Arm® TrustZone® CryptoCell-312

Revision: r0p0

Software Integrators Manual

Confidential

arm

100776_0000_02_en

Release Information

Software Integrators Manual

Release information

Document History

Issue	Date	Confidentiality	Change
0000-00	20 July 2017	Confidential	First official release for boot r0p0
0000-01	19 October 2017	Confidential	First release for boot EAC and runtime DEV r0p0.
0000-02	19 February 2018	Confidential	First release for runtime EAC r0p0.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product status

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

Web address

http://www.arm.com

Contents

Arm® TrustZone® CryptoCell-312 Software Integrators Manual

List of Figures
Arm® TrustZone® CryptoCell-312 Software Integrators Manual

List of Tables
Arm® TrustZone® CryptoCell-312 Software Integrators Manual

Preface

This preface introduces the *Arm® TrustZone® CryptoCell-312 Software Integrators Manual*.

It contains the following:

About this book

This book provides an overview of the Arm® TrustZone® CryptoCell-312 (CryptoCell-312) software product, its features and capabilities, as well as integration guidelines.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is written for system designers, system integrators, and programmers who are designing or programming a System-on-Chip (SoC) that uses Arm® TrustZone® CryptoCell-312 .

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an overview of the CryptoCell-312 software.

Chapter 2 Product deliverables
This chapter lists CryptoCell-312 product deliverables and describes its build environment.

Chapter 3 Product features
This chapter describes CryptoCell-312 product features.

Chapter 4 Integrating CryptoCell-312 software
This chapter details the steps required for the integration of CryptoCell-312 software into a SoC.

Chapter 5 Offline tools
This chapter describes CryptoCell-312 PC (offline) tools.

Chapter 6 CryptoCell-312 production-line tools
This chapter describes CryptoCell-312 production-line (device-run) tools.

Appendix A Random key generation
This appendix details how to generate a random key, and provides output examples.

Appendix B CryptoCell-312 supported algorithms
This appendix details algorithms supported by CryptoCell-312.

Appendix C Integration test
This appendix describes the CryptoCell-312 integration test.

Appendix D Revisions
This appendix describes the technical changes between released issues of this book.

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.

Table 1 Terms and abbreviations

Term	Abbreviation
Host	SoC processor
Kcp	Constant Provisioning Key
HUK	Device Root Key
LCS	Life-Cycle State
NVM-FSM	NVM-Manager state-machine
NVM-MGR	NVM-Manager module
PKA	Public Key Accelerator
RKEK	Root Key Encryption Key

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

Arm publications

- *Arm® AMBA® AXI and ACE Protocol Specification* (Arm IHI0022E).
- *Arm® AMBA® APB Protocol Specification* (Arm IHI0024C).
- *Power State Coordination Interface (PSCI) System Software on Arm® systems* (Arm DEN0022C).

The following confidential books are available to licensees:

- *Arm® TrustZone® CryptoCell-312 Technical Reference Manual* (Arm 100774).
- *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual* (Arm 100775).
- *Arm® TrustZone® CryptoCell-312 Software Developers Manual* (Arm 100777).
- *Arm® TrustZone® CryptoCell-312 Runtime Software Release Note* (ARM-EPM-129117).
- *Arm® TrustZone® TRNG Characterization Application Note* (Arm 100685).

Other publications

- *ANSI X3.92-1981: Data Encryption Algorithm*
- *ANSI X3.106-1983: Data Encryption Algorithm – Modes of Operation*
- *ANSI X9.31-1988: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA)*
- *ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*
- *ANSI X9.52-1998: Triple Data Encryption Algorithm Modes of Operation*
- *X9.62-2005: Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*
- *X9.63-2011: Public Key Cryptography for the Financial Services Industry – Key Agreement and Key Transport Using Elliptic Curve Cryptography*
- *BSI AIS-31: Functionality Classes and Evaluation Methodology for True Random Number Generators*
- *FIPS Publication 46-3: Data Encryption Standard (DES)*
- *FIPS Publication 81: DES Modes of Operation*
- *FIPS Publication 140-2: Security Requirements for Cryptographic Modules*
- *FIPS Publication 180-4: Secure Hash Standard (SHS)*
- *FIPS Publication 186-4: Digital Signature Standard (DSS)*
- *FIPS Publication 197: Advanced Encryption Standard*
- *FIPS Publication 198-1: The Keyed-Hash Message Authentication Code (HMAC)*
- *ISO/IEC 9797-1: Message Authentication Codes (MACs) -- Part 1: Mechanisms using a block cipher*
- *ISO/IEC 18033-2:2006: Information technology -- Security techniques -- Encryption algorithms -- Part 2: Asymmetric ciphers*
- *IEEE 802.15.4: IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*
- *IEEE 1363-2000: IEEE Standard for Standard Specifications for Public-Key Cryptography*
- *NIST SP 800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*
- *NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*
- *NIST SP 800-38A Addendum: Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*
- *NIST SP 800-38B: Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*
- *NIST SP 800-38C: Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*
- *NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*
- *NIST SP 800-38E: Recommendation for Block Cipher Modes of Operation: the XTSAES Mode for Confidentiality on Storage Devices*
- *NIST SP 800-57 Rev. 4: Recommendation for Key Management – Part 1: General*
- *NIST SP 800-56A Rev. 2: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*
- *NIST SP 800-67 Rev. 1: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*
- *NIST SP 90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators – App C.*
- *NIST SP 90B: DRAFT Recommendation for the Entropy Sources Used for Random Bit Generation*
- *NIST SP 90C: Recommendation for Random Bit Generator (RBG) Constructions*
- *NIST SP 108: Recommendation for Key Derivation Using Pseudorandom Functions*
- *NIST SP 135 Rev. 1: Recommendation for Existing Application-Specific Key Derivation Functions*
- *Public-Key Cryptography Standards (PKCS) #1 v1.5: RSA Encryption*

- *Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications*
- *Public-Key Cryptography Standards (PKCS) #3: Diffie Hellman Key Agreement Standard*
- *Public-Key Cryptography Standards (PKCS) #7 v1: Cryptographic Message Syntax Standard*
- *RFC 2104: HMAC: Keyed-Hashing for Message Authentication*
- *RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm*
- *RFC 3566: The AES-XCBC-MAC-96 Algorithm and Its Use with IPsec*
- *RFC 3686: Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)*
- *RFC 4106: The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)*
- *RFC 4309: Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)*
- *RFC 4543: The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH*
- *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*
- *RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*
- *SEC 2 v1: Recommended Elliptic Curve Domain Parameters*
- *SEC 2 v2: Recommended Elliptic Curve Domain Parameters*
- *Standards for Efficient Cryptography Group (SECG): SEC1 Elliptic Curve Cryptography*
- *Universal Flash Storage Host Controller Interface (UFSHCI), Version 2.1 JESD223C*

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm TrustZone CryptoCell-312 Software Integrators Manual*.
- The number 100776_0000_02_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter provides an overview of the CryptoCell-312 software.

It contains the following sections:

1.1 Overview of CryptoCell-312

CryptoCell-312 is a comprehensive security solution for low-power, low-area designs. It provides foundational security services for the platform, for example, Secure Boot and key management, as well as accelerating cryptographic operations requested by trusted platform components.

CryptoCell-312 is aimed at the IoT market, and other areas where there is potential usage of an Arm Cortex®-M processor, and is intended for use in trusted environments, for example, in Arm systems with TrustZone technology.

In most common configurations, the foundational security services of CryptoCell-312 are integrated into the ROM of the platform. Cryptography and management services are integrated into a trusted or secure OS.

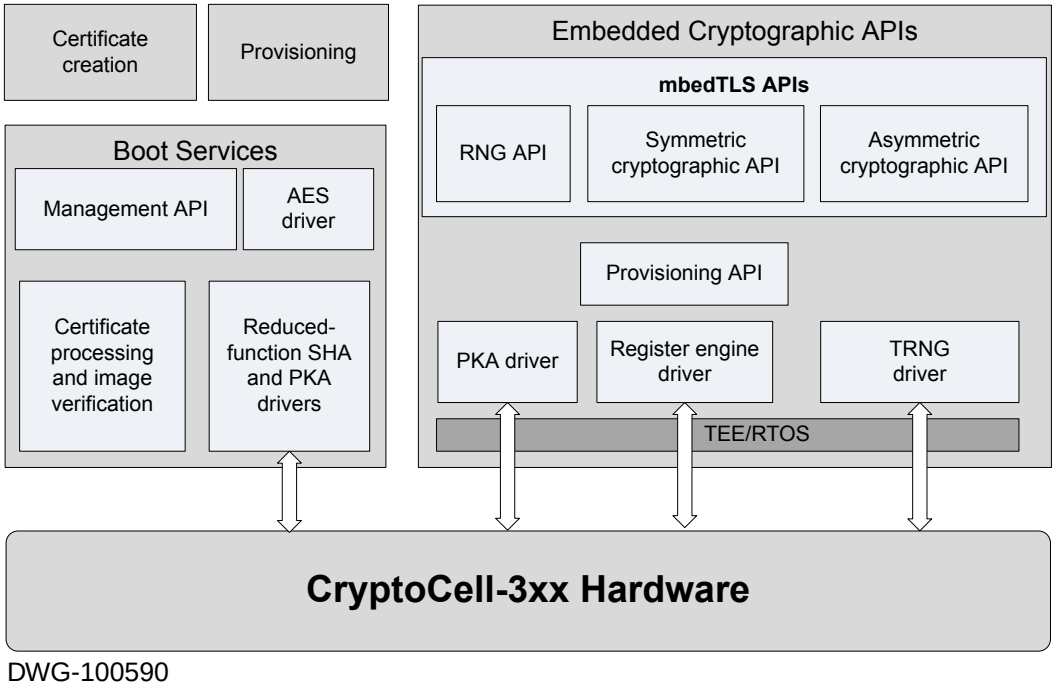

Figure 1-1 CryptoCell-312 software block diagram

1.2 Compliance

CryptoCell-312 complies with the following specifications:

- *ANSI X9.31-1988: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA)*, compliant excluding C.9.
- *ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*, compliant with sections 7.1, 7.2, 7.3, 7.4, 7.5.1, 7.7.1, 7.7.2, 8.1.1, 8.1.2, 8.1.3, 8.1.4 and Annex B.
- *X9.62-2005: Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*, compliant sections 7.2, 7.3, and 7.4.1 – prime curves.
- *X9.63-2011: Public Key Cryptography for the Financial Services Industry – Key Agreement and Key Transport Using Elliptic Curve Cryptography*, compliant with sections 5.2, 5.3, 5.4.1, 5.6.2, 5.6.3, 5.7, 5.9, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 (EC over FP).
- *BSI AIS-31: Functionality Classes and Evaluation Methodology for True Random Number Generators*, compliant in an implementation using FETRNG driver.
- ChaCha, a variant of Salsa20
- Curve25519: New Diffie-Hellman Speed Records
- Ed25519: High-Speed High-Security Signatures
- *FIPS Publication 180-4: Secure Hash Standard (SHS)*, compliant excluding support for truncated hash operation.
- *FIPS Publication 186-4: Digital Signature Standard (DSS)*, compliant with sections 5.1, 6.2, 6.3, 6.4, B.1.2, B.2.2, B.3.6, B.4.2, C.3.1, C.3.3, C.3.5, C.9, and D.1.2.
- *FIPS Publication 197: Advanced Encryption Standard*, support only 128-bit and 256-bit keys.
- *FIPS Publication 198-1: The Keyed-Hash Message Authentication Code (HMAC)*
- *ISO/IEC 9797-1: Message Authentication Codes (MACs) -- Part 1: Mechanisms using a block cipher*, compliant with CBC-MAC without padding, output transformation based on sections 6.2, 6.3.1, 6.4, 6.5.1, and 7.1.
- *ISO/IEC 18033-2:2006: Information technology -- Security techniques -- Encryption algorithms -- Part 2: Asymmetric ciphers*, compliant with sections 10.2, 10.2.1, 10.2.3 and 10.2.4.
- *IEEE 802.15.4: IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, compliant with CCM* (section 7 and Annex B).
- *IEEE 1363-2000: IEEE Standard for Standard Specifications for Public-Key Cryptography*, compliant with sections 7.2.1, 8 (excluding 8.2.6, 8.2.7, 8.2.8, 8.2.9), 10.3, 11, 12.2, 13 (excluding RIPEMD-160) and 14 (excluding RIPEMD-160).
- *NIST SP 800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, the second phase in the CryptoCell-312 TRNG characterization process is compliant with this.
- *NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, compliant with sections 6.1, 6.2, 6.4, and 6.5.
- *NIST SP 800-38B: Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*
- *NIST SP 800-38C: Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*
- *NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*
- *NIST SP 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*, compliant with section 6.
- *NIST SP 800-56A Rev. 2: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, compliant with sections 5.1, 5.2, 5.3, 5.4, 5.5.1.1, 5.6.1, 5.6.2.3, 5.7.1.1, 5.7.1.2 and 5.8.2.
- *NIST SP 90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators – App C.*, compliant with section 10.2 – DRBG mechanism based on block ciphers.
- *NIST SP 90B: DRAFT Recommendation for the Entropy Sources Used for Random Bit Generation*, compliant with section 4.4 tests in runtime SW.

- *NIST SP 90C: Recommendation for Random Bit Generator (RBG) Constructions*
- *NIST SP 108: Recommendation for Key Derivation Using Pseudorandom Functions*, compliant with section 5.1.
- *NIST SP 135 Rev. 1: Recommendation for Existing Application-Specific Key Derivation Functions*
- The Poly1305-AES message-authentication code
- *Public-Key Cryptography Standards (PKCS) #1 v1.5: RSA Encryption*, compliant with backwards compatibility required by PKCS#1 Version 2.1.
- *Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications*, compliant excluding ASN.1 syntax.
- *Public-Key Cryptography Standards (PKCS) #3: Diffie Hellman Key Agreement Standard*
- *Public-Key Cryptography Standards (PKCS) #7 v1: Cryptographic Message Syntax Standard*, compliant with section 10.3 – padding scheme.
- *RFC 2104: HMAC: Keyed-Hashing for Message Authentication*, compliant with SHA1.
- *RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm*
- *RFC 3566: The AES-XCBC-MAC-96 Algorithm and Its Use with IPsec*, compliant excluding support for truncation to 96-bits.
- *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, compliant with section 4 – Secure Boot and Secure Debug certificates.
- *RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*
- *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols*
- *SEC 2 v1: Recommended Elliptic Curve Domain Parameters*, compliant with section 2 160* domains. Smaller domains are not supported.
- *SEC 2 v2: Recommended Elliptic Curve Domain Parameters*, compliant with section 2.
- *Standards for Efficient Cryptography Group (SECG): SEC1 Elliptic Curve Cryptography*, compliant with sections 2.1.1, 2.2.1, 3.1.1, 3.2, 3.3.1, 3.6.1, 4, and 6.1.
- SRP: The Secure Remote Password Protocol.

Chapter 2
Product deliverables

This chapter lists CryptoCell-312 product deliverables and describes its build environment.

It contains the following sections:

2.1 Product components

The following subsections describe the components supplied with the software release package.

This section contains the following subsections:

2.1.1 Secure Boot services software package

The Secure Boot services software provides Secure Boot and other secure services that must reside in ROM.

It must be integrated with the boot ROM of the Host processor, according to the guidelines provided in *4.1 Integration guidelines* on page 4-53.

The Secure Boot services package supports two certificate flavors. The flavor must be defined, based on what is integrated inside the system, by changing the `CC_CONFIG_SB_X509_CERT_SUPPORTED` configuration flag:

X.509 certificate:

`CC_CONFIG_SB_X509_CERT_SUPPORTED = 1`.

Proprietary certificate:

`CC_CONFIG_SB_X509_CERT_SUPPORTED = 0`.

Note

You must integrate the Secure Boot services into the boot ROM of the device before usage. For more information, see *4.2.2 ROM customization* on page 4-59.

The Secure Boot services software is accompanied by offline tools for facilitating Secure Boot, Secure Debug and ICV provisioning. For more information, see *5.3 Secure Debug PC tools* on page 5-70 and *5.4 Secure asset-provisioning PC tool* on page 5-73 respectively.

2.1.2 Secure Boot services integration test

The Secure Boot services integration test package verifies that the Secure Boot services were integrated correctly.

For more information, see *Appendix C Integration test* on page Appx-C-89.

2.1.3 Runtime software package

The runtime software provides cryptographic acceleration functionality, and support for other secure services.

The library supports Mbed™ TLS APIs. For more information, see *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

Note

The software library may require adaptation to your platform before usage. For more information, see *4.2.3 Runtime software customization* on page 4-60.

2.1.4 Runtime integration test

The runtime integration test package for verifying that the runtime library was integrated correctly.

For more information, see *Appendix C Integration test* on page Appx-C-89.

2.1.5 Production-line tools

The CryptoCell-312 production-line tools are standalone tools that are loaded to the device during the manufacturing process.

ICV factory tool:

The ICV factory tool is used to program the ICV keys, as well as information in the OTP memory of the device.

For more information on the ICV factory tool, see *6.2 ICV factory tool* on page 6-80.

For more information on OTP, see *3.2 OTP memory* on page 3-29.

OEM factory tool:

The OEM factory tool is used to program the OEM assets to the OTP memory of the device.

For more information on the OEM factory tool, see *6.3 OEM factory tool* on page 6-82.

2.2 Project tree

Each package is composed of several of the following components (directories), as applicable:

Table 2-1 Project tree structure

Path	Description
`host/src`	Host-execution domain source code. This is the primary directory for building the release.
`codesafe`	Source code of modules that participate in multiple environment builds.
`shared`	Header files of inter-domain APIs and Source code that can be used on different domains simultaneously.
`utils`	Scripts and tools to be used on development or production PC. For example, image signing tool.

The Host source code also contains the following for each product component:

- Runtime software: `host/src/cc3x_lib`: The CryptoCell-312 runtime library source code.
- Secure Boot services: `sbromlib`: The Host domain build environment for the Secure Boot and ROM API library. The source code itself is located in `codesafe/src/secure_boot_debug`, `host/src/cc3x_sbromlib`.
- ICV factory tool and OEM factory tool: `host/src/cc3x_productionlib`.

2.3 Host code build environment

CryptoCell-312 SW supports several compile-time configurations. These configurations are separated into Mbed TLS configuration and CryptoCell-312 configuration.

The configurable parameters of CryptoCell-312 are controlled using flags that reside in the `proj.ext.cfg` file, located in the root folder.

The available configuration flags are:

Secure Boot services and runtime-SW services:

`CC_CONFIG_SB_X509_CERT_SUPPORTED`. For more information, see *3.4.1 Certificate flavors* on page 3-37.

Runtime SW and ICV factory tool:

- `CC_CONFIG_TRNG_MODE`. For more information, see *TRNG characterization* on page 4-60.
- `CC_CONFIG_SUPPORT_SRP`. If this flag is set to zero, the SRP code is not compiled as part of the CryptoCell-312 library.
- `CC_CONFIG_CC_CHACHA_POLY_SUPPORT`. If this flag is set to zero, the ChaCha and POLY algorithms are not compiled as part of the CryptoCell-312 library.
- `CC_CONFIG_SUPPORT_EXT_DMA`. If this flag is set to zero, the external DMA feature is not compiled as part of the CryptoCell-312 library.
- `CC_CONFIG_SUPPORT_SB_RT`. If this flag is set to zero, the runtime Secure Boot feature is not compiled as part of the CryptoCell-312 library.
- `USE_MBEDTLS_CRYPTOCELL`. This flag must be set to one. It defines that the CryptoCell-312 SW library is compiled with Mbed TLS.

For more information about the Mbed TLS configuration file, see *3.8.2 Controlling cryptographic features* on page 3-49.

The following makefile commands are available:

Build host components:

Run `make` from the `host/src`. This recursively builds all host elements and copies them into the `host/lib`, `host/include`, and `host/bin` output directories.

Clean old builds:

Run `make clean`.

Install the output binaries:

Run `make target_install`. This installs the output binaries into the root file system of the development target, as defined by the *`TARGET_ROOT`* environment variable, assuming the target root file system is mounted on the build machine.

A specific element, for example a test, can also be built by executing `make` inside the element directory.

Note

If the element depends on a specific library, that library must be built and exported to `host/lib`, and its `.h` files exported to `host/include`, before building that dependent executable.

Note

CryptoCell-312 both exposes and uses Mbed TLS APIs. This creates a circular dependency.

The build environment depends on the following environment variables, effective in the current shell session:

Table 2-2 Build environment variables

Environment variable	Description
ARCH	The architecture of the target Host processor. The following architectures are supported: `arm`.
OS	The target OS. The following OSes are supported: `linux`, `no_os` and `freertos`. You must define and implement the relevant OS. The *OS* variable is defined in `proj.cfg` files, under `host/` and `utils/src`, as `TEE_OS`. You must adapt it to your OS.
CROSS_COMPILE	Defines the cross-compiler toolchain prefix used to compile CryptoCell-312 SW.
TARGET_ROOT	The path to the directory within the root file system that the target mounts. For example, `work/nfsroot`. This is required for the *target_install* option of the Makefile.

2.4 Build environment offline tools

The offline tools generate certificates for Secure Boot and Secure Debug asset provisioning packages. These packages are used to provision assets to the device and as part of the OEM factory tool flow.

The offline tools are Python scripts built by executing `make` from the `utils/src` directory, on a Host Linux machine, and tested on Ubuntu. Python can be downloaded from: *http://www.python.org/download*.

All tools use OpenSSL for cryptographic operations. OpenSSL can be downloaded from: *http://www.openssl.org*.

For the latest supported versions of the development tools listed in this section, see *Arm® TrustZone® CryptoCell-312 Runtime Software Release Note*.

For more information on offline tools, see *5.1 Overview of offline tools* on page 5-65.

Chapter 3
Product features

This chapter describes CryptoCell-312 product features.

It contains the following sections:

3.1 Embedded secret keys

CryptoCell-312 secret keys are divided into the several types, described in the following subsections:

Device key:
A secret value that is burned in OTP memory and used for the derivation of other keys.

Platform key:
An ICV key placed in the RTL and used for provisioning during CM LCS and DM LCS.

Class keys:
CryptoCell-312 supports two sets of class keys: ICV and OEM.

Secret keys are are burned using the either the ICV factory tool or OEM factory tool.

Note

In RMA LCS, all keys saved in the OTP are burned to ones, and not loaded to the AES HW registers. Instead, a random value is set.

This section contains the following subsections:

3.1.1 Device key

The device Root Key (*HUK*) is a secret value that is burned in OTP memory.

The *HUK* is used for the derivation of other keys.

This key is read by the HW as part of the preliminary boot sequence, and is thereafter no longer accessible for reading. It is readable only to the ICV, and only while in CM LCS. It must be unique per device. It can be used only by the AES engine.

Important

For security considerations, it cannot be used for data encryption or decryption.

Note

An *HUK* value of all-ones is invalid.

In CM LCS and DM LCS, *HUK* can be overridden for debug purposes, by writing to the shadow registers.

3.1.2 Platform key

The platform key (*Krtl*) is an ICV key placed in the RTL.

This key is used for provisioning during CM LCS and DM LCS, and is not valid in Secure LCS.

It can only be used during boot, and is locked at runtime. It can be used only by the AES engine.

The Boot services allow the user to derive a key from the platform key, for future use.

3.1.3 Class keys

CryptoCell-312 supports two sets of class keys: ICV and OEM.

Note

If ICV and OEM are the same entity, you must decide which set to use.

ICV keys:

Embedded in the OTP and usable only by the ICV.

Note

The ICV class keys can be locked during boot, preventing further usage.

ICV Provisioning Master Key (*Kpicv*):

Kpicv is a 128-bit AES key that derives a model provisioning key, and is readable by the Host, only in CM LCS.

ICV Code Encryption Key (*Kceicv*):

Kceicv is a 128-bit AES key used to decrypt SW images as part of the Secure Boot process. This key is readable by the Host, only in CM LCS.

Note

In CM LCS and DM LCS, *Kpicv* and *Kceicv* can be overridden for debug purposes by writing to the shadow registers.

OEM keys:

Embedded in the OTP, and usable only by the OEM.

OEM Provisioning Master Key (*Kcp*):

Kcp is a 128-bit AES key that derives a model provisioning key. This key is readable by the Host, only in CM LCS or DM LCS.

OEM Code Encryption Key (*Kce*):

Kce is a 128-bit AES key used to decrypt SW images as part of the Secure Boot process. This key is readable by the Host, only in CM LCS or DM LCS.

Note

In CM LCS and DM LCS, *Kcp* and *Kce* can be overridden for debug purposes by writing to the shadow registers.

3.2 OTP memory

The OTP memory stores permanent CryptoCell-312 configuration data and keys.

This memory must be connected to the CryptoCell-312 AIB bus. Its size must be sufficient to support the mandatory configuration fields, burned in the following OTP memory offsets:

Note

The byte order of each OTP memory 32-bit word is little-endian.

Table 3-1 OTP memory bank layout

32-bit word addressing	Description	Access permissions
`0x00-0x07`	Device key, hardware unique, per-chip, key (*HUK*). For more information, see *3.1.1 Device key* on page 3-27. **Note** Values of all-zeroes or all-ones are illegal and results in *HUK* error.	Readable only in CM LCS.Writable only in CM LCS or RMA LCS.
`0x08-0x0B`	ICV provisioning master key (*Kpicv*). For more information, see ICV keys in *3.1.3 Class keys* on page 3-27.	Readable only in CM LCS. Writable only in CM LCS or RMA LCS.
`0x0C-0x0F`	ICV code encryption key (*Kceicv*). For more information, see ICV keys in *3.1.3 Class keys* on page 3-27. **Note** Referenced as "SSK" in *Arm® Trusted Base System Architecture V1: System Software on Arm*.	Readable only in CM LCS. Writable only in CM LCS or RMA LCS.
`0x10`	ICV-programmed flags.b	For read permissions, see *Table 3-2 ICV programmed flags* on page 3-31. Writable only in CM LCS or RMA LCS.
`0x11-0x18`	Root-of-Trust Public Key. May be used in one of the following configurations: • A single 256-bit SHA256 digest of the Secure Boot public key (*Hbk*). • Two 128-bit truncated SHA256 digests of Secure Boot public keys 0 and 1: — `0x11-0x14`: *Hbk0*, ICV key. — `0x15-0x18`: *Hbk1*, OEM key. **Note** • Referenced as "ROTPK" in *Arm® Trusted Base System Architecture V1: System Software on Arm*. • Splitting the *Hbk* field to *Hbk0* and *Hbk1* is not compliant with *Arm® Trusted Board Boot Requirements CLIENT*. • If *Hbk0* is unused, a flag in the ICV flags is set to one.	Readable in all LCSes. Writable in CM LCS or DM LCS.

Table 3-1 OTP memory bank layout (continued)

32-bit word addressing	Description	Access permissions
`0x19-0x1C`	OEM provisioning master key (*Kcp*). For more information, see OEM keys in *3.1.3 Class keys* on page 3-27.	Readable in CM LCS or DM LCS. Writable in DM LCS or RMA LCS.
`0x1D-0x20`	OEM code encryption key (*Kce*). For more information, see OEM keys in *3.1.3 Class keys* on page 3-27. **Note** Referenced as "SSK" in *Arm® Trusted Board Boot Requirements CLIENT*.	Readable in CM LCS or DM LCS. Writable in DM LCS or RMA LCS.
`0x21`	OEM-programmed flags.	For read permissions, see *Table 3-3 OEM programmed flags* on page 3-31. Writable in all security LCSes.
`0x22-0x23`	*Hbk0* Trusted software minimum version anti-rollback counter: Encodes a value between 0 and 63. The version is the number of bits set in the OTP memory word. In case of a single *Hbk* configuration, this field is merged with the `0x24-0x26` field, to construct a single 160-bit NV counter. **Note** • This is the monotonic counter referenced in *Arm® Trusted Base System Architecture V1: System Software on Arm* - requirement R020_TBSA_NV_COUNTER. • This field may optionally be located in the RPMB partition of the flash memory, if exists.	Readable/writeable in all security LCSes.
`0x24-0x26`	*Hbk1* Trusted software minimum version anti-rollback counter: Encodes a value between 0 and 95. The version is the number of bits set in the OTP memory word. In case of a single *Hbk* configuration, this field is merged with the `0x22-0x23` field, to construct a single 160-bit NV counter. **Note** • This is the monotonic counter referenced in *Arm® Trusted Base System Architecture V1: System Software on Arm* - requirement R030_TBSA_NV_COUNTER. It is not fully compliant (smaller range) due to OTP restrictions. • This field may optionally be located in the RPMB partition of the flash memory, if exists.	Readable/writeable in all security LCSes.
`0x27`	General purpose configuration flags.	Readable in all security LCSes. Writeable only in CM LCS.

Table 3-1 OTP memory bank layout (continued)

32-bit word addressing	Description	Access permissions
0x28-0x2B	An OTP DCU 128-bit lock mask that allows the SW to lock the required debug bits with or without a Secure Debug certificate.	Readable in all security LCSes.Writeable in CM LCS or DM LCS.
0x2C-0x7FF	Code and data sections that a user may use for dedicated code or data. **Note** This area is mapped to address 0x00 in the second APB slave.	Readable in all security LCSes. Writeable as defined by HW. For more information, see the *OTP memory layout* appendix in *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual*.

Table 3-2 ICV programmed flags

Bits	Usage	Read access	Write access
[7:0]	Number of zero bits in *HUK*.	Masked for read in any LCS other than CM LCS.	Writeable in CM LCS and RMA LCS.
[14:8]	Number of zero bits in *Kpicv* (128-bit).	Readable only in CM LCS.	Writeable in CM LCS and RMA LCS.
[15]	*Kpicv* "not in use" bit. If *Kpicv* is not in use, this bit is set by the ICV factory tool.	Readable in all security LCSes.	Writeable in CM LCS and RMA LCS.
[22:16]	Number of zero bits in *Kceicv*.	Readable only in CM LCS.	Writeable in CM LCS and RMA LCS.
[23]	*Kceicv* "not in use" bit. If *Kceicv* is not in use, this bit should be set by the ICV factory tool.	Readable in all security LCSes.	Writeable in CM LCS and RMA LCS.
[30:24]	Number of zero bits in *Hbk0*.	Readable in all security LCSes.	Writeable in CM LCS and RMA LCS.
[31]	*Hbk0* "not in use" bit. If *Hbk0* is not in use, this bit should be set by the ICV factory tool.	Readable in all security LCSes.	Writeable in CM LCS and RMA LCS.

Table 3-3 OEM programmed flags

Bits	Usage	Read access	Write access
[7:0]	Number of zero bits in *Hbk1* or *Hbk*.	Readable in all security LCSes.	Writeable in DM LCS and RMA LCS.
[14:8]	Number of zero bits in *Kcp* (128-bit).	Readable only in CM LCS and DM LCS.	Writeable in DM LCS and RMA LCS.
[15]	*Kcp* "not in use" bit. If *Kcp* is not in use, this bit should be set by the OEM factory tool.	Readable in all security LCSes.	Writeable in DM LCS and RMA LCS.
[22:16]	Number of zero bits in *Kce*.	Readable only in CM LCS and DM LCS.	Writeable in DM LCS and RMA LCS.
[23:23]	*Kce* "not in use" bit. If *Kce* is not in use, this bit should be set by the OEM factory tool.	Readable in all security LCSes.	Writeable in DM LCS and RMA LCS.
[29:24]	Reserved.	Always readable.	Always writeable.

Table 3-3 OEM programmed flags (continued)

Bits	Usage	Read access	Write access
[30]	OEM RMA LCS flag.	Readable in all security LCSes.	Writeable in CM LCS, DM LCS and Secure LCS.
[31]	ICV RMA LCS flag.	Readable in all security LCSes.	Writeable in CM LCS, DM LCS and Secure LCS, only if the ICV RMA locking bit in the AO module is not set.

Table 3-4 General purpose flags

Bits	Usage	Read access	Write access
[7:0]	8 bits of general purpose flags. The ICV is responsible for setting the configuration of each bit.	Readable in all security LCSes.	Writeable in all security LCSes.
[31:8]	Reserved.	Always readable.	Always writeable.

3.3 Security life-cycle states

CryptoCell-312 includes a mechanism for managing the security life-cycle state (LCS) of the device.

This mechanism enables the device (both hardware and software) to behave differently in different stages (LCSes) of the device, protecting any security assets once they have been introduced into the device and reducing the risk of IP theft and reverse engineering.

The following figure illustrates possible LCS transitions:

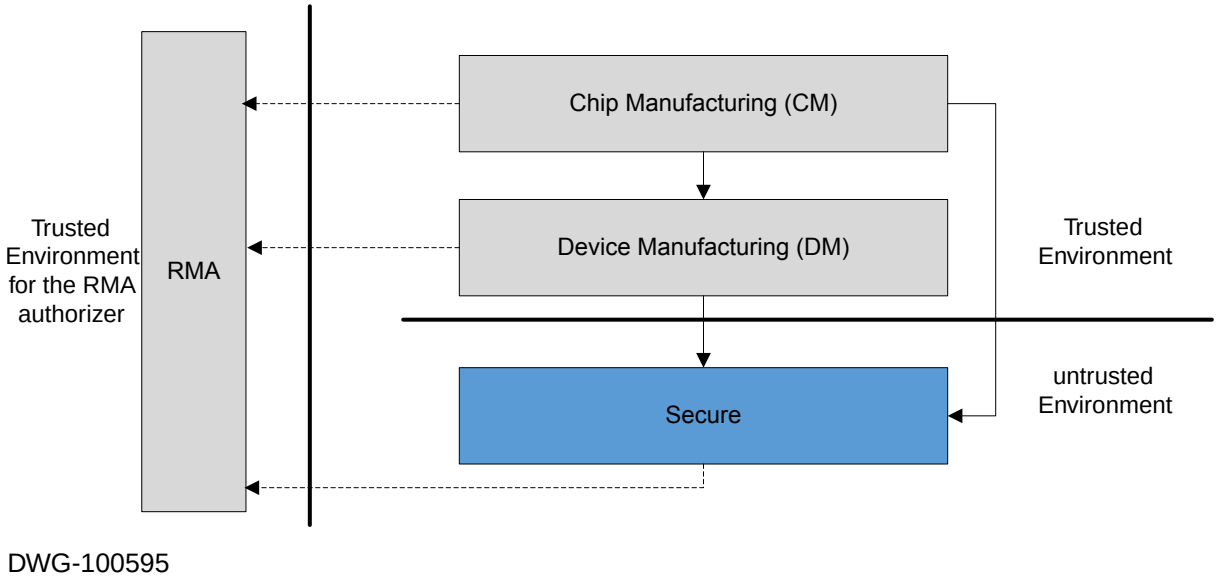

Figure 3-1 Security life-cycle states

The supported LCSes are detailed in the following subsections.

This section contains the following subsections:

- *3.3.1 Chip Manufacturing LCS* on page 3-33.
- *3.3.2 Device Manufacturing LCS* on page 3-34.
- *3.3.3 Secure LCS* on page 3-34.
- *3.3.4 RMA LCS* on page 3-34.

3.3.1 Chip Manufacturing LCS

The Chip Manufacturing (CM) LCS indicates that the chip has not been assembled into a device yet, that is, has not left the production line of the ICV.

In this LCS, all chip debugging and external testing capabilities are enabled, including those that enable access to security-sensitive information. For example, chip test modes and validation functionality.

LCS is CM if the following are true:

- Manufacturer-programmed flags, OTP word `0x10` = zero.
- OEM-programmed flags, OTP word `0x21` = zero.

The *HUK* and its associated Manufacturer-programmed flags, bits [7:0] of OTP word `0x10`, the number of *HUK* zero bits, are programmed in this LCS.

In this lifecycle, all ICV secrets are burned to the OTP.

To transition to DM LCS, the OEM/ICV must program the `0x10` manufacturer flag word in OTP memory.

For more information, see *3.2 OTP memory* on page 3-29.

3.3.2 Device Manufacturing LCS

The Device Manufacturing (DM) LCS is used during device assembly on the production line of the OEM.

In this LCS, debugging and external testing capabilities are still enabled. Internal testing capabilities, for example full-chip internal scan, are disabled.

LCS is DM if the following is true:

- Manufacturer-programmed flags, OTP word `0x10≠0`.
- OEM-programmed flags, bits [7:0] of OTP word `0x21` = zero.

To transition to Secure LCS, the OEM must program all OEM-owned security assets.

3.3.3 Secure LCS

The Secure LCS is used for devices out of the manufacturing line and "in the field".

It permits the execution of security functions, but blocks debugging and testing capabilities. Using Secure Boot is mandatory in this LCS.

LCS is Secure if the following is true:

- Manufacturer-programmed flags, OTP word `0x10`≠0.
- OEM-programmed flags, bits [7:0] of OTP word `0x21`≠0.

3.3.4 RMA LCS

The RMA LCS is a terminal state for devices that are returned to the manufacturer for analysis of fatal failures.

When a device is put into RMA LCS, it loses its existing secret keys, but regains full access to all debugging and testing capabilities, as available in *3.3.1 Chip Manufacturing LCS* on page 3-33. All cryptographic engines are usable for testing, but the root keys change for each boot phase. Between cold-boots, the device's cryptographic functions can be tested, but no permanent secrets can be derived from the root keys:

1. *HUK* is replaced with a different random value with each boot cycle. Therefore, any previously-saved data that is protected by a key derived from *HUK* is lost. However, keys can still be derived from *HUK* and used until the next boot cycle.
2. *Kce* and *Kceicv* are invalidated, so Secure Boot can be used only in non-encrypted mode.
3. *Kcp* and *Kpicv* are invalidated, so provisioning can no longer be done based on the previous values.

3.4 Boot services

Boot services include Secure Boot and Secure Debug.

Secure Boot:

A certificate-based mechanism using RSA private-public key scheme, designed to guarantee that only authenticated, and optionally encrypted, software images are loaded on a target system.

To guarantee system integrity, the Secure Boot must be used for loading of any code at any stage. For example, Second-stage boot loader or the OS.

For more information, see *3.4.2 Secure Boot* on page 3-37.

Secure Debug:

A certificate-based mechanism using RSA private-public key scheme, designed to enable secure debugging of the device.
For more information, see *3.4.3 Secure Debug* on page 3-41.

Secure Boot and Secure Debug security is based on the following elements:

OTP secrets:

Provisioned to the device during device manufacturing stage.

ROM code:

A code library linked into the ROM code of the device.

RSA scheme verification:

Both Secure Boot and Secure Debug are compliant with *Arm® Trusted Board Boot Requirements CLIENT* requirements.

Secure Boot and Secure Debug verification is performed over a certificate chain that may be comprised of two or three certificates.

The following diagram illustrates a three-level certificate-chain verification:

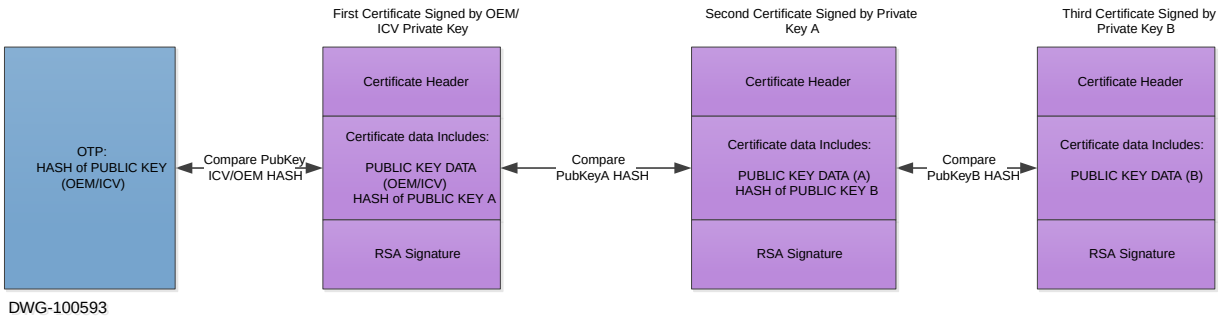

Figure 3-2 Certificate-chain verification

Secure Boot and Secure Debug certificate chains are different:

- For more information on the Secure Boot certificate chain, see *Secure Boot flow* on page 3-38.
- For more information on the Secure Debug certificate chain, see *Secure Debug flow* on page 3-42.

The following diagram details the overall certificate-verification process. This process is generic for all certificate types.

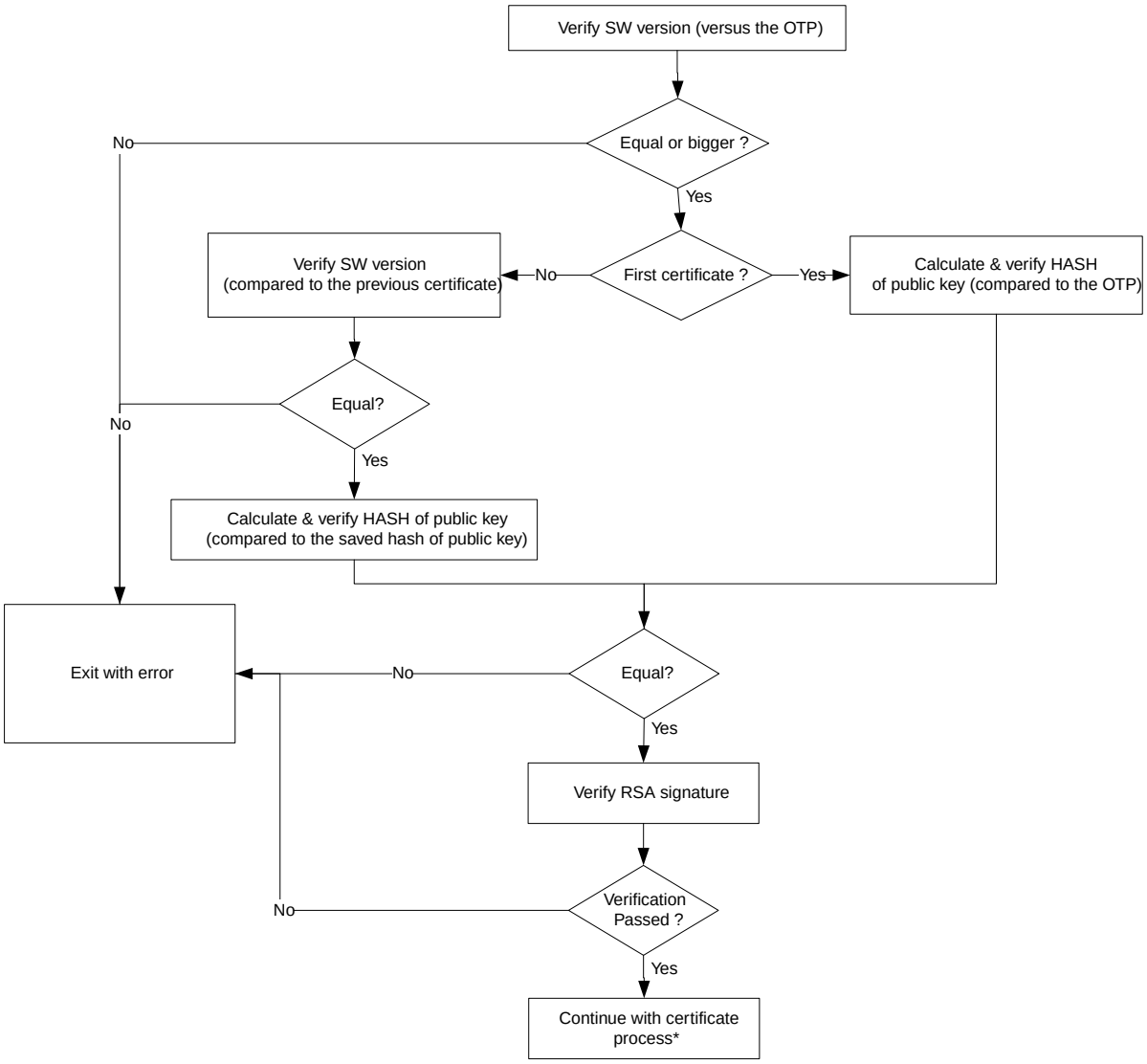

Figure 3-3 Certificate verification process

The certificate verification process includes:

1. Get the public key from the certificate and calculate its hash.
2. Compare the calculated hash:
   - If it is the first certificate in the chain, compare it with the hash value stored in the OTP.
   - Otherwise, compare it with the saved hash from the previous certificate.
3. Verify the RSA signature using the certificate's public key.
4. Verify the SW version, if exists, against the values in the OTP and the previous certificate. All certificates in the chain must have the same SW version, which must be bigger than or equal to the version saved in the OTP.
5. Save the public key hash of the next certificate, unless it is the last certificate in the chain.

This section contains the following subsections:

- *3.4.1 Certificate flavors* on page 3-37.
- *3.4.2 Secure Boot* on page 3-37.
- *3.4.3 Secure Debug* on page 3-41.

3.4.1 Certificate flavors

The Secure Boot services package supports two certificate flavors. The flavor must be defined, based on what is integrated inside the ROM, by changing the `CC_CONFIG_SB_X509_CERT_SUPPORTED` configuration flag:

- `CC_CONFIG_SB_X509_CERT_SUPPORTED = 1` - X.509 certificate.
- `CC_CONFIG_SB_X509_CERT_SUPPORTED = 0` - proprietary certificate.

Note

- Secure Boot and Secure Debug certificates must use the same format - either proprietary or X.509.
- X.509 certificates are larger than proprietary certificates.

3.4.2 Secure Boot

Secure Boot is a certificate-based mechanism using RSA private-public key scheme, designed to guarantee that only authenticated, and optionally encrypted, software images are loaded on a target system.

Secure Boot is composed of the following elements:

- Server-side tools for creating signed Secure Boot certificates for a given set of software images. For more information, see *Secure Boot certificate creation* on page 3-38.
- Device-side Secure Boot APIs that must be integrated in the boot sequence of the target device. For more information on APIs, see *Arm® TrustZone® CryptoCell-312 Software Developers Manual*. These routines are called to parse the certificate chain, and only allow loading of approved code. For more information, see *Secure Boot sequence* on page 4-56.

The CryptoCell-312 Secure Boot mechanism provides the following main features:

- Authenticates system images during the boot process, based on a root-of-trust kept in secure non-volatile memory.
- The trust model for software distribution is based on asymmetric cryptography RSA signatures with the *Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications* schemes. This enables all devices to share a single software image and use the same public key.
- Performs in-field boot time software integrity verification using the same public-key cryptography mechanism as is used for software distribution.
- Supports verification of multiple software images during a single boot sequence.
- Supports verification of multi-component software images.
- Supports chaining of secondary public keys signed by the root-of-trust public key, to permit parts of the system image to be signed by other trusted developers.
- Supports optional software versioning and revocation of specific software versions.
- Supports loading of code images encrypted using the code encryption key. For more information, see *3.1.3 Class keys* on page 3-27
- Provides tools for the system and software providers to support key management, device provisioning, and signing of software images.

Secure Boot behavior depends on the current security LCS:

- In CM LCS, Secure Boot does not match the certificate with the key in OTP memory, as the key is not programmed yet.
- In DM LCS, it depends on the signing key:
  - If the signing key is *Hbk0*, the key should exist on the OTP, burned in CM LCS, and therefore verifies it.
  - If the signing key is *Hbk* or *Hbk1*, Secure Boot ignores it, as it does in CM LCS.

Secure Boot certificate creation

CryptoCell-312 includes server-side tools that are used to create signed Secure Boot certificates for a given set of software images.

The key certificate tool:
This tool is used for signing either the public key for verification of the content certificate, or the next-level key certificate, according to the scheme used.

For more information, see *5.2 Secure Boot PC tools* on page 5-66.

The content certificate tool:
This tool is used for processing a list of software images: hash computation and optional encryption, and signing the list of hash values with a private key matching the public key that was signed by the key certificate.

Each certificate can be generated by a different entity, thereby enabling different parties to place authenticated code on the device, while sharing only the public key with the generator of the parent certificate.

The resulting certificates must be placed in the flash memory of the device, and the software images must be placed in the storage addresses referenced by the content certificate.

Secure Boot flow

The Secure Boot certificate chain is comprised of key certificates and content certificates.

Key certificates:
Mainly used to validate the next certificate in the chain.
Content certificates:
Used to load and validate software components.

Key-certificate validation is performed according to the flow shown in *Figure 3-3 Certificate verification process* on page 3-36. The same flow serves as the first part of content-certificate verification.

1. Call `CC_SbCertVerifySingle` to verify each certificate in the chain separately and sequentially.For more information, see *Secure Boot sequence* on page 4-56.
2. If the certificate validation passed successfully, Secure Boot continues to process the certificate based on the certificate type:

   For key certificates:
   The processing includes saving the hash of the next certificate.
   For content certificates:
   The certificate processing includes loading and verifying the SW images.

The following drawing details the Content certificate processing. This processing is done in a loop for every SW image that is signed in the certificate:

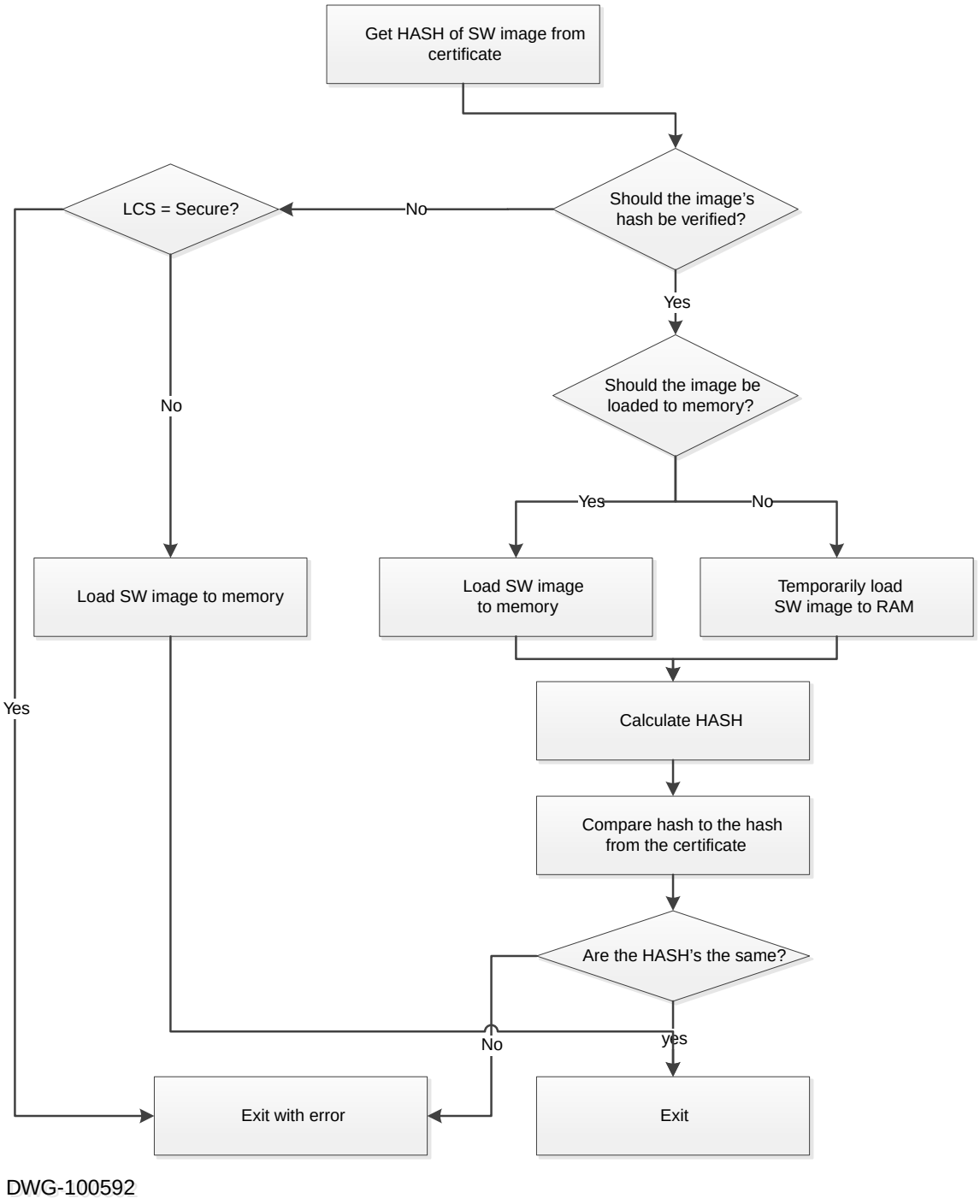

Figure 3-4 Content certificate verification process

The content certificate contains the following information for every image it is to verify:

- The address where the SW image is loaded to [load address].
- The flash address where the SW image is stored [storage address].
- The size of the SW image.

The content certificate allows you to choose the loading and cryptographic schemes.

Loading schemes:

These are the loading mechanism used to load the SW images.

Secure Boot supports the following loading schemes:

Load and verify:
The Secure Boot code copies the SW image from the flash to a given memory address, and verifies the hash value of the image.

Verify in memory:
The Secure Boot code only verifies the hash value of the image while the image resides in RAM.

Load from flash:
The SW image is only loaded to a given memory address, without hash verification.

Note

This case is not permitted in Secure LCS.

Note

If SW images are encrypted, they must be loaded to memory.

Cryptographic schemes:

This is the SW-image verification process. The encryption scheme and the verification scheme. During certificate creation, you must decide between the following cryptographic schemes:

- Calculating hash on the plain (decrypted) SW image.
- Calculating hash on the encrypted image.

Note

- The cryptographic scheme used is defined in the content certificate.
- Content-loading is performed through a user callback-function. You must make sure that this function does not copy from restricted or secure regions.

You must choose one of the following supported Secure Boot certificate schemes:

Two-level SB certificate scheme:
The two-level SB certificate chain order is key certificate → content certificate.

Three-level SB certificate scheme:
The three-level SB certificate chain order is key certificate → key certificate → content certificate.

The advantage of using the three-level Secure Boot certificate scheme over the two-level certificate scheme is that it provides renewability and classification: if an OEM key is leaked, the certificates can be easily replaced, with the existing *Hbk* that is in OTP.

For more information on certificate-chain verification, see the *Figure 3-3 Certificate verification process on page 3-36* figure.

Secure Boot limitations

Secure Boot has the following functional limitations:

- Each certificate must be contiguous in flash memory.
- Each software image to be authenticated must:
  - be contiguous in flash memory.
  - have an address that is aligned to the data bus width: 32-bits or 64-bits.
  - have a size that is a multiple of 16 Bytes.

3.4.3 Secure Debug

Secure Debug is a certificate-based mechanism using RSA private-public key scheme, designed to enable secure debugging of the device.

Secure Debug supports the following features:

- Performs boot-time verification of debug certificates that enable authenticated debugging of secure domains, controlled by the DCU registers, on the SoC.
- Allows an authorizing party to shift the device into RMA LCS by using the same certificate mechanism.

A debug certificate enables a specific list of debug interfaces on a specific device. Each OEM or ICV generates enabler debug certificates that are signed using your private key, which is rooted in the on-chip OTP memory. These certificates enable a requesting party, for example, a software developer, to debug certain interfaces. The developer extends these to actual debug certificates by embedding the device IDs, and signing the full certificate.

Each certificate chain is device-specific, defined by the `SOC_ID`. The `SOC_ID` is unique-per-devic, and created using the *HUK*, making it a unique identifier.

Note

The OEM or ICV must provide a method of retrieving a `SOC_ID` of the target device, which is the result of `CC_BsvSocIDCompute`, without having to first enable debugging.

To enable debugging, your ROM boot code loads the debug certificate and passes it to the `CC_BsvSecureDebugSet`, which verifies the signatures on the certificate, both on the enabler part and the developer debug certificate. If verification is successful, the debug interfaces indicated in the certificate are enabled.

Secure Debug behavior is dependent on the current LCS:

In CM LCS

The certificates are verified by the RSA signature, as done in the Secure Boot. This means that any valid certificate may be used.

In DM LCS

- If the certificate is signed with *Hbk1* or *Hbk*, the Secure Debug does not verify the root key and the certificates are only verified by the RSA signature, as in Secure Boot. This means that any valid certificate may be used.
- If the certificate is signed with *Hbk0* the Secure Debug behavior is as defined in Secure LCS.

In Secure LCS

- The first certificate is verified against the OTP.
- The unique device `SOC_ID` is required to create the debug certificate. It is only validated in this LCS. For more information, see *Boot ROM completion in Secure LCS* on page 4-55.

Note

Each certificate chain is valid only in the LCS it was created for.

Note

In all LCSes, other than Secure LCS, the DCUs are open by default. The default behavior definitions are set in the RTL. For more information, see the *Secure debug behavior in different lifecycles* section in the *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual*.

Secure Debug certificate creation

CryptoCell-312 includes server-side tools that are used for creating signed Secure Debug certificates for debugging purposes.

The key certificate tool:

This tool is used for signing the public key for verification of the enabler certificate. This is an optional tool, which may be used according to the required certificate chain.

For more information, see *5.2 Secure Boot PC tools* on page 5-66.

The enabler debug certificate tool:

This tool is used for creating the enabler debug certificate, which defines the following:

- A mask determining which DCU bits are open for editing by the developer.
- A mask specifying which of the DCU bits are locked after successful Secure Debug.

The developer debug certificate tool:

This tool is used for creating the developer debug certificate, which defines the following:

- Which DCU bits the developer wishes to set.
- The `SOC_ID`.

The resulting certificates must be placed in the memory of the device.

Secure Debug flow

Secure Debug certificate processing is integrated into the cold boot sequence, and supported in all LCSes.

Once a debug certificate is verified during the boot sequence of the device, the Secure Debug boot code uses a dedicated hardware interface to output control words (DCUs) that enable the defined debug mechanisms in the system.

The following diagram illustrates the Secure Debug processing flow:

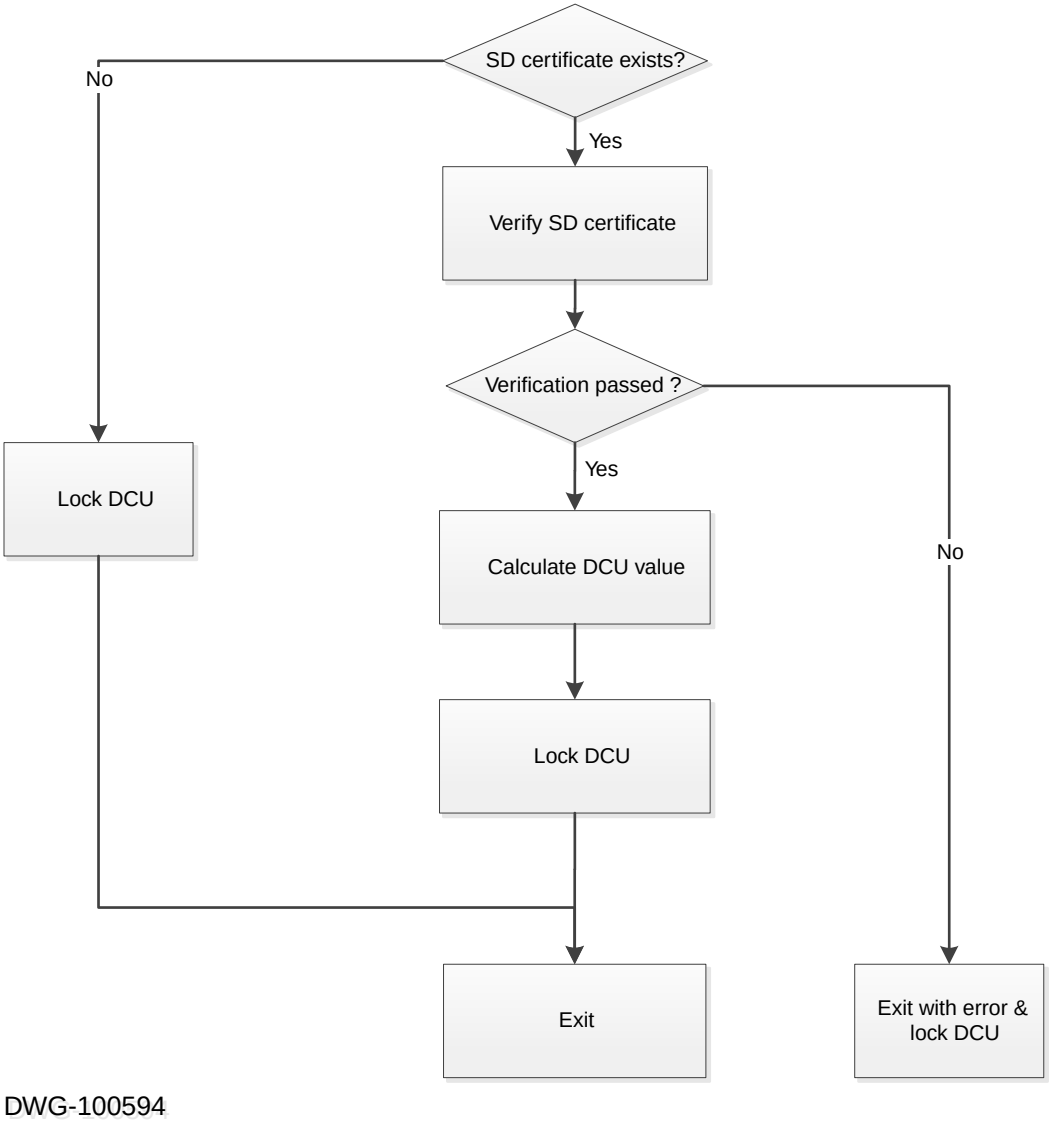

Figure 3-5 Secure Debug certificate processing

A device can be debugged in one of the following ways:

Using the Secure Debug certificate mechanism.

Using the HW set of default DCU values to enable or disable debugging:

The DCU value is calculated according to the overlapping bits set in the permanent values and default values of the HW DCUs. The overlapping bits are calculated according to the LCS of the device.

DCU locking is done according to the lock mask set in the OTP.

The Secure Debug DCU authorization mechanism consists of the following SW and HW components:

HW DCU restriction mask:
The DCU restriction mask is used to control access to the DCU. If a bit is set, the correlating DCU bit is owned by the ICV, and only the ICV can set this DCU bit. If a bit is not set, that is equals zero, the correlating DCU bit is owned by the OEM, and only the OEM can set this DCU bit. For more information, see *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual*.

HW-DCU permanent disable values.
HW-DCU default values for each security LCS.
SW enabler and developer values, received from the certificate.

To operate Secure Debug in ROM code, use the `CC_BsvSecureDebugSet` API. For more information, see *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

Note

- The `CC_BsvSecureDebugSet` API must be called, even if no debugging is required, to lock the DCU register according to the security LCS.
- Content-loading is performed through a user callback-function. You must make sure that this function does not copy from restricted or secure regions.

For a description of the integration of this API in the cold boot sequence, see *4.1.1 Cold boot sequence boot ROM guidelines* on page 4-53.

The Secure Debug mechanism is based on chains of certificates compliant with *Arm® Trusted Base System Architecture V1: System Software on Arm* and *Arm® Trusted Board Boot Requirements CLIENT*.

For the detailed certificate-verification process, see the *Figure 3-3 Certificate verification process* on page 3-36 figure.

You must choose between the following Secure Debug certificate schemes:

Two-level SD certificate scheme:
The two-level SD certificate chain order is key certificate → enabler debug certificate.

Three-level SD certificate scheme:
The three-level SD certificate chain order is key certificate → enabler debug certificate → developer debug certificate.

The advantage of using the three-level Secure Debug certificate scheme over the two-level certificate scheme is that the key certificate is protected by the software version, and thus the chain can be revoked if any of the private keys of the chain has been compromised.

DCU handling

DCU calculation and locking is done according to the signing authority.

An ICV-signed certificate chain:

- The DCU value is calculated from the overlapping bits set in the enabler mask, the developer value, and the ICV bits in the HW restriction mask.
- The DCU lock value is calculated from the combination of bits set in the enabler lock mask, the default bit set in the OTP, and the OEM bits in the HW restriction mask.

An OEM-signed certificate chain:

- The DCU value is calculated from the overlapping bits set in the enabler mask, the developer value, and the OEM bits in the HW restriction mask.
- The DCU lock value is calculated from the combination of bits set in the enabler lock mask, the default bit set in the OTP, and the ICV bits in the HW restriction mask.

If ICV and OEM are a single entity:

- The DCU value is calculated from the overlapping bits set in the enabler mask and the developer value.

Note

If there are bits set to zero in the permanent mask, these bits can never be set to one, even if the enabler mask requests it.

- The DCU lock value is calculated from the combination of bits set in the enabler lock mask, and the default bit set in the OTP (word `0x28-0x2B`). For more information, see *3.2 OTP memory* on page 3-29

The following tables provide more information on DCU and DCU lock calculation:

Table 3-5 DCU calculation

First certificate signing entity	DCU value
ICV	DCU_{DEV} & DCU_{ENB} & DCU restrictions (ICV owned bits) & permanent disable.
OEM	DCU_{DEV} & DCU_{ENB} & !DCU restrictions (OEM owned bits) & permanent disable.
ICV or OEM (full *Hbk*)	DCU_{DEV} & DCU_{ENB} & permanent disable.

Table 3-6 DCU lock calculation

First certificate signing entity	DCU value
ICV	($Lock_{ENB}$ & DCU restrictions (ICV owned bits)) | (!DCU restrictions (OEM owned bits) & $Lock_{OTP}$)
OEM	($Lock_{ENB}$ & !DCU restrictions (OEM owned bits)) | (DCU restrictions (ICV owned bits) & $Lock_{OTP}$)
ICV or OEM (full *Hbk*)	$Lock_{ENB}$

Note

If there is no certificate chain, or an error occurs during Secure Debug certificate verification, the DCUs are locked according to the default value in the OTP.

3.5 Non-Volatile counters

Secure Boot and Secure Debug certificates use monotonic non-volatile (NV) revocation counters to prevent rollback attacks.

NV counters are defined in *Arm® Trusted Base System Architecture V1: System Software on Arm* and *Arm® Trusted Board Boot Requirements CLIENT*.

CryptoCell-312 includes the following NV counters, maintained in OTP memory:

ICV counter:
A 64-bit counter.

OEM counter:
A 96-bit counter.

If there is only one entity in the system, CryptoCell-312 includes a single counter with the combined size of 160 bits.

——— **Note** ———

The size of the OEM counter is not fully-compliant with *Arm® Trusted Base System Architecture V1: System Software on Arm*, due to OTP-size limitations.

———————————

Each Secure Boot certificate includes an NV counter value.

When a certificate is parsed, its NV counter is compared to its existing (programmed) value:

- Certificates with an NV counter value that is lower than the programmed value are rejected.
- Certificates with an NV counter value that is equal to or higher than the programmed value are processed.

After a full certificate-chain is successfully validated, if the NV counter of the certificates in the chain is higher than the programmed value of this counter, the programmed value in the OTP is updated to match the new value found in the certificate chain. As a result, any older certificates using lower NV counter values become invalid.

——— **Note** ———

- Each certificate in the same chain must have the same NV counter value.
- Counter-update requires the device to support in-field OTP memory programmability.

———————————

3.6 Power management

CryptoCell-312 requires some states to be retained for the entire duration of the system lifecycle. An example of this would be currently-active keys.

To achieve this, the host system needs to latch the CryptoCell-312 "always-on" signals in your always-on power domain. For more information, see the *Power and clock domains guidelines* section in the *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual*.

CryptoCell-312 may be powered down independently from the main power control of the system. A power management unit (PMU) can identify the CryptoCell-312 power state by referring to its IDLE-state signal, and using it to build the following power-saving schemes:

The `core_clk` gating mechanism.

For more information, see the *Implementing core_clk gating mechanism* section in the *Arm® TrustZone® CryptoCell-312 Configuration and Integration Manual*.

A full power-down of CryptoCell-312.

The CryptoCell-312 SW only enables a power-down signal when there are no active cryptographic operations. A full power-down requires the PMU SW to call the CryptoCell-312 SW before powering off the device, then calls it again after powering up the device, to allow saving and restoring state values.

CryptoCell-312 maintains information regarding its cryptographic operations using the `CC_PalPowerSaveModeSelect` PAL function. This function is called for every cryptographic operation, and when the operation completes. You must adapt this function to the PMU implementation of your system, and notify it when CryptoCell-312 is idle. The reference implementation uses an atomic counter, which increases when CryptoCell-312 is processing an operation, and decreases when the operation completes. When CryptoCell-312 is to be powered down, the PMU must call the `mbedtls_mng_suspend` API. When CryptoCell-312 is powered up, the PMU must call the `mbedtls_mng_resume` API to restore CryptoCell-312 to its previous state.

3.7 Device provisioning

Provisioning of CryptoCell-312 basic secrets is a complex process.

For more information, see *4.3 Provisioning devices* on page 4-62.

This section contains the following subsections:

3.7.1 Boot ICV asset provisioning

CryptoCell-312 provides functions to support secure provisioning of ICV assets to devices during the boot phase.

These assets are distributed as binary objects that are encrypted and authenticated using AES in CCM mode.

Provisioning support includes both hardware and software components embedded in the devices, as well as the tools required for packaging assets on the computers of the ICV:

On the PC side:
The asset-provisioning tool provided by Arm takes a binary asset and an asset identifier, and generates an encrypted asset object. This asset object includes the encrypted asset data combined with authentication information using AES-CCM for encryption and authentication, with the key derived from the *Kpicv* key and the asset identifier.

On the device side:
A corresponding asset-access function, embedded in the software boot code of the device, performs the reverse operation: verifying the authenticating information of an encrypted asset object, and extracting the asset from it.

For more information, see *5.4 Secure asset-provisioning PC tool* on page 5-73.

3.7.2 Runtime asset provisioning

This section describes the procedure for secure provisioning of OEM or ICV-specific secret assets (for example, DRM keys) or integrity-critical data (for example, certificates), to the device.

CryptoCell-312 provides functions to support secure provisioning of ICV or OEM assets to devices. These assets are distributed as binary objects that are encrypted and authenticated using AES in CCM mode. Provisioning support includes both hardware and software components embedded in the devices, as well as the tools required for packaging assets on the ICV or OEM's computers:

- On the PC side: The asset-provisioning tool provided by Arm takes a binary asset, and an "asset identifier", and generates an encrypted asset object. This asset object includes the encrypted asset data, combined with authentication information that uses AES-CCM for encryption and authentication. The key used for this operation is derived from either the *Kpicv* key (for the ICV), or from the *Kcp* (for the OEM).
- On the device side: A corresponding asset-access function embedded in the device's software runtime code performs the reverse operation: verifying an encrypted asset object's authenticating information, and extracting the asset from it.

3.8 Cryptographic acceleration

CryptoCell-312 supports Arm Mbed TLS APIs.

Mbed TLS is an open source SSL library licensed by Arm that enables developers to include cryptographic and SSL/TLS capabilities in their embedded products, with a minimal coding footprint.

You can choose between HW-accelerated cryptographic operations and the SW cryptographic implementation of Arm Mbed TLS for each feature supported by both Mbed TLS and CryptoCell-312:

- The Mbed TLS cryptographic implementation provides an interface to the standard cryptographic operations, for example, AES, RSA or ECC.
- The dedicated CryptoCell-312 APIs provide an interface to the non-standard or specific CryptoCell-312 operations, for example, key derivation using *HUK* or external DMA operations.

This section contains the following subsections:

- *3.8.1 Build process* on page 3-49.
- *3.8.2 Controlling cryptographic features* on page 3-49.

3.8.1 Build process

This section details the steps needed to build Mbed TLS for use with CryptoCell-312.

To build Mbed TLS, perform the following steps:

1. Download the Mbed TLS package from *https://tls.mbed.org/*, and save it to the same directory as the package root of CryptoCell-312, at the same level of `codesafe` and `host`.
2. From the root directory, execute the following commands:
   a. `./prepare_mbedtls.sh lib`
   b. `make -C host/src`

Once these steps are completed successfully, both the CryptoCell-312 and Mbed TLS libraries are built.

For information on the Mbed TLS build process, see the Mbed TLS knowledge base, which can be found at *https://tls.mbed.org/kb*.

Note

The `prepare_mbedtls.sh` script can be used to fetch the Mbed TLS code from GitHub by using the `./prepare_mbedtls.sh clone` command.

3.8.2 Controlling cryptographic features

Mbed TLS and CryptoCell-312 are flexible in terms of which features are compiled in each.

To control which components are Mbed TLS-based or CryptoCell-312-based, you must edit the `config-cc312.h` configuration file. This file is located in `shared/include/mbedtls`. It includes all the flags that are supported by Mbed TLS, with the additional `XXX_ALT` definition. These definitions are required for the components that are accelerated by the HW.

By default, the package is delivered with the minimal required features that CryptoCell-312 accelerates. You may add any other features by enabling them in `config-cc312.h`. This adds the features to the Mbed TLS libraries using SW-only implementation.

Additional features that are not part of the Mbed TLS library, for example key derivation using *HUK*, are controlled through the CryptoCell-312configuration file.

Your applications must include the Mbed TLS headers, which can be found after compilation under the `mbedtls/include` directory. The `config-cc312.h` configuration file must also be included, to assure that the same definitions apply to both the CryptoCell-312 library and the application that calls it.

3.9 Secure Boot services for the runtime library

CryptoCell-312 provides Secure Boot building blocks for you to build a Full FOTA process with.

The Secure Boot provided for the runtime library has almost the same functionality as the Secure Boot provided for the ROM library, with the following differences:

- The runtime component supports verification of images without loading them to the RAM.
- The runtime component allows the user to store images and certificates in memory and not in the flash, as done in the ROM Secure Boot.
- The runtime component does not update the SW version, that is, the NV counter. The following Secure Boot, as part of the boot process, updates it. For more information on the Secure Boot process, see *Secure Boot flow* on page 3-38.

  For more information on the certificates creation, see *Secure Boot certificate creation* on page 3-38.

3.10 External DMA support

CryptoCell-312 allows the user to configure the cryptographic engine without the data flow.

For this, you must have an external HW DMA that is responsible for writing and reading the data via the APB bridge.

CryptoCell-312 allows the user to start this operation and finish it, that is, release the cryptographic engine. While the engine is used, the software blocks any other cryptographic operations through the cryptographic APIs.

The external DMA operation supports only simple operations:

- AES.
- Hash.
- ChaCha

Chapter 4
Integrating CryptoCell-312 software

This chapter details the steps required for the integration of CryptoCell-312 software into a SoC.

It contains the following sections:

4.1 Integration guidelines

This section provides general guidelines for integrating CryptoCell-312 software into your environment.

It provides recommendations for how CryptoCell-312 is integrated, administered, and maintained.

Note

If you decide not to use Secure Boot to load code in a trusted way, the `CC_BsvFatalErrorSet` API must be called to disable the use of HW keys, TRNG, and asymmetric cryptography.

This section contains the following subsections:

4.1.1 Cold boot sequence boot ROM guidelines

This section provides a template for the boot sequence implemented by the boot ROM of the device.

To comply with the security requirements, the boot ROM or first-stage boot-loader, must follow the sequence described below:

1. To initialize the CryptoCell-312 ROM library, call `CC_BsvInit`.

   Note

   This must be the first CryptoCell-312 ROM API called.

2. To retrieve the current LCS and perform additional initializations, call `CC_BsvLcsGetAndInit` and verify the following conditions in the following order:
   a. If an error is returned, the HW could not determine the LCS or the *HUK*, likely due to OTP memory errors, and therefore CryptoCell-312 is unusable:
      a. Set CryptoCell-312 fatal error flag by calling `CC_BsvFatalErrorSet`. Fatal error state ensures that all HW keys are masked, and asymmetric and random operations are unavailable.
      b. Call the Secure Debug API to lock the DCU.
      c. Abort the boot sequence.

   Note

   We recommend that the device be completely disabled, not just security-wise, but all functionality.

3. To initialize the CryptoCell-312 HW and platform security definitions, call `CC_BsvCoreClkGatingEnable`, `CC_BsvSecModeSet` and `CC_BsvPrivModeSet`.

Reaching this point in the flow means that the LCS is valid, and is either CM LCS, DM LCS, Secure LCS or RMA LCS.

Continue to follow the guidelines according to the returned LCS, as detailed in *Boot ROM completion* on page 4-54.

Note

In case of any error, excluding errors from the Secure Debug API, the boot flow must be aborted, and the security disabled by calling the `CC_BsvFatalErrorSet` API.

The following pseudo-code illustrates the cold boot sequence described above.

```
void BootROM_ColdBootSequence()
{
ret = CC_BsvInit(ccHwBaseAddr);
if (ret != OK)
```

```
DeviceCompleteDisable();
ret = CC_BsvLcsGetAndInit(ccHwBaseAddr, pLcs);
if (ret != OK){
CC_BsvFatalErrorSet(ccHwBaseAddr);
/* this is done in order to lock the DCU and not enable debug */
debugRet = CC_BsvSecureDebugSet(..., NULL, ..., pRmaFlag);
Abort_bootCode(); /* doesn't return */
}

ret = DebugCertificateExists(pDebugCert);
if (ret == OK){ /*In case
          debug certificate exists */

debugRet = CC_BsvSecureDebugSet(..., pDbgCertPkg, ..., pRmaFlag);
}
else{
debugRet = CC_BsvSecureDebugSet(..., NULL, ..., pRmaFlag); /* Must be called to lock the
DCUs*/
}
if ((*pLcs != RMA) && (((debugRet == OK) && *pRmaFlag) || rmaEntrySignalIsSet))
BootROM_RmaModeEntry(); /* Doesn't return */

switch (*pLcs) {
case CM:BootROM_CM();
break;
case DM:BootROM_DM();
break;
case Secure:BootROM_Secure();
break;
case RMA:BootROM_RMA();
break;
default:DeviceCompleteDisable();
break;
}
}
```

Boot ROM completion

The cold boot sequence final steps are determined by LCS, as detailed in the following subsections.

Secure Debug should be called before Secure Boot.

When in Secure LCS, Secure Debug might set certain bits, resulting in CryptoCell-312 resetting and masking HW keys. If Secure Boot is then run using images that were encrypted using *Kce* or *Kceicv*, decryption fails due to HW-key masking. You can override the CryptoCell-312 reset operation by setting the reset override bit to one.

Boot ROM completion in CM LCS

This section details how to complete the cold boot sequence in CM LCS.

To complete boot ROM sequence in CM LCS, perform the following steps:

1. Check for the existence of an ICV factory tool signature. If found:
   a. Run the ICV factory tool, as described in *6.2.2 ICV factory tool loading methods* on page 6-80.
   b. At the end of the ICV factory tool, a PoR must be initiated, and the device is reboot in DM LCS.

Note

Secure Boot and Secure Debug have no meaning in CM LCS. The Hash of the first certificate public key is not validated at this stage, only the RSA signature of the certificate. However, Secure Boot and Secure Debug can be run for testing purposes.

Note

In CM LCS, OTP is still filled with zeros and therefore, the HW keys relevant for Secure Boot are also filled with zeros. SW-image encryption should be done with zeros.

Boot ROM completion in DM LCS

This section details how to complete the cold boot sequence in DM LCS.

To complete boot ROM sequence in DM LCS, perform the following steps:

1. Check for the existence of OEM factory tool signature. If found:
   a. Execute OEM factory tool, as described in *6.3.2 OEM factory tool loading methods* on page 6-82.
   b. At the end of the OEM factory tool, a PoR must be initiated, and the device is reboot in Secure LCS.
2. If a valid Secure Debug certificate exists:
   a. Call Secure Debug to enable debugging or to lock the DCUs.

      Note

      Secure Debug is not mandatory in this LCS, so that error handling can be less strict than in Secure LCS.

3. If a valid Secure Boot certificate exists:
   a. Load trusted code, second-stage boot loader, or the OS, using *Secure Boot sequence* on page 4-56.

      Note

      Secure Boot is not mandatory in this LCS, so that error handling can be less strict than in Secure LCS.

   Note

   In this LCS, Secure Boot skips *Hbk* and *Hbk1* verification and only verifies *Hbk0*. However, it fails if the images are encrypted, as *Kce* is missing.

4. Go to the trusted code, and continue with the cold boot sequence, as described in *4.1.2 Cold boot sequence in runtime software* on page 4-57.

Boot ROM completion in Secure LCS

This section details how to complete the cold boot sequence in Secure LCS.

To complete boot ROM sequence in Secure LCS, perform the following steps:

1. If `SOC_ID` was not previously exported:
   a. Call `CC_BsvSocIDCompute`.
   b. Export the resulting `SOC_ID`, so that it can be accessed by a developer wishing to debug the device. For example, placing it in a designated address in the flash memory.

      Note

      `SOC_ID` is not a secret, so its export media does not have to be secure.

   Note

   This step can also be performed later, from the runtime software. If performed from the runtime software, the device boot sequence might malfunction, preventing `SOC_ID` export, and potentially block debugging.

2. Call Secure Debug, with one of the following:
   - A valid certificate chain: to allow debugging.
   - NULL: to lock the DCUs.

   For more information, see *Secure Debug flow* on page 3-42.
3. If Secure Debug returned with the RMA flag set to ON, continue with the *RMA LCS entry sequence* on page 4-57.
4. Call Secure Boot with a valid certificate chain to load on of the following using *Secure Boot sequence* on page 4-56:

- Trusted code.
- Second-stage boot loader.
- The OS.

If there are two entities in the system, both ICV and OEM, the ICV must do the following at the end of the boot flow:

- Lock its keys by calling the `CC_BsvICVKeyLock` API, to prevent the OEM from using one of them.
- Prevent the OEM from a single transition to RMA by calling the `CC_BsvICVRMAFlagBitLock` API.For more information on proper transitioning to RMA LCS, see *RMA LCS entry sequence* on page 4-57.

5. Go to the trusted code, and continue with the cold boot sequence, as described in *4.1.2 Cold boot sequence in runtime software* on page 4-57.

Boot ROM completion in RMA LCS

This section details how to complete the cold boot sequence in RMA LCS.

To complete boot ROM sequence in RMA LCS, perform the following steps:

1. Load trusted code.
2. Continue with the cold boot sequence, as described in *4.1.2 Cold boot sequence in runtime software* on page 4-57.

Important

When entering RMA LCS, the *HUK*, *Kce*, *Kcp*, *Kpicv*, and *Kceicv* HW keys, are permanently erased (set to one on the OTP), and are not loaded into the HW registers.

Kce and *Kceicv* are not valid, so Secure Boot can only be used if the software modules are not encrypted.

Secure Boot sequence

This section details the Secure Boot sequence.

The Secure Boot sequence is composed of the following steps:

1. To initialize the certificate processing, call `CC_SbCertChainVerificationInit`.
2. To verify the first key certificate, call `CC_SbCertVerifySingle`.

   Note

   In CM LCS or DM LCS, the public key used to sign the first key certificate might not be verified against the *Hbk* in OTP memory, if the *Hbk* is not yet burned. However, wrong signature or hash values of specific software modules still result in an error returned.

3. To verify the second key certificate if using a three-level Secure Boot certificate scheme, call `CC_SbCertVerifySingle` again.
4. To verify the hash values of the content certificate or software images, and optionally decrypt or load the authenticated software modules, call `CC_SbCertVerifySingle`.

Note

If an error is returned by any of the APIs in the flow, abort the sequence and return an error. In case of an error, Secure Boot does not clear the SW images from the RAM. Cleanup of these images from RAM is your responsibility.

The following pseudo-code illustrates the Secure Boot sequence described in this section.

```
CCError_t BootROM_SecureBootSequence()
{
ret = CC_SbCertChainVerificationInit(keyA_CertPkgInf);
if (ret != OK) return ret;
ret = CC_SbCertVerifySingle(… keyA_CertPkgInf …);
if (ret != OK) return ret;
#ifdef THREE_LEVEL_SCHEME
```

```
ret = CC_SbCertVerifySingle(… keyB_CertPkgInf …);
if (ret != OK) return ret;
#endif /* THREE_LEVEL_SCHEME */
ret = CC_SbCertVerifySingle(… content_CertPkgInf …);
return ret;
}
```

Secure Debug sequence

This section details the Secure Debug sequence.

The Secure Debug sequence is composed of the following steps:

1. Read the certificate and load it to memory.
2. Program and lock the DCU bits or return the `pEnableRmaMode` flag, as defined by the Secure Debug certificate, by calling `CC_BsvSecureDebugSet`.

Note

When the RMA LCS indicator is set, the DCUs are locked.

RMA LCS entry sequence

Entering RMA LCS is triggered by using the Secure Debug certificates.

The following flow describes the entry sequence, dependent on the number of entities in the system:

If ICV and OEM are two separate entities, the RMA LCS entry sequence must be as follows:

1. Create a Secure Debug certificate chain signed by the OEM, where the RMA flag in the enabler debug certificate is set.
2. Call the Secure Debug flow. If the `pEnableRmaMode` flag returned is set to one, call `CC_BsvRmaModeEnable` and perform a PoR to reactivate boot ROM. The device boots in its current security lifecycle state (Secure LCS, DM LCS or CM LCS).
3. Create a Secure Debug certificate chain signed by the ICV, where the RMA flag in the enabler debug certificate is set.
4. Call the Secure Debug flow. If the `pEnableRmaMode` flag returned is set to one, call `CC_BsvRmaModeEnable` and perform a PoR to reactivate boot ROM. The device boots in RMA LCS.

If ICV and OEM are a single entity, the RMA LCS entry sequence must be a follows:

1. Create a Secure Debug certificate chain. In this chain, the RMA flag in the enabler debug certificate is set.
2. Call the Secure Debug flow, if the `pEnableRmaMode` flag returned is set to one, call `CC_BsvRmaModeEnable` and perform a PoR to reactivate boot ROM. The device boots in RMA LCS.

Note

The flow must be in this order otherwise the function `CC_BsvSecureDebugSet` returns an error.

4.1.2 Cold boot sequence in runtime software

This section provides a template for using CryptoCell-312 software during the remainder of the boot sequence implemented by the device.

To use the CryptoCell-312 software library during the remainder of the boot sequence, perform the following steps:

1. Load and boot the OS using the *Secure Boot sequence* on page 4-56.
2. Initialize the runtime software library in the OS by calling `CC_LibInit`. It must be the first library API called.
   - If this API returns `CC_LIB_RET_RND_INST_ERR`, then TRNG failed to initialize. This means that CryptoCell-312 and its library are functional, but any attempt to use functionality that requires random vectors, for example key/IV generation, or signature operations, fails.

Note

If the device is in RMA LCS, the TRNG must be initialized to generate a random value for the *HUK*. If the API returns the above error, the *HUK* is set to all zeroes and not to a randomized value.

- Any other error returned is irrecoverable, resulting from either the HAL or PAL layers. This means that CryptoCell-312 and its library are not functional.

Note

In a multi-threaded system, we recommend initializing the DRBG context for each thread.

4.1.3 Runtime considerations

During runtime, all software APIs can be called.

For more information on APIs, see *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

These APIs use a number of mutexes for thread safety, each protecting a different resource. If the OS supports concurrent execution, the Mutex implementation in the PAL module must be adapted to the relevant platform/OS infrastructure.

`CC_LibInit` is the initialization entry point of the library. It initializes all Mutexes, and is not protected by any runtime Mutex. This API must be called exactly once. Its operation must be atomic: no other API in this library can be used by any thread before it returns.

Note

If the OS does not support multi-threading, the PAL implementation can be left empty.

4.1.4 CryptoCell-312 power management sequence

This section details the power-manaement sequence.

Before CryptoCell-312 is powered down, the PMU must call the `mbedtls_mng_suspend` API. This API verifies that there are no pending cryptographic operations, takes all the mutexes, and signals the HW that CryptoCell-312 is about to be powered down. This API also requires a buffer, allocated in a non-powered down memory, to save CryptoCell-312 parameters.

After the power down, when CryptoCell-312 is to be resumed, the `mbedtls_mng_resume` API must be called. This API is frees all the mutexes and restores the internal state.

Note

It is the responsibility of the partner to save all the variables used by them in CryptoCell-312 processes, for example RND context, including the CryptoCell-312 mutexes, and restore this information before calling the resume API.

4.2 Adaptation to your environment

This section describes the adaptations that might be required to customize the CryptoCell-312 software, both ROM and runtime, for your platform.

CryptoCell-312 Software and ROM include two layers that other modules use to abstract access to services that might have different implementations on different platforms:

Hardware Adaptation Layer (HAL)
This layer abstracts access to CryptoCell-312 HW resources.

Platform Adaptation Layer (PAL)
This layer abstracts access to functionality which is potentially platform-dependent, such as memory allocation, memory tools, and string manipulation.

These layers must be reviewed and adapted to your specific environment.

This section contains the following subsections:

4.2.1 Build environment customization for different architectures

You must adapt the build environment to your architecture and toolchain.

To adapt the build environment, perform the following steps:

1. Define the respective cross-toolchain environment in the `host/Makefile.defs` file.
2. Update the `shared/include/pal/cc_pal_compiler.h` file.

Arm toolchain definitions use the arm-Xilinx EABI toolchain. Supporting additional toolchains might require the following changes:

1. Update `cc_pal_compiler.h` to support your compiler.
2. Add an additional architecture name to the following line in `Makefile.defs`, following the `### Toolchain setup ###` comment:

```
ARCH_SUPPORTED = arm
```

4.2.2 ROM customization

This section details the ROM components that you must customize to your own environment.

ROM HAL customization

The HAL layer includes all HW-specific functionality, for example, registers, read and write macros, and interrupt handling.

In the provided code, the `SB_HalWaitInterrupt` interrupt handling function is an example, implemented as a polling loop. You can change this behavior and connect CryptoCell interrupts into the interrupt handler of your OS.

ROM HAL register macros are defined in `cc_hal_sb_plat.h`.

HAL functions are located in the `cc_hal_sb.c` file.

HW registers base address

All ROM library APIs require the base address of the CryptoCell-312 HW registers as an input parameter (`DX_BASE_CC`).

This address must be defined correctly in your ROM code.

OTP memory write

The `CC_BsvOTPWordWrite` API is used by CryptoCell-312 code that requires programming the OTP memory.

NV counters and RMA flags are the only fields that require change during the lifetime of the device.

For more information, see *3.2 OTP memory* on page 3-29.

4.2.3 Runtime software customization

This section details the runtime components that you must customize to your own environment.

Runtime PAL customization

Runtime PAL abstracts access to functionality which is potentially platform-dependent, such as memory allocation, memory tools, and string manipulation.

The runtime package is delivered with several PAL implementations, under `host/src/pal/`, that are tailored to the Arm-specific development environment.

As part of porting to your platform, this implementation must be replaced with one that matches the capabilities of your OS, and placed under `host/src/pal/`.

Note

Make sure that the *`TEE_OS`* variable is set correctly, so that the correct PAL implementation is selected. For more information, see *2.3 Host code build environment* on page 2-23.

The functionality that is abstracted by PAL (for example, string manipulation and memory tools), rarely differs from the default implementation.

Note

You must review the existing implementation under `shared/include/pal` and `host/src/pal` to verify that it matches your platform. The existing PAL implementation of Mutex support is tailored to Arm-specific development environment, and might not function on your platform.

For function descriptions, see *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

Runtime HAL customization

Runtime HAL abstracts access to CryptoCell-312 HW resources.

Runtime HAL functions are located in the `cc_hal.c` file, and defined in `shared/include/cc_hal.h`.

The HAL interrupt-handling function must be adapted to your OS. The CryptoCell-312 code implementation uses polling for the *`NO_OS`* or *`LINUX`* implementation and for the FreeRtos interrupt handler.

DX_BASE_CC

The `DX_BASE_CC` macro defines the base address for all CryptoCell-312 register references.

```
#define DX_BASE_CC 0xAAAAAAAA
```

It is defined in `shared/hw/include/dx_reg_base_host.h`, and must be modified to match the mapping of CryptoCell-312 on your platform.

TRNG characterization

CryptoCell-312 supports several TRNG flavors.

The 800-90B TRNG driver:

Supports the *NIST SP 90B: DRAFT Recommendation for the Entropy Sources Used for Random Bit Generation* standard.

The FE TRNG driver:

Supports the *BSI AIS-31: Functionality Classes and Evaluation Methodology for True Random Number Generators* standard.

The default TRNG driver for CryptoCell-312 is the FE TRNG driver.

The TRNG type is set in `proj.ext.cfg`, by the `CC_CONFIG_TRNG_MODE` parameter.

For FE TRNG driver:

Set `CC_CONFIG_TRNG_MODE` = 0.

For 800-90B TRNG driver:

Set `CC_CONFIG_TRNG_MODE` = 1.

For each TRNG mode, there are configurable parameters that you must be adjust to your target platform, according to the characterization process. For more information, see *Arm® TrustZone® TRNG Characterization Application Note*. These parameters are given to CryptoCell-312 using the `CC_PalTrngParamGet` API. This api must be implemented according to your system.

4.3 Provisioning devices

Provisioning of CryptoCell-312 basic secrets to devices is a complex process.

It must be carried out in the correct order, as detailed below:

ICV operations:

1. Prior to chip-tapeout:Define *Kprtl* and its embedding in the RTL of the chip.

   For more information, see the *Provisioning Key* section of *Arm® TrustZone® CryptoCell-312 Technical Reference Manual*.
2. Post chip-tapeout: Perform TRNG Characterization.
3. Software-related configuration decisions:
   a. *Hbk* scheme:
      Decide which of the following schemes to use:
      - Two 128-bit truncated hash values of PubKB0 and PubKB1.

        In this configuration, key #0 (*Hbk0*) belongs to the ICV, and key #1 (*Hbk1*) belongs to the OEM.
      - A single 256-bit non-truncated hash of PubKB0.

        In this configuration, this single key (*Hbk*) belongs to the OEM.

   b. Secure Boot:
      - Decide whether to use the code encryption feature.
      - Decide whether to use the minimal version feature.
      - If using two truncated hash values, define an ICV RSA private-public key-pair, PrvKB0 and PubKB0, for the device class, series or sub-series, or even unique per device. For an example of creating an RSA key pair,see *Appendix A Random key generation* on page Appx-A-84.

   c. Run the ICV factory tool to populate ICV-related fields in OTP memory.For more information, see *3.7.1 Boot ICV asset provisioning* on page 3-48.

OEM operations:

1. Define an OEM RSA private-public key-pair, PrvKB1 and PubKB1, for the device class, series or sub-series, or even unique per device. For an example of creating an RSA key pair, see *Appendix A Random key generation* on page Appx-A-84.
2. Run the OEM factory tool to populate OEM-related fields in OTP memory.

This section contains the following subsection:

4.3.1 OTP memory population

This section describes the manufacturing sequence of operations required to populate its OTP memory in order to achieve a fully-functional device.

The operational flow must be fully integrated in the device *4.1.2 Cold boot sequence in runtime software* on page 4-57.

——— **Note** ———

Parts of the programming procedure described in this section, requires the OTP memory to support independent programming of every bit in the same word.

———————————

OTP memory programming is performed in two parts: first by the ICV calling the ICV factory tool, and then by the OEM calling the OEM factory tool.

4.4 Debugging using Secure Debug

This section describes debugging using the Secure Debug mechanism.

To debug using the Secure Debug mechanism, the developer must perform the following steps:

1. Generate a private-public RSA developer key-pair and submit the generated public key to the OEM or ICV.For information on generating a private-public RSA key-pair using OpenSSL, see *Appendix A Random key generation* on page Appx-A-84.
2. Obtain an enabler debug certificate from the OEM or ICV, including the generated public key, allowed domain mask, and requested locking mask.
3. Extract the `SOC_ID` of the device.

   Note

   - The OEM or ICV must provide a method of retrieving the `SOC_ID` of the target device, the result of `CC_BsvSocIDCompute`, without having to first enable debugging.One suggested implementation is to have the device ROM code compute the `SOC_ID` and place it in a specific location in the flash memory, where it can be accessed by the developer.
   - In all security LCSes other than Secure LCS, the `SOC_ID` is not needed, and is ignored by the device. It can have dummy value.

4. Use the developer debug certificate to extend the enabler debug certificate package to a full debug certificate package, by adding a developer debug certificate signed with the developer key, including the `SOC_ID` of the device, and its desired domain mask.For more information, see *5.3.2 developer debug certificate tool* on page 5-71.
5. Place the final debug certificate package in a designated location in the flash memory, where the boot ROM can look for it. Once a debug certificate is identified, the boot ROM reads it and uses `CC_BsvSecureDebugSet` to verify it.

Upon receiving the developer key-pair, the OEM or ICV must produce an enabler debug certificate that signs the public key of the developer, and delivers it to the developer. According to the SD certificate scheme, this requires:

1. For a three-level SD certificate scheme only: running the offline tools and using the resulting key certificate as input for step 2. For more information, see *5.2 Secure Boot PC tools* on page 5-66
2. Running the developer debug certificate to create a signed enabler debug certificate, used to debug a given set of secure domains. For more information, see *5.3.1 enabler debug certificate tool* on page 5-70.

The OEM or ICV can also use the Secure Debug mechanism for entry into RMA LCS, in which case:

The enabler debug certificate:

Holds the RMA LCS indication instead of the debug mask.

The developer debug certificate:

Signed by the OEM or ICV and not by the developer. It is used only for a specific `SOC_ID`, and its debug mask is ignored.

For more information, see *RMA LCS entry sequence* on page 4-57.

Chapter 5
Offline tools

This chapter describes CryptoCell-312 PC (offline) tools.

It contains the following sections:

5.1 Overview of offline tools

The offline tools are python scripts used on a Host Linux machine, to generate Secure Boot, Secure Debug certificates, and asset packages.

These python scripts call a C library that wraps the required OpenSSL functionality.

The python scrits that create certificates receive a configuration file (`<cfg_file>`) as a parameter. This configuration file details the parameters used in the generated certificate. Examples of such a configuration are provided in the Secure Boot services release package under `utils/src/cc3x_boot_cert/example`.

For information on building the offline tools, see *2.4 Build environment offline tools* on page 2-25.

5.2 Secure Boot PC tools

The Secure Boot solution includes the key certificate tool, the content certificate tool, and the *Hbk* generation tool.

The key certificate tool:
Generates a key certificate.

The content certificate tool:
Generates a content certificate.

The *Hbk* generation tool:
Calculates the *Hbk* used for certificate verification and provisioning.

This section contains the following subsections:

- *5.2.1 Key certificate tool* on page 5-66.
- *5.2.2 Content certificate tool* on page 5-67.
- *5.2.3 Hbk generation tool* on page 5-68.

5.2.1 Key certificate tool

The `cert_key_util.py` tool generates a key certificate. All parameters are supplied through a configuration file (*<cfg_file>*).

```
Prompt> cert_key_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ".

Table 5-1 cert_key_util.py configuration file parameters

Parameter	I/O	Description
`[KEY-CFG]`	I	Mandatory header.
`cert-keypair`	I	File holding the RSA keypair for signing this certificate, in PEM format.
`cert-keypair-pwd`	I	Passphrase for the keypair file, in txt format. For enhanced security, this parameter can be omitted, and then the tool prompts for direct TTY input.
`hbk-id`	I	The ID of the OTP memory *Hbk* field, that the public key of this certificate is verified against: • 0 = 128bit *Hbk0* (ICV) • 1 = 128bit *Hbk1* (OEM) • 2 = 256bit *Hbk* The ROM code uses this field only if this certificate is the first certificate in: • A two-level SB certificate chain • A three-level SB certificate chain • A three-level Secure Debug chain However, since the tool is not aware of the usage of the certificate, this parameter cannot be omitted.
`nvcounter-val`	I	NV counter value: • 0..64 for the ICV counter. • 0..96 for the OEM counter. • 0..160 for the full counter, if OEM and ICV are a single entity.
`next-cert-pubkey`	I	File holding the RSA public key for signing the next certificate in the chain, either second Secure Boot key certificate, Secure Boot content certificate or enabler debug certificate, in PEM format.
`cert-pkg`	O	Key certificate package output file. Binary format.

5.2.2 Content certificate tool

The `cert_sb_content_util.py` tool generates a Secure Boot content certificate.

All parameters are supplied through a configuration file (*`cfg_file`*).

```
Prompt> cert_sb_content_util.py cfg_file
```

Note

A parameter can be omitted by leaving it empty, that is "param= ", or by omitting the entire field name.

Table 5-2 cert_sb_content_util.py configuration file parameters

Parameter	I/O	Description
`[CNT-CFG]`	I	Mandatory header.
`cert-keypair`	I	File holding the RSA keypair for signing this certificate, in PEM format.
`cert-keypair-pwd`	I	Passphrase for the keypair file, in txt format. For enhanced security, this parameter can be omitted, and then the tool prompts for direct TTY input.
`load-verify-scheme`	I	SW image verification scheme. Supported schemes are: **0** Load from Flash to memory, with full hash verification. **1** Full hash verification in Flash, with no loading to memory. **2** Full hash verification in memory, with no loading from Flash. **3** Loading from Flash to memory without verification. **Note** Loading without verification is not recommended and is not allowed in Secure LCS.
`crypto-type`	I	Cryptographic verification and decryption mode: **0** AES and hash are calculated on the plain image. **1** AES is calculated on the plain image, and the hash is calculated on the encrypted image. **Note** Arm recommendeds that you use the first mode.

Table 5-2 cert_sb_content_util.py configuration file parameters (continued)

Parameter	I/O	Description
`aes-ce-id`	I	The ID of the key that is used, in case encryption is required: **0** None. **1** *Kceicv*. **2** *Kce*. **Note** A certificate chain is signed by a specific entity: either ICV or OEM. The *Hbk* id and `aes-ce-id` must be compliant with this entity.
`aes-enc-key`	I	Text file containing the encryption key for the authenticated SW images (*Kce* or *Kceicv*). Comma-separated hex Bytes ("`0x12,0x34` …") ending with ". Optional: can be omitted if no encryption is needed.
`images-table`	I	Text file containing the list of SW image files to be processed. Each line refers to a single image, with the following parameters: `<image file name> <32b mem load addr> <32b flash store addr> <32b image max size> <encryption flag: 0 – not encrypted, 1 - encrypted>` **Note** • `0xffffffff` as an address signifies a non-existing address. • If using the 'Full hash verification in Flash, with no loading to memory' loading scheme, the memory address is `0xffffffff`. • If using the 'Full hash verification in memory, with no loading from Flash' loading scheme, the storage address is `0xffffffff`. • If encryption is used, an output file named `<image file name>_enc.bin` is created for each encrypted image.
`nvcounter-val`	I	The NV counter value: **0..64** The ICV counter. **0..96** The OEM counter. **0..160** The full counter, if OEM and ICV are a single entity.
`cert-pkg`	O	Content certificate package output file. Binary format.

5.2.3 *Hbk* generation tool

The `hbk_gen_util.py` tool is used for calculating the *Hbk* used for certificate verification and provisioning.

The output of the tool is in a format suitable for programming into the OTP memory of the device, as well as use by other tools.

To run the tool, open a command line in the python scripts directory and execute the following command:

```
prompt> hbk_gen_util.py -key <key file name> [-endian <output endianness>] [-hash_format
<hash format>]
```

Table 5-3 hbk_gen_util.py parameters

Parameters	Optional/ Mandatory	Description
Key file name	M	RSA public key filename in PEM format.

Output endianness	O	Sets the output endian type:
		`B`
		big endian.
		`L`
		little endian.
		Default value is little endian.
		Note
		If you set this value, you must also set the hash format value.

Hash format	O	Either of the following strings:
		`SHA256`
		Full SHA-256 hash. This is the single *Hbk* format. This is the default value if no parameter is provided.
		`SHA256_TRUNC`
		SHA-256 hash truncated to its leftmost 128-bit. This is the *Hbk0* or *Hbk1* format.
		Note
		If you set this value, you must also set the output endianness value.

The output of the tool is written to two files:

Table 5-4 hbk_gen_util.py output files

Output File	Description
`prim_key_hash.txt`	Holds the key hash.
`zero_bits_in_hash.txt`	Holds the number of zero bits in the hash result.

5.3 Secure Debug PC tools

The Secure Debug solution includes the enabler debug certificate tool and final debug certificate tool.

The enabler debug certificate tool:

Used by the OEM to create an enabler debug certificate package, with or without a key certificate. For more information, see *5.3.1 enabler debug certificate tool* on page 5-70

The final debug certificate tool:

Used by the final developer to generate a full signed debug certificate. For more information, see *5.3.2 developer debug certificate tool* on page 5-71

This section contains the following subsections:

5.3.1 enabler debug certificate tool

The `cert_dbg_enabler_util.py` tool is run by the OEM or ICV to create the enabler debug certificate package.

All parameters are supplied through a configuration file (*<cfg_file>*):

- Optional key certificate package, if there is a key certificate in the chain.
- OEM/ICV private key, in password-protected PEM format.
- Developer public key, in plain PEM format.
- Flag indicating in which security LCSes this debug certificate is valid.
- "Bounding mask" on the debug capabilities this developer is permitted to use.
- "Locking mask" of the debug capabilities.

This tool generates the "enabler debug certificate" part of a debug certificate chain, as described in *Secure Debug certificate creation* on page 3-42.

```
Prompt> cert_dbg_enabler_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ", or by omitting the entire field name.

Table 5-5 cert_dbg_enabler_util.py configuration file parameters

Parameter	I/O	Description
`[ENABLER-DBG-CFG]`	I	Internal non-configurable mandatory header.
`cert-keypair`	I	File holding the RSA keypair for signing this certificate, in PEM format.
`cert-keypair-pwd`	I	Passphrase for the keypair file, in txt format. **Note** For enhanced security, this parameter can be omitted, causing the tool to prompt for direct TTY input.
`lcs`	I	The LCS that this certificate is intended for: • 0: CM LCS • 1: DM LCS • 5: Secure LCS • 7: RMA LCS
`rma-mode`	I	Defines whether to use this certificate for entry into RMA LCS, by setting to non-zero value. Mandatory if debug mask is not defined. Cannot be defined together with debug-mask.

Table 5-5 cert_dbg_enabler_util.py configuration file parameters (continued)

Parameter	I/O	Description
`debug-mask`	I	The DCU mask allowed by the OEM. A 128-bit mask in 4x32-bit hex format: • `debug-mask`[0-31] = `0x00112233`. • `debug-mask`[32-63] = `0x44556677`. • `debug-mask`[64-95] = `0x8899AABB`. • `debug-mask`[96-127] = `0xCCDDEEFF`. This parameter is mandatory if `rma-mode` is not defined. Cannot be defined together with `rma-mode`.
`debug-lock`	I	Additional DCU lock mask required by the OEM. A 128-bit mask in 4x32-bit hex format: bits [31:0], [63:32], [95:64], and [127:96]. For example, `0x7000000f`. This parameter is mandatory if `rma-mode` is not defined. These bits are added to the OTP-based mask defined by the OEM and ICV.
`hbk-id`	I	The ID of the OTP memory *Hbk* field, that the public key of this certificate is verified against: • 0: 128-bit *Hbk0* • 1: 128-bit *Hbk1* • 2: 256-bit *Hbk* Must be defined for the two-level debug certificate scheme. This parameter is mandatory if `rma-mode` is defined.
`key-cert-pkg`	I	Key certificate package, generated by `x509_cert_key_util.py`, if the three-level debug certificate scheme is used. Must be defined for the three-level debug certificate scheme.
`next-cert-pubkey`	I	File holding the RSA public key for signing the next certificate in the chain (the developer debug certificate), in PEM format.
`cert-pkg`	O	enabler debug certificate package output file, in binary format. Includes the key certificate, if exists, and the enabler debug certificate.

5.3.2 developer debug certificate tool

The `cert_dbg_developer_util.py` tool is run by the developer, and generates a full signed debug certificate.

The full signed debug certificate includes the key certificate (if one exists), the enabler debug certificate, and the developer debug certificate.

1. Receives the following parameters, supplied through a configuration file (*<cfg_file>*):
   - Developer private key, in password-protected PEM format.
   - enabler debug certificate package.
   - Target device's `SOC_ID`.
   - Allowed debug capabilities.
2. Creates a developer debug certificate.
3. Creates the final debug certificate package, by bundling the developer debug certificate package with the enabler debug certificate package.

The developer debug certificate tool resides in the same directory as the enabler debug certificate tool. For more information, see *5.3.1 enabler debug certificate tool* on page 5-70.

```
Prompt> cert_dbg_developer_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ", or by omitting the entire field name.

Table 5-6 cert_dbg_developer_util.py configuration file parameters

Parameter	I/O	Description
`[DEVELOPER-DBG-CFG]`	I	Internal non-configurable mandatory header.
`cert-keypair`	I	File holding the RSA keypair for signing this certificate, in PEM format.
`cert-keypair-pwd`	I	Passphrase for the keypair file, in txt format. **Note** For enhanced security, this parameter can be omitted, causing the tool to prompt for direct TTY input.
`soc-id`	I	Binary file holding the 32-Byte *SOC_ID*.
`debug-mask`	I	The DCU mask allowed by the OEM. A 128-bit mask in 4x32-bit hex format: • `debug-mask`[0-31] = `0x00112233` • `debug-mask`[32-63] = `0x44556677` • `debug-mask`[64-95] = `0x8899AABB` • `debug-mask`[96-127] = `0xCCDDEEFF`. This parameter is mandatory if `rma-mode` is not defined. Cannot be defined together with `rma-mode`.
`enabler-cert-pkg`	I	The enabler debug certificate package.
`cert-pkg`	O	Output file of final certificate package. Includes the key certificate (if exists), the enabler debug certificate, and the developer debug certificate. Binary format.

5.4 Secure asset-provisioning PC tool

The CryptoCell-312 solution provides the asset-provisioning pc tool to support the provisioning flow.

For more information on the provisioning flow, see *3.7 Device provisioning* on page 3-48.

The asset-provisioning tool generates a secure asset package composed of the encrypted asset data and authentication information, by performing the following operations:

1. Derives an asset-specific key from the class key: *Kpicv* for the ICV, or *Kcp* for the OEM. This asset-specific key is compliant with *NIST SP 108: Recommendation for Key Derivation Using Pseudorandom Functions*.
2. Encrypts the supplied asset using AES-CCM.
3. Generates an encrypted asset object that contains the encrypted asset and its authentication tag.

All parameters to this tool are supplied through a configuration file (`<cfg_file>`).

```
prompt> asset_provisioning_util.py <cfg_file>
```

Table 5-7 asset_provisioning_util.py configuration file parameters

Parameter	I/O	Description
`[ASSET-PROV-CFG]`	I	Internal non-configurable mandatory header.
`key-filename`	I	File holding the encrypted ICV key (*Kpicv* or *Kcp*).
`keypwd-filename`	I	Passphrase for the key file, in txt format. **Note** For enhanced security, this parameter can be omitted, causing the tool to prompt for direct TTY input.
`asset-id`	I	A 32-bit word identifying the asset in hex format. For example, `0x7000000f`.
`asset-filename`	I	Filename of the asset data in a binary format. Data size must be <= 512 and a multiple of 16 Bytes.
`asset-pkg`	O	Filename for placing the output asset package. If not specified, the `asset_pkg.bin` filename is used.

5.5 ICV factory PC tool

The ICV factory PC tool is a python script located under `utils/src/cmpu_asset_pkg_util/`.

After compilation, the `cmpu_asset_pkg_util.py` script is copied to `utils/bin/`, and must be executed from there.

The `cmpu_asset_pkg_util.py` tool generates an encrypted secret: *Kceicv* or *Kpicv*. The tool must be executed for each secret that needs to be encrypted.

All parameters are supplied through a configuration file (*<cfg_file>*).

```
Prompt> cmpu_asset_pkg_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ".

Table 5-8 cmpu_asset_pkg_util.py configuration file parameters

Parameter	I/O	Description
`[CMPU-ASSET-CFG]`	I	Mandatory header.
`asset-type`	I	Defines the asset type: *Kpicv* or *Kceicv*.
`unique-data`	I	File holding the 16-Byte unique user-data in binary format. We recommend using *Hbk0*, if it exists.
`key-filemame`	I	File holding the encrypted *Krtl* in binary format.
`keypwd-filename`	I	Passphrase for the *Krtl* file in txt format. For enhanced security, this parameter can be omitted for the tool to prompt for direct TTY input.
`asset-filename`	I	The secret to be encrypted in binary format.
`pkg-filename`	O	Package output file in binary format.

5.6 OEM factory PC tools

The OEM factory PC tools are used to encrypt all OEM or ICV secrets using a key derived from the *Krtl*, which is an ICV secret.

The main purpose of the OEM factory tool is to allow the OEM to encrypt its secrets, OEM platform keys: *Kcp* or *Kce*, and send them to the manufacturing process as encrypted data. For this purpose, CryptoCell-312 provides the following PC tools for the OEM:

- The OEM key request generation tool
- The ICV key response tool
- The OEM asset packaging tool

For more information on the OEM factory tool, see *6.3 OEM factory tool* on page 6-82.

The following diagram demonstrates the platform-key encryption flow using these tools:

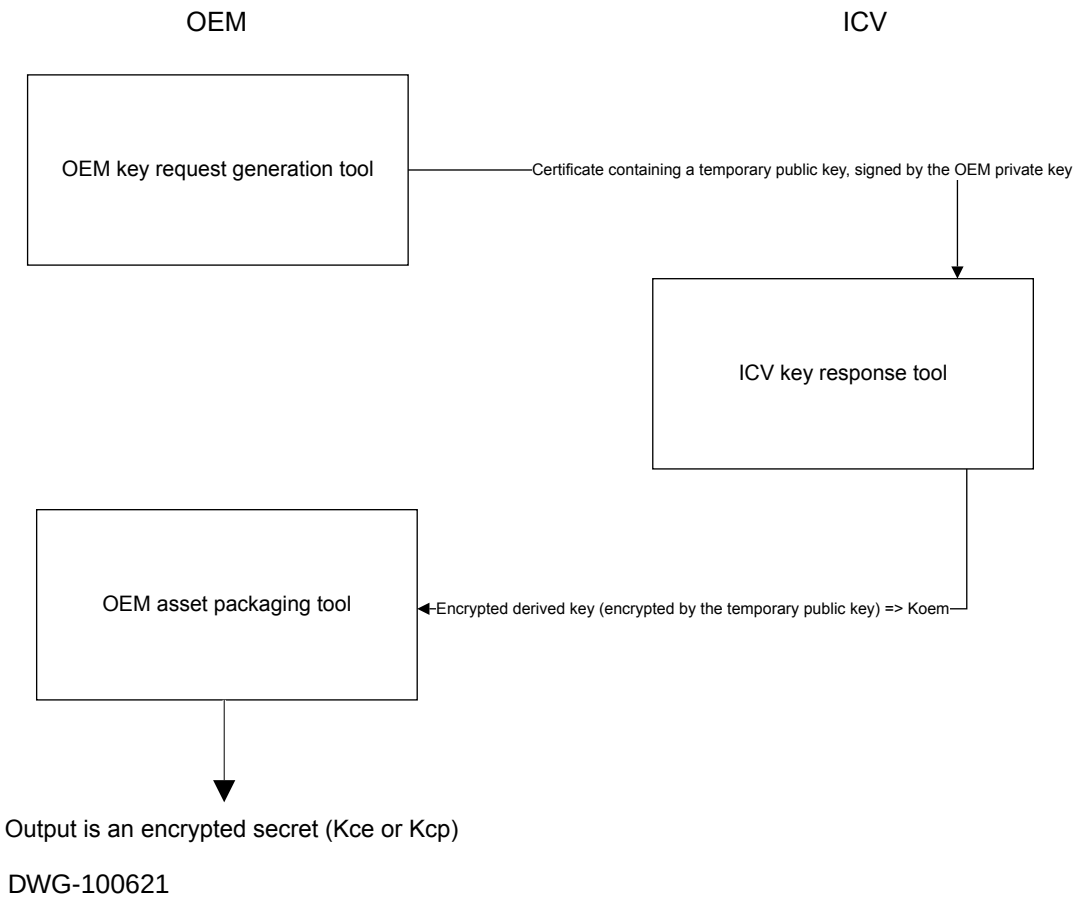

Figure 5-1 Platform-key encryption flow

This section contains the following subsections:

5.6.1 OEM key request generation tool

This tool issues a request to the ICV for a key to encrypt OEM assets.

The OEM cannot access the *Krtl* directly, as it is an ICV asset. The *Krtl* is the root for all ICV and OEM factory secrets.

The output of this tool is a request file in certificate format. This file contains a temporary RSA public key that is used by the ICV to encrypt the derived key response.

The OEM key request generation tool is a python script located under `utils/src/dmpu_asset_pkg_util/oem_key_request/`.

After compilation, the `dmpu_oem_key_request_util.py` script is copied to `utils/bin/`, and must be executed from there.

All parameters are supplied through a configuration file (*<cfg_file>*).

```
Prompt> dmpu_oem_key_request_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ".

Table 5-9 dmpu_oem_key_request_util.py configuration file parameters

Parameter	I/O	Description
`[DMPU-OEM-KEY-REQ-CFG]`	I	Mandatory header.
`oem-main-keypair`	I	File holding the RSA key-pair of the OEM, used to generate the *Hbk*. In PEM format.
`oem-main-keypwd`	I	Passphrase for the key-pair file of the OEM, in txt format. For enhanced security, this parameter can be omitted, for the tool to prompt for direct TTY input.
`oem-enc-pubkey`	I	File holding a temporary RSA public key. This file is used by the ICV to encrypt the response with. **Important** This key must be different to the main OEM key, to avoid key hygiene violation, thst is, a key that is used for different operations.
`oem-cert-pkg`	O	Certificate output file.

5.6.2 ICV key response tool

This tool returns the derived *Koem* key to the OEM.

The *Koem* is a derivation based on the *Krtl* and the *Hbk* of the OEM. It is encrypted with the temporary public key that the OEM sent in the request certificate.

For more information on the request certificate, see *5.6.1 OEM key request generation tool* on page 5-75.

The OEM decrypts the returned *Koem* to derive the key used to encrypt the secrets of the OEM.

The ICV key response tool is a python script located under `utils/src/dmpu_asset_pkg_util/icv_key_response/`.

After compilation, the `dmpu_icv_key_response_util.py` script is copied to `utils/bin/`, and must be executed from there.

All parameters are supplied through a configuration file (*<cfg_file>*).

```
Prompt> dmpu_icv_key_response_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ".

Table 5-10 dmpu_icv_key_response_util.py configuration file parameters

Parameter	I/O	Description
`[DMPU-ICV-KEY-RES-CFG]`	I	Mandatory header.
`oem-cert-pkg`	I	OEM key request certificate package output file, in binary format.
`key-filename`	I	File holding the encrypted *Krtl*, in binary format.
`keypwd-filename`	I	Passphrase for the *Krtl* file, in txt format. For enhanced security, this parameter can be omitted, for the tool to prompt for direct TTY input.
`icv-enc-oem-key`	O	Output file containing the encrypted OEM key.

5.6.3 OEM asset-packaging tool

This tool returns encrypted secrets: *Kce* or *Kcp*.

The OEM asset-packaging tool is a python script located under `utils/src/dmpu_asset_pkg_util/oem_asset_package/`. After compilation, the `dmpu_oem_asset_pkg_util.py` script is copied to `utils/bin/`, and must be executed from there.

All parameters are supplied through a configuration file (*<cfg_file>*).

```
Prompt> dmpu_oem_asset_pkg_util.py <cfg_file>
```

Note

A parameter can be omitted by leaving it empty, that is "param= ".

Table 5-11 dmpu_oem_asset_pkg_util.py configuration file parameters

Parameter	I/O	Description
`[DMPU-OEM-ASSET-CFG]`	I	Mandatory header.
`asset-type`	I	Defines the asset type: encryption key (*Kce*) or provisioning key (*Kcp*).
`icv-enc-oem-key`	I	File containing the OEM key (*Koem*) received from the ICV as the output of *5.6.2 ICV key response tool* on page 5-76.
`oem-enc-keypair`	I	File holding the temporary RSA keypair for decrypting the *Koem*, in PEM format.
`oem-enc-keypwd`	I	Passphrase for the OEM encryption key-pair file, in txt format. For enhanced security, this parameter can be omitted, for the tool to prompt for direct TTY input.
`asset-filename`	O	The asset (secret) to create this package for: *Kce* or *Kcp*.
`oem-cert-pkg`	O	The output package file.

Chapter 6
CryptoCell-312 production-line tools

This chapter describes CryptoCell-312 production-line (device-run) tools.

It contains the following sections:

6.1 Production-line tools overview

CryptoCell-312 includes tools that allow the ICV or OEM to load and burn the device secrets during production.

There are two basic tools, supplied as libraries, one for the chip manufacturer (ICV) and one for the OEM.

CryptoCell-312 includes the following production-line tools:

The ICV factory tool:

A combination of a PC tool and a library, used to program the chip-manufacturer keys, as well as information in the OTP memory of the device.

For more information on the ICV factory tool, see *6.2 ICV factory tool* on page 6-80. For more information on the ICV factory PC tool, see *5.5 ICV factory PC tool* on page 5-74.

The OEM factory tool:

A combination of three PC tools and a library that enables the burning of the OEM secrets in a secure way.

For more information on the OEM factory tool, see *6.3 OEM factory tool* on page 6-82. For more information on the OEM factory PC tools, see *5.6 OEM factory PC tools* on page 5-75.

6.2 ICV factory tool

The ICV factory tool is a library that performs the initial population (programming) of the basic ICV secrets of the device in the OTP-NVM of the chip.

This tool works only in CM LCS, and implicitly transitions the chip from the CM LCS to the DM LCS.

You must create an application that calls this library and loads it to the device and executes it during the manufacturing process.

The following are ICV keys and assets that the tool burns to the OTP:

The unique device key, *HUK*:
The library uses the CryptoCell-312 TRNG and dedicated DRBG driver to create the *HUK*, and therefore needs the TRNG characterization information. This information is maintained in a header file under `shared/include/trng`. This file must be updated before the library is compiled and executed. For more information on the *HUK*, see *3.1.1 Device key* on page 3-27.For more information on TRNG characterization, see *Arm® TrustZone® TRNG Characterization Application Note*.

The ICV class keys, *Kpicv* and *Kceicv* (optional):
These keys can be in either plain text or encrypted. To encrypt the keys, see *5.5 ICV factory PC tool* on page 5-74. If the keys are not used, a specific unused bit is burned. For more information about the unused bit, see *3.2 OTP memory* on page 3-29.

The ICV Root of Trust, *Hbk0*:
This value is only mandatory if there are two different entities in the system, that is both an ICV and an OEM.

ICV SW minimal version:
Initial SW version value (the revocation counter). This is an optional input.

ICV DCU lock bits:
The lock bits mask of the ICV. This is an optional input.

General configuration bits in the general-purpose flag word:
This is an optional input.

Note

The ICV factory tool must be run even if there is only a single entity in the system: ICV or OEM. If the ICV secrets are not needed, you can use the `unneeded` flag. For more information, see *CryptoCell production-library APIs* in *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

This section contains the following subsections:

6.2.1 Building the ICV factory tool

To build the ICV factory tool, the user must take the `libcmpu.a` library and create an application that calls it.

The CryptoCell-312 release package includes a reference application to illustrate what this application should include and do.

6.2.2 ICV factory tool loading methods

This section describes two possible methods for loading the ICV factory tool to the on-chip RAM before execution.

Loading using TIC

This section details the suggested method for loading the tool using TIC.

To load the tool using TIC, perform the following steps:

1. Modify the Host boot ROM of the chip to execute the tool when it identifies a predefined TIC signature in a predefined SRAM address.
2. Use production-floor TIC to place the tool in the device Host processor executable memory and trigger its execution by the Host processor, as follows:
   a. TIC holds Host processor in reset.
   b. TIC loads the tool executable into the SRAM of the device.
   c. TIC places a signature, recommended to include checksum and CRC to avoid false signature identification, in a predefined SRAM location.
   d. TIC releases Host processor from reset.
   e. Host boot ROM searches for the TIC signature, and executes the tool.
   f. TIC clears the SRAM.
   g. TIC resets the Host processor again.

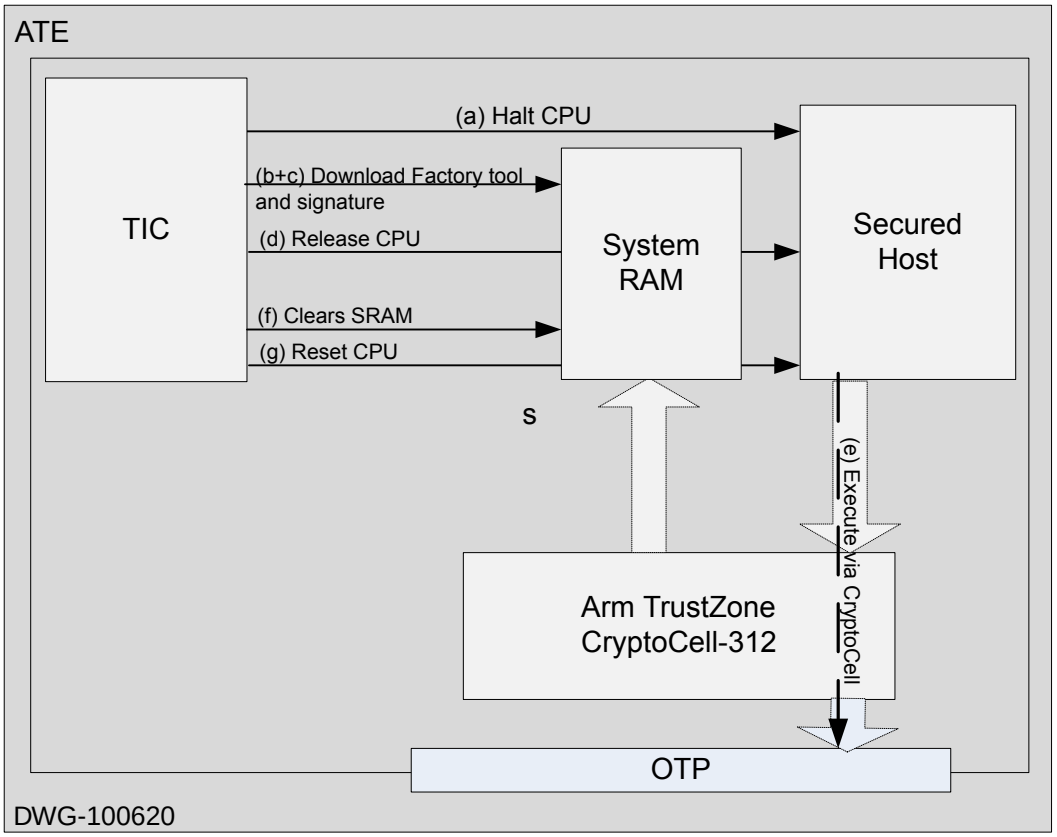

Figure 6-1 Tool production flow

Loading using SoC flash

This section details the suggested method for loading the tool using flash memory.

To load the tool using flash memory:

1. Burn the tool to an off-chip NOR flash drive.
2. Use TIC to make the Host boot from the NOR flash and execute the tool from it.

6.3 OEM factory tool

The OEM factory tool is a library that burns the OEM secrets and assets to the device in DM LCS.

You must create an application that calls this library and loads it to the device during the manufacturing process, to be executed by the secure Host. The OEM assets that are burned to the OTP using this tool can be in either plain text or encrypted.

The following are OEM keys and assets that the tool burns to the OTP:

The OEM class keys, *Kcp* and *Kce* (optional):
: These keys can be in either plain text or encrypted. To encrypt the keys, see *5.6 OEM factory PC tools* on page 5-75. If the keys are not used, a specific unused bit is burned. For more information about the unused bit, see *3.2 OTP memory* on page 3-29.

The OEM Root of Trust, *Hbk1* or *Hbk*:
: If there are two different entities in the system, that is both an ICV and an OEM, its size is half the size of SHA-256. Otherwise, it is the full *Hbk*.

OEM SW minimal version:
: The initial SW version value (the revocation counter).

OEM DCU lock bits:
: The lock bits mask of the OEM. This is an optional input.

Note

The OEM factory tool must be run even if there is only a single entity in the system: ICV or OEM. If the OEM secrets are not needed, you may use the `unneeded` flag. For more information, see *CryptoCell production-library APIs* in *Arm® TrustZone® CryptoCell-312 Software Developers Manual*.

This section contains the following subsections:

6.3.1 Building the OEM factory tool

To build the OEM factory tool, the user must take the `libdmpu.a` library and create an application that calls it.

The CryptoCell-312 release package includes a reference application to illustrate what this application should include and do.

6.3.2 OEM factory tool loading methods

This section describes optional methods for loading the OEM factory tool to on-chip RAM before execution.

Loading using TIC

This section details the suggested method for loading the tool using TIC.

To load the tool using TIC, perform the following steps:

1. Modify the Host boot ROM of the chip to execute the tool when it identifies a predefined TIC signature in a predefined SRAM address.
2. Use production-floor TIC to place the tool in the device Host processor executable memory and trigger its execution by the Host processor, as follows:
   a. TIC holds Host processor in reset.
   b. TIC loads the tool executable into the SRAM of the device.
   c. TIC places a signature, recommended to include checksum and CRC to avoid false signature identification, in a predefined SRAM location.
   d. TIC releases Host processor from reset.
   e. Host boot ROM searches for the TIC signature, and executes the tool.

f. TIC clears the SRAM.
g. TIC resets the Host processor again.

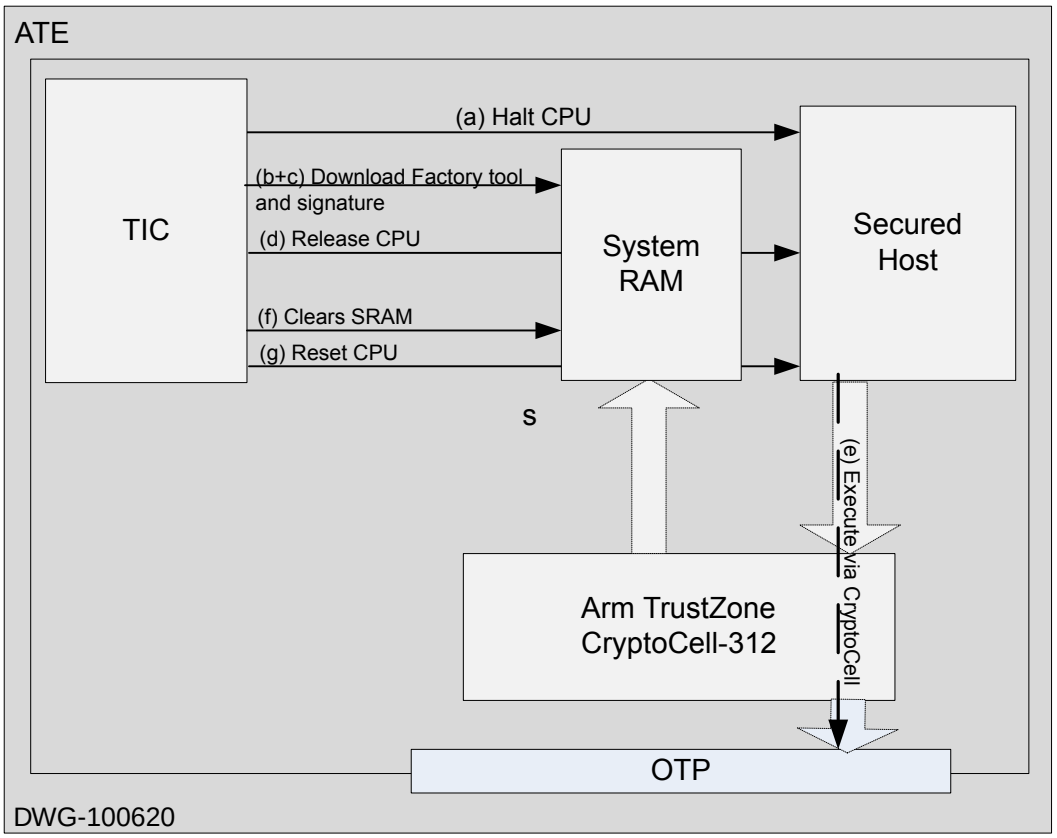

Figure 6-2 Tool production flow

Loading using SoC flash

This section details the suggested method for loading the tool using flash memory.

To load the tool using flash memory:
1. Burn the tool to an off-chip NOR flash drive.
2. Use TIC to make the Host boot from the NOR flash and execute the tool from it.

Appendix A
Random key generation

This appendix details how to generate a random key, and provides output examples.

It contains the following section:

A.1 Generating a random key

This section details the process for generating a random key.

To create a random RSA private/public key pair in PEM encrypted format, using OpenSSL, perform the following steps:

1. Execute `openssl genrsa -aes256 -out key.pem 3072`
2. Insert a passphrase when required.

The output of the `genrsa` command resembles the following:

```
Private-Key: (3072 bit)
                modulus:
                00:cc:84:9e:d4:2e:9f:b6:8b:a8:44:a6:a6:e7:6b:
                c0:d3:cb:21:74:8c:d6:8f:ef:eb:83:a1:9e:79:2f:
                c0:f0:94:87:5c:e2:50:26:8c:75:1f:68:4b:ee:8c:
                d2:c1:59:09:57:06:bd:c3:b9:61:d7:f7:d5:1a:43:
                69:4d:71:99:1f:99:93:5e:05:54:aa:f4:56:ca:f6:
                57:db:78:7b:ae:d0:28:5a:14:81:45:bd:83:3e:c1:
                5e:12:9d:a5:bb:7a:2d:2e:7d:dd:f2:f0:15:4b:49:
                f2:85:9d:8c:12:cd:50:24:4a:b9:2f:bb:b9:dc:16:
                49:4b:fd:5c:8d:0e:bc:e0:95:df:f1:e1:8c:81:40:
                b3:03:49:60:98:6f:2b:bc:d7:c6:a7:04:dc:99:03:
                d1:ae:32:e8:9b:12:e1:dc:1d:9d:f6:d1:23:50:ac:
                8b:7a:20:97:53:e6:ef:28:55:50:51:f3:91:70:c6:
                30:0a:4c:40:23:57:f8:11:70:d1:82:5c:5f:e8:a8:
                e7:b7:7d:74:f3:fb:e6:dd:13:c4:0f:92:0d:e0:29:
                66:ee:c2:cb:95:58:af:b3:bc:af:ee:8d:26:cf:71:
                e9:3a:ac:96:b3:20:b9:70:d7:e3:4a:0d:c3:0a:02:
                81:67:b1:14:52:e8:51:ee:b9:60:83:de:28:76:32:
                19:82:a4:04:76:22:88:b5:96:15:99:d9:59:e1:4d:
                fc:59:6c:93:13:ab:bf:b2:5f:14:72:8a:0d:67:2a:
                04:56:90:25:22:10:a0:c3:3b:d5:f2:23:50:79:47:
                15:90:3e:e7:d5:60:cd:e8:be:2c:5f:83:40:86:52:
                52:b5:ab:a2:4c:fc:72:5b:aa:97:25:77:54:67:9d:
                ef:0a:7d:37:dd:b7:11:f7:c1:0a:1a:f9:e4:92:3f:
                c9:fa:e0:6d:be:ca:f2:2f:c8:18:89:70:fe:4b:c8:
                b3:b8:42:cb:58:f6:ee:0f:22:33:38:24:27:51:06:
                0b:b8:3e:17:db:c3:58:58:90:a1
                publicExponent: 65537 (0x10001)
                privateExponent:
                00:b4:60:6f:58:c2:3a:38:9c:d9:ee:9a:70:f0:00:
                78:14:13:be:d0:f5:7a:33:7e:ad:df:ed:86:40:69:
                de:a8:10:c7:86:23:4f:ff:7f:b7:b8:d8:9a:c7:ad:
                4a:20:1e:ef:fe:e2:42:31:c2:69:ca:39:99:a8:cd:
                35:ad:88:f8:3d:07:8d:12:ca:6e:cf:4c:2a:d4:5d:
                0a:45:d5:f0:77:d0:bd:16:1d:a9:9a:21:b8:04:7c:
                35:05:04:27:6c:bf:27:e4:2e:3c:b7:8c:09:ab:da:
                6f:5c:6b:04:64:7c:d2:27:00:04:6e:56:0a:69:52:
                fa:98:5b:bf:e9:e4:ca:ae:e5:60:9b:0b:8b:89:5f:
                cf:c2:a1:4f:e1:81:83:38:cd:b2:36:12:32:a1:7d:
                dd:cb:de:b8:8a:34:d0:20:fb:ba:1b:3a:7f:04:22:
                f9:ec:df:9b:b7:f3:20:91:19:15:7b:01:f0:0f:5f:
                ac:75:18:a5:24:c6:ec:52:2d:87:8a:8a:fe:b2:bd:
                2b:9c:1e:35:02:a2:ba:49:01:b5:4d:a1:73:f7:53:
                5e:80:91:70:8a:ea:fd:a4:90:23:44:3f:ef:f1:00:
                dd:68:f9:ec:d5:eb:d8:c9:88:89:b5:c1:e7:3f:36:
                f9:62:ca:f4:d4:00:c9:05:93:83:50:5e:7f:8a:00:
                04:03:6f:36:c4:b3:fb:54:58:ec:3e:ed:c8:67:11:
                48:3d:e9:af:ee:50:79:f5:08:5c:6a:ad:30:b6:28:
                2b:08:0d:93:9c:95:fe:12:42:27:54:08:e1:b5:86:
                25:45:5c:66:f1:f1:0b:30:b1:4a:89:04:fc:b2:ed:
                3b:f2:ba:f3:3e:9c:59:0b:bd:a8:6d:3f:ad:04:60:
                85:96:9a:67:bf:f9:64:41:25:80:9e:9b:50:33:db:
                ee:1f:1b:7c:ba:16:3e:d3:19:63:b2:f2:c2:af:c6:
                1e:e4:a8:4a:09:63:6a:a5:31:27:25:0a:40:a6:be:
                d2:7e:0f:40:98:98:72:01:eb:b5prime1:
                ...
                prime2:
                ...
                exponent1:
                ...
                exponent2:
                ...
                coefficient:
                ...
```

To later decode this key pair:

1. Execute `openssl rsa -in key.pem -text`.
2. Insert the passphrase when required.

To extract the public key from the key pair file, execute `openssl rsa -in keypair.pem -out pubkey.pem -pubout`. No password is needed for public key extraction.

Appendix B
CryptoCell-312 supported algorithms

This appendix details algorithms supported by CryptoCell-312.

For more information, see *1.2 Compliance* on page 1-17.

It contains the following section:

B.1 Supported algorithms

This section lists the basic algorithms and modes that CryptoCell-312 supports.

Table B-1 Supported algorithms

Algorithm	Mode	Key sizes in bits
AES	ECB, CBC, CTR, OFB, XTS, CBC-CTS mode 1	128, 192, 256
AES MAC	CMAC, CBC-MAC, XCBC-MAC	128, 192, 256
AES-CCM	N/A	128, 192, 256
DES or TDES	ECB, CBC	64, 128, 192
Hash	SHA1, SHA2 (SHA224, SHA256, SHA384, SHA512), MD5	N/A
HMAC	SHA1, SHA2 (SHA224, SHA256, SHA384, SHA512), MD5	N/A
DRBG	*NIST SP 90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators – App C.* AES_DRBG	256
TRNG	Full Entropy, compliant with *Arm® TrustZone® CryptoCell-312 FIPS 140-2 Non-Proprietary Security Policy*, *BSI AIS-31: Functionality Classes and Evaluation Methodology for True Random Number Generators*, and *NIST SP 90B: DRAFT Recommendation for the Entropy Sources Used for Random Bit Generation*	N/A
RSA PKCS#1	Encryption, signature schemes: • *Public-Key Cryptography Standards (PKCS) #1 v2.1: RSA Cryptography Specifications* • *Public-Key Cryptography Standards (PKCS) #1 v1.5: RSA Encryption* Using SHA1, SHA224, SHA256, SHA384, SHA512	2048, 3072, 4096
RSA key generation	N/A	2048, 3072
Diffie-hellman	• *ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography* • *Public-Key Cryptography Standards (PKCS) #3: Diffie Hellman Key Agreement Standard*	1024, 2048, 3072
Key derivation	• *ANSI X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography* • *NIST SP 108: Recommendation for Key Derivation Using Pseudorandom Functions*	N/A
Elliptic curve digital signature	Digital signature	N/A

Appendix C
Integration test

This appendix describes the CryptoCell-312 integration test.

You must implement a subset of a function to serve as an abstraction layer between the integration test and the operating system of your choice.

It contains the following sections:

C.1 Common integration tests

This section describes common CryptoCell-312 integration tests.

You must implement a subset of a function to serve as an abstraction layer between the integration test and the operating system of your choice.

This section contains the following subsections:

C.1.1 Platform HAL integration tests

Platform HAL is responsible for initializing the board. This might include mapping of addresses and toggling modules at boot time.

Test_HalPlatformInit

This function initializes platform, that is, maps CryptoCell-312 HW base address and environment HW base address in processMap.

```
uint32_t Test_HalPlatformInit(void)
```

Returns:

- 0: success.
- 1: failure.

Test_HalPlatformFree

This function unmaps CryptoCell-312 HW base address and environment HW base address in processMap.

```
void Test_HalPerformPowerOnReset(void)
```

Test_HalPerformPowerOnReset

This function performs PoR of CryptoCell-312, AO and environment registers.

```
void Test_HalPerformPowerOnReset(void)
```

Test_HalCheckLcs

This function reads the LCS register and verifies that the LCS value is correct.

```
uint32_t Test_HalCheckLcs(uint32_t nextLcs)
```

Returns:

- 0 on success or 1 on failure.

Table C-1 Test_PalMapAddr parameters

I/O	Parameter	Description
I	`nextLcs`	The address of the LCS register.
O	`??`	??

C.1.2 Address mapping integration tests

The main purpose of these tests is to map the physical address of CryptoCell-312 registers to the virtual address of the OS.

Test_PalMapAddr

This function maps a physical address to a virtual address.

```
void *Test_PalMapAddr(void *physAddr, void *startingAddr, const char *filename, size_t size,
uint8_t protAndFlagsBitMask)
```

When no memory management unit is present, and the access to the physical memory is straightforward, the mapping function returns the address of the memory-mapped CryptoCell registers.

Returns:

- A valid virtual address on success, or NULL on failure.

Table C-2 Test_PalMapAddr parameters

I/O	Parameter	Description
I	`physAddr`	The physical address.
I	`startingAddr`	The preferred static address for mapping.
I	`filename`	The filename, when using a file-based system. The `/dev` memory device path that enables access to memory.
I	`size`	The contents of a file mapping are initialized using size bytes starting at the `startingAddr` offset in the file.
I	`protAndFlagsBitMask`	Optional flags for permissions.

Test_PalUnmapAddr

This function unmaps the given virtual address.

```
void Test_PalUnmapAddr(void *virtAddr, size_t size)
```

Table C-3 Test_PalUnmapAddr parameters

I/O	Parameter	Description
I	`virtAddr`	The virtual address to unmap.
I	`size`	The size of memory to unmap.

C.1.3 Memory integration tests

The integration test only uses DMA-able continuous memory.

Test_PalDMAContigBufferAlloc

This function allocates a DMA-contiguous buffer, and returns its address.

```
void *Test_PalDMAContigBufferAlloc(size_t size);
```

Returns:

- An address of the allocated buffer.

Table C-4 Test_PalDMAContigBufferAlloc parameters

I/O	Parameter	Description
I	`buffSize`	The buffer size in Bytes.

Test_PalDMAContigBufferFree

This function frees resources previously allocated by `Test_PalDMAContigBufferAlloc()`.

```
void Test_PalDMAContigBufferFree(void *pvAddress)
```

Table C-5 Test_PalDMAContigBufferFree parameters

I/O	Parameter	Description
I	`pvAddress`	The address of the allocated buffer to free.

C.1.4 Thread integration tests

Test_PalThreadCreate

This function creates a thread.

```
ThreadHandle Test_PalThreadCreate(size_t stackSize,                                    void
          *(*threadFunc)(void *),                                  void *args,
          const char *threadName,                                  uint8_t
          nameLen, uint8_t DmaAble);
```

The user must call `Test_PalThreadDestroy()` to destroy the thread.

Returns:

- The `threadFunc` address on success. NULL on failure.

Table C-6 Test_PalThreadCreate parameters

I/O	Parameter	Description
I	`stackSize`	The stack size in bytes.
I	`threadFunc`	The thread function.
I	`args`	The input arguments for the thread function.
I	`threadName`	The name of the thread.
I	`nameLen`	The length of the thread.
I	`DmaAble`	Determines whether the stack should be DMA-able: • True - DMA-able. • False - not DMA-able.

Test_PalThreadDestroy

This function destroys a thread.

```
uint32_t Test_PalThreadDestroy(ThreadHandle threadHandle);
```

Returns:
- 0 on success. 1 on failure.

Table C-7 Test_PalThreadDestroy parameters

I/O	Parameter	Description
I	`threadHandle`	The thread structure.

Test_PalThreadJoin

This function waits for a thread to terminate (blocking).

```
uint32_t Test_PalThreadJoin(ThreadHandle threadHandle, void *threadRet)
```

If that thread has already terminated it returns immediately.

Returns:
- 0 on success. 1 on failure.

Table C-8 Test_PalThreadJoin parameters

I/O	Parameter	Description
I	`threadHandle`	The thread structure. Not in use for FreeRTOS.
I	`threadRet`	The status of the target thread.

C.1.5 Time integration tests

Implements time-sensitive functions that are based on the underlying operating system.

Test_PalDelay

This function suspends execution of the calling thread for microsecond intervals.

```
void Test_PalDelay(const uint32_t msec)
```

Table C-9 Test_PalDelay parameters

I/O	Parameter	Description
I	`msec`	The time to suspend execution, in microseconds.

C.2 Boot services integration tests

This section describes the CryptoCell-312 boot services integration tests.

You must implement a subset of a function to serve as an abstraction layer between the integration test and the operating system of your choice.

This section contains the following subsections:

C.2.1 Boot services register integration tests

The following tests check access to CryptoCell-312 boot services registers.

BSVIT_READ_REG

This function reads the register value from `offset`.

```
BSVIT_READ_REG(offset)
```

Returns:

- The value of the register.

Table C-10 BSVIT_READ_REG parameters

I/O	Parameter	Description
I	`offset`	The offset from the beginning of the register file.

BSVIT_WRITE_REG

This function writes the value set in `val` to register at `wordOffset`.

```
BSVIT_WRITE_REG(wordOffset, val)
```

Table C-11 BSVIT_WRITE_REG parameters

I/O	Parameter	Description
I	`wordOffset`	The offset of the register to overwrite.
O	`val`	The new value to write.

C.2.2 Boot services OTP integration tests

OTP implementation is partner-specific.

To run the integration test on an FPGA or simulation environment, you must have an implementation of the OTP module. The following functions must be adapted to your implementation.

BSVIT_WRITE_OTP

This function writes the value set in `val` to the OTP at `wordOffset`.

```
BSVIT_WRITE_OTP(wordOffset, val)
```

Table C-12 BSVIT_WRITE_OTP parameters

I/O	Parameter	Description
I	`wordOffset`	The offset of the OTP to overwrite.
O	`val`	The new value to write.

BSVIT_READ_OTP

This function reads the OTP value from `offset`.

```
BSVIT_READ_OTP(offset)
```

Returns:

- The value of the register.

Table C-13 BSVIT_READ_OTP parameters

I/O	Parameter	Description
I	`offset`	The offset from the beginning of the OTP file.

C.2.3 Boot services flash integration tests

The flash layer allows implementing flash-like behavior in systems and configurations that do not have physical flash modules.

bsvIt_flashInit

This function initiates the flash module.

```
BsvItError_t bsvIt_flashInit(size_t flashSize)
```

It must be called before other flash operations. This function initiates all that is needed to imitate flash operations.

Returns:

- `BSVIT_ERROR_OK` on success.
- `BSVIT_ERROR_FAIL` on failure.

Table C-14 bsvIt_flashInit parameters

I/O	Parameter	Description
I	`flashSize`	The size of the flash to initialize.

bsvIt_flashFinalize

This function closes a resource allocated for the Flash PAL module.

```
BsvItError_t bsvIt_flashFinalize(void)
```

It can be used for deallocation or a type of reset.

Returns:

- `BSVIT_ERROR_OK` on success.
- `BSVIT_ERROR_FAIL` on failure.

bsvIt_flashWrite

This function writes to flash at the offset set in `addr`.

```
BsvItError_t bsvIt_flashWrite(uint32_t addr, uint8_t* buff, size_t len)
```

Returns:

- `BSVIT_ERROR_OK` on success.
- `BSVIT_ERROR_FAIL` on failure.

Table C-15 bsvIt_flashWrite parameters

I/O	Parameter	Description
I	`addr`	The offset from the start of the flash.
I	`buf`	The buffer to write to flash.
I	`len`	The length of data to write to flash.

bsvIt_flashRead

This function reads from flash at the `addr` address, and writes to the `buf` buffer.

```
BsvItError_t bsvIt_flashRead(uint32_t addr, uint8_t* buff, size_t len)
```

Returns:

- `BSVIT_ERROR_OK` on success.
- `BSVIT_ERROR_FAIL` on failure.

Table C-16 bsvIt_flashRead parameters

I/O	Parameter	Description
I	`addr`	The offset from the start of the flash.
O	`buf`	The buffer to fill with read data.
I	`len`	The length of data to read from flash.

C.2.4 Boot services logging integration tests

Log entries are embedded in the integration test, and are intended to debug and output the test result to your chosen output.

BSVIT_PRINT

This function prints a log entry.

```
BSVIT_PRINT(format, ...)
```

Table C-17 BSVIT_PRINT parameters

I/O	Parameter	Description
I	`format`	The preferred output format.

BSVIT_TEST_START

This function starts the test.

```
BSVIT_TEST_START(testName)
```

It is called at the beginning of every test. You can configure it to print a formatted line that indicates the test has started.

Table C-18 BSVIT_TEST_START parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

BSVIT_TEST_RESULT

This function returns the test result.

```
BSVIT_TEST_RESULT(testName)
```

It is called at the end of every test. You can configure it to print a formatted line that indicates if the test has ended successfully.

Table C-19 BSVIT_TEST_RESULT parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

BSVIT_PRINT_ERROR

This function prints an error message to log.

```
BSVIT_PRINT_ERROR(format, ...)
```

Table C-20 BSVIT_PRINT_ERROR parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

BSVIT_PRINT_DBG

This function prints a debug message.

```
BSVIT_PRINT_DBG(format, ...)
```

It is skipped unless compiled with `TEST_DEBUG`.

Table C-21 BSVIT_PRINT_DBG parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

C.3 Runtime integration tests

This section describes the CryptoCell-312 runtime integration tests.

This section contains the following subsections:

C.3.1 Runtime register integration tests

The following tests check access to CryptoCell-312 runtime registers.

RUNIT_READ_REG

This function reads the register value from `offset`.

```
RUNIT_READ_REG(offset)
```

Returns:

- The value of the register.

Table C-22 RUNIT_READ_REG parameters

I/O	Parameter	Description
I	`offset`	The offset from the beginning of the register file.

RUNIT_WRITE_REG

This function writes the value set in `val` to register at `wordOffset`.

```
RUNIT_WRITE_REG(wordOffset, val)
```

Table C-23 RUNIT_WRITE_REG parameters

I/O	Parameter	Description
I	`wordOffset`	The offset of the register to overwrite.
O	`val`	The new value to write.

C.3.2 Runtime OTP integration tests

OTP implementation is partner-specific.

To run the integration test on an FPGA or simulation environment, you must have an implementation of the OTP module. The following functions must be adapted to your implementation.

RUNIT_WRITE_OTP

This function writes the value set in `val` to the OTP at `wordOffset`.

```
RUNIT_WRITE_OTP(wordOffset, val)
```

Table C-24 RUNIT_WRITE_OTP parameters

I/O	Parameter	Description
I	`wordOffset`	The offset of the OTP to overwrite.
O	`val`	The new value to write.

RUNIT_READ_OTP

This function reads the OTP value from `offset`.

```
RUNIT_READ_OTP(offset)
```

Returns:

- The value of the register.

Table C-25 RUNIT_READ_OTP parameters

I/O	Parameter	Description
I	`offset`	The offset from the beginning of the OTP file.

C.3.3 Runtime flash integration tests

The flash layer allows implementing flash-like behavior in systems and configurations that do not have physical flash modules.

runIt_flashInit

This function initiates the flash module.

```
RsvItError_t runIt_flashInit(size_t flashSize)
```

It must be called before other flash operations. This function initiates all that is needed to imitate flash operations.

Returns:

- `RUNIT_ERROR_OK` on success.
- `RUNIT_ERROR_FAIL` on failure.

Table C-26 runIt_flashInit parameters

I/O	Parameter	Description
I	`flashSize`	The size of the flash to initialize.

runIt_flashFinalize

This function closes a resource allocated for the Flash PAL module.

```
RsvItError_t runIt_flashFinalize(void)
```

It can be used for deallocation or a type of reset.

Returns:

- `RUNIT_ERROR_OK` on success.
- `RUNIT_ERROR_FAIL` on failure.

runIt_flashWrite

This function writes to flash at the offset set in `addr`.

```
RsvItError_t runIt_flashWrite(uint32_t addr, uint8_t* buff, size_t len)
```

Returns:

- `RUNIT_ERROR_OK` on success.
- `RUNIT_ERROR_FAIL` on failure.

Table C-27 bsvIt_flashWrite parameters

I/O	Parameter	Description
I	`addr`	The offset from the start of the flash.
I	`buf`	The buffer to write to flash.
I	`len`	The length of data to write to flash.

runIt_flashRead

This function reads from flash at the `addr` address, and writes to the `buf` buffer.

```
RsvItError_t runIt_flashRead(uint32_t addr, uint8_t* buff, size_t len)
```

Returns:

- `RUNIT_ERROR_OK` on success.
- `RUNIT_ERROR_FAIL` on failure.

Table C-28 bsvIt_flashRead parameters

I/O	Parameter	Description
I	`addr`	The offset from the start of the flash.
O	`buf`	The buffer to fill with read data.
I	`len`	The length of data to read from flash.

C.3.4 Runtime logging integration tests

Log entries are embedded in the integration test, and are intended to debug and output the test result to your chosen output.

RUNIT_PRINT

This function prints a log entry.

```
RUNIT_PRINT(format, ...)
```

Table C-29 RUNIT_PRINT parameters

I/O	Parameter	Description
I	`format`	The preferred output format.

RUNIT_PRINT_HEADER

This function prints the header of the integration test.

```
RUNIT_PRINT_HEADER()
```

RUNIT_PRINT_DBG

This function prints a debug message.

```
RUNIT_PRINT_DBG(format, ...)
```

It is skipped unless compiled with `TEST_DEBUG`.

Table C-30 RUNIT_PRINT_DBG parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

RUNIT_PRINT_ERROR

This function prints an error message to log.

```
RUNIT_PRINT_ERROR(format, ...)
```

Table C-31 RUNIT_PRINT_ERROR parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

RUNIT_PRINT_BUF

This function prints the buffer of length `_size` with a `_label` label.

```
RUNIT_PRINT_BUF(_buf, _size, _label)
```

Table C-32 RUNIT_PRINT parameters

I/O	Parameter	Description
I	`_buf`	The buffer to print.
I	`_size`	The size of the buffer to print.
I	`_label`	The label to show in the printout.

RUNIT_TEST_START

This function starts the test.

```
RUNIT_TEST_START(testName)
```

It is called at the beginning of every test. You can configure it to print a formatted line that indicates the test has started.

Table C-33 RUNIT_TEST_START parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

RUNIT_TEST_RESULT

This function returns the test result.

```
RUNIT_TEST_RESULT(testName)
```

It is called at the end of every test. You can configure it to print a formatted line that indicates if the test has ended successfully.

Table C-34 RUNIT_TEST_RESULT parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

RUNIT_SUB_TEST_START

This function starts the sub-test.

```
RUNIT_SUB_TEST_START(_testName)
```

It is called at the beginning of every sub-test. You can configure it to print a formatted line that indicates the sub-test has started.

Table C-35 RUNIT_SUB_TEST_START parameters

I/O	Parameter	Description
I	`_testName`	The name of the sub-test.

RUNIT_SUB_TEST_RESULT

This function returns the sub-test result.

```
RUNIT_SUB_TEST_RESULT(_testName)
```

It is called at the end of every sub-test. You can configure it to print a formatted line that indicates if the sub-test has ended successfully.

Table C-36 RUNIT_SUB_TEST_RESULT parameters

I/O	Parameter	Description
I	`_testName`	The name of the sub-test.

RUNIT_SUB_TEST_RESULT_W_PARAMS

This function returns the sub-test result, with optional test-specific information.

```
RUNIT_SUB_TEST_RESULT_W_PARAMS(_testName, _format, ...)
```

It is called at the end of every sub-test. You can configure it to print a formatted line that indicates if the sub-test has ended successfully.

Table C-37 RUNIT_SUB_TEST_RESULT_W_PARAMS parameters

I/O	Parameter	Description
I	`_testName`	The name of the sub-test.
I	`_format`	The entry format.
I	`...`	Format arguments.

C.4 ICV factory tool integration tests

This section describes the CryptoCell-312 ICV factory tool integration tests.

You must implement a subset of a function to serve as an abstraction layer between the integration test and the operating system of your choice.

This section contains the following subsections:

C.4.1 CMPUIT_PRINT

This function prints a log entry.

```
CMPUIT_PRINT(format, ...)
```

Table C-38 CMPUIT_PRINT parameters

I/O	Parameter	Description
I	`format`	The preferred output format.

C.4.2 CMPUIT_TEST_START

This function starts the test.

```
CMPUIT_TEST_START(testName)
```

It is called at the beginning of every test. You can configure it to print a formatted line that indicates the test has started.

Table C-39 CMPUIT_TEST_START parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

C.4.3 CMPUIT_TEST_RESULT

This function returns the test result.

```
CMPUIT_TEST_RESULT(testName)
```

It is called at the end of every test. You can configure it to print a formatted line that indicates if the test has ended successfully.

Table C-40 CMPUIT_TEST_RESULT parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

C.4.4 CMPUIT_PRINT_ERROR

This function prints an error message to log.

```
CMPUIT_PRINT_ERROR(format, ...)
```

Table C-41 CMPUIT_PRINT_ERROR parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

C.4.5 CMPUIT_PRINT_DBG

This function prints a debug message.

```
CMPUIT_PRINT_DBG(format, ...)
```

It is skipped unless compiled with `TEST_DEBUG`.

Table C-42 CMPUIT_PRINT_DBG parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

C.5 OEM factory tool integration tests

This section describes the CryptoCell-312 OEM factory tool integration tests.

You must implement a subset of a function to serve as an abstraction layer between the integration test and the operating system of your choice.

This section contains the following subsections:

C.5.1 DMPUIT_PRINT

This function prints a log entry.

```
DMPUIT_PRINT(format, ...)
```

Table C-43 DMPUIT_PRINT parameters

I/O	Parameter	Description
I	`format`	The preferred output format.

C.5.2 DMPUIT_TEST_START

This function starts the test.

```
DMPUIT_TEST_START(testName)
```

It is called at the beginning of every test. You can configure it to print a formatted line that indicates the test has started.

Table C-44 DMPUIT_TEST_START parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

C.5.3 DMPUIT_TEST_RESULT

This function returns the test result.

```
DMPUIT_TEST_RESULT(testName)
```

It is called at the end of every test. You can configure it to print a formatted line that indicates if the test has ended successfully.

Table C-45 DMPUIT_TEST_RESULT parameters

I/O	Parameter	Description
I	`testName`	The name of the test.

C.5.4 DMPUIT_PRINT_ERROR

This function prints an error message to log.

```
DMPUIT_PRINT_ERROR(format, ...)
```

Table C-46 DMPUIT_PRINT_ERROR parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

C.5.5 DMPUIT_PRINT_DBG

This function prints a debug message.

```
DMPUIT_PRINT_DBG(format, ...)
```

It is skipped unless compiled with `TEST_DEBUG`.

Table C-47 DMPUIT_PRINT_DBG parameters

I/O	Parameter	Description
I	`format`	The entry format.
I	`...`	Format arguments.

Appendix D
Revisions

This appendix describes the technical changes between released issues of this book.

It contains the following section:

- *D.1 Revision history* on page Appx-D-108.

D.1 Revision history

Table D-1 Issue 0000-00

Change	Location	Affects
First release	-	r0p0-00bet0 boot.

Table D-2 Differences between issue 0000-00 and issue 0000-01

Change	Location	Affects
Renamed all utilities: tools	Entire document	All
Removed the following sections: • *Runtime software package* • *Runtime integration test* • *Production-line utilities*	*2.1 Product components* on page 2-20	r0p0-00eac0 boot.
Updated section and added freeRTOS to supported operating systems in the the *Table 2-2 Build environment variables* on page 2-24 table.	*2.3 Host code build environment* on page 2-23	All
Updated.	*3.3.1 Chip Manufacturing LCS* on page 3-33	All
Updated.	*3.4 Boot services* on page 3-35	All
Minor updates.	*3.4.3 Secure Debug* on page 3-41	All
Minor updates.	*Secure Debug flow* on page 3-42	All
Expanded.	*DCU handling* on page 3-44	All
Removed the *Power managment* section.	*Chapter 3 Product features* on page 3-26	r0p0-00eac0 boot.
Minor updates.	*3.7 Device provisioning* on page 3-48	All
Removed the *OEM asset provisioning* section.	*Chapter 3 Product features* on page 3-26	r0p0-00eac0 boot.
Updated.	*4.1 Integration guidelines* on page 4-53	All
Updated.	*4.1.1 Cold boot sequence boot ROM guidelines* on page 4-53	All
Updated.	*Boot ROM completion in CM LCS* on page 4-54	All
Updated.	*Boot ROM completion in DM LCS* on page 4-54	All
Updated.	*Boot ROM completion in Secure LCS* on page 4-55	All
Updated.	*Boot ROM completion in RMA LCS* on page 4-56	All
Minor updates.	*Secure Boot sequence* on page 4-56	All
Updated.	*RMA LCS entry sequence* on page 4-57	All
Removed the following subsections: • *Cold boot sequence runtime software* • *Runtime considerations* • *CryptoCell-312 power management*	*4.1 Integration guidelines* on page 4-53	r0p0-00eac0 boot.

Table D-2 Differences between issue 0000-00 and issue 0000-01 (continued)

Change	Location	Affects
Removed the following subsections: • *Runtime software customization*	*4.2 Adaptation to your environment* on page 4-59	r0p0-00eac0 boot.
Removed the following subsections: • *Provisioning devices*	*Chapter 4 Integrating CryptoCell-312 software* on page 4-52	r0p0-00eac0 boot.
Updated.	*4.4 Debugging using Secure Debug* on page 4-63	All
Updated, and removed the following subsections: • *X.509 key certificate utility* • *X.509 content certificate utility*	*5.1 Overview of offline tools* on page 5-65	All
Updated parameter descriptions.	*5.2.1 Key certificate tool* on page 5-66	All
Updated parameter descriptions.	*5.2.2 Content certificate tool* on page 5-67	All
Updated.	*5.3 Secure Debug PC tools* on page 5-70	All
Updated and removed the following subsections: • *OEM certificate request utility* • *ICV OEM-key generation utility* • *OEM asset packing utility*	*5.4 Secure asset-provisioning PC tool* on page 5-73	r0p0-00eac0 boot.
Removed the *Production-line utilites* chapter.	-	r0p0-00eac0 boot.
Added section.	*Appendix A Random key generation* on page Appx-A-84	r0p0-00eac0 boot.
Added section.	*Appendix B CryptoCell-312 supported algorithms* on page Appx-B-87	r0p0-00eac0 boot.

Table D-3 Differences between issue 0000-01 and issue 0000-02

Change	Location	Affects
Added *Arm® TrustZone® CryptoCell-312 Runtime Software Release Note*.	*Additional reading* on page 11	All revisions.
Renamed and rephrased.	*1.1 Overview of CryptoCell-312* on page 1-16	All revisions.
Added section.	*2.1.3 Runtime software package* on page 2-20	r0p0-00eac0 runtime.
Added section.	*2.1.4 Runtime integration test* on page 2-21	r0p0-00eac0 runtime.
Added section.	*2.1.5 Production-line tools* on page 2-21	r0p0-00eac0 runtime.
Changed path to ICV and OEM factory tools.	*2.2 Project tree* on page 2-22	r0p0-00eac0 runtime.
• Added runtime and ICV factory tool flags. • Added second note. • Removed note from OS variable.	*2.3 Host code build environment* on page 2-23	r0p0-00eac0 runtime.
Updated.	*2.4 Build environment offline tools* on page 2-25	All revisions.

Table D-3 Differences between issue 0000-01 and issue 0000-02 (continued)

Change	Location	Affects
Restructured.	*3.1 Embedded secret keys* on page 3-27	All revisions.
Merged the *ICV keys* and *OEM keys* subsections into this section.	*3.1.3 Class keys* on page 3-27	All revisions.
Added section.	*3.6 Power management* on page 3-47	r0p0-00eac0 runtime.
Added section.	*3.8 Cryptographic acceleration* on page 3-49	r0p0-00eac0 runtime.
Added section.	*3.9 Secure Boot services for the runtime library* on page 3-50	r0p0-00eac0 runtime.
Added section.	*3.10 External DMA support* on page 3-51	r0p0-00eac0 runtime.
Added section.	*4.1.2 Cold boot sequence in runtime software* on page 4-57	r0p0-00eac0 runtime.
Added section.	*4.1.3 Runtime considerations* on page 4-58	r0p0-00eac0 runtime.
Added section.	*4.1.4 CryptoCell-312 power management sequence* on page 4-58	r0p0-00eac0 runtime.
Added section and its subsections.	*4.2.3 Runtime software customization* on page 4-60	r0p0-00eac0 runtime.
Added the *4.3 Provisioning devices* on page 4-62 section and its subsections.	*4.3 Provisioning devices* on page 4-62	r0p0-00eac0 runtime.
Renamed from *ICV asset provisioning*.	*3.7.1 Boot ICV asset provisioning* on page 3-48	r0p0-00eac0 runtime.
Added the *3.7.2 Runtime asset provisioning* on page 3-48 section.	*3.7 Device provisioning* on page 3-48	r0p0-00eac0 runtime.
Merged into *5.4 Secure asset-provisioning PC tool* on page 5-73 ands updated.	*ICV asset provisioning tool*	r0p0-00eac0 runtime.
Renamed chapter and added the following sections: • *5.5 ICV factory PC tool* on page 5-74 • *5.6 OEM factory PC tools* on page 5-75	*Chapter 5 Offline tools* on page 5-64	r0p0-00eac0 runtime.
Added chapter.	*Chapter 6 CryptoCell-312 production-line tools* on page 6-78	r0p0-00eac0 runtime.

Table D-3 Differences between issue 0000-01 and issue 0000-02 (continued)

Change	Location	Affects
Subdivided into: • *C.2 Boot services integration tests* on page Appx-C-94. • *C.1 Common integration tests* on page Appx-C-90. Added the following sections: • TestHalCheckLCS • *C.3 Runtime integration tests* on page Appx-C-98. • *C.4 ICV factory tool integration tests* on page Appx-C-103. • *C.5 OEM factory tool integration tests* on page Appx-C-105.	*Appendix C Integration test* on page Appx-C-89	r0p0-00eac0 runtime.
Removed the *BSVIT_PRINT_FUNC_AND_LEVEL* section.	*C.2 Boot services integration tests* on page Appx-C-94	r0p0-00eac0 runtime.