Security User's Guide

M-20893-002
June 2023

© SCILLC, 2023
Previous Edition © 2023

Table of Contents

Page

Security User's Guide 1

Table of Contents 2

1. Introduction 7

1.1 Summary 7

1.2 Document Conventions 7

1.3 Further Reading 9

2. Overview of Security Features 10

2.1 Partitioning Between Secure and Non-Secure 10

2.2 Cryptographic Hardware Features Supporting Security 10

2.3 Bluetooth Specific Features 11

2.4 Additional Security Features 11

2.5 Security Mechanism Upon Boot 11

2.6 Side Channel Attacks 12

2.7 Operational States 12

3. Device and Life Cycle States 13

3.1 Device States 13

3.2 Life Cycle States 15

3.2.1 Root of Trust State LCS Properties and Transition Requirements 16

4. Energy Harvesting State (EH_STATE) 18

4.1 EH_STATE Features 18

4.2 Revoking EH_STATE 18

4.3 EH_STATE Security Features 18

4.3.1 Overview 18

4.3.2 Locking Process 19

4.3.3 Locking a Device 20

4.3.4 Setting the DCU Bits 21

4.3.5 Setting the SOC ID 21

4.3.6 Unlocking a Locked Device for Debug 21

4.3.6.1 Unlock 22

4.3.6.2 Relock 22

4.3.7 Revoking the EH_STATE 22

5. Secure Root of Trust State (ROT_STATE) 23

5.1 ROT_STATE Features 23

5.2 Chip Manufacture State (LCS_CM) 23

5.2.1 LCS_CM to LCS_DM Transition 24

5.3 Device Manufacture State (LCS_DM) 24

5.3.1 LCS_DM to LCS_SE Transition 24

5.4 Secure State (LCS_SE) 25

5.4.1 LCS_SE to LCS_RMA and LCS_DM to LCS_RMA 25

5.5 Return Merchandise Authorization State (LCS_RMA) 26

5.6 ROT_STATE Constraints 26

5.7 Roots Of Trust (RoTs) 26

5.7.1 Overview 26

5.7.2 Root of Trust: The Mechanics 27

5.7.3 Self-Signed Certificates 27

5.7.3.1 Types of Certificate 28

5.7.4 Certificate Chains 29

5.7.4.1 Authenticating an Application to be Executed 29

5.7.5 Roots of Trust Summary 29

5.8 Secure RoT Resources 30

5.8.1 NVM Storage 30

5.8.1.1 Additional Information 31

5.8.2 Secure Provisioning 32

5.8.3 Application Certificates and Configuration Items 32

5.8.3.1 Key Certificate Generation 32

5.8.3.2 Content Certificate Generation 32

5.8.3.3 Application Signing 33

5.8.4 Debug Certificates and SOC ID 33

5.9 Secure Boot Flow 35

5.9.1 Application Signing 35

5.10 Secure Debug Flow 37

5.11 Secure Applications 39

5.11.1 Application Signing 39

5.11.2 Application Configuration Block 39

5.11.3 Application Image 40

6. Debug Certificate Loading Process And Constraints 41

6.1 Reserving Space for the Application Configuration Block and Debug Certificates 41

6.2 Loading Debug Certificates to a Device 42

6.3 The Data Exchange Unit (DEU) 43

6.3.1 Data Exchange Flow 43

6.3.2 Data Exchange Unit Protocols 44

7. Security Tool Support 47

7.1 Getting Started 47

7.1.1 Overview 47

7.1.2 Software Installation 47

7.1.3 Hardware Set Up 47

7.1.4 RSLSec PC-Based Tool 48

7.1.4.1 RSLSec Common Options 49

7.2 EH State Configuration And Usage 50

7.2.1 Overview 50

7.2.2 Using LCS_EH Features in RSLSec 50

7.2.2.1 Updating a device 52

7.2.2.2 Unlocking a device 53

7.2.2.3 Relocking a device 53

7.2.2.4 RSLSec Command Examples for LCS_EH 53

7.3 RoT Secure Mode Configuration And Usage 55

7.3.1 Provisioning Process 56

7.3.2 Required Assets When Securing A Device 57

7.3.3 Initial Resource Creation 59

7.3.3.1 Key Generation and Usage 59

7.3.3.2 Asset Organization for Demonstration 59

7.3.3.3 Krtl 60

7.3.3.4 Root of Trust Key Pair 60

7.3.3.5 Provisioning and Code Encryption Keys 61

7.3.4 Provisioning CM to DM 62

7.3.5 Unlocking a Device Using Debug Certificates 64

7.3.5.1 Overview of Required Assets 64

7.3.5.2 Creation of Keys, Certificates and Unlocking 65

7.3.5.2.1 LCS_DM 65

7.3.5.2.2 LCS_SE 66

7.3.6 Provisioning DM to SE 67

7.3.7 Transition to LCS_RMA 71

7.3.8 Creating and Executing a Secure Application 72

7.3.8.1 Creating a Secure Application 72

7.3.8.2 Creating and Executing a Secure Application 73

7.3.8.3 Creating a Secure Application in LCS_DM using HBK0 74

7.3.8.4 Creating a Secure Application in LCS_SE using HBK1 75

7.4 Transition from EH_STATE to ROT_STATE 76

7.4.1 Revoking EH_STATE 76

CHAPTER 1

Introduction

1.1 Summary

IMPORTANT: onsemi plans to lead in replacing the terms “white list", "master" and "slave” as noted in this product release. We have a plan to work with other companies to identify an industry wide solution that can eradicate non-inclusive terminology but maintains the technical relationship of the original wording. Once new terminologies are agreed upon, we will update all documentation live on the website and in all future released documents.

This group of topics provides an overview of the security features available on RSL15, and how to use them. These topics will be of interest to anyone who wants to incorporate security into their RSL15-based applications and products. The security features have their roots in the Arm® TrustZone® CryptoCell™-312 Security IP, which is used in conjunction with the Arm Cortex®-M33 processor and the boot ROM.

RSL15 provides a security solution that is flexible in its deployment options, yet robust against attack when properly configured. This is supported by:

- The use of cryptographic techniques to form a cornerstone of the solution
- Utilizing industry standard techniques wherever possible, to achieve the required level of security and best practice when defining cryptographic assets
    - Any cryptographic algorithms are compliant with the relevant standards
- Extensible solutions with the potential to support future devices beyond RSL15
- A hardware-based True Random Number Generator (TRNG), given that a core consideration in any cryptographic operation is the availability of random numbers
- Hardware accelerators for standard cryptographic algorithms
- A secure storage capability for cryptographic keys and other assets
- The Root Of Trust provides:
    - a managed device life cycle
    - a Root of Trust (RoT) embedded in hardware
    - a secure boot facility to ensure that only verified and authenticated code can be executed
    - a secure debug facility to ensure that only authenticated users can access the system through the debug port
- Standard APIs to allow simpler firmware support
    - The API for using the hardware accelerators allows for a high level of software abstraction.

NOTE: For the purposes of this document, *device manufacturer* typically refers to an RSL15 user who would be building their own application, but in some circumstances the words can refer to other parts of the supply chain.

Depending on your needs you might wish to start with any of the main sections:

1. General overview material indicating the main features provided by the device
2. Theory of operation material detailing the modes and states of the system and how they work together to provide secure devices
3. Tool support section detailing how the provided tools can be used to perform the functions outlined in the theory section—including specific examples where appropriate

1.2 Document Conventions

The following typographical conventions are used in this documentation:

`monospace font`
: Assembly code, macros, functions, registers, defines and addresses.

italics
: File and path names, or any portion of them.

<angle brackets and bold>
: Optional parameters and placeholders for specific information. To use an optional parameter or replace a placeholder, specify the information within the brackets; do not include the brackets themselves.

Bold
: GUI items (text that can be seen on a screen).

Note, Important, Caution, Warning

Information requiring special notice is presented in several attention-grabbing formats depending on the consequences of ignoring the information:

NOTE: Significant supplemental information, hints, or tips.

IMPORTANT: Information that is more significant than a Note; intended to help you avoid frustration.

CAUTION: Information that can prevent you from damaging equipment or software.

WARNING: Information that can prevent harm to humans.

Registers:

Registers are shown in `monospace font` using their full descriptors, depending on which core the register is accessing. The full description takes the form **<PREFIX><GROUP>_<REGISTER>**.

All registers are accessible from the Arm Cortex-M33 processor.

A register prefix of `D_` is used in the following circumstances:

- In cases where there are multiple instances of a block of registers, the summary of the registers at the beginning of the Register section have slightly different names from the detailed register sections below that table. For example, the `DMA*_CFG0` registers are referred to as `DMA_CFG0` when we are defining bit-fields and settings.

The firmware provides access to these registers in two ways:

- In the flat header files (e.g.: *sk5_hw_flat_cid*.h*), each register is individually accessible by directly using the naming provided in this manual. This is helpful for assembly and low-level C programming.
- In the normal header files (e.g.: *sk5_hw_cid*.h*), each register group forms a structure, with the registers being defined as members within that structure. The structures defined by these header files provide access to registers under the naming conventions `PREFIX_GROUP->REGISTER` (for the structure) and `GROUP->REGISTER` (for the register).

- For more information, see the Hardware Definitions chapter of the *RSL15 Firmware Reference*.

Default settings for registers and bit fields are marked with an asterisk (*).

Any undefined bits must be written to 0, if they are written at all.

Numbers

In general, numbers are presented in decimal notation. In cases where hexadecimal or binary notation is more convenient, these numbers are identified by the prefixes "0x" and "0b" respectively. For example, the decimal number 123456 can also be represented as 0x1E240 or 0b11110001001000000.

Sample Rates

All sample rates specified are the final decimated sample rates, unless stated otherwise.

1.3 Further Reading

The following documents are installed with the RSL15 system, in the default location *C:/Users/***`<your_user_name>`***/AppData/Local/Arm/Packs/ONSemiconductor/RSL15/***`<version_number>`***/documentation*. These manuals are available only in PDF format:

- *Arm TrustZone CryptoCell-312 Software Developers Manual*
- multiple CEVA manuals in the */ceva* folder

For even more information, consult these publicly-available documents:

- *Armv8M Architecture Reference Manual* (PDF download available from https://developer.arm.com/documentation/ddi0553/latest).
- *Arm Cortex-M33 Processor Technical Reference Manual*, revision r1p0, from https://developer.arm.com/documentation/100230/0100
- *Bluetooth Core Specification version 5.2*, available from https://www.bluetooth.com/specifications/adopted-specifications
- TrustZone documentation available from the Arm website at https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m
- Other ArmCortex-M33 publications, available from the Arm website at https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33

For information about the Evaluation and Development Board Manual and its schematics, go to the RSL15 web page and navigate to the EVB page.

CHAPTER 2

Overview of Security Features

This topic introduces you to the security-related features designed into RSL15, particularly those features which are used to ensure that the device itself is operating in a secure manner. This includes procedures such as verifying that the firmware being executed is valid and authentic, and ensuring that access to the device is restricted to authorized users only.

Overviews of the other security features are provided in the topics that follow. For full details of the hardware-centric features, refer to the *RSL15 Hardware Reference*. For firmware features, refer to the *RSL15 Firmware Reference*. For detailed information on the secure bootloader sample application, see the *Secure Bootloader Usage* group of topics.

2.1 Partitioning Between Secure and Non-Secure

The Arm Cortex-M33 processor includes the Arm TrustZone for Cortex-M Devices technology, providing a separation between the secure and non-secure worlds. This allows memory and other resources to be configured for access by secure code, or by both secure and non-secure code. This separation allows for trusted and non-trusted execution environments, protecting sensitive memory areas or peripheral devices from being accessed incorrectly.

While the device initially restricts access of any memory or device resources to secure code only, a secure application can configure the device to allow specific items to be available to non-secure code. Additional information related to these features is provided in the Arm CryptoCell-312 Security IP *RSL15 Hardware Reference*, and in the*RSL15 Firmware Reference*, as well as the applicable third party documents listed in the onsemi HTML documentation under **More Information**.

2.2 Cryptographic Hardware Features Supporting Security

RSL15 provides the following which—while security-related—do not form part of the secure operation of the device but allow user applications access to security-related functionality:

- Hardware accelerators that provide support for cryptographic operations, including secure data storage, transmission, and authentication of wired transmissions
- The APIs to the cryptography features explained in the *Arm TrustZone CryptoCell-312 Software Developers Manual*
- Sample code, which is provided for many of the cryptographic operations.

IMPORTANT: The `CC312` functions, when using hardware accelerators, use a WFI instruction while waiting for the hardware accelerator to complete its processing. Therefore, these functions must not be called in the context of any interrupt routines that have a priority higher than, or equal to, the `CC312` interrupt.

The Arm CryptoCell-312 security IP provides facilities to support the following features:

- Support for a True Random number Generator (TRNG)
  - There are two independent TRNG mechanisms defined in RSL15, each of which meets different industry standards. An application needs to use whichever Arm Cryptocell-312 library (*.a* file) corresponds to its required TRNG mode.
- Symmetric and asymmetric cryptography, including support for the following algorithms:
  - AES
  - SRP
  - SHA
  - CCM

  - GCM
  - RSA
  - ECDSA
  - ECDH
  - ECIES
  - CTRDRBG
  - CHACHA
  - MAC
  - DHM
  - Key derivation
- Device life cycle state management
- Root of Trust (RoT) access policy enforced by hardware mechanisms
  - An RoT ownership model allows for multiple distinct trust anchors.

2.3 Bluetooth Specific Features

Features supporting security and privacy of Bluetooth communications are implemented as defined by the Bluetooth Specification, and are described in the Bluetooth Low Energy Baseband section of the *RSL15 Hardware Reference*and the related third party GAP documentation. This is because the Bluetooth stack is responsible for securing communications, which is separate from the support for securing the device as described in this topic.

2.4 Additional Security Features

- Key and asset provisioning, management, and isolation, across different device operational scenarios
- Support for secure debug facilities, which can be validated and authenticated via the hardware RoT

2.5 Security Mechanism Upon Boot

In addition to the basic hardware facilities, RSL15 also holds the secure boot and secure debug implementation in ROM to prevent corruption, guaranteeing that the RoT is verified on boot.

The secure keys and assets required to manage the secure boot process are held in NVM memory in a manner that ensures any corruption is detected during the power-up sequence.

Any corruption detected in the NVM renders the device invalid and prevents any application code from executing, while also restricting access to the device from the debug port.

There are three boot paths:

1. Cold boot: The system is coming up from full reset and Root of Trust is evaluated if needed.
2. Warm boot: The system is coming up from a wake state, but the Root of Trust has already been evaluated, so the system does not re-evaluate it. This is sometimes referred to as wakeup from flash. For a successful warm boot without retention (`ACS_VDDRET_CTRL_VDDCRET_DISABLE` and/or `ACS_VDDRET_CTRL_VDDMRET_DISABLE` field set in the `ACS_VDDRET_CTRL`), make sure that the Arm CryptoCell-312 is always enabled, which is the default.
3. Custom boot: The system is waking up from sleep and uses a RAM vector to restart. Again, the Root of trust has already been evaluated so this is not redone.

Therefore, each time the Life Cycle State changes, a cold boot (full reset) is required afterward for it to take effect. For more information on the boot paths themselves, refer to the ROM Initialization Sequence section of the *RSL15 Firmware Reference*

A secure bootloader should be used to perform software image validation and authentication during system boot and on updates. An example is provided in the RSL15 sample application collection and is described in the *Secure Bootloader Usage* guide.

2.6 Side Channel Attacks

The main item that is not specifically addressed in the security implementation is that of device hardening against side channel attacks. While there is provision in the hardware to detect and handle corruption of sensitive data, there is no specific protection against Differential Power Analysis (DPA)-type attacks.

2.7 Operational States

RSL15 can operate in a number of distinct states governed by the contents of internal status information stored in non-volatile memory.

A device manufacturer can choose how they wish the device to operate: in the energy harvesting state, which is a low-power less-secure state; or in the Root of Trust fully-secured state.

These two states have certain trade-offs which need to be accounted for in any design.

- Energy Harvesting State (EH_STATE)
  - Fast power-up times
  - Lower power operation
  - Security hardware can be disabled if not required
  - Ability to secure the device debug port using a 128-bit key
  - Easier to configure and use
  - Less secure implementation, but good enough for some applications
- RoT Secure State (ROT_STATE)
  - Longer power-up times as more work needs to be done to validate and authenticate the firmware being executed
  - Cryptographically secure Root of Trust embedded in hardware
  - Potentially two independent Roots of Trust are available.
  - Managed life cycle ensures that the devices, once secured, are protected.
  - Manufacturing requires more configuration, and potentially more provisioning infrastructure
  - Care must be taken to manage keys and certificates appropriately.

If you choose to release a product in energy harvesting state, take care to lock the device in that state. For RoT state, follow the proper flow to lock the device in that state. The EH_STATE locking procedure is described in Section 4.3.2 “Locking Process” on page 19, and the RoT locking procedure is described in Section 5.8.2 “Secure Provisioning” on page 32. Further information about releasing a device in EH_STATE is described in Section 3.1 “Device States” on page 13.

CHAPTER 3

Device and Life Cycle States

This documentation uses the concepts of device state and life cycle state (LCS) throughout.The short forms for the device states use of a suffix of *_STATE, as in EH_STATE. The LCS acronym and these abbreviations are used for convenience in the documentation and typically do not refer to anything in any provided code. The meanings of the terms are described in detail in the document, but the "Device States and Life Cycle States" figure (Figure 1) gives an overview of their use and how the device states and life cycle states relate to each other.

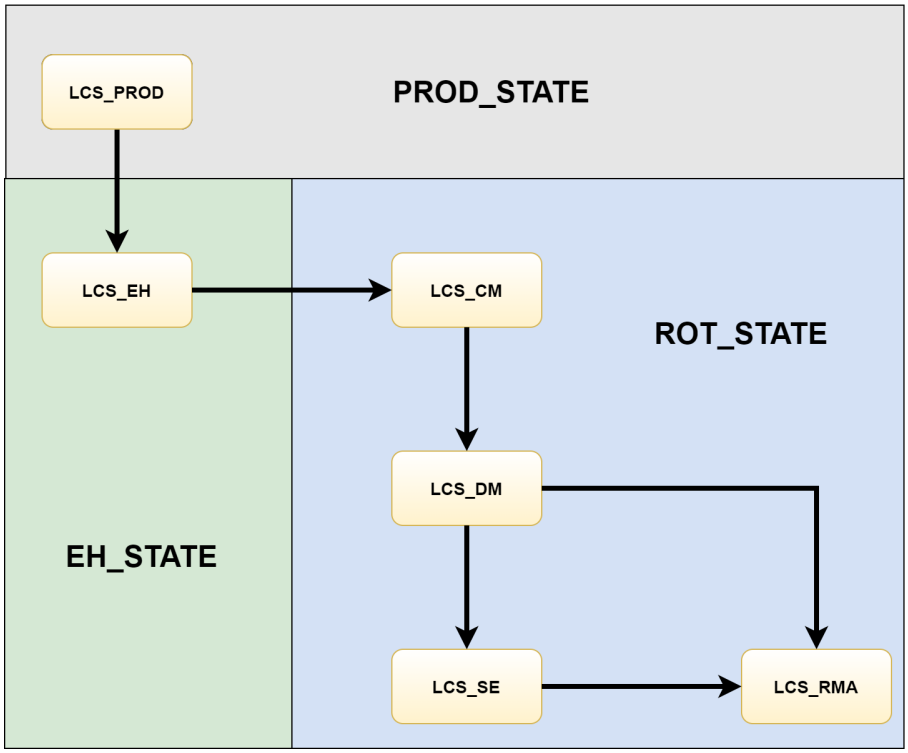

Figure 1. Device States and Life Cycle States

As a reminder, for the purposes of this document, "device manufacturer" typically refers to an RSL15 user who would be building their own application, but in some circumstances the words can refer to other parts of the supply chain.

3.1 Device States

RSL15 allows a device to be deployed in one of two states of operation:

- A lower power, less secure Energy Harvesting state (abbreviated to EH_STATE)
- A secure state, which ensures strong validation and authentication using one or more Roots of Trust (RoT) embedded in hardware (abbreviated to ROT_STATE).

A third state is employed during device production, but this is disabled prior to a device leaving the manufacturing process (abbreviated to PROD_STATE).

In each state, there is strict control over how the states can be changed; only a subset of transitions are allowed. This is shown in the "Flow Through Possible Device States" figure (Figure 2)

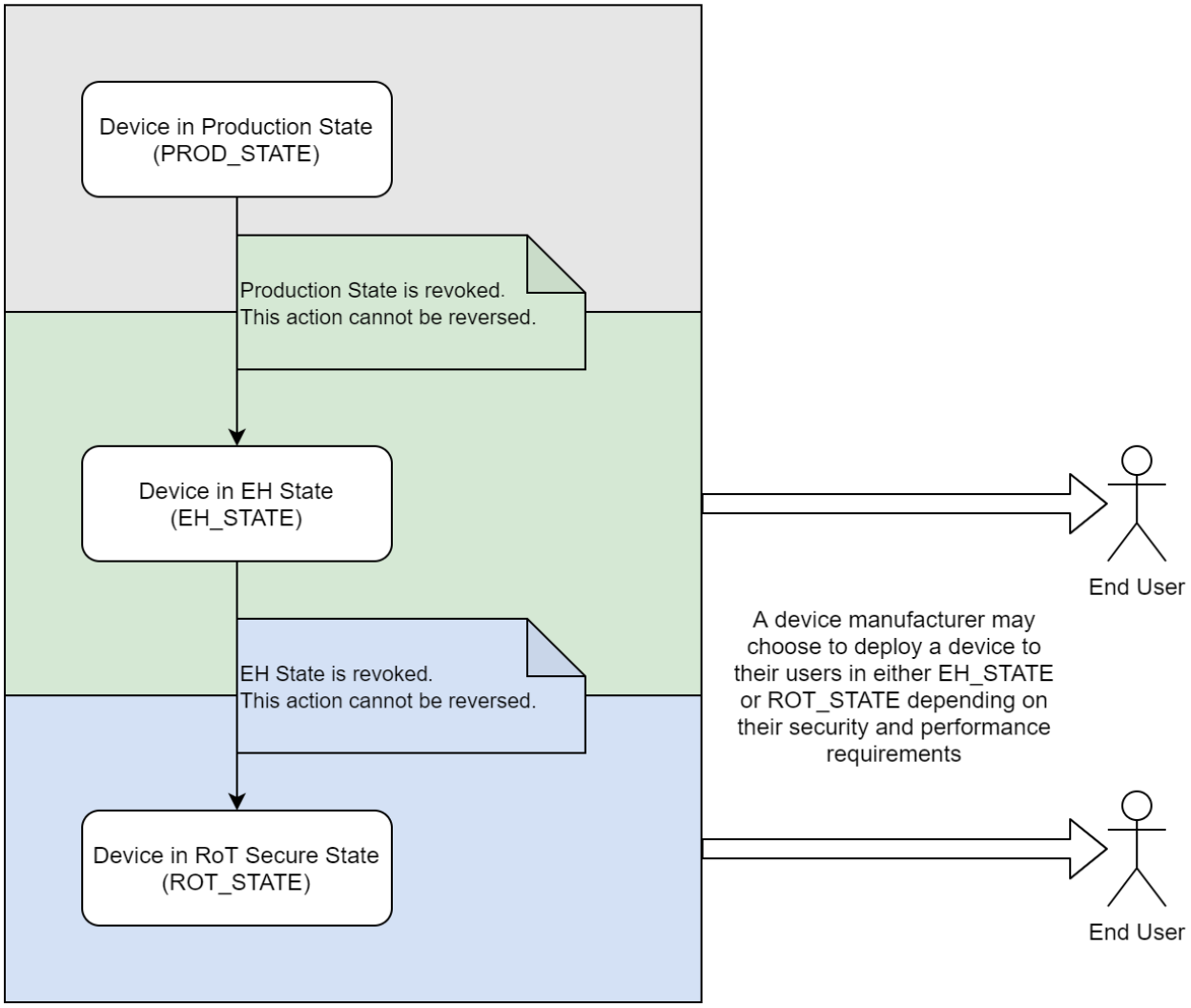

Figure 2. Flow Through Possible Device States

If a device manufacturer wishes to deploy devices in EH_STATE when locked, they need to safeguard against the EH_STATE becoming corrupted and the device automatically transitioning to an unsecured ROT_STATE.

- In the default configuration, the ROT_STATE is unsecured and awaiting device provisioning.
- This means that the debug port are open, and any IP held in flash may be compromised.

To counteract this situation, if locked devices are employed in EH_STATE, the NVM contents used by the ROT_STATE must be set to values which ensure that the device powers up secured. This ensures that any unexpected transition to the ROT_STATE results in the device starting up in LCS_SE, which locks the ports by default. More about how to do this is explained later in this document, but one way is to intentionally write invalid data to the signature words section in NVM.

3.2 Life Cycle States

In addition to the device states, there is also a managed life cycle operating in conjunction with those states, to further refine how a device can be configured.

A device always exists in a single life cycle state (LCS) dependent on the contents of the device NVM and the internal status.

A device can be in one of the following life cycle states:

- LCS_PROD: The Production LCS corresponds directly to the PROD_STATE.
- LCS_EH: The Energy Harvesting LCS corresponds directly to the EH_STATE.
- LCS_CM: Chip Manufacture is the initial LCS in ROT_STATE. The device is unsecured in this state.
- LCS_DM: Device Manufacture is the state in which the chip is partially provisioned and the device is secured.
- LCS_SE: Secure state is the state in which the chip is fully provisioned, and also the state in which devices are normally released to the public.
- LCS_RMA: Return Merchandise Authorization (return to manufacturer) is designed for devices being returned in an unsecured state. All secret information has been wiped.

All life cycle transitions are governed by the contents of the NVM, enforced by the ROM. All life cycle transitions occur during a power-on reset of the device.

Life cycle changes can occur in accordance with the "Flow Through Valid Life Cycle States" figure (Figure 3).

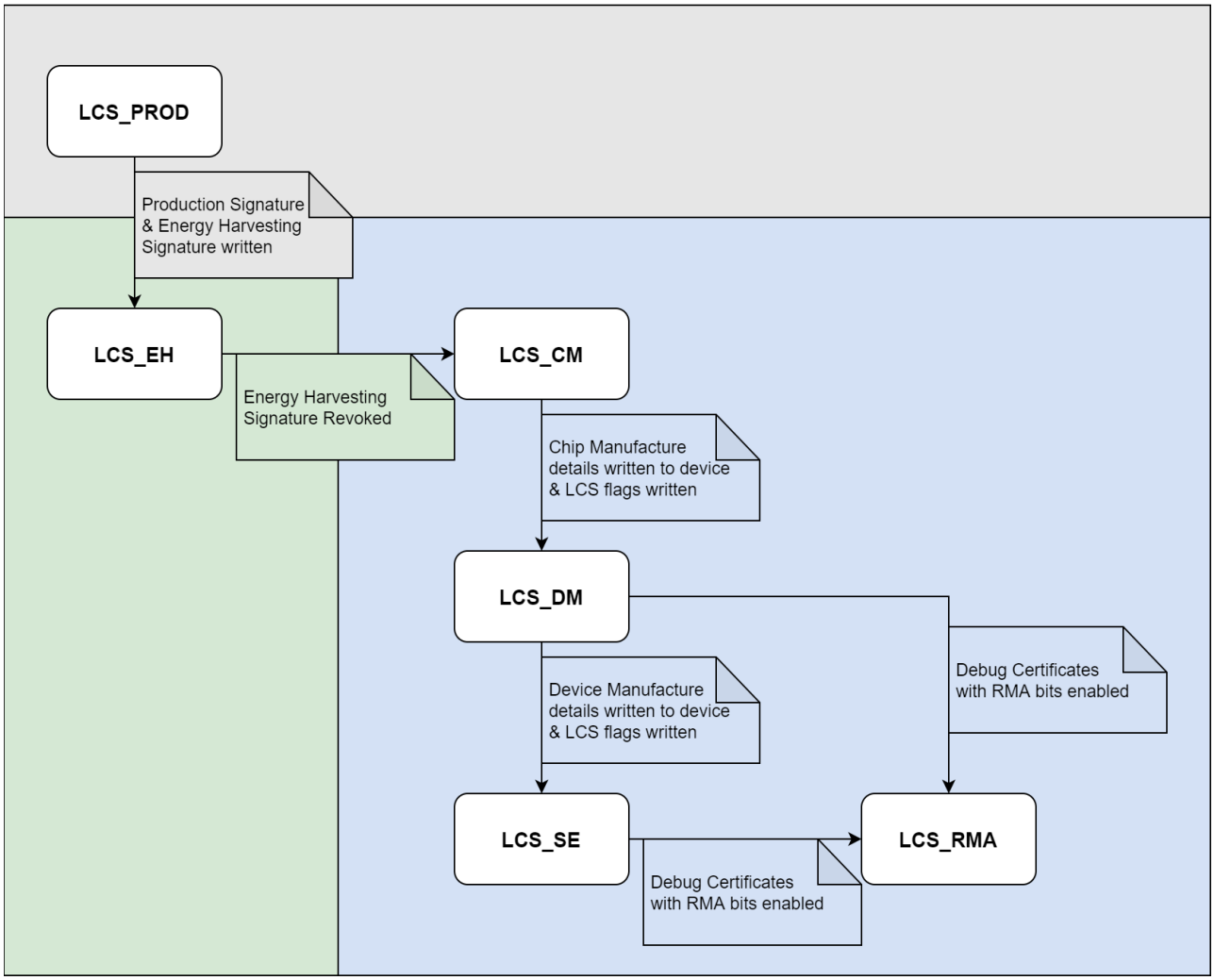

Figure 3. Flow Through Valid Life Cycle States

NOTE: Life Cycle State transitions are one-way only.

The rest of this section focuses on those life cycle states which form part of ROT_STATE.

3.2.1 Root of Trust State LCS Properties and Transition Requirements

Life cycle management is controlled by a combination of hardware components and the secure boot firmware embedded in ROM. The life cycle state is governed by the contents of the NVM memory, which is interpreted during system power-up.

The life cycle state is determined as one of the defined states, provided the contents of the NVM can be validated. If any corruption in the NVM memory is detected, the device reverts to a failure mode with the debug port locked, and makes no attempt to execute any application firmware.

As indicated in the "Flow Through Valid Life Cycle States" figure (Figure 3), ROT_STATE has paths for four valid LCSs and four valid LCS transitions. Any other attempted LCS transitions, including attempts to reverse an LCS transition, constitutes an error condition and results in the device being locked.

In all life cycle states, any application firmware must be properly configured for execution in the system. This means the following items must be available:

- The application context structure must be set up to provide details of the key and content certificates describing the application.
- One or two key certificates providing the trust chain must be defined and packaged with the application.
- A content certificate describing the application and allowing it to be authenticated must be provided.
- The application itself must be signed in accordance with the content certificate.

These items are described in more detail later, and the process for creating and combining them into a loadable image is also explained.

For now it is important to understand that the items detailed above are used to verify that an application has not been corrupted, and that it can be authenticated against a known identity.

CHAPTER 4

Energy Harvesting State (EH_STATE)

As previously indicated, the EH_STATE is provided to support very low power systems which may require fast startup times.

This state is the default for any chips coming out of manufacturing, and provides very limited security features, such as debug port locking.

4.1 EH_STATE Features

- EH_STATE allows for fast device startup by not employing the hardware Root of Trust in the system.
- The boot process bypasses the application authentication steps, performing only cursory checks on the application before trying to execute it.
- This state also allows some level of debug port locking; the user can specify a 128-bit key that is stored in NVM.
    - This key, once present, is used by the ROM to enforce the debug port locking functionality.
    - Once a device is locked, the debug port can only be accessed by the provision of the key through the Data Exchange Unit (DEU).
- The state also allows for a specific SOC ID to be written to each device, which can be read back at a later stage by the DEU if required.
    - This enables device manufacturers to identify their devices uniquely if needed.
    - The SOC ID in EH mode is a 32-bit value. For more detail about the SOC ID, see Section 4.3.5 “Setting the SOC ID” on page 21.

4.2 Revoking EH_STATE

As this is the default state for devices delivered from manufacturing, a mechanism is provided to allow devices to be switched to the RoT Secure state (ROT_STATE).

This is a one time operation. Once the state has been revoked it cannot be re-enabled, because the act of revoking it corrupts the EH signature in NVM, which physically cannot be undone. The device remains in ROT_STATE.

4.3 EH_STATE Security Features

4.3.1 Overview

Devices are delivered from manufacturing in the Energy Harvesting state (EH_STATE) by default, and hence also in the corresponding life cycle state LCS_EH.

When in LCS_EH, it is possible to implement a lightweight security model which allows the device to be locked, preventing unauthorized access to the code or stored secrets.

In order to lock a device in LCS_EH, a device must be provided with a key and the debug control unit (DCU) lock bit values. These two pieces of information combine to ensure only the required debug facilities are allowed in normal operation. At the same time, it still allows the device to be unlocked, which is achieved by providing the correct key through a defined port.

It is also possible to transition a device from LCS_EH to the more secure life cycle states by revoking the state.

When in LCS_EH, the following facilities are available:

1. Configuration Updates
    - Provide a key to the device allowing debug access when the device is locked.
        - This is a 128-bit value, which can be unique to each device.
    - Write the DCU values that specify the features available when the device is locked.
        - Refer to the Arm CryptoCell-312 registers in the *RSL15 Hardware Reference* for the specific registers, bit positions, and values.
    - Write a SOC ID to the device; this ID can be used later for identification.
        - This is a 32-bit value, which can be unique to each device or made common for a family of devices.
2. Revocation
    - Revoke the ability to use the LCS_EH, transitioning the device to the Secure life cycle model.

4.3.2 Locking Process

The flow chart in the "Locking Decision Process in Energy Harvesting State" figure (Figure 4) describes the process employed when deciding how to lock a device in EH_STATE.

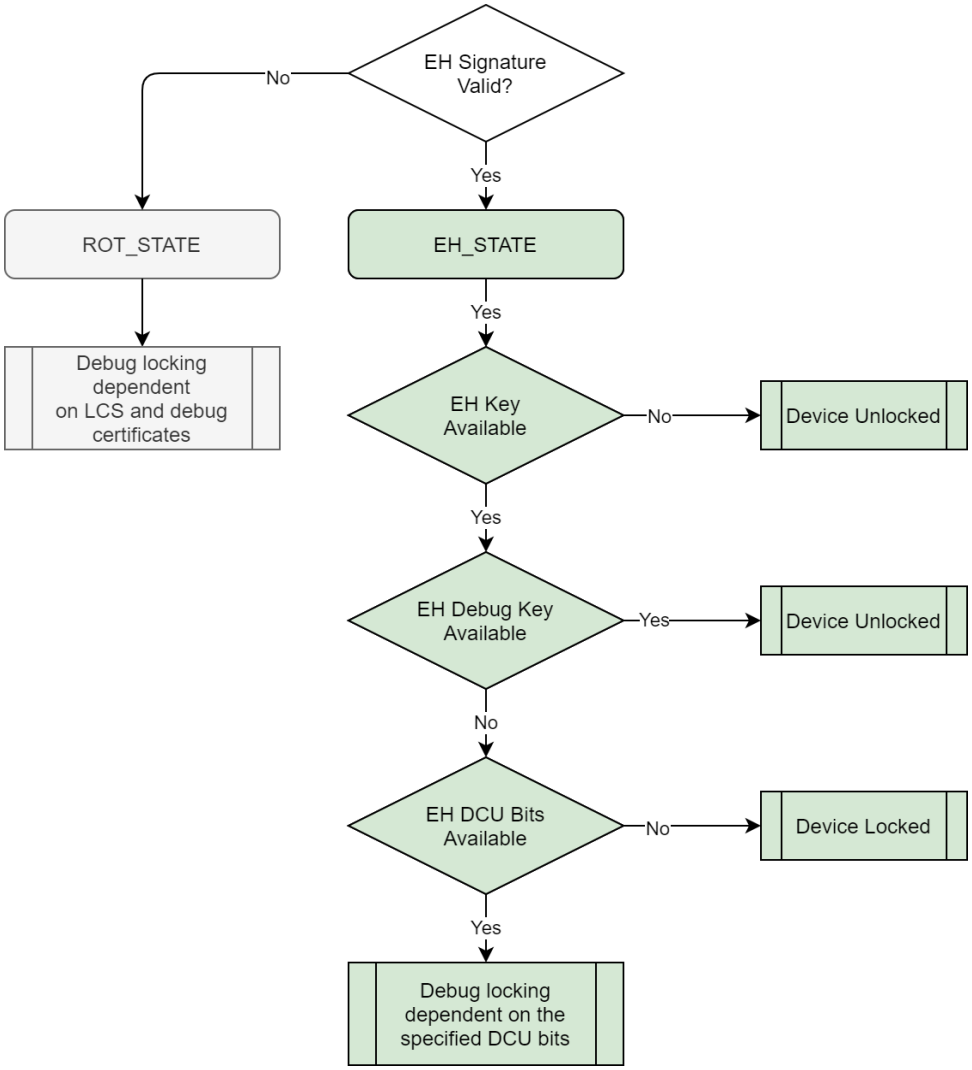

Figure 4. Locking Decision Process in Energy Harvesting State

4.3.3 Locking a Device

A device is locked by ensuring that there is no access to the debug port. This is performed by the ROM during a system reset process and cannot be changed until the next system reset.

The ROM uses the contents of a specific part of the NVM to determine the state of a device. A signature word and consistency bits are used to identify whether a device is supposed to be in EH_STATE.

When in EH_STATE, the device is unlocked by default. To lock the device, a 128-bit key value must be provided, which is then stored in NVM along with some consistency bits.

On a cold boot, the ROM examines the signature word. If the device is found to be in EH_STATE, the ROM then looks for a key. If found, the key is used to determine the lock state of the device.

Properties of the key value:

- The key is a 128-bit value.
- It is not used in any cryptographic operations.
- It can be a designated value or a random sequence of bits that are known to the manufacturer.
- These bits are stored with associated consistency bits so that corruption can be detected.
- Anyone with the key can unlock the debug port of the device.
- The key can be the same value across a range of devices, or can be unique to each device. This is up to the device manufacturer to decide.

4.3.4 Setting the DCU Bits

To provide some finer control to the device locking process, it is possible to set individual features in the DCU to be locked or unlocked. For more details of the DCU feature, refer to the *RSL15 Hardware Reference*.

Due to the nature of the NVM, it is necessary to invert the DCU bits when writing to NVM. As such, we generally refer to nDCU bits.

Properties of the DCU bits:

- This is a 128-bit value, with the bits inverted.
- The inverted bits are written to the DCU registers, controlling the debug port access and other DCU-controlled features.
- These bits are stored with associated consistency bits so that corruption can be detected.

4.3.5 Setting the SOC ID

RSL15 provides a concept of SOC ID for EH_STATE. This is a 32-bit value which is designed to be unique for each device; however, it is an option for the SOC ID to be the same for a family of devices.

The SOC ID is designed specifically to allow a device to be identified when the debug port is locked. It is up to the device manufacturer to determine how to use it for their particular use cases.

The SOC ID can only be written when the device is unlocked. Once a device is locked, the SOC ID is readable via the data exchange unit.

By providing this feature it is possible for a device manufacturer to maintain a mapping between SOC ID and the key value written to a device. This can be used to ensure that a single key being compromised does not affect all devices.

4.3.6 Unlocking a Locked Device for Debug

Once a device is ready for release, the lock key is written and the device is deployed to the field with the debug port locked.

In this scenario, it is often the case that a bug or some unexpected behavior is found after deployment, and a locked device needs to be debugged. Therefore, some mechanism is needed to allow debug of a locked device.

This is a very similar process to that required when running in the ROT_STATE, so similar techniques can be used.

RSL15 provides the concept of a Data Exchange Unit (DEU), which allows some very limited interaction with the device even when the debug port is locked. Using this mechanism, it is possible to introduce a debug key to the device and then have that key control the state of the debug port. Once the debugging session has completed, the key can be erased and the device returns to the previous locked state.

To unlock the device for debugging, three pieces of information are required:

- The key to unlock the device. This must match the key stored in the device NVM; otherwise, the unlock request is ignored.
- The DCU values to be written when the device is being unlocked. This makes it possible to only unlock specific features. Refer to the *RSL15 Hardware Reference* for more information.
- The DCU Lock values to be written. This allows some DCU bits to be left unlocked after the boot process. Normally the DCU is locked on completion of the boot process, and application code cannot change it.

Each of these three items consist of 128-bit values.

4.3.6.1 Unlock

In general, the process of unlocking the device using the debug port consists of the following:

- Connect to the DEU, which causes a system reset allowing the ROM to manage the DEU interactions.
- Introduce the Key/DCU/Lock values to the devices; these are stored in RAM.
- Using RSLSec as described in Chapter 7 "Security Tool Support" on page 47 so that they can survive a system reset.
- Disconnect from the DEU; this causes another system reset, and the ROM interprets the flash contents and acts accordingly.

4.3.6.2 Relock

Erasing the debug key is very similar to unlocking the debug port using the unlock steps:

- Connect to the DEU which causes a system reset allowing the ROM to manage the DEU interactions.
- Erase the values in flash.
- Disconnect from the DEU; this causes another system reset, and the ROM interprets the flash contents and acts accordingly.

4.3.7 Revoking the EH_STATE

To transition from the EH_STATE to the ROT_STATE, it is necessary to invalidate the signature words held in NVM. As this is defined as one-time-programmable, it is a permanent state change that cannot be recovered from.

The NVM holds the following four items with corruption-detecting bits:

- Energy Harvesting Signature
- Lock Key
- nDCU Bits
- SOC ID

By invalidating one or all of these words, the device reverts to ROT_STATE and then follows the secure boot process. Once invalidated, the bits are unrecoverable, so this is a one-way transition.

CHAPTER 5

Secure Root of Trust State (ROT_STATE)

The ROT_STATE is provided to support applications where the security of the system needs to be much stronger than in EH_STATE.

5.1 ROT_STATE Features

The ROT_STATE provides:

- A **managed life cycle** for devices, ensuring that only specific functionality is available in each life cycle state (LCS)
    - Some features, such as debug access, are turned off by default in certain life cycle states.
    - If a feature is required in a specific LCS that has been disabled, it can be granted using cryptographically secure certificates.
- A secure boot facility whereby any firmware being executed by the ROM must be cryptographically verified and authenticated
- Secure debug facilities where the debug port can only be enabled via the use of cryptographically secure certificates
- A secure storage area for assets that are used to control the LCS behavior, and includes such items as:
    -
    - The RoT hash, provisioning key, and code encryption key are specific to each RoT in the system.
    - RSL15 supports two distinct Roots of Trust.
    - The Root of Trust (RoT) hash value, which allows for certificate authentication ($H_{BK0}/_1$)
    - The storage of provisioning keys, to allow secure assets to be introduced to the system (K_{picv}/K_{cp})
    - The storage of code encryption keys, allowing code to be decrypted from flash to RAM during the ROM startup (K_{ceicv}/K_{ce})
    - The Hardware Unique Key (HUK), which is an identifier unique to each device
    - The SOC ID is an externally visible identifier that can be used to uniquely identify the device. This is a 128-bit value derived from other properties of the device. The SOC ID can be used in two different ways, depending on the device state.
- A secure mechanism to introduce debug certificates to the system, and the ability to revoke their use
- Anti-rollback measures to ensure older software cannot be executed on the device

The ROT_STATE cannot be revoked once a device has been transitioned into it. The device cannot be reverted back to EH_STATE.

5.2 Chip Manufacture State (LCS_CM)

LCS_CM is defined as the chip manufacture state, and is the default state when the NVM is empty. In this state, the debug port is open and firmware may be loaded to the board. There is no Root of Trust programmed in the system, so applications can not yet be authenticated against a known identity. However, it is possible to ensure that they are internally consistent and have not been corrupted.

This state has features very similar to the EH_STATE/LCS_EH, but has no facility to lock the debug port yet. It is not intended for delivery to end users; it is for performing initial application debugging and testing, and allows initial provisioning of the Initial Chip Vendor (ICV) Root of Trust.

The act of provisioning the ICV (Initial Chip Vendor) data establishes the first Root of Trust in the system.

5.2.1 LCS_CM to LCS_DM Transition

To transition a device from LCS_CM to LCS_DM, specific information needs to be created and written to the NVM. Once this data is written, the device powers up in the device manufacture state (LCS_DM), when the device is reset.

Once the device is transitioned to LCS_DM, no further provisioning of the LCS_CM information is possible.

The following data items are required when provisioning the ICV data:

- H_{BK0}: This is a 128-bit hash value generated by truncating to use the top 128 bits of a SHA-256 signature of an RSA public key.
    - The RSA public key is used to authenticate the first key certificate in any authentication operation.
    - The first key certificate is signed by the RSA private key corresponding to the public key.
    - This use of an asymmetric key pair allows authentication of the key certificate provider as the owner of the key pair.
- Kpicv: This is a 128-bit AES key used when introducing secure assets to the device.
    - A secure asset is any unit of data that has been encrypted using the AES key and is being provided for storage on the device in a secure manner.
- Kceicv: This is a 128-bit AES key, used when encrypting application data that needs to be decrypted to RAM prior to execution.
    - RSL15 provides a mechanism to allow secure components of the system to be stored encrypted in flash, and then decrypted during the secure boot process.

In addition to these items, additional flag information and consistency bits are written to the NVM during the provisioning process.

The hardware unique key (HUK) is also calculated at this time, and stored in the NVM.

More details regarding these items are provided in Section 7.3.4 “Provisioning CM to DM” on page 62.

5.3 Device Manufacture State (LCS_DM)

LCS_DM is defined as the device manufacture state, and is the only valid state after LCS_CM. In this state, the debug port is locked by default, and debug certificates must be provided to open the debug port. The use of debug certificates is explained in Section 5.8 “Secure RoT Resources” on page 30.

It is expected in LCS_DM that the ICV Root of Trust has been programmed in the system. In this state, any application firmware being executed as part of the secure boot process must be validated and authenticated against the H_{BK0} identity. This state is not intended for delivery to end users. It is provided for performing initial application debugging and testing, and to allow initial provisioning of the Original Equipment Manufacturer (OEM) Root of Trust.

The act of provisioning the OEM data establishes the second Root of Trust in the system.

5.3.1 LCS_DM to LCS_SE Transition

This transition is very similar to the LCS_CM to LCS_DM transition and requires very similar data to be provided. In order to transition a device from LCS_DM to LCS_SE, specific information needs to be created and written to the NVM. Once this is done, the device is reset and powers up in LCS_SE. When the device has transitioned to LCS_SE, no further provisioning of the LCS_DM information is possible.

The following data items are required when provisioning the OEM data:

- H_{BK1}: This is a 128-bit hash value generated by truncating a SHA-256 signature of an RSA public key.
  - The RSA public key is used to authenticate the first key certificate in any authentication operations.
  - The first key certificate is signed by the RSA private key corresponding to the public key.
  - This use of an asymmetric key pair allows authentication of the key certificate provider as the owner of the key pair.
- Kcp: This is a 128-bit AES key, used when introducing secure assets to the device.
  - A secure asset is any unit of data that has been encrypted using the AES key and is being provided for storage on the device in a secure manner.
- Kce: This is a 128-bit AES key, used when encrypting application data that needs to be decrypted to RAM prior to execution.
  - RSL15 provides a mechanism to allow secure components of the system to be stored encrypted in flash,and then decrypted during the secure boot process.

In addition to these items, more flag information and consistency bits are written to the NVM during the provisioning process.

5.4 Secure State (LCS_SE)

The LCS_SE or secure state is the expected state for devices being delivered to final customers. In this LCS, all debug port is locked by default and application firmware must be authenticated prior to execution. Verification or authentication failures result in the device entering a failure state and no application being executed.

The only valid life cycle transition from this state is to LCS_RMA, which requires authentication from both the ICV and OEM RoTs.

5.4.1 LCS_SE to LCS_RMA and LCS_DM to LCS_RMA

The transition to LCS_RMA state follows the same process, whether it originates from LCS_DM or LCS_SE. This transition is handled after the devices have been manufactured, so the process is different from the provisioning processes outlined for previous LCS transitions. When transitioning to LCS_RMA, the expectation is that the device has an issue and must be returned to the manufacturer for fault analysis.

The device could be coming from the secure state, so its firmware might not be able to be updated; in such a case, some other mechanism needs to be employed to manage the transition. Since the debug port is likely to be locked, the transition can be handled via the introduction of special debug certificates that force the LCS transition instead of unlocking the device.

The return to RMA state requires acknowledgement from both possible RoTs in the system, so two debug certificates need to be loaded: one for the ICV RoT and one for the OEM RoT.

This transition is a multi-stage process:

1. RMA Debug Enable certificates need to be generated for both ICV and OEM RoTs. These each contain a reference to the current LCS which must match the device LCS.
2. For each RMA Debug Enable certificate, a new RMA Debug Developer certificate needs to be created.
3. The OEM RMA Debug certificate needs to be loaded to the device, and the device then reset to allow the ROM to process the certificate.
4. The ICV RMA Debug certificate needs to be loaded to the device, and the device then reset to allow the ROM to process the certificate.
5. At this stage the device will boot into LCS_RMA on future resets.

5.5 Return Merchandise Authorization State (LCS_RMA)

In the RMA LCS, the device is destined to be returned to the manufacturer.

- This state is irrevocable; the device must not be used when in this state.
- This state allows application firmware to be executed if it can be verified and authenticated via one of the RoTs.
- Any secret keys stored in the NVM have been erased, so it is not possible to perform any application decryption or use any secured assets.
- Debug port is opened to allow fault analysis by the manufacturer.

If a device manufacturer wishes to ensure that the application code cannot be executed in RMA, the device certificates which form the RoT verification must be erased.

If a device manufacturer wishes to ensure that the application code cannot be read, the flash memory must be erased once the device has transitioned to the RMA state.

5.6 ROT_STATE Constraints

Because of the nature of the ROT_STATE design, there are several constraints that need to be considered:

- The authentication and verification of firmware executing on the device is a fairly complex process, and as such, slows down the system's initial boot time.
    - This does not affect wakeup time from sleep modes.
    - Only signed code can be executed by the ROM; if the verification or authentication fails, the ROM enters a failure state.
- The Secure Boot features are embedded in the ROM and hardware of the system; they can only be changed on a cold reset of the device.
    - Therefore, it is not possible to start a debug session on an active device if the debug certificates have not already been loaded.
    - Life cycle state changes can only occur on a cold reset of the system.
- The managed life cycle model ensures that once a life cycle transition has been made, it cannot be reversed.
- The managed life cycle model includes a Return to Manufacture (RMA) state. This requires authorization from all Roots of Trust in the system.
- Corruption of the LCS configuration data causes the device to revert to a locked state, where there is no access from the debug port and firmware on the device cannot be executed.
    - This is an expected outcome from the security implementation.
    - Management of the life cycle states using the included tools ensures that the configuration data is maintained consistently.

5.7 Roots Of Trust (RoTs)

5.7.1 Overview

RSL15 provides facilities to support two independent hardware Roots of Trust (RoTs).

A Root of Trust provides a secure mechanism to verify that code being executed on the device has been verified and authenticated.

- Verification involves checking that the firmware image being executed has not been corrupted. It uses a cryptographic signature to verify that the code is correct.
- Authentication involves checking that the firmware image has been provided by a trusted party, by ensuring that the key used to sign the image is actually owned by the firmware provider.

  - The authentication process uses asymmetric encryption where the public key is known to the device and the private key is used to sign the image.
  - In this way, the system can guarantee that the image has been signed by someone authorized to sign the firmware, and that the image has not been tampered with or corrupted.

In RSL15, the information required to enable this process is held in hardware and in the secure processing element.

- The ROM manages the boot process and ensures that an application has been signed and authenticated using assets stored in secure storage on the device.
- The ROM is immutable, hence the secure boot process can be trusted to verify the application firmware.
- The keys and other assets used to enable this process are held in NVM memory in a secure fashion, with anti-tamper detection to ensure consistency. This is achieved by the inability to overwrite ones with zeros in this area of NVM which behaves like OTP. Therefore, counting the zeros in the protected areas combined with accessing only through the secure IP block, provides anti-tamper protection.

Having the ROM manage the boot process, with immutable keys and assets ensures that the RoT is embedded in the hardware of the device. As indicated, RSL15 maintains two sets of RoT keys and assets allowing for two independent Roots of Trust. These can be enabled and used as required.

Since the specific parameters controlling the RoT are stored in NVM, these must be provisioned as part of the initial device setup. It is possible to use a single set of RoT parameters across a family of devices. Alternatively, unique parameters can be allocated to each individual device. The device manufacturer decides this based on their specific use cases.

5.7.2 Root of Trust: The Mechanics

In its simplest terms, the Root of Trust consists of a known protocol to follow when authenticating information, combined with some token used for proving authenticity.

This protocol is implemented by the device's ROM and the token is stored in NVM memory—which is configured such that corruptions can be detected. This provides a high level of security for the initial application, which can then perform further validations if required.

The system uses asymmetric cryptography for the tokens, storing a representation of an RSA public key on the device. Specifically, a 128-bit hash of the public key is stored in the NVM, together with metadata to ensure that corruptions can be detected.

This public key hash is then used to verify and authenticate certificates that accompany the firmware application. The use of certificates to introduce key information is common in security, and the processes employed in ROM follow standard techniques.

5.7.3 Self-Signed Certificates

The device uses self-signed certificates based on an RSA key hash to authenticate and, if necessary, unencrypt assets. Such an asset could be a binary application for the device to execute, or a debug certificate provided to unlock debug functionality.

A self-signed certificate consists of four distinct parts:

- A header defining the type of the certificate, with any additional metadata the certificate provides to allow it to be understood
- A public key, used to verify the signature of the certificate

- A certificate data section containingthe information that the certificate is delivering
- A signature of the certificate, signed using a secure private key that corresponds to the public key above

By knowing the public key, it is possible to verify that the contents of the certificate have not been corrupted, and that the certificate is internally consistent.

Authenticity can be proven by verifying that the hash of the public key matches some known data. In the simplest form of a single certificate, the public key hash is compared against the hash stored in the RoT. (See the "Self-Signed Certificate Verification" figure (Figure 5).)

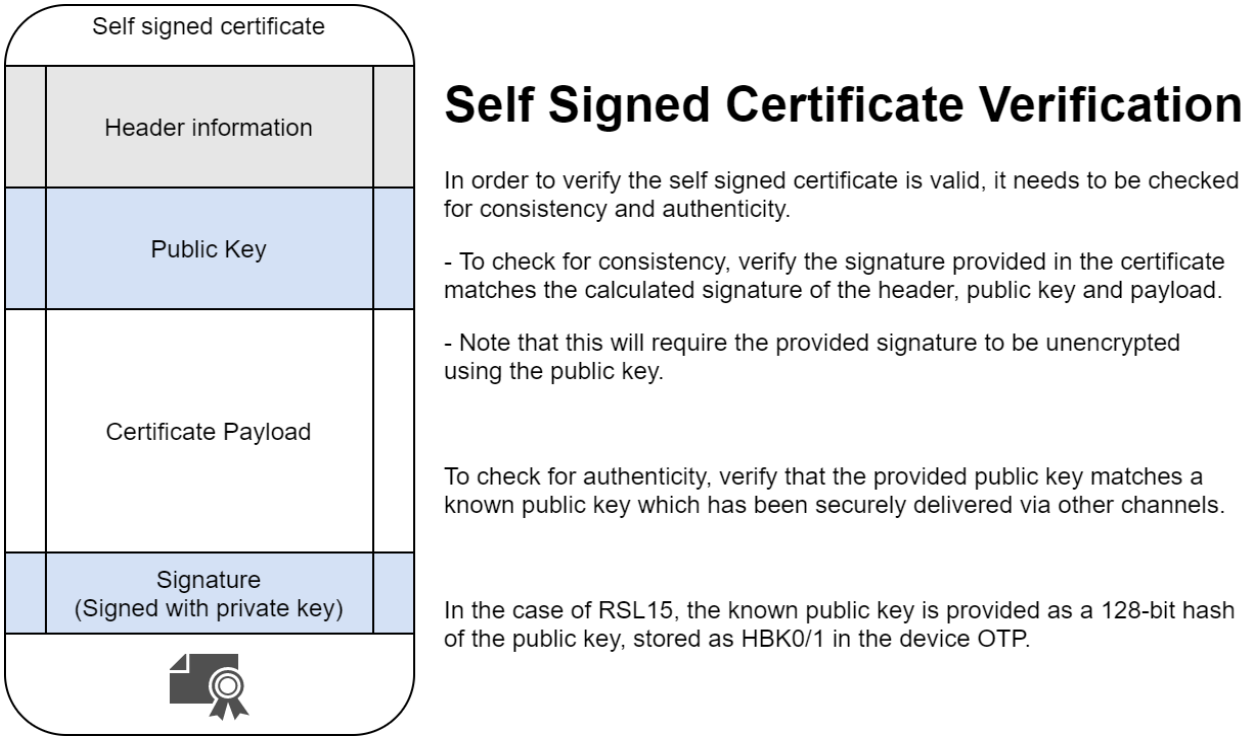

Figure 5. Self-Signed Certificate Verification

5.7.3.1 Types of Certificate

There are four types of certificate used as part of the secure boot process, each one providing a different function:

- Key certificates are used to build certificate chains, by providing a payload consisting of the hash of the public key contained in the next certificate.
- Content certificates are used to identify the secure parts of an image to be executed by the Boot ROM. These are validated and, if necessary, decrypted.
- Debug enabler certificates are provided by the owner of the Root of Trust—this could be the device manufacturer or chip manufacturer—and are used to specify the precise debug features that can be enabled by a developer. These settings can be applied to a range of devices.
- Debug certificates are used by developers to enable the allowed debug features on a specific device.

5.7.4 Certificate Chains

It is possible to chain certificates so that the first certificate provides information allowing subsequent certificates to be authenticated. This provides a flexible way of changing keys if one key becomes compromised.

In the ROM protocol, two or three certificates can be used in any authentication operation.

5.7.4.1 Authenticating an Application to be Executed

When an application is loaded into the device for execution by the Boot ROM, the application must be packaged with at least two certificates: a key certificate and a content certificate (see the "Two-Certificate Authentication" figure (Figure 6)). The certificate chain can also be extended by using two key certificates and a content certificate (see the "Three-Certificate Authentication" figure (Figure 7)).

The ROM processes each certificate in order and the application is executed only if all verification and authentication steps are valid.

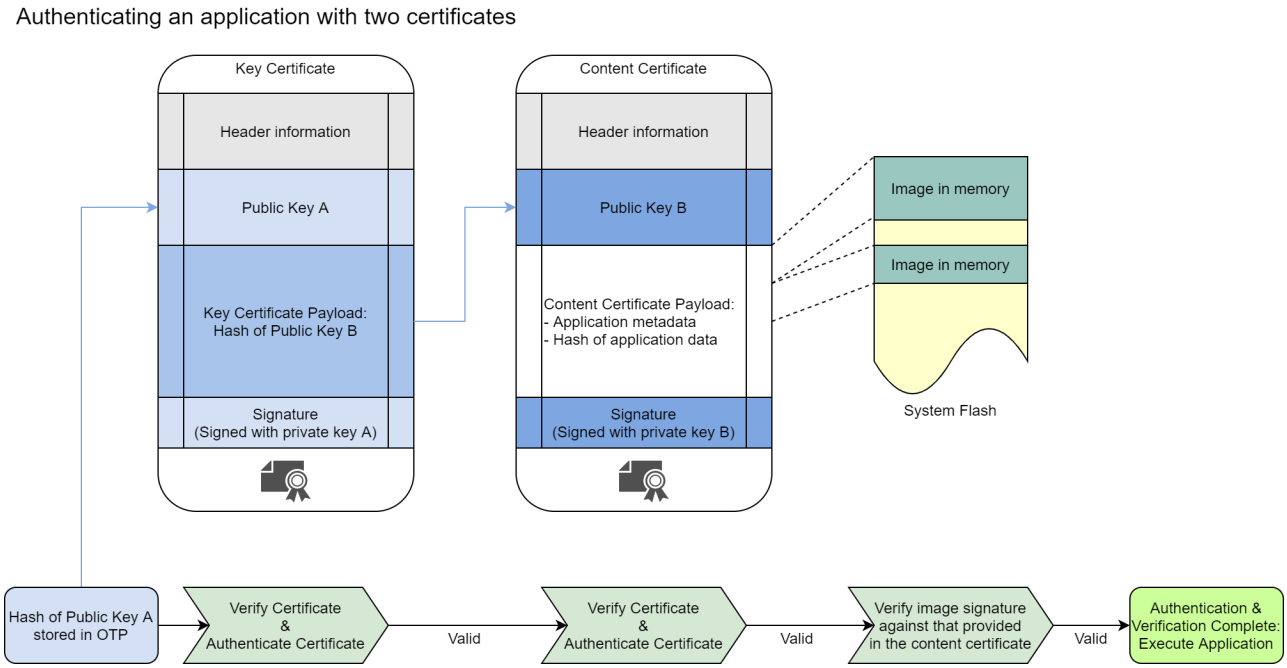

Figure 6. Two-Certificate Authentication

Figure 7. Three-Certificate Authentication

5.7.5 Roots of Trust Summary

The establishment of a secure Root of Trust relies on the device having the public key associated with the first asymmetric key pair. As the public key itself is quite large, a truncated hash of the public key is stored on the device NVM; this is then used to authenticate the first key certificate in a chain of certificates.

Each certificate can be verified in isolation, as the public key forms part of the certificate contents; therefore, the information is available to check that the certificate is valid. Each key certificate allows the authentication of the next certificate in the chain, using the hashed public key. This ensures that the final application has been verified as original, and has been provided by an authenticated party.

IMPORTANT: The private keys must be kept strictly secure.

5.8 Secure RoT Resources

Certain configuration items are required for supporting the correct operation of the Secure RoT.

These are split into three broad categories, as follows:

- Configuration items held in secure NVM storage
- Certificates and other configuration items supporting secure boot
- Certificates required to support secure debug

Understanding how these items relate to each other, and what is needed at which stage of the product life cycle, is important to your implementation of the security system.

5.8.1 NVM Storage

The NVM is protected by the Arm CryptoCell-312 security block; only certain parts of the NVM can be read or written at any given point in the device life cycle.

This is normally managed during device provisioning either from LCS_CM → LCS_DM, or from LCS_DM → LCS_SE.

The "Asset Storage in NVM" table (Table 1) describes the assets that are stored in NVM.

Table 1. Asset Storage in NVM

Item name	Size (bits)	Written By	Description
Hardware Unique Key (HUK)	256	ICV	The HUK is a unique 256-bit value calculated during device provisioning. It is never directly visible to application code or provided outside of the device, but can be used to derive other unique keys.
K_{picv}	128	ICV	This is the ICV provisioning key, used to encrypt assets that need to be decrypted securely on the device. It is a 128-bit AES key provided by the ICV.
K_{ceicv}	128	ICV	This is the ICV code encryption key. It is used to decrypt any code tha is stored encrypted and must be decrypted prior to use. This is a 128-bit AES key provided by the ICV.
ICV Flags	32	ICV	These are flag bits used to indicate if the ICV provisioning process has completed.

Table 1. Asset Storage in NVM (Continued)

H_{BK0}	128	ICV	This is a 128-bit hash value of the public key associated with the ICV. This is used to determine the ICV Root of Trust for any key certificate provided to the system. The hash value is a 128-bit truncated SHA-256 calculation of the public key.
H_{BK1}	128	OEM	This is a 128-bit hash value of the public key associated with the OEM. This is used to determine the OEM Root of Trust for any key certificate provided to the system. The hash value is a 128-bit truncated SHA-256 calculation of the public key.
K_{cp}	128	OEM	This is the OEM provisioning key, used to encrypt assets that need to be decrypted securely on the device. It is a 128-bit AES key provided by the OEM.
K_{ce}	128	OEM	This is the OEM code encryption key. It is used to decrypt any code that is stored encrypted and must be decrypted prior to use. This is a 128-bit AES key provided by the OEM.
OEM Flags	32	OEM	These flag bits are used to indicate if the OEM provisioning process has completed.
H_{BK0} Minimum Version	64	ICV	This contains 64 bits, which can be used to provide anti-rollback protection for the ICV RoT. This is initialized during ICV provisioning and then updated when newer firmware is loaded to the device by the ICV.
H_{BK1} Minimum Version	96	OEM	This contains 96 bits which can be used to provide anti-rollback protection for the OEM RoT. This is initialized during OEM provisioning and then updated when newer firmware is loaded to the device by the OEM.
Configuration Flags	32	ICV	These ICV configuration flags are set up during ICV provisioning.
DCU Lock Mask	128	ICV/OEM	The specific user-supplied DCU lock bits are set according to ICV or OEM provisioning. The first 64 bits are owned by the ICV. The second 64 bits are owned by the OEM.

5.8.1.1 Additional Information

For both the ICV and OEM provisioning operations, the following items are required:

1. RSA Public/Private key pair
   - This forms the basis of the secure boot and secure debug processes.
   - The truncated SHA-256 of the public key is stored in the $H_{BK0}/_1$ locations in the NVM.
   - The private key is used to sign the first key certificate in the secure chain of trust.
   - This RSA key pair belongs to either the ICV or OEM, and the private key must be kept secure.
2. A 128-bit AES provisioning key
   - This is required if any secure asset provisioning is performed.

3. A 128-bit AES code encryption key
    - This is required if there is any code decrypted as part of the secure boot process.
    - If all code is unencrypted and executing from flash, this may be omitted.

The RSLSec tool, which is explained in detail in a later part of the document, can be used to generate all of these keys and hashes if required.

5.8.2 Secure Provisioning

A user may need to secure the keys and hashes before loading them to the device. For instance, if a third party is performing the provisioning of the device, it is necessary to provision it without revealing the keys to the third party. In this case, the RTL key (K_{rtl}) can be used to securely wrap the various assets before provisioning them.

This K_{rtl} is a common key embedded in the hardware of the device and is never visible to application firmware. This is the same key in all RSL15 devices.

IMPORTANT: This key value can be provided by onsemi in an encrypted form on request.

The K_{rtl} is only used when provisioning the devices. After provisioning, the K_{rtl} is no longer necessary when using the device.

5.8.3 Application Certificates and Configuration Items

In order to configure an application for use, it must be securely signed, and appropriate key and content certificates must be provided.

RSL15 uses a proprietary format for the certificates in the Root of Trust. These are similar in content to standard X.509 certificates, but are in a more compact form, with only the core information.

5.8.3.1 Key Certificate Generation

A key certificate securely provides a public key hash for the next certificate in the chain. The next certificate might be another key certificate, or a content certificate describing an application; in either case, the process is the same.

To create a new key certificate, the following items are required:

1. The RSA key pair to sign the key certificate in privacy-enhanced mail (PEM) format
    - If the private key is encrypted, the password of the PEM file is also needed.
2. The public key of the next certificate in the chain of trust, again in PEM format

5.8.3.2 Content Certificate Generation

A content certificate describes an application image. It defines the areas of program memory that must be verified and authenticated, and defines whether any of the application is encrypted and needs to be loaded to RAM prior to execution.

To create a new content certificate where the application, or part of the application, is stored unencrypted and executes from flash memory, the following items are required:

1. The RSA key pair to sign the key certificate in PEM format
    - If the private key is encrypted, the password of the PEM file is also needed.
2. The image that is executed when the content certificate has been verified and authenticated

To create a new content certificate where the application is stored encrypted in flash and needs to be decrypted into RAM for execution, the following items are required:

1. The RSA key pair to sign the key certificate in PEM format
    - If the private key is encrypted, the password of the PEM file is also needed.
2. The image that is executed when the content certificate has been verified and authenticated
3. The AES key used for performing the encryption operation, with the password if required
4. A mapping table to allow the translation of flash addresses to RAM, used when decrypting and loading the image

Execution of encrypted images in this way is possible, but less practical due to the relative flash and RAM sizes. Generally, the application is stored in plain text in flash and executed directly from there.

5.8.3.3 Application Signing

A secure application can be thought of as having the following components:

- A configuration block which can be used to locate the various certificates associated with the RoT
- One or two key certificates which are used to authenticate the application
- A single content certificate which is used to verify and authenticate the image being loaded
- The application image itself

All of these component parts are combined into a single load module which can be stored in the flash of the device.

The configuration block holds the base addresses of each of the key and content certificates and for the purposes of RSL15 is always stored at a fixed location in flash. The certificates and application may reside anywhere in the free flash memory.

5.8.4 Debug Certificates and SOC ID

In order to unlock the debug features of a device, another form of debug certificate must be presented to the ROM during system power-up.

Again, the debug certificate is verified using the RoT associated with the certificate. This makes it possible to have debug certificates with different properties provided by different owners of the RoT.

There are two forms of debug certificate used in the system, and one of each is required for enabling the debug facilities.

- Debug enabler certificate
    - This certificate is specified for a distinct RoT owner, and must match the RoT owner that is specified in the first key certificate establishing the RoT.
    - The debug enabler certificate is provided by the owner of the RoT and identifies the specific debug features that can be opened by a developer.
        - Debug facilities are controlled by the debug control unit (DCU), and specific bits are set in the debug mask to enable those features during debug.
        - The debug enabler certificate also allows the owner of the RoT to lock specific bits of the DCU so that they cannot be re-opened after the boot process has completed. This ensures that any items the ICV or OEM do not want to provide access to can be locked out regardless of the developer’s wishes.

        - Debug enabler certificates are also associated with a specific LCS, and they limit the effect of the certificate to devices which are that specific LCS. This allows an enable certificate to be safely issued for LCS_DM, as it cannot be used to open a secure device.
    - The debug enabler certificate is also used to specify the transition to RMA for a given device.
    - This is a general certificate, applying to a range of devices that have the same RoT, so a single enabler certificate can be used with multiple discrete devices.

- Debug developer certificate
    - The debug developer certificate is used to open up the debug port on a specific device.
    - It references an enabler certificate, and can open up DCU bits which have not been locked by the enabler certificate.
        - This allows only a subset of the features provided by the enabler certificate to be used by a developer.
        - Different enabler certificates can be provided to different developers, giving them different limitations of access.
    - The debug developer certificate references a device by its SOC ID.
        - This is a unique value for a given device, which can be retrieved from the device to create the certificate.

As an example of the relationship between the enabler and developer certificates, consider a device that is secured, with the debug port locked.

The "Unlocking the Debug Port on a Secure Device" figure (Figure 8) demonstrates the unlocking process:

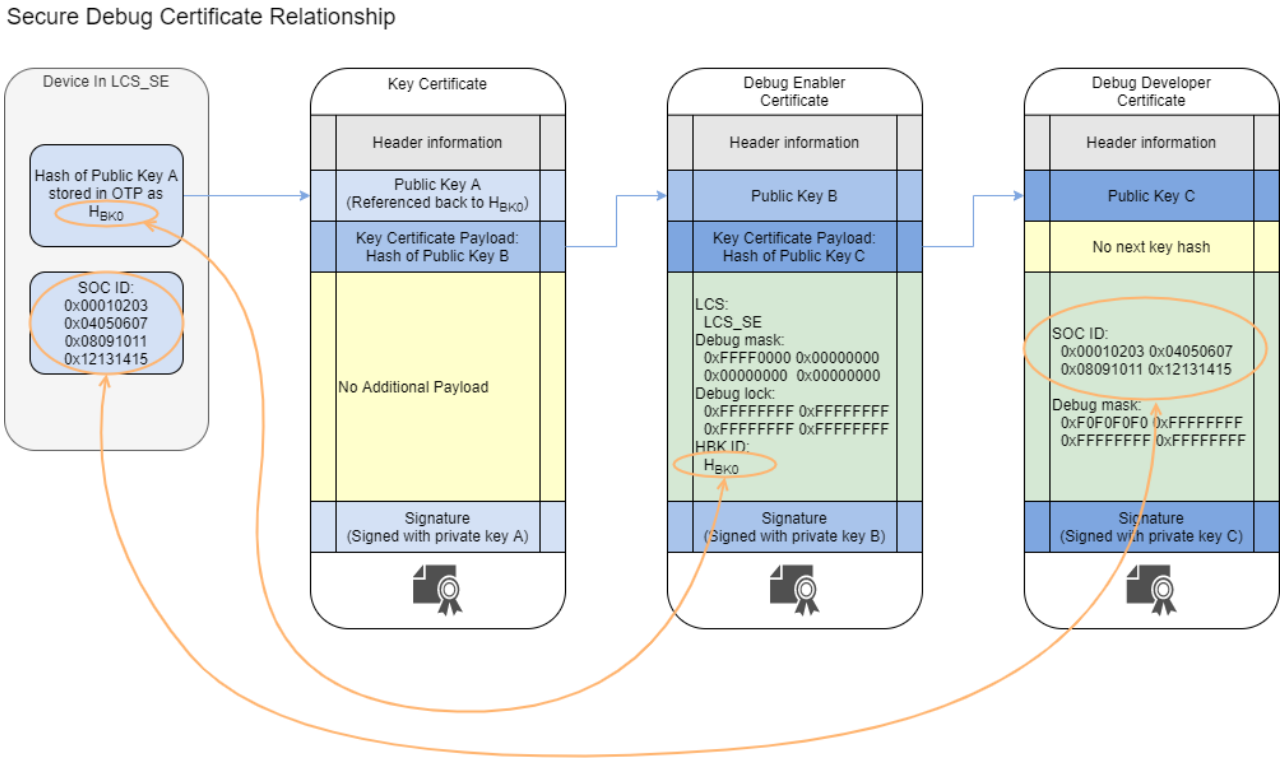

Figure 8. Unlocking the Debug Port on a Secure Device

Points to Note:

- The RoT Hash H_{BK0} refers to the ICV key pair used to sign the first key certificate.
- This is the same HBK ID indicated in the debug enabler certificate.
    - If the RoT keys used to sign the key certificate do not match the HBK ID, the enabler certificate is invalid.
- The SOC ID of the device is used in the debug developer certificate to allow the device to unlock.
    - If the SOC ID does not match, the device does not unlock.
- The debug mask applied to the device is the logical AND of the debug masks defined in the enabler certificate and the developer certificate.
    - Only bits allowed by the enabler certificate can be enabled by the developer.
    - The resultant mask enabled in the "Unlocking the Debug Port on a Secure Device" figure (Figure 8) example is 0xF0F00000 0x00000000 0x00000000 0x00000000.

5.9 Secure Boot Flow

The secure boot process in ROT_STATE follows a similar structure to the boot process in EH_STATE, but adds verification and authentication steps to the flow.

5.9.1 Application Signing

Application signing, described in Section 5.9 “Secure Boot Flow” on page 35, provides the basic components that allow the ROM to verify and authenticate the application. The flow chart in the "Secure Boot Process" figure (Figure 9) gives a high level overview of this process.

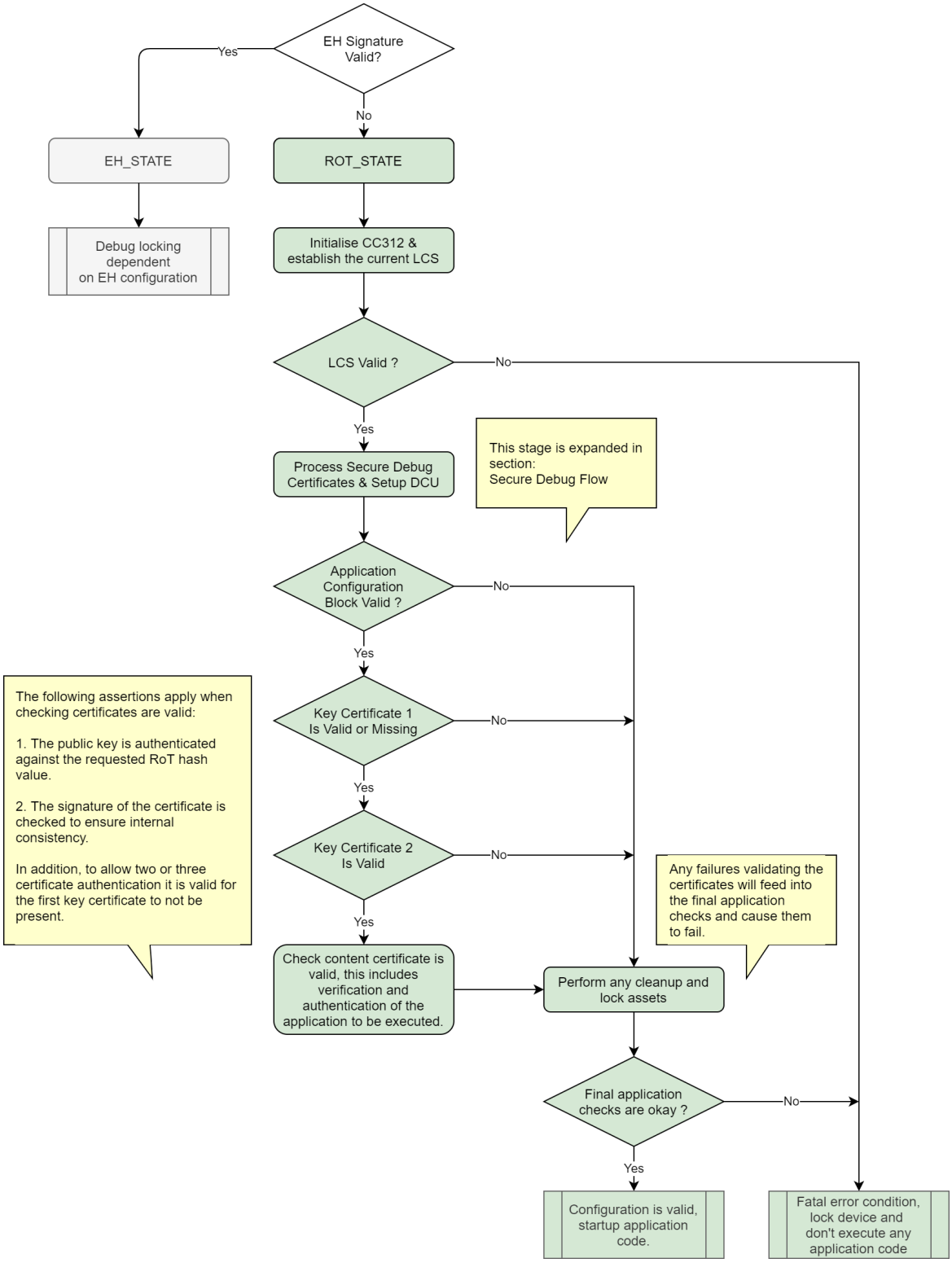

Figure 9. Secure Boot Process

As can be seen from the flow, any failures in the verification and authentication stages result in the device being locked and no application code being executed.

This ensures that the device only operates correctly if valid application code is loaded.

Note that the 'No' path following the "LCS Valid?" check, as shown in the "Secure Boot Process" figure (Figure 9), causes the system to lock up completely in situations where the security block cannot be initialized correctly or the LCS is in an invalid state.

In these situations the device has detected a corruption in the NVM, in which case the device does not have a valid configuration, or the security IP has detected some other situation that prevents the security block from being used. To prevent the exposure of secret information or incorrect application behavior, the device enters an unrecoverable error state.

5.10 Secure Debug Flow

The mechanisms for handling debug certificates are very similar to the mechanisms for handling the key and content certificates mentioned in Section 5.9 “Secure Boot Flow” on page 35.

However, in this case, rather than deciding if an application is valid to be executed, the purpose of the debug certificate chain is to configure the debug port and other features governed by the Debug Control Unit (DCU).

As indicated in Section 5.8 “Secure RoT Resources” on page 30, the debug certificate chain normally consists of the following:

- A key certificate which has been signed with a private key associated with a specific RoT in the system
    - The key certificate is optional, as it is possible instead to sign the debug enabler certificate with the RoT private key. However, this is not recommended.
- A debug enabler certificate that identifies which specific DCU bits can be unlocked, and the LCS of the device being debugged
- A debug developer certificate that identifies the requested DCU bits being unlocked and the SOC ID of the device being debugged

The flow chart in the "Secure Debug Flow" figure (Figure 10) shows the stages of verification debug certificate chains.

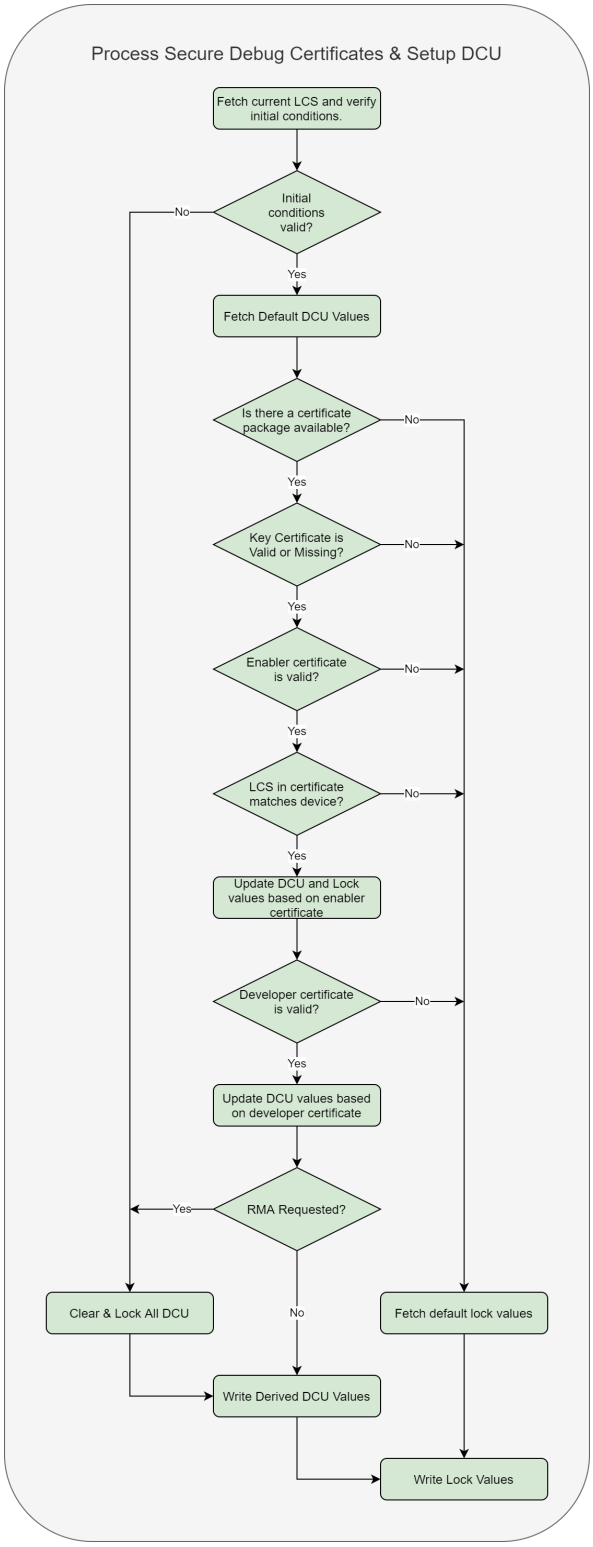

Figure 10. Secure Debug Flow

At the end of the secure debug flow, the DCU mask bits and DCU lock bits are both set to one of three conditions:

1. If there are no debug certificates, the default states for the DCU Mask and lock bits are set.
2. If there is a problem with the provided debug certificates, or a request to transition to LCS_RMA, the DCU mask bits are cleared and all lock bits are set.
3. If there is a valid set of debug certificates, the DCU mask and lock bits are set based on the values in the certificates.

5.11 Secure Applications

5.11.1 Application Signing

Application signing is described in Section 5.8.3.3 “Application Signing” on page 33. The Key and Content certificates are described in Section 5.7.3.1 “Types of Certificate” on page 28.

5.11.2 Application Configuration Block

The application configuration block is a single contiguous block of memory at a fixed location in flash. For RSL15 this is located at the base of the data flash, at address 0x00158000.

The layout of the configuration block is shown in the "Application Configuration Block Layout" figure (Figure 11):

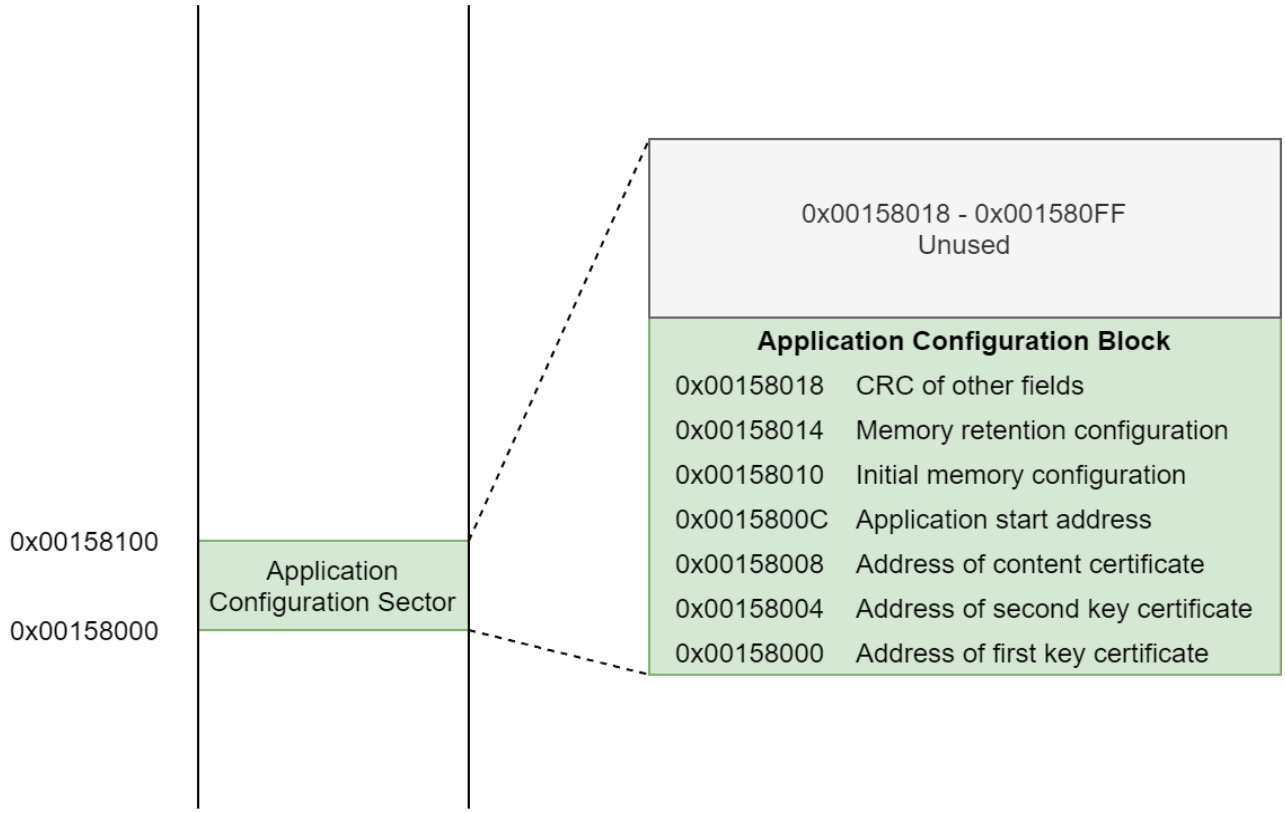

Figure 11. Application Configuration Block Layout

Where:

- Address of first key certificate
  - Defines the base address of the first key certificate in a three-certificate Root of Trust chain
  - If the system is using a two-certificate Root of Trust chain, this address value needs to be set to 0xFFFFFFFF.
- Address of second key certificate
  - Defines the base address of the second key certificate in the Root of Trust chain
- Address of content certificate
  - Defines the base address of the content certificate associated with the application
- Application start address
  - Defines the start address of the application. This points to the base address of the interrupt vector table of the application so that the stack pointer and reset vector can be located.
- Initial memory configuration
  - Defines the initial memory configuration being enabled by the ROM
  - Only used on cold boot of the system. Defines the data written to the `SYSCTRL_MEM_POWER_STARTUP` and `SYSCTRL_MEM_POWER_ENABLE` registers. (See the *RSL15 Hardware Reference* for more register information.)
- Memory retention configuration
  - Defines the memory retention policy the ROM needs to apply
  - Only used on cold boot of the system and defines the data written to the `SYSCTRL_MEM_RETENTION_CFG` register.
- CRC of other fields
  - Defines a CCITT CRC calculated over the other six words of the configuration block
  - If this does not match the calculated CRC during power-up, the configuration block is deemed unusable.

5.11.3 Application Image

The application image is any valid RSL15 image that can be stored in flash on the device.

Normally it is expected to execute completely from flash. However, it is possible to store an encrypted image in flash and have the ROM decrypt it to RAM prior to execution.

At present, any application must execute wholly from flash or from RAM. There is no option to have the ROM load portions of the application to RAM for execution. If partial execution from RAM is required, this needs to be handled at the application level.

CHAPTER 6

Debug Certificate Loading Process And Constraints

In both the EH_STATE and ROT_STATE there are concepts of application configuration and debug certificates.

These are blocks of data in fixed locations which are reserved for use by the ROM. The actual usage and layout of these data blocks are different in the two states.

- EH_STATE
  - In EH_STATE the application configuration is optional allowing the image to be located away from the base of flash.
  - If the application configuration block is not available or not valid, the ROM boots by default from the base of flash memory if a suitable application exists there.
  - The debug certificate in EH_STATE consists of a single reserved area of memory held in a single 256-byte sector of Data Flash.
- ROT_STATE
  - The application configuration data is mandatory in ROT_STATE as it defines the memory locations of the various Root of Trust certificates and hence allows the RoT to be established.
  - The debug certificates, if present, are stored in a reserved block of memory held in contiguous sectors of data flash.

6.1 Reserving Space for the Application Configuration Block and Debug Certificates

On RSL15, the application configuration and debug certificates are held in data flash in consecutive sectors. (Data flash in RSL15 has a sector size of 256 bytes.)

- One sector for the application configuration
- One sector for the EH_STATE debug certificate
- Ten sectors for the ROT_STATE debug certificates

Therefore, a total of 3 KB are reserved for use by the different security mechanisms.

The layout of this is shown in the "Reserved Space for Security Mechanisms" figure (Figure 12).

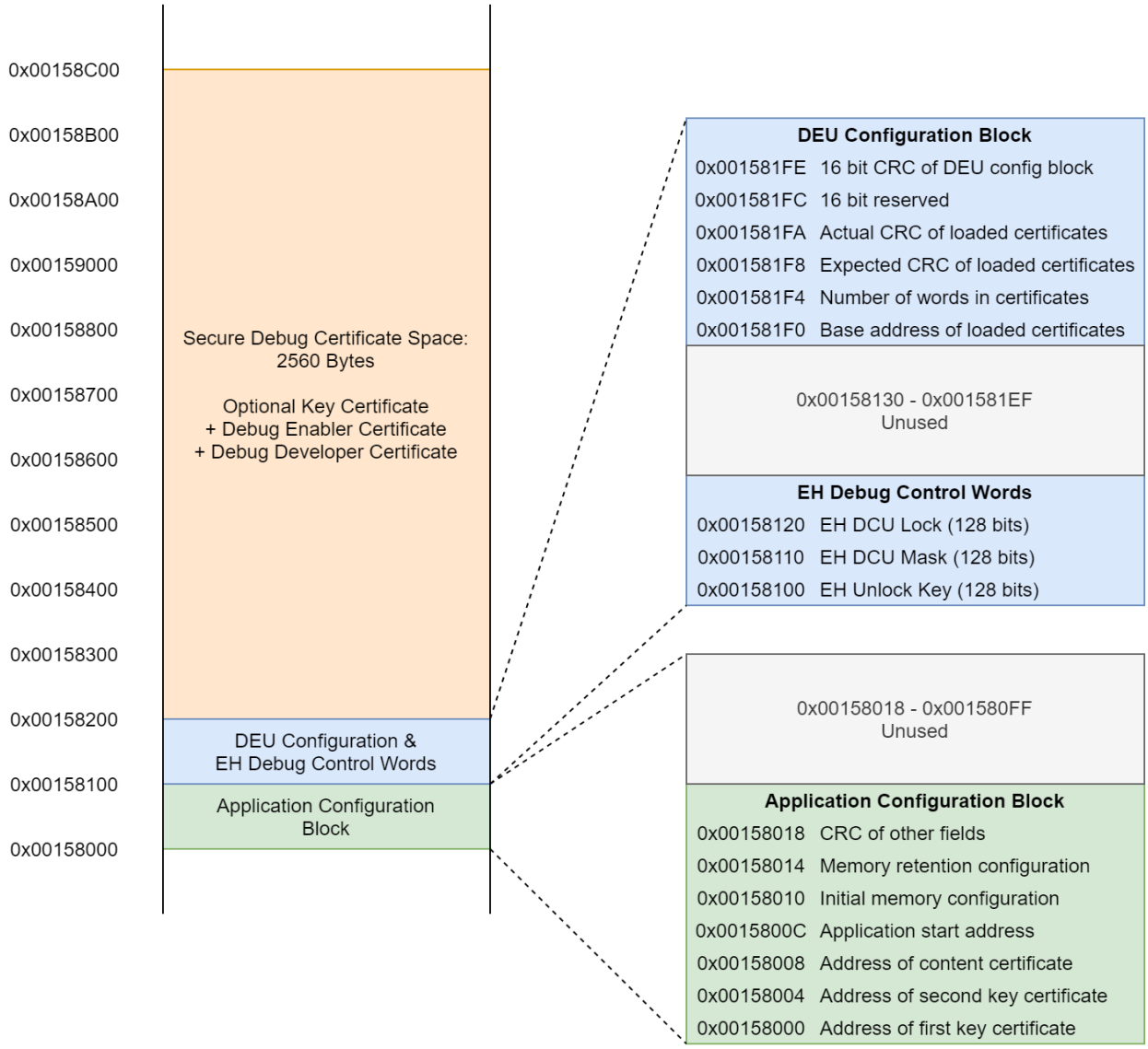

Figure 12. Reserved Space for Security Mechanisms

6.2 Loading Debug Certificates to a Device

When trying to debug a secure device, the initial condition is naturally that the debug port are locked; therefore some other mechanism is required in order to get the debug certificates into the device.

In normal operation, it is possible that the debug certificates could be loaded to the device via a serial port, via an over-the-air connection, or some other way specific to a users application. All these methods, however, require that the application firmware on the device is working well enough to support the transfer and storage of the certificates.

As a default mechanism, RSL15 allows the use of the Data Exchange Unit (DEU) to transfer data into and out of the device in a strictly controlled form.

This feature has been provided with the express purpose of being able to create debug developer certificates and have the packages loaded into the device in a securely managed manner.

The DEU may be used in either EH_STATE or ROT_STATE to perform the following operations:

- Retrieve the SOC ID from the device.
- Load Debug Certificates to the correct location in the device depending on the EH_STATE/ROT_STATE.
- Erase any debug certificate information, hence re-locking the device.

6.3 The Data Exchange Unit (DEU)

The DEU is explained in more detail in the Hardware Reference Manual, however for the purposes of this document it can be thought of as a secure communications channel which is managed by the hardware and ROM in the device.

It provides limited, strictly-controlled features and is not accessible outside of the ROM.

6.3.1 Data Exchange Flow

The DEU communication is managed by the ROM and the hardware in the device.

This is a secure channel that causes a cold reset on the first attempt to access it, hence passing control to the ROM to perform any interaction.

On completion of the interaction with the device it again causes a cold reset, allowing the ROM to re-evaluate the Root of Trust and debug requirements in the system.

In this way, no application code may access the DEU as the ROM ensures that the communication is controlled.

This process is demonstrated in the "Data Exchange Flow" figure (Figure 13):

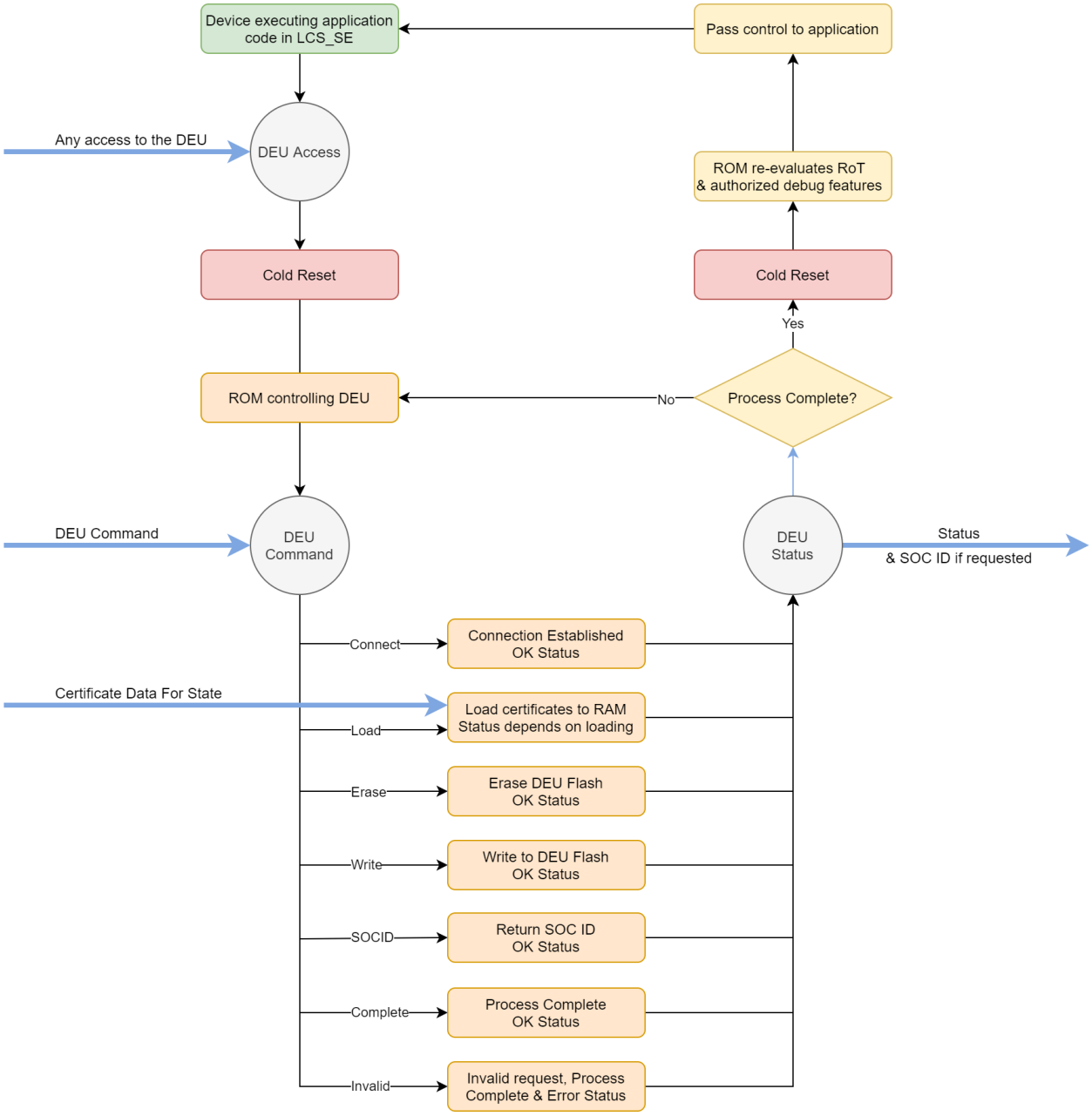

Figure 13. Data Exchange Flow

6.3.2 Data Exchange Unit Protocols

As can be seen from the diagram above, the DEU allows a controlled interaction to take place to perform a limited number of operations.

In the general case, a sequence of commands would be provided to a device in order to perform a specific function. These are indicated in the "Loading a Debug Certificate" figure (Figure 14), the "Erasing a Certificate Area" figure (Figure 15), and the "Retrieving a Device SOC ID" figure (Figure 16).

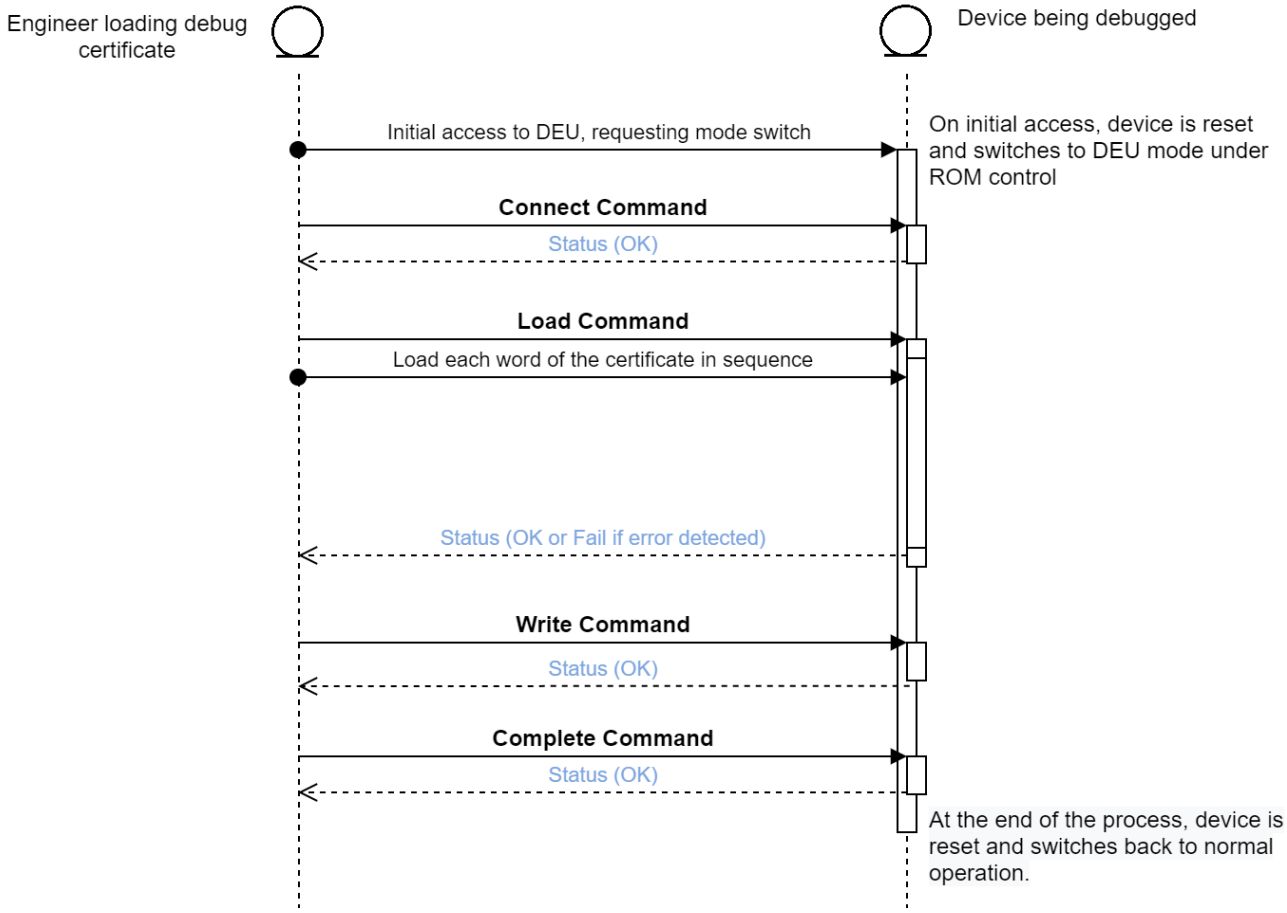

Figure 14. Loading a Debug Certificate

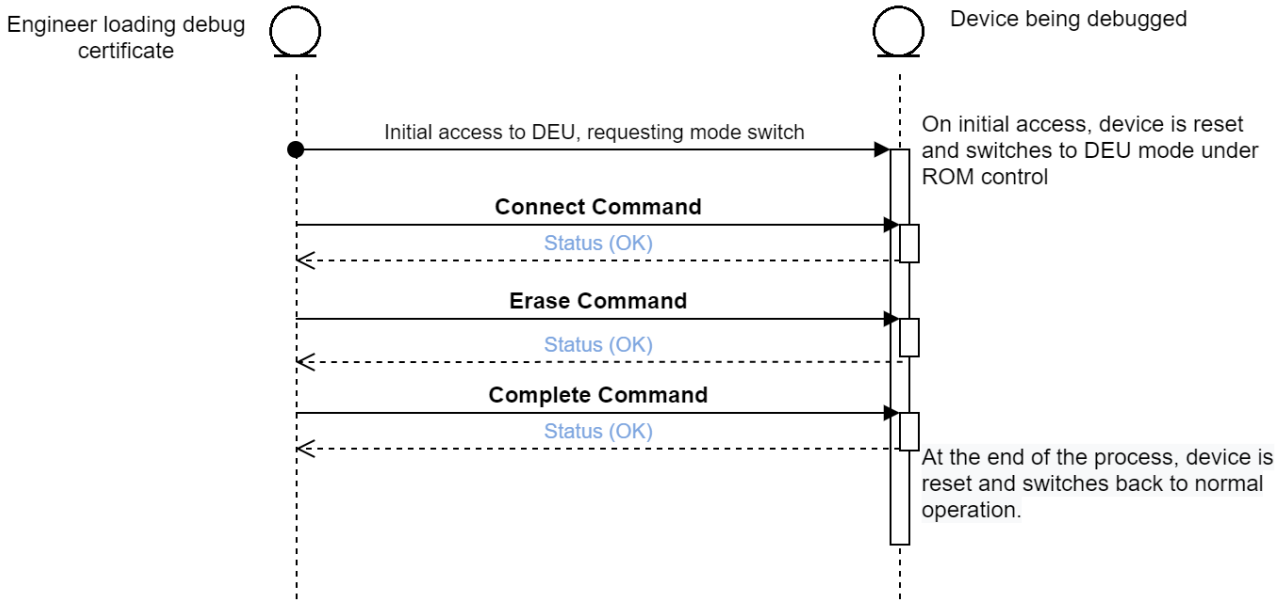

Figure 15. Erasing a Certificate Area

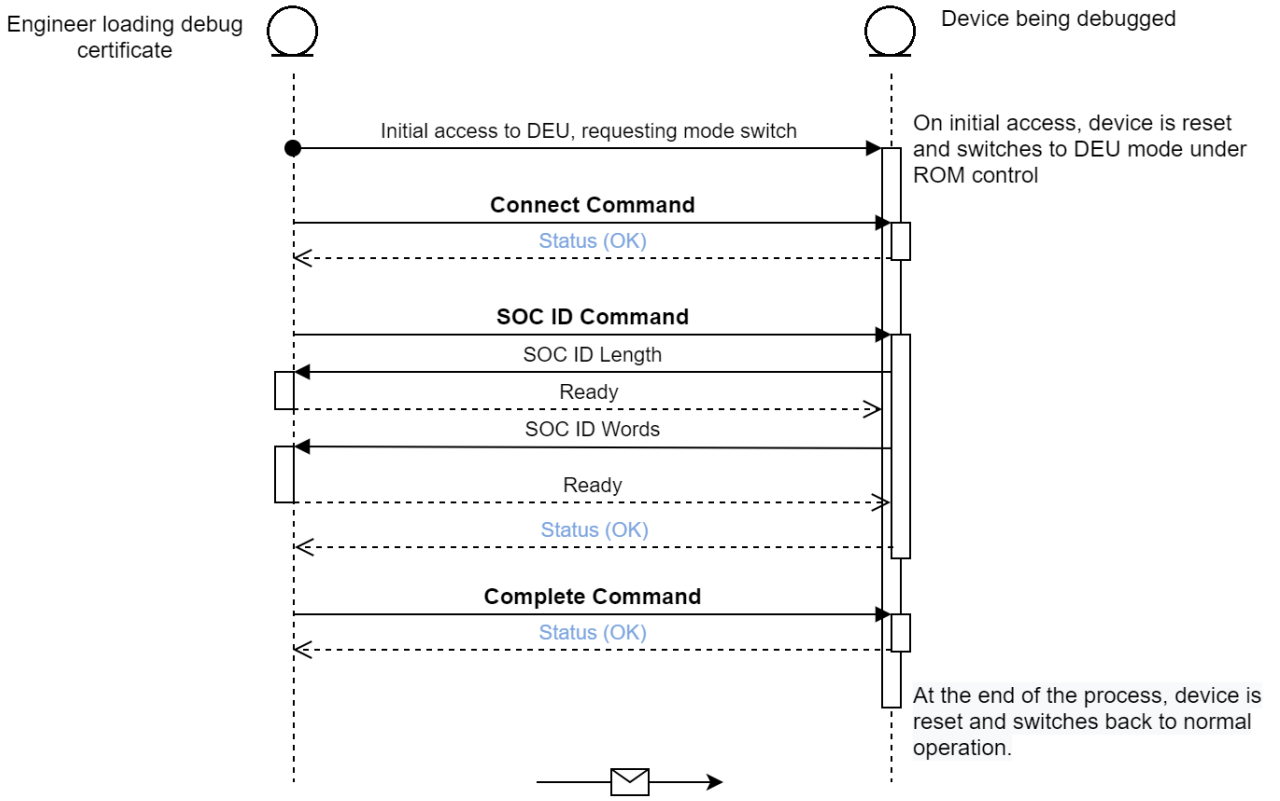

Figure 16. Retrieving a Device SOC ID

CHAPTER 7

Security Tool Support

This topic describes RSLSec, a utility to help manage the transitions through all of the device states and life cycle states. The example commands given throughout this topic are intended as a tutorial.

CAUTION: It is possible for the device to get into a state where transitioning to the desired state is no longer possible (by design). As such, be cautious when learning how to establish a flow that brings you from a new device through to a secure device.

7.1 Getting Started

7.1.1 Overview

Many operations are possible when using the security features of an RSL device, including state transitions, life cycle transitions, debug support, application signing, and more. The RSL Security Tooling (RSLSec) application is designed to make RSL15 security features and their usage as straightforward as possible.

RSLSec consists of two parts:

- A PC-based component which handles the following general operations:
  - Creation of keys
  - Signing of applications
  - Generation of RoT hashes
  - Generation of key and content certificates
  - Generation of debug certificates
  - Management of the process enabling debug of secure devices
  - Management of the transition to RMA state
- An embedded application, used to perform specific life cycle transitions and associated asset provisioning:
  - EH → CM (Effectively, revocation of LCS_EH)
  - CM → DM
  - DM → SE
  - DM → RMA
  - SE → RMA

These two tools combine to meet the needs of RSL devices in all life cycle states.

7.1.2 Software Installation

RSLSec is distributed as a single Windows® executable that combines the PC-based component and the embedded application in one place. To install the software, copy the executable to a suitable location and run it from there with the appropriate parameters (arguments). The tool is designed to be a command-line utility, so it can easily be used in batch scripts or as part of an automated process.

7.1.3 Hardware Set Up

The RSLSec tool is designed to communicate with RSL15 devices using the JTAG/SWD pins. The tool uses the J-Link JTAG programmer when communicating with the device. No other JTAG programmers are supported at this time.

NOTE: The security tools use SWD or the JTAG interface, so these pins must accessible during device provisioning.

In normal operation, RSLSec communicates with the device using either the debug port, or—in cases where the debug port is locked—the Data Exchange Unit (DEU).

7.1.4 RSLSec PC-Based Tool

The RSLSec PC-based tool is provided as a command-line utility with various operating modes, each of which has limited help available.

IMPORTANT: Prior to performing any provisioning operations, we recommend that the flash is in an erased state; this prevents any user code adversely affecting the process.

Some code examples follow:

Top Level Help Messages

```
C:\Development\RSLSec>rslsec --help


usage:
RSLSec [-h] {eh,icv,oem,secure,rma,trust,util} ...


RSL Security Tooling


positional arguments:

  {eh,icv,oem,secure,rma,trust,util}
                        /* Available Security Functions */
    eh                  /* EH Mode Operations */
    icv                 /* Chip Manufacture Operations */
    oem                 /* Device Manufacture Operations */
    secure              /* Secure Operations */
    rma                 /* Return to Manufacture Operations */
    trust               /* Root of Trust Operations */
    util                /* Utility helper operations */

optional arguments:
  -h, --help  /*          show this help message and exit */
```

Top Level EH Help Messages

```
C:\Development\RSLSec>rslsec eh -h


usage:
RSLSec eh [-h] {update,revoke,unlock,relock} ...


EH Mode Operations


positional arguments:
```

```
  {update,revoke,unlock,relock}
                        /* Available LCS_EH Operations */
    update              /* Update the LCS_EH configuration */
    revoke              /* Revoke LCS_EH, transition to LCS_CM */
    unlock              /* Unlock a locked device with the key */
    relock              /* Relock a previously unlocked device */

optional arguments:
  -h, --help            /* show this help message and exit */
```

EH Update help messages

```
C:\Development\RSLSec>rslsec eh update -h


usage:


RSLSec eh update [-h] [--out OUT] [--target TARGET] [--write]
                      [--socid SOCID] [--key KEY KEY KEY KEY]
                      [--ndcu NDCU NDCU NDCU NDCU]

Update the LCS_EH configuration


optional arguments:

  -h, --help            /* show this help message and exit */
  --out OUT             /* File to which the loadable package needs to be written */
  --target TARGET       /* Target connection [RSL15, RSL15-284] */
  --write               /* Update the attached target with the given options */
  --socid SOCID         /* 32 bit SOCID */
  --key KEY KEY KEY KEY
                        /* 128 bit Unlock Key */
  --ndcu NDCU NDCU NDCU NDCU
                        /* 128 bit nDCU Enables */
```

7.1.4.1 RSLSec Common Options

Some RSLSec command options are common to more than one mode of the device. These are indicated in the help sections at the level where they are mentioned, but for clarity, are also documented below.

--target

The `--target` option defines the device with which the utility is communicating. For RSL15, this can be omitted because the default is a standard 512 K device, which is appropriate for RSL15.

--out

This specifies the file to which any loadable packages are written prior to being downloaded to a device. In general this is used to dump iHex formatted files; however, it can also contain other output data depending on the context of the command.

--write

Where the RSLSec command is used to update a device, this option causes the write to happen. If this flag is omitted, any expected generated package files are created but the attached device is not updated. By default this flag is omitted to prevent accidental device updates.

7.2 EH State Configuration And Usage

7.2.1 Overview

Devices delivered from manufacturing are by default in the Energy Harvesting state (EH_STATE), and therefore, in the corresponding life cycle state LCS_EH.

When in LCS_EH, it is possible to implement a lightweight security model that allows the device to be locked, preventing unauthorized access to the code or stored secrets.

To lock a device in LCS_EH, the device must be provided with a key and the DCU bit values. These two pieces of information combine to ensure that only the required debug facilities are allowed in normal operation, while also allowing the device to be unlocked when the correct key is provided via a defined port.

It is also possible to transition a device from LCS_EH to the more secure life cycle states by revoking the state.

When in LCS_EH, the following facilities are available:

1. Configuration Updates
    - Provide a key to the device, allowing debug access when the device is locked.
        - This is a 128-bit value, which might be unique to each physical device or might be unique to a device model, depending on the choice of the entity controlling this part of the process.
    - Write the debug control unit (DCU) values that specify the features available when the device is locked.
        - This is a 128-bit value.
    - Write a SOC ID to the device; the SOC IDcan be used later for identification.
        - This is a 32-bit value, which can be unique to each device.
2. Revocation
    - Revoke the ability to use the LCS_EH, transitioning the device to the secure life cycle model.

As shown in Section 7.1.4 “RSLSec PC-Based Tool” on page 48, the RSLSec command-line utility can be used for all these tasks.

7.2.2 Using LCS_EH Features in RSLSec

All LCS_EH features can be accessed using the `eh` sub-command within RSLSec.

Once the LCS_EH features have been selected then the update, unlock and relock options are available. These are shown below.

High level help in EH state

```
rslsec eh --help

usage:

RSLSec eh [-h] {update,revoke,unlock,relock} ...

/* EH Mode Operations */

positional arguments:
```

```
  {update,revoke,unlock,relock}
                        /* Available LCS_EH Operations */
    update              /* Update the LCS_EH configuration */
    revoke              /* Revoke LCS_EH, transition to LCS_CM */
    unlock              /* Unlock a locked device with the key */
    relock              /* Relock a previously unlocked device */

 optional arguments:

  -h, --help            /* show this help message and exit */
 rslsec eh update --help

 usage:
 RSLSec eh update [-h] [--out OUT] [--target TARGET] [--write]
         [--socid SOCID] [--key KEY KEY KEY KEY][--ndcu NDCU NDCU NDCU NDCU]


 /* Update the LCS_EH configuration */

 optional arguments:

  -h, --help            /* show this help message and exit */
  --out OUT             /* File to which the loadable package is to be written */
  --target TARGET       /* Target connection [RSL15] */
  --write               /* Update the attached target with the given options */
  --socid SOCID         /* 32 bit SOCID */
  --key KEY KEY KEY KEY
                        /* 128 bit Unlock Key */
 --ndcu NDCU NDCU NDCU NDCU
                        /* 128 bit NDCU Enables


 rslsec eh unlock --help

 //usage:
 RSLSec eh unlock [-h] [--target TARGET] [--write] --key KEY KEY KEY KEY [--dcu DCU DCU
DCU DCU]
                        [--lock LOCK LOCK LOCK LOCK]

 //Unlock a locked device with the key

 //optional arguments:
  -h, --help            /* show this help message and exit */
  --target TARGET       /* Target connection [RSL15] */
  --write               /* Update the attached target with the given options */
  --key KEY KEY KEY KEY
                        /* 128 bit Unlock Key */
  --dcu DCU DCU DCU DCU
                        /* 128 bit nDCU Enables */
  --lock LOCK LOCK LOCK LOCK
                        /* 128 bit DCULock values Enables */
 rslsec eh relock --help

 usage:

 RSLSec eh relock [-h] [--target TARGET] [--write]
```

```
/* Relock a previously unlocked device */


optional arguments:

  -h, --help       /* show this help message and exit */
  --target TARGET  /* Target connection [RSL15, RSL15-284] */
  --write          /* Update the attached target with the given options */
```

7.2.2.1 Updating a device

When updating a device, three possible pieces of information can be written: the key, the nDCU bits, and the SOC ID. Each field is controlled independently, each can be updated in individual operations.

Once a field has been updated, its value must not be changed. Any changes would be detected by the ROM code and would invalidate the LCS_EH state of the device.

```
--key < 32-bit word > < 32-bit word > < 32-bit word > < 32-bit word >
```

- The Key
  - This is a 128-bit value provided as four 32-bit words.
  - It is defined by the device manufacturer and can be unique to each device, or can be common across a family of devices.
  - This key is only used to determine whether the device needs to be locked or unlocked, and is not intended for use in any encryption operations.
  - If a device has been locked, a user can unlock the device for debug purposes by providing the key through the dedicated DEU interface.

```
--ndcu < 32-bit word > < 32-bit word > < 32-bit word > < 32-bit word >
```

- The DCU Bits
  - This is a 128-bit value provided as four 32-bit words.
  - This is defined as the nDCU bits, so each bit is the inverse of the value that needs to be written to the DCU.
    - For instance, in order to completely lock the device, all enable bits need to be set to zero in the DCU, so this value would be '0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF'.
  - This is defined by the device manufacturer, but must reflect the DCU bit settings identified in the hardware documentation.
  - Again, this value is used to determine if the device needs to be locked, and if so, what bits need to be set.

```
--socid < 32-bit word >
```

- The SOC ID
  - This is a single 32-bit value.
  - This is defined by the device manufacturer, and can be recalled from the device using the DEU `socid` command.
  - This value is not used internally to the device, it is provided to allow device manufacturers to differentiate individual devices.

7.2.2.2 Unlocking a device

Once a device has been locked, the debug port becomes inactive and it is not possible to communicate with the device through them. To unlock a device for allowing debug, additional information must be loaded to it indicating what functionality is required.

To maintain consistency with the ROT_STATE terminology, the data which is loaded to the device in this case is referred to as a debug certificate. It is not signed content or cryptographically secure, but the basic function is analogous to that situation.

There are three parts to the debug certificate in this case.

```
--key < 32-bit word > < 32-bit word > < 32-bit word > < 32-bit word >
```

- The Key
  - This is a 128-bit value provided as four 32-bit words.
  - It must match the key written at the device’s locking.

```
--dcu < 32-bit word > < 32-bit word > < 32-bit word > < 32-bit  word >
```

- The DCU Bits
  - This is a 128-bit value provided as four 32-bit words.
  - This is defined as the DCU bits, not inverted.
    - For example,to completely unlock the device, all enable bits are set to one in the DCU. Therefore, this value would be '0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF'.

```
--lock < 32-bit word > < 32-bit word > < 32-bit word > < 32-bit word >
```

- The DCU Lock bits
  - This is a 128-bit value provided as four 32-bit words.
  - This allows DCU bits to be locked after the ROM has completed. When bits are locked, it is not possible for those bits in the DCU to be changed until the next power cycle.
  - This feature provides for the possibility of leaving some bits unlocked after the boot process; they can be controlled by firmware.
  - Generally this feature is not required; however, it is provided to maintain compatibility with the ROT_STATE behavior.
  - If this option is omitted, the default behavior is to lock all DCU bits on completion of the ROM processing.

7.2.2.3 Relocking a device

When relocking a device, no options are required; the actions are performed via the DEU, erasing the loaded debug certificates.

7.2.2.4 RSLSec Command Examples for LCS_EH

Examples of updating a device

Starting from a new device, check the SOC ID. It is initially not set, so the value returned is zero.

```
c:\Development\RSLSec>rslsec util socid
['0x00000000']
```

Set the SOC ID to some known value, and check that it is written correctly. (Note that the value of 1024 in decimal is shown as 0x400 hex.)

```
c:\Development\RSLSec>rslsec eh update --socid 1024 --write
c:\Development\RSLSec>rslsec util socid
['0x00000400']
```

Update the key and nDCU bits.

```
c:\Development\RSLSec>rslsec eh update --ndcu 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF
0xFFFFFFFF --key 1 2 3 4 --write
```

The device is now locked, and any attempts to access the debug port result in the debugger failing to connect. An example is shown of attempting to connect using J-Link. The output after this attempt might look slightly different, but the main point is that it has been rendered unable to connect (Cannot connect to target).

```
J-Link>connect
Please specify device / core. <Default>: RSL15
Type '?' for selection dialog
Device>
Please specify target interface:
  J) JTAG (Default)
  S) SWD
  T) cJTAG
TIF>s
Specify target interface speed [kHz]. <Default>: 4000 kHz
Speed>
Device "RSL15" selected.

Connecting to target via SWD
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00

****** Error: Could not find core in Coresight setup
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Cannot connect to target.
J-Link>
```

Unlock the device, setting debug port to be open.

```
c:\Development\RSLSec>rslsec eh unlock --dcu 0x00001FFE 0 0 0 --key 1 2 3 4 --write
```

Verify that a debug connection can now be established.

```
J-Link>connect
Device "RSL15" selected.

Connecting to target via SWD
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x410FD214. Implementer code: 0x41 (ARM)
Found Cortex-M33 r0p4, Little endian.
FPUnit: 4 code (BP) slots and 0 literal slots
Security extension: implemented
Secure debug: enabled
CoreSight components:
ROMTbl[0] @ E00FF000
ROMTbl[0][0]: E000E000, CID: B105900D, PID: 000BBD21 Cortex-M33
ROMTbl[0][1]: E0001000, CID: B105900D, PID: 000BBD21 DWT
ROMTbl[0][2]: E0002000, CID: B105900D, PID: 000BBD21 FPB
Cortex-M33 identified.
J-Link>
```

Relock the device. this operation occurs via the DEU, so there is no need for the `--write` option.

```
c:\Development\RSLSec>rslsec eh relock
```

Again, verify that the ports are locked.

```
J-Link>connect
Device "RSL15" selected.

Connecting to target via SWD
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00

****** Error: Could not find core in Coresight setup
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Found SW-DP with ID 0x0BE12477
AP map detection skipped. Manually configured AP map found.
AP[0]: AHB-AP (IDR: Not set)
AP[0]: Skipped. Invalid implementer code read from CPUIDVal[31:24] = 0x00
Cannot connect to target.
J-Link>
```

7.3 RoT Secure Mode Configuration And Usage

This section details the steps required to establish a secure Root of Trust in a new device that has had its EH signature revoked.

7.3.1 Provisioning Process

Provisioning a device involves updating the specific fields in the NVM that cause life cycle state transitions. Provisioning is used to move a device in ROT_STATE from LCS_CM to LCS_DM, and then to LS_SE. Transitioning to RMA is outside the scope of the provisioning process.

Provisioning normally happens in two places:

1. The ICV (initial chip vendor) owner uses the K_{rtl} of the device to provision some secret information to the NVM and transition the device from LCS_CM to LCS_DM.
2. The OEM owner requires a safe form of the K_{rtl} to provision their own secrets to the device, and to transition it from LCS_DM to LCS_SE.

The "Provisioning Process Overview" figure (Figure 17) demonstrates this process.

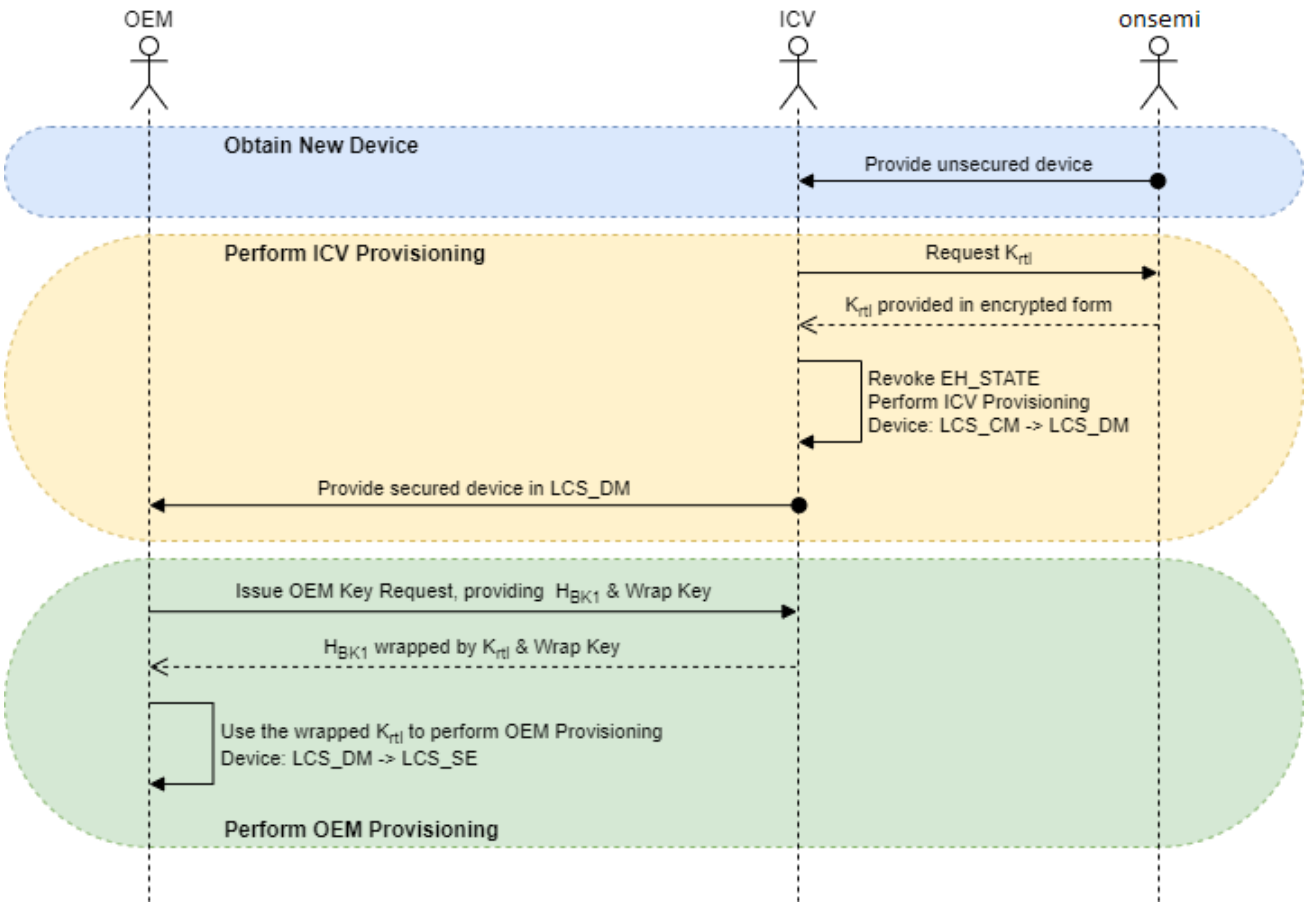

Figure 17. Provisioning Process Overview

The approach shown in the "Provisioning Process Overview" figure (Figure 17) is the general case where the ICV, OEM, and onsemi are different parties. It is equally feasible for the three roles to be performed by a single party, but the processes involved must be the same.

7.3.2 Required Assets When Securing A Device

The "Secure State Transitioning through Full Life Cycle" figure (Figure 18) describes the steps and required assets used when transitioning a secure device through the full life cycle.

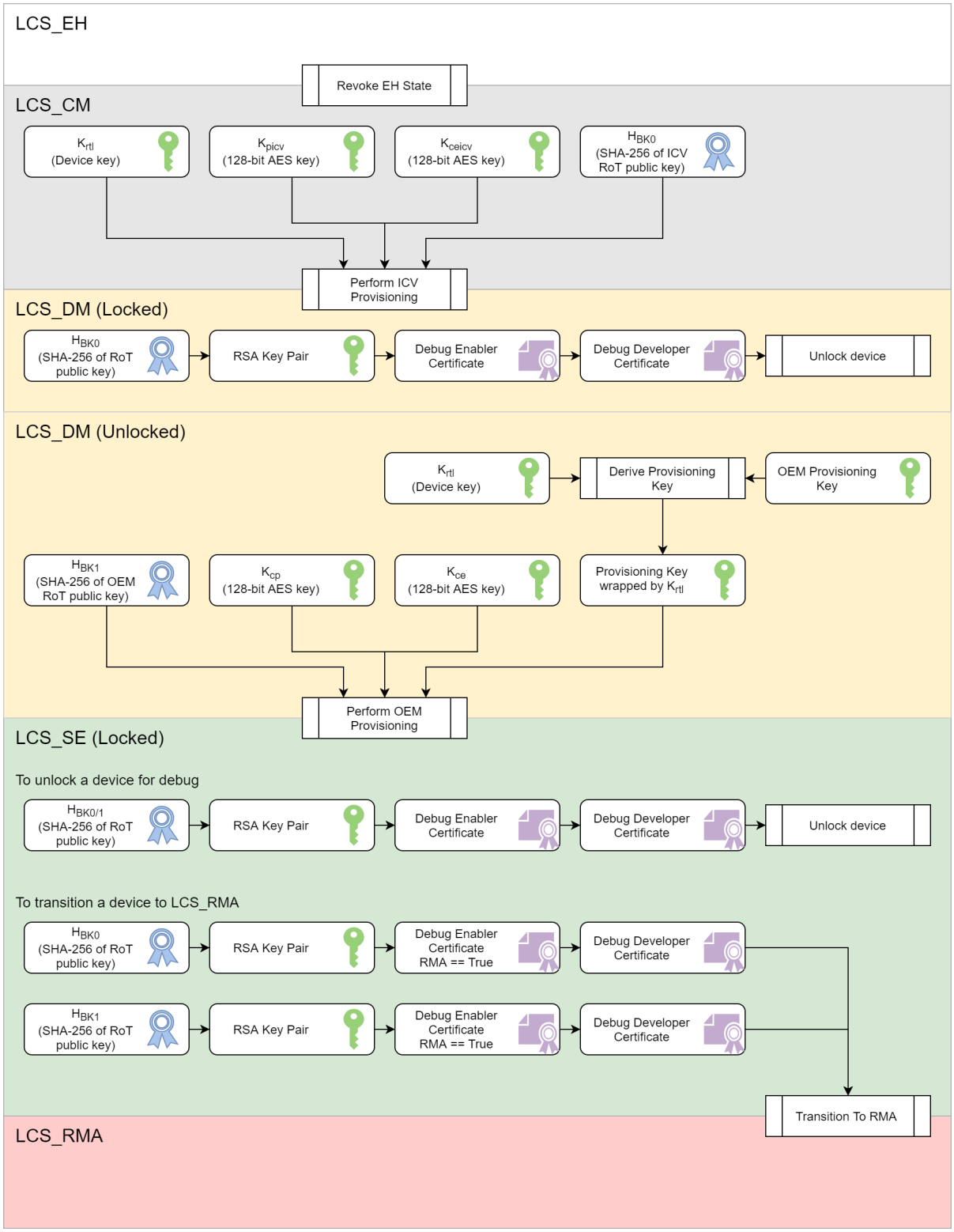

Figure 18. Secure State Transitioning through Full Life Cycle

7.3.3 Initial Resource Creation

7.3.3.1 Key Generation and Usage

This documentation uses specific names for the different keys used in the secure RoT process.

The processes involved in creating a 3072-bit RSA key, or a 128-bit AES key, are the same regardless of the end usage of the key, so these techniques can be used to create keys for other uses as well.

This section focuses on the highlighted part of the "Initial Resource Creation" figure (Figure 19).

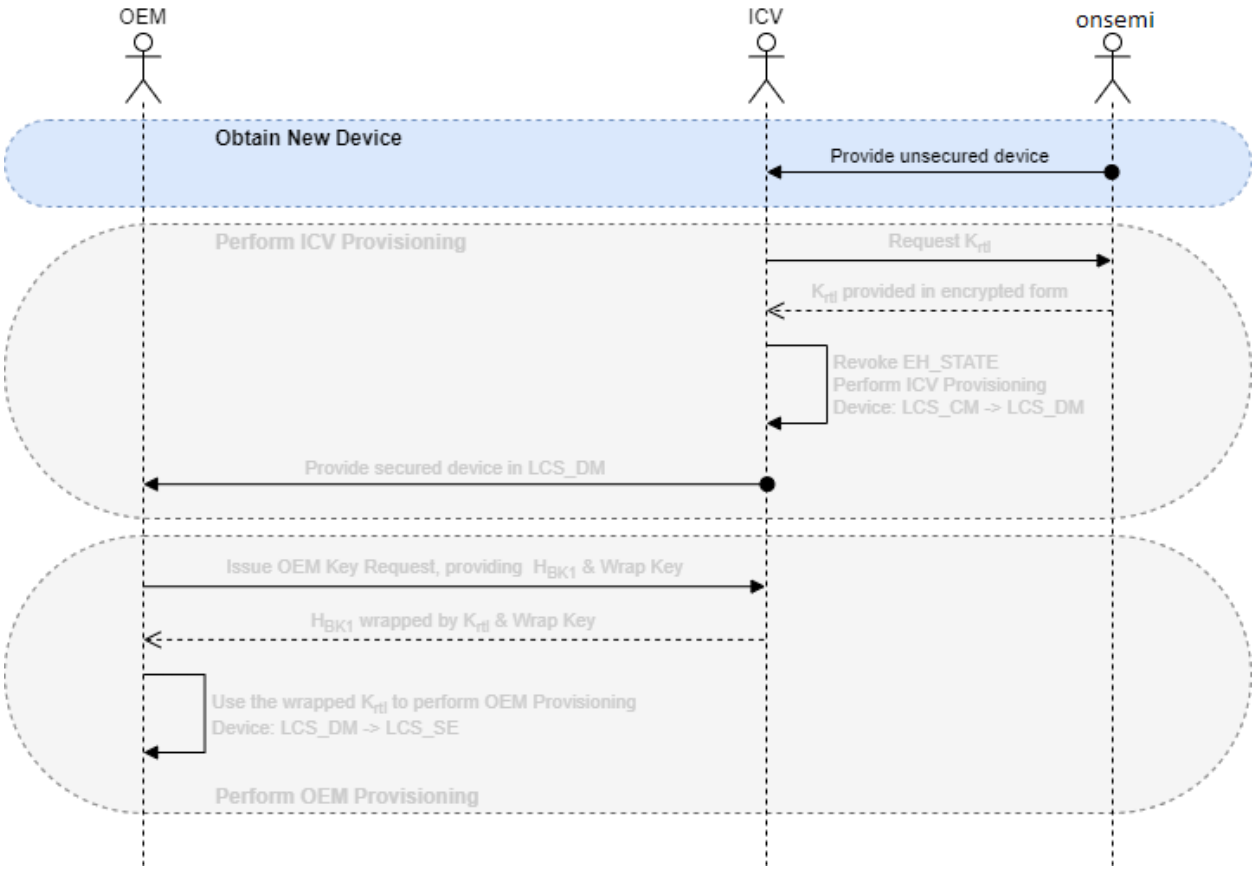

Figure 19. Initial Resource Creation

7.3.3.2 Asset Organization for Demonstration

For the purpose of this document, assets refers to any binary object that might be used to support the secure Roots of Trust.

Some of the items this can include are listed below:

- Encryption Keys
- Signatures for content
- Certificates
- Application binary images

To make the references to these assets easier to understand in the example extracts, the following folder structure is assumed:

Example Folder Structure

```
./assets                            /* Top level folder containing all assets */
        /apps                       /* Applications folder which contains test */
                                    /* applications to be signed */
                /RSL15              /* Definition of the device type for the */
                                    /* applications */
                        /default    /* The default application folder which */
                                    /* contains unencrypted applications signed with */
                                    /* specific keys */
                        /encrypted  /* The encrypted application folder which */
                                    /* contains encrypted applications, again */
                                    /* signed with specific keys */
                        /...        /* The ... indicates there may be more folders of */
                                    /* a similar form */
                /...
        /cert                       /* Top level folder containing generated */
                                    /* certificates */
                /hbk0               /* Folder containing certificates associated with */
                                    /* HBK0 */
                        /...
                /hbk1               /* Folder containing certificates associated with */
                                    /* HBK1 */
                        /...
        /keys                       /* Top level folder for generated keys, the */
                                    /* following folder indicate some of the keys that */
                                    /* are supported */
                /hbk0               /* hbk0 & hbk1 comprise RSA 3072-bit public */
                                    /* private key pairs with their associated hash */
                                    /* values */
                /hbk1
                /kce                /* kce, kceicv, kcp, kpicv are 128-bit AES keys */
                /kceicv
                /kcp
                /kpicv
                /krtl               /* krtl is a 128-bit AES key which matches the RTL */
                                    /* version in the device */
                /...
```

7.3.3.3 K_{rtl}

The K_{rtl} for a given device is held in the hardware and is not made visible to any application code. To support provisioning operations with the RSLSec tool, it is necessary to use the K_{rtl} to encrypt an asset package. The flow of the procedure is to encrypt the asset package, then load it to a device, where it is decrypted by the tools helper on the device during the provisioning process, and securely stored.

The K_{rtl} is a single constant value across all devices in the same family, so all RSL15 devices have the same K_{rtl} embedded in their hardware. This key is sensitive so it must be handled securely and treated as a secret asset. If a customer requires access to the K_{rtl} to allow the provisioning operations to proceed, it can be obtained in an encrypted form from onsemi.

7.3.3.4 Root of Trust Key Pair

A Root of Trust uses a hash of an RSA 3072-bit public key as the basis for all authentication operations.

In order to create a suitable RSA key pair with the associated hashes, the RSLSec tool can be used as shown in the following example:

Creation of RSA Key Pairs

```
C:\Development\RSLSec>rslsec trust make --help

        usage:
RSLSec trust make [-h] [--target <TARGET>] [--name <NAME>]
                       [--phrase <PHRASE>]
                       {hbk,hbk0,hbk1,kpicv,kceicv,kcp,kce} folder

        positional arguments:

  {hbk,hbk0,hbk1,kpicv,kceicv,kcp,kce}
                        /* Define which key to make */

  folder                /* Define the folder to which the files need to be written */

        optional arguments:

  -h, --help            /* show this help message and exit */
  --target <TARGET>     /* Target connection [RSL15] */
  --name <NAME>         /* Define the name associated with the various files */
  --phrase <PHRASE>     /* Define the passphrase associated with the private key */

C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0
C:\Development\RSLS
```

Where:

- hbk0 defines the type of key that is being made, in this case specifying that a RoT RSA key pair with associated hash is being created, associated with H_{BK0}.
  - For RoT RSA key pairs, the valid values are hbk0, hbk1, and hbk, depending on the end use of the key pair.
- No passphrase has been provided for the private key; therefore, a random passphrase is generated and used.
- The various generated files are stored in the specified folder *./assets/keys/hbk0*.
  - *hbk0.prv.pem*: Holds the generated private key in an encrypted PEM file
  - *hbk0.pub.pem*: Holds the public key associated with the generated private key, again in PEM format
  - *hbk0.pwd*: Holds the passphrase required to use the private key
  - *hbk0.sha.bin*: Contains the truncated SHA-256 signature of the public key
  - *hbk0.sha.txt*: Contains a textual form of the binary signature file

The generation of a 3072-bit RSA key pair is a long process, which can take several minutes to complete.

7.3.3.5 Provisioning and Code Encryption Keys

Provisioning and code encryption keys are defined as 128-bit AES keys, and can be created in exactly the same way as the RSA keys defined above.

Generation of code encryption and provisioning keys for the CMPU provisioning process takes the following form:

Creation of AES Keys

```
C:\Development\RSLSec>rslsec trust make kpicv ./assets/keys/kpicv

C:\Development\RSLSec>rslsec trust make kceicv ./assets/keys/kceicv
```

Where:

- kpicv / kceicv define the types of keys being generated.
  - For AES keys, the valid values are kpicv, kceicv, kcp and kce.
- Again. no passphrase has been supplied, so the tool generates a random passphrase to be used when securing the keys.
- The various generated files aare placed in the specified folders,'*./assets/keys/kpicv* and *./assets/keys/kceicv*
  - *kpicv.bin* / *kceicv.bin*: 128-bit AES key stored as a binary object
  - *kpicv.enc* / *kceicv.enc*: The same key stored in a secure form, encrypted with the passphrase
  - *kpicv.pwd* / *kceicv.pwd*: The passphrase to be used to decrypt the secure AES key file
  - *kpicv.txt* / *kceicv.txt*: The AES key in a human readable form

7.3.4 Provisioning CM to DM

This section covers the highlighted area in the "ICV Provisioning" figure (Figure 20):

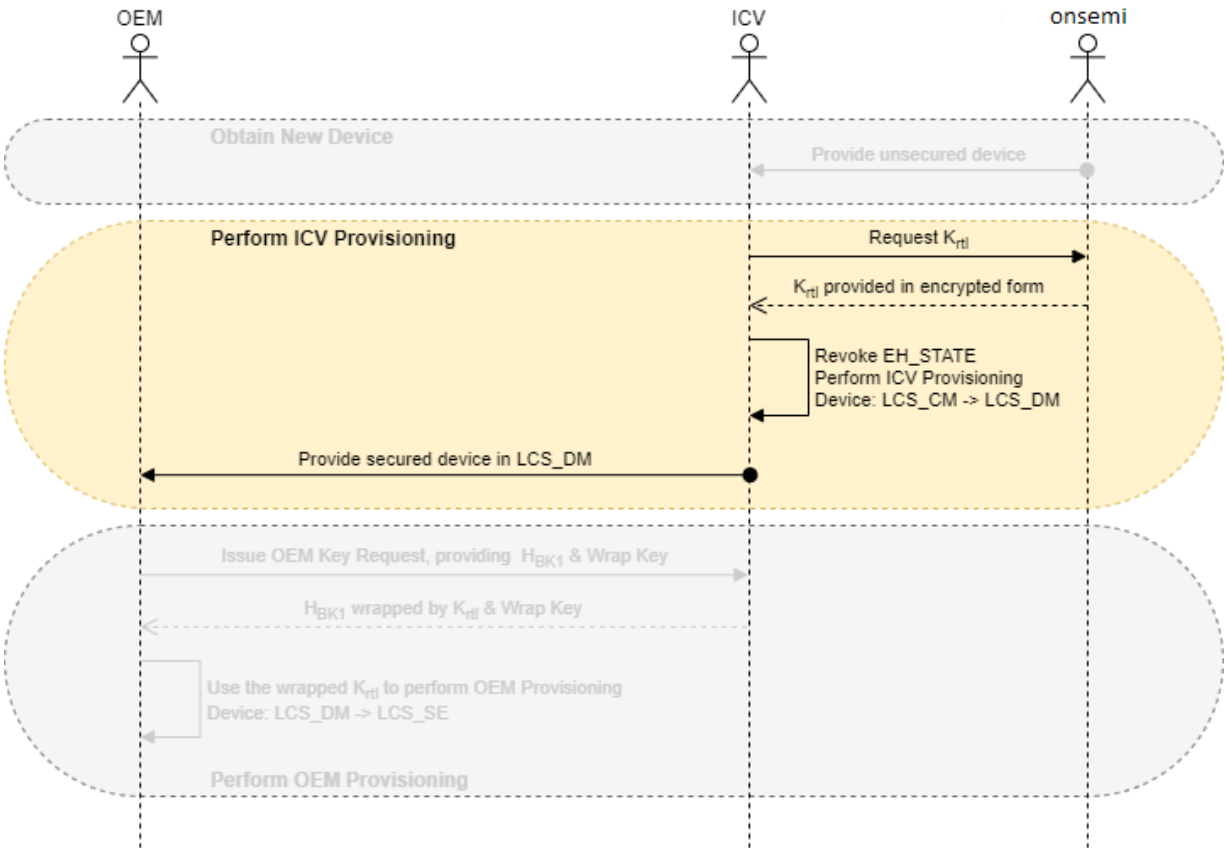

Figure 20. ICV Provisioning

As previously mentioned, the process to go from LCS_CM to LCS_DM requires specific data to be written to the NVM of the device. This is indicated in the diagram in Section 7.3.1 “Provisioning Process” on page 56, and is as follows:

- K_{picv}
- K_{ceicv}
- H_{BK0}
- There could be additional metadata such as minimum software versions and initial DCU states included here, but the default values are enough for this example.

The process for obtaining this data is described in detail in Section 7.3.3 “Initial Resource Creation” on page 59. However, the specific commands are repeated below, to demonstrate the full process.

```
 /* Revoke the EH_STATE to switch to LCS_CM */
 C:\Development\RSLSec>rslsec eh revoke --write --target RSL15

 /* Create a new HBK0, Kpicv and Kceicv */
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0

 C:\Development\RSLSec>rslsec trust make kpicv ./assets/keys/kpicv

 C:\Development\RSLSec>rslsec trust make kceicv ./assets/keys/kceicv

/* locks the debug port. The names and paths could change as relevant to the */
/* assets used. */

 C:\Development\RSLSec>rslsec icv provision --krtl ./assets/keys/krtl/fpga/fpga.enc --
krtlpwd ./assets/keys/krtl/fpga/fpga.pwd --hbk0 ./assets/keys/hbk0/hbk0.sha.bin --kpicv
./assets/keys/kpicv/kpicv.enc --kpicvpwd ./assets/keys/kpicv/kpicv.pwd --kceicv
./assets/keys/kceicv/kceicv.enc --kceicvpwd ./assets/keys/kceicv/kceicv.pwd --write --
target RSL15
```

Full details of the `icv provision` command can be found below:

```
 C:\Development\RSLSec>rslsec icv provision --help


 usage: RSLSec icv provision [-h] [--out OUT] [--target TARGET] [--write]
                             [--krtl KRTL] [--hbk0 HBK0] [--kpicv KPICV]
                             [--kceicv KCEICV] [--krtlpwd KRTLPWD]
                             [--kpicvpwd KPICVPWD] [--kceicvpwd KCEICVPWD]
                             [--minversion MINVERSION] [--config CONFIG]
                             [--dcu DCU DCU DCU DCU]

 optional arguments:
   -h, --help            /* Show this help message and exit */
   --out OUT             /* File to which the loadable package needs to be written */
   --target TARGET       /* Target connection [RSL15] */
   --write               /* Update the attached target with the given options */
   --krtl KRTL           /* File containing the encrypted Krtl */
   --hbk0 HBK0           /* File containing the 128 bit unique HBK0 */
   --kpicv KPICV         /* File containing the 128 bit provisioning key */
   --kceicv KCEICV       /* File containing the 128 bit code encryption key */
   --krtlpwd KRTLPWD     /* File containing the password to unencrypt the Krtl */
   --kpicvpwd KPICVPWD   /* File containing the password to unencrypt the Kpicv */
```

```
  --kceicvpwd KCEICVPWD
                        /* File containing the password to unencrypt the Kceicv */
  --minversion MINVERSION
                        /* ICV Minimum software version */
  --config CONFIG       /* ICV user config word */
  --dcu DCU DCU DCU DCU
                        /* ICV default DCU values */
```

The flags used for ICV provisioning are shown in the "IVC Provisioning Flags" table (Table 2).

Table 2. IVC Provisioning Flags

Flag	Type	Description
--out	File name	The name of a file to which the configuration data is written Not required for provisioning operations, but provided to allow the information to be stored for later use if needed
--target	Device name	The name of the device being provisioned (generally RSL15)
--write	N/A	Instructs the tool to update the connected device by writing the configuration information to the NVM
--krtl & --krtlpwd	File names	Specifies two files, one containing the encrypted K_{rtl} and the second containing the passphrase to decrypt the key for use
--hbk0	File name	File containing the 128-bit hash value of the public key associated with the Root of Trust
--kpicv & --kpicvpwd	File names	Specifies two files, one containing the encrypted K_{cpicv} and the second containing the passphrase to decrypt the key for use
--kceicv & --kceicvpwd	File names	Specifies two files, one containing the encrypted K_{ceicv} and the second containing the passphrase to decrypt the key for use
--minversion	Integer <64	Defines the minimum version of the firmware that can be used on the device
--config	32-bit integer	Defines the ICV configuration word
--dcu	4 * 32-bit integer	Defines the values of 128-bit DCU Default: 0x00001FFE, 0x00001FFE, 0x00000000, 0x00000000

7.3.5 Unlocking a Device Using Debug Certificates

When a device is in LCS_DM or LCS_SE, the default behavior is to have the debug port locked, providing no access to the device.

To access a locked device, debug certificates must be provided, which grant specific access rights to the debug port and additional features controlled by the DCU.

This process is outlined Section 1.1 “Debug Certificate Loading Process And Constraints” on page 1.

The following section demonstrates in more detail the actual process involved using the RSLSec tool.

7.3.5.1 Overview of Required Assets

In order to create and use a valid debug certificate a number of assets are required:

1. The public/private RSA key pair associated with the specific Root of Trust (Root of Trust Key)
2. A debug enabler key, used to create and sign the debug enabler certificate (Debug Enabler Key)
3. When unlocking secure devices, the SOC ID of the device must be known. This is allocated during ICV provisioning and must be requested from the device to be unlocked.
4. A debug developer key, used to create and sign the debug developer certificate (Debug Developer Key)

The "Debug Certificate Chain" figure (Figure 21) shows how these components relate to each other.

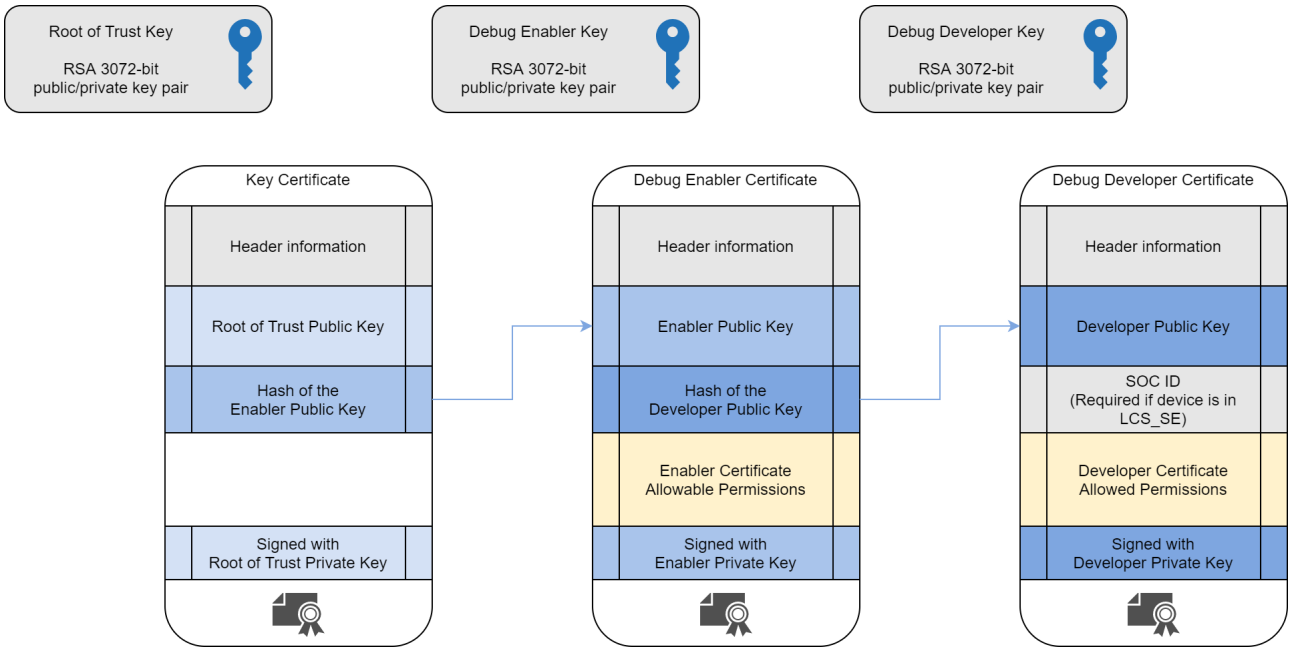

Figure 21. Debug Certificate Chain

It is possible for the owner of each of these keys to be different, and as such, to have no knowledge of the private key or the others. For instance, the entity providing the debug enabler certificate can provide their public key to the owner of the Root of Trust to have them provide the first level key certificate. This maintains the security of the RoT private keys.

In addition, a developer public key can be incorporated into an enablement certificate, allowing generate developer certificates to be generated for multiple devices. In each case the private keys are kept secure within their domain, but the full chain of trust can be authenticated.

7.3.5.2 Creation of Keys, Certificates and Unlocking

7.3.5.2.1 LCS_DM

Each of the three required keys can be generated in exactly the same way. The device has already been provisioned, so the H_{BK0} is already known, but the debug enabler and debug developer keys need to be created.

Once the keys are available, a key certificate, debug enabler certificate, and debug developer certificate are required.

Creation of debug certificates and unlocking of a device in LCS_DM

```
 /* Create a debug enabler RSA key */
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_deb_en

 /* Create a debug developer RSA key pair */
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_deb_dev

 /* Create the first key certificate */
 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk0/key_0.crt --hbk
hbk0 --keypair ./assets/keys/hbk0/hbk0.prv.pem --pwd ./assets/keys/hbk0/hbk0.pwd --
pubkey ./assets/keys/hbk0_deb_en/hbk0_deb_en.pub.pem

 /* Create the debug enabler certificate */
 C:\Development\RSLSec>rslsec trust cert enabler --out ./assets/cert/hbk0/dbg_en_dm.crt
--keypair ./assets/keys/hbk0_deb_en/hbk0_deb_en.prv.pem --pwd ./assets/keys/hbk0_deb_
en/hbk0_deb_en.pwd --pubkey ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pub.pem --keycert
./assets/cert/hbk0/key_0.crt --hbk hbk0 --lcs dm

 /* Create the debug developer certificate */
 C:\Development\RSLSec>rslsec trust cert developer --out ./assets/cert/hbk0/dbg_dev_
dm.crt --keypair ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.prv.pem --pwd
./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pwd --enabler ./assets/cert/hbk0/dbg_en_dm.crt

 /* And unlock the device */
 C:\Development\RSLSec>rslsec secure unlock --cert ./assets/cert/hbk0/dbg_dev_dm.crt --
target RSL15
```

7.3.5.2.2 LCS_SE

When creating a debug certificate for a device in LCS_SE, the debug developer certificate requires the SOC ID of the device.

Creation of debug certificates and unlocking of a device in LCS_SE (Using HBK0)

```
 /* Create a debug enabler RSA key */
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_deb_en

 /* Create a debug developer RSA key pair */
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_deb_dev

 /* Create the first key certificate */
 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk0/key_0.crt --hbk
hbk0 --keypair ./assets/keys/hbk0/hbk0.prv.pem --pwd ./assets/keys/hbk0/hbk0.pwd --
pubkey ./assets/keys/hbk0_deb_en/hbk0_deb_en.pub.pem

 /* Create the debug enabler certificate */
 C:\Development\RSLSec>rslsec trust cert enabler --out ./assets/cert/hbk0/dbg_en_se.crt
--keypair ./assets/keys/hbk0_deb_en/hbk0_deb_en.prv.pem --pwd ./assets/keys/hbk0_deb_
en/hbk0_deb_en.pwd --pubkey ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pub.pem --keycert
./assets/cert/hbk0/key_0.crt --hbk hbk0 --lcs se

 /* Query the SOC ID from the device */
```

```
 C:\Development\RSLSec>rslsec util socid --target RSL15
 [0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn
0xnnnnnnnn]

 /* Create the debug developer certificate */
 C:\Development\RSLSec>rslsec trust cert developer --socid 0xnnnnnnnn 0xnnnnnnnn
0xnnnnnnnn 0xnnnnnnnn 0xnnnnnnnn  0xnnnnnnnn  0xnnnnnnnn 0xnnnnnnnn --out
./assets/cert/hbk0/dbg_dev_se.crt --keypair ./assets/keys/hbk0_deb_dev/hbk0_deb_
dev.prv.pem --pwd ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pwd --enabler
./assets/cert/hbk0/dbg_en_se.crt

 /* And unlock the device */
 C:\Development\RSLSec>rslsec secure unlock --cert ./assets/cert/hbk0/dbg_dev_se.crt --
target RSL15
```

Using the H_{BK1}, the Root of Trust is very similar, but uses the --hbk1 flag where appropriate and is derived from a chain of trust starting at the H_{BK1} RSA keys.

7.3.6 Provisioning DM to SE

The process for moving a device from LCS_DM to LCS_SE (secure) is similar to that used when transitioning from LCS_CM to LCS_DM, as seen in the highlighted portion of the "OEM Provisioning" figure (Figure 22).

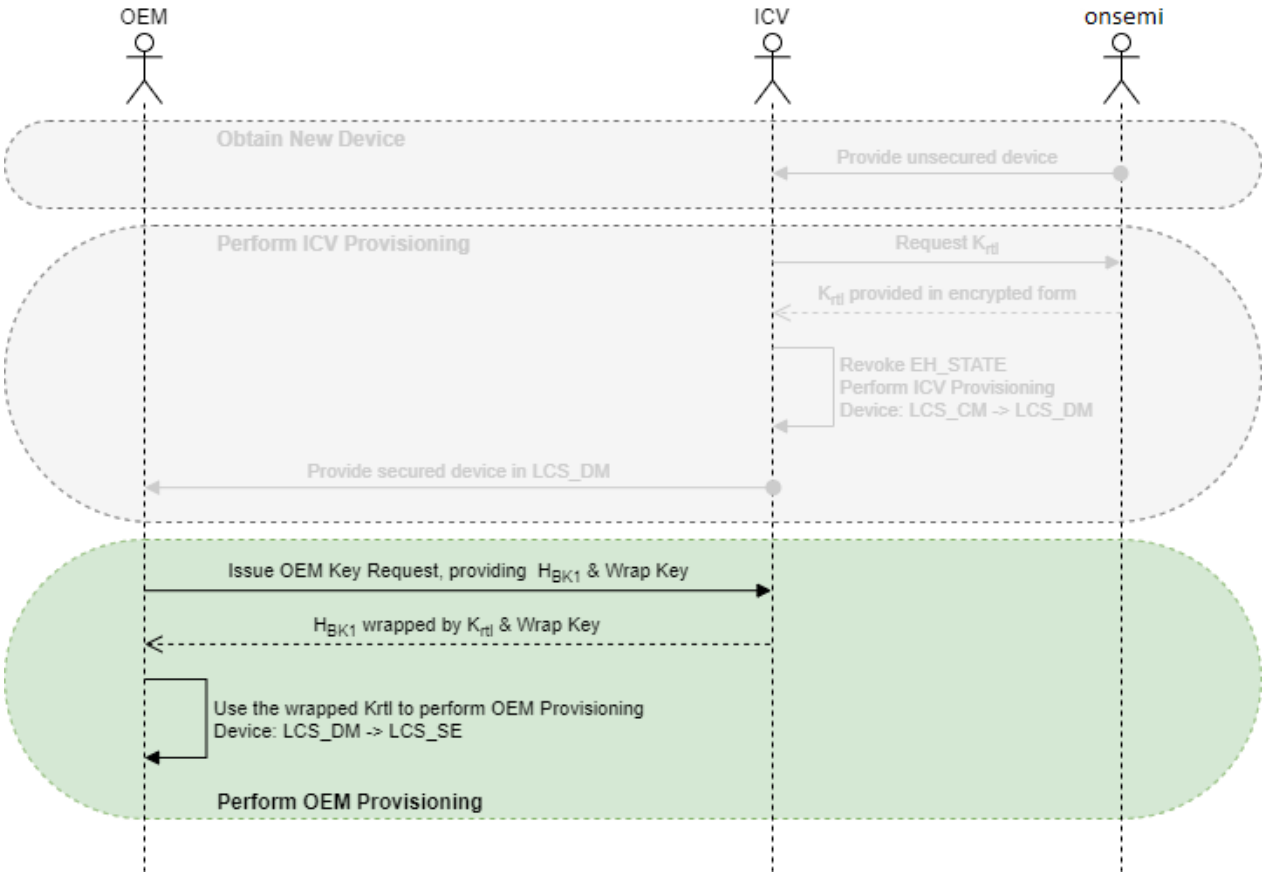

Figure 22. OEM Provisioning

In this case, the specific data items that need to be provisioned to the device are:

- K_{cp}
- K_{ce}
- HBK1
- Potentially, additional metadata such as minimum software versions and initial DCU states, but the default values are enough for this example

This stage might be performed by a party other than the owner of the ICV RoT. In that case, we recommend that the K_{rtl} be held securely by the ICV, and that the OEM use a wrapped OEM provisioning key to provision the assets.

To get the wrapped OEM provisioning key, the OEM must issue a request for the wrapped key package to the ICV, who can then encrypt the key using the K_{rtl} and provide it back to the OEM. This returned wrapped key package can be used in the provisioning process.

Other than this initial stage, the process is similar to the LCS_CM → LCS_DM sequence:

- Create the H_{BK1} RSA key pair and SHA256 hash value.
- Create a OEM_WRAP_REQUEST RSA key pair and SHA256 hash value.
- Create the 128-bit AES keys for code encryption and provisioning.

- Request a wrapped key from the ICV and use that wrapped key to provision the OEM assets.
  - This locks the device.
- Create the debug enabler and debug developer RSA key pairs and SHA256 hash values.
- Create the key certificate using the H_{BK1} key pair.
- Create the debug enabler certificate using the enable key pair.
- Read the SOC ID from the device to be unlocked.
- Create the debug developer certificate using the developer key pair and the SOC ID.
- Unlock the device using the debug certificate.

This process is shown below:

OEM Provisioning Process

```
/* Make folders for some of the keys and certificate needed */
 C:\Development\RSLSec>mkdir assets\cert\hbk1

 /* Create the Root of Trust RSA key pair and associated hash files */
 C:\Development\RSLSec>rslsec trust make hbk1 ./assets/keys/hbk1

 /* Create a wrapping key for the HBK1 wrap request */
 C:\Development\RSLSec>rslsec trust make hbk1 ./assets/keys/oem_wrap_request

 /* Create the provisioning and code encryption AES keys */
 C:\Development\RSLSec>rslsec trust make kcp ./assets/keys/kcp

 C:\Development\RSLSec>rslsec trust make kce ./assets/keys/kce

 /* Need to request the wrapping OEM key from the ICV */
 C:\Development\RSLSec>rslsec oem request --keypair ./assets/keys/hbk1/hbk1.prv.pem --
pwd ./assets/keys/hbk1/hbk1.pwd --pubkey ./assets/keys/oem_wrap_request/oem_wrap_
request.pub.pem --out ./assets/keys/oem_wrap_request/oem_wrap_request.req.pkg

 /* ICV responds with a wrapped OEM version of the key */
 C:\Development\RSLSec>rslsec icv response --krtl ./assets/keys/krtl/fpga/fpga.enc --
krtlpwd ./assets/keys/krtl/fpga/fpga.pwd --OEM ./assets/keys/oem_wrap_request/oem_wrap_
request.req.pkg --out ./assets/keys/oem_wrap_request/oem_wrap_request.rsp.pkg

/* Provision the ICV assets to the device */
 C:\Development\RSLSec>rslsec oem provision --hbk1 ./assets/keys/hbk1/hbk1.sha.bin --
icvenc ./assets/keys/oem_wrap_request/oem_wrap_request.rsp.pkg --oemenc
./assets/keys/oem_wrap_request/oem_wrap_request.prv.pem --oemencpwd ./assets/keys/oem_
wrap_request/oem_wrap_request.pwd --kcp ./assets/keys/kcp/kcp.enc --kcppwd
./assets/keys/kcp/kcp.pwd --kce ./assets/keys/kce/kce.enc --kcepwd
./assets/keys/kce/kce.pwd --write --target RSL15

           /* The device is in LCS_SE and has its debug port locked */


/* Create RSA keys for use when creating the debug enabler and developer certificates */

 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk1_deb_en
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk1_deb_dev
```

```
 /* Fetch the SOC ID and store it in a file */


 C:\Development\RSLSec>rslsec util socid --target RSL15 > socid.txt

  /* Create a key certificate using the Root of Trust hash, and chain the enabler */

 /* and developer certificates from that */

 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk1/key_0.crt --hbk
hbk1 --keypair ./assets/keys/hbk1/hbk1.prv.pem --pwd ./assets/keys/hbk1/hbk1.pwd --
pubkey ./assets/keys/hbk1_deb_en/hbk1_deb_en.pub.pem

 C:\Development\RSLSec>rslsec trust cert enabler --out ./assets/cert/hbk1/dbg_en_se.crt
--keypair ./assets/keys/hbk1_deb_en/hbk1_deb_en.prv.pem --pwd ./assets/keys/hbk1_deb_
en/hbk1_deb_en.pwd --pubkey ./assets/keys/hbk1_deb_dev/hbk1_deb_dev.pub.pem --keycert
./assets/cert/hbk1/key_0.crt --hbk hbk1 --lcs se

 C:\Development\RSLSec>rslsec trust cert developer --socidFile socid.txt --out
./assets/cert/hbk1/dbg_dev_se.crt --keypair ./assets/keys/hbk1_deb_dev/hbk1_deb_
dev.prv.pem --pwd ./assets/keys/hbk1_deb_dev/hbk1_deb_dev.pwd --enabler
./assets/cert/hbk1/dbg_en_se.crt

 /* Now unlock the device using the newly created certificate */
 C:\Development\RSLSec>rslsec secure unlock --cert ./assets/cert/hbk1/dbg_dev_se.crt --
target RSL15
```

Full details of the `oem provision` command can be found below:

```
 C:\Development\RSLSec>rslsec oem provision --help

 usage: RSLSec oem provision [-h] [--out OUT] [--target TARGET] [--write]
                             [--hbk HBK | --hbk1 HBK1] [--icvenc ICVENC]
                             [--oemenc OEMENC] [--kcp KCP] [--kce KCE]
                             [--oemencpwd OEMENCPWD] [--kcppwd KCPPWD]
                             [--kcepwd KCEPWD] [--minversion MINVERSION]
                             [--dcu DCU DCU DCU DCU]

 optional arguments:
   -h, --help            /* show this help message and exit */
   --out OUT             /* File to which the loadable package needs to be written */
   --target TARGET       /* Target connection [RSL15] */
   --write               /* Update the attached target with the given options */
   --hbk HBK             /* File containing the HBK */
   --hbk1 HBK1           /* File containing the HBK1 */
   --icvenc ICVENC       /* File containing ICV encoded provisioning key */
   --oemenc OEMENC       /* File containing the RSA key pair to decrypt the ICV package
*/
   --kcp KCP             /* File containing the 128 bit provisioning key */
   --kce KCE             /* File containing the 128 bit code encryption key */
   --oemencpwd OEMENCPWD
                         /* File containing the password for the encryption keys */
   --kcppwd KCPPWD       /* File containing the password to unencrypt the Kcp */
   --kcepwd KCEPWD       /* File containing the password to unencrypt the Kce */
   --minversion MINVERSION
                         /* OEM Minimum software version */
```

```
  --dcu DCU DCU DCU DCU
                        /* OEM default DCU values */
```

The flags used for OEM provisioning are shown in the "OEM Provisioning Flags" table (Table 3).

Table 3. OEM Provisioning Flags

Flag	Type	Description
--out	File name	This flag is used to provide the name of a file to which the configuration data needs to be written. It is not required for provisioning operations, but is provided to allow the information to be stored for later use if needed.
--target	Device name	This is the name of the device being provisioned; generally RSL15.
--write	n/a	This flag instructs the tool to update the connected device by writing the configuration information to the NVM.
--hbk, --hbk1	File name	This is the file containing the 128-bit hash value of the public key associated with the Root of Trust (256 bit if --hbk using single RoT) .
--icvenc	File name	This flag is the file containing an ICV encoded provisioning key.
--oemenc & --oemencpwd	File names	This specifies two files: the first containing the encrypted OEM wrapping key, and the second containing the passphrase to decrypt the key for use.
--kcp & --kcppwd	File names	This specifies two files: the first containing the encrypted K_{cp}, and the second containing the passphrase to decrypt the key for use.
--kce & --kcepwd	File names	This specifies two files: the first containing the encrypted K_{ce}, and the second containing the passphrase to decrypt the key for use.
--minversion	Integer <96	This defines the minimum version of the firmware that can be used on the device.
--config	32-bit integer	This defines the ICV configuration word.
--dcu	4 * 32-bit integer	This defines the values of 128-bit DCU. Default: 0x00001FFE, 0x00001FFE, 0x00000000, 0x00000000

7.3.7 Transition to LCS_RMA

Transitioning to LCS_RMA is required when devices need to be returned to the manufacturer due to device failure. This process involves erasing secret information on the device, and opens the debug port to allow fault analysis. Since the device could contain secret information from both the ICV and OEM Roots of Trust, specific signed certificates from both parties must be present to erase the secret information and to unlock the debug port. These certificates are authenticated against the specified RoT in the same way as normal debug certificates.

The same process is followed for both roots of trust:

1. A special debug enabler certificate is created, indicating that the RMA switch is allowed on any devices with the RoT hashes (H_{BK0}/H_{BK1}).
2. A special debug developer certificate is created, indicating which specific device can be transitioned to LCS_RMA.

3. The certificate is loaded to the device in the same way as other debug certificates, and the device performs a power on reset.

The transition to RMA occurs only when both of the Roots of Trust provide a suitable debug certificate.

The process is outlined below:

```
          /* You can reuse the key certificates generated during the initial */
          /* provisioning of both ICV and OEM assets. */


 /* Make sure you know the SOC ID of the device. */


 C:\Development\RSLSec>rslsec util socid --target RSL15 > socid.txt

 /* Create the HBK0 debug enabler and developer keys to support the transition. */
 C:\Development\RSLSec>rslsec trust cert enabler --out ./assets/cert/hbk0/rma_en_se.crt
--keypair ./assets/keys/hbk0_deb_en/hbk0_deb_en.prv.pem --pwd ./assets/keys/hbk0_deb_
en/hbk0_deb_en.pwd --pubkey ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pub.pem --keycert
./assets/cert/hbk0/key_0.crt --hbk hbk0 --lcs se --rma --target RSL15

 C:\Development\RSLSec>rslsec trust cert developer --socidFile socid.txt --out
./assets/cert/hbk0/rma_dev_se.crt --keypair ./assets/keys/hbk0_deb_dev/hbk0_deb_
dev.prv.pem --pwd ./assets/keys/hbk0_deb_dev/hbk0_deb_dev.pwd --enabler
./assets/cert/hbk0/rma_en_se.crt

 /* Create the HBK1 debug enabler and developer keys to support the transition. */
 C:\Development\RSLSec>rslsec trust cert enabler --out ./assets/cert/hbk1/rma_en_se.crt
--keypair ./assets/keys/hbk1_deb_en/hbk1_deb_en.prv.pem --pwd ./assets/keys/hbk1_deb_
en/hbk1_deb_en.pwd --pubkey ./assets/keys/hbk1_deb_dev/hbk1_deb_dev.pub.pem --keycert
./assets/cert/hbk1/key_0.crt --hbk hbk1 --lcs se --rma --target RSL15

 C:\Development\RSLSec>rslsec trust cert developer --socidFile socid.txt --out
./assets/cert/hbk1/rma_dev_se.crt --keypair ./assets/keys/hbk1_deb_dev/hbk1_deb_
dev.prv.pem --pwd ./assets/keys/hbk1_deb_dev/hbk1_deb_dev.pwd --enabler
./assets/cert/hbk1/rma_en_se.crt

 /* Apply the certificates. Note that the order is important. */
 C:\Development\RSLSec>rslsec secure unlock --cert ./assets/cert/hbk1/rma_dev_se.crt --
target RSL15

 /* Finally erase the certificates by relocking the device. As the device is */
 /* now in LCS_RMA, the debug port remains open. */

 C:\Development\RSLSec>rslsec secure relock --target RSL15
```

On completion of this process, the device is unlocked and the ICV/OEM provisioning and code encryption keys are erased.

7.3.8 Creating and Executing a Secure Application

7.3.8.1 Creating a Secure Application

The Root of Trust provides mechanisms to ensure that an application being executed by the ROM meets criteria for it to be verified and authenticated. The mechanisms, criteria and procedure are explained here.

Verification of an application ensures that the delivered image has not been corrupted in transit or changed in any way. This is done by calculating the SHA256 signature of the application and comparing it with that provided by the associated content certificate. In addition, the minimum software version of the application must be greater than or equal to that defined in the NVM for the particular Root of Trust. This is an anti-rollback measure allowing older images to be disabled.

Authentication of applications ensures that the delivered image has been created by the correct author, i.e., the owner of the Root of Trust private keys. This is done by ensuring that the chain of trust defined by the key and the content certificates can be cryptographically proven to have been signed by the Root of Trust private key associated with the stored $H_{BK0/1}$ hash values.

7.3.8.2 Creating and Executing a Secure Application

A secure application can be any application that normally executes on RSL15, with the qualification that it is accompanied by a suitable application configuration element and a Root of Trust chain embedded in one of the Roots of Trust in the system: H_{BK0} or H_{BK1}.

The "Secure Application Layouts" figure (Figure 23) shows two possible layouts of a secure application in flash memory.

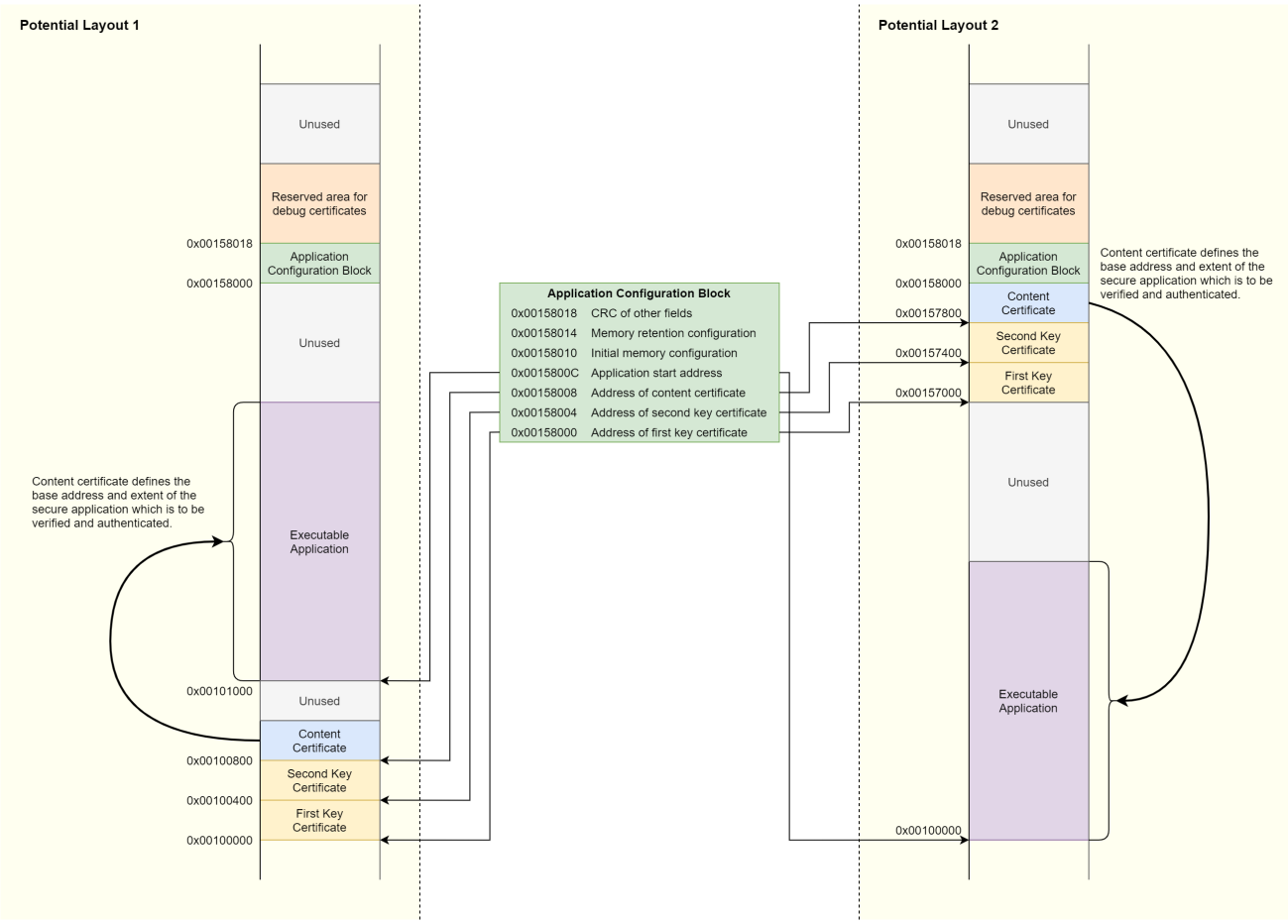

Figure 23. Secure Application Layouts

As can be seen from the diagram, the only fixed data is the application configuration sector, which holds the locations of the certificates and program start address. The certificates and program can be placed anywhere in flash, as long as the appropriate addresses are set up correctly.

Any applications started by the ROM in LCS_DM or LCS_SE must have associated Root of Trust certificate chains and be correctly configured for the ROM to be able to execute them. If this information is not available, the ROM cannot start an application, and instead executes a failure state.

7.3.8.3 Creating a Secure Application in LCS_DM using H_{BK0}

The process of creating a secure application is the same, regardless of the HBK hash used or the life cycle state.

In LCS_DM, only the H_{BK0} hash has been provisioned to the device, so this is used to demonstrate the process.

The following actions need to be performed:

- Two RSA keys are required to generate the key certificate chain.
- Two key certificates are required to establish the Root of Trust certificate chain.
    - The first key certificate is signed by the Root of Trust key HBK0 and provides the key 1 public key.
    - The second key certificate is signed by the key 1 and provides the key 2 public key.
- The content certificate defines the loadable, verified, and authenticated sections of the binary image.
    - The content certificate is signed by key 2.
- The two key certificates, the content certificate, and the application image are combined with a loadable application configuration block to create the loadable binary.

```
          /* This assumes the device has been transitioned to LCS_DM, and */
          /* suitable debug certificates have been loaded to unlock the device. */
          /* It makes use of previously created HBK0 Root of Trust certificates. */

 /* Create the key certificates */

  C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_key_1
 C:\Development\RSLSec>rslsec trust make hbk0 ./assets/keys/hbk0_key_2

 /* Create the key certificates */
 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk0/key_1.crt --hbk
hbk0 --keypair ./assets/keys/hbk0/hbk0.prv.pem --pwd ./assets/keys/hbk0/hbk0.pwd --
pubkey ./assets/keys/hbk0_key_1/hbk0_key_1.pub.pem

 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk0/key_2.crt --hbk
hbk0 --keypair ./assets/keys/hbk0_key_1/hbk0_key_1.prv.pem --pwd ./assets/keys/hbk0_key_
1/hbk0_key_1.pwd --pubkey ./assets/keys/hbk0_key_2/hbk0_key_2.pub.pem

 /* Create the content certificate */
 C:\Development\RSLSec>rslsec trust cert content --out ./assets/cert/hbk0/content.crt --
keypair ./assets/keys/hbk0_key_2/hbk0_key_2.prv.pem --pwd ./assets/keys/hbk0_key_2/hbk0_
key_2.pwd --image ./images/RSL15/blinky/blinky.hex --key1 --key2 --target RSL15

 /* Pack the signed image with the certificates */
 C:\Development\RSLSec>mkdir .\assets\apps
```

```
 C:\Development\RSLSec>mkdir .\assets\apps\RSL15

 C:\Development\RSLSec>mkdir .\assets\apps\RSL15\hbk0

 C:\Development\RSLSec>rslsec trust pack --out ./assets/apps/RSL15/hbk0/blinky.hex --
key1 ./assets/cert/hbk0/key_1.crt --key2 ./assets/cert/hbk0/key_2.crt --content
./assets/cert/hbk0/content.crt --image ./images/RSL15/blinky/blinky.hex --target RSL15 -
-firstKeyAddress 0x00159000 --secondKeyAddress 0x00159400 --contentAddress 0x00159800

 /* Load the image */
 C:\Development\RSLSec>rslsec util load ./assets/apps/RSL15/hbk0/blinky.hex --target
RSL15 --write
```

7.3.8.4 Creating a Secure Application in LCS_SE using H_{BK1}

In LCS_SE, the H_{BK0} and H_{BK1} hashes have been provisioned to the device so either can be used to authenticate an application. In this example the H_{BK1} is used.

The following actions need to be performed:

- Two RSA keys are needed to generate the key certificate chain.
- Two key certificates are required to establish the Root of Trust certificate chain.
  - The first key certificate is signed by the Root of Trust key H_{BK1} and provides the key 1 public key.
  - The second key certificate is signed by the key 1 and provides the key 2 public key.
- The content certificate defines the loadable, verified and authenticated sections of the binary image.
  - The content certificate is signed by key 2.
  - At the time of creating the content certificate, you need to specify whether key certificate 1 and key certificate 2 are going to be used. This information is required during certificate creation to make sure that the certificate size is correct.
- The two key certificates, the content certificate, and the application image are combined with a loadable application configuration block to create the loadable binary.

```
        /* This assumes the device has been transitioned to LCS_SE, and suitable */
        /* devbug certificates have been loaded to unlock the device. */
        /* It makes use of previously created HBK1 Root of Trust certificates. */

 /* Create RSA keys to be used when creating the three certificate RoT chain. */
 C:\Development\RSLSec>rslsec trust make hbk1 ./assets/keys/hbk1_key_1

 C:\Development\RSLSec>rslsec trust make hbk1 ./assets/keys/hbk1_key_2

 /* Create the key certificates */
 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk1/key_1.crt --hbk
hbk1 --keypair ./assets/keys/hbk1/hbk1.prv.pem --pwd ./assets/keys/hbk1/hbk1.pwd --
pubkey ./assets/keys/hbk1_key_1/hbk1_key_1.pub.pem>
 C:\Development\RSLSec>rslsec trust cert key --out ./assets/cert/hbk1/key_2.crt --hbk
hbk1 --keypair ./assets/keys/hbk1_key_1/hbk1_key_1.prv.pem --pwd ./assets/keys/hbk1_key_
1/hbk1_key_1.pwd --pubkey ./assets/keys/hbk1_key_2/hbk1_key_2.pub.pem

 /* Create the content certificate */
```

```
 C:\Development\RSLSec>rslsec trust cert content --out ./assets/cert/hbk1/content.crt --
keypair ./assets/keys/hbk1_key_2/hbk1_key_2.prv.pem --pwd ./assets/keys/hbk1_key_2/hbk1_
key_2.pwd --image ./images/RSL15/blinky/blinky.hex --key1 --key2 --target RSL15

 /* Pack the signed image with the certificates */
 C:\Development\RSLSec>mkdir .\assets\apps\RSL15\hbk1

 C:\Development\RSLSec>rslsec trust pack --out ./assets/apps/RSL15/hbk1/blinky.hex --
key1 ./assets/cert/hbk1/key_1.crt --key2 ./assets/cert/hbk1/key_2.crt --content
./assets/cert/hbk1/content.crt --image ./images/RSL15/blinky/blinky.hex --target RSL15 -
-firstKeyAddress 0x00159000 --secondKeyAddress 0x00159400 --contentAddress 0x00159800

 /* Load the image */
 C:\Development\RSLSec>rslsec util load ./assets/apps/RSL15/hbk1/blinky.hex --target
RSL15 --write
```

LCS_SE and HBK0/HBK1

In LCS_SE, it is possible to boot from either Root of Trust, so an application signed with H_{BK0} can still be executed successfully in LCS_SE.

One or two key certificates

The descriptions above show the process when two key certificates are used to provide additional protection to the Root of Trust keys. It is also possible to use a single key certificate, signed with the $H_{BK0/1,}$ and provide the public key of the key pair used to sign the content certificate. This mode is valid and results in fast power-up times from a power on reset condition. If you are only using one key certificate, you need to set `--key1` or `--key 2` to `false` when generating the content certificate. `--key1` and `--key2` are used during the generation of the content certificate to set the certificate size.

7.4 Transition from EH_STATE to ROT_STATE

7.4.1 Revoking EH_STATE

Revoking access to the EH_STATE is straightforward and follows a similar process to that outlined for other EH_STATE operations. There are no options specific to this command (though the options common to all commands are still applicable). The revocation action cannot be reversed once it has completed.

Example Usage

```
 c:\Development\RSLSec>rslsec eh revoke --write
```

This command connects to an RSL15 device and update the EH_STATE configuration to render it invalid. On completion, the device is reset and has transitioned to ROT_STATE in LCS_CM.

NOTE: This action is not reversible.

```
 High level help from 'RSLSec eh revoke --help'
 c:\Development\RSLSec>rslsec eh revoke --help

 //usage:
  RSLSec eh revoke [-h] [--out OUT] [--target TARGET] [--write]

 //Revoke LCS_EH, transition to LCS_CM
```

```
//optional arguments:
  -h, --help       show this help message and exit
  --out OUT        File to which the loadable package is being written
  --target TARGET  Target connection [RSL15]
  --write          Update the attached target with the given options
```

Windows is a registered trademark of Microsoft Corporation. Arm, Cortex, Keil, and uVision are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All other brand names and product names appearing in this document are trademarks of their respective holders.

IAR Embedded Workbench is a registered trademark of IAR Systems AB.

onsemi and the onsemi logo are trademarks of Semiconductor Components Industries, LLC dba onsemi or its subsidiaries in the United States and/or other countries. onsemi owns the rights to a number of patents, trademarks, copyrights, trade secrets, and other intellectual property. A listing of onsemi's product/patent coverage may be accessed at www.onsemi.com/site/pdf/Patent-Marking.pdf. onsemi is an Equal Opportunity/Affirmative Action Employer. This literature is subject to all applicable copyright laws and is not for resale in any manner.

Copyright 2023 Semiconductor Components Industries, LLC ("onsemi"). All rights reserved. Unless agreed to differently in a separate onsemi license agreement, onsemi is providing this "Technology" (e.g. reference design kit, development product, prototype, sample, any other non-production product, software, design-IP, evaluation board, etc.) "AS IS" and does not assume any liability arising from its use; nor does onsemi convey any license to its or any third party's intellectual property rights. This Technology is provided only to assist users in evaluation of the Technology and the recipient assumes all liability and risk associated with its use, including, but not limited to, compliance with all regulatory standards. onsemi reserves the right to make changes without further notice to any of the Technology.

The Technology is not a finished product and is as such not available for sale to consumers. Unless agreed otherwise in a separate agreement, the Technology is only intended for research, development, demonstration and evaluation purposes and should only be used in laboratory or development areas by persons with technical training and familiarity with the risks associated with handling electrical/mechanical components, systems and subsystems. The user assumes full responsibility/liability for proper and safe handling. Any other use, resale or redistribution for any other purpose is strictly prohibited.

The Technology is not designed, intended, or authorized for use in life support systems, or any FDA Class 3 medical devices or medical devices with a similar or equivalent classification in a foreign jurisdiction, or any devices intended for implantation in the human body. Should you purchase or use the Technology for any such unintended or unauthorized application, you shall indemnify and hold onsemi and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that onsemi was negligent regarding the design or manufacture of the board.

The Technology does not fall within the scope of the European Union directives regarding electromagnetic compatibility, restricted substances (RoHS), recycling (WEEE), FCC, CE or UL, and may not meet the technical requirements of these or other related directives.

THE TECHNOLOGY IS NOT WARRANTED AND PROVIDED ON AN "AS IS" BASIS ONLY. ANY WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE HEREBY EXPRESSLY DISCLAIMED.

TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL ONSEMI BE LIABLE TO CUSTOMER OR ANY THIRD PARTY. IN NO EVENT SHALL ONSEMI BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY NATURE WHATSOEVER (INCLUDING, BUT NOT LIMITED TO, LOSS OR DISGORGEMENT OF PROFITS, LOSS OF USE AND LOSS OF GOODWILL), REGARDLESS OF WHETHER ONSEMI HAS BEEN GIVEN NOTICE OF ANY SUCH ALLEGED DAMAGES, AND REGARDLESS OF WHETHER SUCH ALLEGED DAMAGES ARE SOUGHT UNDER CONTRACT, TORT OR OTHER THEORIES OF LAW.

Do not use this Technology unless you have carefully read and agree to these limited terms and conditions. By using this Technology, you expressly agree to the limited terms and conditions. All source code is onsemi proprietary and confidential information.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:

Literature Distribution Center for onsemi

19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA

Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada

Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada

Email: orderlit@onsemi.com

N. American Technical Support:

800-282-9855 Toll Free USA/Canada

Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

onsemi Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative

M-20893-002