



# Arm<sup>®</sup> RSE in Arm<sup>®</sup> Neoverse<sup>®</sup> CSS V3

Revision r1p2

## Operational Guidance

**Non-Confidential**

Copyright © 2025–2026 Arm Limited (or its affiliates).  
All rights reserved.

**Issue 02**

111266\_r1p2\_02\_en



## Arm® RSE in Arm® Neoverse® CSS V3 Operational Guidance

This document is Non-Confidential.

Copyright © 2025–2026 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (111266\_r1p2\_02\_en) was issued on 2026-03-02. There might be a later issue at <https://developer.arm.com/documentation/111266>

The product revision is r1p2.

See also: [Proprietary notice](#) | [Product and document information](#) | [Useful resources](#)

### Start reading

If you prefer, you can skip to [the start of the content](#).

### Intended audience

This guidance is targeted integrators and users of RSE.

### Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email [terms@arm.com](mailto:terms@arm.com).

### Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>.

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

# Contents

<b>1. RSE Overview.....</b>	<b>6</b>
1.1 Product deliverables.....	7
1.2 RSE hardware features.....	8
1.3 OTP Usage and Rationale.....	10
1.4 Cryptographic Device Integration.....	11
1.5 Hash Usage Guidance.....	11
1.6 Software development on RSE.....	12
<b>2. RSE software functionality.....</b>	<b>13</b>
2.1 Functionality overview.....	14
2.2 Pre-software boot.....	14
2.2.1 DMA boot in CM life cycle state (Virgin mode) and RMA life cycle state.....	16
2.2.2 DMA boot in CM lifecycle state, non-virgin modes.....	17
2.2.3 DMA boot in DM lifecycle state.....	22
2.2.4 DMA boot in SE lifecycle state.....	25
2.2.5 DMA common end procedure in ROM.....	27
2.2.6 Boot stages signaling.....	28
2.3 Lifecycle and provisioning workflow.....	29
2.3.1 Platform lifecycle management.....	29
2.3.2 Secure system boot.....	30
2.3.3 Multi-subsystem boot concept.....	33
2.3.4 Device attestation.....	36
2.3.5 Assets provisioning and management.....	37
2.4 Software access to system memory and peripherals.....	44
<b>3. Hardware modules operational guidance.....</b>	<b>45</b>
3.1 Lifecycle Manager.....	45
3.1.1 SoC lifecycle management.....	46
3.1.2 LCM and RSE boot process.....	47
3.2 DMA-350.....	48
3.2.1 DMA operations during early boot.....	48
3.2.2 Common end procedure.....	49
3.3 Integrity checker.....	49

3.3.1 Integrity checker usage by DMA during boot.....	49
3.3.2 Integrity checker usage by software in runtime.....	50
3.4 Address Translation Unit.....	52
3.5 Key Management Unit.....	52
3.5.1 KMU PRBG seed configuration.....	54
3.5.2 Verification of hardware key slots.....	55
3.5.3 Programming software key slots.....	55
3.5.4 Exporting a key from a key slot.....	56
3.5.5 Invalidating a key slot.....	56
3.6 Integrating third-party cryptographic engines.....	57
3.6.1 Hardware attachment requirements.....	57
3.6.2 KMU and key handling.....	57
3.6.3 Boot ROM customization hooks.....	58
3.6.4 Driver and firmware expectations.....	58
3.6.5 Validation checklist.....	58
3.7 RSE integrated cryptographic engines.....	59
3.7.1 Hash.....	59
3.7.2 Advanced Encryption Standard.....	60
3.7.3 Public Key Accelerator.....	60
3.8 SRAMs.....	68
3.8.1 Instruction tightly coupled memory.....	69
3.8.2 Data tightly coupled memory.....	69
3.8.3 L1 instruction cache.....	72
3.8.4 L1 Data Cache.....	72
3.8.5 Volatile memory.....	72
3.8.6 Prevention of ECC errors caused by speculative cache prefetching.....	77
3.9 TRNG and DRBG.....	79
3.10 Message handling unit.....	79
3.10.1 Integrity.....	80
3.10.2 Confidentiality.....	80
3.10.3 Replay protection.....	80
3.11 Security Alarm Manager.....	81
3.11.1 Handling of RSE cold reset initiated by SAM.....	87
3.12 Persistent state interface.....	90
3.13 Local system counter.....	91

<b>4. RSE debug scenarios.....</b>	<b>92</b>
<b>5. Software countermeasures.....</b>	<b>93</b>
5.1 Fault injection and perturbation.....	93
5.2 Memory dump prevention.....	94
5.3 Handling detected faults.....	94
5.4 Limited Fault Tolerance counter.....	94
5.5 Side-channel protection.....	95
5.6 Lightweight random generation.....	96
5.7 Address Translation Unit as a firewall.....	96
<b>6. RSE Handshake Flow.....</b>	<b>97</b>
<b>Proprietary notice.....</b>	<b>100</b>
<b>Product and document information.....</b>	<b>102</b>
Product status.....	102
Revision history.....	102
Conventions.....	103
<b>Useful resources.....</b>	<b>106</b>

# 1. RSE Overview

Arm® Runtime Security Engine (RSE) is a security subsystem. It serves as the system-on-chip (SoC) hardware Root of Trust (RoT) and an isolated attestation enclave. It provides an isolated execution environment for sensitive processes and data.

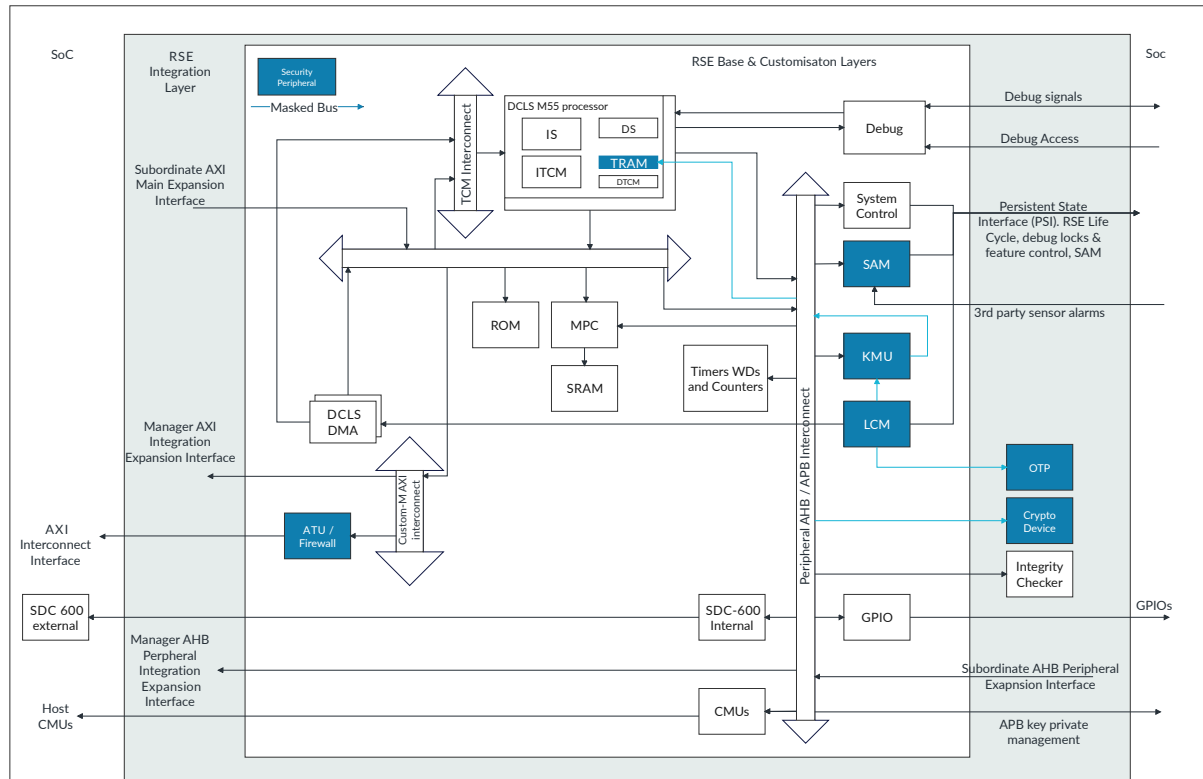
The RSE fulfills the requirements of the Arm Confidential Compute Architecture (CCA) Hardware Enforced Security (HES) specification. It is defined in the Arm CCA Security Model 1.0. It targets Android Protected VMs using the Google Boot Certificate Chain (BCC) attestation scheme. For this purpose, it implements the [Trusted Computing Group \(TCG\) Device Identifier Composition Engine \(DICE\) Protection Environment \(DPE\)](#) specification.

RSE is designed to protect the most sensitive assets of the SoC. It includes countermeasures against physical side channels and fault injection attacks. The RSE is designed to be compliant with the SESIP Profile for Platform Security Architecture (PSA) Certified™ Level 3. It also targets Common Criteria Vulnerability Analysis AVA\_VAN.3 and higher.

RSE includes the following main features:

- Provides platform life cycle management and applies security policies associated with each life cycle state
- Controls SoC debug features
- The RSE, with Trusted Firmware-M (TF-M) running on it, manages the SoC boot process. It authenticates and loads the initial software components, then collects measurements for all subsequent components.
- The RSE, with Trusted Firmware-M (TF-M), provides attestation services for the SoC in various formats.
- Provides a high level of protection against tampering, fault injection, side-channel attacks, and other malicious attacks. It uses defense-in-depth with hardware and software mitigations.
- In some configurations, RSE allows an application to run from memory that is outside the boundaries of the RSE. It provides security mechanisms that protect the storage and execution of applications and their data in remote memory. Examples include Quad SPI (QSPI) flash or dynamic random-access memory (DRAM).

The following diagram provides an overview of the boundary between RSE and SoC.

**Figure 1-1: Boundary between RSE and SoC**

Arm RSE delivery includes the Base and Customization layers. The integrator adds an Integration Layer. After integration is complete, the Base, Customization, and Integration layers together form the RSE. The security boundaries of the product as Root of Trust are the boundaries of the RSE Integration Layer.

For more information on Integration Layer and integrating RSE into the SoC, see your SoC or Compute Subsystem (CSS) reference manual.

To ensure that RSE functions as a Root of Trust and is operated securely, software developed on RSE must follow functional and security guidance. This document provides that guidance.

## 1.1 Product deliverables

The RSE deliverables are RTL, documentation, and open-sourced TF-M software.

RTL supplied for RSE contains Base and Customization layers. The integrator is expected to implement RSE Integration layer by adding peripherals, such as cryptographic engines. After the RSE Integration Layer development is complete, the RSE security boundaries are defined by the Integration Layer. The complete RSE includes the Base, Customization, and Integration layers.

After the development of the full RSE is completed by the integrator, RSE is expected to be integrated into a SoC. It will serve as a Root of Trust (RoT) and an isolated secure enclave.

## 1.2 RSE hardware features

The RSE hardware includes the following components:

### Single Arm® Cortex®-M55 CPU core

- An Arm V8-M processor with:
  - M-Profile Vector Extension (MVE)
  - Floating Point Unit (FPU)
  - Digital Signal Processing (DSP) extensions
  - Caches
  - Tightly Coupled Memories (TCMs) with ECC protection
  - Implementation Defined Attribution Unit (IDAU), and ETM, working in Dual-Core Lock Step (DCLS)

### Processor memory subsystem

- ROM
- Two banks of System Volatile Memory, SRAMs with:
  - Memory Protection Controllers (MPC)
  - Exclusive Access Monitor (EAM)
  - ECC protection

### Security peripherals and mechanisms

- Arm® CoreLink™ DMA-350 Controller with Implementation Defined Attribution Unit (IDAU) and Manager Security Controller (MSC). Operates in Dual-Core Lock Step (DCLS)
- Arm® Lifecycle Manager (LCM) v1.0
- Arm® Key Management Unit (KMU) v1.0
- Arm® Security Alarm Manager (SAM) v1.0
- Arm® Address Translation Unit (ATU) v1.0
- Memory Protection Unit (MPU) inside the Cortex-M55 CPU
- Arm® TRAM v1.0 (encryption and address scrambling for the DTCM)
- Arm® CoreSight™ secure debug channel
- Integrity Checker with Manager Security Controller (MSC)

### Additional Integration

For Compute Subsystems (CSS), additional integration is required, specifically:



- Non-volatile, One Time Programmable (OTP) memory for keys, credentials, configuration, data, and code
- In OTP, plan to store BL1-2 (ROM code patch and expansion) and DMA descriptor expansions. Basic DMA descriptors reside in ROM. See [OTP Usage and Rationale](#) for details of:
  - OTP artifacts, criteria, and lifecycle handling.
  - True Random Number Generator (TRNG) source; store TRNG provisioning records and DRBG seeding metadata in OTP (runtime entropy comes from TRNG hardware)
- A cryptographic accelerator

The RSE also integrates the Arm PrimeCell UART (PL011) controller for serial port logging.

The physical interfaces of RSE are listed in [Table 1-1: Physical interfaces](#) on page 9

**Table 1-1: Physical interfaces**

Function	Description
Memory extension	RSE can access selected memory locations in the SoC address space, including registers, SRAM, flash, and (if present) DRAM. The exact interfaces and accessible ranges depend on the SoC integration and system configuration. These locations are outside RSE and are therefore considered to be untrusted.
Combined Message Unit (CMU)	CMU is a bidirectional mailbox-like interface composed of two unidirectional MHUs: Host Receiver (downstream to RSE) and Host Sender (upstream to the SoC). It is used for command, event, and data exchange between RSE firmware and SoC software.
Debug	Two debug access interfaces, which are disabled and locked when RSE exits from reset. They automatically unlock during certain life cycle stages. RSE firmware can also unlock them. See <a href="#">RSE debug scenarios</a> for more details.
GPIO	Eight general-purpose I/O ports are provided for various indications sent from the SoC to RSE, and from the RSE to the SoC.
SoC timestamp	Allows the RSE to track the SoC system counter state at the RSE SoC System Timer. The RSE does not rely on this timer value as secure.
Persistent State Interface (PSI)	The PSI is an interface through which the RSE exposes to the SoC its state. It includes: <ul style="list-style-type: none"> <li>• Output of Life Cycle State (LCS)</li> <li>• Debug enable and features control for the SoC</li> <li>• SAM status signals</li> <li>• Status signals from General Purpose Retention Register (GRETREG)</li> <li>• DMA-350 State output signals</li> </ul>
Power and Clock Control	Q-Channel interfaces for power control and clock control.
Resets and Clocks	Includes reset and clock signals.
Debug Expansion	Includes debug timestamp, trigger, trace and debug communication signals.
APB key private management	Provides to the SoC an optional symmetric key for its encryption or content protection needs.

## 1.3 OTP Usage and Rationale

OTP stores immutable or append-only artifacts that underpin the Root of Trust (RoT). Lifecycle handling includes provisioning records and monotonic counters that protect against rollback.

### Purpose

Use OTP for immutable or append-only artifacts that must survive resets and cannot be modified without lifecycle transitions or secure update flows.

### Artifacts

- Keys and credentials required for boot and attestation
- Configuration values and fuses controlling security posture
- Data and code that extend ROM functionality:
  - BL1-2 (ROM code patch and expansion)
  - DMA descriptor expansions (basic DMA descriptors are in ROM)
- Entropy-related metadata and records:
  - True Random Number Generator (TRNG) provisioning records and health-test outcomes
  - Deterministic Random Bit Generator (DRBG) seeding metadata and monotonic counters (runtime entropy comes from TRNG hardware)

### Selection Criteria

- Immutability: Artifact must not require arbitrary rewrites
- Security impact: Changing the artifact could weaken the Root of Trust
- Size constraints: Fits within OTP capacity and alignment requirements
- Recoverability: Has a defined fallback when OTP entries are absent

### Lifecycle Handling

- Tamper and rollback protection: Include versioning and monotonic counters to prevent downgrades
- Append-only updates: Use new slots rather than rewriting existing entries
- Auditability: Record provenance and integrity data linked to lifecycle state

### References

- For details of OTP fits within the overall hardware integration requirements, see [RSE hardware features](#).

## 1.4 Cryptographic Device Integration

Guidance for integrating platform cryptographic accelerators covers initialization, entropy sourcing, KMU-based key management, and lifecycle behavior.

### Initialization and Entropy

Ensure devices are initialized early in boot as required, and source runtime entropy from TRNG hardware. Store provisioning records and DRBG seeding metadata in OTP; do not store entropy in OTP.

### Symmetric and Public-key Functions

Provide drivers that meet side-channel requirements and use KMU export flows for key provisioning. Validate AES/PKA operations with known-answer tests.

### Security and Lifecycle

Connect alarms to SAM, respect lifecycle state transitions, and document reset behavior so secrets are cleared appropriately.

## 1.5 Hash Usage Guidance

Hash usage in RSE covers safe patterns and pitfalls, with recommended alternatives for integrity and authentication.

### Restrictions

Avoid using simple hash functions directly for integrity protection, authentication, or key derivation without appropriate constructions (HMAC, KDFs). Use authenticated encryption or MACs for integrity/authentication.

### Allowed Uses

- Non-cryptographic checksums for transport or storage where tamper resistance is not required.
- As part of HMAC, HKDF, or other standardized compositions.

### Alternatives

- Use AEAD modes (for example, AES-GCM) for combined confidentiality and integrity.
- Use HMAC for message authentication and integrity when encryption is not required.

### Operational Checklist

- Confirm that any use of hash functions in software follows standardized constructions.
- Document rationale and threat model when deviating from recommended patterns.

## 1.6 Software development on RSE

RSE is an isolated execution environment inside the SoC. It serves as the Root of Trust (RoT) for boot and measurement, and acts as an attestation enclave.

This document describes the main functions of the RSE and explains how to implement them using RSE hardware.

Consider these guidelines when implementing software securely on the RSE and enabling communication with other SoC components.

## 2. RSE software functionality

RSE software implements several functions that are essential to its role as the Root of Trust (RoT). It also includes additional platform-specific functionality.

Typical functions implemented by the RSE software include the following.

- Management of platform lifecycle and associated security policies
- Management of platform long-term keys in One-Time-Programmable (OTP) memory
- Provisioning of assets to the OTP memory
- Boot management
  - Pre-software boot of the RSE
  - The RSE secure boot
  - Orchestrating secure boot of the platform
- Device attestation and sealing services Platform Security Architecture (PSA) and DICE Protection Environment (DPE )
- Cryptographic services
- Debug authentication
- Secure data storage services
- Firmware update services

Arm provides a reference software implementation for the RSE. It is part of [Trusted Firmware-M RSE Platform Documentation](#). The implementation follows the guidance provided in this document. The same guidance should be taken into account when developing your own software solution for the RSE.

The following topics describe the RSE boot process. It includes:

- Pre-software boot
- Software boot
- System boot orchestration
- RSE and platform lifecycle management
- Software security services implemented on the RSE and provided by Trusted Firmware-M (TF-M)

Guidance also covers manufacturing processes such as design-for-test (DFT) procedures and asset provisioning.

## 2.1 Functionality overview

In a typical platform, the RSE boots first from its ROM. It follows a secure boot process to load its first mutable bootloader, which performs platform-specific hardware configuration.

Some platforms are complex and consist of multiple dies, chiplets, or chips. Each subsystem that boots independently contains a separate RSE that manages its lifecycle and boot process as its Root of Trust (RoT). In such platforms, the first mutable bootloader of each RSE performs a handshake with other RSE instances in the system. It verifies their identities. It then constructs a shared virtual identity that combines the contributions of all RSE instances.

The RSE can include more than one mutable bootloader that loads and executes in sequence. For example, the expansion bootloader can reside in OTP, and the RSE runtime firmware can reside in flash.

The RSE boot process follows this sequence:

1. The RSE boot ROM cryptographically verifies and loads the OTP bootloader into SRAM.
2. The OTP bootloader stages the runtime firmware from system flash into a staging area in the SoC, such as SRAM.
3. The runtime firmware performs additional hardware configuration.
4. The runtime firmware launches the secure boot of other processing elements in the platform, such as the System Control Processor (SCP), MCP, LCP, and Application Processor (AP).
5. The RSE loads the initial boot images for these processing elements from system flash, cryptographically verifies them, optionally decrypts them, and loads them into the respective SRAM memories.
6. The RSE enables each processing element to run. It can toggle the CPUWAIT signal on a given PE (such as the SCP). Alternatively, it can signal to the SCP that a specific element can be brought online (such as the AP).

For reference bootloaders of RSE as implemented as part of Trusted Firmware-M (TF-M), refer to [Secure System Boot](#).

After loading the initial boot images of all the relevant processing elements, RSE verifies and loads its own runtime image.

The RSE runtime firmware provides runtime security services. The services can be invoked through Message Handling Unit (MHU) channels from various clients. The MHU peer-to-peer connection guarantees that only authorized clients can communicate with RSE.

## 2.2 Pre-software boot

Unlike a typical M-class execution environment, the RSE boot does not start with processor execution from the Boot ROM. Before the RSE processor is operational, a dedicated DMA engine

initiates the boot. The DMA applies required hardware security configurations before the processor bootloader starts. These configurations ensure that code executes securely.

The DMA engine is protected against fault injections by Dual-Core Lock Step (DCLS). This means the DMA logic is duplicated, and a comparison is applied at each stage. Whenever a mismatch is detected, the RSE raises an alarm and reports it to the Security Alarm Manager (SAM).

The DMA can be programmed to perform a series of operations. Branching between operation series can be chosen by triggers. Triggers originate from the Lifecycle Manager (LCM), the Security Alarm Manager (SAM), and the Integrity Checker. These triggers select the DMA sequence based on the current lifecycle state and Test or Production (TP) mode. The selection also depends on the progress of SAM and the Integrity Checker.

In life cycle states related to security enabled (SE), the DMA copies into the SAM register configuration using values stored in the OTP. The DMA is capable of setting trim values to integrator's additional secure sensors to assure that the RSE processor would boot fully protected.

This topic describes example boot flows in various lifecycle states and modes. It covers the DMA flow, access using debug interfaces, and initial processor flows.

In life cycle states related to provisioning (CM and DM), some boot steps involve the user using a debug interface to load blobs. These blobs contain an encrypted provisioning program and encrypted assets for RSE SRAM.

**Table 2-1: DMA Triggers implemented by RSE to control DMA boot flow**

DMA Trigger ID	Description
Trigger 0	LCM Diagnostic Mode.  This trigger is set when the LCM detects a virgin OTP or when LCS=RMA. A virgin OTP implicitly means LCS=CM. When Trigger 0 is set, Triggers 1, 2, and 3 cannot be set.
Trigger 1	LCM Mission Mode.  This trigger is set when LCM!=RMA. This implies the OTP is not virgin. When Trigger 1 is set, either Trigger 2 or Trigger 3 can be set.
Trigger 2	LCM Mission-Provisioning Mode (Secure Provisioning). This trigger is set when the LCM detects that LCS=CM or when the LCS=DM. When Trigger 2 is set, Trigger 3 cannot be set.
Trigger 3	LCM Mission-Security-Enabled Mode. This trigger is set when the LCM detects that LCS=SE. When Trigger 3 is set, Trigger 2 cannot be set.
Trigger 4	Security Alarm Manager (SAM) configuration loaded successfully. This trigger is set when the SAM integrity checker completes its operation successfully as part of the flows which follow Trigger 3.
Trigger 5	Integrity Checker completed successfully. This trigger is set when the integrity checker completes its operation successfully as part of the flows which follow Trigger 3.

## 2.2.1 DMA boot in CM life cycle state (Virgin mode) and RMA life cycle state

When the RSE is in a virgin (blank) state, external systems cannot supply the required provisioning information (assets or blobs). Other system components depend on a functional RSE to boot. Therefore, the testing and provisioning flow uses JTAG. JTAG allows the user to load provisioning blobs or programs into RSE SRAM. The RSE can execute them following a warm reset.

Boot in the Chip Manufacturing (CM) lifecycle state in virgin mode is similar to boot in the Return Merchandise Authorization (RMA) lifecycle state. In both cases, no assets are stored in OTP or in other memories and peripherals, such as the LCM, KMU, or crypto engines.

After the RSE exits virgin mode, the DFT capabilities used for chip testing at the fabrication facility are disabled. Debug and DFT capabilities are re-enabled once the LCS is set to RMA.

The following steps describe the pre-software boot flow orchestrated by the DMA in virgin mode when the lifecycle state is Chip Manufacturing or RMA. Branches in the flow are implemented using DMA channels that use triggers to signal specific events.

### LCS is CM, Virgin mode

This section describes the DMA/LCM boot flow in CM virgin mode.



The DMA uses GPIO[4:0] on each channel to signal debug visibility before the RSE processor runs. The RSE processor also uses its own GPIO[3:0] for debug visibility to signal boot progress. For more information see [Boot stages signaling](#).

---

1. The RSE exits from cold reset. The Debug Control Units (DCUs) are closed.
  2. The processor is held at CPUWAIT, LCM starts, the DMA starts its ROM flow.
  3. The LCM reads the OTP and determines the LCS (CM) and that the chip is virgin.
  4. The DCUs are open, but DCU force disable register (LCM\_DCU\_FORCE\_DISABLE) blocks JTAG and any other debug access
  5. Trigger 0 (LCM Diagnostic Mode) is set.
- 



From this stage DFT can be executed.

---

6. DMA P0 runs.
7. DMA P0 sets DMA Boot Enable to 0 to prevent rerun of the DMA in a warm reset.
8. DMA P0 starts P1 in channel 1 which starts waiting for trigger 0.
9. DMA P0 branches into P2 in channel 0 which waits for trigger 1.
10. DMA P1 is triggered (trigger 0) into Test Flow.



11. DMA P1 stops channel P0 execution.
12. DMA P1 branches to the [DMA common end procedure in ROM](#).
13. The processor boots from the ROM
14. The processor ROM code detects that the LCS is CM, chip is virgin.
15. The processor sets GPIO[3:0] = 0x1 to indicate Virgin chip detected - Boot ROM is idle.
16. The processor loops forever (or WFI).



At this stage the debugger can debug RSE.

---

### LCS is RMA

This section describes ROM actions when LCS is RMA.

1. The processor ROM code detects that the LCS is RMA.
2. The processor ROM wipes CM and DM assets (keys) from the OTP (writes 0xFF over all bytes of the current assets). This is a security requirement.
3. The processor sets GPIO[3:0] = 0x3 to indicate RMA chip detected - Boot ROM is idle.
4. The processor loops forever (or WFI).



At this stage the debugger can debug RSE.

---

## 2.2.2 DMA boot in CM lifecycle state, non-virgin modes

The DMA boot flow executes when the RSE is in Chip Manufacturing (CM) lifecycle state and the chip is in non-virgin mode (TCI or PCI).

The boot flow covers JTAG-driven TP mode configuration, DMA and LCM initialization, and the secure provisioning sequence.

### Preconditions

The following conditions apply:

- LCS is CM and the chip is in non-virgin mode (TCI or PCI). The OTP is programmed and no longer blank.
- If secure provisioning starts after the SoC is powered off, the RSE executes the boot flow for CM (virgin mode) and RMA lifecycle states.

## JTAG user starts CM provisioning step 1 (set TP Mode)

This section introduces the JTAG-driven TP mode configuration for CM non-virgin.

1. JTAG stops Processor execution.
2. JTAG sets the 'TP Mode Config' word in the OTP to PCI or to TCI.



After the next LCM reset, the LCM will detect that it is in non-virgin mode. This operation can be done directly by the JTAG, or using a program that the JTAG loads into RSE SRAM and starts its execution.

3. JTAG performs cold reset to RSE which also resets the LCM which finds that the RSE is not virgin anymore.
4. After the reset, the LCM disables the DCUs, which disconnects JTAG.
5. The LCM detects that the device is no longer virgin. The DFT scan is permanently disabled unless TP mode is set to TCI.
6. The user can power off the SoC or RSE. When secure provisioning is required later, powering up the RSE triggers a cold reset that continues from Step 1 of the non-virgin CM provisioning flow.

## LCS is CM, Non-Virgin chip (either TCI or PCI)

This section describes the DMA/LCM boot flow in CM non-virgin.



The DMA uses its GPIO[4:0] for each channel to signal debug visibility before the RSE processor is running. The RSE processor also uses its own GPIO[3:0] to signal boot progress. For more information, see [Boot stages signaling](#).

1. The RSE exits cold reset.
2. The DCUs are closed.
3. The Processor is held at CPUWAIT, LCM starts, DMA starts its ROM flow.
4. The LCM reads the OTP and determines the LCS (CM) and that the chip is non-virgin.
5. The DCUs are open, but DCU Force Disable blocks JTAG access. DFT cannot be executed. Trigger 1 (LCM Mission mode) and Trigger 2 (LCM Mission-Provisioning mode) are set.
6. DMA P0 runs.
7. DMA P0 sets DMA Boot Enable to 0 to block rerun of the DMA in a warm reset.
8. DMA P0 starts P1 in channel 1 which starts waiting for trigger 0.
9. DMA P0 branches into P2 in channel 0 which waits for trigger 1.
10. DMA P2 is triggered (trigger 1 from step 5) into Non-Test Flow. P2 runs.
11. DMA P2 stops P1 execution.
12. DMA P2 starts the execution of P3 in channel 1 which waits for trigger 2.

13. DMA P2 branches into P4 in channel 0.
14. DMA P3 is triggered (trigger 2 from step 5) into Provisioning Flow. P3 runs.
15. DMA P3 stops P4 in channel 0.
16. DMA set INITSVTOR0 to its reset value with lock bit set.
17. DMA P3 branches to the [DMA common end procedure in ROM](#).
18. The Processor boots off the ROM.
19. The Processor ROM code finds in the Syndrome register that the reset is due to cold reset.
20. The Processor ROM code finds that the LCS is CM, non virgin and has nothing else to do.
21. The Processor sets GPIO[3:0] = 0x2 – Non virgin chip detected - Boot ROM is idle.
22. The Processor loops forever (or WFI).

### JTAG user starts CM provisioning step 2 (load CM secure provisioning blob into RSE SRAM)

This section covers loading the CM provisioning blob and initiating SP.

Preconditions: The LCS is CM on a production (PCI) or test (TCI) chip. If the SoC powers up for provisioning after TP mode is set and shelving is complete, provisioning resumes. In that case, the flow starts at Step 1 in [LCS is CM, Non-Virgin chip \(either TCI or PCI\)](#).

1. JTAG stops Processor execution.
2. The JTAG copies a CM provisioning package (blob) to either:
  - An address known to the ROM (for example, VM0)
    - If the target location for the blob is unknown to the ROM, save the address to GRETEXREG. During the SP-active boot flow, the ROM can use GRETEXREG as a pointer to the package.
3. This package is authenticated and encrypted using a key derived from the RTL Key (KRTL). It includes:
  - The CM provisioning tool and assets – CM keys (GUK, KP\_CM, KCE\_CM)
  - The CM ROTPK digest (to be written to OTP)
  - BL1-2 for the CPU (to be written to OTP)
  - DMA programs (to be written to OTP)
4. JTAG copies a secure provisioning initiation program into RSE VM0 or ITCM and starts its execution.



The following steps must be performed by a program that JTAG loads into RSE SRAM. Step 5 disables JTAG until the next cold reset.

---

5. The Secure provisioning initiation program writes the value of DCU\_SP\_DISABLE\_MASK\_VAL (0xAAAA\_AAAA) to LCM's DCU\_EN0 register. Otherwise, the next step would trigger a Debug Signals Security Checker alarm.
6. The secure provisioning initiation program enables secure provisioning (SP) by writing to the LCM SP\_ENABLE register. This must be done by a program running from RSE SRAM. Step 10 cannot be executed from JTAG after this write completes.
7. The LCM requests RSE warm reset (LCMRSTREQ). As part of this reset, the Processor and DMA are reset, but the DMA does not run (boot\_en = 0).
8. The LCM is not reset. The LCM sets the DCUs to DCU\_SP\_DISABLE\_MASK\_VAL (0xAAAA\_AAAA), which disconnects JTAG.
9. Because the Processor is active, it refuses the warm reset unless it executes WFI.
10. Secure provisioning initiation program disables interrupts and executes WFI in a loop.



At this point the Processor is reset.

### LCS is CM, Non-Virgin chip (either TCI or PCI), SP is active

1. RSE Processor, DMA exit from warm reset. TRAM encryption is reset as well but DTCM maintains its content.



DMA does not start any program (boot\_en=0). LCM exports KRTL to the KMU.

2. The processor boots from the ROM (its CPUWAIT was cleared during Step 4 in [DMA common end procedure in ROM](#)).



- The LCM was not reset and maintains its state.
- The DCUs remain disabled by the LCM (unless the chip is TCI).
- The KMU was not reset and maintains its state. It keeps the TRAM key that was set at step 1 in [DMA common end procedure in ROM](#). That procedure was called at step 17 in [LCS is CM, Non-Virgin chip \(either TCI or PCI\)](#).

3. Using LCM registers, the processor determines that LCS=CM on a non-virgin chip (TCI or PCI). It also detects that SP is active. The Syndrome register indicates that the reset is due to secure provisioning.
4. The processor sets GPIO[3:0] = 0x4 – CM Secure provisioning starts.
5. The processor instructs the KMU to export key slot 7 to the TRAM. It then re-enables TRAM encryption. The TRAM registers were reset by the warm reset.

6. The processor checks for a known CM provisioning blob package header at a known address in VM0. Alternatively, it uses the GRETEXREG register as a pointer to that package.
7. The processor authenticates and decrypts the secure provisioning blob from VM0. It loads the program into ITCM and the secrets into DTCM. This uses the crypto device and keys derived from KRTL, which are already stored in the KMU.



This ROM code must be customized for the platform's crypto device. This is a limitation, because the ROM code must know the crypto device.

8. If authentication failed or no blob found, Processor sets GPIO[3:0] = 0x5 (CM Secure provisioning ended with authentication error) and stops (loops forever).



At this stage, DTCM and ITCM contain the authenticated, decrypted blob with the provisioning program and CM key assets. Other provisioning materials are present as assets.

9. ROM jumps into the CM provisioning program in ITCM

## CM provisioning program



This downloaded program must be customized for the platform hardware.

1. The processor randomizes the HUK (using an integrated TRNG source).
2. The processor writes the HUK to the OTP and verifies it.
3. The processor writes the CM ROTPK digest, CM keys, and flags from the package (decrypted into the DTCM) to the OTP.
4. The processor writes BL1-2 from the decrypted package to the OTP.
5. The processor writes DMA programs from the decrypted package to the OTP.
6. If an OTP write error or another error occurs, the processor sets GPIO[3:0] = 0x6 (CM secure provisioning ended with error) and halts. Otherwise, it sets RSE GPIO[3:0] = 0x7 — CM secure provisioning ended successfully.
7. Optionally, the processor may wait in a WFI loop for the user to attach a debugger.
8. The processor performs a cold reset of the RSE.



At this stage, when the LCM completes its reset, the LCS becomes DM.

9. The user may detect RSE GPIO[3:0] (CM flow end) and power off the RSE. If successful, the next power-up can occur in the DM facility.

### 2.2.3 DMA boot in DM lifecycle state

The DMA boot flow in Device Manufacturing (DM) lifecycle state starts after the RSE resets at the end of CM. It can also start after a user-initiated reset (based on a GPIO indication), or when the chip powers on in a DM facility.

#### Preconditions

The following conditions apply:

- LCS is DM; the chip is TCI or PCI (non-virgin)
- CM assets in OTP
- BL1-2 bootloader and DMA programs in OTP
- DFT disabled if TP=PCI; enabled if TP=TCI
- Debug disabled until LCS valid (DM); remains disabled until DMA releases force-disable in the [DMA common end procedure in ROM](#)
- Secrets from the CM provisioning flow may remain in RSE memories if the RSE was reset during provisioning
- Crypto device and KMU reset

#### DM boot flow procedure

The DM boot flow proceeds in two stages.



The DMA uses GPIO[4:0] signals on each channel for debug visibility before the RSE processor starts running. The RSE processor uses GPIO[3:0] signals to indicate boot progress. For more information, see [Boot stages signaling](#).

---

Stage 1: DMA and LCM initialization.

1. The RSE exits cold reset.
2. The DCUs are closed.
3. The processor is held at CPUWAIT while the LCM starts and the DMA starts its ROM flow.
4. The LCM reads the OTP, determines that LCS=DM, and confirms the chip is non-virgin.
5. If the chip is TCI, the DCUs are open. The DCU Force Disable register (LCM\_DCU\_FORCE\_DISABLE) prevents JTAG access. Trigger 1 (LCM Mission mode) and Trigger 2 (LCM Mission-Provisioning mode) are set.
6. DMA P0 runs.
7. DMA P0 sets DMA boot enable to 0 (boot\_en = 0) to prevent the DMA from rerunning after a warm reset.

8. DMA P0 starts P1 on channel 1, which waits for Trigger 0.
9. DMA P0 branches to P2 on channel 0, which waits for Trigger 1.
10. DMA P2 is triggered by Trigger 1 into the non-test flow. P2 runs.
11. DMA P2 stops P1.
12. DMA P2 starts P3 on channel 1, which waits for Trigger 2.
13. DMA P2 branches to P4 on channel 0.
14. DMA is triggered by Trigger 2 into the provisioning flow. P3 runs.
15. DMA P3 stops P4 on channel 0.
16. DMA sets INITSVTOR0 to its reset value with the lock bit set.
17. DMA P3 branches to the [DMA common end procedure in ROM](#).

Stage 2: Processor ROM actions.

1. The processor boots from the ROM.
2. The processor ROM code determines from the Syndrome register that the reset is a cold reset.
3. The processor ROM code determines that LCS = DM and takes no further action.
4. The processor sets GPIO[3:0] = 0x8 — DM: Boot ROM is idle.
5. The processor halts (for example, by entering a WFI loop).

### JTAG user starts DM provisioning step 1 (load DM secure provisioning blob into RSE SRAM)

Introduces JTAG-based DM provisioning.

1. JTAG stops CPU execution.
2. JTAG copies a DM provisioning package (blob) to an address in VM0 known to the ROM. Alternatively, it can copy the package to another SRAM address and save this address pointer in the GRETEXREG register, which is not reset by SP.
3. The provisioning package is authenticated and encrypted using a key derived from KRTL. It includes:
  - A DM provisioning tool
  - Assets (DM keys: KP\_DM, KCE\_DM to be written to the OTP)
  - A firmware update tool (FOTA tool)
4. JTAG copies a secure provisioning initiation program into RSE VM0 or ITCM and starts its execution.



The following steps must be performed by a program that the JTAG loads into RSE SRAM. Step 5 disables JTAG until the next cold reset.

---

5. The secure provisioning initiation program writes the value of DCU\_SP\_DISABLE\_MASK\_VAL (0xAAAA\_AAAA) to the LCM DCU\_EN0 register. Otherwise, the next step would trigger a Debug Signals Security Checker alarm.
6. The secure provisioning initiation program enables secure provisioning (SP) by writing to the SP\_ENABLE register in the LCM.



Note

- LCM requests RSE warm reset (LCMRSTREQ)
- Processor and DMA are reset, but the DMA does not run (boot\_en = 0)
- LCM is not reset
- LCM sets the DCUs to DCU\_SP\_DISABLE\_MASK\_VAL (0xAAAA\_AAAA), which disconnects JTAG
- Processor is active; refuses warm reset unless executing WFI

7. Secure provisioning initiation program disables interrupts and executes WFI in a loop.



Note

At this point the CPU is reset.

### LCS is DM, Non-Virgin chip (either TCI or PCI), SP is active

Describes actions during active DM secure provisioning.

1. The RSE processor, DMA, and KMU exit warm reset.



Note

The DMA does not start any program (boot\_en = 0). The LCM exports KRTL, HUK, and CM keys to the KMU.

2. The processor boots from the ROM (its CPUWAIT was cleared at Step 4 in [DMA common end procedure in ROM](#)).



Note

The LCM is not reset and maintains its state. The DCUs remain closed by the LCM (unless the chip is TCI).

3. Using LCM registers, the processor determines that LCS=DM and that SP is active. The Syndrome register indicates that the reset is due to secure provisioning.
4. The processor sets GPIO[3:0] = 0x9 — DM secure provisioning starts.
5. The processor instructs the KMU to export key 7 to the TRAM again and re-enables TRAM encryption. The TRAM registers were reset by the warm reset.



6. The processor checks for a known DM provisioning blob package header at a known address in VM0. Alternatively, it uses the GRETEXREG register as a pointer to that package.
7. The processor authenticates and decrypts the secure provisioning blob from VM0. It loads the program into ITCM and the secrets into DTCM. This uses the crypto device and keys derived from KRTL, which are already stored in the KMU.



This ROM code must be customized for the platform's crypto device. This is a current limitation because the ROM must recognize the specific device.

8. If authentication fails or no blob is found, the processor sets GPIO[3:0] = 0xA (DM secure provisioning ended with authentication error) and halts.



At this stage, DTCM and ITCM contain the authenticated and decrypted blob. It includes the provisioning program, DM key assets, and other provisioning materials. Treat all of these as assets.

9. The ROM jumps to the DM provisioning program in ITCM.

### DM Provisioning program

The provisioning program performs the following actions.

1. The processor writes the DM keys and flags from the package (decrypted in DTCM) to the OTP.
2. The processor writes BL2 and/or a firmware update tool from the package to system flash, authenticated and encrypted using HUK-derived keys.
3. If an OTP write error or another error occurs, the processor sets GPIO[3:0] = 0xB (DM secure provisioning ended with error) and halts. Otherwise, it sets RSE GPIO[3:0] = 0xC (DM secure provisioning ended successfully).
4. The processor performs a cold reset of the RSE. At this stage, when the LCM completes its reset, the LCS becomes SE.
5. The user can detect RSE GPIO[3:0] (DM flow end) and power off the RSE. If successful, the next power-up can occur in the end user's facility.

## 2.2.4 DMA boot in SE lifecycle state

This stage starts after the RSE resets at the end of DM. It also starts after a user-initiated reset or when the chip powers on in a user facility after it was provisioned.

Preconditions:

- LCS is SE; the chip is TCI or PCI (non-virgin)
- CM assets are in OTP
- BL1-2 bootloader and DMA programs are in OTP

- DM assets are in OTP
  - BL2 and/or firmware update tool are in system flash
  - DFT is disabled (unless the chip is in TCI)
  - Crypto device and KMU are reset.
1. The RSE exits cold reset.
  2. The DCUs are closed.
  3. The processor is held at CPUWAIT while the LCM starts and the DMA starts its ROM flow.
  4. The LCM reads the OTP and determines that LCS=SE.
  5. If the chip is TCI, the DCUs are open; in PCI, they remain closed.
  6. Trigger 1 (Mission mode) and Trigger 3 (Mission–Security Enabled mode) are set.
  7. DMA P0 runs.
  8. DMA P0 sets DMA boot enable to 0 (boot\_en = 0) to prevent the DMA from rerunning after a warm reset.
  9. DMA P0 starts P1 on channel 1, which waits for Trigger 0.
  10. DMA P0 branches to P2 on channel 0, which waits for Trigger 1.
  11. DMA P2 is triggered by Trigger 1 into the non-test flow. P2 runs.
  12. DMA P2 stops P1.
  13. DMA P2 starts P3 on channel 1, which waits for Trigger 2.
  14. DMA P2 branches to P4 on channel 0, which waits for Trigger 3.
  15. DMA P4 is triggered by Trigger 3 into the Security-Enabled 1 flow. P4 (ROM DMA OTP loader) runs.
  16. DMA P4 copies from OTP the DMA program P6 (DMA program 6) to VM0 (fixed size).
  17. DMA P4 starts the Integrity Checker to verify the DMA program in VM0.
  18. DMA P4 waits for Trigger 5.
  19. If the Integrity Checker detects a bad integrity value, it raises an alarm that resets the RSE. Otherwise, the Integrity Checker sets Trigger 5 (Integrity Checker completed successfully).
  20. DMA P4 receives Trigger 5 after a successful integrity check.
  21. DMA P4 jumps to the DMA program P6 in VM0. P6 runs.
  22. DMA P6 copies the SAM configuration from OTP to the SAM.
  23. The SAM performs an integrity check.
    - If the integrity check fails, the SAM raises an alarm that resets the RSE.
    - If the integrity check passes, the SAM sets Trigger 4 (SAM configuration loaded successfully) after the External Sensors Ready signal is set.
  24. DMA P6 waits for Trigger 4. When triggered, DMA P6 may perform additional optional tasks, such as:
    - Setting trim values for analog sensors that require non-default thresholds.

- Clearing analog sensor status registers.
- Copying from OTP to the SAM an updated configuration that enables inputs from the trimmed sensors.

25. DMA P6 branches to the [DMA common end procedure in ROM](#).



Note

Only the first SAM configuration results in a successful trigger (T4). The second setup of the SAM completes silently on success; if a bad integrity value is detected, the SAM resets the RSE.

## 2.2.5 DMA common end procedure in ROM

The common end procedure is called by each DMA boot flow at its final stage. All flows require the same end operations. These DMA descriptors are stored in ROM to minimize OTP space usage.



Note

The DMA uses GPIO[4:0] signals on each channel for debug visibility before the RSE processor starts running. The RSE processor uses GPIO[3:0] to indicate boot progress. For more information, see [Boot stages signaling](#).

### 1. Set up TRAM encryption:

Note: Key slot 7 is not yet ready for use. Software completes this key slot programming after it runs

- The DMA copies eight random words (denoted A) from the TRNG hardware source to VM0. OTP retains provisioning records. Runtime entropy is sourced from TRNG hardware.
- The DMA copies A from VM0 to KMU key slot 7, which now belongs to the TRAM.
- The DMA copies A from VM0 to the TRAM key registers.
- The DMA copies eight new random words (denoted B) from the TRNG hardware source to the same address in VM0. This action wipes TRAM key A from VM0, which is a security requirement. It also generates a random value used to wipe the TRAM in later steps.
- The DMA sets the TRAM control register to enable TRAM encryption.

### 2. Clear TRAM content.

- The DMA copies B from VM0 to the entire TRAM.
- The DMA copies eight new random words (denoted C) from the TRNG hardware source to the same address in VM0. This operation clears the random value B from VM0 as a security requirement.

### 3. The DMA releases the DCU Force Disable overrides.

Note: At this point, the debugger can access the RSE processor D-AHB through JTAG. Debug access enablement depends on the lifecycle state and whether the chip is TCI or PCI. The

processor is still waiting, but the debugger can attempt to start its operation if debug is enabled.

4. The DMA releases CPUWAIT.
5. The DMA stops the current DMA channel's execution.

Note: At this point, if the debugger does not stop the processor, the processor boots from the ROM.

6. The processor sets RSE GPIO[3:0] = 0xD — SE ROM boot.

## 2.2.6 Boot stages signaling

It is useful to expose indications of the RSE pre-software boot progress to the SoC and to external users. Before the processor runs, the DMA provides indications through its GPIO signals. Afterwards, the processor (Boot ROM and provisioning software running in a sealed environment without JTAG access) provides indications through GPIO[3:0].

The DMA programs include five optional GPIO signals for each of the four channels. Signal encoding is defined in the DMA-350 programs. These signals are exposed on the PSI interface and indicate the progress of the DMA programs the engine executes.

The RSE processor (Boot ROM or provisioning programs) may use GPIO or GRETREG coding (visible at the PSI interface as RSEPSISTATUS signals). Make these GPIOs visible to the SoC and to users. They are triggered by software executing on the RSE processor. Use them to track the progress of secure provisioning and the overall pre-software boot flow.

**Table 2-2: Suggested coding values for GPIO boot signalling**

GPIO [3:0]	Status description
0x0	Cold-boot default (RSE)
0x1	Virgin chip detected – Boot ROM idle
0x2	Non-virgin chip detected – Boot ROM idle
0x3	RMA chip detected – Boot ROM idle
0x4	CM secure-provisioning started
0x5	CM secure-provisioning failed: authentication error
0x6	CM secure-provisioning failed: other error
0x7	CM secure-provisioning completed successfully
0x8	DM chip detected – Boot ROM idle
0x9	DM secure-provisioning started
0xA	DM secure-provisioning failed: authentication error
0xB	DM secure-provisioning failed: other error
0xC	DM secure-provisioning completed successfully
0xD	Security Engine (SE) ROM boot in progress

## 2.3 Lifecycle and provisioning workflow

Guidance explains RSE lifecycle transitions, provisioning, secure boot, and attestation. It provides a high-level overview and links to the individual lifecycle and provisioning processes.

- Platform lifecycle management
- Secure system boot
- Device attestation
- Assets provisioning and management

Together, these flows define the end-to-end manufacturing and trust-establishment workflow for the RSE and the platform.

### 2.3.1 Platform lifecycle management

The RSE is equipped with a Persistent State Interface (PSI), which provides a mechanism to convey information to the SoC.

The information provided by the PSI can differ between lifecycle states.

The following sections describe the components of this information.

#### 2.3.1.1 Lifecycle state

The lifecycle state is an output from the Lifecycle Manager (LCM) module. It reflects the lifecycle state of the RSE. Because the RSE is the SoC Root of Trust (RoT), the LCS also reflects the overall lifecycle state of the SoC. For more information, see [Lifecycle Manager](#).

#### 2.3.1.2 Lifecycle state valid

The `lcs_is_valid` signal indicates whether the lifecycle state signals contain a valid value. If the LCS detects an error that prevents determination of the lifecycle state, this signal is not set. In that case, the LCS also sets the `fatal_err` signal to indicate that lifecycle state reporting does not provide accurate information.

#### 2.3.1.3 DCU enable

These signals represent SoC debug and feature control. Their assignment is SoC-specific and can include enabling debug for various processing elements in the SoC and enabling or disabling licensed features.

## 2.3.2 Secure system boot

Secure boot establishes a chain of trust so that only authenticated software can run during platform initialization. Each stage verifies and transfers control to the next stage, preserving integrity and authenticity throughout the boot flow.

The chain starts from an immutable boot image in the RSE read-only memory (ROM). The RSE is the only element on the platform with boot ROM. As a result, the RSE boot ROM (Bootloader stage 1-1, BL1-1) serves as the platform's Root of Trust (RoT). The BL1-1 reference code ships as part of the RSE Bill of Materials (BOM).

After BL1-1, the boot continues through additional stages that extend the chain of trust. A representative layout of these components is described in [Trusted Firmware-M RSE Platform Documentation](#).

### 2.3.2.1 BL1-1 setting of the TRAM key slot in KMU

To protect assets stored in the TRAM before RSE reset, the DMA wipes the TRAM. It does this before the processor runs and before debug is enabled. This operation uses RNG bits from an RNG source (for example, OTP with RNG). It sets the TRAM key to a new random value. The DMA stores this key value in KMU key slot 7. It can be reused when the RSE exits memory retention, in which the TRAM encryption logic resets. Because the DMA cannot perform all required steps to set KMU key slot 7, BL1-1 completes these operations.

For details of the TRAM control interface, see [TRAM considerations](#).

1. The KMU key slot 7 is configured with the TRAM key register address. The KMU register KMUDKPA7 is written with the address of the TRKEYLBO register (0x5015\_D008).
2. The KMU key slot 7 control register is configured to align with the TRAM key registers, which are eight contiguous 32-bit registers. The DMA fills all eight key slot registers and sets the export address. The KMUKSC7 register is set to 0x01D6\_0100. For the field breakdown, see the KMUKSC field configuration table in [TRAM considerations](#).



After the VKS bit is set, the KMU checks the key slot and reports the result in the KSR bit (bit 25) of this register.

3. The software reads the KMUKSC7 register to verify that the DMA has correctly set the key slot. If bit [25] (Key Slot Ready) is not set, the operation has failed. When this occurs, an error can be reported through the PSI or by resetting the RSE. The error reason can be read from the KMUIS register of the KMU.
4. The keys stored in the KMU and the TRAM are verified to ensure they match. The DMA writes the same key value to both peripherals. To confirm this, a random word (for example, 0x12345678) is written to a TRAM location. The KMU key slot 7 is exported to the TRAM by writing bit 28 (EK) to KMUKSC7.

5. When the TRAM content does not match the expected value, the TRAM contents are wiped and its ECC becomes corrupted. The software will eventually fail when it attempts to read from the TRAM due to an ECC error alarm.
6. To verify this process without triggering an ECC alarm reset, the SAM is configured to disable its reset response during the check. The TRAM location is read and compared with the expected value (for example, 0x12345678). If the values do not match, an error is indicated using PSI signals or by resetting the RSE.

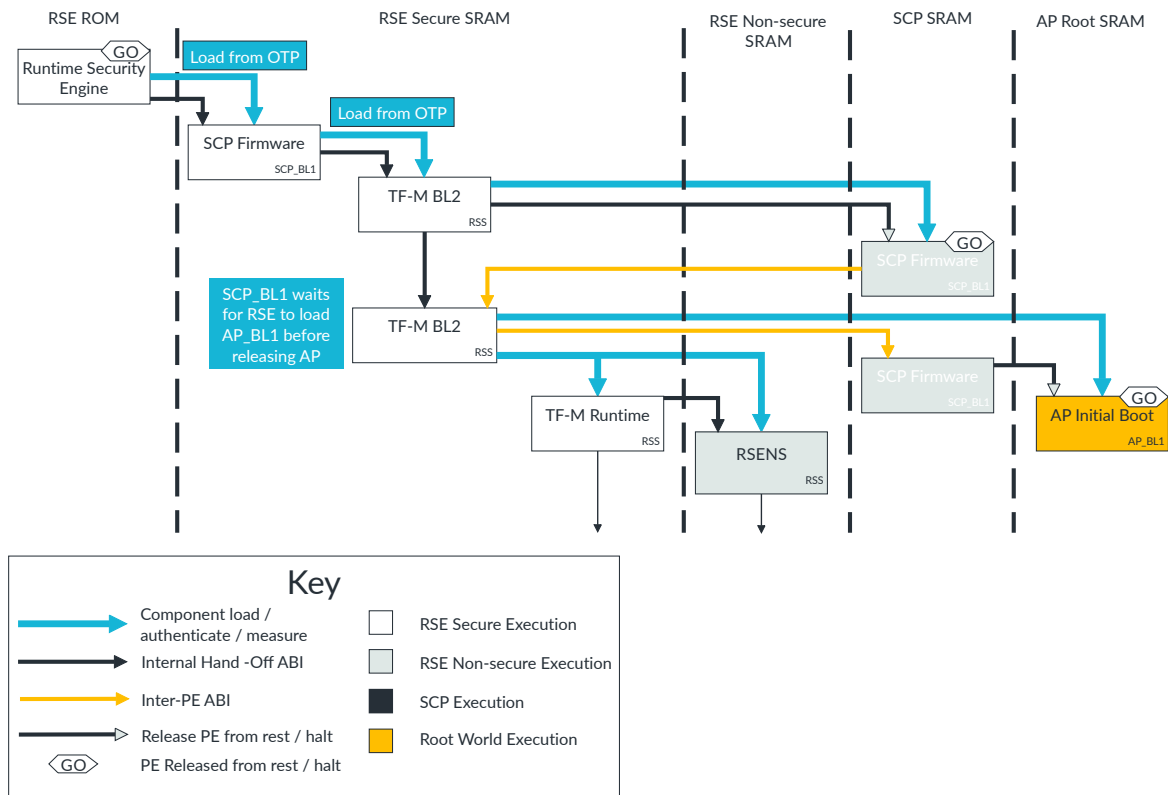
### 2.3.2.2 RSE processor boot flow

The RSE boot process consists of several boot stages. The following table describes the bootloaders and their functions for illustration purposes. Trusted Firmware-M (TF-M) uses the following bootloader scheme, but the scheme can vary depending on platform requirements.

**Table 2-3: Trusted Firmware-M (TF-M) bootloader scheme**

Name	Code residing in	Code running from	Main functionality	Authentication method
BL1-1	ROM	ROM	<p>Queries the lifecycle state and applies different flows for each lifecycle state.</p> <p>In the SE lifecycle, supports RSE secure debug at early ROM boot.</p> <p>Downloads the BL1-2 image from OTP to SRAM and verifies its digest against the OTP value.</p>	SHA-256 hash of the BL1-2 downloaded to SRAM compared to a digest value in OTP.
BL1-2	OTP	SRAM	<p>Configures clocks, interconnects, and flash access.</p> <p>Establishes trust with other RSE instances in the system (multi-RSE handshake).</p> <p>Downloads the BL2 image from flash to SRAM and authenticates it against the OTP value.</p>	NIST SP800-208 stateful hash (LMS) quantum-safe algorithm limited to 1024 uses.
BL2	Flash	SRAM	<p>Configures the system, performs authentication, and loads the SCP and AP images.</p> <p>Supports secure debug of the SCP and AP at their first instruction.</p>	ECDSA (with an option to move to post-quantum asymmetric crypto in the future).
Runtime	Flash	SRAM	Provides runtime security services such as DPE, measured boot, and attestation.	N/A

The following diagram shows the loading flow of the RSE reference bootloaders.

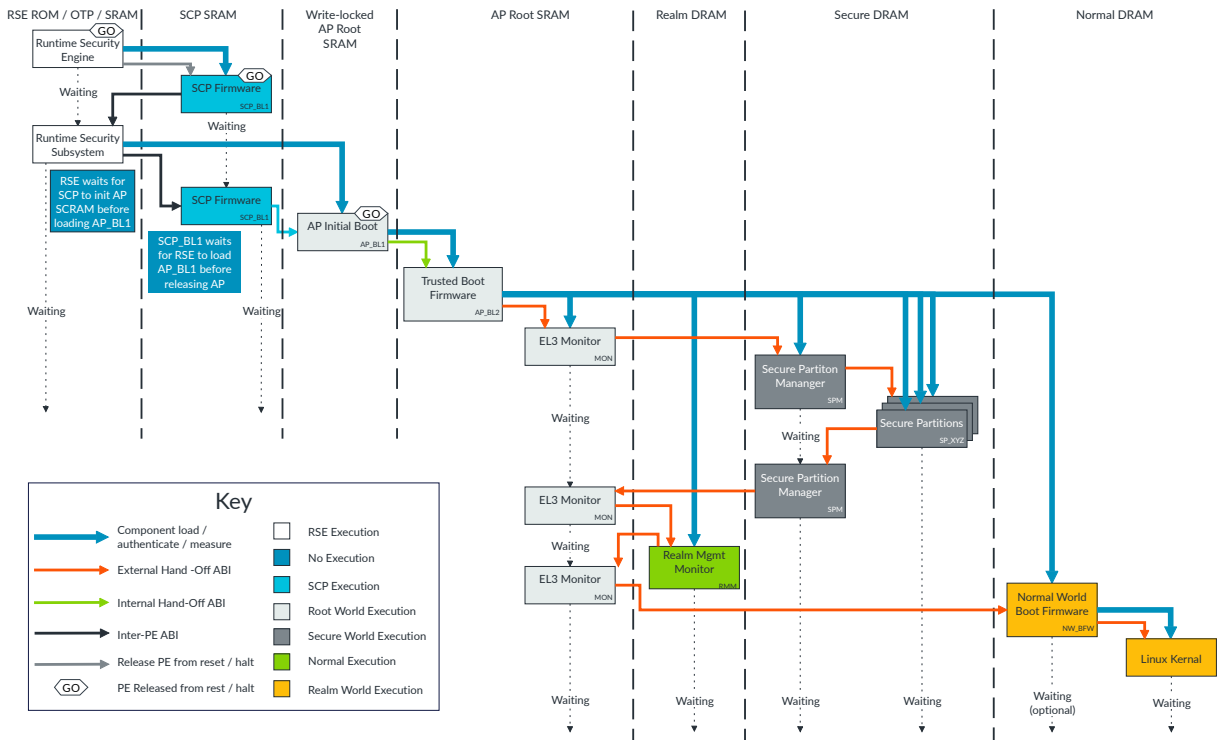
**Figure 2-1: Loading flow of RSE reference bootloaders**

### 2.3.2.3 System boot orchestration

The RSE acts as the system Root of Trust (RoT) and is responsible for starting and orchestrating the system boot.

1. The RSE is the first element in the system to boot and typically the only component with a Boot ROM.
2. When running, the RSE authenticates and runs initial images of other system components, such as the System Control Processor (SCP) and Application Processor (AP).
3. The RSE can optionally authenticate and start additional system images as required.
4. In some implementations, secure boot operates as a chain of trust, as shown in the diagram below.
5. All software components should have their measurements registered with the RSE for attestation services.
6. For improved security, the RSE should perform the measurements itself.



**Figure 2-2: SoC boot orchestration**

### 2.3.3 Multi-subsystem boot concept

Some systems contain more than one subsystem, where each subsystem includes an RSE. A subsystem might be a die, a chiplet, or a system on chip (SoC). These subsystems are designed to boot in coordination.

Each RSE in every subsystem starts its boot process at the same time. All RSE instances in the complete system perform a synchronization handshake to establish mutual trust between subsystems. This handshake confirms that all RSE elements possess a common Group Unique Key (GUK). Each RSE is also provisioned with its own unique Hardware Unique Key (HUK), which identifies it as an independent Root of Trust (RoT).

The outcome of the trust establishment process is the generation of a virtual HUK. This key represents the identity of the SoC, combining both compute and companion subsystems. The virtual HUK enables system-level security services such as attestation and sealing.

The following keys must be provisioned in the RSE OTP.

### 2.3.3.1 Hardware Unique Key

The HUK is a derivation key that defines the unique identity of the current subsystem. Each RSE in the SoC has a different HUK.

### 2.3.3.2 Group Unique Key

The GUK is a derivation key that defines the group identity and is shared by all RSE instances in the SoC. In some cases, the same GUK might be shared across a broader group, such as a silicon batch or product SKU.

### 2.3.3.3 RSE software behavior control using OTP-provisioned values

Every RSE in a multi-chiplet system runs the same software stack (BL1\_1, BL1\_2, BL2, and Runtime). The difference in boot paths between subsystems is determined by values provisioned to the OTP of each RSE instance.

The following sections describe specific OTP fields that control RSE behavior in a multi-RSE system.

#### 2.3.3.3.1 RSE identifier

The RSE identifier is a unique enumerator assigned to each RSE in the system. It is used to define the order of participation in the cross-subsystem handshake protocol. The RSE with identifier 0 initiates the cross-subsystem handshake protocol.

#### 2.3.3.3.2 RSE routing table

The routing table defines the communication interfaces between RSE instances in the system. Each RSE identifier in the table is associated with sideband interfaces such as MHU channel numbers and I3C addresses. These mechanisms are used for outbound communication from the current RSE. The table also defines whether the sideband link must be secured. It specifies which security properties are required, such as integrity, confidentiality, and rollback protection.

For example, in monolithic systems, the link must be protected against side-channel attacks (confidentiality) and fault injections (integrity). In systems where the link is external to the PCB, such as multi-socket systems, physical probing and man-in-the-middle attacks must be considered. Replay protection must be provided.

#### 2.3.3.3.3 RSE images information

This data structure contains information about the non-volatile memory used to store software images (such as QSPI or UFS flash). It allows the RSE to use the appropriate software driver for flash operations. The data structure includes a list of images and the expected storage addresses.

It also specifies the media type and whether images are accessible directly or through a sideband interface.

The complete data structure is stored in flash rather than in the RSE OTP to save OTP space and provide flexibility. Only the first entry in the table is provisioned to the OTP of all RSE instances in the system. This entry refers to BL2, the first bootloader located in flash.

#### 2.3.3.3.4 RSE NVM access

The NVM access flag is a binary value indicating whether the subsystem that contains the current RSE includes non-volatile memory (typically flash). This memory stores software images that the RSE authenticates and loads during boot.

The RSE with this flag set accesses the non-volatile memory directly. It loads the images specified in the RSE boot bitfield (for example, RSE\_BL2, SCP firmware, and AP BL1). It is also responsible for sending these images to other subsystems according to the NVM distribution table.

The RSE with this flag cleared requests the RSE\_BL2 and SCP firmware via a sideband interface from another RSE defined in its NVM distribution table. During later boot stages, when the inter-die link is initialized, it loads its software images from the non-volatile memory on another subsystem. This occurs through the main interconnect or inter-die link. Before accessing the memory, it must receive a message from the RSE that has NVM access. The message confirms that the memory has been initialized and configured. Access to the appropriate memory device follows the references defined in the images information table.

#### 2.3.3.3.5 RSE image distribution table

The image distribution table defines how software images are shared between RSE instances.

For an RSE with NVM access, the list contains details of RSE instances expected to request images over a side-channel. The list might be empty if all RSEs in the system have their own non-volatile memory.

For an RSE without NVM access, the table specifies where it can request its images from. It also includes a flag indicating whether full cryptographic image verification is done by the receiver RSE or by the sender. If the sender performs the verification, the receiver only performs a non-cryptographic integrity check to ensure the image has not been modified during transfer.

#### 2.3.3.3.6 RSE boot bitfield

The boot bitfield defines which components each RSE boots. For example, in a typical compute subsystem, the RSE sets all bits for its components, SCP firmware, and AP BL1. In a companion subsystem, which does not contain an AP cluster, the RSE sets only the bits for its own components and the SCP firmware.

#### 2.3.3.3.7 RSE attestation owner

The attestation owner flag is a binary value. It indicates whether the current RSE is responsible for creating attestation certificates and providing them to the verifier. An RSE with this flag set collects attestation data from all RSE instances listed in its routing table.

#### 2.3.3.3.8 System memory map

To orchestrate the system boot flow, the RSE must access various system address locations. Examples include SCP registers, SCP SRAM, and AP SRAM. These addresses may vary between SoCs or subsystems. Each RSE is provisioned with the system address list it needs to access.

### 2.3.3.4 Cross-subsystem RSE handshake

During the RSE ROM bootloader (BL1-1) stage, all RSE instances in the subsystems perform a handshake. This handshake verifies that all subsystems are in the same lifecycle state and that they possess the same Group Unique Key (GUK).

After this verification, all subsystems establish a shared virtual identity known as the Virtual HUK (vHUK). The vHUK combines the contributions of all subsystems and serves as the unique identity of the entire platform. It is also used in security services such as attestation. The RSE instance designated as the attestation owner collects measurements from all other RSE instances in the system.

Regardless of the system topology, RSE instances operate in a flat structure with no hierarchy. Each RSE uses its unique identifier to communicate in an RSE-to-RSE manner. The RSE instance with the lowest identifier (0) initiates the process by communicating with the next lowest RSE listed in its routing table.

The handshake protocol can vary depending on the security requirements for the link connecting the RSE instances. Information about link security requirements is defined in the routing table of each RSE instance.

## 2.3.4 Device attestation

The RSE provides attestation for the SoC hardware and software. Attestation creates cryptographically signed reports containing claims about the hardware state of the SoC and details of the boot flow and software components. These signed claims enable a verifier to confirm the trustworthiness of the attestation information.

As the platform Root of Trust (RoT), the RSE is the boot anchor for the entire SoC. It contains the only Boot ROM in the platform and is implicitly trusted. It authenticates and runs initial boot components, including its own bootloaders and the platform's initial boot code for other processing elements. For subsequent components, the RSE authenticates and runs them. It measures them cryptographically or receives measurements from other boot components as part of the chain of trust. Thus, the RSE maintains measurements of all boot components across the platform.

Measurements and metadata of software boot components collected during boot serve as claims in attestation reports. Claims include the platform lifecycle state and debug or feature enablement states controlled by the RSE. Confidential keys provisioned in the RSE OTP generate cryptographic keys for signing these reports.

The RSE supports attestation in multiple formats based on system requirements and integration needs.

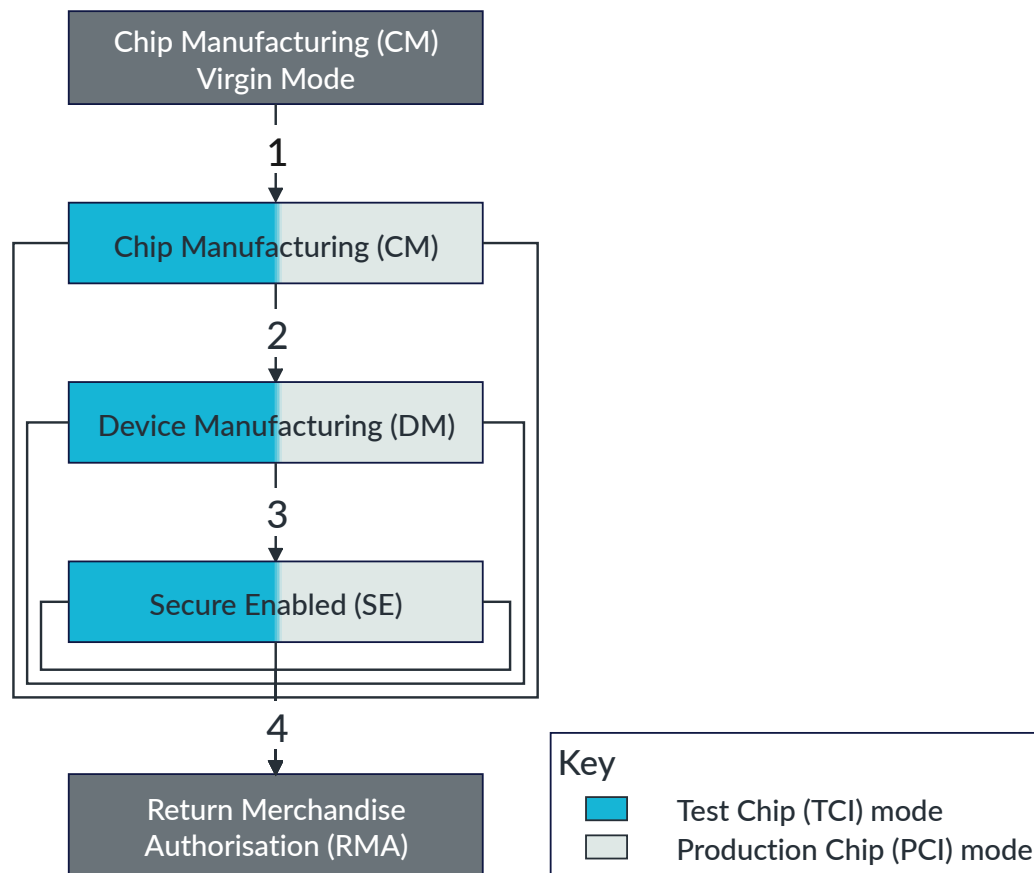
### 2.3.5 Assets provisioning and management

To function as the system Root of Trust (RoT), the RSE requires provisioning. Provisioning is the process of loading and storing secrets, configuration parameters, the DMA boot expansion program, and processor execution code. These are provisioned into the RSE OTP and, optionally, into a flash device.

The provisioning process is enforced by lifecycle management. During this process, the RSE transitions through several lifecycle states, each of which applies its own security policy.

Guidance describes transitions between RSE lifecycle states and the provisioning process.

The following diagram shows the lifecycle state transition flow.

**Figure 2-3: RSE lifecycle transitions**

The flow operates in either Test Chip (TCI) mode or Production Chip (PCI) mode, but not a combination of both.

The transitions shown in the diagram between each lifecycle state occur as follows:

1. The RSE programs the TP mode (TCI or PCI) at power-on reset.
2. During this “Secure Provisioning” phase, the processor resets and writes CM dummy secrets to the OTP at power-on reset.
3. During this “Secure Provisioning” phase, the processor resets and writes DM secrets to the OTP at power-on reset.
4. At power-on reset, the system configures the RMA state. This transition can occur from the CM, DM, or SE lifecycle states to move to the RMA lifecycle state.

The treatment of the KRTL by the Lifecycle Manager (LCM) during secure provisioning depends on the lifecycle state and TP mode. The Key Management Unit (KMU) is referenced in the table below.

Table: Lifecycle key treatment during provisioning

LCS	Chip mode	LCM treatment of keys	Notes
CM	PCI	Exports KRTL to KMU	KRTL is used to derive keys for decrypting CM assets that are written to the OTP.
CM	TCI	No export of keys	A dummy KRTL is used to derive keys for decrypting dummy CM assets written to the OTP.
DM	PCI	Exports KRTL and CM keys to KMU	KRTL and CM keys are used to derive keys for decrypting CM or DM assets written to the OTP.
DM	TCI	KRTL not exported  Dummy CM keys exported	Dummy KRTL and CM keys are used to derive keys for decrypting dummy CM or dummy DM assets written to the OTP.

### 2.3.5.1 Running DFT tests on the RSE and its SoC

A blank chip starts in the Chip Manufacturing (CM) lifecycle state in Virgin TP mode. This means nothing is programmed into the chip's OTP.

At this stage, the user can run various Design for Test (DFT) operations on the RSE and SoC, including OTP tests, LBIST, and MBIST. The DFT process writes die-tracking data and computed trim values for the SoC into the OTP. Provisioning begins only after all tests complete and trim values are successfully written to the OTP.



The OTP region written by the DFT user does not affect the RSE's Chip Manufacturing (CM) lifecycle state or Virgin TP mode. These states are controlled by the Lifecycle Manager (LCM), which ignores the DFT OTP region.



DFT for the RSE runs while the lifecycle state is CM and Virgin. During secure provisioning in CM, and in the DM and SE lifecycle states, DFT is disabled. DFT is enabled again when the lifecycle state transitions to RMA.

### 2.3.5.2 Locking down the RSE

During blank chip, wafer, or die testing, including DFT tests, some SoCs or RSE instances may be identified as defective. These chips are usually binned and not provisioned. Before binning, Arm recommends locking down the RSE to make it inoperable.

The lock-down operation is only possible when the Lifecycle Manager (LCM) is in the Chip Manufacturing (CM) lifecycle state and Virgin TP mode. For the detailed flow, see “LCM Lockdown” in [Arm® Lifecycle Manager Specification](#).

**Note**

When the RSE is locked down, its `dcu_en` signals are set to disabled, which is the same configuration as a security-enabled production chip. As a result, JTAG debugging is disabled for both the RSE and the rest of the SoC where the `dcu_en` signals are connected.

If the chip will not be binned, the chip manufacturer should consider transitioning to the RMA state to enable debug operations.

### 2.3.5.3 First provisioning step: chip manufacturing

After DFT tests complete, the next provisioning step is to program the TP (test/production) mode in OTP. TP mode defines whether the chip operates as a Test Chip (TCI) or a Production Chip (PCI). This distinction enables testability for TCI and security for PCI.

Test chips are intended for debugging and bring-up. They mimic the lifecycle of a production chip with the following differences.

- In TCI, the RTL key (KRTL) is not accessible. KRTL is a shared secret between the chip manufacturer's secure room and the RSE. It derives keys used to authenticate and decrypt provisioning blobs that carry real manufacturer assets. When KRTL use is required in TCI, a zero key is used instead.
- In TCI, a separate, more permissive debug policy is defined in RTL. Typically, all debug controls are open in every lifecycle state. Provisioning blobs for TCI must never contain real secrets; they must contain dummy assets for test only.

#### 2.3.5.3.1 Chip manufacturing provisioning — programming guidance

Programming TP mode is performed by writing the selected value to the TP Mode Config word in OTP. The write occurs through the LCM. For the detailed flow, see "Secure provisioning programmer flow" in [Arm® Lifecycle Manager Specification](#). These operations can occur via JTAG directly. They can also occur by a program that the JTAG loads into RSE SRAM and that runs on the processor.

If a value other than TCI or PCI is written, the LCM enters an invalid mode, effectively bricking the chip.

TP mode becomes valid after a power-on reset. The reset can be initiated by JTAG, by software setting `SWRESETREQ` in the `SWRESET` register, or by power-cycling the SoC.

After TP mode configuration, the RSE remains in the CM lifecycle state but is no longer virgin. At this point, the RSE is ready to begin chip manufacturer secure provisioning.

To start secure provisioning, the user loads an encrypted and authenticated blob into a predefined SRAM location in the RSE over JTAG. The blob contains the provisioning software and assets. The sequence that follows enforces secure provisioning policy while JTAG is disabled:



1. The value DCU\_SP\_DISABLE\_MASK\_VAL (0xAAAA\_AAAA) is written to the LCM DCU\_ENO register. Otherwise, the Debug Signals Security Checker raises an alarm in the next step.
2. A special value is written to the LCM SP\_ENABLE register to enter secure provisioning (SP). Because the DCU\_ENO write disconnects JTAG, this SP\_ENABLE write must be performed by a program running from RSE SRAM.
3. The LCM asserts a warm reset of the RSE. The reset affects most of the RSE except the LCM.
4. The secure provisioning security policy takes effect. In PCI, all debug interfaces are disabled.
5. In PCI, the LCM exports KRTL to its key slot in the KMU. BL1-1 runs again under SP policy.



In TCI, the LCM does not export KRTL to the KMU and the slot cannot be used. In this case, the ROM bootloader should export a dummy all-zero key to the KMU slot. It should then decrypt blobs by using that dummy key. This enables a provisioning flow similar to PCI, but with debug enabled.

During SP, the ROM bootloader (BL1-1) expects an authenticated and encrypted provisioning blob at a predefined SRAM address. The blob is installed via the JTAG interface before JTAG is disabled. The blob contains the provisioning executable and secure data (CM keys and assets). BL1-1 authenticates and decrypts the blob by using a key derived from KRTL that is now available in the KMU. It stores the decrypted content in SRAM and starts the provisioning program.

The provisioning program typically decrypts the manufacturer's assets into the TRAM by using a key derived from KRTL. It then randomizes the hardware unique key (HUK) for this RSE instance by using the on-board TRNG.

The provisioning software then writes the following keys to OTP.

- HUK
- GUK
- KP\_CM
- KCE\_CM
- ROTPK

CM assets can also include configuration data and other keys. See “Secure provisioning programmer flow” in [Arm® Lifecycle Manager Specification](#) for the exact sequence.



For security, the provisioning program should randomize the order of writing HUK, GUK, KP\_CM, and KCE\_CM. It should also randomize the order of writing each of the eight words for each key.



If you store proprietary secrets in general-purpose OTP space, and they are not part of the standard CM assets, take extra care. Encrypt them by using a key derived from the HUK. This prevents exposure in DM when debug is enabled and reduces the risk of disclosure by electron microscopy. Include an integrity code to protect these secrets against tampering.

After writing the CM keys, the provisioning software writes the DMA program descriptors and the next bootloader (BL1-2) to OTP.

If available at the CM facility, the provisioning software can also include BL2 and write it to flash. Otherwise, BL2 is written during the DM provisioning flow.

When OTP programming completes, a power-on reset is required. The reset can be initiated by setting SWRESETREQ in the SWRESET register or by power-cycling the SoC. The reset transitions the lifecycle to DM. The provisioning software can signal completion or failure to the user by using GPIO or PSI signals. At this stage, JTAG cannot reset the RSE because JTAG is disabled by the SP flow.

To mitigate unauthorized CM provisioning blobs, the ROM DMA flow should set the INITSVTOR0 register to its reset value and set INITSVTOR0LOCK to lock it.

#### 2.3.5.4 Second provisioning step: device manufacturing

In the Device Manufacturing (DM) lifecycle state, debug interfaces are disabled. This follows the hardware configuration parameters defined in the Lifecycle Manager (LCM). In this state, the RSE uses the SDC-600 to communicate with the user to obtain the DM provisioning blob and store it in SRAM. Once the entire blob is in SRAM, the remaining steps proceed as they do during chip manufacturing (CM) provisioning.

By the end of the process, the following DM assets are provisioned.

- KP\_DM
- KCE\_DM

DM provisioning can also write to other unpopulated OTP offsets and, if required, to flash memory. As in CM provisioning, proprietary secrets must be protected for integrity and confidentiality.

#### 2.3.5.5 Device return: transition to RMA lifecycle state

The RSE can transition to the Return Merchandise Authorization (RMA) lifecycle state from any other lifecycle state. This transition is performed by writing the CM and DM RMA flags in the OTP and then power-cycling the RSE.

Transitioning from the Security-Enabled (SE) lifecycle state can be exploited by attackers or occur by mistake. Therefore, this transition is protected by certificates verified by the RSE software. Typically, two certificates are required. Each is generated by a different party.

- One certificate is produced by the chip or SoC vendor (CM).
- One certificate is produced by the device manufacturer (DM).

Successful verification of each certificate by the RSE software sets the corresponding flag in the OTP.

- CM RMA flag for successful CM certificate verification.
- DM RMA flag for successful DM certificate verification.

The design can also support a single certificate that sets both CM and DM RMA flags upon successful verification.

A typical sequence for the transition to RMA is as follows:

1. The device is returned to the DM, who creates and applies the DM RMA certificate.
2. The device is then sent to the CM, who creates and applies the CM RMA certificate.
3. The RSE remains in the SE lifecycle state until both CM and DM RMA flags are set in the OTP.

To prevent unauthorized transitions, the DM\_RMA\_LOCK register can be used. If this feature is implemented, one of the RSE bootloaders sets this volatile register at every boot. This prevents later-stage software (such as the RSE runtime) from modifying the DM RMA flag in the OTP.

When the RSE receives an RMA transition certificate (which may include both CM and DM requests), it authenticates the certificate.

If the DM\_RMA\_LOCK register is set, the RSE stores the certificate or its parameters in a known SRAM location and reboots.

On the next boot, before locking the DM\_RMA\_LOCK register, the bootloader locates and authenticates the stored certificate.

If authentication succeeds, the RSE sets the DM RMA OTP flag (and CM flag if applicable) and performs a cold reset. This allows the Lifecycle Manager (LCM) to detect the transition and update the lifecycle state.

The RSE transitions to RMA only when both CM and DM RMA flags are set in the OTP.

After entering the RMA state, the bootloader overwrites the confidential keys and the ROTPK in the OTP. It writes ones to addresses 0x00–0xDF using word accesses.

The RSE can also transition to the RMA state from the CM or DM lifecycle states. This scenario is used when a defect is detected in the chip during earlier stages. In such cases, the RMA transition wipes all secrets provisioned to the chip, and no RMA certificates are required.

For detailed flows, see “Transition from any state into RMA (Phase 1 – CM)” and “Transition from any state into RMA (Phase 2 – DM)” in [Arm® Lifecycle Manager Specification](#).

**Note**

In the RMA lifecycle state, the RSE can be debugged through JTAG.

**Note**

The current lifecycle state can be read from the LCS\_VALUE register of the LCM at any time.

---

## 2.4 Software access to system memory and peripherals

RSE software must control SoC peripherals and memories as part of its operation. RSE software accesses these resources through the ATU. The ATU translates a 32-bit RSE logical address to a 52-bit SoC physical address.

Accessing a peripheral by the CPU requires that its address be defined as DEVICE. Otherwise, the transaction may be cached and cause unexpected results. In the current RSE, the address range is routed to the ATU and then to the SoC via the AXI Interconnect Interface. It uses memory access attributes by default. This range is `ATU_DATA: 0x6000_0000..0x6FFF_FFFF` (non-secure) and `0x7000_0000..0x7FFF_FFFF` (secure). That default means accesses to this region may be cached. To work around this, when software accesses SoC peripherals, it must set an MPU region as DEVICE for the destination device address range.

## 3. Hardware modules operational guidance

Operational guidance supports firmware developers configuring and using RSE hardware modules. Guidance covers secure interactions, lifecycle handling, and system integrity.

This topic covers the following areas:

- Lifecycle management through the Lifecycle Manager (LCM), which controls SoC lifecycle transitions.
- Secure pre-processor boot operations using the DMA-350.
- Memory integrity verification using the Integrity Checker.
- Address translation for secure SoC memory access through the Address Translation Unit (ATU).
- Secure key handling using the Key Management Unit (KMU)
- Management of SRAMs (ITCM, DTCM, VM0/VM1)
- TRAM encryption.

### 3.1 Lifecycle Manager

The Lifecycle Manager (LCM) plays a key role in the RSE boot process. It is one of the two components, along with the DMA, that operate during the earliest stages of RSE boot.

The LCM is responsible for controlling and monitoring lifecycle transitions across the SoC. It enforces access permissions, security policies, and debug interface availability based on the current lifecycle state. During early boot, the LCM also interacts with the DMA to initialize hardware security configurations before the processor starts executing code.

LCM functions include:

- Managing lifecycle transitions such as chip manufacturing (CM), device manufacturing (DM), security-enabled (SE), and return merchandise authorization (RMA).
- Handling security configurations for each state.
- Controlling debug accessibility through DCU enable and disable settings.
- Providing lifecycle state information to other components, such as the RSE and SoC subsystems.

Through its coordination with the DMA and other hardware modules, the LCM ensures that the SoC boots securely. It also manages transitions between lifecycle states in a controlled manner.

### 3.1.1 SoC lifecycle management

The Lifecycle Manager (LCM) within the RSE manages security policies for each lifecycle state of the RSE and the SoC.

**Table 3-1: Lifecycle states for the RSE**

Lifecycle state	Abbreviation	Description	Typical usage	Access and debug capabilities
Chip manufacturing	CM	Initial state after fabrication. OTP is blank or partially programmed.	Silicon provisioning and test	Full debug access (JTAG); DFT access enabled while TP mode is not programmed
Device manufacturing	DM	CM provisioning completed. Device was then cold reset. OEM assets can be provisioned.	OEM provisioning environment	Debug enabled in TCI (not PCI); RSE DFT disabled
Secure enable	SE	Device is fully provisioned and secure. All assets are in place.	Field and production use	Debug enabled in TCI (not PCI); secure mode only
Return merchandise authorization	RMA	Final state allowing limited access for device return and analysis.	Returns and fault analysis	Full debug access (JTAG); DFT access enabled

The LCM maintains 128 dcu\_en output signals. These signals enable or disable debug and feature controls for the RSE, the processing elements (PEs), or other SoC components. Each signal's behavior depends on the current lifecycle state. The integrator sets default signal values for each state using configuration parameters.

#### 3.1.1.1 TP mode configuration

If the LCM is set to TP=PCI (production chip indicator):

- In the CM lifecycle state, all debug and DFT controls are enabled to allow production testing and provisioning. Once secure provisioning starts, all debug and DFT controls are disabled.
- In the DM lifecycle state, debug testing controls are disabled to protect provisioned CM assets. Debug remains disabled once secure provisioning starts.
- In the SE lifecycle state, all debug and DFT controls are disabled.
- In the RMA lifecycle state, all debug and DFT controls are enabled.

If the LCM is set to TP=TCI (test chip indicator):

- In every lifecycle state, all RSE debug and DFT controls are enabled.

### 3.1.1.2 DCU\_EN signal assignments

The table below describes how the LCM assigns dcu\_en signals for internal RSE functions and integrator-defined usage.

**Table 3-2: DCU\_EN signal assignments**

Signal range	Function and usage
0–25	Used by the RSE to control its own debug authenticated enable signals. Signals are paired (positive and negative) for security.
26–31	Reserved for future RSE usage.
32–127	Available for integrator use through the PSI interface. Can be connected to SoC PE authenticated debug signals or feature controls.

For security reasons, Arm recommends using paired signals with inverse logic for each controlled feature. A value of 0b01 enables a feature or debug capability, and 0b10 disables it. Any other value (0b00 or 0b11) disables the feature or debug capability and triggers an alarm signal to the RSE's Security Alarm Manager (SAM). The reverse-bit logic acts as an integrity check for fault injection protection. The RSE checks the integrity of the first 26 bits (13 pairs) and raises an alarm if inconsistencies are detected. Arm recommends verifying the integrity of the remaining dcu\_en signals close to their target components.

### 3.1.2 LCM and RSE boot process

During the RSE boot process, the Lifecycle Manager (LCM) initializes the DCU (debug control unit) security policy. Initialization is based on the current lifecycle state and TP mode. The LCM extracts lifecycle state (LCS) and TP mode parameters from the OTP before the processor is released from reset.

The first stage of the process is the LCM reading the TP mode from the OTP.

**Table 3-3: LCM TP modes**

TP mode	Description
Virgin	TP mode not yet provisioned
Test Chip (TCI)	TCI indication is set
Production Chip (PCI)	PCI indication is set
Invalid	TP mode has an illegal or invalid value

If an invalid TP mode is detected, the LCM enters a fatal error state and prevents the RSE from booting. This condition can result from a faulty OTP or a security attack. Setting an illegal TP mode value can also be used intentionally by the manufacturer to disable or “brick” the RSE. This can be done when a defective SoC or RSE is detected during DFT testing.

When a fatal error occurs, the RSE remains stuck because the lifecycle state is not initialized and booting cannot continue. The fatal error condition is also reported to the Security Alarm Manager (SAM) as an alarm input. A typical response to this alarm is a system reset. If the fault is

intermittent, such as one caused by an external attack, the RSE may boot normally on the next reset cycle.

If the TP mode is virgin or valid, the LCM computes its lifecycle state by reading multiple entries in the OTP. If an OTP error or an attempted attack is detected, the LCM enters fatal error mode, which prevents the RSE from booting. If both the lifecycle state and TP mode are valid, the LCM signals that the LCS is valid. From this point, registers and signals that indicate the lifecycle state and TP mode can be used.

The LCM reads the OTP fields required to extract the lifecycle state twice, with a random delay between reads. This delay protects against timing and fault injection attacks by making it difficult to synchronize a double fault attack. To generate this random delay, the LCM requires an entropy seed initialized by subsystem hardware. An appropriate source for this entropy is the TRNG hardware integrated in the RSE or the TRNG provided by the integration layer. OTP should only store provisioning records and seed metadata, not runtime entropy. The LCM does not operate until the random seed is initialized.

## 3.2 DMA-350

The DMA-350 module operates during the early boot stages of the RSE as a trusted hardware state machine. It ensures that the RSE executes in the most secure state before the processor is released from reset. The DMA is a programmable hardware engine that runs a program (a linked list of descriptors) starting from a boot address in ROM.

To reduce dependence on ROM content, the DMA performs the most critical initialization tasks directly from ROM. This also allows configuration using parameters unavailable when the OTP is blank. After OTP provisioning is complete and the lifecycle state transitions to Security Enabled (SE), the DMA continues with its expansion program stored in the OTP.

To protect against OTP read attacks, the DMA copies its expansion program from the OTP to VM0 memory (which is ECC protected). It then performs an integrity check on the copied program and executes the verified code from VM0. Because the expansion program can be updated with future OTP versions, this mechanism provides flexibility to fix bugs or update parameters without modifying the ROM.

### 3.2.1 DMA operations during early boot

When the lifecycle state is SE and debug ports are automatically closed, the DMA performs the following sequence.

1. Copies the DMA expansion program from the OTP to VM0, verifies its integrity, and starts executing the verified program from VM0.
2. Executes the following operations as part of the expansion program:



- Copies the Security Alarm Manager (SAM) configuration for pre-sensor settings into the SAM. Values from the OTP override SAM reset defaults and typically enable all digital alarm signals and their actions.
- Sets trim values for the RSE sensors that require trimming before the processor is released.
- Clears RSE sensor status registers to prevent false alarms triggered before trimming.
- Copies the SAM configuration for post-sensor settings into the SAM. This configuration typically enables SAM inputs from the trimmed sensors.

### 3.2.2 Common end procedure

The following end procedure applies to all DMA boot flows.

1. Enables Trusted RAM (TRAM) encryption, which encrypts the entire DTCM.
2. Wipes the TRAM (DTCM) by writing constant data. This removes any leftover secrets from memory and generates valid ECC values for the DTCM, preparing it for partial writes.
3. Opens debug ports.
4. Releases the processor from reset.

## 3.3 Integrity checker

The Integrity Checker (IC) peripheral validates the integrity of data in memory by computing and comparing hash values. It operates during both the boot process and runtime to ensure that only verified data and code are executed by the system.

During the RSE boot process, the DMA configures and triggers the Integrity Checker. It verifies critical assets and code segments as they are loaded into SRAM. This verification ensures that only untampered content executes, forming a critical part of the secure boot flow.

At runtime, software uses the Integrity Checker to verify the integrity of data structures or code that reside in SRAM or other accessible regions. This ongoing verification maintains the integrity of trusted assets throughout the system's operational lifetime and helps detect any tampering or memory corruption.

### 3.3.1 Integrity checker usage by DMA during boot

During early boot, the DMA uses the integrity checker to verify the DMA programs it copies from OTP to SRAM. This process ensures that only authentic and unaltered programs run during boot.

When the integrity checker completes verification, it signals the DMA with a trigger if the check passes. If verification fails, it raises an alarm.

To use the integrity checker in this scenario, the DMA configures the registers as follows:

1. The ICAE register remains at its reset value (0x7E) to enable alarms for all detected error types. The DMA does not need to write to this register.
2. The ICIE register remains at its reset value (0x00) to disable all interrupts, as the DMA cannot service interrupts.
3. The ICDA register points to the SRAM buffer containing the data to be verified. Align this address on an 8-byte boundary.
4. The ICDL register specifies the size of the ICDA SRAM buffer. Align this size on an 8-byte boundary.
5. The ICEVA register points to the SRAM buffer holding the expected integrity check value. Align this address on an 8-byte boundary. The buffer size depends on the algorithm selected in the ICC register. The DMA typically places the expected value immediately after the data buffer, so ICEVA usually points directly after the ICDA buffer.
6. The ICC register starts the integrity checker operation. For DMA boot, use the field values in the following table.

**Table 3-4: ICC field values for DMA boot**

Field	Value	Description
Start	1	Starts the integrity check operation
Algorithm	0	Zero Count, typical for OTP content
OpMode	0	Check mode
MatchTriggerDisable	0	Enables trigger signal on match
EnCompValOut	0	Not used in check mode
ARPROT	0b001	Privileged access, secure, data access
AWPROT	0b000	Not used in check mode
ARCACHE	0b0000	Device non-bufferable
AWCACHE	0b0000	Not used in check mode

### 3.3.2 Integrity checker usage by software in runtime

Software can use the Integrity Checker (IC) to calculate and verify the integrity of a memory region or data structure. This capability applies to SRAM, DRAM, or flash memory. It serves both security and efficiency purposes.

When software uses the IC to verify a buffer against an expected value, the following general flow applies.

1. Set the ICAE register to 0x00 to disable alarms. The DMA leaves this register set to a nonzero value, so software must clear it at least once.
2. Set the ICIE register to define which completion interrupts are required. A value of 0xFF enables completion and all error interrupts. Because the reset value of this register is 0x00, software must set it as needed.
3. Set the ICDA register to point to the SRAM buffer that holds the data to be verified. This address must be aligned on an 8-byte boundary.

4. Set the ICDL register to define the size of the ICDA SRAM buffer. The size must also be aligned on an 8-byte boundary.
5. Set the ICEVA register to point to the SRAM buffer that holds the expected integrity check value. This address must be aligned on an 8-byte boundary. The size of this buffer is defined by the algorithm selected in the ICC register.
6. Set the ICC register to start the Integrity Checker operation. For software integrity checks with interrupts enabled, use the field values shown in the table below.

**Table 3-5: ICC field values for software integrity check use case with interrupts**

Field	Value	Description
Start	1	Starts the integrity check operation
Algorithm	0	Zero Count – typical for OTP content
OpMode	0	Check Mode
MatchTriggerDisable	1	Disables trigger output to the DMA
EnCompValOut	0	Not used in Check Mode
ARPROT	0bXXX	As required for read access to ICDA and ICEVA buffers
AWPROT	0b000	Not used in Check Mode
ARCACHE	0bXXXX	As required for read access to ICDA and ICEVA buffers
AWCACHE	0b0000	Not used in Check Mode

The current IC implementation supports only the zero count algorithm. This is indicated in the ICBC register.

If interrupts are enabled, software handles the interrupt by reading the ICIS register. Software then confirms that the Done bit is set. If interrupts are disabled, the software polls the ICIS register. It continues polling until the Done bit is set.

After completion, software checks whether any error bits are set. If no error bits are set, the integrity check verification has succeeded. Once complete, the software must clear the Done bit and any handled error bits by writing to the ICIC register.

### 3.3.2.1 Integrity calculation using the integrity checker

The Integrity Checker (IC) can also calculate the integrity check value of a buffer and output the computed value to memory. When using the IC in this mode, the following configuration applies:

1. Set the ICCVA register to point to the SRAM buffer that will hold the computed integrity check value. The address must be aligned on an 8-byte boundary. The size of this buffer is determined by the selected integrity calculation algorithm but must also be aligned on 8 bytes.
2. In the ICC control register, configure the following:
  - Set OpMode to 1 to indicate that the IC should compute the integrity value.
  - Set EnCompValOut to 1 to instruct the IC to write the computed value to the memory pointed to by ICCVA.

- Configure AWPROT and AWCACHE as required for write access to the memory referenced by ICCVA.

The IC always performs an integrity check, regardless of whether the computed value is written to memory. At the end of the calculation, the IC places the result in the ICCVAL register. If EnCompValOut is set, the IC also writes the computed integrity value to the memory location defined by ICCVA.

## 3.4 Address Translation Unit

The Address Translation Unit (ATU) maps SoC physical memory locations to addresses in the RSE memory space. It allows flexible memory allocation while providing a hardware-level firewall between the RSE and the SoC.

For detailed register descriptions and programming information, see [Arm® Address Translation Unit Specification](#).

Because the RSE operates as an isolated execution environment, its memory is not visible to the rest of the SoC. General SoC memory can be accessed from the RSE only after the ATU has been configured to map a specific SoC address range. This mapping prevents unintended data leakage from the RSE to the SoC, which could result from a software bug, vulnerability exploitation, or physical attack.

To maintain the ATU as an effective security boundary, the following guidelines should be followed:

- Operate ATU regions on demand. Map a region only when SoC memory access is required and unmap it immediately after use.
- Configure each ATU region to the smallest size possible, as allowed by the ATU's granularity (4 KB, 8 KB, or 16 KB).
- Ensure that software correctly resets or clears ATU mappings before entering a low-power or reset state to avoid residual access permissions.
- Avoid overlapping ATU mappings that could create unintended access paths between secure and non-secure address spaces.

## 3.5 Key Management Unit

The Key Management Unit (KMU) securely stores symmetric keys used by the RSE. It prevents software access to keys while allowing them to be loaded directly into cryptographic engines. This design removes software involvement in key handling and reduces the risk of side-channel attacks.

For detailed register descriptions and programming information, see [Arm® Key Management Unit Specification](#).

The KMU supports 32 slots for symmetric 256-bit keys, including 7 hardware keys and 25 software keys.

Hardware key slots are always locked and inaccessible to RSE software. They are loaded into the KMU directly by the Lifecycle Manager (LCM), which manages hardware keys stored in OTP. Each hardware key slot's export address, export method, and lock state are configured using hardware parameters. Software can invalidate hardware keys, preventing further use until the next cold reset, when the LCM reloads them.

Software instructs the KMU to verify each hardware key slot to confirm that the key registers are valid and ready for use.

**Table 3-6: Hardware keys and usage**

Key Name	Owner	Purpose
KRTL	Chip manufacturer	Used for secure provisioning in LCS=CM and LCS=DM. Not available in other lifecycle states.
HUK	Chip manufacturer	Hardware Unique Key. Unique to device. Used for cryptographic derivation.
GUK	Chip manufacturer	Group Unique Key. Shared across a group of devices. Used for device identification to ensure authenticity.
KP_CM	Chip manufacturer	CM provisioning key. Used to provision CM assets during manufacturing and deployment.
KCE_CM	Chip manufacturer	CM code encryption key. Used for encrypting chip manufacturer code images.
KP_DM	Device manufacturer	DM provisioning key. Used to provision DM assets and during deployment.
KCE_DM	Device manufacturer	DM code encryption key. Used for encrypting device manufacturer code.

A key is not exported from the LCM to the KMU in the following cases.

- The key is invalid.
- The key does not exist in OTP.
- The current lifecycle state does not permit export (for example, when secure provisioning is not active).

In these cases, the KMU key slot registers remain unpopulated. Software should not attempt to verify or use key slots that are unpopulated in the current lifecycle state. If verification is attempted, it fails and sets the KSKRSM bit in the KMUIS register.

Hardware key exports depend on the lifecycle state, TP mode (PCI or TCI), and whether secure provisioning is active (CM or DM). The following table shows the conditions under which hardware keys are exported. A '+' indicates that the key is exported in that scenario.

**Table 3-7: Hardware key export scenarios**

Key slot #	Key name	CM (virgin)	CM TCI + SP	CM PCI +SP	DM TCI +SP	DM PCI +SP	SE TCI	SE PCI	RMA
0	KRTL	-	-	+	-	+	-	-	-
1	HUK	-	-	-	+	+	+	+	-
2	GUK	-	-	-	+	+	+	+	-
3	KP_CM	-	-	-	+	+	+	+	-
4	KCE_CM	-	-	-	+	+	+	+	-
5	KP_DM	-	-	-	-	-	+	+	-

Key slot #	Key name	CM (virgin)	CM TCI + SP	CM PCI +SP	DM TCI +SP	DM PCI +SP	SE TCI	SE PCI	RMA
6	KCE_DM	-	-	-	-	-	+	+	-

The following sequence describes how software interacts with the KMU to manage software key slots:

1. The RSE secure software accesses the KMU registers that control software key slots.
2. The software loads a symmetric key into an available KMU slot.
3. The KMU verifies the slot by checking that the key has been loaded correctly.
4. The software locks the slot. The KMU prevents any key from being used unless the slot is locked.
5. After locking, the software commands the KMU to verify the slot again to confirm that it is ready for use.



All symmetric keys used by RSE software should be stored in the KMU instead of SRAM. When a cryptographic operation requires a key, the KMU loads it directly into the appropriate cryptographic engine.



If an external cryptographic accelerator is integrated with the RSE, reserve one software key slot for the PKA memory encryption key.



KMU key slot 7 (software key slot) is reserved for the TRAM encryption key. This slot is configured by the DMA during early boot as part of enabling TRAM encryption. Software must configure registers KMUKSC7 and KMUDKPA7, lock the key slot, and verify it. This key slot is also used when the RSE exits memory retention to restore TRAM encryption context.

### 3.5.1 KMU PRBG seed configuration

Before the KMU exports keys, the software initializes the Pseudo-Random Bit Generator (PRBG) state machine with a 128-bit seed. This initialization ensures that the key export process is protected by strong entropy.

The seed is written to the KMUPRBGSI register in four consecutive 32-bit writes. This operation occurs before any key material is exported from the KMU to a cryptographic engine.

The seed value can be sourced from the True Random Number Generator (TRNG). It can also be generated using the Deterministic Random Bit Generator (DRBG) function, depending on system configuration.

### 3.5.2 Verification of hardware key slots

Before software uses the hardware key slots, it triggers the KMU to verify each slot. This ensures the keys are valid and consistent.

The verification process proceeds as follows:

1. The software reads the KMUKSC register as configured by hardware defaults. It then writes the value back with the VKS bit set. This prompts the KMU to verify the consistency of the key slot.
2. The software reads the KMUKSC register again and verifies that the KSR bit is set. This bit indicates that the verification of the key slot was successful.
3. The software checks the KMUIS register to confirm that no errors have occurred. The following error bits must not be set: WDADDKPA, KSNL, KSKRSM, KSDPANS, MWKSW, AKSWPI, or WDALSBDKPA.

Each hardware key slot must be verified individually using this sequence before the KMU exports keys or they are used by cryptographic engines.

### 3.5.3 Programming software key slots

Software should use KMU key slots instead of storing keys in SRAM for several reasons:

- The same key may be reused frequently.
- KMU storage improves key security by isolating key access in hardware.
- Some peripherals, such as TRAM, lose their state in low-power modes and must reload keys from the KMU.

The following sequence describes how software programs a KMU software key slot:

1. The software computes the key to be stored in the KMU, typically by deriving it from another hardware or software key. The computed key is placed in DTCM.
2. The software selects an unused key slot. If the key slot is not cleared, the software invalidates it by setting the IKS bit in the KMUKSC register of that slot.
3. The software copies the key from DTCM to the KMUKSK<n> key registers according to key size. The KMU supports 128-bit and 256-bit keys. For security, the key registers should be written in random order.
4. The software verifies the key registers by reading them back from the KMU and comparing them with the values in DTCM. The registers should be read and compared in random order.
5. The software writes to the KMUDKPA<n> register the address of the first key register for the destination cryptographic device.
6. The software sets the KMUKSC register fields according to the key size and export style. It also sets the NMNKW bit to 1, locks the key slot, and sets the VKS bit to prompt the KMU to verify key consistency.
7. The software reads the KMUKSC register and verifies that the KSR bit is set. This indicates that verification succeeded.

8. The software checks the KMUIS register to confirm that no errors occurred. The following error bits must not be set: WDADDKPA, KSNL, KSKRSM, KSDPANS, MWKSW, AKSWPI, or WDALSBDKPA.

Each software key slot must be programmed and verified individually before the key is used by cryptographic engines or exported to other hardware.

### 3.5.4 Exporting a key from a key slot

When a KMU key slot is configured, it includes a specific address where the key is exported when prompted by software. The software triggers this operation by setting the EK bit in the KMUKSC register. This action instructs the KMU to export the key to the configured target address.

The following sequence describes the key export process:

1. The software sets the EK bit in the KMUKSC register to prompt the KMU to export the key.
2. The KMU exports the key to the destination address configured in the key slot.
3. The software polls the KMUIS register and waits until the KEC bit is set, confirming that the export completed successfully.
4. The software checks that no error bits are set in the KMUIS register.
5. If the KEC bit is enabled in the KMUIE register, the KMU raises an interrupt to signal the completion of the export.

After the export completes, the software can proceed with cryptographic operations that use the exported key.

### 3.5.5 Invalidating a key slot

When software determines that a key slot is no longer required, it invalidates the slot to prevent its use by other software stages. The invalidation process depends on whether the key slot is hardware or software.

#### 3.5.5.1 Hardware key slots

For hardware key slots, invalidation is permanent for the current power cycle. The slot is invalidated by setting the IKS bit in the KMUKSC register. When this bit is set, the KSIP (Key Slot Invalidate Permanent) bit in the KMUKSC register is automatically set by hardware. The key slot remains invalid until the next power-on reset.

#### 3.5.5.2 Software key slots

For software key slots, invalidation wipes the contents of the key slot. After invalidation, the slot can be reused to store a new key.



### 3.5.5.3 Verification

After software issues the invalidation command, it reads back the KMUKSC register and checks that the KSR (Key Slot Ready) bit is cleared. This indicates that the key slot is no longer valid.

## 3.6 Integrating third-party cryptographic engines

Most RSE deployments extend the Integration Layer with partner or customer specific cryptographic accelerators. Hardware and firmware expectations ensure external engines align with RSE security architecture and behave consistently across lifecycle states.

### 3.6.1 Hardware attachment requirements

Guidance outlines mandatory hardware integration requirements before listing specific items.

- Register aperture: Expose the engine through the Integration Layer peripheral window. Follow the RSE bus security conventions (secure attribution via SAU/IDAU and MPU deny-by-default).
- Clocking and reset: Connect the engine to the RSE-managed clock/reset domains. Cold reset must clear key material. Warm reset preserves state only when explicitly required.
- Error signaling: Route alarm, parity, and fault indicators into the Security Alarm Manager (SAM) so system firmware can respond like it does for integrated engines.

### 3.6.2 KMU and key handling

Use the KMU to provision and export keys to external engines without exposing raw key material to the CPU. The following steps must be satisfied:

1. Reserve one KMU software key slot per external engine function that requires keys (for example, AES session keys or PKA private keys).
2. Program the KMUDKPA registers with the destination key register addresses in the external engine.
3. Use the existing KMU export and verification flow so that keys never transit through the processor core.
4. Document whether the engine wipes its key registers automatically on reset; if not, add a DMA scrub step to the boot flow.

### 3.6.3 Boot ROM customization hooks

The RSE ROM only knows about a minimal set of engines. When a platform integrates a different accelerator, update the Integration Layer ROM hooks. The goals are:

- Early-boot provisioning code can trigger self-tests or microcode downloads required by the external engine.
- The DMA “common end procedure” loads any engine-specific context (key ladders, tweak values, or PRBG seeds).
- Lifecycle-dependent code paths (CM, DM, SE) select the correct driver entry points for the new engine.

### 3.6.4 Driver and firmware expectations

Provide robust, side-channel-resilient drivers and define runtime interaction models before implementing the following items.

- Implement constant-time drivers that match the side-channel guarantees of the hardware and reuse the countermeasures listed in [Side-channel protection](#).
- Define a mailbox or doorbell protocol if the engine needs long-running operations so that the RSE scheduler can pre-empt safely.
- Provide integration tests that authenticate, encrypt/decrypt, and verify known vectors so QA can run them during bring-up.

### 3.6.5 Validation checklist

Following this guidance keeps third-party cryptographic IP aligned with expectations for the integrated cryptographic subsystem. It avoids surprises during lifecycle provisioning and simplifies future maintenance.

Area	Validation item
Power/reset	Cold reset clears secrets; warm reset behavior documented and exercised.
KMU interface	Keys export successfully, parity alarms trigger on fault injection, and software never reads raw keys.
Fault reporting	Engine raises SAM alarms for parity/ECC errors and propagates fatal alerts to the reset controller.
Performance	Throughput/latency targets measured to ensure the external engine meets platform budget.
Security	Penetration checks cover glitching, fault-injection, and timing attacks specific to the new engine.

## 3.7 RSE integrated cryptographic engines

The RSE Base and Integration layers do not contain cryptographic engines by default. The integrator of the RSE is responsible for connecting cryptographic engines within the RSE Integration layer.

In the Arm reference design, the RSE includes a set of integrated cryptographic engines that provide essential cryptographic functions.

The RSE integrated cryptographic subsystem includes the following components:

- Hash engine
  - SHA-1, SHA-224, SHA-256
- AES engine
  - Key sizes of 128, 192, and 256 bits
  - Modes of operation:
    - Encryption and decryption: ECB, CBC, CTR
    - Message authentication code: CBC-MAC
    - Encryption and authentication: CCM
- Optional PKA engine
  - Performs regular and modular arithmetic operations ranging from 128 bits to 4160 bits

Hash and AES are part of the default Integration Layer deliverables. The PKA engine is available only on platforms that integrate a compatible public-key accelerator. Otherwise, handle public-key operations with another accelerator or in software.

The RSE cryptographic subsystem includes hardware-level protections against side-channel and fault-injection attacks. However, to achieve full protection, cryptographic drivers must include complementary software-level countermeasures.

All cryptographic engines are expected to include both hardware and software protection against side-channel and fault-injection attacks.

### 3.7.1 Hash

The RSE integrated hash engine does not provide side-channel protection during hash operations. Therefore, it must not be used with secret or sensitive data, because it does not provide side-channel protection. The hash engine is suitable only for non-confidential data such as firmware images, configuration files, or integrity verification values.

The RSE integrated hash engine also does not include integrity protection for its configuration registers or operation flow. Software must ensure that each hash operation completes successfully. Software must also verify that no tampering occurred during processing. This can be verified by

comparing the computed hash result against a precomputed, expected hash value. If a mismatch is detected, the operation may have been corrupted or interfered with.

### 3.7.2 Advanced Encryption Standard

The RSE integrated Advanced Encryption Standard (AES) engine includes protection against side-channel attacks, fault injections, and differential fault analysis (DFA). These protections rely on three main mechanisms that must be properly configured and seeded.

- AES masking
- AES dummy rounds (up to three)
- AES double operation (DFA countermeasure)

To ensure full protection, software must initialize and enable these countermeasures.

#### 3.7.2.1 AES masking

This countermeasure is always active. During each AES operation, random masks are generated on the fly using an internal linear feedback shift register (LFSR) module. Software must seed the LFSR by writing an 8-bit random seed 16 times to the `RNG_SEED` register, producing a 128-bit seed value.

#### 3.7.2.2 AES dummy rounds

This countermeasure is enabled by default. The number of dummy rounds is determined by the same LFSR that generates AES mask values. No additional configuration is required other than seeding the LFSR.

#### 3.7.2.3 AES double operation (DFA countermeasure)

This countermeasure protects against differential fault analysis (DFA) attacks. Software must enable it by writing 1 to the `AES_DFA_IS_ON` register before performing AES operations.

### 3.7.3 Public Key Accelerator

The Public Key Accelerator (PKA) is available on platforms that integrate a compatible accelerator in the RSE integration layer. Platforms without this integration must provide their own big-number engine or run public-key workloads in software.

PKA provides big-number primitives used by public-key cryptography:

- Modular arithmetic (addition, subtraction, multiplication, division)
- Regular arithmetic (addition, subtraction, multiplication, division)
- Modular inversion

- Modular exponentiation
- Logical operations (AND, OR, XOR, SHIFT)

Implement cryptographic algorithms in software. Examples include Rivest–Shamir–Adleman (RSA) or Elliptic Curve Cryptography (ECC) signature/verification and encryption/decryption. Use PKA operations where appropriate.

Apply side-channel and fault-injection hardening when composing algorithms.

### 3.7.3.1 RSA

The RSA engine operates using the following parameters and should be protected by applying blinding countermeasures to prevent side-channel and fault attacks.

**Table 3-9: Input parameters**

m	Public	Message to sign or decrypt
N	Public	RSA public modulus
e	Public	RSA public exponent
p, q	Private	Large prime numbers such that $p * q = N$
d	Private	RSA private exponent, satisfying $e*d = 1 \bmod (p - 1)(q - 1)$

#### 3.7.3.1.1 Basic RSA scheme

The following basic RSA scheme is not secure and should not be used:

$$S = m^d \bmod N$$

#### 3.7.3.1.2 Secured RSA scheme

This secured RSA scheme includes countermeasures against Simple Power Analysis (SPA), Differential Power Analysis (DPA), and Differential Fault Analysis (DFA). It assumes constant-time operation of the hardware multiplier and uses base, modulus, and exponent blinding. This combination provides maximum protection against side-channel attacks.

```
// Generate blinding values
modulusBlindingValue -- random 32-bit integer
exponentBlindingValue -- random 64-bit integer
messageBlindingValue -- random 32-bit integer

// Calculate blinded modulus
N_ = modulusBlindingValue * N

// Calculate blinded exponent
d_ = d + exponentBlindingValue * Φ(N)

// Calculate blinded base
m_ = m + messageBlindingValue * N
```

```
// Perform blinded exponentiation
S_ = m_ ^ d_ mod N_

// Calculate unblinded result
S = S_ mod N

// Verify signature or encryption
M_ = S ^ e mod N
Verify securely: M_ == M

// Return signature or decrypted value
Return S
```

This method ensures that all intermediate operations are randomized, reducing data correlation in power or timing traces and making attacks significantly harder to execute.

3.7.3.2 RSA-CRT

The RSA-CRT uses the following parameters and should be secured using the following scheme.

Table 3-10: Input parameters

m	public	Message to sign/decrypt
N	public	RSA public modulus
e	public	RSA public exponent
p, q	private	Large prime numbers so that p * q = N
d	private	RSA private exponent, satisfying e*d = 1 mod (p - 1)(q - 1)
d <sub>p</sub>	private	CRT exponent such that d <sub>p</sub> = d mod (p - 1)
d <sub>q</sub>	private	CRT exponent such that d <sub>q</sub> = d mod (q - 1)
i <sub>q</sub>	private	q <sup>-1</sup> mod p

Basic (unsecured) RSA-CRT scheme

```
Sp = m ^ dp mod p
Sq = m ^ dq mod q
S = CRT(Sp, Sq) = Sq + q * (( Sp - Sq) * iq ) mod p
```

The result S is the same as

```
S = m ^ d mod N
```

Secured RSA-CRT scheme

The below scheme is aimed to provide countermeasures against SPA, DPA and DFA attacks. It assumes constant time implementation of hardware multiplier.

The scheme uses base, moduli and exponents blinding. This combination protects against the majority of possible attacks.

```
//Generate blinding values
modulusBlindingValue -- random prime 32-bit integer
```

```

exponentBlindingValue-- random 32-bit integer
messageBlindingValue -- random 32- bit integer

//Calculate blinded moduli
p_ = modulusBlindingValue * p
q_ = modulusBlindingValue * q

//Calculate blinded exponents
dp_ = dp + exponentBlindingValue * (modulusBlindingValue - 1) (p - 1)
dq_ = dq + exponentBlindingValue * (modulusBlindingValue - 1) (q - 1)

//Calculate blinded bases
m_ = m + messageBlindingValue * N

//Calculate first partial exponentiation
Sp_ = m ^ dp_ mod p_

//Calculate second partial exponentiation
Sq_ = m ^ dq_ mod q_

//Apply CRT recombination
S_ = CRT(Sp_, Sq_) = Sq_ + q * ((iq * (Sp_ - Sq_)) mod p_)

//Reduce the result modulo N
S = S_ mod N

//Verify signature/encrypt back
M_ = S ^ e mod N

Verify securely: M'== M (compare twice, halt if no match)

//Return signature/decrypted value
Return S

```

### 3.7.3.3 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) algorithms involve performing operations such as point multiplication. They also construct cryptographic schemes like Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve Diffie-Hellman (ECDH). To achieve a secure implementation, countermeasures must be applied at every level of the ECC computation process.

ECC implementations must protect against side-channel leakage, differential fault analysis (DFA), and simple or differential power analysis (SPA/DPA). Countermeasures include operand and scalar randomization, point blinding, constant-time arithmetic, and error detection mechanisms.

For maximum security, each ECC operation should include:

- Randomized projective coordinates for point operations.
- Randomized scalar values by adding a multiple of the curve order.
- Input validation for curve parameters and public points.
- Use of fault detection through result verification or redundant computation.

Hardware modules that support ECC acceleration, such as the PKA or the RSE integrated AES engine, should enable all available countermeasures. Software should validate that blinding and randomization seeds are correctly applied before executing ECC operations.

### 3.7.3.3.1 Montgomery curves

For Montgomery curves such as Curve25519, not all countermeasures described for Weierstrass curves apply.

#### 3.7.3.3.1.1 Random projective coordinates

This countermeasure represents affine coordinates  $(x, y)$  as projective coordinates  $(X, Y, Z)$ , where  $Z$  is randomized. For Montgomery curves, these coordinates represent a point  $(x, y)$  as  $(X, Y, Z)$ , where  $x = X / Z$  and  $y = Y / Z$ . This randomization provides a blinding-like countermeasure against differential power analysis (DPA).

#### 3.7.3.3.1.2 Point multiplication

Side-channel-protected point multiplication is implemented using the uniform Montgomery ladder algorithm.

**Figure 3-1: A uniform Montgomery ladder**

---

#### Algorithm 8: A uniform Montgomery ladder

---

**Input:**  $k = \sum_{i=0}^{\ell-1} k_i 2^i$  with  $k_{\ell-1} = 1$ , and  $\mathbf{x}(P)$  for  $P$  in  $\mathcal{E}_{(A,B)}(\mathbb{F}_q)$   
**Output:**  $(X_k, Z_k) \in \mathbb{F}_q^2$  s.t.  $(X_k : Z_k) = \mathbf{x}([k]P)$  if  $P \notin \{O, T\}$ , otherwise  $Z_k = 0$ .  
**Cost:**  $\ell - 1$  calls to **xADD**,  $\ell$  calls to **xDBL**, and  $\ell - 1$  calls to **SWAP**

- 1  $(x_0, x_1) \leftarrow (\mathbf{xDBL}((X_P, Z_P)), (X_P, Z_P))$
- 2 **for**  $i = \ell - 2$  **down to** 0 **do**
- 3      $(x_0, x_1) \leftarrow \mathbf{SWAP}((k_{i+1} \text{ xor } k_i), (x_0, x_1))$
- 4      $(x_0, x_1) \leftarrow (\mathbf{xDBL}(x_0), \mathbf{xADD}(x_0, x_1, (X_P, Z_P)))$
- 5  $(x_0, x_1) \leftarrow \mathbf{SWAP}(k_0, (x_0, x_1))$
- 6 **return**  $x_0$

---

Where:

```
xDBL :  $\mathbf{x}(P) \rightarrow \mathbf{x}([2]P)$ 
xADD :  $(\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(PQ)) \rightarrow \mathbf{x}(P \# Q)$ 
SWAP : see figure below
```



**Figure 3-2: SWAP: Constant-time conditional swap****Algorithm 7: SWAP: Constant-time conditional swap.****Input:**  $b \in \{0, 1\}$  and a pair  $(x_0, x_1)$  of objects encoded as  $n$ -bit strings**Output:**  $(x_b, x_{1-b})$ 

```

1  $b \leftarrow (b, \dots, b)_n$ 
2  $v \leftarrow b \text{ and } (x_0 \text{ xor } x_1)$  // bitwise and, xor; do not short-circuit and
3 return  $(x_0 \text{ xor } v, x_1 \text{ xor } v)$ 

```

In the Montgomery ladder algorithm above, the first pair initialization is modified as follows:

```
(x0; x1) <-- (x(O); x(P))
```

Instead of the original Montgomery ladder initialization:

```
(x0; x1) <-- (x(P); x([2]P))
```

This modification makes the loop length independent of the scalar  $k$ . The formulas for xDBL and xADD behave correctly under these inputs. As a result,  $x_0$  and  $x_1$  remain unchanged until the first non-zero bit of  $k$  is encountered. This allows a constant-length loop by accepting scalars as fixed-length bitstrings.

### 3.7.3.3.1.3 Scalar splitting

Before scalar multiplication, the scalar is split as follows:

```
r = random 128-bit number
```

Compute:

```

1.  $P_1 = rP$ 
2.  $P_2 = (k - r)P$ 
3.  $P_3 = (k - 2r)P = P_1 - P_2$ 
4.  $[k]P = P_1 + P_2$  (using  $P_3$ )

```

This approach provides additional protection against side-channel attacks by randomizing intermediate computations.

### 3.7.3.3.2 Weierstrass curves

Weierstrass elliptic curve operations in the Runtime Security Engine (RSE) include scalar multiplication and point validation. Countermeasures include scalar blinding, randomized coordinates, and curve integrity checks.

#### Point multiplication

The point multiplication is implemented using left-to-right window method of size 2. The values of  $2P$ ,  $4P$  and  $-P$ ,  $-2P$ ,  $-4P$  are computed before performing the multiplication.

The multiplication uses Jacobian coordinates representation, so that the pre-calculated points use randomized  $Z$  coordinate.

Multiplication algorithm:

```
a: affine (X,Y)
j: Jacobian (X,Y,Z), where  $x=X/Z^2$ ,  $y=Y/Z^3$ 

Convert P to m (with random Z): calculate (mm)  $2P$  and  $4P$ ; initialize other
multiples  $-P$ ,  $-2P$ ,  $-4P$  (negate y)

Obfuscate scalar size: calculate either  $k * p$  or  $-(-k) * p$  in case the scalar
has leading zeros

variable carry: always 0 or -1

variable i: the current bit pair of the scalar (start from top)

BP(i): bit pair,  $k[i+1] * 2 + k[i]$ 

variable s: initialized by  $2P$  or  $4P$  depending on BP(i), carry set accordingly:
(value of BP, initialized with, carry)
( 1, 2, -1)
( 2, 2, 0)
( 3, 4, -1)

for(i -= 2; i >= 0; i -= 2):
    multiply s by 4 (perform 2 double operations)
    add one of the multiples to s:
    (carry * 4 + BP(i), which multiple to add, the new value of carry)
    (-4, -4, 0)
    (-3, 2, -1)
    (-2, 2, 0)
    (-1, -1, 0)
    ( 0, 1, -1)
    (+1, 1, 0)
    (+2, 2, 0)
    (+3, 4, -1)

variable t: s-P, used only then carry is -1

R(w): set result to be w or -w

if carry == -1 then R(t) else R(s)

convert the result to affine coordinate
```

This is a countermeasure against side-channel attacks (SCA).

## Random projective coordinates

This countermeasure represents affine coordinates (x, y) as a set of projective coordinates (X, Y, Z) where Z is randomized. For Weierstrass curves these are Jacobian coordinates. A point (x, y) is represented as (X, Y, Z), where  $x = X / Z^2$  and  $y = Y / Z^3$ .

This is a blinding-like countermeasure against differential power analysis (DPA).

## Point validation

Point validation verifies whether the supplied point lies on the specified curve or not. Point validation should be performed before and after scalar multiplication. If base point does not belong to original curve, error should be returned, and no further calculation performed.

This is a countermeasure against invalid point attacks, including differential fault analysis (DFA).

## Curve integrity check

The curve integrity check is to detect faults on curve parameters before any other calculations are done.

This is a countermeasure against invalid curve attacks (fault attacks, FA).

## Scalar blinding

Perform scalar blinding as follows.

```
Generate small random value r
```

Then the new scalar is

```
(K + r * order_n)
```

The multiplication result with P is unchanged for the original and blinded scalars. This holds because

```
(K + r * order_n) * P = K * P + r * order_n * P = K * P (order_n * P = infinity)
```

Scalar blinding is needed to obfuscate the higher bits of the scalar. Scalar splitting alone cannot achieve this.

## Scalar splitting

Instead of computing  $[s]P$ , the secret scalar s is split using a random divisor r. This obtains quotient  $s_1$  and remainder  $s_2$ , such that  $s = s_1r + s_2$ .

```
s1 = S / r
s2 = S mod r

r - 128-bit random number
```

Compute:

$$\backslash[s\backslash]P = \backslash[s1\backslash]\backslash[r\backslash]P + \backslash[s2\backslash]P, \text{ which is } \backslash[s/r\backslash](\backslash[r\backslash]P) + \backslash[s \bmod r\backslash]P$$

## Elliptic Curve Digital Signature Algorithm

To protect against leakage of ephemeral and secret keys during ECDSA computation, use an improved scheme. Follow these steps:

1. Calculate  $e = \text{HASH}(m)$
2. Let  $z$  be the  $L_n$  leftmost bits of  $e$ , where  $L_n$  is the bit length of the group order  $n$  ( $z$  be greater than but not longer.)
3. Select a cryptographically secure random integer  $k$  from  $\backslash[1, n-1\backslash]$
4. Calculate the curve point  $(x1, y1) = k G$
5. Calculate  $r = x1 \bmod n$ . If  $r = 0$ , go back to step 3.
6. Calculate  $s = k^{-1} (z + rd) \bmod n$ . If  $s=0$ , go back to step 3.
7. The signature is the pair  $(r,s)$ .

Notes on secure implementation of ECDSA scheme:



Note

1. Perform scalar split.
2. Perform scalar blinding.
3. Whenever a multiplication is performed (such as in step 6), randomize multiplier and multiplicand order.
4. Before and after performing a scalar multiplication, always perform a point validation.
5. All scalar multiplication should be performed in randomized projective coordinates.

## Related information

[Side-channel protection](#) on page 95

[Public Key Accelerator](#) on page 60

## 3.8 SRAMs

The RSE contains multiple types of SRAM, each designed for a specific function and security level. These SRAMs are used by different subsystems and hardware modules within the RSE.

The following sections describe each SRAM type, its purpose, and how it interacts with other components during boot and runtime.

- Instruction tightly coupled memory (ITCM)
- Data tightly coupled memory (DTCM)
- Volatile memory (VMO and VM1)
- Trusted RAM (TRAM)

Each SRAM type has distinct access permissions, retention characteristics, and use cases, which are detailed in the following subsections.

### 3.8.1 Instruction tightly coupled memory

The Instruction Tightly Coupled Memory (ITCM) stores software instructions executed by the processor. It provides fast, deterministic access to code and supports ECC protection for reliability.

The ITCM in the RSE is 32 KB in size and 32 bits wide. It includes error correction code (ECC) protection and supports partial writes.

When a partial write occurs, the memory controller performs a read-modify-write operation on a 32-bit entry. It checks the ECC during the read phase and recomputes the ECC during the write phase, both of which occur transparently to software. However, if the ITCM is not initialized before a partial write, the ECC calculation can cause an error because the original ECC value is undefined.

To avoid ECC errors, software or the DMA must write code to word-aligned addresses with word-aligned content sizes when populating the ITCM.

### 3.8.2 Data tightly coupled memory

The Data Tightly Coupled Memory (DTCM) stores data used by the processor. In the RSE, the DTCM is 32 KB in size and 128 bits wide. It is divided into four banks, each 32 bits wide and protected by error correction code (ECC).

The DTCM memory controller supports partial writes in each of its four banks. A partial write triggers a read-modify-write operation on a 32-bit entry. During this operation, the controller checks the ECC on read and recomputes it on write. These operations occur transparently to software. However, if the DTCM is not initialized before a partial write, the ECC computation can produce an error because the original ECC value is undefined.

In the RSE, the DMA initializes the DTCM ECC by writing across the entire DTCM memory before the processor starts. The DTCM functions as a secure memory with protection mechanisms that allow it to store sensitive data. This secure capability is provided through the Trusted RAM (TRAM) peripheral. The DTCM address is scrambled, and its contents are encrypted and wiped at cold boot. For more information, see the [Common end procedure](#).

#### 3.8.2.1 TRAM considerations

The DMA enables TRAM encryption early in the boot process before the processor runs. It sets the TRAM encryption key registers and saves the key in KMU key slot 7. However, the DMA does not configure the remaining registers for key slot 7.

The software completes the setup for key slot 7 as follows:

1. The key registers K<sub>MUKSK</sub><n><0> to K<sub>MUKSK</sub><n><7> are already configured by the DMA.

2. The KMUDKPA<n> register is set to 0x5015\_D008. This address points to the first key register of the TRAM.
3. The KMUKSC<n> register is set to 0x00D6\_0100 for a 256-bit key. The following table shows the fields that compose this value.

**Table 3-11: KMUKSC field configuration for TRAM key setup**

Field	Value	Description
DPWD	0x00	Write delay of 0 (back-to-back writes).
DPAI	0x01	Write address increment of 4 bytes.
DPDW	0b10	Write width of 32 bits.
NDPW	0b01	Number of writes is 8.
NMNKW	0b1	Generate a new mask for every write.
WMD	0b0	Write mask is enabled.
LKS	0b1	Lock the key slot controls.
LKSKR	0b1	Lock key slot key registers.
VKS	0b1	Verify key slot configuration.

### 3.8.2.2 Trusted RAM

Trusted RAM (TRAM) provides side-channel and fault-injection protection for data stored in the processor's DTCM RAM. It obfuscates and scrambles memory addresses and encrypts data using the lightweight encryption scheme QUARMIN-32. This mechanism protects secret data from side-channel observation during read or write operations. The memory also includes error correction code (ECC) protection against fault injection.

#### 3.8.2.2.1 TRAM initialization during boot

Enabling TRAM protection when the DTCM already contains data written by software is complex, as it immediately makes previous data inaccessible. At the early boot stages of the RSE, before the processor starts, the DMA sets up TRAM protection with random keys and enables it. The DMA saves the TRAM encryption key in KMU key slot 7. This allows software to reload it after the RSE exits memory retention mode, where most RSE logic—including TRAM protection—loses context.

The DMA cannot verify the KMU key slot after setup. Software must perform this verification at early boot. If verification fails, it is treated as a fatal error. After successful verification, software confirms that the TRAM key matches the key stored in the KMU. It performs a read/write consistency test on a DTCM location. If the keys do not match, an ECC error occurs. This triggers a reset by the Security Alarm Manager (SAM) due to a processor RAS event.

### 3.8.2.2.2 TRAM reinitialization after memory retention

When the RSE exits memory retention, TRAM protection is disabled. To re-enable it, software loads the encryption and address-scrambling key values from the KMU key slot (set by the DMA) and re-enables TRAM protection.

### 3.8.2.2.3 TRAM memory wipe

As a security measure, after the DMA sets up TRAM protection, it writes random data across the entire DTCM. This operation removes secrets from previous RSE runs and reconditions ECC for all DTCM entries.

### 3.8.2.2.4 Secure use of DTCM with TRAM

For secure and efficient use of the DTCM, the following guidance applies.

- Keep RSE confidential assets in DTCM RAM, such as:
  - Asymmetric keys (for example, attestation keys)
  - Device Identifier Composition Engine (DICE) identities (CDIs), context handles, and DICE Protection Environment (DPE) values
  - Symmetric keys, if KMU key slots are not used (KMU key slots are preferred)
  - Long-term assets that will later be stored in OTP during provisioning
- Align assets stored in DTCM to word boundaries for efficiency and reliability.
- Load the TRAM encryption key using a KMU software key slot. After configuration, lock the key slot using the LKSKR field of the KMUKSC<n> register. Locking prevents accidental or malicious modification and is retained when the RSE enters retention mode.

### 3.8.2.2.5 TRAM interface

The TRAM interface is mapped to the peripheral region at address 0x5015\_D000. Its register layout depends on the RSE customization layer but generally follows the structure below.

Field name	Offset	Access	Description
trbc	0x000	R	TRAM build configuration register
trc	0x004	R/W	TRAM configuration register
trkey[8]	0x008	W	TRAM key registers (8 × 32-bit words)
reserved_0	0x028–0xFCC	—	Reserved
pidr4	0xFD0	R	Peripheral ID 4
reserved_1[3]	0xFD4–0xFDC	—	Reserved
pidr0	0xFE0	R	Peripheral ID 0
pidr1	0xFE4	R	Peripheral ID 1
pidr2	0xFE8	R	Peripheral ID 2
pidr3	0xFEC	R	Peripheral ID 3

Field name	Offset	Access	Description
cidr0	0xFF0	R	Component ID 0
cidr1	0xFF4	R	Component ID 1
cidr2	0xFF8	R	Component ID 2
cidr3	0xFFC	R	Component ID 3

### 3.8.3 L1 instruction cache

The L1 instruction cache in the RSE is 16 KB in size and protected by error correction code (ECC). It is used by the processor when executing code stored in VM0 or VM1 memory regions.

For optimal performance, the software enables the instruction cache and configures the corresponding MPU region as cacheable.

### 3.8.4 L1 Data Cache

Its size in RSE can be 4KB or 8KB depending on the needs. This SRAM is ECC protected. This cache is used by the CPU to handle data from VM0, VM1.

For performance reasons, the software should enable the data cache and set a respective MPU region to be cacheable.

### 3.8.5 Volatile memory

VM0 and VM1 are 512 KB static random-access memories (SRAMs) located contiguously in the Runtime Security Engine (RSE) memory map. Each memory is 64 bits wide and protected by error-correcting code (ECC). VM0 and VM1 can be accessed concurrently and independently. Code can execute from VM0 while VM1 handles reads and writes in parallel, provided the Advanced eXtensible Interface (AXI) interconnect is not the bottleneck.

Each memory implements Single Error Correction and Double Error Detection (SECCDED) ECC. ECC is calculated over full 64-bit words. All writes to VM0 and VM1 must be 64-bit aligned quad-word accesses. Partial or unaligned writes are accepted, but they trigger the Partial Write Detector, mark the ECC invalid, and raise an interrupt.

See [Correcting ECC following partial writes to VM0 and VM1](#) for recovery of partially written entries. For error handling, see [Correcting ECC following SEC ECC error in VM0 and VM1](#) and [Handling of DED ECC error in VM0 and VM1](#).



RSE-100 hardware has a limitation. VM0 and VM1 must operate in the same minimum power mode (either MEM\_RET or ON). It is not possible to mix power modes. These modes are configured using the same value in PDCM\_PD\_VMR<m>\_SENSE.MIN\_PWR\_STATE where m is 0 and 1.



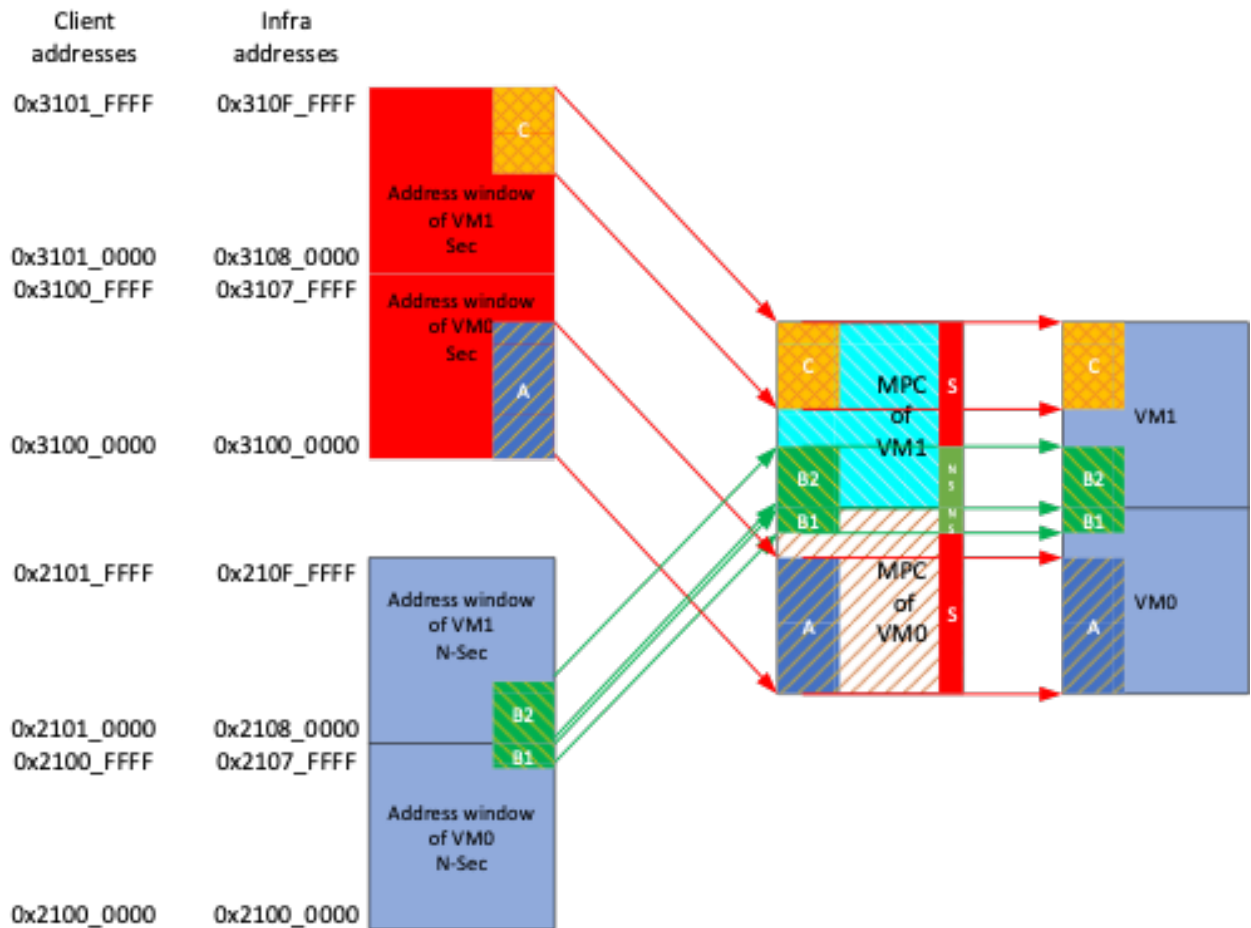
VM0 and VM1 support both secure and non-secure accesses. Base address 0x2100\_0000 is used for non-secure accesses; base address 0x3100\_0000 is used for secure accesses. Software must ensure that secure and non-secure allocations within VM0 and VM1 do not overlap physically.

A Memory Protection Controller (MPC) is positioned in front of each VM. Each MPC, and its VM, interpret only the address bits that fit the memory size. For a 512 KB VM, the MPC interprets 19 address bits. As a result, the MPC cannot differentiate the security state of in-flight transactions. Maintain secure/non-secure separation through region mapping within the MPC.

Allocate secure and non-secure regions within VM0 and VM1 so that the least significant 19 address bits of each region do not overlap. Configure each MPC to permit secure or non-secure access as required.

The diagram illustrates the following:

- Address range A is located at the bottom of the secure address window of VM0. The MPC for VM0 configures the corresponding blocks as secure.
- Address range C is located at the top of the secure address window of VM1. The MPC for VM1 configures the corresponding top blocks as secure.
- Address range B spans both VM0 and VM1, split into two parts:
  - B1 is at the top of the non-secure address window of VM0.
  - B2 is at the bottom of the non-secure address window of VM1. The MPCs for both VMs configure these respective blocks as non-secure.
- To protect non-secure software from accessing secure memory, the intermediate MPC blocks in both VMs are configured as secure.

**Figure 3-3: VM0, VM1**

### 3.8.5.1 Avoid partial writes to VM0 and VM1

To reduce partial writes to VM0 and VM1, configure both memory spaces as cacheable (write-back with read and write allocation). This setup enables the processor cache to perform read-modify-write operations transparently, lowering the risk of incomplete writes.

When parts of VM0 or VM1 require coherent access for Direct Memory Access (DMA) operations, follow these steps.

1. Invalidate the affected cache lines after software writes and before DMA reads.
2. Invalidate the affected cache lines after DMA writes and before software reads.

These steps minimize the chance of partial writes.

DMA engines, such as the Arm DMA-350 or the Arm RSE-integrated AES engine, must.

- Write to 64-bit aligned addresses.
- Write complete 64-bit words.

The processor's store buffer (STB) can write directly to VM0 or VM1 memory, bypassing the cache. Such writes may occasionally be partial.

The RSE includes a Partial Write Detector to identify and allow correction of these rare partial writes. For details, see [Correcting ECC following partial writes to VM0 and VM1](#).

### 3.8.5.2 Avoid polluting the data cache when copying large data

When software copies large data buffers from one location to another, cache pollution can occur. If one or both memory regions are cacheable, the copied data can fill the data cache. This can evict lines that may still be needed after the copy completes. This leads to performance degradation because subsequent accesses to the evicted data cause cache misses.

To prevent this issue, software can use the DMA to perform large buffer copies instead of the processor. DMA transfers bypass the data cache and avoid polluting it.

If DMA is not used, the software can reduce cache pollution with the following approach.

1. Temporarily configure the source and destination addresses as non-cacheable for the duration of the copy.
2. Copy the data efficiently by performing read and write multiple operations in a loop using as many registers as possible.
3. Restore the cache settings of the affected memory regions after the copy completes.

When the destination address is located in VM0 or VM1, writes must be 64-bit aligned to avoid partial writes. For more information, see [Avoid partial writes to VM0 and VM1](#).

### 3.8.5.3 Correcting ECC following partial writes to VM0 and VM1

A partial write to VM0 or VM1 can result in corrupted ECC for the target 64-bit double word. Although the data in SRAM is likely correct if no attack occurred, software should fix the ECC quickly to prevent data corruption over time.

VM0 and VM1 include partial write detectors. When a partial write occurs, the detector marks the affected 64-bit double word as having invalid ECC. It raises `IRQ[61]` and sets the SAM event `VMPARWRITE`. This event is typically masked to prevent a reset response.

The Security Alarm Manager (SAM) captures the address of the partial write. It includes a capture address register for each VM, while `IRQ[61]` is shared across all implemented VMs. If multiple partial writes occur before the software handles the event, the SAM captures the most recent event address. However, all partially written double words are marked in memory.

When software reads a marked address, the ECC error does not raise an alarm. Instead, the partial write detector raises `IRQ[61]` again. This behavior allows software to correct the ECC.

Software must clear SAM's reset response to the `VMPARWRITE` event and enable `IRQ[61]` by setting bit[11] of `SAMRRLS2` to 0. This configuration is typically applied by SAM override values copied by the DMA from the OTP.

The `IRQ[61]` handler performs the following steps (secure access required):

1. Check if bit `VMPARWRITE` (bit[18]) in the `SAMES0` register of SAM is set. If not, the interrupt is spurious; proceed to step 7.
2. Read the `VMPWCA0` or `VMPWCA1` register of SAM to determine whether VM0 or VM1 signaled the partial write. If the result is `NULL`, the VM did not issue the signal. Otherwise, the register contains `AddressLSB`—the 16 or 19 least significant bits of the address where the partial write occurred.
3. Construct a secure or non-secure address by adding `AddressLSB` to the corresponding base address of the VM.
4. Load the double word from the address and store it back to the same address. This operation triggers the partial write interrupt again.
5. Write `NULL` to the `VMPWCA0` or `VMPWCA1` register to prepare it for the next event.
6. Clear the `VMPARWRITE` bit using the `SAMECL0` register.
7. Acknowledge and clear the interrupt.

#### 3.8.5.4 Correcting ECC following SEC ECC error in VM0 and VM1

The ECC controller for VM0 and VM1 detects and corrects single-bit errors. The corrected data is provided to the reader, but the affected memory entry is not automatically fixed by hardware.

When this event occurs in either VM0 or VM1, `IRQ[37]` is raised. The Security Alarm Manager (SAM) sets the event `scECCVM` for the VM in which the error occurred. This event is typically masked to prevent an automatic reset response.

Such an error might result from a random alpha particle strike or a transient electrical event. If the cause is suspected to be an attack, software can configure SAM to trigger a reset in response. Alternatively, it can log the masked event, apply the correction flow below, and escalate to a reset if multiple events occur.

The SAM captures the address where the corrected ECC error occurred. Each VM has its own capture address register, but `IRQ[37]` is shared between all implemented VMs. If multiple single-bit ECC errors occur before software handles the first one, SAM captures only the most recent address. However, hardware will correct any previously affected memory entries again on subsequent reads. When software reads a memory location that has been corrected, the hardware re-raises `IRQ[37]`. The hardware also updates the SAM event with the corresponding address.

Software must clear SAM's reset response to `scECCVM` by setting bits 15, 19, 23, and 27 of `SAMRRLS2` to 0 and enable `IRQ[37]`. This configuration is typically applied through SAM override values that the DMA copies from the OTP.

The `IRQ[37]` handler performs the following steps (secure access required):

1. Check whether a single-bit ECC error is reported for VM0 or VM1 by verifying that bit 19 (`sceccvm` for VM0) or bit 20 (`sceccvm` for VM1) in the `sames0` register of SAM is set.
2. Read `vmsceeca0` or `vmsceeca1` to obtain the least significant bits of the address (`AddressLSB`) for the affected entry. If `NULL`, the error is in the other VM. Add `AddressLSB` to the base address of the relevant VM to compute the full address of the error location.
3. Load the double word from the computed address and immediately store it back. This operation triggers the ECC correction process again, ensuring that the ECC is updated.
4. Write `NULL` to `vmsceeca0` or `vmsceeca1` to prepare the register for the next event.
5. Clear the corresponding `sceccvm` bit (19 or 20) in `sames0` by writing to the same bit in `samecl0`.
6. Acknowledge and clear the interrupt.

### 3.8.5.5 Handling of DED ECC error in VM0 and VM1

The ECC controller for VM0 and VM1 detects, but does not correct, double-bit ECC errors. When such an event occurs in VM0 or VM1, the SAM raises the corresponding alarm event (`ECCERRVM0` or `ECCERRVM1`). It captures the offending address in the `vmdueeca0` or `vmdueeca1` registers. These registers retain their values across cold resets.

At power-on reset, the SAM is completely reset, so the following process applies only to cold resets. After a cold boot, the following applies if the reset is due to a SAM cold boot. This condition is indicated by the `SAMWRSTREQ` bit set in the `RESET_SYNDROME` register. If tracking of uncorrected ECC errors is required, the RSE software performs the following actions:

1. Check that a double uncorrected ECC error occurred by verifying that bit 23 (`ECCERRVM0`) or bit 24 (`ECCERRVM1`) of the `sames0` register is set. This step ensures that the SAM did not trigger the reset for another error or attack.
2. Read the `vmdueeca0` or `vmdueeca1` register to obtain the least significant bits (`AddressLSB`) of the address where the error occurred. If the value is `NULL`, the error occurred in the other VM bank.
3. Add the `AddressLSB` to the base address of the corresponding VM bank to compute the full address of the DED error.
4. If required, log the offending address and increment the counter of DED events. Optionally, the device can be permanently disabled (“bricked”) if too many such errors occur.
5. Store a double word at the target address. If the cold reset reloads the VM0 and VM1 contents, this step can be skipped.
6. Write `NULL` to the `vmdueeca0` or `vmdueeca1` register to prepare it for the next event.

### 3.8.6 Prevention of ECC errors caused by speculative cache prefetching

ITCM, DTCM, VM0, and VM1 memories are ECC protected. At power-up, their ECC values are not initialized. Software or the DMA can write to these memories to initialize ECC. This process

can be time-consuming, except for the DTCM where the TRAM setup initializes ECC automatically. Guidance explains safe use of ECC-protected memories without pre-writing all entries.

Typical software reads only the memory regions it has previously written or initialized. Writing data generates valid ECC tags, so there is no risk of reading incorrect ECC values. Executing code from uninitialized memory is a software bug. Any attempt to do so will trigger an ECC error and reset the RSE. This behavior acts as a security measure because unpredictable behavior is prevented. The same principle applies to data access. For more details, see [Avoid partial writes to VM0 and VM1](#).

For performance reasons, both the instruction and data caches should be enabled. The processor cache may speculatively prefetch the next cache line of code or data. The Cortex-M55 processor prefetch behavior differs for code and data.

### 3.8.6.1 Prefetching of code from VM0 and VM1

Speculative instruction fetch occurs when a branch instruction is detected. The prefetcher reads the cache line for the instruction following the branch instruction. If the branch occurs at the end of a code section, the prefetcher might read beyond the valid code area. This behavior can trigger an ECC error. Instruction prefetching is always enabled by the processor.

To prevent this issue:

1. Write zeros to the cache line (32 bytes on the Cortex-M55 processor) immediately after the last cache line address of the valid code section, if no other written section follows it.
2. Set the MPU region for the address following the end of the code to “Execute Never” or “Device” to disable cache prefetching.

### 3.8.6.2 Prefetching of code from ITCM

ITCM is accessed directly by the processor instruction pipeline and does not use the instruction cache. The instruction pipeline may read ahead to fill the decode buffer. If this happens at the end of the code in ITCM, it may attempt to access an uninitialized ECC entry.

To prevent this issue:

1. Write zeros to the next word-aligned address at the end of the code.
2. Set the MPU region for the address following the end of the code to “Execute Never” to disable pipeline prefetching.

### 3.8.6.3 Prefetching of data from VM0, VM1, and DTCM

The Cortex-M55 processor prefetcher predicts data access patterns. It performs prefetching based on observed strides in data streams (for example, accessing columns in a matrix). When prefetching occurs within valid data, no issue arises because ECC is valid. However, when the prefetcher

accesses memory beyond initialized data (for example, at the end of a data section), an ECC error occurs and halts execution.

To minimize the risk of ECC errors:

- Combine data sections so that they are contiguous where possible (subject to security constraints) to reduce uninitialized gaps.
- After the end of the final data section, write zeros up to the next 8 KB boundary. The Cortex-M55 prefetcher does not cross this boundary.
- Alternatively, configure the MPU region following the end of the data section as “Device,” which disables the prefetcher.

These precautions are especially important for VM0 and VM1. These memories are not initialized by the DMA during power-on reset. In contrast, the DTCM is fully initialized for security reasons and is smaller in size, allowing quick ECC setup.

## 3.9 TRNG and DRBG

The RSE requires random numbers for cryptographic operations. Random number generation depends on a True Random Number Generator (TRNG) source, which is typically slower than a deterministic generator. To improve performance, a Deterministic Random Bit Generator (DRBG) uses a seed from a high-entropy TRNG source to produce random values efficiently. The seed can be reused for multiple DRBG operations before the DRBG requires reseeding from the TRNG.

The TRNG hardware is integrated into the RSE subsystem and provides true entropy for seeding DRBGs and other cryptographic operations. On boot, the TRNG provides sufficient randomness for seeding the Lifecycle Manager (LCM). It also generates random keys used by the Direct Memory Access (DMA) to configure Trusted RAM (TRAM) encryption. Provisioning records and DRBG seeding metadata should be stored in One-Time Programmable (OTP) memory; runtime entropy is always sourced from TRNG hardware.

When software starts, it switches the TRNG to high-quality mode before use. For PUFSecurity One-Time Programmable (OTP) memory, software sets the `rnd_sel` bit in the register at offset `0x0024` from `RNG_BS` to 1. For detailed setup and operation, refer to the PUFSecurity guideline flows.

## 3.10 Message handling unit

The Message Handling Unit (MHU) provides a communication channel between the RSE and other components in the SoC. It operates as a peer-to-peer interface that connects directly to a specific target.

Each MHU handles unidirectional message transfers from a sender to a receiver. To enable bidirectional communication, two MHUs are required—one for each direction.

Messages can be transferred directly through the sender and receiver frames or indirectly through shared memory.

When passing messages to or from a destination, software must apply appropriate security considerations. This includes validating the source and destination addresses, ensuring message integrity, and following access control policies defined for secure and non-secure domains.

### 3.10.1 Integrity

The internal logic and registers of the MHU hardware module are not protected against fault injection. Therefore, RSE software must ensure the integrity of data handled through the MHU.

For hardware registers, the RSE software performs a read-after-write check when loading values into MHU registers. This confirms that written data matches the expected value and has not been modified by a fault event.

The communication protocol implemented on top of the MHU must also include integrity protection for all data transmitted through the channel. This protection can be achieved by using cryptographic checks such as message authentication codes (MACs) or hash-based verification.

### 3.10.2 Confidentiality

Depending on the data transmitted through the MHU, designers must determine whether confidentiality protection is required. For example, when secrets are passed through the MHU, they must be transmitted in an encrypted or obfuscated form.

Encryption is required if the MHU channel is accessible to potential attackers. Obfuscation (masking) is sufficient when only side-channel protection is necessary and direct access to the MHU is not possible.

For instance, when two RSE instances are connected directly within an SoC and exchange secrets, masking protects the data from side-channel leakage. When the communication channel extends across the PCB or beyond, cryptographic encryption must be used to protect the transmitted secrets.

### 3.10.3 Replay protection

Depending on the system design and the physical communication channel, designers must determine whether replay protection is required. For example, when using an MHU channel between two trusted RSE instances within the same SoC, the channel cannot be physically probed. In such cases, replay protection may not be necessary.

However, when the MHU channel connects entities on the PCB or across different systems, data freshness becomes critical. Replay protection must be implemented to prevent attackers from reusing previously transmitted valid messages. This can be achieved through mechanisms such as sequence counters, nonces, or timestamps embedded in the message protocol.



## 3.11 Security Alarm Manager

The Security Alarm Manager (SAM) collects alarm signals in the RSE and applies predefined responses to specific events. An alarm is a fault detection event triggered by a monitoring mechanism inside or outside the RSE.

SAM receives 64 alarm signals from digital detection and sensing circuits designed to flag abnormal faults. Treat any detected anomaly as a potential attack on the RSE, unless explicitly documented otherwise for a specific event. Alarm signals from RSE digital sensors connect to SAM inputs.

Some input events are grouped into a single SAM input. When a combined event is triggered, ensure software handles all contributing sub-events for that SAM input.

Certain events are only logged in `SAMES` registers. For example, bus parity errors are routed to a reset response and can be analyzed by software after reset.

When a reset occurs due to a SAM response, the `SAMES` registers retain their values across a cold reset. This retention enables post-reset analysis of the cause of the SAM-initiated reset. Other events also have error record registers that persist across cold boots. Examples include:

- `SAMCDRES` records processor-generated DCLS errors. These are reset when SAM resets the RSE, enabling post-reset analysis.
- `SAMRRS` records processor-generated RAS errors and is also reset when SAM resets the RSE.
- `VMPWCA` captures the location of a partial write to VM0 or VM1.

Input alarm signals 48–63 are routed to the RSE top and are visible to the integration layer. Integrators can connect analog sensors, such as voltage glitch, over/under voltage, clock glitch, or over/under temperature sensors. They can also connect digital sensors, such as active shields, to these alarm interfaces.

Disable unused alarm inputs by tying them to 0.

Allow analog sensors time to stabilize after power-up. During this period, analog sensors may generate false alarms. Set the External Sensors Ready signal only after all analog sensors are stable.

SAM registers have power-on reset defaults to protect the RSE during boot. When the RSE exits reset, internal digital sensors are enabled. Alarm inputs from the integration layer remain disabled. After reset, SAM waits for new register configurations (typically loaded by DMA) and for the External Sensors Ready signal. Once both are applied, SAM triggers DMA to continue the RSE boot flow. If incorrect integrity check values are written to the configuration registers, SAM signals an internal alarm routed to a reset response.

After a reset, initialize SAM using DMA or software. In the RSE, DMA typically performs this during `LCS=SE`. It copies SAM configuration values from OTP to the SAM configuration registers. These registers are protected by an Integrity Check Value (ICV). SAM computes an integrity check on the copied configuration and applies it to shadow registers only if the check matches. At runtime, configuration may include analog sensor enablement, as required by the integrator.

## SAMIM, SAMEM, and SAMRRLS

These registers configure how SAM enables, handles, and exports input events.

Each input connected to SAM must be explicitly enabled by setting its corresponding bit to 1 in the `SAMIM` register. At power-on reset, only SAM internal events and RSE digital sensors or checkers are enabled. The 16 input events from the integration layer at the Sensor Alarm Interface are masked by default at reset. The relevant bits for these events may be set in the SAM section of the OTP. They can be loaded automatically by the DMA or later by software.

The following table lists the SAM input events with their recommended response settings and masks. Each event's response enable bit and response number determine the routing bits in the `SAMRRLS` registers. Each event (0–63) is represented by four bits in the `SAMRRLS0`–`SAMRRLS7` registers. They start from bit 0 of `SAMRRLS0` and increase with the event ID.

Input event mask enable refers to the bit in the `SAMIM` register. Event export mask enable refers to the bit in the `SAMEM` register.

**Table 3-13: SAM input events**

Event Index	Event source	Event signal name	Response enable	Response number for this event	Input event mask enable	Event export mask enable
0	SAM Configuration Integrity Checker.	SAM Configuration Integrity Error	1	0 – Cold Reset	1	1
1	SAM Watchdog Timer	SAM WD interrupt	0	2 - NMI.	1	1
2	SAM Duplication Error	sam_duperr	1	0 – Cold Reset	1	1
3	LCM Fatal Error	lcm_fatal_err, ORed with debug_signals_security_checker_err.	1	0 – Cold Reset	1	1
4	CPU Lockup	LOCKUP	1	0 – Cold Reset	1	1
5	ATU Error	ATUERR	1	0 – Cold Reset	1	1
6	KMU Parity Error	kmu_parerr	1	0 – Cold Reset	1	1
7	Crypto Combined Parity Error	cr_comb_par_err	1	0 – Cold Reset	1	1
8	SIC Parity Error	SIC_PARITY_ERROR ORed with the SIC WMF parity error	1	0 – Cold Reset	1	1
9	AES DFA	cc_aes_dfa_err	1	0 – Cold Reset	1	1
10	AES Parity Error	cc_aes_par_err	1	0 – Cold Reset	1	1
11	DMA-350 DCLS Error	DMADCLSERR	1	0 – Cold Reset	1	1
12	PSI (AON) Parity Error		1	0 – Cold Reset	1	1

Event Index	Event source	Event signal name	Response enable	Response number for this event	Input event mask enable	Event export mask enable
13	Bus Parity Error	BUSPARERR	1	0 – Cold Reset	1	1
14	Processor DCLS	CPUDCLSERR	1	0 – Cold Reset	1	1
15	Processor RAS events set 0	CPURASERR0	1	0 – Cold Reset	1	1
16	Processor RAS events set 1	CPURASERR1	1	0 – Cold Reset	1	1
17	Processor RAS events set 2	CPURASERR2	1	0 – Cold Reset	1	1
18	SRAM Partial Write	VMPARWRITE	0	0 – Cold Reset (N/A)	1	1
19	VM0 Single SEC ECC Error	SCECCVM0	0	0 – Cold Reset (N/A)	1	1
20	VM1 Single SEC ECC Error	SCECCVM1	0	0 – Cold Reset (N/A)	1	1
21	Reserved	Reserved	0	0 – Cold Reset (N/A)	0	1
22	Reserved	Reserved	0	0 – Cold Reset (N/A)	0	1
23	VM0 ECC Error	ECCERRVM0	1	0 – Cold Reset	1	1
24	VM1 ECC Error	ECCERRVM1	1	0 – Cold Reset	1	1
25	Reserved	Reserved	0	0 – Cold Reset (N/A)	0	1
26	Reserved	Reserved	0	0 – Cold Reset (N/A)	0	1
27	Reserved	Reserved	0	0 – Cold Reset (N/A)	1	1
28	Reserved	Reserved	0	0 – Cold Reset (N/A)	1	1
29	ATU Parity Error	ATUPARERR	1	0 – Cold Reset	1	1
30	Processor Private Region Parity Error	PPRPARERR	1	0 – Cold Reset	1	1
31	System Control Registers Block Parity Error	SCPARERR	1	0 – Cold Reset	1	1
32	CPU PPB Parity Error	PPBPARERR	1	0 – Cold Reset	1	1
33	Secure Access Configuration Registers Block Parity Error	SACPARERR	1	0 – Cold Reset	1	1

Event Index	Event source	Event signal name	Response enable	Response number for this event	Input event mask enable	Event export mask enable
34	Non-Secure Access Configuration Registers Block Parity Error	NSACPARERR	1	0 – Cold Reset	1	1
35	Integrity Checker alarm	ic_alarm ORed (in integration layer) with parity error detection at the Integrity Checker APB and AXI interfaces.	1	0 – Cold Reset	1	1
36	TRAM parity error	TRAMPARERR – OR of the TRAM's parerr and of its WMF parerr signals.	1	0 – Cold Reset	1	1
37..47	Reserved		0	0 – Cold Reset (N/A)	0	0
48..63	External sensor event		0	0 – Cold Reset (N/A)	0	0



The VM partial write and corrected ECC errors (event ids 18 to 20) have response disabled because they are handled by IRQ. They however have input mask enabled because the SAM has to capture the address location of the error.

The resulting recommended setup values for SAM registers are:

**Table 3-14: Recommended SAM configuration**

Register	Value
SAMIM0	0xF99F_FFFF
SAMIM1	0x0000_001F
SAMEM0	0xFFFF_FFFF
SAMEM1	0x0000_001F
SAMRRLS0	0x8888_8828
SAMRRLS1	0x8888_8888
SAMRRLS2	0x8000_0088
SAMRRLS3	0x8880_0008
SAMRRLS4	0x0000_8888
SAMRRLS5	0x0000_0000
SAMRRLS6	0x0000_0000
SAMRRLS7	0x0000_0000

## SAMEC

The RSE supports four event counters (SAMEC0–SAMEC3). The additional counter registers (SAMEC4–SAMEC7) are reserved. Event counters are used to filter transient analog sensor events that may occur shortly after boot. Each counter decrements from a programmed starting value each time its connected event signals. When the counter reaches zero, the event is reconnected to the SAMEM register and can raise an alarm. This mechanism helps suppress false alarms that occur during sensor stabilization after reset.

The offsets of unimplemented counters within the SAM register block are reserved and behave as **RAZ/WI** (read-as-zero, write-ignored). The Integrity Check Value (ICV) excludes these unimplemented counters from its zero-count calculation. For simplicity, DMA or software can copy a complete register block from OTP or memory, because writes to unimplemented **SAMEC** registers are ignored. Any value can be written; however, all-0xFF bytes are commonly used for OTP simplicity. Manufacturing practicality and OTP size should also be considered.

The copied configuration content must pass an integrity check before the ICV value is written to the **SAMICV** register.

The recommended setup values for these registers assume that no input signals require filtering and that space is reserved in OTP for unimplemented registers.

**Table 3-15: Recommended counter register configuration values**

Register	Value	Notes
SAMEC0	0x00000000	Counter disabled
SAMEC1	0x00000000	Counter disabled
SAMEC2	0x00000000	Counter disabled
SAMEC3	0x00000000	Counter disabled
SAMEC4	0xFFFFFFFF	Unimplemented counter
SAMEC5	0xFFFFFFFF	Unimplemented counter
SAMEC6	0xFFFFFFFF	Unimplemented counter
SAMEC7	0xFFFFFFFF	Unimplemented counter



Any value can be written to unimplemented counters because writes are ignored. The ICV does not include these counters. All 0xFF values simplify DMA copy and OTP provisioning, although they increase OTP programming time.

## SAMECTIV

When the **SAMEC** event counters are enabled to filter out unstable or transient input events, the count configuration must balance sensitivity and reliability. If false events are unpredictable, overestimating the counter value can suppress genuine alarms.

Spurious alarms typically occur only shortly after boot, while analog sensors stabilize. If the expected stabilization period is known, **SAMECTIV** can be configured to count down using RSE clock cycles. When the countdown expires, SAM forces all active **SAMEC** counters to expire, ensuring that event filtering does not continue indefinitely.

## SAMRRLS

The eight **SAMRRLS** registers define routing for each unmasked input event to one of eight response action signals. Each recorded event (0–63) has four routing control bits within the **SAMRRLS** registers. They start from **SAMRRLS0** bits [3:0] for event 0 and increase sequentially with event IDs.

Bits [2:0] select which response signal (0–7) is raised when the event is detected. Bit [3] acts as an enable bit that allows the event to be detected without necessarily raising an output signal.

This capability enables software to monitor specific events through interrupts instead of hardware-triggered actions.

**Table 3-16: SAM response action output signals and RSE handling**

Response Action ID	Signal Name	RSE Action Taken
0	SAMCRSTREQ	Cold Reset of the RSE. This is the primary reset mechanism triggered by hardware faults or configuration integrity violations. Should reset the subsystem cleanly.
1	SAMWRSTREQ	Warm Reset Request. Triggers a warm reset, typically used for recoverable faults. It must be edge-detected externally to avoid latching issues in the SAM. This action is not recommended to be used, as warm reset of the RSE is reserved for the secure provisioning flow.
2	NMI Interrupt	Non-Maskable Interrupt. Used for high-priority events that require immediate software handling. Cleared once software clears the corresponding SAM event status.
3	Critical Fault IRQ	Critical Severity Interrupt. Signals urgent software attention; requires event clearing in SAMESx registers to deassert.
4	Severe Fault IRQ	Severe Fault Interrupt. Treated as a significant fault requiring immediate software intervention. Also cleared via software by clearing SAMESx.
5	SAM_ACTION_5	User-Defined Action. Integrator-defined response action. Typically software-monitored and cleared by event status register management.
6	SAM_ACTION_6	User-Defined Action. Another general-purpose response route. Typically used for integration-specific behaviors.
7	SAM_ACTION_7	User-Defined Action. Flexible integration hook. Often connected to interrupt or reset infrastructure in custom SoC settings.

## SAMWDCIV

When an alarm response is routed to an interrupt, software should handle it promptly. If software becomes unresponsive, the SAM watchdog provides a safeguard mechanism. The watchdog starts counting down when any unmasked response action is triggered.

To configure the watchdog, software writes a timeout value (in RSE clock cycles) to the `SAMWDCIV` register. It also sets one or more response action mask bits corresponding to the enabled interrupt responses. When the timeout expires without a valid software response, SAM raises alarm signal 1, which should be routed to a cold reset action.

At reset exit, the response action for SAM event 1 is disabled. If the watchdog mechanism is required, software must enable the corresponding response action after reset.

## SAMEM

The `SAMEM` registers control the export of detected events recorded in the `SAMES` registers to the RSE PSI signals for system-level handling. Exposing detailed fault information externally is generally not desirable for security reasons. To prevent disclosure of sensitive information, multiple events can be combined into a single exported signal using `SAMEM` configuration. This approach allows the SoC to monitor fault status without revealing specific internal alarm conditions.

### 3.11.1 Handling of RSE cold reset initiated by SAM

Alarm signals routed to SAM are serviced by the SAM. It typically responds with a cold reset of the RSE.

The SAM holds contents of the `SAMES0` and `SAMES1` status registers and additional error-specific record registers across the reset. This allows software to handle the offending errors after the reset, if that is required.

Most of these errors are assumed to happen due to an attack and the reset is an appropriate, immediate consequence. However, software may perform further handling following the cold reset according to security policy. Some examples might include:

- Logging offending addresses and numbers of attempts
- Bricking the platform permanently

After cold boot, if the reset is due to SAM action (`SAMCRSTREQ` bit in `RESET_SYNDROME` is set), the RSE software might perform the following actions. This applies when further handling of SAM events is required.

- Check that `SAMCRSTREQ` bit in `RESET_SYNDROME` register is set
- Check the `SAMES0` and `SAMES1` registers to determine SAM issued a reset
- Handle double un-corrected ECC error in VM0, VM1
- Handle CPU DCLS reported errors.
- Handle CPU RAS reported errors.
- Handle SAM reported errors.
- Clear reported SAM errors status bits using `SAMECL0` and `SAMECL1`
- Clear `SAMCRSTREQ` bit in `RESET_SYNDROME` register



This operation requires writing `RESET_SYNDROME` register with all bits set to one except `SAMCRSTREQ` which is set to zero as this register has RWOC behaviour.

---

#### 3.11.1.1 Handling of processor DCLS reported errors

The Cortex-M55 processor in the RSE detects and reports Dual-Core Lockstep (DCLS) errors using hardware signals. These signals remain low when no error is detected. When an error occurs, they are combined using a logical OR into the `CPUDCLSERR` alarm signal. This signal is connected as an input event to the Security Alarm Manager (SAM). Once raised, the corresponding DCLS signals are captured in the `SAMCDRES` register, which retains its value across a cold reset.

At power-on reset, SAM is fully reset. Therefore, this procedure applies only after a cold reset. After a cold boot, if the reset was caused by SAM, check whether tracking of processor DCLS errors is required. If it is, the RSE software performs the following actions:

1. Check if bit [14] (CPUDCLSERR) in the `SAMES0` register is set, which indicates that a DCLS error occurred.
2. Read the `SAMCDRES` register to obtain the `DCLSErrorValue`. Each bit set in `DCLSErrorValue` corresponds to a detected failure.
3. Clear the reported DCLS error signals by writing the same `DCLSErrorValue` back to the `SAMCDRES` register. This register uses RW1C (Read-Write 1 to Clear) behavior.

### 3.11.1.2 Handling of processor RAS reported errors

The Cortex-M55 processor in the RSE detects and reports Reliability, Availability, and Serviceability (RAS) errors through dedicated signals. These signals remain low when no error is detected. When an error occurs, they are combined into the `CPURASERR0`, `CPURASERR1`, and `CPURASERR2` alarm signals. These signals are connected as input events to the Security Alarm Manager (SAM). Once any of these signals are asserted, the corresponding RAS error values are captured in the `SAMRRES0`, `SAMRRES1`, and `SAMRRES2` registers. These registers persist across a cold reset.

At power-on reset, SAM is fully reset. Therefore, the following applies only after a cold reset. After a cold boot, if the reset was caused by a SAM event, determine whether accounting of processor RAS errors is required. If it is, the RSE software performs the following steps. Apply these steps for each register `x` = [0-2]:

1. Check if `CPURASERRx` (bits [17:15]) in the `SAMES0` register are set, which indicates that a RAS error occurred.
2. Read the corresponding `SAMRRESx` register to obtain the `RASErrorValue`. If `RASErrorValue` is zero, another RAS register signaled the fault.
3. Write the same `RASErrorValue` back to the `SAMRRESx` register to clear it. The `SAMRRESx` registers use RW1C (Read-Write 1 to Clear) behavior.

### 3.11.1.3 Handling of various SAM reported errors

Recommendations for handling SAM error or alarm events include the source, typical cause, and the default response — typically a Runtime Security Engine reset.

External sensor events added by the integrator are connected to SAM inputs [63:48] and logged in the `SAMES1` status register bits [63:48]. Handling of these external events depends on the specific platform implementation.

**Table 3-17: SAM event sources and recommended handling**

Event Index	Event Source	Possible Reason for Event	Default Handling
0	SAM Configuration Integrity Checker	The SAM checks integrity of written configuration. If a mismatch is detected, this event is triggered—possibly due to attack or corruption.	Reset of the RSE
1	SAM Watchdog Timer	If the interrupt service routine does not clean up the event that caused an interrupt in a timely manner, the watchdog timer expires and triggers this event. Causes include unhandled interrupts, too short watchdog, or attack.	Reset of the RSE



Event Index	Event Source	Possible Reason for Event	Default Handling
2	SAM Duplication Error	Parity/duplication checker in SAM detects a mismatch— may be due to attack or corruption of register values.	Reset of the RSE
3	LCM Fatal Error	LCM detects fatal error in OTP, possibly due to attack or corruption. After reset, if not attacked, error may not reoccur.	Reset of the RSE
4	CPU Lockup	CPU lockup, assumed to be due to attempted attack on CPU.	Reset of the RSE
5	ATU Error	Parity error in ATU register or access to disabled address region—may be attack or software bug.	Reset of the RSE
6	KMU Parity Error	Parity error in KMU register, assumed due to attack.	Reset of the RSE
7	Crypto Combined Parity Error	Parity error in PKA RAM or WMF for crypto device, assumed due to attack.	Reset of the RSE
8	SIC Parity Error	Parity error in SIC register or WMF for programming port, assumed due to attack.	Reset of the RSE
9	AES DFA	AES DFA countermeasure detects attack on crypto device.	Reset of the RSE
10	AES Parity Error	AES parity countermeasure detects attack on crypto device.	Reset of the RSE
11	DMA-350 DCLS Error	DMA-350 DCLS detects mismatch, assumed due to attack on DMA device.	Reset of the RSE
12	PSI (AON) Parity Error	Parity error on PSI interface, assumed due to attack.	Reset of the RSE
13	Bus Parity Error	Parity error on RSE interconnect or peripherals, assumed due to attack.	Reset of the RSE
14	Processor DCLS	M55 CPU DCLS detects mismatch, assumed due to attack.	Reset of the RSE
15	Processor RAS events set 0	M55 CPU RAS detects error in core, caches, or TCMs ECC— possible attack. See [Handling of CPU RAS reported errors].	Reset of the RSE
16	Processor RAS events set 1	M55 CPU RAS detects error in core, caches, or TCMs ECC— possible attack. See [Handling of CPU RAS reported errors].	Reset of the RSE
17	Processor RAS events set 2	M55 CPU RAS detects error in core, caches, or TCMs ECC— possible attack. See [Handling of CPU RAS reported errors].	Reset of the RSE
18	SRAM Partial Write	Partial write detected; see [Correcting ECC following Partial Writes to VM0, VM1].	Default action disabled; IRQ interrupts software
19	VM0 Single ECC Error Detected and Corrected	Single ECC error detected/corrected in VM0; see [Correcting ECC following single detected and corrected ECC error in VM0, VM1].	Default action disabled; IRQ interrupts software
20	VM1 Single ECC Error Detected and Corrected	Single ECC error detected/corrected in VM1; see [Correcting ECC following single detected and corrected ECC error in VM0, VM1].	Default action disabled; IRQ interrupts software
21	VM2 Single ECC Error Detected and Corrected	Single ECC error detected/corrected in VM2. VM2 not implemented in this RSE version.	Default action disabled
22	VM3 Single ECC Error Detected and Corrected	Single ECC error detected/corrected in VM3. VM3 not implemented in this RSE version.	Default action disabled
23	VM0 ECC Error	Double uncorrected ECC error in VM0; see [Handling of double un-corrected ECC error in VM0, VM1].	Reset of the RSE
24	VM1 ECC Error	Double uncorrected ECC error in VM1; see [Handling of double un-corrected ECC error in VM0, VM1].	Reset of the RSE

Event Index	Event Source	Possible Reason for Event	Default Handling
25	VM2 ECC Error	ECC error in VM2. VM2 not implemented in this RSE version.	Default action disabled
26	VM3 ECC Error	ECC error in VM3. VM3 not implemented in this RSE version.	Default action disabled
27	SRAM MPC Combined Parity Error	MPC parity error for SRAM. In current RSE, MPCs do not generate alarm.	Reset of the RSE (no software handling needed)
28	SIC MPC Parity Error	MPC parity error for SIC. In current RSE, MPCs do not generate alarm.	Reset of the RSE (no software handling needed)
29	ATU Parity Error	Parity error in ATU register, assumed due to attack.	Reset of the RSE
30	Processor Private Region Parity Error	Parity error in Processor Private Region registers, assumed due to attack.	Reset of the RSE
31	System Control Registers Block Parity Error	Parity error in System Control Registers block, assumed due to attack.	Reset of the RSE
32	CPU PPB Parity Error	Parity error in CPU Private Peripheral Bus region registers, assumed due to attack.	Reset of the RSE
33	Secure Access Configuration Registers Block Parity Error	Parity error in Secure Access Configuration registers, assumed due to attack.	Reset of the RSE
34	Non-Secure Access Configuration Registers Block Parity Error	Parity error in Non-Secure Access Configuration registers, assumed due to attack.	Reset of the RSE
35	Integrity Checker alarm	Integrity checker detects bad value, combined with parity error detection at the Integrity Checker APB and AXI interfaces, assumed due to attack.	Reset of the RSE
36	TRAM alarm	TRAM parity error combined with its WMF parity error, assumed due to attack.	Reset of the RSE

## 3.12 Persistent state interface

The persistent state interface (PSI) is a 128-bit output from the RSE. It provides the SoC with information about the RSE's internal state, enabling other system components to monitor key lifecycle and configuration parameters.

The PSI communicates both static and dynamic data reflecting the RSE's operational context. Examples include lifecycle state, debug configuration, and device identity. This allows the SoC to synchronize its behavior with the RSE and enforce consistent security and functional policies.

The PSI outputs include:

- The SoC lifecycle state
- Debug configuration for components associated with the current lifecycle state
- Enumeration information when multiple RSE instances are present (multi-chip designs)
- Device type: production chip (PCI) or test chip (TCI)
- SoC-specific flavor or SKU information

The PSI updates automatically when the RSE transitions between lifecycle states or when configuration data changes. It provides a reliable interface for secure communication of RSE status to other SoC elements.

### 3.13 Local system counter

The RSE provides two interfaces for the local system counter (LSC): the LSC Control Base and the LSC Read Base. These are defined in the RSE customization layer.

Secure software uses the LSC Control Base to configure the trusted RSE counter. This counter supplies timing input to the four system timers in the RSE base layer. The LSC Read Base allows software to read the current LSC value, similar to reading the system timers.

Access to the LSC Control Base is currently restricted to secure software. It is not available through the Peripheral Protection Controller (PPC), which is typically controlled by the `PERIPHSPPPCEXP0`, `PERIPHSPPPC0`, and `PERIPHNSPPPCEXP0` registers.

Non-secure software must access the LSC Read Base via a non-secure callable (NSC) function implemented by secure software. Alternatively, non-secure software can use the system timers to read the current timer value.

## 4. RSE debug scenarios

Common RSE debug situations span lifecycle states and provisioning stages. They involve System on Chip (SoC) components and debug interfaces. Examples include design for test (DFT), Joint Test Action Group (JTAG), and general-purpose I/O (GPIO). Lifecycle state (LCS) values such as Chip Manufacturing (CM), Device Manufacturing (DM), and Return Merchandise Authorization (RMA) determine availability. Scenarios also reference the Lifecycle Manager (LCM), Security Alarm Manager (SAM), One-Time Programmable (OTP) memory, and Direct Memory Access (DMA). They also reference Bootloader stage 1-2 (BL1-2).

- DFT in an SoC with RSE
  - Conditions: Blank (virgin) RSE or RMA
  - Notes: Access via JTAG, SCAN, or GPIO. DFT and debug remain available until OTP provisioning starts, or after the RSE is moved to RMA. The user can take control of the RSE or monitor DFT-enable signals.
- Debug RSE internals
  - Conditions: LCS = CM / RMA
  - Notes: JTAG is allowed in these states, except during secure provisioning unless TCI mode is enabled. Debug is not available in SE unless the chip operates in TCI mode. The user must take control of the RSE.
- Debug SoC processing elements (PEs) without RSE software on a blank chip
  - Conditions: Blank chip
  - Notes: The LCM enables SoC debug without requiring RSE control. RSE debug is also enabled in this state.
- Provision a blank chip with CM assets and contents
  - Conditions: Post-DFT on blank chip; TP mode set
  - Notes: Program CM assets (KP\_CM, KCE\_CM, ROTPK), the DMA program, and BL1-2. Also program sensor trims and SAM configuration into OTP. See [DMA boot in CM lifecycle state, non-virgin modes (TCI or PCI)] for details.
- Provision a CM-provisioned chip with DM assets
  - Conditions: LCS = DM
  - Notes: Program DM assets (KP\_DM, KCE\_DM) and remaining OTP values. See [DMA Boot in DM lifecycle state](#) for details.

## 5. Software countermeasures

Guidance strengthens RSE firmware security through software countermeasures. Measures complement hardware protections to resist active and passive attacks.

Countermeasures focus on mitigating the following threats:

- Fault injection and glitching attacks
- Memory corruption, including buffer overflows and invalid pointer accesses
- Side-channel leakage and timing attacks
- Data tampering or replay during software execution

Software countermeasures include a combination of coding practices, runtime checks, and algorithmic protections. The following sections describe key strategies and implementation recommendations to enhance RSE firmware resilience.

### 5.1 Fault injection and perturbation

The RSE uses a Dual-Core Lockstep (DCLS) processor architecture to protect processor execution from various types of faults. This design reduces the amount of software countermeasures required to protect the RSE against fault injection and perturbation attacks. However, it does not eliminate the need for additional software-based protection.

Although the processor and DMA-350 modules in the RSE are duplicated for redundancy, other peripherals are not. Therefore, software must apply additional verification and redundancy when interacting with non-duplicated peripherals.

#### Read-after-write and double-read of hardware registers

Faults may occur when reading or programming hardware registers belonging to non-duplicated peripherals. To mitigate this risk, software should verify critical register values using read-after-write checks. This ensures that configuration values have been written correctly. Similarly, when reading such registers, a double-read should be performed to confirm that the first read was not affected by a transient fault or attack.

Adding a random delay between successive register accesses makes it harder for an attacker to inject identical faults during both operations. A short, pseudo-random delay can be generated efficiently by reading the KMU registers `KMURD_8`, `KMURD_16`, and `KMURD_32`.

#### Default failure handling

Although the DCLS processor architecture provides strong protection against execution faults, software must still follow secure coding best practices. This ensures defense in depth. Functions should initialize return values with error codes and update them to indicate success only after the function completes successfully. This approach ensures that unexpected conditions default to failure, maintaining safe and predictable software behavior.

## 5.2 Memory dump prevention

All secrets and keys in the DTCM (protected by TRAM) should be kept in a separate MPU region. This region must be locked for reading and writing when exchanging messages with external systems (for example, via MHU or shared memory). This countermeasure prevents sending secrets to the external world or overriding them by mistake. It also reduces risk from malicious intervention by software or hardware.

There are two memory-protected regions in the DTCM.

- RSE services (secure partition)
- application (non-secure partition).

RSE services must store all secrets, such as cryptographic keys, DRBG seeds, and sensitive states, in the DTCM area designed for RSE services.

## 5.3 Handling detected faults

RSE includes a Security Alarm Manager (SAM) that handles detected attacks. These attacks may be signaled by the Lifecycle Manager, CPU, or DMA-350 DCLS fault detectors. Other sources include the Integrity Checker, ECC of various memories, or platform-specific sensors integrated at the SoC level.

To operate correctly, this module must be configured appropriately.

It is recommended that:

- A response is defined for all allocated SAM events
- Cold reset is triggered in case no software handling is required before resetting RSE
- NMI or interrupt is raised in case some software handling is required before resetting RSE

## 5.4 Limited Fault Tolerance counter

The Runtime Security Engine (RSE) maintains a Limited Fault Tolerance (LFT) counter in one-time programmable (OTP) memory. The LFT counter tracks all detected alarms. This mechanism monitors repeated or persistent hardware or security faults.

When you enable the LFT feature in OTP, RSE BL1 processes each alarm after reset. The process assumes that each alarm triggers a reset and that alarms clear before normal boot continues.

At boot, software reads the SAM Event Status (`SAMES`) registers and compares active events with the disabled alarms defined in the SAM Input Mask (`SAMIM`). If any enabled alarm bit is set, software

increments the LFT counter. The software then clears all active alarms using the SAM Event Clear (SAMECL) registers.

During initialization, RSE BL1 monitors the LFT counter. If the counter reaches its maximum value, RSE halts the boot process and permanently locks the device.

Set the LFT counter size to 1024 bits if only digital alarms exist in the base RSE design. If you integrate environmental sensors, increase the counter capacity to handle the expanded set of alarms.

## 5.5 Side-channel protection

Software-level countermeasures reduce side-channel leakage and complement hardware protections to avoid exposing secret data through timing, power, or electromagnetic analysis.

### Word-aligned access to secrets

All access to secrets, including cryptographic keys and state variables, must occur on word boundaries. Secrets should always be stored in word-aligned buffers to ensure consistent memory access and reduce unintended timing variations. Byte-level access must be avoided to minimize observable differences in memory access behavior.

### Randomized access to secrets

Access to secret data should occur in a randomized order. Arm recommends that software performs randomized read and write operations rather than sequential or word-by-word access. This randomization reduces the risk of data-dependent patterns that can be exploited through side-channel observation.

### Scrubbing and salting initial memory

Memory areas intended to store secrets must be scrubbed and overwritten with random values before use. This applies to both general memory regions and dedicated key registers.

Examples of secure memory scrubbing include the following. Terminology used: Key Management Unit (KMU), Trusted RAM (TRAM), Advanced Encryption Standard (AES), Direct Memory Access (DMA), and data tightly-coupled memory (DTCM).

- Scrubbing target memory before exporting from the KMU to TRAM or to RSE integrated AES engine key registers.
- Overwriting secrets with random values instead of zeros when clearing them.
- Using the DMA to scrub DTCM with random data after enabling TRAM encryption.

These practices help prevent residual data from being recovered after operations involving secret information.

## 5.6 Lightweight random generation

Many software countermeasures depend on randomization, such as randomizing access order or scrubbing data with random values. For these operations, cryptographically strong entropy is not required. Instead, lightweight and fast algorithms that generate random values efficiently without significant hardware overhead should be used.

A typical lightweight random number generator is an LFSR-based implementation, such as XORSHIFT or simple LFSR variants. These algorithms are fast, compact, and suitable for non-cryptographic randomization tasks used in countermeasures.

The LFSR should be seeded once during boot and then extended whenever new random bits are needed. Seeding should be performed using true entropy from dedicated hardware sources when available. If hardware entropy is unavailable when randomization is required, the seed can be generated using static chip-unique data. Combine this with a non-volatile monotonic counter. This combination ensures that every RSE instance produces a unique and unpredictable random sequence at each boot.

## 5.7 Address Translation Unit as a firewall

The Address Translation Unit (ATU) acts as a memory access firewall. If an unintended access reaches the ATU and no matching address region is enabled, the ATU blocks the access and raises an alarm signal.

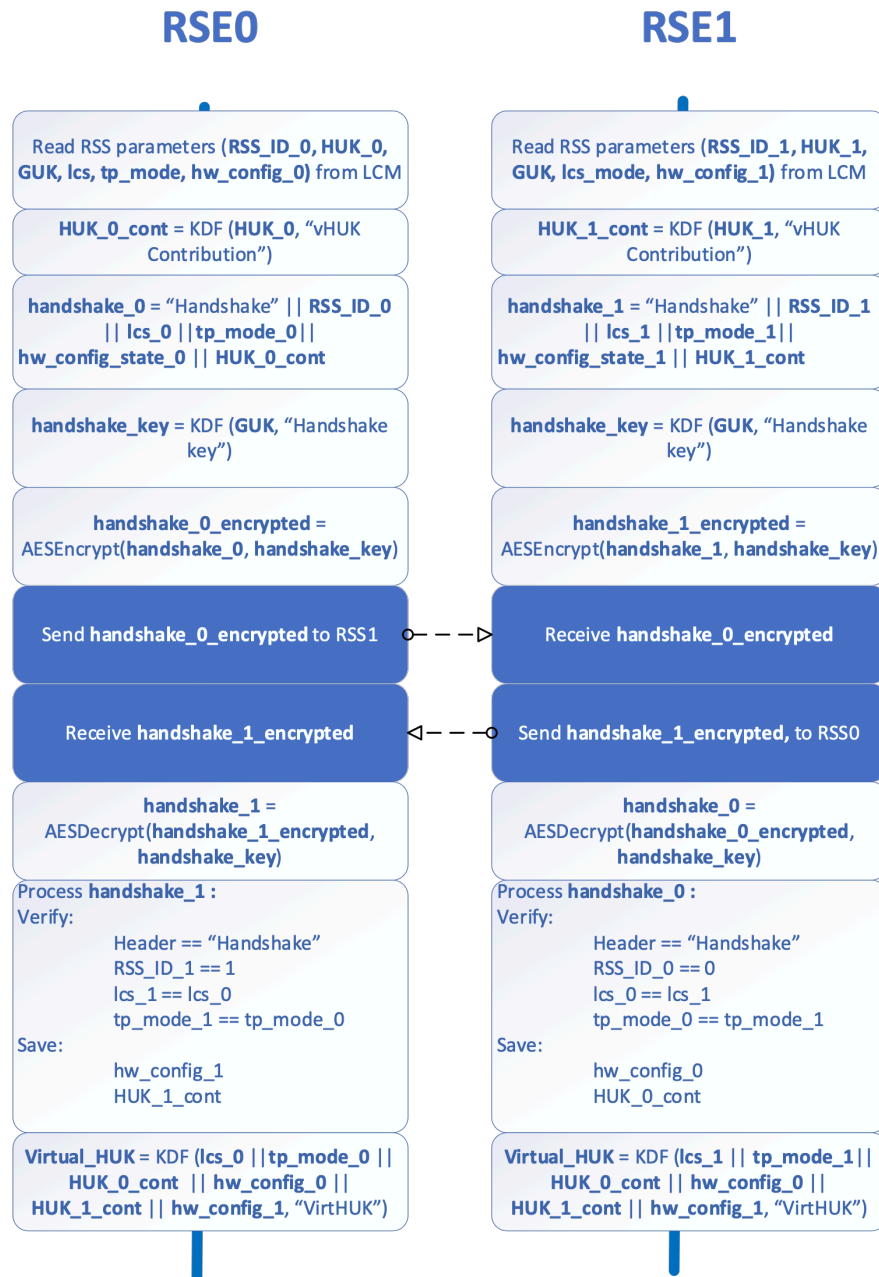
Software should enable ATU regions only when access to external memory or registers is required. After completing the operation, the software should disable the ATU region again to minimize the risk of unauthorized or accidental access.



## 6. RSE Handshake Flow

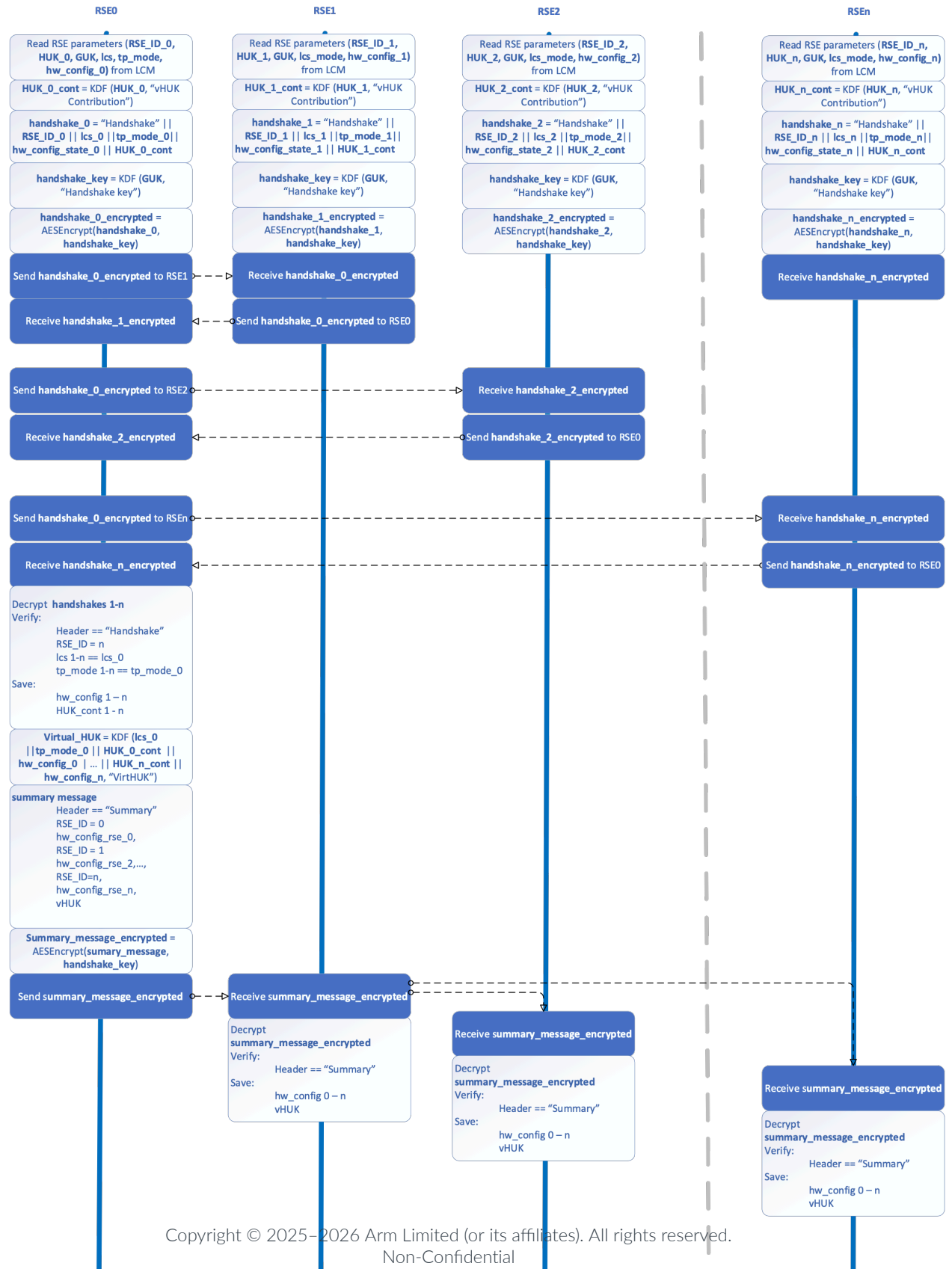
In case of two RSE instances, the flow would be as follows.

**Figure 6-1: Two RSE handshake flow**



In the general case of n RSE instances, the flow is as follows.

Figure 6-2: n RSE handshake flow





# Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

# Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

## Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

### Product completeness status

The information in this document is Final, that is for a developed product.

### Product revision status

The r1p2 identifier indicates the revision status of the product described in this manual, where:

<b>rx</b>	Identifies the major revision of the product.
<b>py</b>	Identifies the minor revision or modification status of the product.

## Revision history

These sections can help you understand how the document has changed over time.

### Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

#### Document history

Issue	Date	Confidentiality	Change
0102-02	2 March 2026	Non-Confidential	Structure changes and editorial improvements
0102-01	31 October 2025	Non-Confidential	Initial Release

### Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in [Document release information](#) on page 102.

Table 2: Issue 01

Change	Location
First release	-

Table 3: Issue 02

Change	Location
No technical changes documented	-
Document restructured	Complete document

## Conventions

The following subsections describe conventions used in Arm documents.

### Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: [developer.arm.com/glossary](https://developer.arm.com/glossary).

### Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
<b>bold</b>	Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments.  For example: <div>MRC p15, 0, &lt;Rd&gt;, &lt;CRn&gt;, &lt;CRm&gt;, &lt;Opcode_2&gt;</div>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, <b>IMPLEMENTATION DEFINED</b> , <b>IMPLEMENTATION SPECIFIC</b> , <b>UNKNOWN</b> , and <b>UNPREDICTABLE</b> .



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.

---



You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

---



This information is important and needs your attention.

---



This information might help you perform a task in an easier, better, or faster way.

---



This information reminds you of something important relating to the current content.

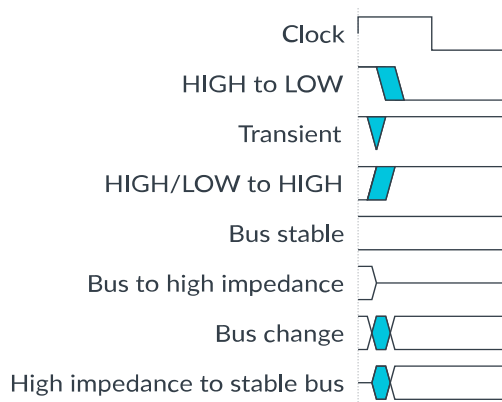
---

## Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



**Figure 1: Key to timing diagram conventions**

## Signals

The signal conventions are:

### Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

### Lowercase n

At the start or end of a signal name, n denotes an active-LOW signal.

# Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on [developer.arm.com/documentation](https://developer.arm.com/documentation).

Confidential documents are only available to licensees, when logged in. Each document link in the following tables provides direct access to the online version of the document.

Arm architecture and specifications	Document ID	Confidentiality
<a href="#">Arm® Address Translation Unit Specification</a>	107714	Non-Confidential
<a href="#">Arm® Key Management Unit Specification</a>	107715	Non-Confidential
<a href="#">Arm® Lifecycle Manager Specification</a>	107616	Non-Confidential

Non-Arm resources	Document ID	Organization
<a href="#">Trusted Computing Group (TCG) Device Identifier Composition Engine (DICE) Protection Environment (DPE)</a>	–	TCG
<a href="#">Trusted Firmware-M RSE Platform Documentation</a>	–	Trusted Firmware