            except ClientError as e:
                return False
        try:
            pub_key = serialization.load_der_public_key(public_key, backend=default_backend())
            if not isinstance(pub_key, ec.EllipticCurvePublicKey):
                raise HSMError("Unsupported key type for verification")
            curve = pub_key.curve
            if isinstance(curve, ec.SECP256R1):
                hash_alg = hashes.SHA256()
            elif isinstance(curve, ec.SECP384R1):
                hash_alg = hashes.SHA384()
            elif isinstance(curve, ec.SECP521R1):
                hash_alg = hashes.SHA512()
            else:
                raise HSMError(f"Unsupported curve for verification: {curve}")
            digest = hashes.Hash(hash_alg, backend=default_backend())
            digest.update(data)
            message_digest = digest.finalize()
            pub_key.verify(signature, message_digest, ec.ECDSA(hash_alg))
            return True
        except Exception as e:
            return False
```

7.1.2 Create and Sign the OEM_BL and Application

This step takes care of signing the Application using the relevant CUSTOMER key from the HSM and MCUBoot's imgtool. An additional required operation is to update the OEM_BL (MCUBoot) so that the public portion of that same signing key (CUSTOMER_PK) is injected at the right offset.

Requirements for this step are:

- imgtool (Python)
- CUSTOMER keys (in the HSM)
- Application srec or bin file
- MCUBoot srec file

At the time of writing, imgtool does not natively support signing operations using keys stored in an HSM. However, given it is an open-source project, its functionality can be extended or modified to support alternative signing backends.

The example provided in this document does not modify imgtool itself. Instead, it demonstrates how an application image can be produced while ensuring that the actual cryptographic signing operation is performed by an HSM. Imgtool is used to generate the image structure and the hash to be signed, while a temporary private key is used solely to satisfy imgtool's signing workflow. The resulting hash is then signed by the HSM, and the image is updated to replace the placeholder signature with the HSM-generated signature. At no point is the production private key exposed outside the HSM.

```
# MCUBoot TLV types
TLV_KEYHASH  = 0x01
TLV_SHA256   = 0x10
TLV_ECDSA256 = 0x22

# RA8 / MCUBoot specific configuration values
header_size  = 0x200
align        = 128
max_align    = 128
slot_size    = 0x20000
max_sectors  = 4
version      = "1.0.0"

def sign_mcuboot_image_with_hsm(
    input_bin: Path,
    output_bin: Path,
    app_key_id: str,
    hsm_client
):
    with tempfile.TemporaryDirectory() as tmpdir:
        tmpdir = Path(tmpdir)

        # 1. Generate a temporary MCUBoot image with a placeholder signature
        tmp_key = tmpdir / "tmp_priv.pem"
        tmp_signed = tmpdir / "image_tmp.bin"

        subprocess.run([
            "imgtool", "sign",
            "--key", str(tmp_key),
            "--header-size", hex(header_size),
            "--pad-sig",
            "--align", str(align),
            "--max-align", str(max_align),
            "--slot-size", hex(slot_size),
            "--max-sectors", str(max_sectors),
            "--version", version,
            "--pad-header",
```

```
            "--pad",
            "--confirm",
            str(input_bin),
            str(tmp_signed),
        ], check=True)

        image = bytearray(tmp_signed.read_bytes())

        # 2. Locate TLV entries: SHA256, KEYHASH and ECDSA signature
        img_size = struct.unpack("<I", image[12:16])[0]
        tlv_start = header_size + img_size

        offset = tlv_start + 4
        image_hash = None
        keyhash_offset = None
        keyhash_len = None
        sig_offset = None
        sig_len = None

        while offset < len(image) - 4:
            tlv_type, tlv_len = struct.unpack("<HH", image[offset:offset + 4])

            if tlv_type == TLV_SHA256:
                image_hash = image[offset + 4:offset + 4 + tlv_len]

            elif tlv_type == TLV_KEYHASH:
                keyhash_offset = offset + 4
                keyhash_len = tlv_len

            elif tlv_type == TLV_ECDSA256:
                sig_offset = offset + 4
                sig_len = tlv_len

            offset += 4 + tlv_len

        # 3. Replace KEYHASH with the hash of the HSM public key
        app_key_der = hsm_client.get_public_key(app_key_id)
        app_key_hash = hashlib.sha256(app_key_der).digest()
        image[keyhash_offset:keyhash_offset + keyhash_len] = app_key_hash

        # 4. Sign the image hash using the HSM
        hsm_signature = hsm_client.sign_digest(app_key_id, image_hash)

        # 5. Replace placeholder signature with the HSM-generated signature
        image[sig_offset:sig_offset + sig_len] = hsm_signature.ljust(sig_len, b"\x00")

        # 6. Write final image
        output_bin.write_bytes(image[:slot_size])
```

To improve readability and focus on the core concept of signing an MCUBoot image using an HSM, this code omits a few important details. In a complete implementation, developers would need to include input validation, bounds checking, TLV header updates, error handling, hash verification, and other safeguards to ensure the integrity of the final image. The snippet is intended purely as a guideline to illustrate the workflow and key steps, not as a complete, copy-paste solution. Refer to Chapter 8 for a complete implementation.

A more robust approach is to extend imgtool with PKCS#11 support, allowing signing operations to be performed directly by the HSM. PKCS#11 is a widely adopted standard across HSM vendors and provides a consistent, secure interface for key usage without exposing private key material. Readers are encouraged to implement this integration themselves or use an existing imgtool variant that supports PKCS#11-based signing, depending on their production requirements and tooling preferences.

As discussed in 6.3.15, MCUBoot embeds the public key needed to verify the application at a specific offset, which can be extracted from the MCUBoot .map file if unknown. Furthermore,

the key is embedded in its full DER format (91 bytes). The code below shows how CUSTOMER_PK can be embedded into the OEM_BL srec file.

```
def calculate_checksum(hex_data) -> int:
    """Calculate SREC checksum (one's complement of sum of bytes)"""
    total = 0
    for i in range(0, len(hex_data), 2):
        total += int(hex_data[i:i+2], 16)
    return (~total) & 0xFF

def inject_customer_key(customer_key_id, hsm_client, start_addr, oem_bl_srec):
    customer_key_der = hsm_client.get_public_key(customer_key_id)
    key_end = start_addr + len(customer_key_der)
    output_lines = []

    # Read all lines
    with open(oem_bl_srec, 'r') as f:
        lines = f.readlines()

    for line in lines:
        line = line.strip()

        # Non-S3 records - keep as-is
        if not line or not line.startswith('S3'):
            output_lines.append(line)
            continue

        # Parse S3 record
        count = int(line[2:4], 16)
        addr = int(line[4:12], 16)
        data_len = count - 5
        record_end = addr + data_len

        # Check if this record overlaps with customer key zone
        overlaps = not (record_end <= start_addr or addr >= key_end)

        if not overlaps:
            # No overlap - keep original line unchanged
            output_lines.append(line)
            continue

        # This record contains customer key zone - inject key bytes
        data_hex = line[12:12 + data_len * 2]
        data = bytearray.fromhex(data_hex)

        # Calculate overlap range within this record
        inject_start = max(0, start_addr - addr)
        inject_end = min(data_len, key_end - addr)
        key_offset = max(0, addr - start_addr)

        # Inject key bytes
        for i in range(inject_start, inject_end):
            data[i] = customer_key_der[key_offset + (i - inject_start)]

        # Rebuild S3 record with new checksum
        record_data = f"{count:02X}{addr:08X}{data.hex().upper()}"
        checksum = calculate_checksum(record_data)
        new_line = f"S3{record_data}{checksum:02X}"
        output_lines.append(new_line)

    # Write back
    with open(oem_bl_srec, 'w') as f:
        for line in output_lines:
            f.write(line + "\n")

    # Regenerate S5 record count
    data_record_count = sum(1 for line in output_lines if line.startswith(('S1', 'S2', 'S3')))

    # Rebuild file with correct S5
```

```
    final_lines = []
    for line in output_lines:
        if line.startswith('S5'):
            # Replace old S5 with correct count
            s5_data = f"S503{data_record_count:04X}"
            s5_checksum = calculate_checksum(s5_data[2:])
            final_lines.append(f"{s5_data}{s5_checksum:02X}")
        else:
            final_lines.append(line)

    # Write final version
    with open(oem_bl_srec, 'w') as f:
        for line in final_lines:
            f.write(line + "\n")
```

Finally, the bootloader and application binaries must be combined into a single image, with each component placed at its correct flash offset. Rather than re-implementing an S-record parser and generator, the code relies on a well-established external tool such as *srec_cat*, which is widely used, tested, and maintained.

The code below shows an example wrapper class around *srec_cat*, illustrating how such a tool can be integrated into the workflow. In addition to invoking *srec_cat* for format conversion and image merging, the class includes helper logic for segment extraction, overlap detection, OSM record handling, and basic validation of address ranges. This allows bootloader and application images to be safely combined while preserving critical records such as execution start addresses and option-setting memory.

For clarity, the example intentionally omits documentation, exhaustive validation, and robust error handling. These aspects, along with a complete implementation, are covered in Chapter 8. The purpose here is to demonstrate the structure and responsibilities of an image-combination stage within a provisioning tool.

```
@dataclass
class SrecSegment:
    start_addr: int
    end_addr: int
    size: int
    source_file: str

class SrecCombiner:

    def __init__(self, srec_cat: Path):
        self.srec_cat = srec_cat

    def bin_to_srec(self, binary_file: Path, output_srec: Path, offset: int, exec_start_addr:
Optional[int] = None) -> Path:
        exec_start_addr = exec_start_addr or offset
        cmd = [
            str(self.srec_cat),
            str(binary_file),
            "-binary",
            "-offset", f"0x{offset:X}",
            "-execution-start-address", f"0x{exec_start_addr:X}",
            "-o", str(output_srec)
        ]
        subprocess.run(cmd, check=True)
        return output_srec

    def combine_srec_files(self, srec_files: List[Path], output_file: Path, check_overlap: bool =
True) -> Path:
        if check_overlap:
            overlaps = self._check_overlaps(srec_files)
            if overlaps:
                # handle overlaps
```

```
                pass
        cmd = [str(self.srec_cat)] + [str(f) for f in srec_files] + ["-o", str(output_file)]
        subprocess.run(cmd, check=True)
        return output_file

    def _check_overlaps(self, srec_files: List[Path]) -> List:
        segments = []
        for f in srec_files:
            segments.extend(self._extract_segments(f))
        overlaps = []
        for i, seg1 in enumerate(segments):
            for seg2 in segments[i+1:]:
                if not (seg1.end_addr <= seg2.start_addr or seg2.end_addr <= seg1.start_addr):
                    overlaps.append((seg1, seg2))
        return overlaps

    def _extract_segments(self, srec_file: Path) -> List[SrecSegment]:
        ranges = []
        for line in srec_file.read_text().splitlines():
            if not line.startswith("S3"):
                continue
            addr = int(line[4:12], 16)
            count = int(line[2:4], 16)
            ranges.append((addr, addr + count - 5))

        if not ranges:
            return []

        ranges.sort(key=lambda x: x[0])
        GAP_THRESHOLD = 0x10000
        segments = []
        start, end = ranges[0]
        for s, e in ranges[1:]:
            if s <= end + GAP_THRESHOLD:
                end = max(end, e)
            else:
                segments.append(SrecSegment(start, end, end-start, srec_file.name))
                start, end = s, e
        segments.append(SrecSegment(start, end, end-start, srec_file.name))
        return segments

    def extract_osm_records(self, bootloader_srec: Path) -> List[str]:
        osm_records = []
        for line in bootloader_srec.read_text().splitlines():
            if line.startswith("S3") and (line[4:9].upper() in ("0300A", "27030")):
                osm_records.append(line)
        return osm_records

    def add_osm_records(self, combined_srec: Path, osm_records: List[str], bootloader_srec:
Optional[Path] = None):
        lines = [l for l in combined_srec.read_text().splitlines() if not l.startswith(("S5",
"S7"))]
        lines.extend(osm_records)

        # Simplified S5 record (data record count)
        data_count = sum(1 for l in lines if l.startswith(("S1", "S2", "S3")))
        s5_data = f'03{data_count:04X}'
        s5_checksum = (~sum(bytes.fromhex(s5_data)) & 0xFF)
        new_s5 = f'S5{s5_data}{s5_checksum:02X}'

        # S7 reset vector
        new_s7 = self._extract_reset_vector(bootloader_srec) if bootloader_srec else
'S70502000000F8'

        combined_srec.write_text("\n".join(lines + [new_s5, new_s7]) + "\n")

    def _extract_reset_vector(self, bootloader_srec: Path) -> str:
        memory = {}
        for line in bootloader_srec.read_text().splitlines():
```

```
            if line.startswith("S3"):
                addr = int(line[4:12], 16)
                count = int(line[2:4], 16) - 5
                data_bytes = bytes.fromhex(line[12:12 + count*2])
                for i, b in enumerate(data_bytes):
                    memory[addr + i] = b
        if 0x02000004 in memory:
            import struct
            rh = struct.unpack('<I', bytes([memory[0x02000004 + i] for i in range(4)]))[0]
            s7_data = f'05{rh:08X}'
            checksum = (~sum(bytes.fromhex(s7_data)) & 0xFF)
            return f'S7{s7_data}{checksum:02X}'
        return 'S70502000000F8'
```

7.1.3 Generate Key and Code Certificates

This step completes the chain of trust, linking the FSBL to MCUBoot. This is achieved via the two binary files Key Certificate and Code Certificate, which structures are reported below.

Table 4 - Key Certificate structure

Field		Size (bytes)	Description
Header	Magic	4	Set to 0x6B657963
	Manifest Version	4	Set to 0x00010000
	Flags	4	Reserved (0x00000000)
	Reserved	20	Reserved (All 0)
TLV Length		4	Length in bytes of TLV fields (0x000000AC)
TLV ECC PUBKEY	Type & Length	4	Set to 0x00088010
	Value	64	ECC P-256 OEM_ROOT_PK
TLV KEYHASH	Type & Length	4	Set to 0x10144008
	Value	32	SHA2-256 hash value of OEM_BL_PK
TLV EXPECTED SIG	Type & Length	4	Set to 0x20088410
	Value	64	Signature

Table 5 - Code Certificate structure

Field		Size (bytes)	Description
Header	Magic	4	Set to 0x636F6463
	Manifest Version	4	Set to 0x00010000
	Flags	4	Set to 0x00000000
	Load Addr	4	Set to 0x02000000
	Dest Addr	4	Set to 0x02000000
	Image Size	4	Size of OEM_BL set in multiple of 16, min size 64 bytes
	Image Version	4	Version number of OEM_BL. 1 – 64

	Build Number	4	Set to 0x00000000
TLV Length		4	Length in bytes of TLV fields
TLV ECC PUBKEY	Type & Length	4	Set to 0x01088010
	Value	64	ECC P-256 OEM_BL_PK
TLV EXPECTED CRC	Type & Length	4	Set to 0x40000001
	Value	4	CRC32 of OEM_BL
TLV SIGNER_ID	Type & Length	4	Set to 0x10144008
	Value	32	SHA2-256 hash value of OEM_BL_PK
TLV EXPECTED SIG	Type & Length	4	Set to 0x25088410
	Value	64	Signature of [Code Certificate | OEM_BL] signed with OEM_BL_SK

Requirements for this step are:

- OEM_ROOT keys (in HSM)
- OEM_BL keys (in HSM)
- OEM_BL binary (updated with embedded APP_PK)
- Meta data: bootloader load address, bootloader region size, image version

The Key Certificate is verified with the OEM_ROOT_PK injected via the .rkey, is signed with OEM_ROOT_SK and its function is to verify OEM_BL_PK.

The code below shows how to generate the key certificate, using the HSM class and the function `extract_raw_public_key_qxqy(…)` described in 7.1.1. Note that all the fields are in big-endian format.

```
def generate_key_certificate(oem_root_sk_handle: str, oem_root_pk: bytes, oem_bl_pk: bytes,
hsm_client) -> bytes:

    # Extract raw 64-byte public keys
    oem_root_pk_raw = extract_raw_public_key_qxqy(oem_root_pk)
    oem_bl_pk_raw = extract_raw_public_key_qxqy(oem_bl_pk)

    # Header: Magic + Manifest Version + Flags/Reserved
    header = struct.pack(
        '<II24s',
        0x6B657963,         # Header Magic 'keyc'
        0x00010000,         # Manifest Version
        b'\x00'*24          # Flags & Reserved
    )

    # TLV ECCPUBKEY (Root PK)
    tlv_eccpubkey = struct.pack('<I', 0x00088010) + oem_root_pk_raw

    # TLV KEYHASH (SHA256 of OEM_BL_PK)
    bl_pk_hash = hashlib.sha256(oem_bl_pk_raw).digest()
    tlv_keyhash = struct.pack('<I', 0x10144008) + bl_pk_hash

    # signature placeholder, will be replaced after signing
    tlv_expected_sig_placeholder = struct.pack('<I', 0x20088410) + b'\x00'*64

    # Compute TLV length: sum of all TLVs except the TLV length field itself
    tlv_length_value = len(tlv_eccpubkey) + len(tlv_keyhash) + len(tlv_expected_sig_placeholder)
    tlv_length = struct.pack('<I', tlv_length_value)
```

```
    # Payload to sign
    payload = header + tlv_length + tlv_eccpubkey + tlv_keyhash

    # Sign with HSM OEM_ROOT_SK
    digest = hashlib.sha256(payload).digest()
    signature = hsm_client.sign_digest(oem_root_sk_handle, digest)

    # TLV EXPECTED_SIG
    tlv_expected_sig = struct.pack('<I', 0x20088410) + signature

    # Final Key Certificate
    return payload + tlv_expected_sig
```

The Code Certificate is verified with the OEM_BL_PK from the Key Certificate, is signed with OEM_BL_SK and its function is to verify the second stage bootloader (MCUBoot). Similarly to the Key Certificate above, code to generate the code certificate is found below:

```
def generate_code_certificate(
    oem_bl_sk_handle: str,
    oem_bl_pk: bytes,
    oem_bl_binary: bytes,
    load_addr: int,
    dest_addr: int,
    image_version: int,
    build_number: int,
    hsm_client,
    oem_bl_pk_hash: Optional[bytes] = None,
) -> bytes:
    """
    Generate Code Certificate that authenticates the OEM Bootloader binary
    """
    # Extract raw 64-byte BL PK
    oem_bl_pk_raw = extract_raw_public_key_qxqy(oem_bl_pk)

    # Pad binary: multiple of 16, min 64 bytes
    img_size = len(bl_binary)
    if img_size < 64:
        bl_binary += b'\xFF' * (64 - img_size)
        img_size = 64
    if img_size % 16 != 0:
        pad = 16 - (img_size % 16)
        bl_binary += b'\xFF' * pad
        img_size += pad


    # Header: Magic + Metadata
    header = struct.pack(
        '<IIIIIIII',
        0x636F6463,       # 'codc'
        0x00010000,       # Manifest Version
        0x00000000,       # flags
        load_addr,        # Load Addr
        dest_addr,        # Dest Addr
        img_size,         # Image Size
        image_version,    # Image Version
        build_number      # Build Number
    )

    # TLV ECCPUBKEY (Bootloader PK)
    tlv_eccpubkey = struct.pack('<I', 0x01088010) + oem_bl_pk_raw

    # TLV EXPECTED_CRC
    bl_crc32 = zlib.crc32(bytes(bl_binary)) & 0xFFFFFFFF
    tlv_crc = struct.pack('<I', 0x40000001) + struct.pack('<I', bl_crc32)
```

```
    # TLV SIGNER_ID (SHA256 of BL PK) - needs to be the same as keyhash field value in key
certificate
    bl_pk_hash = hashlib.sha256(oem_bl_pk_raw).digest()
    tlv_signer = struct.pack('<I', 0x10144008) + bl_pk_hash

    # placeholder, will be replaced after signing
    tlv_expected_sig_placeholder = struct.pack('<I', 0x25088410) + b'\x00'*64

    # Compute TLV length: sum of all TLVs except the TLV length field itself
    tlv_length_value = len(tlv_eccpubkey) + len(tlv_crc) + len(tlv_signer) + len(tlv_expected_sig)
    tlv_length = struct.pack('<I', tlv_length_value)

    # Payload to sign: header + TLVs + BL binary
    payload = header + tlv_length + tlv_eccpubkey + tlv_crc + tlv_signer + bl_binary


    # Sign with BL secret key
    digest = hashlib.sha256(payload).digest()
    signature = hsm_client.sign_digest(oem_bl_sk_handle, digest)

    # TLV EXPECTED_SIG
    tlv_expected_sig = struct.pack('<I', 0x25088410) + signature

    # Final Code Certificate
    return header + tlv_length + tlv_eccpubkey + tlv_crc + tlv_signer + tlv_expected_sig
```

7.2 Phase 3: Programming and Lockdown

The final phase takes care of programming the artifacts created in the previous phase onto the device. The following sections capture the commands required for these operations and their Python implementation.

For the commands to run successfully, the device needs to be in “Boot Mode”, meaning the boot firmware is executing and receptive to commands being sent to the device.

There are two types of packets exchanged: Command and Data, each following its own structure:

Table 6 - Command Packet Format

Symbol	Size	Value	Description
SOH	1 byte	01h	Start of command packet.
LNH	1 byte		Packet length (length of CMD + Command information) [High].
LNL	1 byte		Packet length (length of CMD + Command information) [Low].
CMD	1 byte		Command code.
Command information	0–255 bytes		Command information. Examples: - For Write command: Start/End address - For Baudrate setting command: UART baudrate
SUM	1 byte		Sum data of LNH + LNL + CMD + Command information (expressed as two’s complement). Example: LNH + LNL + CMD + Command information(1) + ... + Command information(n) + SUM = 00h.
ETX	1 byte	03h	End of packet.

Table 7 - Data Packet Format

Symbol	Size	Value	Description
SOD	1 byte	81h	Start of data packet.
LNH	1 byte		Packet length (length of RES + Data) [High].
LNL	1 byte		Packet length (length of RES + Data) [Low].
RES	1 byte		Response code.
Data	1–1024 bytes		Transmit data. Examples: - For data transmission: Write data. - For status transmission: status code (STS), status details (ST2) and failure address (ADR).
SUM	1 byte		Sum data of LNH + LNL + RES + Data (expressed as two's complement). Example: LNH + LNL + RES + Data(1) + Data(2) + ... + Data(n) + SUM = 00h.
ETX	1 byte	03h	End of packet.

Before issuing commands, the communication with the boot firmware needs to be established. This is achieved with the following protocol:

1. Host sends 3 bytes of 0x00
2. Boot firmware replies with ACK (0x00)
3. Host sends Generic Code (0x55)
4. Boot firmware replies with Boot Code (0xC6)

More details on the Boot Firmware can be found in *RA8M1 Boot Firmware* (R01AN7140EU0130).

To simplify the development of the provisioning process, a class-based interface for handling communication with an RA8 device according to the Boot Firmware protocol, is proposed in the code below. This assumes communication happen over serial.

The code also contains the implementation of the sync protocol to boot mode explained above.

```
class DLMState(IntEnum):
    """Device Lifecycle Management states."""
    CM = 0x01
    OEM = 0x04
    LCK_BOOT = 0x06
    RMA_REQ = 0x07
    RMA_ACK = 0x08
    RMA_RET = 0x09

class ProtectionLevel(IntEnum):
    """Protection levels."""
    PL2 = 0x02
    PL1 = 0x03
    PL0 = 0x04

class AuthenticationLevel(IntEnum):
    """Authentication levels."""
    AL2 = 0x02
    AL1 = 0x03
    AL0 = 0x04

class RA8ProvisioningClient:

    # Configuration constants
    SECURE_ALIAS_BASE = 0x02000000
```

```
    SECURE_ALIAS_LIMIT = 0x021F7FFF
    NONSEC_ALIAS_BASE = 0x12008000
    NONSEC_ALIAS_LIMIT = 0x121F7FFF
    ERASE_SECTOR_SIZE = 32 * 1024  # 32KB
    BOOT_BLK_START = 0x02000000
    BOOT_BLK_END = 0x02007FFF
    FLASH_BLOCK_SIZE = 128  # 128 bytes

    # OSM regions
    CF_OSM_SEC_BASE = 0x0300A100
    CF_OSM_SEC_SIZE = 384
    CF_OSM_PARAM_BASE = 0x0300A200
    DF_OSM_BASE = 0x27030080
    DF_OSM_SIZE = 720

    # Timing
    SHORT_DELAY = 0.15
    LONG_DELAY = 3.0
    FLASH_WRITE_DELAY = 0.05
    INITIALIZE_DELAY = 5.0

    def __init__(self, com_port: str, baud_rate: int = 9600):
        """
        Initialize RA8 provisioning client.

        Args:
            com_port: COM port name
            baud_rate: Baud rate (default: 9600)
        """
        self.com_port = com_port
        self.baud_rate = baud_rate
        self.ser: Optional[serial.Serial] = None
        self._connected = False

    def connect(self) -> None:
        """Open serial connection."""
        try:
            self.ser = serial.Serial(
                self.com_port,
                self.baud_rate,
                timeout=1,
                write_timeout=1
            )
            time.sleep(self.LONG_DELAY)
            self.ser.reset_input_buffer()
            self.ser.reset_output_buffer()
            self._connected = True
        except Exception as e:
            raise DeviceError(f"Failed to open serial port {self.com_port}: {e}")

    def disconnect(self) -> None:
        """Close serial connection."""
        if self.ser and self.ser.is_open:
            self.ser.close()
            self._connected = False

    def __enter__(self):
        """Context manager entry."""
        self.connect()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Context manager exit."""
        self.disconnect()

    # ===== Low-level communication =====

    def _calc_sum(self, data: bytes) -> bytes:
        """Calculate checksum for packet."""
```

```
        count = sum(data) % 256
        tot = (256 - count) % 256
        return tot.to_bytes(1, "big")

    def _receive_data_packet(self) -> bytearray:
        """Receive data packet: SOD=0x81, reads length, then payload + SUM + ETX."""
        if not self.ser:
            raise DeviceError("Not connected")

        packet = bytearray()
        # Wait for SOD with timeout
        SOD = self.ser.read(1)
        if len(SOD) == 0:
            raise DeviceError("No response from device")

        # Check if it's an error packet (SOH = 0x01) instead of data packet (SOD = 0x81)
        if SOD == b'\x01':
            error_packet = bytearray(SOD)
            # Read at least 7 more bytes: LNH, LNL, RES, data, SUM, ETX
            error_packet += self.ser.read(7)
            raise DeviceError(f"Device returned error packet: {error_packet.hex()}")
        if SOD != b'\x81':
            raise DeviceError(f"Unexpected SOD: {SOD.hex()}")

        packet += SOD
        LNH = self.ser.read()
        packet += LNH
        LNL = self.ser.read()
        packet += LNL
        length = (int.from_bytes(LNH, "big") * 256) + int.from_bytes(LNL, "big") + 2
        for _ in range(length):
            packet += self.ser.read()

        return packet

    def _decode_status_packet(self, pkt: bytes) -> Dict:
        """Decode RES/STS/ST2 and optional ADR for diagnostics."""
        st = {"raw": pkt.hex()}
        try:
            st["SOD"] = pkt[0]
            st["LNH"] = pkt[1]
            st["LNL"] = pkt[2]
            st["RES_RAW"] = pkt[3]
            st["RES"] = pkt[3] & 0x7F  # mask MSB
            st["STS"] = pkt[4] if len(pkt) > 4 else None
            st["ST2"] = pkt[5] if len(pkt) > 5 else None
            st["LEN"] = (st["LNH"] << 8) | st["LNL"]

            if len(pkt) >= 10:
                st["ADR"] = int.from_bytes(pkt[6:10], "big")
        except Exception:
            pass
        return st

    def _communication_setting(self) -> bool:
        """Establish UART/USB boot firmware handshake."""
        if not self.ser:
            raise DeviceError("Not connected")

        loopcount = 20
        while loopcount != 0:
            self.ser.write(b'\x00\x00\x00')
            time.sleep(self.SHORT_DELAY)
            h = self.ser.read()
            if h == b'\x00':
                # ACK received
                return True
            loopcount -= 1
            time.sleep(self.SHORT_DELAY)
```

```
        return False

    def _command_inquiry(self) -> None:
        """Send inquiry command."""
        if not self.ser:
            raise DeviceError("Not connected")

        command = b'\x01\x00\x01\x00\xFF\x03'
        self.ser.write(command)
        time.sleep(self.LONG_DELAY)

    def _command_signature_request(self) -> int:
        """Request device signature."""
        if not self.ser:
            raise DeviceError("Not connected")

        command = b'\x01\x00\x01\x3A\xC5\x03'
        self.ser.write(command)
        time.sleep(self.LONG_DELAY)

        return_packet = self._receive_data_packet()
        RES = return_packet[3] & 0x7F

        if RES != 0x3A:
            raise DeviceError("Signature request failed")

        return RES

    def get_dlm_state(self) -> Tuple[int, int]:
        """Get current DLM state."""
        if not self.ser:
            raise DeviceError("Not connected")

        command = b'\x01\x00\x01\x2C\xD3\x03'
        self.ser.write(command)
        time.sleep(self.LONG_DELAY)

        rp = self._receive_data_packet()
        RES = rp[3] & 0x7F
        DLM = rp[4] if RES == 0x2C else 0xFF

        return RES, DLM

    def get_protection_level(self) -> Tuple[int, int]:
        """Get current protection level."""
        if not self.ser:
            raise DeviceError("Not connected")

        command = b'\x01\x00\x01\x73\x8C\x03'
        self.ser.write(command)
        time.sleep(self.LONG_DELAY)

        rp = self._receive_data_packet()
        RES = rp[3] & 0x7F
        CPL = rp[4] if RES == 0x73 else 0xFF

        return RES, CPL

    def get_authentication_level(self) -> Tuple[int, int]:
        """Get current authentication level."""
        if not self.ser:
            raise DeviceError("Not connected")

        SOH = b'\x01'
        LNH = b'\x00'
        LNL = b'\x01'
        CMD = b'\x75'
        SUM = b'\x8A'
```

```
        ETX = b'\x03'
        cmd = SOH + LNH + LNL + CMD + SUM + ETX

        self.ser.write(cmd)
        time.sleep(self.LONG_DELAY)

        rp = self._receive_data_packet()
        RES = rp[3] & 0x7F
        CAL = rp[4] if RES == 0x75 else 0xFF

        if RES != 0x75:
            raise DeviceError("Read Authentication Level - FAIL")

        return RES, CAL

    def _dlm_state_transition(self, source_state: int, target_state: int) -> bool:
        """Transition DLM state."""
        if not self.ser:
            raise DeviceError("Not connected")

        SOH = b'\x01'
        LNH = b'\x00'
        LNL = b'\x03'
        CMD = b'\x71'
        SDLM = source_state.to_bytes(1, "big")
        DDLM = target_state.to_bytes(1, "big")
        SUM = self._calc_sum(LNH + LNL + CMD + SDLM + DDLM)
        ETX = b'\x03'

        self.ser.write(SOH + LNH + LNL + CMD + SDLM + DDLM + SUM + ETX)
        time.sleep(self.LONG_DELAY)

        rp = self._receive_data_packet()
        RES = rp[3] & 0x7F
        return RES == 0x71

    def initialize_device(self) -> None:
        """
        Initialize device (CM->OEM transition via Initialize command).
        """
        if not self.ser:
            raise DeviceError("Not connected")

        # Get current DLM state
        _, dlm = self.get_dlm_state()

        # Initialize command transitions device to OEM/PL2 regardless of current DLM
        # It clears: User area, Data area, Config area, EEP, Boundary, Key index
        # AND sets PL to PL2 (critical for OEM Root Key programming)
        if dlm == 0x01:  # CM
            self._dlm_state_transition(0x01, 0x04)
            dlm = 0x04  # Update to OEM for Initialize command

        if dlm == 0x04:  # OEM (includes both original OEM and transitioned from CM)
            # Send Initialize command
            SOH = b'\x01'
            LNH = b'\x00'
            LNL = b'\x03'
            CMD = b'\x50'
            SDLM = dlm.to_bytes(1, "big")  # 0x04 (OEM)
            DDLM = b'\x04'  # target OEM
            SUM = self._calc_sum(LNH + LNL + CMD + SDLM + DDLM)
            ETX = b'\x03'

            self.ser.write(SOH + LNH + LNL + CMD + SDLM + DDLM + SUM + ETX)
            time.sleep(self.INITIALIZE_DELAY)

            rp = self._receive_data_packet()
            RES = rp[3] & 0x7F
```

```
            if RES != 0x50:
                raise DeviceError("Initialize failed")

            # Reconnect
            self.disconnect()
            time.sleep(self.LONG_DELAY)
            self.connect()

            if not self._communication_setting():
                raise DeviceError("Failed to reconnect after initialize")

            self.ser.write(b'\x55')
            time.sleep(self.SHORT_DELAY)
            _ = self.ser.read()

        # Verify final state
        _, dlm2 = self.get_dlm_state()
        _, pl2 = self.get_protection_level()
        _, al2 = self.get_authentication_level()

    def connect_to_boot_mode(self) -> None:
        """Connect to boot mode."""
        if not self.ser:
            raise DeviceError("Not connected")

        if not self._communication_setting():
            self._command_inquiry()
            rp = self._receive_data_packet()
            if (rp[3] & 0x7F) != 0x00:
                raise DeviceError("Failed to connect to boot mode")

        self.ser.write(b'\x55')
        time.sleep(self.SHORT_DELAY)
        boot_code = self.ser.read()

        if boot_code != b'\xC6':
            raise DeviceError(f"Unexpected boot code: {boot_code.hex() if boot_code else '-'}")

        self._command_signature_request()
```

7.2.1 Initialise the device

Issue the *initialize* command to transition the device to OEM/PL2. Erase all non-locked flash regions and sets the device to a known factory default.

It is good to wrap the initialize command (0x50) between a DLM state check and a verification step:

Table 8 - Initialize Command Sequence

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
GET_DLM_STATE	0x01	0x00	0x01	0x2C	-	0xD3	0x03
DLM_STATE_TRANSITION	0x01	0x00	0x03	0x71	SDLM = var DDLM = var	calculated	0x03
INITIALIZE	0x01	0x00	0x03	0x50	SDLM = 0x04 DDLM = 0x04	calculated	0x03

GET_PROTECTION_LEVEL	0x01	0x00	0x01	0x73	-	0x8C	0x03
GET_AUTHENTICATION_LEVEL	0x01	0x00	0x01	0x75	-	0x8A	0x03

Code implementing this sequence is already provided in the code above.

7.2.2 Inject keys

The next step is to inject the wrapped keys (*rkey* files). This includes the wrapped OEM_ROOT_PK and optionally, the AL and RMA keys.

The injection consists of a command phase and a data phase:

Table 9 - Program OEM_ROOT_KEY command

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
PROGRAM_OEM_ROOT_KEY	0x01	0x00	0x03	0x2E	KID = var PLK = var	calculated	0x03

Table 10 - Program OEM_ROOT_KEY data

Command	SOD	LNH	LNL	RES	Data	SUM	ETX
PROGRAM_OEM_ROOT_KEY	0x81	0x00	0x85	0x2E	KEY_DATA = 132 bytes	calculated	0x03

The code below makes use of the `RA8ProvisioningClient` reported at the beginning of this chapter.

```
def program_oem_root_key(
    self,
    oem_key_file: Path,
    key_id: int = 0,
    permanent_lock: bool = False
) -> None:

    message_bytes, encrypted_key_size = self._parse_rkey_file(oem_key_file)

    # Command packet: 0x2E, KID, PLK
    SOH = b'\x01'
    LNH = b'\x00'
    LNL = b'\x03'
    CMD = b'\x2E'
    KID = key_id.to_bytes(1, "big")
    PLK = (b'\x00' if permanent_lock else b'\xFF')  # 0x00=lock, 0xFF=no-lock
    SUM = self.client._calc_sum(LNH + LNL + CMD + KID + PLK)
    ETX = b'\x03'

    command = SOH + LNH + LNL + CMD + KID + PLK + SUM + ETX
    self.client.ser.write(command)
    time.sleep(self.client.LONG_DELAY)
    rp = self.client._receive_data_packet()

    # Decode response status
    RES = rp[3] & 0x7F
    STS = rp[4] if len(rp) > 4 else None

    if RES != 0x2E:
```

```
        raise DeviceError(f"OEM root key command failed: device returned RES=0x{RES:02X}")
    # STS is non-fatal, continue

    message_data = message_bytes[24:]
    if len(message_data) < 128:
        message_data = message_data + (b'\xFF' * (128 - len(message_data)))

    SOD = b'\x81'
    LNH = b'\x00'
    LNL = b'\x85'
    RES = b'\x2E'
    SKR = b'\x00' * 4
    key_data = SKR + message_data[0:32] + message_data[32:48] + message_data[48:128]
    SUM = self.client._calc_sum(LNH + LNL + RES + key_data)
    ETX = b'\x03'

    packet = SOD + LNH + LNL + RES + key_data + SUM + ETX
    self.client.ser.write(packet)
    time.sleep(self.client.LONG_DELAY)
    rp = self.client._receive_data_packet()

    # Decode data response
    RES = rp[3] & 0x7F
    STS = rp[4] if len(rp) > 4 else None

    if RES != 0x2E:
        raise DeviceError(f"OEM root key data response failed: RES=0x{RES:02X}")
    if STS is not None and STS != 0x00:
        if STS == 0xDB:
            raise DeviceError(
                "OEM Root Key already programmed (STS=0xDB). Full chip erase required"
            )
        else:
            raise DeviceError(f"OEM root key programming failed: STS=0x{STS:02X}")
```

To inject the lifecycle management keys, the commands used are:

Table 11 - Program AL keys command

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
INJECT_DLM_KEY	0x01	0x00	0x02	0x28	KYTY = var	calculated	0x03

With KYTY being the key type selector: 0x01 for AL2, 0x02 for AL1.

Table 12 - Program AL keys data

Command	SOD	LNH	LNL	RES	Data	SUM	ETX
INJECT_DLM_KEY	0x81	0x00	0x55	0x28	KEY_DATA = 84 bytes	calculated	0x03

```
def inject_dlm_key(self, key_file: Path, key_type: int) -> bool:
    if not self.client.ser:
        raise DeviceError("Not connected")

    key_names = {0x01: "AL2", 0x02: "AL1"}

    message_bytes, encrypted_key_size = self._parse_rkey_file(key_file)
    if message_bytes is None:
```

```
        return False

    SOH = b'\x01'
    LNH = b'\x00'
    LNL = b'\x02'
    CMD = b'\x28'
    KYTY = key_type.to_bytes(1, 'big')
    SUM = self.client._calc_sum(LNH + LNL + CMD + KYTY)
    ETX = b'\x03'

    self.client.ser.write(SOH + LNH + LNL + CMD + KYTY + SUM + ETX)
    time.sleep(self.client.LONG_DELAY)

    rp = self.client._receive_data_packet()
    if (rp[3] & 0x7F) != 0x28:
        return False

    SOD = b'\x81'
    LNH = b'\x00'
    LNL = b'\x55'
    RES = b'\x28'
    SKR = b'\x00' * 4
    key_data     =     SKR     +     message_bytes[24:56]     +     message_bytes[56:72]     +
message_bytes[72:72+encrypted_key_size]
    SUM = self.client._calc_sum(LNH + LNL + RES + key_data)
    ETX = b'\x03'

    self.client.ser.write(SOD + LNH + LNL + RES + key_data + SUM + ETX)
    time.sleep(self.client.LONG_DELAY)

    rp = self.client._receive_data_packet()
    if (rp[3] & 0x7F) == 0x28 and rp[4] == 0x00:
        return True

    return False
```

7.2.3 Data Programming

This steps programs the combined bootloader + application binary and the certificates (Key and Code).

The bootloader and application are programmed together, using the combined binary file assembled at the earlier stage. Along with the firmware, the Option Setting Memory values (embedded within the firmware binary) also need to be written.

An ERASE command is issued to clear the required memory area, after which the WRITE command (and data) is used repeatedly to write all the data in chunks, at the right.

Table 13 - Erase and Write commands

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
ERASE	0x01	0x00	0x09	0x12	SAD = var, 4B EAD = var, 4B	calculated	0x03
WRITE	0x01	0x00	0x09	0x13	SAD = var, 4B EAD = var, 4B	calculated	0x03

Table 14 - Write data

Command	SOD	LNH	LNL	RES	Data	SUM	ETX
WRITE	0x81	var	var	0x13	DATA = var, 1-1024B	calculated	0x03

With LNH:LNL = length of RES + DATA (data up to 1024 bytes per packet)

The code below provides a class implementation that uses the RA8ProvisioningClient to program the bootloader and application files in srec format, as well as the Option Setting Memory. It also contains the additional helper functions needed to parse the records and prepare the data in the correct format.

```
class RA8SRECProgrammer:
    """Handles SREC file programming operations for RA8 devices."""

    def __init__(self, client: RA8ProvisioningClient):
        self.client = client

    def _parse_srec_file_strict(self, filename: Path) -> List[Tuple[int, bytes]]:
        """Parse S1/S2/S3 records, verify checksum, return list of (address, bytes)."""
        blocks = []
        try:
            with open(filename, 'r') as f:
                for ln, line in enumerate(f, 1):
                    line = line.strip()
                    if not line or line[0] != 'S':
                        continue

                    rtype = line[1]
                    if rtype in ('1', '2', '3'):
                        count = int(line[2:4], 16)
                        addr_len = 4 if rtype == '1' else (6 if rtype == '2' else 8)
                        address = int(line[4:4+addr_len], 16)
                        data_start = 4 + addr_len
                        data_end = 4 + (count * 2)
                        data_hex = line[data_start:data_end-2]
                        raw = bytes.fromhex(line[2:data_end])

                        if (sum(raw) & 0xFF) != 0xFF:
                            raise ValueError(f"SREC line {ln}: checksum mismatch")

                        data = bytes.fromhex(data_hex)
                        blocks.append((address, data))
                    else:
                        # S0/S5/S7/S8/S9 ignored
                        continue

            return blocks
        except Exception as e:
            raise DeviceError(f"Failed to parse SREC file: {e}")

    def _merge_contiguous_blocks(self, blocks: List[Tuple[int, bytes]], phrase_size: int = 128) -
> List[Tuple[int, bytes]]:
        """Merge sorted (address, bytes) blocks unless the gap contains at least one aligned, full
write phrase."""
        if not blocks:
            return []

        cur_addr, cur_data = blocks[0]
        cur_data = bytearray(cur_data)
        merged = []

        for addr, data in blocks[1:]:
            data = bytes(data)
```

```
            cur_end = cur_addr + len(cur_data) - 1

            if addr <= cur_end:
                # Overlap: expand and overlay
                overlap_start = addr
                overlap_end = min(cur_end, addr + len(data) - 1)

                if addr + len(data) - 1 > cur_end:
                    extend_len = (addr + len(data) - 1) - cur_end
                    cur_data += b'\xFF' * extend_len
                    cur_end = cur_addr + len(cur_data) - 1

                for i in range(overlap_start, overlap_end + 1):
                    cur_data[i - cur_addr] = data[i - addr]
                continue

            # Non-overlapping: check gap
            gap_start = cur_end + 1
            gap_end = addr - 1
            gap_len = addr - (cur_end + 1)

            start_aligned = ((gap_start + phrase_size - 1) // phrase_size) * phrase_size
            last_phrase_start = (gap_end // phrase_size) * phrase_size
            has_full_aligned_phrase = (start_aligned + phrase_size - 1) <= gap_end

            if has_full_aligned_phrase:
                merged.append((cur_addr, bytes(cur_data)))
                cur_addr, cur_data = addr, bytearray(data)
            else:
                cur_data += b'\xFF' * gap_len
                cur_data += data

        merged.append((cur_addr, bytes(cur_data)))
        return merged

    def _remap_alias(self, address: int, to_secure: bool = True) -> int:
        """Map upper/non-secure alias -> lower/secure alias."""
        if to_secure:
            if self.client.NONSEC_ALIAS_BASE <= address <= self.client.NONSEC_ALIAS_LIMIT:
                offset = address - self.client.NONSEC_ALIAS_BASE
                return self.client.SECURE_ALIAS_BASE + offset
        else:
            if self.client.SECURE_ALIAS_BASE <= address <= self.client.SECURE_ALIAS_LIMIT:
                offset = address - self.client.SECURE_ALIAS_BASE
                return self.client.NONSEC_ALIAS_BASE + offset

        return address

    def _classify_region(self, addr: int) -> str:
        """Classify memory region."""
        if self.client.SECURE_ALIAS_BASE <= addr <= self.client.SECURE_ALIAS_LIMIT:
            return "code_secure"
        if self.client.NONSEC_ALIAS_BASE <= addr <= self.client.NONSEC_ALIAS_LIMIT:
            return "code_nonsec"
        if 0x0300A000 <= addr <= 0x0300AFFF:
            return "option_setting"
        if 0x37000000 <= addr <= 0x37002FFF:
            return "data_flash"
        if 0x22000000 <= addr <= 0x22FFFFFF:
            return "sram"
        return "other"

    def _range_check_block(self, addr: int, data_len: int) -> bool:
        """Check if block is within valid range."""
        end_addr = addr + data_len - 1
        in_secure = (
            self.client.SECURE_ALIAS_BASE <= addr <= self.client.SECURE_ALIAS_LIMIT and
            self.client.SECURE_ALIAS_BASE <= end_addr <= self.client.SECURE_ALIAS_LIMIT
        )
```

```
        in_nonsec = (
            self.client.NONSEC_ALIAS_BASE <= addr <= self.client.NONSEC_ALIAS_LIMIT and
            self.client.NONSEC_ALIAS_BASE <= end_addr <= self.client.NONSEC_ALIAS_LIMIT
        )
        return in_secure or in_nonsec

    def _align_down(self, addr: int, size: int) -> int:
        """Align address down."""
        return addr - (addr % size)

    def _align_up(self, addr: int, size: int) -> int:
        """Align address up."""
        return ((addr + size - 1) // size) * size

    def _command_erase(self, start_addr: int, end_addr: int) -> bool:
        """Erase inclusive range [start_addr..end_addr], aligned externally."""
        if not self.client.ser:
            raise DeviceError("Not connected")

        SOH = b'\x01'
        LNH = b'\x00'
        LNL = b'\x09'
        CMD = b'\x12'
        SAD = start_addr.to_bytes(4, 'big')
        EAD = end_addr.to_bytes(4, 'big')
        SUM = self.client._calc_sum(LNH + LNL + CMD + SAD + EAD)
        ETX = b'\x03'

        pkt = SOH + LNH + LNL + CMD + SAD + EAD + SUM + ETX
        self.client.ser.write(pkt)
        time.sleep(self.client.LONG_DELAY)

        rp = self.client._receive_data_packet()
        st = self.client._decode_status_packet(rp)
        return (st.get("RES") == 0x12) and (st.get("STS") in (0x00, None))

    def _command_write(self, address: int, data: bytes) -> bool:
        """Write data to flash with robust RES/STS checking."""
        if not self.client.ser:
            raise DeviceError("Not connected")

        # Command packet (announce address span)
        SOH = b'\x01'
        LNH = b'\x00'
        LNL = b'\x09'
        CMD = b'\x13'
        SAD = address.to_bytes(4, 'big')
        EAD = (address + len(data) - 1).to_bytes(4, 'big')
        SUM = self.client._calc_sum(LNH + LNL + CMD + SAD + EAD)
        ETX = b'\x03'

        self.client.ser.write(SOH + LNH + LNL + CMD + SAD + EAD + SUM + ETX)
        time.sleep(self.client.FLASH_WRITE_DELAY)

        rp = self.client._receive_data_packet()
        st = self.client._decode_status_packet(rp)

        if st.get("RES") != 0x13:
            # NACK
            return False

        if st.get("STS") not in (0x00, None):
            # NACK
            return False

        # Data chunks (<=1024 bytes each)
        offset = 0
        while offset < len(data):
            chunk = data[offset:offset+1024]
```

```
            SOD = b'\x81'
            length = len(chunk) + 1
            LNH = (length >> 8).to_bytes(1, 'big')
            LNL = (length & 0xFF).to_bytes(1, 'big')
            RES = b'\x13'
            SUM = self.client._calc_sum(LNH + LNL + RES + chunk)
            ETX = b'\x03'

            self.client.ser.write(SOD + LNH + LNL + RES + chunk + SUM + ETX)
            time.sleep(self.client.FLASH_WRITE_DELAY)

            rp = self.client._receive_data_packet()
            st = self.client._decode_status_packet(rp)

            if st.get("RES") != 0x13:
                # NACK
                return False
            if st.get("STS") not in (0x00, None):
                # NACK
                return False

            offset += len(chunk)

        return True

    def _write_block_in_phrases_aligned(self, start_addr: int, data_bytes: bytes) -> bool:
        """Write data_bytes starting at start_addr using 128-byte, phrase-aligned writes."""
        # Align start down to phrase boundary
        aligned_start = start_addr & ~(self.client.FLASH_BLOCK_SIZE - 1)
        pre_gap = start_addr - aligned_start

        # Build buffer with leading 0xFF for gap, then data
        buf = (b'\xFF' * pre_gap) + data_bytes

        # Pad tail to multiple of FLASH_BLOCK_SIZE
        tail = (self.client.FLASH_BLOCK_SIZE - (len(buf) % self.client.FLASH_BLOCK_SIZE)) %
self.client.FLASH_BLOCK_SIZE
        if tail:
            buf += b'\xFF' * tail

        # Program one phrase per write command
        for i in range(0, len(buf), self.client.FLASH_BLOCK_SIZE):
            phrase_addr = aligned_start + i
            phrase = buf[i:i+self.client.FLASH_BLOCK_SIZE]

            ok = self._command_write(phrase_addr, phrase)
            if not ok:
                raise DeviceError(f"Phrase write failed @ 0x{phrase_addr:08X}")

        return True

    def _erase_srec_span_skip_bootblock(self, blocks: List[Tuple[int, bytes]]) -> bool:
        """Erase union of SREC addresses aligned to 32KB, skipping boot block."""
        if not blocks:
            return True

        min_addr = min(a for a, _ in blocks)
        max_addr = max(a + len(d) - 1 for a, d in blocks)

        start = self._align_down(min_addr, self.client.ERASE_SECTOR_SIZE)
        end = self._align_up(max_addr, self.client.ERASE_SECTOR_SIZE) - 1

        return self._command_erase(start, end)

    def _pad16_ff(data: bytes, max_len: Optional[int] = None) -> bytes:
        """Pad data to a multiple of 16 bytes with 0xFF. Optionally truncate to max_len."""
        tail = (-len(data)) % 16
        padded = data + (b'\xFF' * tail if tail else b'')
        if max_len is not None and len(padded) > max_len:
```

```
            padded = padded[:max_len]
        return padded

    def _collect_osm_srec_records(self, srec_file: Path) -> List[Tuple[int, bytes]]:
        """Collect OSM (Option Setting Memory) records from SREC file."""
        blocks = self._parse_srec_file_strict(srec_file)
        if not blocks:
            return []

        CF_MIN, CF_MAX = 0x0300A100, 0x0300A2FF
        DF_MIN, DF_MAX = 0x27030050, 0x270303FF

        out = []
        for addr, data in blocks:
            end = addr + len(data)

            # CF OSM intersection
            if not (end <= CF_MIN or addr > CF_MAX):
                s = max(addr, CF_MIN)
                e = min(end, CF_MAX + 1)
                piece = data[s - addr : s - addr + (e - s)]
                out.append((s, self._pad16_ff(piece, (CF_MAX + 1) - s)))

            # DF OSM intersection
            if not (end <= DF_MIN or addr > DF_MAX):
                s = max(addr, DF_MIN)
                e = min(end, DF_MAX + 1)
                piece = data[s - addr : s - addr + (e - s)]
                out.append((s, self._pad16_ff(piece, (DF_MAX + 1) - s)))

        out.sort(key=lambda x: x[0])
        return out

    def program_srec_file(self, srec_file: Path, enforce_secure_alias: bool = True) -> None:
        """
        Program SREC file to device.

        Args:
            srec_file: Path to SREC file
            enforce_secure_alias: Whether to remap non-secure addresses to secure alias
        """

        blocks = self._parse_srec_file_strict(srec_file)
        if not blocks:
            raise DeviceError("Failed to parse SREC file or no data records")

        # Sort blocks
        blocks.sort(key=lambda x: x[0])

        # Classify & collect
        code_blocks = []
        skipped = []

        for addr, data in blocks:
            region = self._classify_region(addr)
            if region == "code_secure":
                code_blocks.append((addr, data))
            elif region == "code_nonsec":
                if enforce_secure_alias:
                    code_blocks.append((self._remap_alias(addr, to_secure=True), data))
                else:
                    code_blocks.append((addr, data))
            else:
                skipped.append((addr, len(data), region))

        # Range checks
        for addr, data in code_blocks:
            if not self._range_check_block(addr, len(data)):
```

```
                raise DeviceError(f"SREC code block outside alias: 0x{addr:08X} ..
0x{addr+len(data)-1:08X}")

        # Merge contiguous blocks
        code_blocks = self._merge_contiguous_blocks(code_blocks)

        # Erase range (skip boot block)
        if not self._erase_srec_span_skip_bootblock(code_blocks):
            raise DeviceError("Erase failed before programming")

        # Write
        for i, (address, data) in enumerate(code_blocks):
            if not self._write_block_in_phrases_aligned(address, data):
                raise DeviceError(f"Write failed at address 0x{address:08X}")


    def program_osm_from_srec(self, srec_file: Path) -> None:
        """Program OSM (Option Setting Memory) from SREC file."""
        osm_records = self._collect_osm_srec_records(srec_file)
        if not osm_records:
            return
        for addr, data in osm_records:
            self._command_write(addr, data)
```

There isn't a dedicated command for each certificate, they are programmed together with the same command:

Table 15 - Program Certificates command

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
PROGRAM_CERTIFICATES	0x01	0x00	0x06	0x26	MAC = 0x02 KCS = var, 2B CCS = var, 2B	calculated	0x03

With MAC being 0x02 (HMAC-SHA256), KCS = Key Certificate size (big-endian), CCS = Code Certificate size (big-endian)

Table 16 - Program Certificates data

Command	SOD	LNH	LNL	RES	Data	SUM	ETX
PROGRAM_CERTIFICATES	0x81	var	var	0x26	KEY_CERT = var, N bytes CODE_CERT = var, M bytes	calculated	0x03

With LNH:LNL = length of RES + KEY_CERT + CODE_CERT

The code below provides a class implementation that uses the `RA8ProvisioningClient` to program the two certificates.

```
class RA8CertificateProgrammer:
    """Handles certificate programming operations for RA8 devices."""

    def __init__(self, client: RA8ProvisioningClient):
        self.client = client
```

```
    def _send_cert_single(self, keydata: bytes, certdata: bytes) -> bytearray:
        """Send certificate data in single packet (≤1024B)."""
        length = len(keydata) + len(certdata) + 1  # RES(1) + Data(N+M)
        SOD = b'\x81'
        LNH = (length >> 8).to_bytes(1, 'big')
        LNL = (length & 0xFF).to_bytes(1, 'big')
        RES = b'\x26'
        SUM = self.client._calc_sum(LNH + LNL + RES + keydata + certdata)
        ETX = b'\x03'

        pkt = SOD + LNH + LNL + RES + keydata + certdata + SUM + ETX

        self.client.ser.reset_input_buffer()
        self.client.ser.reset_output_buffer()
        self.client.ser.write(pkt)
        time.sleep(self.client.LONG_DELAY)
        time.sleep(self.client.LONG_DELAY)

        ack = self.client._receive_data_packet()
        return ack

    def program_certificates(self, key_cert_data: byte, code_cert_data: byte) -> None:
        """Program Key Certificate and Code Certificate (CMD 0x26)."""
        # Command phase (0x26)
        SOH = b'\x01'
        CMD = b'\x26'
        MAC = b'\x02'  # HMAC-SHA256
        KCS = len(key_cert_data).to_bytes(2, 'big')
        CCS = len(code_cert_data).to_bytes(2, 'big')
        LNH = b'\x00'
        LNL = b'\x06'  # CMD+MAC+KCS+CCS = 6 bytes
        SUM = self.client._calc_sum(LNH + LNL + CMD + MAC + KCS + CCS)
        ETX = b'\x03'

        cmd = SOH + LNH + LNL + CMD + MAC + KCS + CCS + SUM + ETX
        self.client.ser.write(cmd)
        time.sleep(self.client.LONG_DELAY)

        ack_cmd = self.client._receive_data_packet()
        st_cmd = self.client._decode_status_packet(ack_cmd)


        if st_cmd.get("RES") != 0x26:
            raise DeviceError(f"Code certificate command failed (RES != 0x26, got
0x{st_cmd.get('RES', 0):02X})")

        if st_cmd.get("STS") not in (0x00, None):
            if st_cmd.get("STS") == 0xD3:
                # skipped, cert already present
                return
            else:
                # Any other non-zero STS is a real error
                raise DeviceError(f"Certificate command failed with
STS=0x{st_cmd.get('STS'):02X}")

        cert_ret = self._send_cert_single(key_cert_data, code_cert_data)
        cert_stat = self.client._decode_status_packet(cert_ret)

        if cert_stat.get("RES") != 0x26:
            raise DeviceError(f"Code certificate programming failed (RES != 0x26, got
0x{cert_stat.get('RES', 0):02X})")

        if cert_stat.get("STS") not in (0x00, None):
            sts = cert_stat.get('STS')
            if sts == 0xDC:
                # Certificate version error - user needs to increase version
                raise DeviceError(f"Certificate version too low (STS=0xDC)")
            else:
```

```
            raise DeviceError(f"Certificate programming failed with STS=0x{sts:02X}")
```

7.2.4 Verification

The command used to program the certificates also triggers a built-in cryptographic verification, that is the boot firmware authenticates the bootloader binary using the data from the Code Certificate, verifying the signature and the image version. If the verification fails, the OEM_BL_digest is not generated and the boot firmware results in error.

Whilst this helps in detecting any error that might happen during the programming phase, it is customary to verify the flashed data via a CRC check.

In addition to this, before permanently locking it, it is good to reboot the device and verify the correct execution of the bootloader and application.

7.2.5 Lock the Device

The final step is to lock the device. This is achieved by transitioning its Protection Level to PL0 and optionally disable the initialize command, to make the current Root of Trust permanent (the firmware can still be updated, as long as it is signed with the correct key).

The command below assumes the transition is from PL2 to PL0. Alternatively, the current protection level can be read with a PROTECTION_LEVEL_REQUEST command.

Table 17 - Lock device command sequence

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
PROTECTION_LEVEL_TRANSIT	0x01	0x00	0x03	0x72	SPL = 0x02 DPL = 0x04	calculated	0x03
DISABLE_INITIALIZE	0x01	0x00	0x03	0x51	PMID = 0x01 PRMT = 0x00	calculated	0x03

For a fully locked device, the DLM state of the device can be transitioned to LCK_BOOT, which irreversibly disables the boot mode.

Table 18 - LCK_BOOT command

Command	SOH	LNH	LNL	CMD	CMD Information	SUM	ETX
LCK_BOOT	0x01	0x00	0x03	0x51	PMID = 0x02 PRMT = 0x00	calculated	0x03

Similarly to the previous commands, the code below uses `RA8ProvisioningClient` to handle the communications with the RA8 device.

```
def _transition_protection_level(self, target_pl: int) -> bool:
    _, current_pl = self.client.get_protection_level()
```

```
    SOH = b'\x01'
    LNH = b'\x00'
    LNL = b'\x03'
    CMD = b'\x72'
    SPL = current_pl.to_bytes(1, 'big')
    DPL = target_pl.to_bytes(1, 'big')
    SUM = self.client._calc_sum(LNH + LNL + CMD + SPL + DPL)
    ETX = b'\x03'

    self.client.ser.write(SOH + LNH + LNL + CMD + SPL + DPL + SUM + ETX)
    import time
    time.sleep(self.client.LONG_DELAY)

    rp = self.client._receive_data_packet()
    return (rp[3] & 0x7F) == 0x72

def _disable_initialize(self) -> bool:
    SOH = b'\x01'
    LNH = b'\x00'
    LNL = b'\x03'
    CMD = b'\x51'
    PMID = b'\x01'
    PRMT = b'\x00'
    SUM = self.client._calc_sum(LNH + LNL + CMD + PMID + PRMT)
    ETX = b'\x03'

    self.client.ser.write(SOH + LNH + LNL + CMD + PMID + PRMT + SUM + ETX)
    import time
    time.sleep(self.client.LONG_DELAY)

    rp = self.client._receive_data_packet()
    if (rp[3] & 0x7F) == 0x51 and rp[4] == 0x00:
        return True
    return False

def _transition_to_lck_boot(self) -> bool:
    # Use DLM state transition command (0x71)
    # Source: OEM (0x04), Target: LCK_BOOT (0x06)
    result = self.client._dlm_state_transition(0x04, 0x06)
    return result
```

8 Next Steps

The guidelines and examples provided in this document are intended to serve as a security-focused foundation for implementing a secure boot provisioning flow for RA8 devices. The primary emphasis throughout the document is on core secure boot mechanisms and on handling cryptographic material in a production-appropriate manner, specifically by ensuring that private keys are generated, stored, and used exclusively within an HSM.

Rather than addressing every aspect of factory-scale deployment, the material deliberately concentrates on:

- Establishing a correct and verifiable chain of trust
- Implementing secure boot and provisioning flows aligned with the RA8 architecture
- Avoiding manual or file-based private key handling
- Demonstrating how signing and authentication operations can be performed using HSM-backed keys
- Automating the provisioning process

While the core mechanisms and workflows are illustrated through concrete examples and code snippets, additional work is required to adapt and productionise the solution for a specific manufacturing environment.

Readers are expected to build on top of the material presented, tailoring the solution to their own requirements, constraints, and operational context. Typical next steps include, but are not limited to:

- Selecting an HSM that aligns with internal security policies and infrastructure
- Extending the provisioning tool to support batch programming and parallel device provisioning
- Adding production-grade error handling, logging, auditability, and traceability
- Integrating the provisioning flow into existing manufacturing or test automation systems
- Supporting additional device variants, lifecycle commands, or custom workflows

The examples provided in this document intentionally avoid solving factory-scale operational concerns in a prescriptive way. Production environments differ widely, and these aspects are best addressed by extending the secure core described here rather than by attempting a one-size-fits-all solution.

8.1 Full Example

To complement the guidelines presented in this document, a fully working command-line provisioning tool is provided as an example on GitHub. This tool implements the concepts discussed above and extends them with additional functionality, including a complete CLI interface and a more comprehensive end-to-end workflow. It is provided as an example and reference, not as a drop-in production solution.
A sequence diagram representing the flow implemented by the example tool is reported below, split into multiple images to ease the viewing experience.

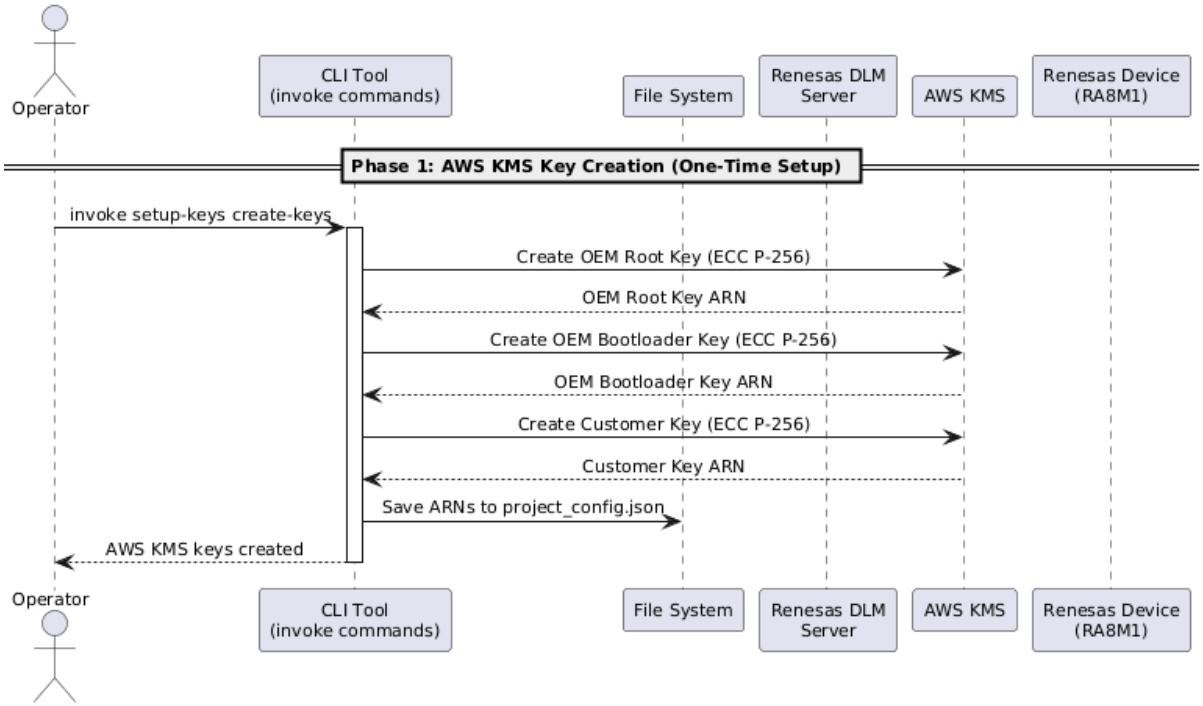


Figure 41 - CLI Tool example phase 1

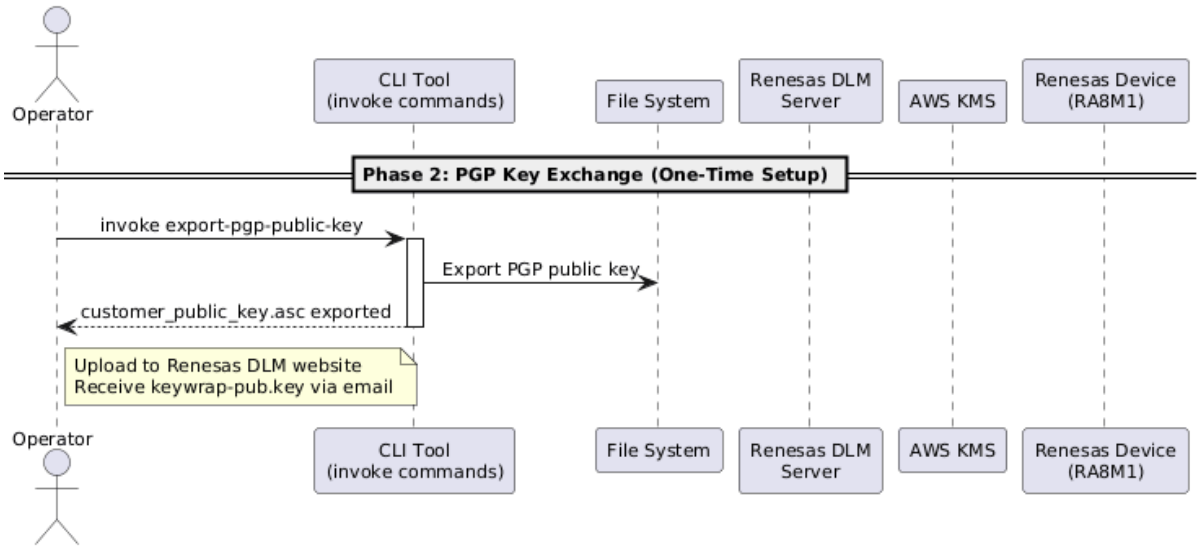


Figure 42 - CLI Tool example phase 2

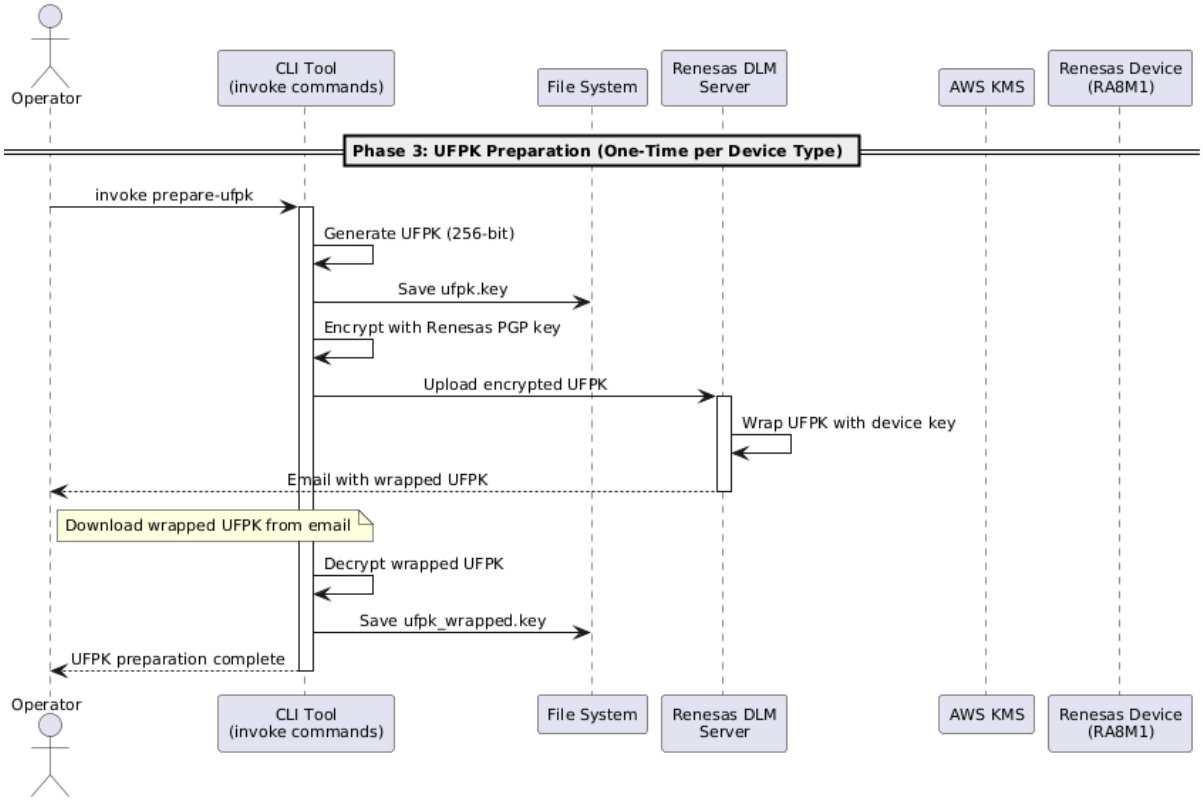


Figure 43 - CLI Tool example phase 3

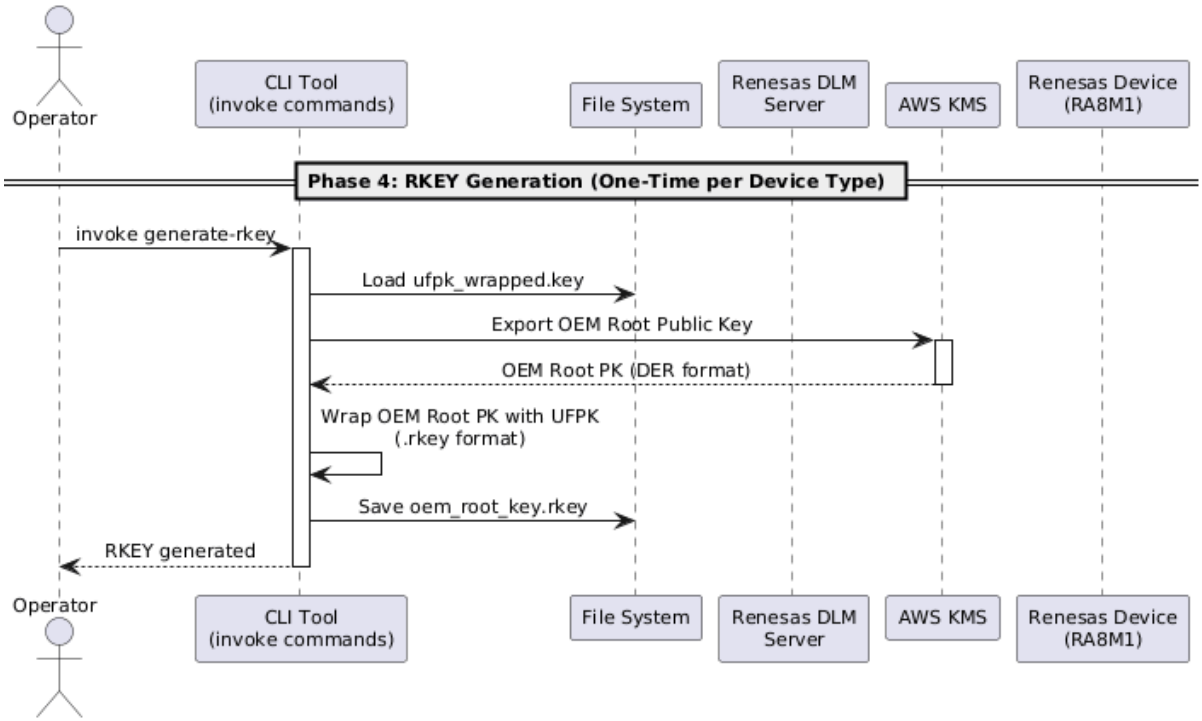


Figure 44 - CLI Tool example phase 4

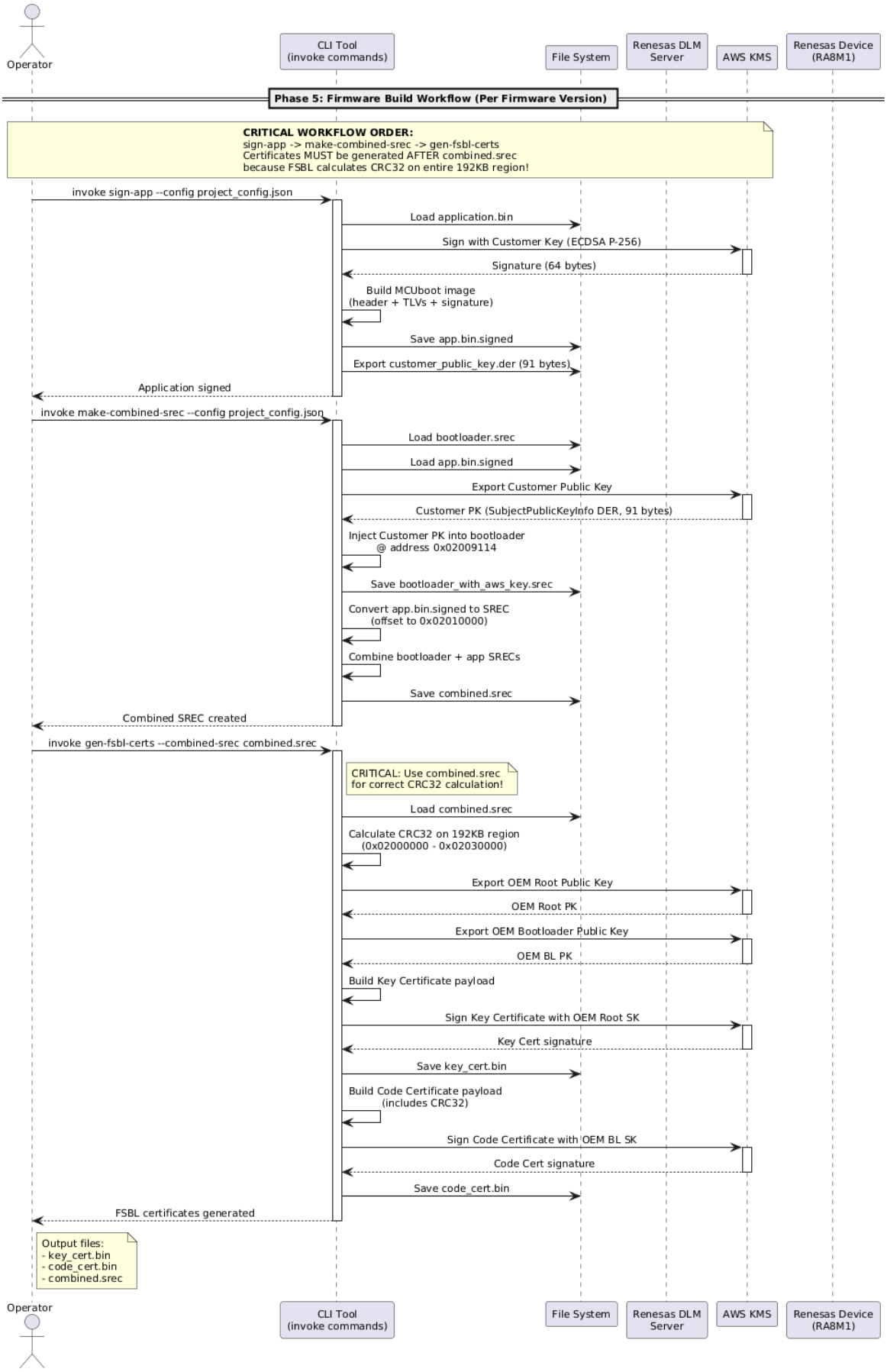


Figure 45 - CLI Tool example phase 5

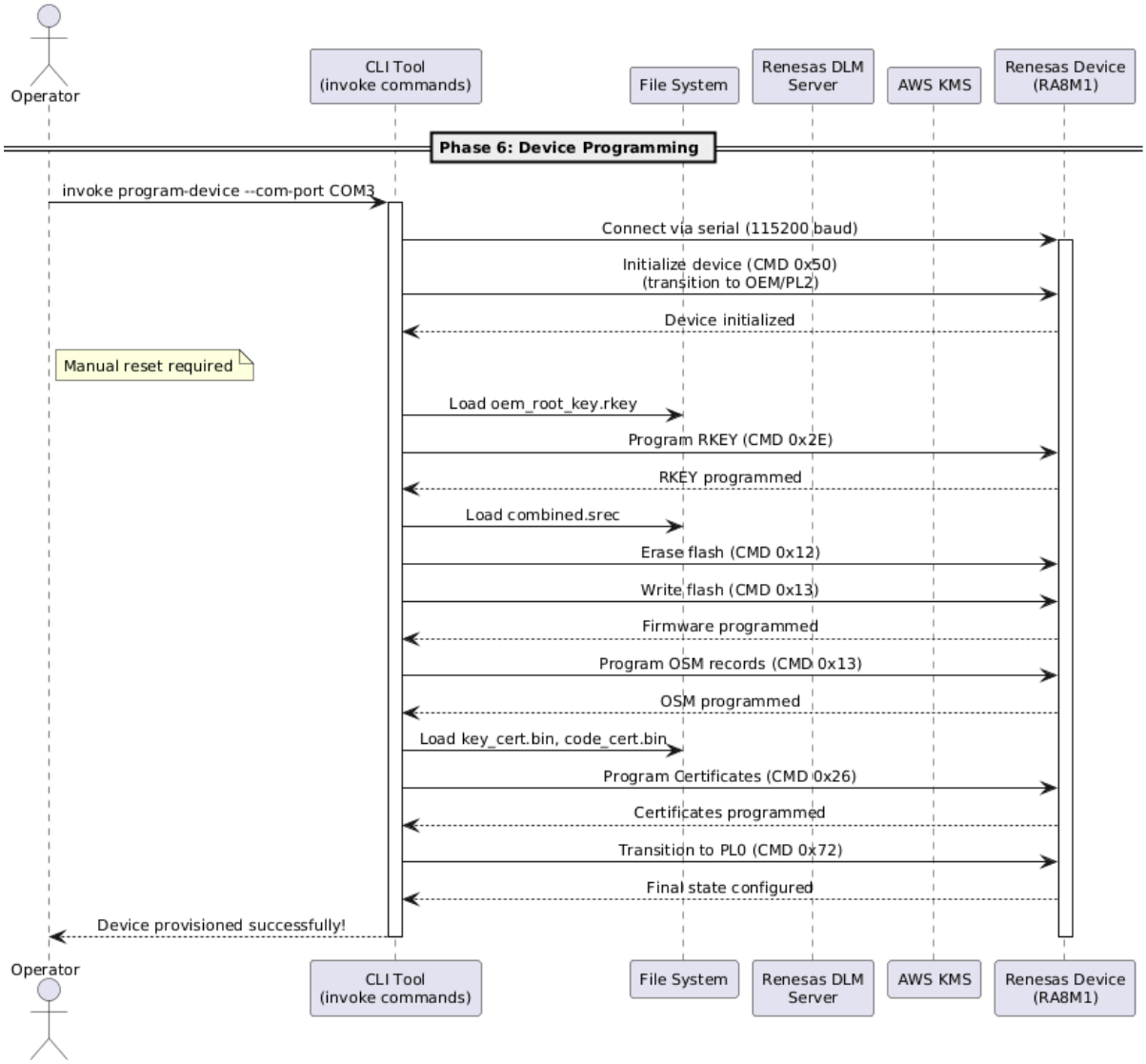


Figure 46 - CLI Tool example phase 6

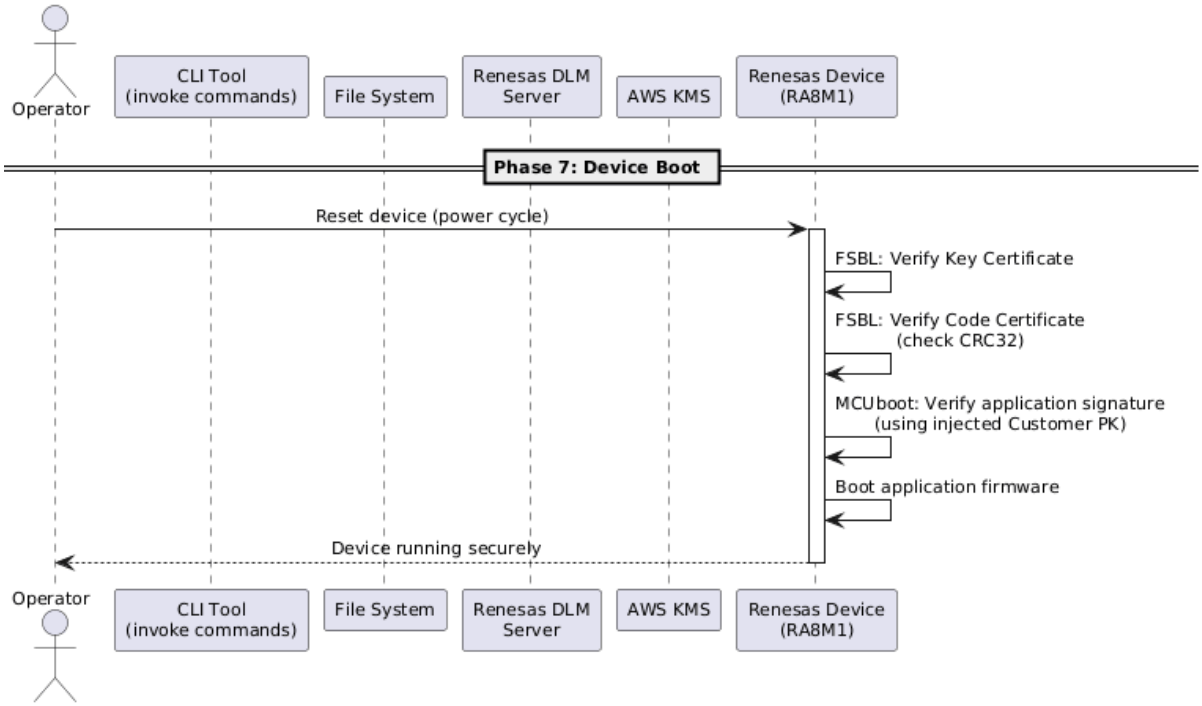


Figure 47 - CLI Tool example phase 7

9 Conclusions

This document has provided a detailed overview of the secure boot architecture on RA8 devices, clarifying how the chain of trust is established and enforced from the immutable ROM FSBL through the second-stage bootloader and into the application firmware. By breaking down the individual components and their interactions, the secure boot process has been demystified and placed into a concrete, implementable context.

Beyond architectural explanation, the document has focused on the practical aspects of **enabling secure boot in a production environment**. The workflows, design choices, and illustrative code snippets presented give readers the necessary building blocks to develop a provisioning solution that meets **production-level security** expectations, rather than relying on ad-hoc or development-oriented practices.

A key outcome of this work is the foundation it provides for **automation**. By consolidating signing, key handling, certificate generation, and device provisioning into a coherent flow, the material establishes a solid baseline for scaling the process to factory use, including batch programming and integration with manufacturing systems.

Central to this approach is the integration of a **Hardware Security Module (HSM)**. Ensuring that private keys are generated, stored, and used exclusively within an HSM significantly reduces the risk of key exposure and aligns the provisioning process with industry best practices. The document consistently emphasises this principle, demonstrating how secure boot can be enabled without ever placing sensitive key material in files or developer workstations.

It is important to note that, while the examples and guidelines provided are intentionally **focused on clarity and security fundamentals**, additional engineering effort is required to harden a provisioning tool for deployment in a production factory environment. This includes aspects such as comprehensive validation, error handling, traceability, throughput optimisation, and operational safeguards.

Taken together, the architectural explanations, security-focused design decisions, and implementation guidelines presented in this document form a robust starting point for OEMs seeking to deploy RA8 devices securely at scale. Readers are encouraged to build upon this foundation to create a provisioning solution tailored to their own production, security and scalability requirements.

United Kingdom

Unit 304-5,
164-180 Union Street,
London
SE1 0LH

General contact email:
info@cryptoquantique.com