arm

Arm® CryptoCell-312

Revision: r1p3

Boot Services Software Developers Manual

Issue 01

101468

Non-Confidential

Arm® CryptoCell-312

Boot Services Software Developers Manual

Release information

Document history

Issue	Date	Confidentiality	Change
0103-01	25-Jul-19	Non-Confidential	First release of Boot Services Software Developers Manual for r1p3.

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

1 Introduction

1.1 Product revision status

The *rmpn* 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, r*1*.

p*n*

Identifies the minor revision or modification status of the product, for example, p*2*.

1.2 Intended audience

This document is written for programmers using the CryptoCell-312 cryptographic APIs.

Familiarity with the basics of security and cryptography is assumed.

1.3 Conventions

The following subsections describe conventions used in Arm documents.

1.3.1 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.

1.3.2 Typographical conventions

Convention	Use
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 bold`	Denotes language keywords when used outside example code.

Convention	Use
monospace italic	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace underline	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `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.
⚠	Caution
✋	Warning
✎	Note

1.4 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-1 Arm publications

Document name	Document ID	Licensee only Y/N
Arm® Armv8-M Architecture Reference Manual	*ARM DDI 0553A.j*	N
Arm® CryptoCell-312 Software Integrators Manual	100776	Y

Table 1-2 Other publications

Document ID	Document name
NIST SP 800-108	*Recommendation for Key Derivation Using Pseudorandom Functions*

1.5 Feedback

Arm welcomes feedback on this product and its documentation.

1.5.1 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.

1.5.2 Feedback on content

If you have comments on content, send an e-mail to errata@arm.com and give:

- The title Arm® CryptoCell-312 Boot Services Software Developers Manual.
- The number 101468.
- 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.

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.

2 Boot Services API layer

2.1 Modules

Here is a list of all modules:

- CryptoCell-312 Boot Services
  - Boot Services APIs
    - Boot Services cryptographic APIs
    - Boot Services cryptographic ROM definitions
    - Boot Services error codes
  - BSV HAL layer APIs
    - BSV HAL layer platform-dependent APIs
    - BSV HAL layer platform-dependent definitions
  - Hash definitions and types
  - BSV PAL layer APIs
    - BSV PAL layer platform-dependent definitions
    - BSV PAL layer platform-dependent types
    - BSV PAL platform-dependent type definitions
  - OTP memory read and write operations
  - Secure Boot APIs and definitions
    - Secure Boot APIs
    - Secure Boot basic type definitions
    - Secure Boot definitions
    - Secure Boot error codes
  - Secure Debug APIs
    - Secure Debug APIs and definitions
  - Secure Boot and Secure Debug definitions
    - Secure Boot and Secure Debug general definitions and structures
    - Secure Boot and Secure Debug error codes

2.2 Data structures

The following are the data structures that are part of the delivery:

- **CCSbCertInfo_t**
- **CCSbCertParserSwCompsInfo_t**
- **CCSbSwVersion_t**

2.3 File list

The following table lists the files that are part of the delivery, and their descriptions:

Table 2-1 List of files

Filename	Description
`bootimagesverifier_api.h`	This file contains the set of Secure Boot APIs.
`bootimagesverifier_def.h`	This file contains definitions used for the Secure Boot and Secure Debug APIs.
`bootimagesverifier_error.h`	This file defines the error codes used for Secure Boot and Secure Debug APIs.
`bsv_api.h`	This file contains the Boot Services APIs and definitions.
`bsv_crypto_api.h`	This file contains the cryptographic ROM APIs.
`bsv_crypto_defs.h`	This file contains the definitions for the cryptographic ROM APIs.
`bsv_error.h`	This file defines the types of error codes that the Boot Services (BSV) APIs return.
`bsv_otp_api.h`	This file contains functions that access the OTP memory for read and write operations.
`cc_crypto_boot_defs.h`	This file contains Secure Boot and Secure Debug definitions.
`cc_hal_sb.h`	This file contains functions used for the HAL layer of the Boot Services.
`cc_hal_sb_plat.h`	This file contains definitions that are used for the Boot Services HAL layer.
`cc_pal_sb_plat.h`	This file contains PAL platform-dependent definitions used in the Boot Services code.
`cc_pal_types.h`	This file contains PAL platform-dependent definitions and types.
`cc_pal_types_plat.h`	This file contains basic PAL platform-dependent type definitions.
`cc_sec_defs.h`	This file contains general hash definitions and types.

Filename	Description
`secdebug_api.h`	This file contains the Secure Debug APIs and definitions.
`secureboot_defs.h`	This file contains basic type definitions for the Secure Boot.
`secureboot_error.h`	This file defines the types of error codes that the Secure Boot code returns.
`secureboot_gen_defs.h`	This file contains all of the definitions and structures used for the Secure Boot and Secure Debug.

2.4 Module documentation

2.4.1 Modules

- **Boot Services APIs**
- **BSV HAL layer APIs**
- **Hash definitions and types**
- **BSV PAL layer APIs**
- **OTP memory read and write operations**
- **Secure Boot APIs and definitions**
- **Secure Debug APIs**
- **Secure Boot and Secure Debug definitions**

2.4.2 Detailed description

Contains all of the APIs and definitions for CryptoCell-312 Boot Services.

2.5 Boot Services APIs

2.5.1 Modules

- **Boot Services cryptographic APIs**
- **Boot Services error codes**

2.5.2 Files

- file `bsv_api.h`

2.5.3 Macros

- #define **CC_BSV_CHIP_MANUFACTURE_LCS** 0x0
- #define **CC_BSV_DEVICE_MANUFACTURE_LCS** 0x1
- #define **CC_BSV_SECURE_LCS** 0x5
- #define **CC_BSV_RMA_LCS** 0x7

2.5.4 Functions

- **CCError_t CC_BsvInit** (unsigned long hwBaseAddress)
- **CCError_t CC_BsvLcsGet** (unsigned long hwBaseAddress, uint32_t *pLcs)
- **CCError_t CC_BsvLcsGetAndInit** (unsigned long hwBaseAddress, uint32_t *pLcs)
- **CCError_t CC_BsvOTPPrivateKeysErase** (unsigned long hwBaseAddress, **CCBool_t** isHukErase, **CCBool_t** isKpicvErase, **CCBool_t** isKceicvErase, **CCBool_t** isKcpErase, **CCBool_t** isKceErase, uint32_t *pStatus)
- **CCError_t CC_BsvFatalErrorSet** (unsigned long hwBaseAddress)
- **CCError_t CC_BsvRMAModeEnable** (unsigned long hwBaseAddress)
- **CCError_t CC_BsvSocIDCompute** (unsigned long hwBaseAddress, **CCHashResult_t** hashResult)
- **CCError_t CC_BsvICVKeyLock** (unsigned long hwBaseAddress, **CCBool_t** isICVProvisioningKeyLock, **CCBool_t** isICVCodeEncKeyLock)
- **CCError_t CC_BsvICVRMAFlagBitLock** (unsigned long hwBaseAddress)
- **CCError_t CC_BsvCoreClkGatingEnable** (unsigned long hwBaseAddress)
- **CCError_t CC_BsvSecModeSet** (unsigned long hwBaseAddress, **CCBool_t** isSecAccessMode, **CCBool_t** isSecModeLock)
- **CCError_t CC_BsvPrivModeSet** (unsigned long hwBaseAddress, **CCBool_t** isPrivAccessMode, **CCBool_t** isPrivModeLock)
- **CCError_t CC_BsvIcvAssetProvisioningOpen** (unsigned long hwBaseAddress, uint32_t assetId, uint32_t *pAssetPkgBuff, size_t assetPackageLen, uint8_t *pOutAssetData, size_t *pAssetDataLen)

2.5.4.1 Detailed description

Contains the Boot Services APIs.

2.5.4.2 Macro definition documentation

2.5.4.2.1 #define CC_BSV_CHIP_MANUFACTURE_LCS 0x0

Defines the CM life-cycle state value.

2.5.4.2.2 #define CC_BSV_DEVICE_MANUFACTURE_LCS 0x1

Defines the DM life-cycle state value.

2.5.4.2.3 #define CC_BSV_RMA_LCS 0x7

Defines the RMA life-cycle state value.

2.5.4.2.4 #define CC_BSV_SECURE_LCS 0x5

Defines the Secure life-cycle state value.

2.5.4.3 Function documentation

2.5.4.3.1 CCError_t CC_BsvCoreClkGatingEnable (unsigned long hwBaseAddress)

This API enables the core_clk gating mechanism, which is disabled during power-up.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.

2.5.4.3.2 CCError_t CC_BsvFatalErrorSet (unsigned long hwBaseAddress)

This function sets the "fatal error" flag in the NVM manager, to disable the use of any HW Keys or security services.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.

2.5.4.3.3 CCError_t CC_BsvIcvAssetProvisioningOpen (unsigned long hwBaseAddress, uint32_t assetId, uint32_t * pAssetPkgBuff, size_t assetPackageLen, uint8_t * pOutAssetData, size_t * pAssetDataLen)

This function unpacks the ICV asset packet and returns the asset data.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`assetId`	The asset identifier.
in	`pAssetPkgBuff`	The asset package word-array formatted to unpack.
in	`assetPackageLen`	The exact length of the asset package in bytes. Must be multiple of 16 bytes.
out	`pOutAssetData`	The decrypted contents of the asset data.
in, out	`pAssetDataLen`	As input: the size of the allocated asset data buffer. The maximal size is 512 bytes. As output: the actual size of the decrypted asset data buffer. The maximal size is 512 bytes.

2.5.4.3.4 CCError_t **CC_BsvICVKeyLock (unsigned long hwBaseAddress,** CCBool_t **isICVProvisioningKeyLock,** CCBool_t **isICVCodeEncKeyLock)**

This function must be called when the user needs to lock one of the ICV keys from further usage.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`isICVProvisioningKeyLock`	The ICV provisioning key mode: CC_TRUE: Kpicv is locked for further usage. CC_FALSE: Kpicv is not locked.
in	`isICVCodeEncKeyLock`	The ICV code encryption key mode: CC_TRUE: Kceicv is locked for further usage. CC_FALSE: Kceicv is not locked.

2.5.4.3.5 CCError_t **CC_BsvICVRMAFlagBitLock (unsigned long hwBaseAddress)**

This function is called by the ICV code to disable the OEM code from changing the ICV RMA bit flag.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.

2.5.4.3.6 CCError_t **CC_BsvInit (unsigned long hwBaseAddress)**

This function must be the first Arm CryptoCell 3xx SBROM library API called.

It verifies the HW product and version numbers and initializes the HW.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.

2.5.4.3.7 CCError_t **CC_BsvLcsGet (unsigned long hwBaseAddress, uint32_t * pLcs)**

This function retrieves the security life-cycle state from the NVM manager.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
out	`pLcs`	The value of the current security life-cycle state.

2.5.4.3.8 CCError_t **CC_BsvLcsGetAndInit (unsigned long hwBaseAddress, uint32_t * pLcs)**

This function retrieves the HW security life-cycle state and performs validity checks.

If LCS is RMA, the function performs additional initializations (sets the OTP secret keys to a fixed value).

If the LCS is invalid, the function returns an error, upon which your code must completely disable the device.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
out	`pLcs`	The returned life-cycle state.

2.5.4.3.9 CCError_t **CC_BsvOTPPrivateKeysErase (unsigned long hwBaseAddress,** CCBool_t **isHukErase,** CCBool_t **isKpicvErase,** CCBool_t **isKceicvErase,** CCBool_t **isKcpErase,** CCBool_t **isKceErase, uint32_t * pStatus)**

This function is called in RMA LCS to erase one or more of the private keys.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`isHukErase`	The HUK secret key: CC_TRUE: HUK is erased. CC_FALSE: HUK remains unchanged.
in	`isKpicvErase`	Kpicv secret key: CC_TRUE: Kpicv is erased. CC_FALSE: Kpicv remains unchanged.
in	`isKceicvErase`	Kceicv secret key: CC_TRUE: Kceicv is erased. CC_FALSE: Kceicv remains unchanged.
in	`isKcpErase`	Kcp secret key: CC_TRUE: Kcp is erased. CC_FALSE: Kcp remains unchanged.
in	`isKceErase`	Kce secret key: CC_TRUE: Kce is erased. CC_FALSE: Kce remains unchanged.
out	`pStatus`	Returned status word.

2.5.4.3.10 CCError_t **CC_BsvPrivModeSet (unsigned long hwBaseAddress,** CCBool_t **isPrivAccessMode,** CCBool_t **isPrivModeLock)**

This function activates the APB privilege filter, allowing only secure transactions to access CryptoCell-312 registers.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`isPrivAccessMode`	The APB privileged mode: CC_TRUE: only privileged accesses are served. CC_FALSE: both privileged and non-privileged accesses are served.
in	`isPrivModeLock`	The privileged lock mode: CC_TRUE: privileged mode is locked for further changes. CC_FALSE: privileged mode is not locked.

2.5.4.3.11 CCError_t **CC_BsvRMAModeEnable (unsigned long hwBaseAddress)**

This function permanently sets the RMA life-cycle state per OEM or ICV.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.

2.5.4.3.12 CCError_t **CC_BsvSecModeSet (unsigned long hwBaseAddress,** CCBool_t **isSecAccessMode,** CCBool_t **isSecModeLock)**

This function controls the APB secure filter, allowing only secure transactions to access CryptoCell-312 registers.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`isSecAccessMode`	The APB secure filter mode: CC_TRUE: only secure accesses are served. CC_FALSE: both secure and non-secure accesses are served.
in	`isSecModeLock`	The APB security lock mode: CC_TRUE: secure filter mode is locked for further changes. CC_FALSE: secure filter mode is not locked.

2.5.4.3.13 CCError_t **CC_BsvSocIDCompute (unsigned long hwBaseAddress, CCHashResult_t hashResult)**

This function derives the unique SoC_ID of the device as hashed (Hbk || AES_CMAC (HUK)).

SoC_ID is required for the creation of debug certificates. Therefore, the OEM or ICV must provide a method for a developer to discover the SoC_ID of a target device without having to first enable debugging. One suggested implementation is to have the ROM code of the device compute the SoC_ID and place it in a specific location in the flash memory, where it can be accessed by the developer.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
out	`hashResult`	The derived SOC ID.

2.5.5 Boot Services cryptographic APIs

2.5.5.1 Modules

- **Boot Services cryptographic ROM definitions**

2.5.5.2 Files

- file `bsv_crypto_api.h`

2.5.5.3 Functions

- **CCError_t CC_BsvSHA256** (unsigned long hwBaseAddress, uint8_t *pDataIn, size_t dataSize, **CCHashResult_t** hashBuff)
- **CCError_t CC_BsvKeyDerivation** (unsigned long hwBaseAddress, **CCBsvKeyType_t** keyType, uint32_t *pUserKey, size_t userKeySize, const uint8_t *pLabel, size_t labelSize, const uint8_t *pContextData, size_t contextSize, uint8_t *pDerivedKey, size_t derivedKeySize)
- **CCError_t CC_BsvAesCcm** (unsigned long hwBaseAddress, **CCBsvCcmKey_t** keyBuf, **CCBsvCcmNonce_t** nonceBuf, uint8_t *pAssocData, size_t assocDataSize, uint8_t *pTextDataIn, size_t textDataSize, uint8_t *pTextDataOut, **CCBsvCcmMacRes_t** macBuf)

2.5.5.4 Detailed Description

Contains the cryptographic ROM APIs:

- SHA-256
- CMAC
- KDF
- CCM

2.5.5.5 Function Documentation

2.5.5.5.1 CCError_t **CC_BsvAesCcm (unsigned long hwBaseAddress,** CCBsvCcmKey_t **keyBuf,** CCBsvCcmNonce_t **nonceBuf, uint8_t * pAssocData, size_t assocDataSize, uint8_t * pTextDataIn, size_t textDataSize, uint8_t * pTextDataOut,** CCBsvCcmMacRes_t **macBuf)**

This function performs a limited AES-CCM decrypt and verify operation required for AES-CCM verification during boot.

AES-CCM combines counter mode encryption with CBC-MAC authentication.

Input to CCM includes the following elements:

- Payload - text data that is both decrypted and verified.
- Associated data (Adata) - data that is authenticated but not encrypted such as a header.
- Nonce - A unique value that is assigned to the payload and the associated data.

Returns:

CC_OK on success.

A non-zero value on failure as defined `bsv_error.h`.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`keyBuf`	A pointer to the 128-bit AES-CCM key.
in	`nonceBuf`	Pointer to the 12-byte Nonce.
in	`pAssocData`	A pointer to the associated data. The buffer must be contiguous.
in	`assocDataSize`	The byte size of the associated data. Limited to (2^16-2^8) bytes.
in	`pTextDataIn`	A pointer to the cipher-text data for decryption. The buffer must be contiguous.
in	`textDataSize`	The byte size of the full text data. Limited to 64KB.
out	`pTextDataOut`	A pointer to the output (plain text data). The buffer must be contiguous.
in	`macBuf`	A pointer to the MAC result buffer.

2.5.5.5.2 CCError_t **CC_BsvKeyDerivation (unsigned long hwBaseAddress,** CCBsvKeyType_t **keyType, uint32_t * pUserKey, size_t userKeySize, const uint8_t * pLabel, size_t labelSize, const uint8_t * pContextData, size_t contextSize, uint8_t * pDerivedKey, size_t derivedKeySize)**

This function derives a secret key from a user key using the key derivation function specified in the "KDF in Counter Mode" section of *NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions*.

Key derivation is based on length l, label L, context C and derivation key Ki. AES-CMAC is used as the pseudorandom function (PRF).

When using this API, the label and context for each use-case must be well defined.

Arm recommends deriving only 256-bit keys from HUK, or 256-bit user keys.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`keyType`	Defines the type of the key provided in *pUserKey: HUK, Krtl, KCP, KPICV, 128-bit User key, and 256-bit User Key.
in	`pUserKey`	A pointer to the buffer holding the user key.
in	`userKeySize`	The size of the user key in bytes (limited to 16 bytes or 32 bytes).

I/O	Parameter	Description
in	pLabel	A string that identifies the purpose for the derived keying material.
in	labelSize	The label size. Must be between 1 to 8 bytes in length.
in	pContextData	A binary string containing the information related to the derived keying material.
in	contextSize	The context size should be between 1 to 32 bytes in length.
out	pDerivedKey	The keying material output. Must be at least the size defined in derivedKeySize.
in	derivedKeySize	The size of the derived keying material in bytes. Limited to 128 bits or 256 bits.

2.5.5.5.3 CCError_t **CC_BsvSHA256 (unsigned long hwBaseAddress, uint8_t * pDataIn, size_t dataSize,** CCHashResult_t **hashBuff)**

This function calculates SHA-256 digest over contiguous memory in an integrated operation.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	hwBaseAddress	The base address for CryptoCell HW registers.
in	pDataIn	A pointer to the input data to be hashed. The buffer must be contiguous.
in	dataSize	The size of the data to be hashed in bytes. Limited to 64KB.
out	hashBuff	A pointer to a word-aligned 32-byte buffer.

2.5.6 Boot Services cryptographic ROM definitions

2.5.6.1 Files

- file `bsv_crypto_defs.h`

2.5.6.2 Macros

- #define **CC_BSV_CMAC_RESULT_SIZE_IN_WORDS** 4
- #define **CC_BSV_CMAC_RESULT_SIZE_IN_BYTES** 16
- #define **CC_BSV_CCM_KEY_SIZE_BYTES** 16
- #define **CC_BSV_CCM_KEY_SIZE_WORDS** 4
- #define **CC_BSV_CCM_NONCE_SIZE_BYTES** 12

2.5.6.3 Typedefs

- typedef uint32_t **CCBsvCmacResult_t**[**CC_BSV_CMAC_RESULT_SIZE_IN_WORDS**]
- typedef uint32_t **CCBsvCcmKey_t**[**CC_BSV_CCM_KEY_SIZE_WORDS**]
- typedef uint8_t **CCBsvCcmNonce_t**[**CC_BSV_CCM_NONCE_SIZE_BYTES**]
- typedef uint8_t **CCBsvCcmMacRes_t**[**CC_BSV_CMAC_RESULT_SIZE_IN_BYTES**]

2.5.6.4 Enumerations

- enum **CCBsvKeyType_t** { **CC_BSV_HUK_KEY** = 0, **CC_BSV_RTL_KEY** = 1, **CC_BSV_PROV_KEY** = 2, **CC_BSV_CE_KEY** = 3, **CC_BSV_ICV_PROV_KEY** = 4, **CC_BSV_ICV_CE_KEY** = 5, **CC_BSV_USER_KEY** = 6, **CC_BSV_END_OF_KEY_TYPE** = 0x7FFFFFFF }

2.5.6.5 Detailed Description

Contains the definitions for the cryptographic ROM APIs.

2.5.6.6 Macro Definition Documentation

2.5.6.7 #define CC_BSV_CCM_KEY_SIZE_BYTES 16

The size of the AES-CCM 128-bit key in bytes.

2.5.6.8 #define CC_BSV_CCM_KEY_SIZE_WORDS 4

The size of the AES-CCM 128-bit key in words.

2.5.6.9 #define CC_BSV_CCM_NONCE_SIZE_BYTES 12

The size of the AES-CCM NONCE in bytes.

2.5.6.10 #define CC_BSV_CMAC_RESULT_SIZE_IN_BYTES 16

The size of the AES-CMAC result in bytes.

2.5.6.11 #define CC_BSV_CMAC_RESULT_SIZE_IN_WORDS 4

The size of the AES-CMAC result in words.

2.5.6.12 Typedef Documentation

2.5.6.13 typedef uint32_t CCBsvCcmKey_t[CC_BSV_CCM_KEY_SIZE_WORDS]

The definition of the AES-CCM key buffer.

2.5.6.14 typedef uint8_t CCBsvCcmMacRes_t[CC_BSV_CMAC_RESULT_SIZE_IN_BYTES]

The definition of the AES-CCM MAC buffer.

2.5.6.15 typedef uint8_t CCBsvCcmNonce_t[CC_BSV_CCM_NONCE_SIZE_BYTES]

The definition of the AES-CCM nonce buffer.

2.5.6.16 typedef uint32_t CCBsvCmacResult_t[CC_BSV_CMAC_RESULT_SIZE_IN_WORDS]

The CMAC result buffer.

2.5.6.17 Enumeration Type Documentation

2.5.6.18 enum CCBsvKeyType_t

The types of AES keys.

Enumerator:

Enum	Description
`CC_BSV_HUK_KEY`	The root key (HUK).
`CC_BSV_RTL_KEY`	The RTL key (Krtl).
`CC_BSV_PROV_KEY`	The OEM provisioning key (Kcp).
`CC_BSV_CE_KEY`	The OEM code encryption key (Kce).
`CC_BSV_ICV_PROV_KEY`	The ICV Provisioning key (Kpicv).
`CC_BSV_ICV_CE_KEY`	The ICV code encryption key (Kceicv).
`CC_BSV_USER_KEY`	The user key.
`CC_BSV_END_OF_KEY_TYPE`	Reserved.

2.5.7 Boot Services error codes

2.5.7.1 Files

- file `bsv_error.h`

2.5.7.2 Macros

- #define **CC_BSV_BASE_ERROR** 0x0B000000
- #define **CC_BSV_CRYPTO_ERROR** 0x0C000000
- #define **CC_BSV_ILLEGAL_INPUT_PARAM_ERR** (**CC_BSV_BASE_ERROR** + 0x00000001)
- #define **CC_BSV_ILLEGAL_HUK_VALUE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000002)
- #define **CC_BSV_ILLEGAL_KCP_VALUE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000003)
- #define **CC_BSV_ILLEGAL_KCE_VALUE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000004)
- #define **CC_BSV_ILLEGAL_KPICV_VALUE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000005)
- #define **CC_BSV_ILLEGAL_KCEICV_VALUE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000006)
- #define **CC_BSV_HASH_NOT_PROGRAMMED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000007)
- #define **CC_BSV_HBK_ZERO_COUNT_ERR** (**CC_BSV_BASE_ERROR** + 0x00000008)
- #define **CC_BSV_ILLEGAL_LCS_ERR** (**CC_BSV_BASE_ERROR** + 0x00000009)
- #define **CC_BSV_OTP_WRITE_CMP_FAIL_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000A)
- #define **CC_BSV_ERASE_KEY_FAILED_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000B)
- #define **CC_BSV_ILLEGAL_PIDR_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000C)
- #define **CC_BSV_ILLEGAL_CIDR_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000D)
- #define **CC_BSV_FAILED_TO_SET_FATAL_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000E)
- #define **CC_BSV_FAILED_TO_SET_RMA_ERR** (**CC_BSV_BASE_ERROR** + 0x0000000F)
- #define **CC_BSV_ILLEGAL_RMA_INDICATION_ERR** (**CC_BSV_BASE_ERROR** + 0x00000010)
- #define **CC_BSV_VER_IS_NOT_INITIALIZED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000011)
- #define **CC_BSV_APB_SECURE_IS_LOCKED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000012)
- #define **CC_BSV_APB_PRIVILEG_IS_LOCKED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000013)

- #define **CC_BSV_ILLEGAL_OPERATION_ERR** (**CC_BSV_BASE_ERROR** + 0x00000014)
- #define **CC_BSV_ILLEGAL_ASSET_SIZE_ERR** (**CC_BSV_BASE_ERROR** + 0x00000015)
- #define **CC_BSV_ILLEGAL_ASSET_VAL_ERR** (**CC_BSV_BASE_ERROR** + 0x00000016)
- #define **CC_BSV_KPICV_IS_LOCKED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000017)
- #define **CC_BSV_ILLEGAL_SW_VERSION_ERR** (**CC_BSV_BASE_ERROR** + 0x00000018)
- #define **CC_BSV_AO_WRITE_FAILED_ERR** (**CC_BSV_BASE_ERROR** + 0x00000019)
- #define **CC_BSV_INVALID_DATA_IN_POINTER_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000001)
- #define **CC_BSV_INVALID_DATA_OUT_POINTER_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000002)
- #define **CC_BSV_INVALID_DATA_SIZE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000003)
- #define **CC_BSV_INVALID_KEY_TYPE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000004)
- #define **CC_BSV_INVALID_KEY_SIZE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000005)
- #define **CC_BSV_ILLEGAL_KDF_LABEL_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000006)
- #define **CC_BSV_ILLEGAL_KDF_CONTEXT_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000007)
- #define **CC_BSV_CCM_INVALID_KEY_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000008)
- #define **CC_BSV_CCM_INVALID_NONCE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000009)
- #define **CC_BSV_CCM_INVALID_ASSOC_DATA_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000A)
- #define **CC_BSV_CCM_INVALID_TEXT_DATA_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000B)
- #define **CC_BSV_CCM_INVALID_MAC_BUF_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000C)
- #define **CC_BSV_CCM_DATA_OUT_DATA_IN_OVERLAP_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000D)
- #define **CC_BSV_CCM_MAC_INVALID_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000E)
- #define **CC_BSV_CCM_INVALID_MODE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x0000000F)

- #define **CC_BSV_INVALID_OUT_POINTER_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000010)
- #define **CC_BSV_INVALID_CRYPTO_MODE_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000011)
- #define **CC_BSV_INVALID_IV_POINTER_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000012)
- #define **CC_BSV_INVALID_RESULT_BUFFER_POINTER_ERROR** (**CC_BSV_CRYPTO_ERROR** + 0x00000013)

2.5.7.3 Detailed Description

Defines the types of error codes that the Boot Services (BSV) APIs return.

2.5.7.4 Macro Definition Documentation

2.5.7.4.1 #define CC_BSV_AO_WRITE_FAILED_ERR (CC_BSV_BASE_ERROR + 0x00000019)

Defines the error code for when an AO write operation fails.

2.5.7.4.2 #define CC_BSV_APB_PRIVILEG_IS_LOCKED_ERR (CC_BSV_BASE_ERROR + 0x00000013)

Defines the error code for when the APB privilege mode is locked.

2.5.7.4.3 #define CC_BSV_APB_SECURE_IS_LOCKED_ERR (CC_BSV_BASE_ERROR + 0x00000012)

Defines the error code for when the APB secure mode is locked.

2.5.7.4.4 #define CC_BSV_BASE_ERROR 0x0B000000

Defines the base error code for the BSV.

2.5.7.4.5 #define CC_BSV_CCM_DATA_OUT_DATA_IN_OVERLAP_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000D)

Defines the error code for overlapping input and output data.

2.5.7.4.6 #define CC_BSV_CCM_INVALID_ASSOC_DATA_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000A)

Defines the error code for an invalid CCM associated data.

2.5.7.4.7 #define CC_BSV_CCM_INVALID_KEY_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000008)

Defines the error code for an invalid CCM key.

2.5.7.4.8 #define CC_BSV_CCM_INVALID_MAC_BUF_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000C)

Defines the error code for an invalid CCM-MAC buffer.

2.5.7.4.9 #define CC_BSV_CCM_INVALID_MODE_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000F)

Defines the error code for an invalid CCM mode.

2.5.7.4.10 #define CC_BSV_CCM_INVALID_NONCE_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000009)

Defines the error code for an invalid CCM Nonce.

2.5.7.4.11 #define CC_BSV_CCM_INVALID_TEXT_DATA_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000B)

Defines the error code for an invalid CCM text data.

2.5.7.4.12 #define CC_BSV_CCM_MAC_INVALID_ERROR (CC_BSV_CRYPTO_ERROR + 0x0000000E)

Defines the error code for when CCM-MAC comparison fails.

2.5.7.4.13 #define CC_BSV_CRYPTO_ERROR 0x0C000000

Defines the cryptographic base error code for the BSV.

2.5.7.4.14 #define CC_BSV_ERASE_KEY_FAILED_ERR (CC_BSV_BASE_ERROR + 0x0000000B)

Defines the error code for when erasing a key in OTP fails.

2.5.7.4.15 #define CC_BSV_FAILED_TO_SET_FATAL_ERR (CC_BSV_BASE_ERROR + 0x0000000E)

Defines the error code for when the Device fails to move to fatal error state.

2.5.7.4.16 #define CC_BSV_FAILED_TO_SET_RMA_ERR (CC_BSV_BASE_ERROR + 0x0000000F)

Defines the error code for when entry to RMA LCS fails.

2.5.7.4.17 #define CC_BSV_HASH_NOT_PROGRAMMED_ERR (CC_BSV_BASE_ERROR + 0x00000007)

Defines the error code for when the hash boot key is not programmed in the OTP.

2.5.7.4.18 #define CC_BSV_HBK_ZERO_COUNT_ERR (CC_BSV_BASE_ERROR + 0x00000008)

Defines the error code for an illegal hash boot key zero count in the OTP.

2.5.7.4.19 #define CC_BSV_ILLEGAL_ASSET_SIZE_ERR (CC_BSV_BASE_ERROR + 0x00000015)

Defines the error code for an illegal asset size.

2.5.7.4.20 #define CC_BSV_ILLEGAL_ASSET_VAL_ERR (CC_BSV_BASE_ERROR + 0x00000016)

Defines the error code for an illegal asset value.

2.5.7.4.21 #define CC_BSV_ILLEGAL_CIDR_ERR (CC_BSV_BASE_ERROR + 0x0000000D)

Defines the error code for an illegal CIDR.

2.5.7.4.22 #define CC_BSV_ILLEGAL_HUK_VALUE_ERR (CC_BSV_BASE_ERROR + 0x00000002)

Defines the error code for an illegal HUK value.

2.5.7.4.23 #define CC_BSV_ILLEGAL_INPUT_PARAM_ERR (CC_BSV_BASE_ERROR + 0x00000001)

Defines the error code for an illegal input parameter.

2.5.7.4.24 #define CC_BSV_ILLEGAL_KCE_VALUE_ERR (CC_BSV_BASE_ERROR + 0x00000004)

Defines the error code for an illegal Kce value.

2.5.7.4.25 #define CC_BSV_ILLEGAL_KCEICV_VALUE_ERR (CC_BSV_BASE_ERROR + 0x00000006)

Defines the error code for an illegal Kceicv value.

2.5.7.4.26 #define CC_BSV_ILLEGAL_KCP_VALUE_ERR (CC_BSV_BASE_ERROR + 0x00000003)

Defines the error code for an illegal Kcp value.

2.5.7.4.27 #define CC_BSV_ILLEGAL_KDF_CONTEXT_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000007)

Defines the error code for an illegal KDF context.

2.5.7.4.28 #define CC_BSV_ILLEGAL_KDF_LABEL_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000006)

Defines the error code for an illegal KDF label.

2.5.7.4.29 #define CC_BSV_ILLEGAL_KPICV_VALUE_ERR (CC_BSV_BASE_ERROR + 0x00000005)

Defines the error code for an illegal Kpicv value.

2.5.7.4.30 #define CC_BSV_ILLEGAL_LCS_ERR (CC_BSV_BASE_ERROR + 0x00000009)

Defines the error code for an illegal LCS.

2.5.7.4.31 #define CC_BSV_ILLEGAL_OPERATION_ERR (CC_BSV_BASE_ERROR + 0x00000014)

Defines the error code for an illegal operation.

2.5.7.4.32 #define CC_BSV_ILLEGAL_PIDR_ERR (CC_BSV_BASE_ERROR + 0x0000000C)

Defines the error code for an illegal PIDR.

2.5.7.4.33 #define CC_BSV_ILLEGAL_RMA_INDICATION_ERR (CC_BSV_BASE_ERROR + 0x00000010)

Defines the error code for an illegal RMA indication.

2.5.7.4.34 #define CC_BSV_ILLEGAL_SW_VERSION_ERR (CC_BSV_BASE_ERROR + 0x00000018)

Defines the error code for an illegal SW version.

2.5.7.4.35 #define CC_BSV_INVALID_CRYPTO_MODE_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000011)

Defines the error code for an illegal cryptographic mode.

2.5.7.4.36 #define CC_BSV_INVALID_DATA_IN_POINTER_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000001)

Defines the error code for an illegal data in pointer.

2.5.7.4.37 #define CC_BSV_INVALID_DATA_OUT_POINTER_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000002)

Defines the error code for an illegal data out pointer.

2.5.7.4.38 #define CC_BSV_INVALID_DATA_SIZE_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000003)

Defines the error code for an illegal data size.

2.5.7.4.39 #define CC_BSV_INVALID_IV_POINTER_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000012)

Defines the error code for an illegal IV pointer.

2.5.7.4.40 #define CC_BSV_INVALID_KEY_SIZE_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000005)

Defines the error code for an illegal key size.

2.5.7.4.41 #define CC_BSV_INVALID_KEY_TYPE_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000004)

Defines the error code for an illegal key type.

2.5.7.4.42 #define CC_BSV_INVALID_OUT_POINTER_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000010)

Defines the error code for an invalid out pointer.

2.5.7.4.43 #define CC_BSV_INVALID_RESULT_BUFFER_POINTER_ERROR (CC_BSV_CRYPTO_ERROR + 0x00000013)

Defines the error code for an illegal result buffer pointer.

2.5.7.4.44 #define CC_BSV_KPICV_IS_LOCKED_ERR (CC_BSV_BASE_ERROR + 0x00000017)

Defines the error code for when the Kpicv is locked.

2.5.7.4.45 #define CC_BSV_OTP_WRITE_CMP_FAIL_ERR (CC_BSV_BASE_ERROR + 0x0000000A)

Defines the error code for when OTP write compare fails.

2.5.7.4.46 #define CC_BSV_VER_IS_NOT_INITIALIZED_ERR (CC_BSV_BASE_ERROR + 0x00000011)

Defines the error code for when the BSV version is not initialized.

2.6 BSV HAL layer APIs

2.6.1 Modules

- **BSV HAL layer platform-dependent APIs**
- **BSV HAL layer platform-dependent definitions**

2.6.2 Files

- file `cc_hal_sb.h`

2.6.3 Functions

- **CCError_t SB_HalWaitInterrupt** (unsigned long hwBaseAddress, uint32_t data)
- void **SB_HalMaskInterrupt** (unsigned long hwBaseAddress, uint32_t data)
- void **SB_HalClearInterruptBit** (unsigned long hwBaseAddress, uint32_t data)

2.6.4 Detailed description

Contains the BSV HAL layer APIs and definitions.

2.6.5 Function documentation

2.6.5.1 void SB_HalClearInterruptBit (unsigned long hwBaseAddress, uint32_t data)

This function clears the interrupt bits that are provided in data in the Interrupt Clear Register (ICR).

Returns:

void

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`data`	The interrupt bits to clear.

2.6.5.2 void SB_HalMaskInterrupt (unsigned long hwBaseAddress, uint32_t data)

This function masks the interrupt bits that are provided in data in the Interrupt Mask Register (IMR).

Returns:

Void

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`data`	The interrupt bits to mask.

2.6.5.3 CCError_t **SB_HalWaitInterrupt (unsigned long hwBaseAddress, uint32_t data)**

This function waits for the Interrupt Request Register (IRR) signal, according to the bits provided in data.

The existing implementation performs a "busy wait" on the IRR. You must adapt this to your system.

Returns:

A non-zero value from `secureboot_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`data`	The interrupt bits to wait for.

2.6.6 BSV HAL layer platform-dependent APIs

Contains functions used for the HAL layer of the Boot Services.

2.6.7 BSV HAL layer platform-dependent definitions

2.6.7.1 Files

- file `cc_hal_sb_plat.h`

2.6.7.2 Macros

- #define **SB_HAL_READ_REGISTER**(addr, val) ((val) = (*((volatile uint32_t*)(addr))))
- #define **SB_HAL_WRITE_REGISTER**(addr, val) ((*((volatile uint32_t*)(addr))) = (unsigned long)(val))

2.6.7.3 Detailed Description

Contains definitions that are used for the BSV HAL layer APIs.

2.6.7.4 Macro Definition Documentation

2.6.7.4.1 #define SB_HAL_READ_REGISTER(addr, val) ((val) = (*((volatile uint32_t*)(addr))))

Reads a 32-bit value from a CryptoCell-312 memory-mapped register.

2.6.7.4.2 #define SB_HAL_WRITE_REGISTER(addr, val) ((*((volatile uint32_t*)(addr))) = (unsigned long)(val))

Writes a 32-bit value to a CryptoCell-312 memory-mapped register.

This macro must be modified to make the operation synchronous. This means that the write operation must complete and the new value must be written to the register before the macro returns. The mechanisms required to achieve this are architecture dependent. For example, the memory barrier in Arm architecture.

2.7 Hash definitions and types

2.7.1 Files

- file `cc_sec_defs.h`

2.7.2 Macros

- #define **HASH_BLOCK_SIZE_IN_WORDS** 16
- #define **HASH_RESULT_SIZE_IN_WORDS** 8
- #define **HASH_RESULT_SIZE_IN_BYTES** 32

2.7.3 typedefs

- typedef uint32_t **CCHashResult_t**[**HASH_RESULT_SIZE_IN_WORDS**]

2.7.4 Detailed description

Contains general hash definitions and types.

2.7.5 Macro definition documentation

2.7.5.1 #define HASH_BLOCK_SIZE_IN_WORDS 16

Defines the hash block size in words.

2.7.5.2 #define HASH_RESULT_SIZE_IN_BYTES 32

Defines the SHA-256 result size in Bytes.

2.7.5.3 #define HASH_RESULT_SIZE_IN_WORDS 8

Defines the SHA-256 result size in words.

2.7.6 typedef documentation

2.7.6.1 typedef uint32_t CCHashResult_t[HASH_RESULT_SIZE_IN_WORDS]

Defines the hash result array.

2.8 BSV PAL layer APIs

2.8.1 Modules

- **BSV PAL layer platform-dependent definitions**
- **BSV PAL layer platform-dependent types**
- **BSV PAL platform-dependent type definitions**

2.8.2 Detailed description

Contains the BSV PAL layer APIs and definitions.

2.8.3 BSV PAL layer platform-dependent definitions

2.8.3.1 Files

- file `cc_pal_sb_plat.h`

2.8.3.2 Typedefs

- typedef uint32_t **CCDmaAddr_t**
- typedef uint32_t **CCAddr_t**

2.8.3.3 Detailed Description

Contains the definitions that are used for BSV PAL layer APIs.

2.8.3.4 Typedef Documentation

2.8.3.4.1 typedef uint32_t CCAddr_t

Defines the CryptoCell address type.

2.8.3.4.2 typedef uint32_t CCDmaAddr_t

Defines the DMA address type.

2.8.4 BSV PAL layer platform-dependent types

2.8.4.1 Files

- file `cc_pal_types.h`

2.8.4.2 Macros

- #define **CC_SUCCESS** 0UL
- #define **CC_FAIL** 1UL
- #define **CC_OK** 0
- #define **CC_UNUSED_PARAM**(prm) ((void)prm)
- #define **CC_MAX_UINT32_VAL** (0xFFFFFFFF)
- #define **CC_MIN**(a, b) (((a) < (b)) ? (a): (b))
- #define **CC_MAX**(a, b) (((a) > (b)) ? (a): (b))
- #define **CALC_FULL_BYTES**(numBits) ((numBits)/CC_BITS_IN_BYTE + (((numBits) & (CC_BITS_IN_BYTE-1)) > 0))
- #define **CALC_FULL_32BIT_WORDS**(numBits) ((numBits)/CC_BITS_IN_32BIT_WORD + (((numBits) & (CC_BITS_IN_32BIT_WORD-1)) > 0))
- #define **CALC_32BIT_WORDS_FROM_BYTES**(sizeBytes) ((sizeBytes)/CC_32BIT_WORD_SIZE + (((sizeBytes) & (CC_32BIT_WORD_SIZE-1)) > 0))
- #define **ROUNDUP_BITS_TO_32BIT_WORD**(numBits) (**CALC_FULL_32BIT_WORDS**(numBits) *CC_BITS_IN_32BIT_WORD)
- #define **ROUNDUP_BITS_TO_BYTES**(numBits) (**CALC_FULL_BYTES**(numBits) *CC_BITS_IN_BYTE)
- #define **ROUNDUP_BYTES_TO_32BIT_WORD**(sizeBytes) (**CALC_32BIT_WORDS_FROM_BYTES**(sizeBytes) *CC_32BIT_WORD_SIZE)

2.8.4.3 Enumerations

- enum **CCBool** { **CC_FALSE** = 0, **CC_TRUE** = 1 }

2.8.4.4 Detailed Description

Contains the definitions and types that are used for the BSV PAL layer APIs.

2.8.4.5 Macro Definition Documentation

2.8.4.5.1 #define CALC_32BIT_WORDS_FROM_BYTES(sizeBytes) ((sizeBytes)/CC_32BIT_WORD_SIZE + (((sizeBytes) & (CC_32BIT_WORD_SIZE-1)) > 0))

This macro calculates the number of full 32-bit words from bytes, where 3 bytes are 1 word.

2.8.4.5.2 #define CALC_FULL_32BIT_WORDS(numBits) ((numBits)/CC_BITS_IN_32BIT_WORD + (((numBits) & (CC_BITS_IN_32BIT_WORD-1)) > 0))

This macro calculates the number of full 32-bit words from bits, where 31 bits are 1 word.

2.8.4.5.3 #define CALC_FULL_BYTES(numBits) ((numBits)/CC_BITS_IN_BYTE + (((numBits) & (CC_BITS_IN_BYTE-1)) > 0))

This macro calculates the number of full bytes from bits, where 7 bits are 1 byte.

2.8.4.5.4 #define CC_FAIL 1UL

Failure definition.

2.8.4.5.5 #define CC_MAX(a, b) (((a) > (b)) ? (a): (b))

Defines maximal calculation.

2.8.4.5.6 #define CC_MAX_UINT32_VAL (0xFFFFFFFF)

Defines the maximal uint32 value.

2.8.4.5.7 #define CC_MIN(a, b) (((a) < (b)) ? (a): (b))

Defines minimal calculation.

2.8.4.5.8 #define CC_OK 0

Success (OK) definition.

2.8.4.5.9 #define CC_SUCCESS 0UL

Success definition.

2.8.4.5.10 #define CC_UNUSED_PARAM(prm) ((void)prm)

This macro handles unused parameters in the code, to avoid compilation warnings.

2.8.4.5.11 #define ROUNDUP_BITS_TO_32BIT_WORD(numBits) (CALC_FULL_32BIT_WORDS(numBits) *CC_BITS_IN_32BIT_WORD)

This macro rounds up bits to 32-bit words.

2.8.4.5.12 #define ROUNDUP_BITS_TO_BYTES(numBits) (CALC_FULL_BYTES(numBits) *CC_BITS_IN_BYTE)

This macro rounds up bits to bytes.

2.8.4.5.13 #define ROUNDUP_BYTES_TO_32BIT_WORD(sizeBytes) (CALC_32BIT_WORDS_FROM_BYTES (sizeBytes) *CC_32BIT_WORD_SIZE)

This macro rounds up bytes to 32-bit words.

2.8.4.6 Enumeration Type Documentation

2.8.4.6.1 enum CCBool

Definition for Boolean type.

Enumerator:

Enum	Description
`CC_FALSE`	Boolean false definition.
`CC_TRUE`	Boolean true definition.

2.8.5 BSV PAL platform-dependent type definitions

2.8.5.1 Files

- file `cc_pal_types_plat.h`

2.8.5.2 Macros

- #define **CCError_t CCStatus**
- #define **CC_INFINITE** 0xFFFFFFFF
- #define **CEXPORT_C**
- #define **CIMPORT_C**

2.8.5.3 Typedefs

- typedef uintptr_t **CCVirtAddr_t**
- typedef uint32_t **CCBool_t**
- typedef uint32_t **CCStatus**

2.8.5.4 Detailed Description

Contains basic definitions that are used for the BSV PAL layer APIs.

2.8.5.5 Macro Definition Documentation

2.8.5.5.1 #define CC_INFINITE 0xFFFFFFFF

Defines an infinite value used to define an unlimited timeframe.

2.8.5.5.2 #define CCError_t CCStatus

Defines the type for a returned error.

2.8.5.5.3 #define CEXPORT_C

Defines the type for a C export.

2.8.5.5.4 #define CIMPORT_C

Defines the type for a C import.

2.8.5.6 Typedef Documentation

2.8.5.6.1 typedef uint32_t CCBool_t

Defines the type for a Boolean variable.

2.8.5.6.2 typedef uint32_t CCStatus

Defines the type for a returned status.

2.8.5.6.3 typedef uintptr_t CCVirtAddr_t

Defines the type for a virtual address.

2.9 OTP memory read and write operations

2.9.1 Files

- file `bsv_otp_api.h`

2.9.2 Functions

- **CCError_t CC_BsvOTPWordRead** (unsigned long hwBaseAddress, uint32_t otpAddress, uint32_t *pOtpWord)
- **CCError_t CC_BsvOTPWordWrite** (unsigned long hwBaseAddress, uint32_t otpAddress, uint32_t otpWord)

2.9.3 Detailed description

Contains functions that access the OTP memory for read and write operations.

You can replace this implementation depending on memory requirements.

2.9.4 Function documentation

2.9.4.1 CCError_t **CC_BsvOTPWordRead (unsigned long hwBaseAddress, uint32_t otpAddress, uint32_t * pOtpWord)**

This function retrieves a 32-bit OTP memory word from a given address.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`otpAddress`	The address of the word in OTP memory to read from.
out	`pOtpWord`	The contents of the word in the OTP memory address defined in otpAddress.

2.9.4.2 CCError_t **CC_BsvOTPWordWrite (unsigned long hwBaseAddress, uint32_t otpAddress, uint32_t otpWord)**

This function writes a 32-bit OTP memory word to a given address.

Before writing, the function reads the current value in the OTP memory word and performs a bit-wise OR to generate the expected value.

After writing, the word is read in the new address and compared to the expected value.

Returns:

CC_OK on success.

A non-zero value from sbrom_bsv_error.h on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`otpAddress`	The address of the word in OTP memory to write to.
in	`otpWord`	The contents of the word to write to the address defined in otpAddress.

2.10 Secure Boot APIs and definitions

2.10.1 Modules

- **Secure Boot APIs**
- **Secure Boot basic type definitions**
- **Secure Boot definitions**
- **Secure Boot error codes**

2.10.2 Detailed description

Contains Secure Boot APIs and definitions.

2.10.3 Secure Boot APIs

2.10.3.1 Files

- file `bootimagesverifier_api.h`

2.10.3.2 Functions

- **CCError_t CC_SbCertChainVerificationInit** (**CCSbCertInfo_t** *certPkgInfo)
- **CCError_t CC_SbCertVerifySingle** (**CCSbFlashReadFunc** flashReadFunc, void *userContext, unsigned long hwBaseAddress, **CCAddr_t** certStoreAddress, **CCSbCertInfo_t** *certPkgInfo, uint32_t *pHeader, uint32_t headerSize, uint32_t *pWorkspace, uint32_t workspaceSize)

2.10.3.3 Detailed Description

Contains Secure Boot APIs.

2.10.3.4 Function Documentation

2.10.3.4.1 CCError_t **CC_SbCertChainVerificationInit (**CCSbCertInfo_t *** certPkgInfo)**

This function initializes the Secure Boot certificate chain processing and the internal data fields of the certificate package.

It must be the first API called when processing Secure Boot certificate chain.

Returns:

CC_OK on success.

A non-zero value from sbrom_bsv_error.h on failure.

Parameters:

I/O	Parameter	Description
in, out	`certPkgInfo`	Pointer to the information about the certificate package

2.10.3.4.2 CCError_t **CC_SbCertVerifySingle (**CCSbFlashReadFunc **flashReadFunc, void * userContext, unsigned long hwBaseAddress,** CCAddr_t **certStoreAddress,** CCSbCertInfo_t * **certPkgInfo, uint32_t * pHeader, uint32_t headerSize, uint32_t * pWorkspace, uint32_t workspaceSize)**

This function verifies a single certificate package (containing either a key or content certificate).

It verifies the following:

- The public key (as saved in the certificate) against its Hash that is either found in the OTP memory (HBK) or in certPkgInfo.
- The certificate's RSA signature.
- The SW version in the certificate must be higher than or equal to the minimum SW version, as recorded on the device and passed in certPkgInfo.
- Each SW module against its Hash in the certificate (for content certificates).

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`flashReadFunc`	Pointer to the flash read function.
in	`userContext`	An additional pointer for flashRead usage. May be NULL.
in	`hwBaseAddress`	CryptoCell HW registers' base address.
in	`certStoreAddress`	Flash address where the certificate is located. This address is provided to flashReadFunc.
in, out	`certPkgInfo`	Pointer to the information about the certificate package.
in, out	`pHeader`	Pointer to a buffer used for extracting the X509 TBS Headers. Must be NULL for proprietary certificates.
in	`headerSize`	The size of pHeader in bytes. Must be 0 for proprietary certificates.
in	`pWorkspace`	A buffer for the internal use of the function.
in	`workspaceSize`	The size of the workspace in bytes. Must be at least CC_SB_MIN_WORKSPACE_SIZE_IN_BYTES.

2.10.4 Secure Boot basic type definitions

Contains basic type definitions for the Secure Boot.

2.10.5 Secure Boot definitions

2.10.5.1 Files

- file `bootimagesverifier_def.h`

2.10.5.2 Macros

- #define **CC_SB_MAX_NUM_OF_IMAGES** 16
- #define **CC_SB_MAX_CERT_SIZE_IN_BYTES** (0x700)
- #define **CC_SB_MAX_CERT_SIZE_IN_WORDS** (**CC_SB_MAX_CERT_SIZE_IN_BYTES**/CC_32BIT_WORD_SIZE)
- #define **CC_SB_MIN_DBG_WORKSPACE_SIZE_IN_BYTES** (0x350)
- #define **CC_SB_MIN_WORKSPACE_SIZE_IN_BYTES** (**CC_SB_MAX_CERT_SIZE_IN_BYTES** + **CC_MAX**(**CC_SB_MIN_DBG_WORKSPACE_SIZE_IN_BYTES**, CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES))

2.10.5.3 Detailed Description

Contains definitions used for the Secure Boot and Secure Debug APIs.

2.10.5.4 Macro Definition Documentation

2.10.5.4.1 #define CC_SB_MAX_CERT_SIZE_IN_BYTES (0x700)

Defines the maximal size of the certificate in bytes.

2.10.5.4.2 #define CC_SB_MAX_CERT_SIZE_IN_WORDS (CC_SB_MAX_CERT_SIZE_IN_BYTES/CC_32BIT_WORD_SIZE)

Defines the maximal size of the certificate in words.

2.10.5.4.3 #define CC_SB_MAX_NUM_OF_IMAGES 16

Defines the maximal number of SW images per content certificate.

2.10.5.4.4 #define CC_SB_MIN_DBG_WORKSPACE_SIZE_IN_BYTES (0x350)

Defines the maximal size of the Secure Debug workspace in bytes. This workspace is used to store the RSA parameters (such as the modulus and the signature).

2.10.5.4.5 #define CC_SB_MIN_WORKSPACE_SIZE_IN_BYTES (CC_SB_MAX_CERT_SIZE_IN_BYTES + CC_MAX(CC_SB_MIN_DBG_WORKSPACE_SIZE_IN_BYTES, CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES))

Defines the minimal size of the workspace.

The Secure Boot APIs use a temporary workspace for processing the data that is read from the flash, prior to loading the SW modules to their designated memory addresses. This size of this workspace must adhere to the following guidelines:

- Be large enough to accommodate the size of the certificates.
- Be twice the size of the data that is read from flash in each processing round.

The definition of CC_SB_MIN_WORKSPACE_SIZE_IN_BYTES is comprised of CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES and additional space for the certificate itself. The certificate resides in the workspace at the same time the SW images data is processed.

The optimal size of the data to read in each processing round is 4KB, based on the standard flash memory page size. Therefore, the size of the double buffer (CC_CONFIG_SB_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES) is defined by default as 8KB in the project configuration file. This can be changed to accommodate the optimal value in different environments. CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES is defined by the Boot Services makefile as equal to CC_CONFIG_SB_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES.

When writing code that uses the Secure Boot APIs, and includes the `bootimagesverifier_def.h` file, the value of CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES must be defined by your makefile to be exactly the same value as was used when compiling the SBROM library. Additionally, CC_SB_X509_CERT_SUPPORTED must be defined in the makefile, according to CC_CONFIG_SB_X509_CERT_SUPPORTED definition.

The size of CC_DOUBLE_BUFFER_MAX_SIZE_IN_BYTES must be a multiple of the hash SHA256 block size (64 bytes).

2.10.6 Secure Boot error codes

2.10.6.1 Files

- file `secureboot_error.h`

2.10.6.2 Macros

- #define **CC_SECUREBOOT_BASE_ERROR** 0xF0000000
- #define **CC_SECUREBOOT_LAYER_BASE_ERROR** 0x01000000
- #define **CC_SB_VERIFIER_LAYER_PREFIX** 1
- #define **CC_SB_DRV_LAYER_PREFIX** 2
- #define **CC_SB_SW_REVOCATION_LAYER_PREFIX** 3
- #define **CC_SB_HAL_LAYER_PREFIX** 6
- #define **CC_SB_RSA_LAYER_PREFIX** 7
- #define **CC_SB_VERIFIER_CERT_LAYER_PREFIX** 8
- #define **CC_SB_X509_CERT_LAYER_PREFIX** 9
- #define **CC_BOOT_IMG_VERIFIER_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_VERIFIER_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_SB_HAL_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_HAL_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_SB_RSA_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_RSA_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_BOOT_IMG_VERIFIER_CERT_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_VERIFIER_CERT_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_SB_X509_CERT_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_X509_CERT_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_SB_DRV_BASE_ERROR** (**CC_SECUREBOOT_BASE_ERROR** + **CC_SB_DRV_LAYER_PREFIX***CC_SECUREBOOT_LAYER_BASE_ERROR**)
- #define **CC_SB_HAL_FATAL_ERROR_ERR** (**CC_SB_HAL_BASE_ERROR** + 0x00000001)
- #define **CC_SB_DRV_ILLEGAL_INPUT_ERR** (**CC_SB_DRV_BASE_ERROR** + 0x00000001)
- #define **CC_SB_DRV_ILLEGAL_KEY_ERR** (**CC_SB_DRV_BASE_ERROR** + 0x00000002)
- #define **CC_SB_DRV_ILLEGAL_SIZE_ERR** (**CC_SB_DRV_BASE_ERROR** + 0x00000003)

2.10.6.3 Detailed Description

Defines the types of error codes that the Secure Boot code returns.

2.10.6.4 Macro Definition Documentation

2.10.6.4.1 #define CC_BOOT_IMG_VERIFIER_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_VERIFIER_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of the boot images verifier (0xF1000000).

2.10.6.4.2 #define CC_BOOT_IMG_VERIFIER_CERT_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_VERIFIER_CERT_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of the boot images verifier certificates (0xF8000000).

2.10.6.4.3 #define CC_SB_DRV_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_DRV_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of the cryptographic driver (0xF2000000).

2.10.6.4.4 #define CC_SB_DRV_ILLEGAL_INPUT_ERR (CC_SB_DRV_BASE_ERROR + 0x00000001)

Defines the error code for an illegal input error.

2.10.6.4.5 #define CC_SB_DRV_ILLEGAL_KEY_ERR (CC_SB_DRV_BASE_ERROR + 0x00000002)

Defines the error code for an illegal key error.

2.10.6.4.6 #define CC_SB_DRV_ILLEGAL_SIZE_ERR (CC_SB_DRV_BASE_ERROR + 0x00000003)

Defines the error code for an illegal size error.

2.10.6.4.7 #define CC_SB_DRV_LAYER_PREFIX 2

Defines the error prefix number for the Secure Boot driver layer.

2.10.6.4.8 #define CC_SB_HAL_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_HAL_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of the NVM (0xF4000000).

2.10.6.4.9 #define CC_SB_HAL_FATAL_ERROR_ERR (CC_SB_HAL_BASE_ERROR + 0x00000001)

Defines the error code for a HAL fatal error.

2.10.6.4.10 #define CC_SB_HAL_LAYER_PREFIX 6

Defines the error prefix number for the Secure Boot HAL layer.

2.10.6.4.11 #define CC_SB_RSA_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_RSA_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of the RSA (0xF7000000).

2.10.6.4.12 #define CC_SB_RSA_LAYER_PREFIX 7

Defines the error prefix number for the Secure Boot RSA layer.

2.10.6.4.13 #define CC_SB_SW_REVOCATION_LAYER_PREFIX 3

Defines the error prefix number for the Secure Boot revocation layer.

2.10.6.4.14 #define CC_SB_VERIFIER_CERT_LAYER_PREFIX 8

Defines the error prefix number for the Secure Boot certificate verifier layer.

2.10.6.4.15 #define CC_SB_VERIFIER_LAYER_PREFIX 1

Defines the error prefix number for the Secure Boot verifier layer.

2.10.6.4.16 #define CC_SB_X509_CERT_BASE_ERROR (CC_SECUREBOOT_BASE_ERROR + CC_SB_X509_CERT_LAYER_PREFIX*CC_SECUREBOOT_LAYER_BASE_ERROR)

Defines the base error code of X.509 certificates (0xF9000000).

2.10.6.4.17 #define CC_SB_X509_CERT_LAYER_PREFIX 9

Defines the error prefix number for the Secure Boot X509 certificate layer.

2.10.6.4.18 #define CC_SECUREBOOT_BASE_ERROR 0xF0000000

Defines the base error code for the different Secure Boot modules.

2.10.6.4.19 #define CC_SECUREBOOT_LAYER_BASE_ERROR 0x01000000

Defines the base error code for the Secure Boot base layer.

2.10.6.5 Secure Boot definitions

2.10.6.5.1 Files

- file `secureboot_defs.h`

2.10.6.5.2 Data Structures

- struct **CCSbCertInfo_t**

2.10.6.5.3 Macros

- #define **SW_REC_SIGNED_DATA_SIZE_IN_BYTES** 44
- #define **SW_REC_NONE_SIGNED_DATA_SIZE_IN_BYTES** 8
- #define **CC_SW_COMP_NO_MEM_LOAD_INDICATION** 0xFFFFFFFFUL

2.10.6.5.4 Detailed Description

Contains basic type definitions for the Secure Boot.

2.10.6.5.5 Macro Definition Documentation

2.10.6.5.6 #define CC_SW_COMP_NO_MEM_LOAD_INDICATION 0xFFFFFFFFUL

Indicates if the SW image needs to be loaded to memory.

2.10.6.5.7 #define SW_REC_NONE_SIGNED_DATA_SIZE_IN_BYTES 8

Defines the additional data size of the SW image.

2.10.6.5.8 #define SW_REC_SIGNED_DATA_SIZE_IN_BYTES 44

Defines the data size of the SW image certificate.

2.11 Secure Debug APIs

2.11.1 Modules

- **Secure Debug APIs and definitions**

2.11.2 Files

- file `secdebug_api.h`

2.11.3 Macros

- #define **CC_BSV_SEC_DEBUG_SOC_ID_SIZE** 0x20

2.11.4 Functions

- **CCError_t CC_BsvSecureDebugSet** (unsigned long hwBaseAddress, uint32_t *pDebugCertPkg, uint32_t certPkgSize, uint32_t *pEnableRmaMode, uint32_t *pWorkspace, uint32_t workspaceSize)

2.11.5 Detailed description

Contains the Secure Debug APIs.

2.11.6 Macro definition documentation

2.11.6.1 #define CC_BSV_SEC_DEBUG_SOC_ID_SIZE 0x20

The Size of the SoC_ID.

2.11.7 Function documentation

2.11.7.1 CCError_t CC_BsvSecureDebugSet (unsigned long hwBaseAddress, uint32_t * pDebugCertPkg, uint32_t certPkgSize, uint32_t * pEnableRmaMode, uint32_t * pWorkspace, uint32_t workspaceSize)

This function enables or disables debug according to the permissions given in the debug certificate, or predefined values.

Enabling or disabling is done through the DCU registers.

For more information, see the *Arm® CryptoCell-312 Software Integrators Manual*.

Returns:

CC_OK on success.

A non-zero value from `bsv_error.h` on failure.

Parameters:

I/O	Parameter	Description
in	`hwBaseAddress`	The base address for CryptoCell HW registers.
in	`pDebugCertPkg`	A pointer to the Secure Debug certificate package. NULL is a valid value.
in	`certPkgSize`	The size of the certificate package in bytes.
out	`pEnableRmaMode`	The RMA entry flag. Non-zero value indicates entry into RMA LCS is required.
in	`pWorkspace`	A pointer to a buffer used internally.
in	`workspaceSize`	The size of the buffer used internally. Minimal size is CC_SB_MIN_DBG_WORKSPACE_SIZE_IN_BYTES.

2.11.8 Secure Debug APIs and definitions

2.11.8.1 Modules

- **Secure Debug APIs**

2.11.8.2 Detailed Description

Contains the Secure Debug APIs and definitions.

2.12 Secure Boot and Secure Debug definitions

2.12.1 Modules

- **Secure Boot and Secure Debug general definitions and structures**
- **Secure Boot and Secure Debug error codes**

2.12.2 Files

- file `cc_crypto_boot_defs.h`

2.12.3 Data structures

- struct **CCSbCertParserSwCompsInfo_t**
- struct **CCSbSwVersion_t**

2.12.4 Macros

- #define **CC_SB_MAX_SIZE_NONCE_BYTES** (2*sizeof(uint32_t))

2.12.5 typedefs

- typedef uint8_t **CCSbNonce_t**[CC_SB_MAX_SIZE_NONCE_BYTES]

2.12.6 Enumerations

- enum **CCSbPubKeyIndexType_t** { **CC_SB_HASH_BOOT_KEY_0_128B** = 0, **CC_SB_HASH_BOOT_KEY_1_128B** = 1, **CC_SB_HASH_BOOT_KEY_256B** = 2, **CC_SB_HASH_BOOT_NOT_USED** = 0xF, **CC_SB_HASH_MAX_NUM** = 0x7FFFFFFF }
- enum **CCswCodeEncType_t** { **CC_SB_NO_IMAGE_ENCRYPTION** = 0, **CC_SB_ICV_CODE_ENCRYPTION** = 1, **CC_SB_OEM_CODE_ENCRYPTION** = 2, **CC_SB_CODE_ENCRYPTION_MAX_NUM** = 0x7FFFFFFF }
- enum **CCswLoadVerifyScheme_t** { **CC_SB_LOAD_AND_VERIFY** = 0, **CC_SB_VERIFY_ONLY_IN_FLASH** = 1, **CC_SB_VERIFY_ONLY_IN_MEM** = 2, **CC_SB_LOAD_ONLY** = 3, **CC_SB_LOAD_VERIFY_MAX_NUM** = 0x7FFFFFFF }
- enum **CCswCryptoType_t** { **CC_SB_HASH_ON_DECRYPTED_IMAGE** = 0, **CC_SB_HASH_ON_ENCRYPTED_IMAGE** = 1, **CC_SB_CRYPTO_TYPE_MAX_NUM** = 0x7FFFFFFF }

2.12.7 Detailed description

Contains Secure Boot and Secure Debug definitions.

2.12.8 Macro definition documentation

2.12.8.1 #define CC_SB_MAX_SIZE_NONCE_BYTES (2*sizeof(uint32_t))

The maximal size of a Secure Boot nonce.

2.12.9 typedef documentation

2.12.9.1 typedef uint8_t CCSbNonce_t[CC_SB_MAX_SIZE_NONCE_BYTES]

The table nonce used in composing IV for decrypting SW components.

2.12.10 Enumeration type documentation

2.12.10.1 enum CCSbPubKeyIndexType_t

Defines a hash boot key.

Enumerator:

Enum	Description
`CC_SB_HASH_BOOT_KEY_0_128B`	A 128-bit truncated SHA-256 digest of public key 0.

Enum	Description
CC_SB_HASH_BOOT_KEY_1_128B	A 128-bit truncated SHA-256 digest of public key 1.
CC_SB_HASH_BOOT_KEY_256B	A 256-bit truncated SHA-256 digest of the public key.
CC_SB_HASH_BOOT_NOT_USED	Hash boot key not used in Secure Boot.
CC_SB_HASH_MAX_NUM	For internal use.

2.12.10.2 enum CCswCodeEncType_t

Defines the types of code encryption for SW images.

Enumerator:

Enum	Description
CC_SB_NO_IMAGE_ENCRYPTION	Plain SW image.
CC_SB_ICV_CODE_ENCRYPTION	Use Kceicv to encrypt the SW image.
CC_SB_OEM_CODE_ENCRYPTION	use Kce to encrypt the SW image.
CC_SB_CODE_ENCRYPTION_MAX_NUM	For internal use.

2.12.10.3 enum CCswCryptoType_t

Defines the cryptographic types for SW images.

Enumerator:

Enum	Description
CC_SB_HASH_ON_DECRYPTED_IMAGE	AES to HASH.
CC_SB_HASH_ON_ENCRYPTED_IMAGE	AES and HASH.
CC_SB_CRYPTO_TYPE_MAX_NUM	For internal use.

2.12.10.4 enum CCswLoadVerifyScheme_t

Defines the loading schemes for SW images.

For more information on each scheme, see the *Arm® CryptoCell-312 Software Integrators Manual*.

Enumerator:

Enum	Description
CC_SB_LOAD_AND_VERIFY	Load & Verify from flash to memory.
CC_SB_VERIFY_ONLY_IN_FLASH	Verify only in flash.
CC_SB_VERIFY_ONLY_IN_MEM	Verify only in memory.
CC_SB_LOAD_ONLY	Load only from flash to memory.

Enum	Description
`CC_SB_LOAD_VERIFY_MAX_NUM`	For internal use.

2.12.11 Secure Boot and Secure Debug general definitions and structures

2.12.11.1 Files

- file `secureboot_gen_defs.h`

2.12.11.2 Typedefs

- typedef uint32_t **CCSbCertPubKeyHash_t**[**HASH_RESULT_SIZE_IN_WORDS**]
- typedef uint32_t **CCSbCertSocId_t**[**HASH_RESULT_SIZE_IN_WORDS**]
- typedef uint32_t(***CCSbFlashReadFunc**) (**CCAddr_t** flashAddress, uint8_t *memDst, uint32_t sizeToRead, void *context)
- typedef uint32_t(***CCBsvFlashWriteFunc**) (**CCAddr_t** flashAddress, uint8_t *memSrc, uint32_t sizeToWrite, void *context)

2.12.11.3 Detailed Description

Contains all the definitions and structures used for the Secure Boot and Secure Debug APIs.

2.12.11.4 Typedef Documentation

2.12.11.4.1 typedef uint32_t(*CCBsvFlashWriteFunc) (CCAddr_t flashAddress, uint8_t *memSrc, uint32_t sizeToWrite, void *context)

Typedef of the Flash write function that you must implement.

Used for writing authenticated and decrypted SW modules to Flash memory.

2.12.11.4.2 typedef uint32_t CCSbCertPubKeyHash_t[HASH_RESULT_SIZE_IN_WORDS]

Definition of a public key hash array.

2.12.11.4.3 typedef uint32_t CCSbCertSocId_t[HASH_RESULT_SIZE_IN_WORDS]

Definition of a SoC_ID array.

2.12.11.4.4 typedef uint32_t(*CCSbFlashReadFunc) (CCAddr_t flashAddress, uint8_t *memDst, uint32_t sizeToRead, void *context)

Typedef of the Flash read function that you must implement.

Used for reading the certificates and SW modules from Flash memory.

It is your responsibility to verify that this function does not copy data from restricted memory regions.

2.12.12 Secure Boot and Secure Debug error codes

2.12.12.1 Files

- file `bootimagesverifier_error.h`

2.12.12.2 Macros

- #define **CC_BOOT_IMG_VERIFIER_INV_INPUT_PARAM** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000001)
- #define **CC_BOOT_IMG_VERIFIER_OTP_VERSION_FAILURE** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000002)
- #define **CC_BOOT_IMG_VERIFIER_CERT_MAGIC_NUM_INCORRECT** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000003)
- #define **CC_BOOT_IMG_VERIFIER_CERT_VERSION_NUM_INCORRECT** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000004)
- #define **CC_BOOT_IMG_VERIFIER_SW_VER_SMALLER_THAN_MIN_VER** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000005)
- #define **CC_BOOT_IMG_VERIFIER_PUB_KEY_HASH_VALIDATION_FAILURE** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000006)
- #define **CC_BOOT_IMG_VERIFIER_RSA_SIG_VERIFICATION_FAILED** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000007)
- #define **CC_BOOT_IMG_VERIFIER_WORKSPACE_SIZE_TOO_SMALL** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000008)
- #define **CC_BOOT_IMG_VERIFIER_SW_COMP_FAILED_VERIFICATION** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000009)
- #define **CC_BOOT_IMG_VERIFIER_CERT_SW_VER_ILLEGAL** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000000D)
- #define **CC_BOOT_IMG_VERIFIER_SW_COMP_SIZE_IS_NULL** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000011)
- #define **CC_BOOT_IMG_VERIFIER_PUBLIC_KEY_HASH_EMPTY** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000014)
- #define **CC_BOOT_IMG_VERIFIER_ILLEGAL_LCS_FOR_OPERATION_ERR** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000015)
- #define **CC_BOOT_IMG_VERIFIER_PUB_KEY_ALREADY_PROGRAMMED_ERR** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000016)
- #define **CC_BOOT_IMG_VERIFIER_OTP_WRITE_FAIL_ERR** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000017)
- #define **CC_BOOT_IMG_VERIFIER_INCORRECT_CERT_TYPE** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000018)

- #define **CC_BOOT_IMG_VERIFIER_ILLEGAL_HBK_IDX** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000019)
- #define **CC_BOOT_IMG_VERIFIER_PUB_KEY1_NOT_PROGRAMMED_ERR** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000001A)
- #define **CC_BOOT_IMG_VERIFIER_CERT_VER_VAL_ILLEGAL** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000001C)
- #define **CC_BOOT_IMG_VERIFIER_CERT_DECODING_ILLEGAL** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000001D)
- #define **CC_BOOT_IMG_VERIFIER_ILLEGAL_KCE_IN_RMA_STATE** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000001E)
- #define **CC_BOOT_IMG_VERIFIER_ILLEGAL_SOC_ID_VALUE** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x0000001F)
- #define **CC_BOOT_IMG_VERIFIER_ILLEGAL_NUM_OF_IMAGES** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000020)
- #define **CC_BOOT_IMG_VERIFIER_SKIP_PUBLIC_KEY_VERIFY** (**CC_BOOT_IMG_VERIFIER_BASE_ERROR** + 0x00000014)

2.12.12.3 Detailed Description

Defines the error codes used for Secure Boot and Secure Debug APIs.

2.12.12.4 Macro Definition Documentation

2.12.12.4.1 #define CC_BOOT_IMG_VERIFIER_CERT_DECODING_ILLEGAL (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000001D)

Defines the error code for an illegal certificate decoding value.

2.12.12.4.2 #define CC_BOOT_IMG_VERIFIER_CERT_MAGIC_NUM_INCORRECT (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000003)

Defines the error code for an illegal certificate magic number.

2.12.12.4.3 #define CC_BOOT_IMG_VERIFIER_CERT_SW_VER_ILLEGAL (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000000D)

Defines the error code for illegal SW version or illegal ID of the SW version.

2.12.12.4.4 #define CC_BOOT_IMG_VERIFIER_CERT_VER_VAL_ILLEGAL (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000001C)

Defines the error code for an illegal certificate version value.

2.12.12.4.5 #define CC_BOOT_IMG_VERIFIER_CERT_VERSION_NUM_INCORRECT (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000004)

Defines the error code for an illegal certificate version.

2.12.12.4.6 #define CC_BOOT_IMG_VERIFIER_ILLEGAL_HBK_IDX (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000019)

Defines the error code for an illegal index of the hash boot key.

2.12.12.4.7 #define CC_BOOT_IMG_VERIFIER_ILLEGAL_KCE_IN_RMA_STATE (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000001E)

Defines the error code for an illegal Kce in RMA LCS.

2.12.12.4.8 #define CC_BOOT_IMG_VERIFIER_ILLEGAL_LCS_FOR_OPERATION_ERR (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000015)

Defines the error code for illegal a life-cycle state (LCS) for the requested operation.

2.12.12.4.9 #define CC_BOOT_IMG_VERIFIER_ILLEGAL_NUM_OF_IMAGES (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000020)

Defines the error code for an illegal number of SW images per content certificate.

2.12.12.4.10 #define CC_BOOT_IMG_VERIFIER_ILLEGAL_SOC_ID_VALUE (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000001F)

Defines the error code for an illegal SoC_ID value.

2.12.12.4.11 #define CC_BOOT_IMG_VERIFIER_INCORRECT_CERT_TYPE (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000018)

Defines the error code for an incorrect certificate type.

2.12.12.4.12 #define CC_BOOT_IMG_VERIFIER_INV_INPUT_PARAM (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000001)

Defines the error code for invalid input parameters.

2.12.12.4.13 #define CC_BOOT_IMG_VERIFIER_OTP_VERSION_FAILURE (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000002)

Defines the error code for an invalid OTP version.

2.12.12.4.14 #define CC_BOOT_IMG_VERIFIER_OTP_WRITE_FAIL_ERR (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000017)

Defines the error code for OTP write failure.

2.12.12.4.15 #define CC_BOOT_IMG_VERIFIER_PUB_KEY1_NOT_PROGRAMMED_ERR (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x0000001A)

Defines the error code for when the hash boot key of ICV is not programmed.

2.12.12.4.16 #define CC_BOOT_IMG_VERIFIER_PUB_KEY_ALREADY_PROGRAMMED_ERR (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000016)

Defines the error code for when the hash of the public key is already programmed.

2.12.12.4.17 #define CC_BOOT_IMG_VERIFIER_PUB_KEY_HASH_VALIDATION_FAILURE (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000006)

Defines the error code for when comparing the public key to the OTP value fails.

2.12.12.4.18 #define CC_BOOT_IMG_VERIFIER_PUBLIC_KEY_HASH_EMPTY (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000014)

Defines the error code for when the hash of public key is not burned yet.

2.12.12.4.19 #define CC_BOOT_IMG_VERIFIER_RSA_SIG_VERIFICATION_FAILED (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000007)

Defines the error code for when verification of the certificate RSA signature fails.

2.12.12.4.20 #define CC_BOOT_IMG_VERIFIER_SKIP_PUBLIC_KEY_VERIFY (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000014)

Defines the error code for unnecessary request to verify hashed public key.

2.12.12.4.21 #define CC_BOOT_IMG_VERIFIER_SW_COMP_FAILED_VERIFICATION (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000009)

Defines the error code for SW image verification failure.

2.12.12.4.22 #define CC_BOOT_IMG_VERIFIER_SW_COMP_SIZE_IS_NULL (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000011)

Defines the error code for an illegal number of SW components (zero).

2.12.12.4.23 #define CC_BOOT_IMG_VERIFIER_SW_VER_SMALLER_THAN_MIN_VER (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000005)

Defines the error code for a smaller SW version in the certificate than is stored in the OTP.

2.12.12.4.24 #define CC_BOOT_IMG_VERIFIER_WORKSPACE_SIZE_TOO_SMALL (CC_BOOT_IMG_VERIFIER_BASE_ERROR + 0x00000008)

Defines the error code for when the workspace buffer provided to the API is too small.

2.13 Data structure documentation

2.13.1 CCSbCertInfo_t struct reference

```
#include <secureboot_defs.h>
```

2.13.1.1 Data Fields

- uint32_t **otpVersion**
- **CCSbPubKeyIndexType_t keyIndex**
- uint32_t **activeMinSwVersionVal**
- **CCHashResult_t pubKeyHash**
- uint32_t **initDataFlag**

2.13.1.2 Detailed description

This structure is used as input to or output of the Secure Boot verification API.

2.13.1.3 Field documentation

2.13.1.3.1 uint32_t CCSbCertInfo_t::activeMinSwVersionVal

The SW version value for the certificate chain.

2.13.1.3.2 uint32_t CCSbCertInfo_t::initDataFlag

[in] Initialization indication. Internal flag.

2.13.1.3.3 CCSbPubKeyIndexType_t CCSbCertInfo_t::keyIndex

The key hash to retrieve: 128-bit HBK0, 128-bit HBK1, or 256-bit HBK.

2.13.1.3.4 uint32_t CCSbCertInfo_t::otpVersion

[in] The NV counter saved in OTP memory.

2.13.1.3.5 CCHashResult_t CCSbCertInfo_t::pubKeyHash

[in/out] In: The hash of the public key (N||Np), to compare to the public key stored in the certificate. Out: The hash of the public key (N||Np) stored in the certificate, to be used for verification of the public key of the next certificate in the chain.

The documentation for this struct was generated from the following file:

- `secureboot_defs.h`

2.13.2 CCSbCertParserSwCompsInfo_t struct reference

```
#include <cc_crypto_boot_defs.h>
```

2.13.2.1 Data Fields

- uint32_t **numOfSwComps**
- **CCswCodeEncType_t swCodeEncType**
- **CCswLoadVerifyScheme_t swLoadVerifyScheme**
- **CCswCryptoType_t swCryptoType**
- **CCSbNonce_t nonce**
- uint8_t ***pSwCompsData**

2.13.2.2 Detailed description

Defines data of SW components.

2.13.2.3 Field documentation

2.13.2.3.1 CCSbNonce_t **CCSbCertParserSwCompsInfo_t::nonce**

The nonce.

2.13.2.3.2 uint32_t CCSbCertParserSwCompsInfo_t::numOfSwComps

The number of SW components.

2.13.2.3.3 uint8_t*CCSbCertParserSwCompsInfo_t::pSwCompsData

A pointer to the start of SW components data.

2.13.2.3.4 CCswCodeEncType_t **CCSbCertParserSwCompsInfo_t::swCodeEncType**

The code encryption type of the SW image.

2.13.2.3.5 CCswCryptoType_t **CCSbCertParserSwCompsInfo_t::swCryptoType**

The cryptographic type of the SW image.

2.13.2.3.6 CCswLoadVerifyScheme_t **CCSbCertParserSwCompsInfo_t::swLoadVerifyScheme**

The loading scheme of the SW image.

The documentation for this struct was generated from the following file:

- `cc_crypto_boot_defs.h`

2.13.3 CCSbSwVersion_t struct reference

```
#include <cc_crypto_boot_defs.h>
```

2.13.3.1 Data Fields

- **CCSbPubKeyIndexType_t keyIndex**
- uint32_t **swVersion**

2.13.3.2 Detailed description

Defines the SW version.

2.13.3.3 Field documentation

2.13.3.3.1 CCSbPubKeyIndexType_t **CCSbSwVersion_t::keyIndex**

The key hash to retrieve: 128-bit HBK0, 128-bit HBK1, or 256-bit HBK.

2.13.3.3.2 uint32_t CCSbSwVersion_t::swVersion

The SW version.

The documentation for this struct was generated from the following file:

- `cc_crypto_boot_defs.h`

3 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.

3.1 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.

3.1.1 Platform HAL integration tests

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

3.1.1.1 Test_ProjInit

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

```
uint32_t Test_ProjInit(void)
```

Returns:

- 0: success.
- 1: failure.

3.1.1.2 Test_ProjFree

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

```
void Test_ProjFree(void)
```

3.1.1.3 Test_ProjPerformPowerOnReset

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

```
void Test_ProjPerformPowerOnReset(void)
```

3.1.1.4 Test_ProjCheckLcs

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

```
uint32_t Test_ProjCheckLcs(uint32_t nextLcs)
```

Returns:

- 0 on success
- `0x00FFFF02` (defined in `test_proj_common.h`) on failure.

Table 3-1 Test_ProjCheckLcs parameters

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

3.1.2 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.

3.1.2.1 Test_PalGetDMABaseAddr

This function returns the start (base) address of the DMA region.

```
unsigned long Test_PalGetDMABaseAddr(void);
```

Returns:

- The DMA base address.
  - When Armv8-M is supported, the Non-Secure DMA base address.

3.1.2.2 Test_PalGetDMABaseAddr_s

This function returns the start (base) address of the Secure DMA region.

```
unsigned long Test_PalGetDMABaseAddr_s(void);
```

Returns:

- The Secure DMA base address.

3.1.2.3 Test_PalGetUnmanagedBaseAddr

This function returns the unmanaged base address.

```
unsigned long Test_PalGetUnmanagedBaseAddr (void);
```

Returns:

- The unmanaged base address.
  - When Armv8-M is supported, the Non-Secure unmanaged base address.

3.1.2.4 Test_PalGetUnmanagedBaseAddr_s

This function returns the Secure unmanaged base address.

```
unsigned long Test_PalGetUnmanagedBaseAddr_s(void);
```

Returns:

- The Secure unmanaged base address.

3.1.2.5 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)
```

The mapping function returns the address of the memory-mapped CryptoCell registers when the following conditions are both true:

- There is no Memory Management Unit.
- The access to the physical memory is straightforward.

Returns:

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

Table 3-2 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.

3.1.2.6 Test_PalUnmapAddr

This function unmaps the given virtual address.

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

Table 3-3 Test_PalUnmapAddr parameters

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

3.1.3 Memory integration tests

The integration test only uses DMA-able continuous memory.

3.1.3.1 Test_PalMemInit

This function initializes DMA memory management.

When Armv8-M is supported, it initializes the Non-Secure DMA memory management.

```
uint32_t Test_PalMemInit(unsigned long newDMABaseAddr,
             unsigned long newUnmanagedBaseAddr,
             size_t DMAsize);
```

Returns:

- 0 on success.
- 1 on failure.

I/O	Parameter	Description
I	`newDMABaseAddr`	The new DMA start address.
I	`newUnmanagedBaseAddr`	The new unmanaged start address.
I	`DMAsize`	The size of the DMA region.

3.1.3.2 Test_PalMemInit_s

This function initializes the Secure DMA memory management.

```
uint32_t Test_PalMemInit_s(unsigned long newDMABaseAddr_s,
               unsigned long newUnmanagedBaseAddr_s,
               size_t SDMAsize);
```

Returns:

- 0 on success.
- 1 on failure.

I/O	Parameter	Description
I	`newDMABaseAddr_s`	New secure DMA start address.
I	`newUnmanagedBaseAddr`	New secure unmanaged start address.
I	`DMAsize`	Secure DMA region size.

3.1.3.3 Test_PalMemFin_s

This function sets the SECURE memory management driver to its initial state.

```
uint32_t Test_PalMemFin_s(void);
```

Returns:

- 0 on success.
- 1 on failure.

3.1.3.4 Test_PalMalloc

This function allocates a buffer in memory.

When Armv8-M is supported, this function is used only for Non-Secure memory allocations.

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

Returns:

- A pointer to the allocated memory.

I/O	Parameter	Description
I	`size`	The requested buffer size in bytes.

3.1.3.5 Test_PalMalloc_s

This function allocates a buffer in the Secure memory region.

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

Returns:

- A pointer to the allocated Secure memory.

I/O	Parameter	Description
I	`size`	The requested buffer size in bytes.

3.1.3.6 Test_PalFree

This function frees allocated memory pointed by `pvAddress`.

When Armv8-M is supported, this function is used only for Non-Secure memory blocks.

```
void Test_PalFree(void *pvAddress);
```

I/O	Parameter	Description
I	`pvAddress`	A pointer to the allocated memory.

3.1.3.7 Test_PalFree_s

This function frees Secure allocated memory pointed by `pvAddress`.

```
void Test_PalFree_s(void *pvAddress);
```

I/O	Parameter	Description
I	`pvAddress`	A pointer to the allocated memory.

3.1.3.8 Test_PalRealloc

This function reallocates the memory block pointed by `pvAddress`.

```
void *Test_PalRealloc (void *pvAddress, size_t newSize);
```

If the function fails to allocate the requested block of memory:

1. A null pointer is returned.
2. The memory block pointed by argument `pvAddress` is not deallocated.

When Armv8-M is supported, this function is used only for Non-Secure memory blocks.

Returns:

- A pointer to the new allocated memory.
- NULL on failure.

I/O	Parameter	Description
I	`pvAddress`	A pointer to the allocated memory.
I	`newSize`	New size of the memory block.

3.1.3.9 Test_PalRealloc_s

This function changes the size of a Secure memory block pointed by `pvAddress`.

```
void *Test_PalRealloc_s (void *pvAddress, size_t newSize);
```

If the function fails to allocate the requested block of memory:

1. A null pointer is returned.
2. The memory block pointed by argument `pvAddress` is not deallocated.

When Armv8-M is supported, this function is used only for Non-Secure memory blocks.

Returns:

- A pointer to the new allocated memory.
- NULL on failure.

I/O	Parameter	Description
I	`pvAddress`	A pointer to the allocated memory.
I	`newSize`	The new size of the memory block.

3.1.3.10 Test_PalDMAContigBufferAlloc

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

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

Returns:

- The address of the allocated buffer.

Table 3-4 Test_PalDMAContigBufferAlloc parameters

I/O	Parameter	Description
I	buffSize	The buffer size in bytes.

3.1.3.11 Test_PalDMAContigBufferAlloc_s

This function allocates a DMA-contiguous buffer in a Secure memory region and returns its address.

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

Returns:

- The address of the Secure allocated buffer.

Table 3-5 Test_PalDMAContigBufferAlloc_s parameters

I/O	Parameter	Description
I	buffSize	The buffer size in bytes.

3.1.3.12 Test_PalDMAContigBufferFree

This function frees resources that `Test_PalDMAContigBufferAlloc()` has previously allocated.

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

Table 3-6 Test_PalDMAContigBufferFree parameters

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

3.1.3.13 Test_PalDMAContigBufferFree_s

This function frees resources that `Test_PalDMAContigBufferAlloc_s()` has previously allocated.

```
void Test_PalDMAContigBufferFree_s(void *pvAddress)
```

Table 3-7 Test_PalDMAContigBufferFree_s parameters

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

3.1.4 Thread integration tests

3.1.4.1 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);
```

To destroy the thread, you must call `Test_PalThreadDestroy()`.

Returns:

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

Table 3-8 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 is DMA-able: True - DMA-able. False - not DMA-able.

3.1.4.2 Test_PalThreadDestroy

This function destroys a thread.

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

Returns:

- 0 on success.
- 1 on failure.

Table 3-9 Test_PalThreadDestroy parameters

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

3.1.4.3 Test_PalThreadJoin

This function waits for a thread to terminate.

```
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 3-10 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.

3.1.5 Time integration tests

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

3.1.5.1 Test_PalDelay

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

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

Table 3-11 Test_PalDelay parameters

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

3.2 Boot Services register integration tests

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

3.2.1 BSVIT_READ_REG

This function reads the register value from `offset`.

```
BSVIT_READ_REG(offset)
```

Returns:

- The value of the register.

Table 3-12 BSVIT_READ_REG parameters

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

3.2.2 BSVIT_WRITE_REG

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

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

Table 3-13 BSVIT_WRITE_REG parameters

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

3.3 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.

3.3.1 BSVIT_WRITE_OTP

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

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

Table 3-14 BSVIT_WRITE_OTP parameters

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

3.3.2 BSVIT_READ_OTP

This function reads the OTP value from `offset`.

```
BSVIT_READ_OTP(offset)
```

Returns:

- The value of the register.

Table 3-15 BSVIT_READ_OTP parameters

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

3.4 Boot Services flash integration tests

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

3.4.1 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 everything that is required to imitate flash operations.

Returns:

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

Table 3-16 bsvIt_flashInit parameters

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

3.4.2 bsvIt_flashFinalize

This function closes a resource that is 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.

3.4.3 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 3-17 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.

3.4.4 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 3-18 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.

3.5 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.

3.5.1 BSVIT_PRINT

This function prints a log entry.

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

Table 3-19 BSVIT_PRINT parameters

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

3.5.2 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 3-20 BSVIT_TEST_START parameters

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

3.5.3 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 when the test completes successfully.

Table 3-21 BSVIT_TEST_RESULT parameters

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

3.5.4 BSVIT_PRINT_ERROR

This function prints an error message to log.

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

Table 3-22 BSVIT_PRINT_ERROR parameters

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

3.5.5 BSVIT_PRINT_DBG

This function prints a debug message.

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

It is skipped unless compiled with `TEST_DEBUG`.

Table 3-23 BSVIT_PRINT_DBG parameters

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

Appendix A Revisions

Table A-1 Differences between 100777 Issue 0102-00 SBROM APIs and 101468 Issue 0103-01

Change	Location	Affects
Renamed the *SBROM APIs* chapter.	Boot Services API layer	r1p3
Removed the `secureboot_basetypes.h` file.	Entire document	r1p3
Added the **Boot Services integration tests** chapter (moved from Software Integrators Manual).	Entire document	r1p3
Added the following tests: • `Test_PalGetDMABaseAddr` • `Test_PalGetDMABaseAddr_s` • `Test_PalGetUnmanagedBaseAddr` • `Test_PalGetUnmanagedBaseAddr_s` • `Test_PalMemInit` • `Test_PalMemInit_s` • `Test_PalMemFin_s` • `Test_PalMalloc` • `Test_PalMalloc_s` • `Test_PalFree` • `Test_PalFree_s` • `Test_PalRealloc` • `Test_PalRealloc_s` • `Test_PalDMAContigBufferAlloc_s` • `Test_PalDMAContigBufferFree_s`	**Boot Services integration tests**	r1p3