Secure Bootloader Guide

M-20892-003
September 2023

© SCILLC, 2023
Previous Edition © 2022

Table of Contents

Page

CHAPTER 1

Introduction

1.1 Summary

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

This group of topics describes the functionality and usage of the secure bootloader with RSL15, along with Platform Security Architecture (PSA) compliance, secure storage, and attestation. RSL15 includes a secure bootloader sample application which can be used by developers to acquire familiarity with the secure bootloader.

1.2 Document Conventions

The following typographical conventions are used in this documentation:

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

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

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

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

Note, Important, Caution, Warning

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

NOTE: Significant supplemental information, hints, or tips.

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

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

WARNING: Information that can prevent harm to humans.

Registers:

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

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

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

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

The firmware provides access to these registers in two ways:

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

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

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

Numbers

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

Sample Rates

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

1.3 Further Reading

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

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

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

- *Armv8M Architecture Reference Manual* (PDF download available from https://developer.arm.com/documentation/ddi0553/latest).
- *Arm Cortex-M33 Processor Technical Reference Manual*, revision r1p0, from https://developer.arm.com/documentation/100230/0100
- *Bluetooth Core Specification version 5.2*, available from https://www.bluetooth.com/specifications/adopted-specifications

- TrustZone documentation available from the Arm website at https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m
- Other ArmCortex-M33 publications, available from the Arm website at https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33

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

CHAPTER 2

Overview

The secure bootloader for RSL15 is a reference application, called *secure_bootloader*, which can be used to develop an end product bootloader that operates in a secure manner. The application has the following four modes providing increasing levels of secure operation, available as needed depending on the end product's use cases:

1. Basic bootloader (non-secure)
2. Secure bootloader (maintains authenticated Root of Trust set up by the ROM)
3. Secure bootloader with secure storage
4. Secure bootloader with secure storage and device attestation

More detailed information about these four modes is found in Chapter 1 "RSL15 Secure Bootloader Usage Options" on page 1.

2.1 Common Features

The four modes of operation build on each other, with available features in one also being available for the next level. For example, all secure bootloader functionality is still available when using the secure storage mode of operation. This is indicated in the "Bootloader Options" figure (Figure 1).

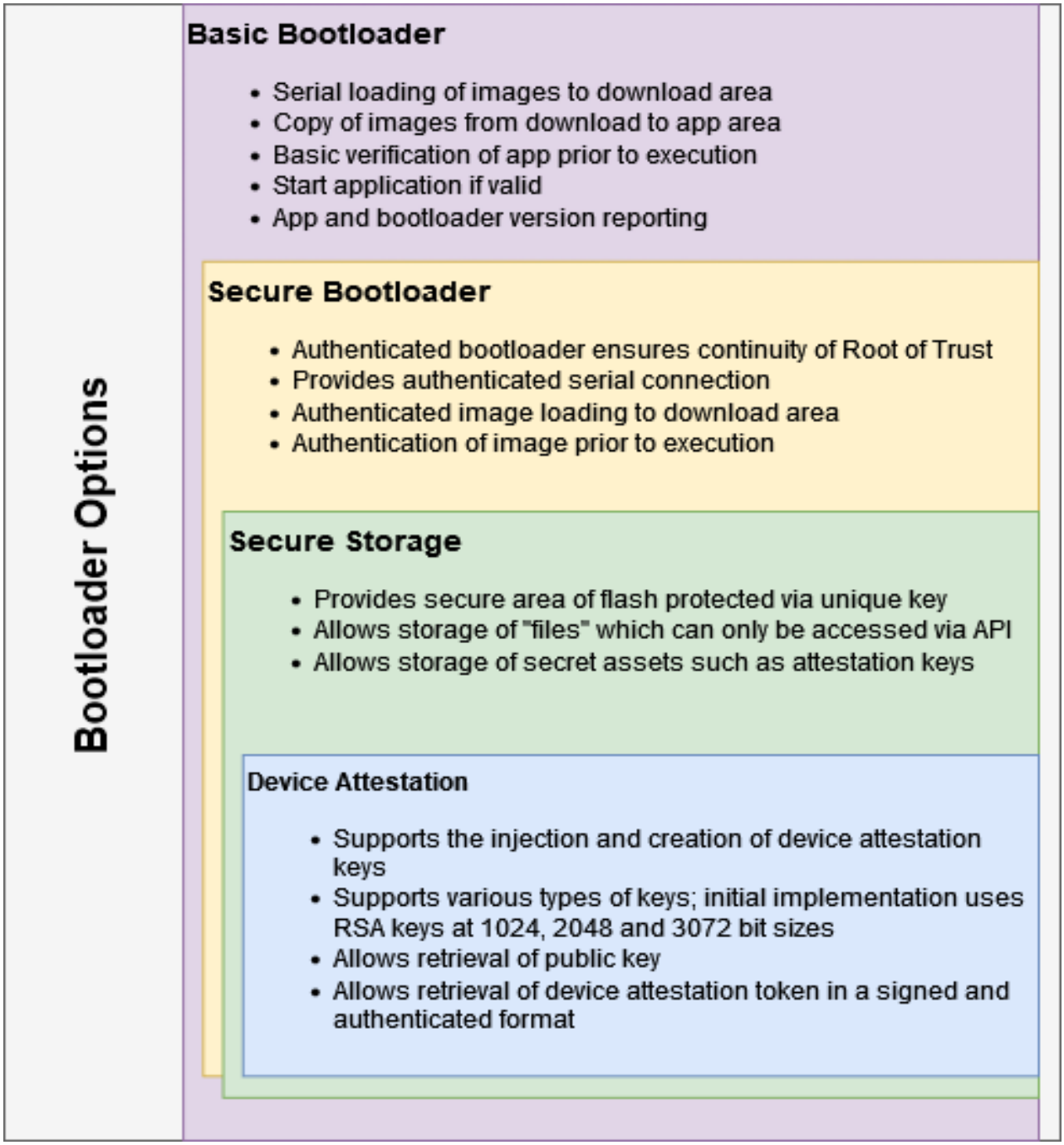

Figure 1. Bootloader Options

The bootloader divides the main flash memory into two areas: the app download area and the app execution area. This division provides a starting point for users who want to use the bootloader for firmware update purposes.

The bootloader application is the first application entry point after reset. It is located at the base address of the main flash (0x00100000).

Upon start, the bootloader checks the app download area for a valid boot image. This application can be a regular application, or a new bootloader. If there is one, the bootloader copies/overwrites this image into the app execution area, invalidates the data in the download area, and boots this new image. If no valid boot image is found in the download area, the bootloader verifies and boots from the app execution area. If no valid image is found in either area, the bootloader defaults to a mode where the UART can be used to provide updates or perform attestation queries. It prints an error message in the RTT Viewer. Sequence diagrams in Chapter 4 "Basic Bootloader" on page 20, Chapter 5 "Secure Bootloader" on page 23, and Chapter 7 "Attestation" on page 29 show detailed step-by-step procedures.

A valid boot image consists of a binary file generated from a *.hex* file. The *.hex* file is created by building the project as usual, but with certain requirements. A valid boot image for an application must have its *.text* section starting at the base address of the app execution area, instead of at the base address of the main flash. Therefore, making a typical sample application—for example, *blinky*—compatible with the bootloader requires modifications to the linker script (*sections.ld*) and startup code (*startup.S*). An example version of *blinky* is provided along with the secure bootloader, in a *utilities* subfolder. The *sections.ld* file must include the correct start locations and sizes for the sections of flash, and the *startup.S* must include the appropriate update to the IVT (ISR Vector Table) for the image descriptor.

IMPORTANT: For proper operation, the defined app excution area and the application's IVT must be aligned to a 512-byte boundary.

Updating the firmware image, as well as basic attestation operations, can be accomplished using the RSLUpdate utility, packaged along with the RSLSec security tools. Examples of how to use RSLUpdate are included in the readme file of the *secure_bootloader* sample application. More information about the secure operation and updating process is described in Section 5 "Secure Bootloader" on page 23.

When GPIO14 is tied to ground, the device enters the bootloader state, expecting updates via the UART.

When GPIO14 it is not tied to ground, and an update is pending, the device tries to process the update and reset; the bootloader tries to execute an application if one if available; and if no application is available, the device goes into the bootloader state.

2.2 RSL15 Secure Bootloader Usage Options

2.2.1 Functionality Access Options

The RSL15 secure bootloader sample application includes the following options for accessing increasing levels of functionality depending on the end product needs:

1. Basic bootloader functionality
2. Secure bootloader with support for providing authenticated and validated loading of applications in addition to the bootloader itself
    - Authenticated transport layer
    - Authenticated images verified on load
    - Authenticated images verified again prior to overwriting an existing image
    - Authenticated images booted by the bootloader
3. Secure storage
    - A limited area of flash memory that is allocated as secure storage
    - Enables the storage and retrieval of encrypted assets
    - Simple filing system that also provides storage for general secure storage in addition to asset storage
4. Attestation
    - Support for the injection or creation of attestation keys
    - Attestation keys are stored in secure storage.
    - A public key can be requested from application code using the bootloader interface.
    - Support for an attestation token, which enumerates the hardware and firmware on the device
    - Support for a standard attestation protocol that is robust against replay attacks
    - Optional support for different types of attestation keys:
        i. AES (not recommended due to lower level of security, and not available in the initial release)
        ii. RSA (provides the smallest increase in bootloader image size, as RSA is already being included in secure boot features)
        iii. ECC (provides good balance between small keys, is robust against attack, requires the most application code to support, and is not available in the initial release)

2.2.2 Configuration

The options are provided as preprocessor definitions, and are available in the API file *bl_options.h*.

2.3 Memory Partitioning Overview

Depending on the feature set used by the bootloader, the amount of flash memory it occupies can change. This allows a bootloader with a lower feature set to be used in cases where, for instance, the Root of Trust or secure storage is not required. When a reduced feature set bootloader is used, the memory partitioning can be changed, allowing for larger user applications to be loaded.

For illustrative purposes, the "Build Configuration Memory Sizes" table (Table 1) shows the expected sizes of each optional build configuration, with expected allocations of memory for application and download areas depending on build options. The precise values are subject to change depending on the actual optimization levels and feature sets you select, but this provides some guideline figures to help you decide which configuration to use.

Table 1. Build Configuration Memory Sizes

		Bootloader		Secure Storage		Application		Download	
		Start Address	Size (KB)	Start Address	Size (KB)	Start Address	Size (KB)	Start Address	Size (KB)
Debug	Basic Bootloader	0x100000	24	0	0	0x106000	236	0x141000	236
	Secure Bootloader	0x100000	52	0	0	0x10D000	224	0x145000	224
	Secure Storage	0x100000	64	0x15A400	11	0x110000	212	0x145000	212
	Attestation	0x100000	108	0x15A400	11	0x11B000	192	0x14B000	192
Release	Basic Bootloader	0x100000	16	0	0	0x104000	240	0x140000	240
	Secure Bootloader	0x100000	44	0	0	0x10B000	228	0x144000	228
	Secure Storage	0x100000	52	0x15A400	11	0x10D000	220	0x144000	220
	Attestation	0x100000	92	0x15A400	11	0x117000	200	0x149000	200

Derivation of these start addresses and sizes is available in *bl_memory.h*; this information is output to the RTT Viewer when debugging the secure bootloader in the onsemi IDE with the RTT Viewer connected. This is also shown in the "Memory Map Diagram" figure (Figure 2).

NOTE: There are variations for an RSL15 device with 284 KB of flash rather than 512 KB.

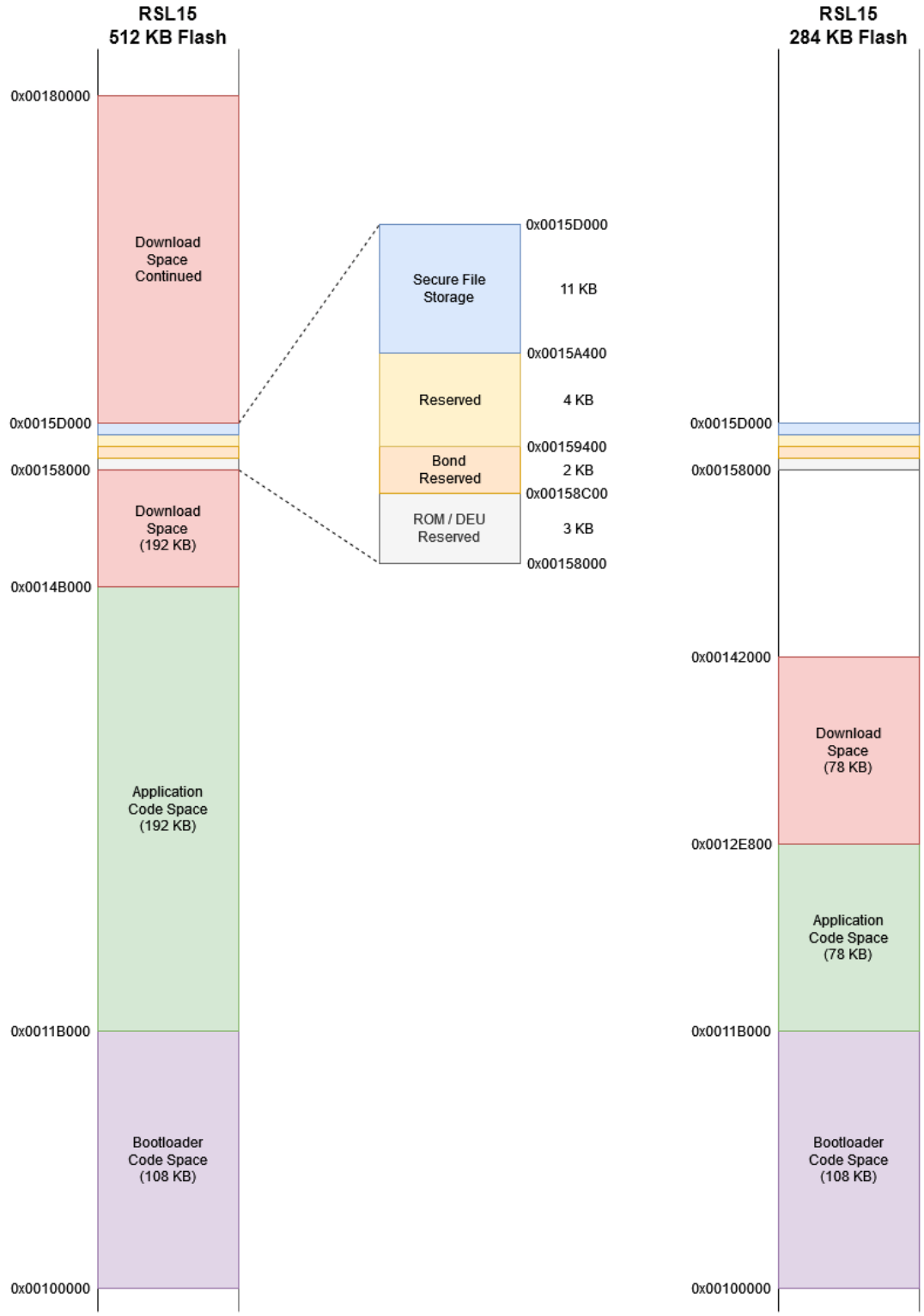

Figure 2. Memory Map Diagram

CHAPTER 3

PSA Compliance Background

The RSL15 secure bootloader is a reference implementation that can be used with associated guidelines to achieve a product that is compliant with Platform Security Architecture (PSA). The RSL15 secure bootloader sample application can be adapted as needed and incorporated into an overall firmware solution. See https://www.psacertified.org/ for the full background on the PSA certification requirements and components. An overview and the specific implementation details for RSL15 are provided here.

3.1 Overview of PSA Compliance

PSA is a concept originated by Arm and managed by third party labs and certification authorities, with the goal of standardizing the security methods across the varying types of connected devices in the semiconductor industry. It provides established best practices, as well as documentation and methods to determine whether a given device meets the outlined standards.

PSA protects sensitive assets (keys, credentials and firmware) by separating them from the application firmware and hardware. It defines a Secure Processing Environment (SPE) for this data, the code that manages it, and its trusted hardware resources.

The "Updatable and Immutable Areas" figure (Figure 3) shows the updatable and the immutable (non-changable) parts of an RSL15-based system that is intended for PSA compliance and follows the PSA Device Model guidelines. This clearly shows the secure bootloader in relation to the other parts of the system. The secure bootloader forms part of the chip scope, but is also part of the updateable components.

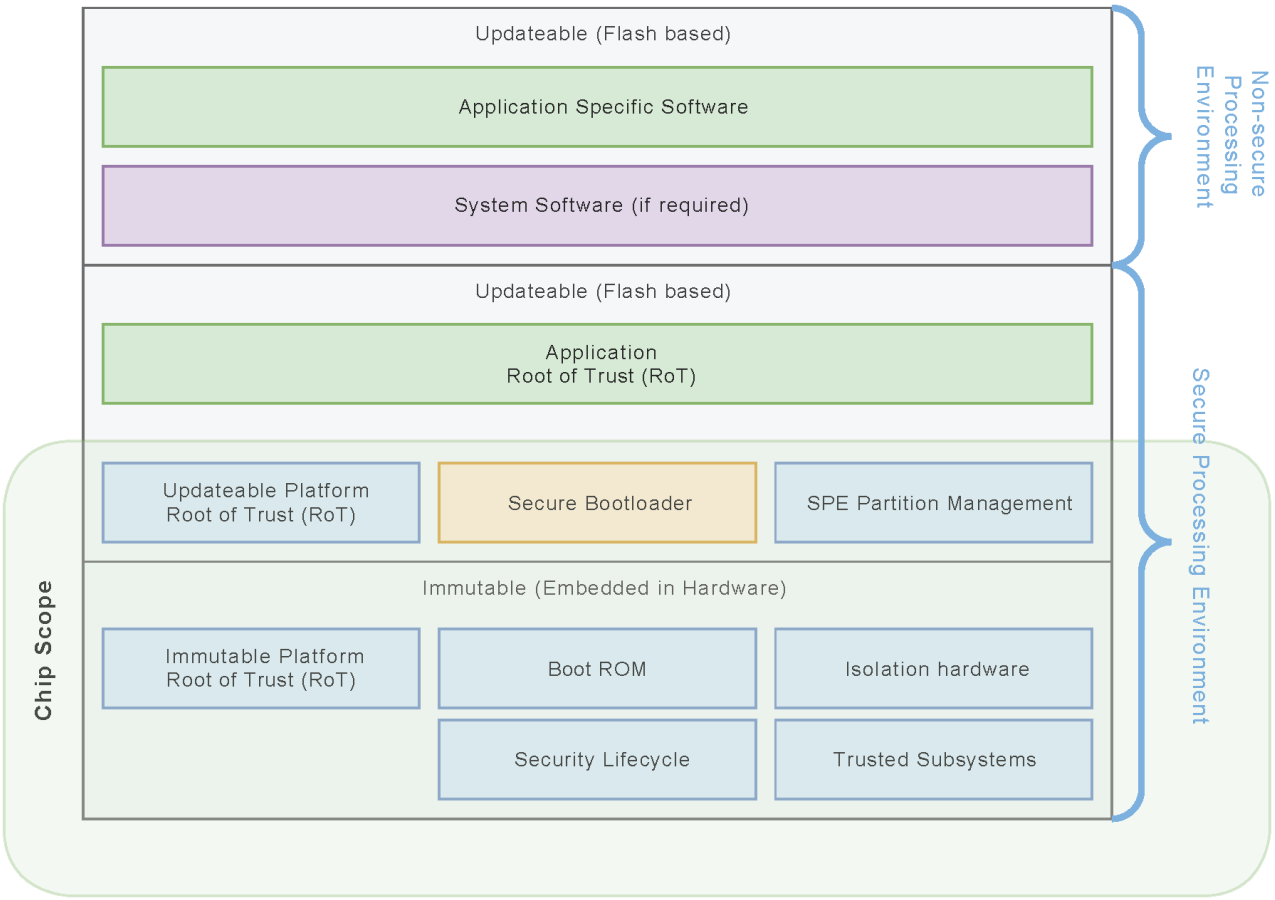

Figure 3. Updatable and Immutable Areas

Similar to the PSA documentation and references available online, we use the following terms in this documentation:

Entity

The device about which the attestation provides information

Manufacturer

The company that made the entity. This can be a chip vendor, a circuit board module vendor, or a vendor of finished consumer products.

Relying Party

The server, service or company that makes use of the information in the Entity Attestation Token (EAT) about the entity. (See Section 7.3 “Attestation Token” on page 32 for more information about the EAT.)

CHAPTER 4

Basic Bootloader

The secure bootloader can be used as a non-secure bootloader if the security features are not needed. In this case, the bootloader provides the following features:

- Booting of an application from flash
- Updating to store a new application in flash
- Updating the bootloader

4.1 General Usage

The bootloader must be loaded onto the device. An application to be used with the bootloader must have its start address in the location expected by the bootloader. See Section 2 “Overview” on page 12 for more information.

The "Standard Load and Update Sequence" figure (Figure 4) illustrates the process for updating the application stored in flash.

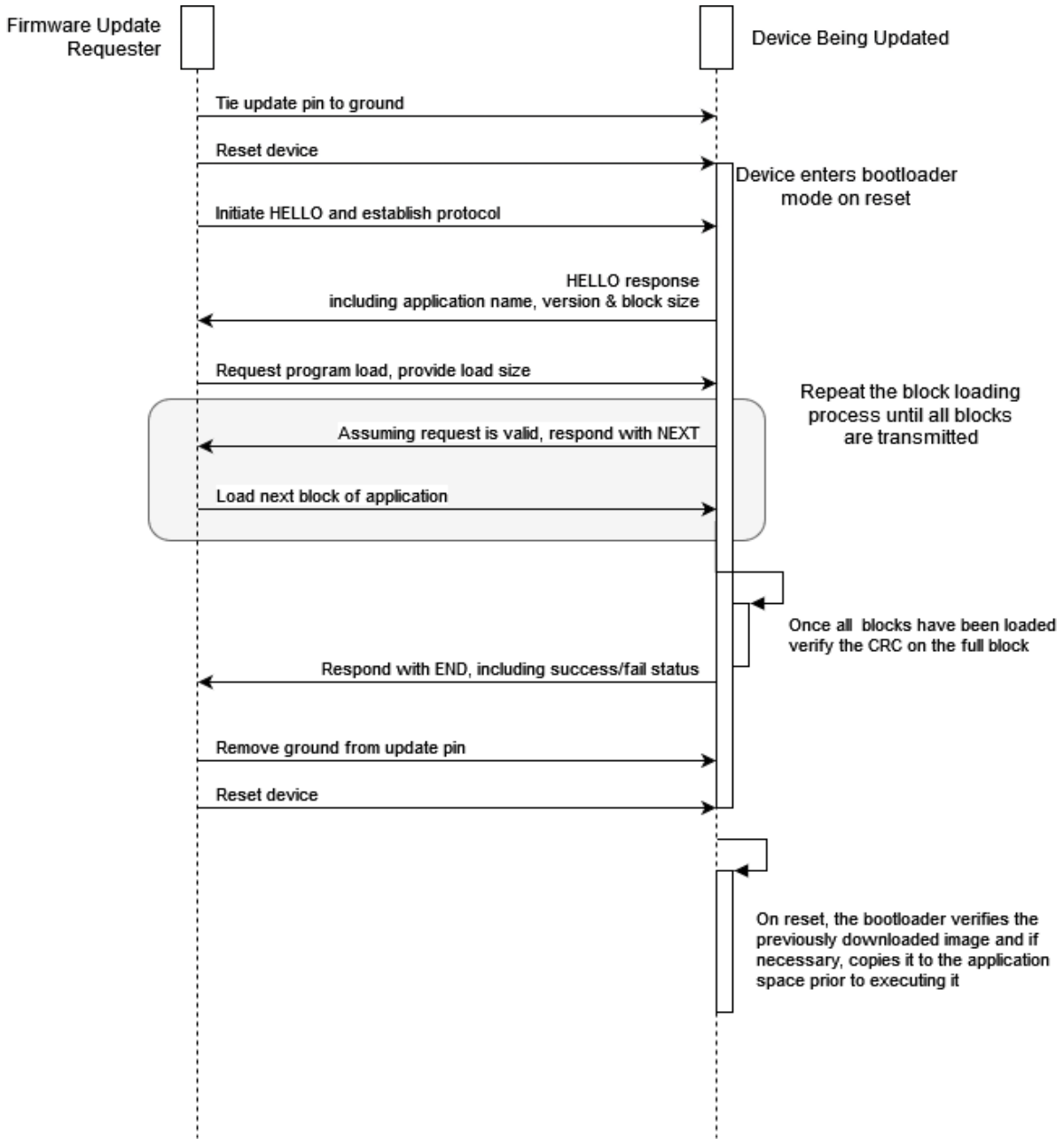

Figure 4. Standard Load and Update Sequence

Each request has a frame consistency sequence on it (a simple CCITT CRC of the data in the request). A block load request includes the length of the application being loaded, and the CRC32 for the whole application. Once a number of blocks have been loaded, the CRC32 is checked to ensure that no frames have been lost or corrupted during transmission. At any stage, if an error is detected (e.g., a timeout or a bad frame), the load process terminates with an `END` message and an indicator of the reason for the failure.

The application used in the firmware update process must include *bootloader.h* and use `SYS_BOOT_VERSION` to set the version number of the new application. The bootloader references this information. See the *Device Firmware Update (DFU) Guide* for further information about this and other basic bootloader features. The standard bootloader sample application is described there, but much of the information is also applicable to the basic bootloader (non-secure) functionality of the secure bootloader operation.

CHAPTER 5

Secure Bootloader

The secure mode of the bootloader provides the following features:

- Booting of a secure application from flash
- Updating to store a new secure application in flash
- Updating the secure bootloader
- Support for PSA (Platform Security Architecture) level 1 compliance

5.1 Booting a Secure Application

The secure bootloader provides firmware integrity and authenticity validation during a secure or trusted boot process. The main interface consists of an initialization function and a function to authenticate a Root of Trust certificate chain based on a given Root of Trust.

```
blSecureBootStatus_t BL_SecureBootAuthenticate(
    uint32_t opkey1, uint32_t opkey2, uint32_t opcontent,
    bool verifyImages, uint32_t relocation);
```

5.2 Updating a Secure Application

When a secure application must be updated, the process is as follows:

- Request an update. The bootloader polls a flag to determine if an update is requested, so this must be set.

```
BL_UpdateType_t BL_UpdateIsAvailable(uint32_t address, uint32_t extent);
```

- Provide a new secure application. The bootloader checks if there is a new image in the download area.

```
BL_UpdateType_t BL_UpdateIsAvailable(uint32_t address, uint32_t extent);
```

- Update the image. The bootloader uses the function with this prototype to perform the update.

```
void BL_UpdateImage(BL_UpdateType_t request,
uint32_t srcAddress, uint32_t dstAddress, uint32_t dstLength);
```

The application must also be authenticated. There are options to share a Root of Trust between the secure application and secure bootloader, such that a tradeoff can be made between boot time and the level of security needed. Alternatively, they can use separate Roots of Trust for increased security, but with increased boot time. The sequence is shown in the "Secure Authenticate/Load/Update Sequence" figure (Figure 5).

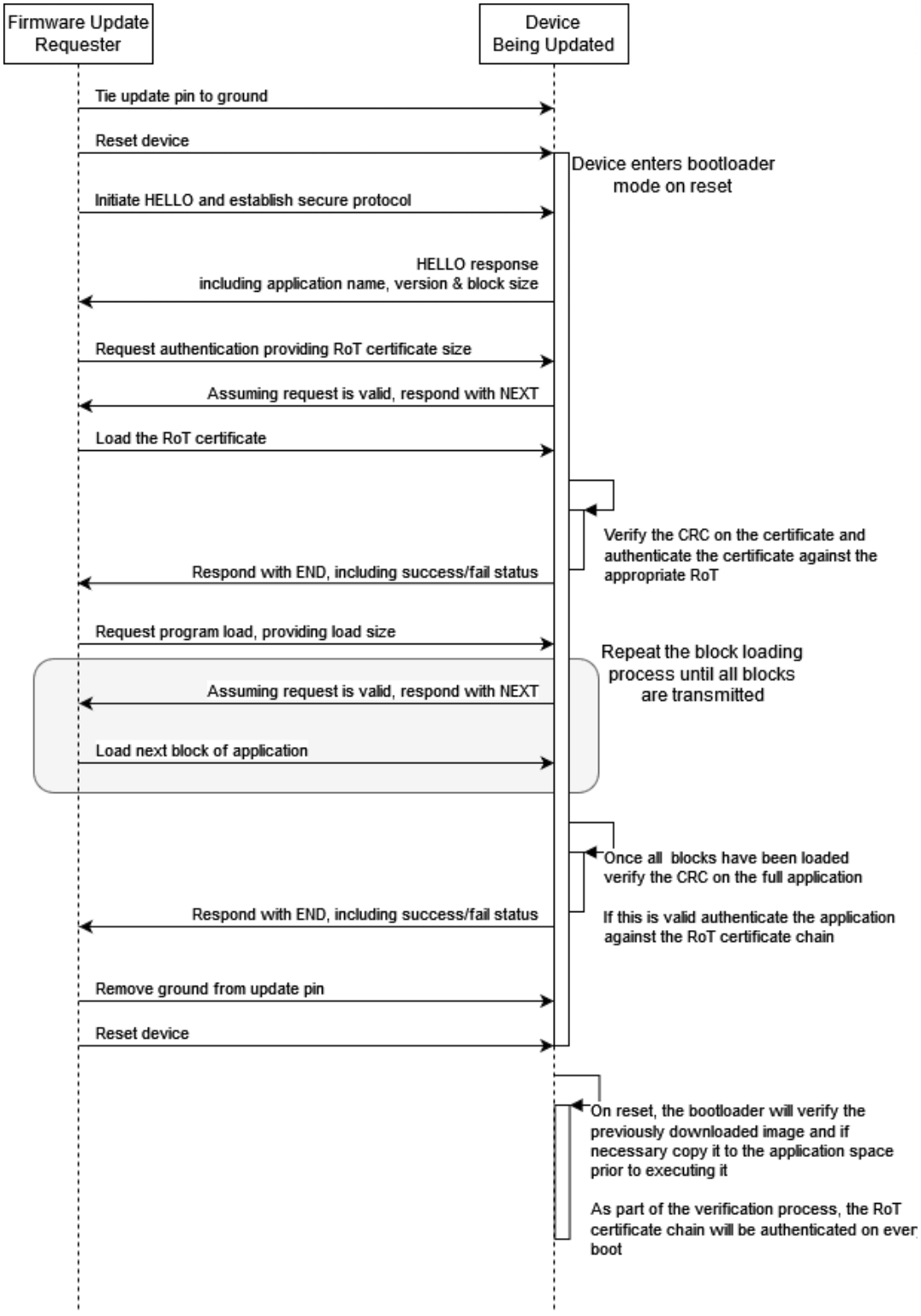

Figure 5. Secure Authenticate/Load/Update Sequence

The update process for the simple bootloader is very similar to that for the secure bootloader, the main differences being that the secure bootloader uses a different CCITT CRC algorithm on each frame and the connection must be pre-authenticated using a valid key certificate.

5.3 Updating the Secure Bootloader Itself

Updating the secure bootloader itself is much the same as updating a secure application. It uses the same function but with a different image. Flags are used for differentiation between a basic bootloader and a secure bootloader to determine the image size needed, since the secure bootloader uses more memory.

To prepare a secure bootloader, take the following steps:

- Ensure that you have, or make sure to create, all the necessary keys and key certificates. See the *RSL15 Security User's Guide* for details on creating secure applications.
- Create a content certificate based on the key certificates.
- Sign the image using the created certificates.

The new bootloader must be fully validated prior to switching to it, so that a complete copy is held in memory.

IMPORTANT: When updating a bootloader via the secure bootloader download mechanisms, care must be taken to ensure that the image being loaded is correct and that the update process is allowed to run to completion.

The update process is in two parts:

1. **Initially, the new application or bootloader is stored in the download area of flash memory.**
2. **When the system is reset, the secure bootloader verifies and, if necessary, authenticates the image in the download area. If these checks pass, the image is copied to the secure bootloader area of flash memory.**

This copying from the download area to the bootloader area by definition corrupts the bootloader at some point before the full image has been copied. If the power is lost during this stage, the system cannot recover, requiring a new bootloader to be loaded using the debug port.

Similarly, if the application or bootloader that has been loaded is faulty or is not a valid bootloader, this renders the secure bootloader unable to function.

In the current design there is no way around this issue; however, there are several mitigation strategies that can be employed to provide a more robust solution. These can include some combination of the following:

- **Store a redundant copy of the bootloader, which can be reverted to in case of major system failure.**
  - **This redundant copy can have limited verification and authentication capabilities if that meets the customer needs.**
- **Partition the bootloader such that it has a mutable and immutable component. The immutable component could be the part that handles the copying from the download area to the secure bootloader area.**
- **Disable the ability to update the bootloader itself except under very specific and controlled circumstances where the user cannot unintentionally render the device completely inoperable.**

5.4 Support for Immutable Portions in the Secure Bootloader

The concept of updateable and immutable portions of a secure processing environment is introduced in Section 3.1 “Overview of PSA Compliance” on page 18. RSL15 has the following support for the immutable portions of the secure processing environment:

- A boot ROM that can handle a Secure Boot and Secure Debug process
- Hardware isolation of cryptographic functions and the storage of security-related assets
- Unique key storage and the concept of a hardware unique key
- A managed security life cycle as described in the *RSL15 Security User's Guide*
- Trusted subsystems providing a separation between the secure and non-secure environments, using TrustZone

CHAPTER 6

Secure Storage

Protected storage is required to hold the keys and any other context that must be maintained. Any secure code has free access to the contents of the secure storage area.

6.1 Secure Storage Area

The bootloader's *sections.ld* file shows the size and start address of the secure storage area:

```
/* Reserve the remaining 11K from the first data sector for secure storage */
 BL_SECURE_STORE (xrw)    : ORIGIN = 0x0015A400, LENGTH = 11K
```

6.2 Content to be Stored in Secure Storage

- RSA public/private key pair
- ECC public/private key pair
- AES key

6.3 API

The API for secure storage is defined in *bl_simple_filer.h* and *bl_file_encryption.h*. The former provides the basic file system handler, and the latter provides the encryption layer. See the API reference or the files in the sample implementation for details.

Some important functions are as follows.

From *bl_simple_filer.h*:

```
BL_FStoreStatus_t BL_FStoreWrite(BL_FSFileId_t id,
uint8_t *buffer, uint16_t size, uint16_t flags);

BL_FStoreStatus_t BL_FStoreWrite(BL_FSFileId_t id,
uint8_t *buffer, uint16_t size, uint16_t flags);

BL_FStoreStatus_t BL_FStoreDelete(BL_FSFileId_t id);

BL_FStoreStatus_t BL_FStoreFileList(
uint8_t *buffer, uint16_t *maxsize, bool showHidden);
```

From *bl_file_encryption.h*:

```
BL_EncryptionStatus_t BL_EncryptBuffer(uint8_t *buffer, size_t length);

BL_EncryptionStatus_t BL_DecryptBuffer(uint8_t *buffer, size_t length);
```

6.4 Basic Operation

The secure bootloader offers a simple file system, primarily for storing attestation keys, but it can also be used for the general storage of small files.

The file system location is defined in *bl_memory.h* and occupies a range of sectors in data flash. Due to the limitations of the flash, the file system space is set to 11 KB.

The file system is organized in blocks that align with the underlying data sectors. Each data sector is 256 bytes in length; therefore, 44 blocks are available for use.

NOTE: A single file can be stored in more than one block. A single block can only contain information for a single file.

The first sector contains the inode table, which describes the blocks that are allocated to each file. There is a single inode entry for each file held in the file system. Each inode is defined as 12 bytes; therefore, a maximum of 21 files are supported by the file system.

Each inode/file contains the following information:

- The list of blocks allocated to the file. This is a 48-bit mask where a 1 indicates that the data block is used by that file.
- The file ID, which is defined as a 16-bit value because space is limited. How this is derived from a textual filename is left to the caller.
- A flags word, which contains a 16-bit value that indicates if the file is readable, writable, or can be deleted
- Size of the file in bytes. This is a 16-bit value because the maximum size of the store is defined as 11 KB. This is large enough to handle any file that can be stored.

IMPORTANT: When using the bootloader in debug mode, the Hardware Unique Key (HUK) is used, and appears differently when debugging compared with its appearance in typical usage. This means that when the key checking is performed, the HUK appears to be invalid, causing all inodes and any prior data stored in the secure storage area to be wiped.

CHAPTER 7

Attestation

Attestation, in this context, refers to providing information about the device to other parties using a very simple, cryptographically secured token; this is part of PSA compliance. Attestation provides a device with the ability to sign an array of bytes with a device private key and return the result to the caller. There are several use cases, ranging from attestation of the device state to generating a key pair and proving that it has been generated inside a secure key store.

7.1 Overview and Background

To maintain the Root of Trust, the secure bootloader only lets you program the device if you prove that you are allowed to do so. A challenge and response process matches certificates; if you do not have a matching certificate, it is not possible for you to program the device.

Part of attestation is the Entity Attestation Token (EAT), which contains claims that are generated in the device RoT. EAT token generation is expected to be performed many times, possibly for each transaction. It is relatively inexpensive because the claims data is small and ECDSA signing is relatively fast. For further details about the EAT, see Section 7.3 “Attestation Token” on page 32.

The token is sent to the device, and then goes to the relying party, which then relays it (without examining or modifying it) to the attestation service for verification.

7.2 Attestation Interface

The PSA Attestation API is a standard interface provided by the PSA Root of Trust.

For attestation within the context of PSA, the key can be generated or injected into the system. For the RSL15 secure bootloader sample application, there is a function to inject a key. If the application is not given an injected key, it creates and internal key.

The main relevant functions for the RSL15 attestation interface are as follows:

- Inject attestation key
- Get attestation token
- Get attestation token size

A summary and the prototypes of the attestation interface functions are provided below. See the relevant parts of the full API and the *bl_attestation.h* header file for further details on function parameters.

7.2.1 Key Injection

- The key injection interface allows for the injection of keys generated externally to the device.
- It also allows for keys to be generated on the device and then stored for later use.
- AES keys are derived from the HUK (Hardware Unique Key) using some form of initial value.
- If a key is provided, the private component key is stored and the public component is returned. If no key is provided, a new key is generated, the private component is stored and the public component is returned. If a symmetric (AES) key is requested, the key is stored and returned.

```
BL_AttestStatus_t BL_AttestInjectKey(
    const uint8_t *key, size_t keySize, BL_AttestKeyType_t type,
    uint8_t *publicKey, size_t publicKeyMaxSize, size_t *publicKeySize);
```

7.2.2 Get Token

To use the token, the attestation client issues some form of challenge and the device needs to respond. Part of the challenge is a random value that needs to be in the token to confirm its validity. This function is used to request the token from the device.

```
BL_AttestStatus_t BL_AttestGetToken(
    const uint8_t *challenge, BL_AttestationChallengeSize_t challengeSize,
 uint8_t *token, uint32_t *tokenSize);
```

7.2.3 Get Token Size

```
BL_AttestStatus_t BL_AttestGetTokenSize(
BL_AttestationChallengeSize_t challengeSize, uint32_t *tokenSize);
```

7.2.4 Key Injection Process

The "Attestation Key Injection Diagram" figure (Figure 6) illustrates the process of key injection.

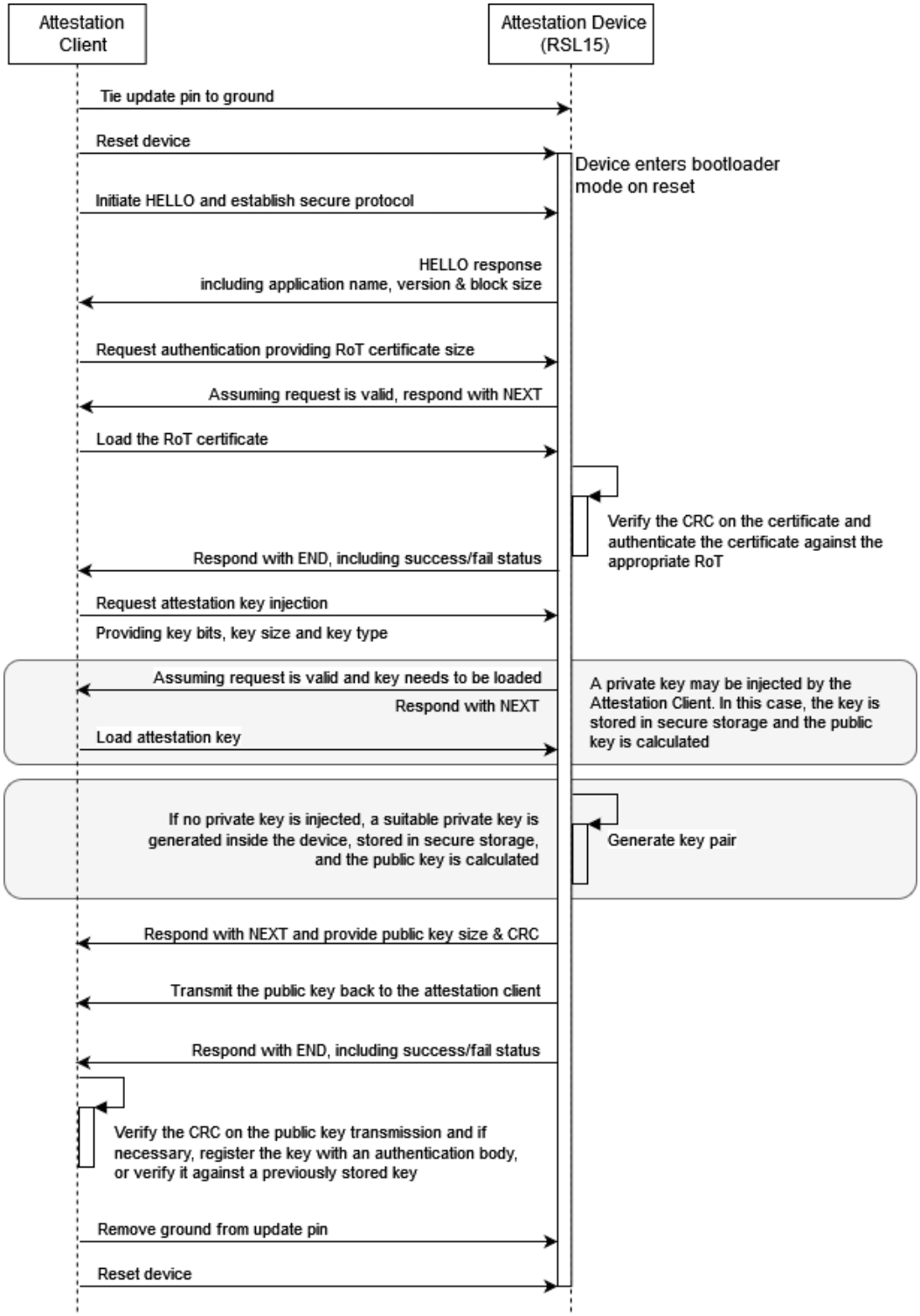

Figure 6. Attestation Key Injection Diagram

7.3 Attestation Token

7.3.1 Format of Token

The attestation token takes the format of what is becoming an industry standard: an entity attestation token (EAT).

For the purposes of this sample code, a variant of the Arm Platform Security Architecture attestation token has been chosen to provide the basic structure. More information is available at: https://www.psacertified.org/blog/what-is-an-entity-attestation-token/.

An EAT is a small blob of data that includes information items and is cryptographically signed. The signing secures the token itself, so that the mechanism that is transmitting the token does not have to provide any security. This allows IoT devices to securely introduce themselves to networks and to IoT platforms. The EAT is wrapped into a compact binary object representation (CBOR)-type message.

Each information item in this token in known as a claim, as shown in the "Items Included in Token" table (Table 2). A claim is a data item, which is represented as a key-value pair.

Table 2. Items Included in Token

Item	Size (bytes)	Description
Authentication Challenge	32/48/64	Input object of random bytes provided by the caller, intended to provide freshness to reports
Instance ID	32	Hash of public key that represents the unique identifier of the instance and is encoded as a byte string
Implementation ID	32	Represents the original implementation signer of the attestation key and identifies the contract between the report and verification. A verification service uses this claim to locate the details of the verification process. Consists of a value encoded as byte string.
Security Lifecycle	4	Positive ID corresponding to secure (0x3000) which represents the current lifecycle state of the instance
Client ID	4	Partition ID which, for the secure bootloader, is a positive value indicating it is a secure caller of the initial attestation API. The value is encoded as a signed integer.
Hardware Version	16	EAN-13, which can be used to reference the security level of the PSA-ROT via a certification website, and is encoded as a text string
Boot Seed	32	Byte string for the random value created at system boot time that allows differentiation of reports from different system sessions
Software Components:		
ROM	32	SHA256 of ROM image
Bootloader	32	SHA256 of Bootloader
Application	32	SHA256 of Application

The size of the attestation token is governed by the fields outlined in the "Items Included in Token" table (Table 2), and by how these values are encoded into a CBOR data stream. In addition, the EAT is wrapped into a signed format which includes a hash of the CBOR data, as well as a signature based on the size and type of the attestation key.

The "Example Token Sizes" table (Table 3) provides some example token sizes for different challenge and key sizes. All values are in bytes.

Table 3. Example Token Sizes

Key Size	1024					2048					3072				
Challenge Size	EAT	Hash	Sign	CBOR	Total	EAT	Hash	Sign	CBOR	Total	EAT	Hash	Sign	CBOR	Total
32	248	32	128	59	467	248	32	256	60	596	248	32	384	60	724
48	264	32	128	59	483	264	32	256	60	612	264	32	384	60	740
64	280	32	128	59	499	280	32	256	60	628	280	32	384	60	756

An example calculation can be found in the sample application source code in *bl_eat.c*.

NOTE: Some items in the EAT are provided as placeholder default values in the secure bootloader sample application. They are intended to be replaced as needed in a final product implementation.

7.3.2 EAT Additional Details

EAT has the capabilities to provide the source of trust, using a cryptographically signed piece of data containing claims that are generated in the device Root of Trust (RoT). The main use is for the relying party to verify the claims made by the device, such as the following:

- The unique identity of the device
- Installed firmware on the device and its integrity status
- Security assurance and certification status (such as a PSA Certified level)
- Manufacturer of the device hardware

Using this information, the relying party can make informed decisions, such as whether the device is legitimate and should be trusted, or what services to enable based on the information it receives. This is shown in the "Relying Party Decision Diagram" figure (Figure 7).

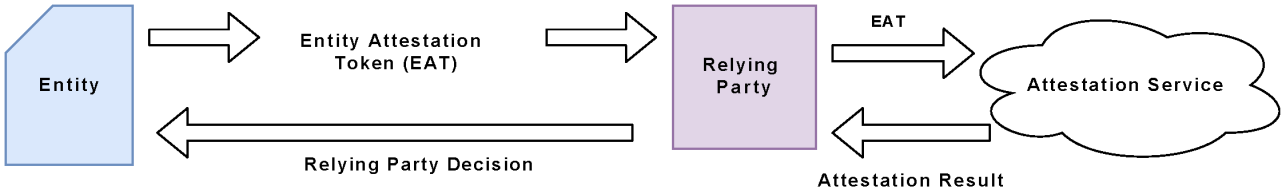

Figure 7. Relying Party Decision Diagram

7.3.3 Attestation Token Request

The "Attestation Token Request" figure (Figure 8) shows the process of requesting an attestation token.

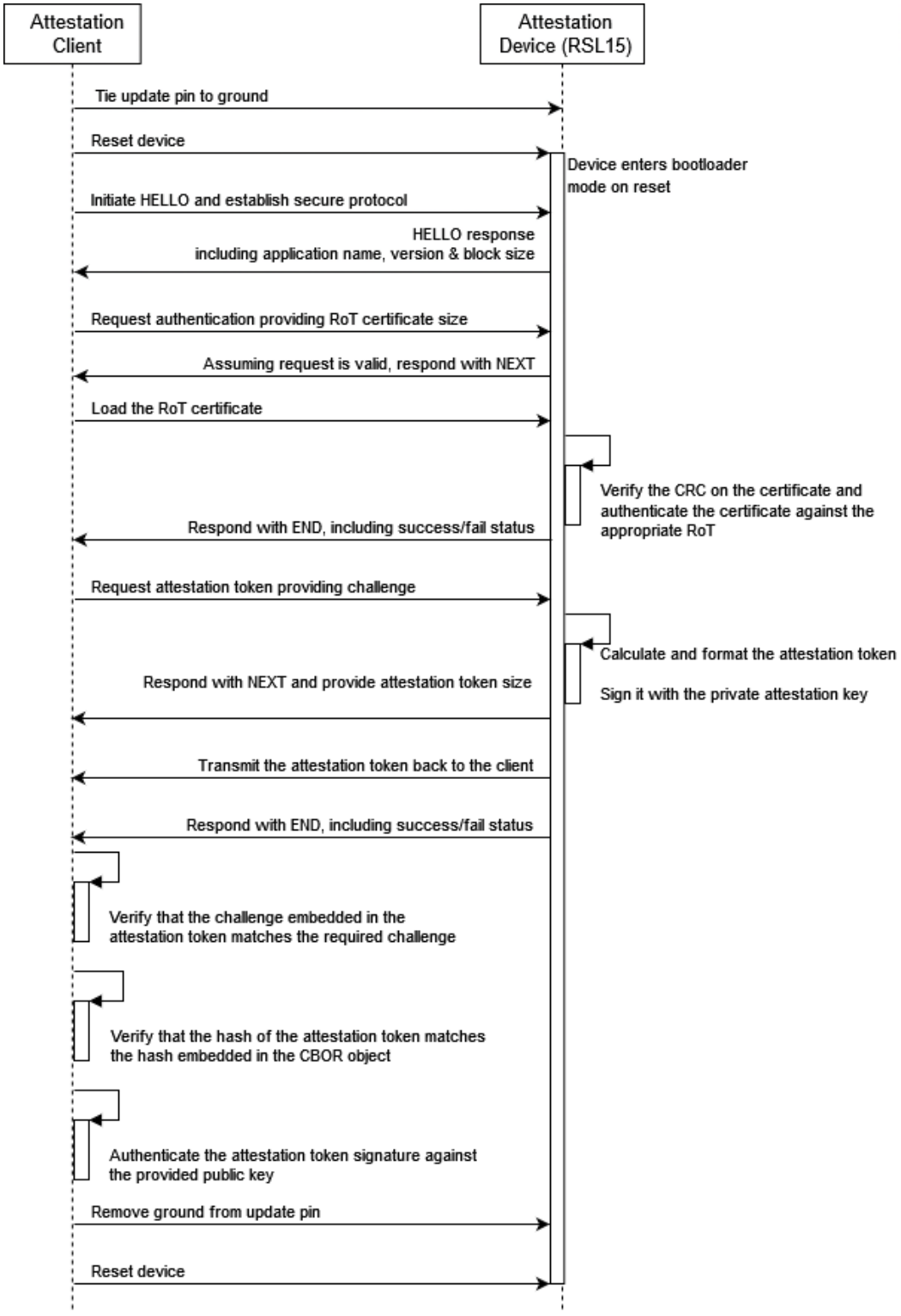

Figure 8. Attestation Token Request

CHAPTER 8

Secure Bootloader Sample Reference

Secure Bootloader Sample Reference.

8.1 SUMMARY

Typedefs

- BL_FCS_t : Define a FCS type.
- BL_BootAppId_t : Define the application ID as a six character string.

Variables

- BL_ImageWorkspace : Defines a common operation buffer for handling images.

Data Structures

- BL_AppConfiguration_t : Define a structure which can map onto the configuration area.
- BL_ImageOperation_t : Buffer used for loading data in chunks, allow 2 blocks.
- BL_ImageSplitRange_t : define an address range which can wrap-around a reserved block
- BL_StatusResponse_t : to maintain backwards compatibility, we use a two byte status for most messages.
- BL_BootAppVersion_t : Define the application version as id and version details.
- BL_HelloResponse_t : Define the contents of a Hello response.

Enumerations

- BL_UpdateType_t : Define the possible update types.
- BL_ConfigStatus_t : Define the configuration status values.
- BL_FCSStatus_t : Define the possible FCS status values.
- BL_FCSAlgorithm_t : Define the possible valid FCS calculators.
- BL_ImageType_t : Define the known image types.
- BL_ImageStatus_t : Define the image status values.
- BL_LoaderCommand_t : Enum specifying each of the valid commands the loader recognizes.
- BL_LoaderStatus_t : Define a set of supported loader status codes.
- BL_LoaderCertType_t : Enum specifying the types of certificate that can be loaded.
- BL_LoaderStatusType_t : Define a type for the status messages.
- BL_UARTStatus_t : Define a set of supported error codes.

Macros

- VT_OFFSET_STACK_POINTER : Vector table offset for the stack pointer.
- VT_OFFSET_RESET_VECTOR : Vector table offset for the reset vector.

- VT_OFFSET_VERSION_INFO : Vector table offset for the version information pointer.
- VT_OFFSET_IMAGE_SIZE : Vector table offset for the used image size pointer.
- VT_OFFSET_CERT_SIZE : Vector table offset for the certificate size.
- BL_CONFIGURATION_BASE : Base address of the boot configuration in flash.
- BL_CONFIGURATION_WORDS : Define the size of the configuration area in words.
- FLASH_BOND_INFO_SIZE
- BL_CODE_SECTOR_SIZE : The image block size when loading data.
- BL_DATA_SECTOR_SIZE : The image block size when loading data.
- BL_FLASH_RESERVED_SIZE : The size of the area reserved for use by the ROM and stack.
- BL_SECURE_STORAGE_BASE : Define the base address of the secure storage area.
- BL_SECURE_STORAGE_SIZE : Define a size for the secure storage area.
- BL_SECURE_STORAGE_TOP : Define the top of the secure storage area.
- BL_BOOTLOADER_BASE : The base address of the bootloader flash.
- BL_BOOTLOADER_KB : Define the size of the bootloader in kB.
- BL_BOOTLOADER_SIZE : The size of the area reserved for use by the bootloader.
- BL_FLASH_CODE_BASE : The base of the code flash.
- BL_FLASH_DATA_BASE : The base of the data flash, offset by the reserved areas.
- BL_FLASH_CODE_TOP : Define the top of code flash in 512K device.
- BL_FLASH_DATA_TOP : Define the top of data flash in 512K device.
- BL_FLASH_CODE_SIZE : Code size is derived from the base and top addresses.
- BL_FLASH_DATA_SIZE : Data size is derived from the base and top addresses.
- BL_APPLICATION_BASE : Define the base address of the application.
- BL_AVAILABLE_SIZE : Define the total available flash for application and download.
- BL_APPLICATION_SIZE : Define the maximum size of an application.
- BL_DOWNLOAD_BASE : Define the base address of the download area.
- BL_DOWNLOAD_SIZE : Define the maximum size of the download area.
- BL_OPT_FEATURE_ENABLED : Indicator that a given feature should be enabled.
- BL_OPT_FEATURE_DISABLED : Indicator that a given features should be disabled.
- BL_OPT_FEATURE_BOOTLOADER : Marker indicating that the bootloader feature is enabled.
- BL_OPT_FEATURE_SECURE_BOOTLOADER : Marker indicating that the bootloader supports authenticated update of images.
- BL_OPT_FEATURE_SECURE_STORAGE : Marker indicating if the secure storage feature is provided.
- BL_OPT_FEATURE_ATTESTATION : Marker indicating if the bootloader supports attestation protocols.
- BL_OPT_ATTEST_KEY_AES : Marker indicating that the attestation feature supports AES keys.
- BL_OPT_ATTEST_KEY_RSA : Marker indicating that the attestation feature supports RSA keys.
- BL_OPT_ATTEST_KEY_ECC : Marker indicating that the attestation feature supports ECC keys.
- BL_OPT_SECURE_FILE_SYSTEM_RESET : Marker indicating that the attestation feature supports AES keys.
- DEBUG_CATCH_GPIO
- UART_CLK : Set UART peripheral clock.
- SENSOR_CLK : Set sensor clock.
- USER_CLK : Set user clock.
- VCC_BUCK_ENABLE : Enable or disable the buck converter.
- BL_TICKER_TIME_MS : Define the time in ms for each interrupt.
- BL_DEBUG : Define the standard verbose/debug tracing routine.
- BL_TRACE : Define the standard tracing routine.
- BL_WARNING : Define the standard warning message routine.
- BL_ERROR : Define the standard error message routine.
- BL_UART_RX_TIMEOUT_MS : Define the receive timeout in milliseconds.
- BL_WATCHDOG_FEED_ME_MS : While waiting for UART input, ensure watch dog is fed.

- BL_UART_TX_TIMEOUT_MS : Define the send timeout in milliseconds.
- BL_UART_MAX_RX_LENGTH : Define the maximum length of a single receive operation.
- BL_UART_MAX_TX_LENGTH : Define the maximum length of a single send operation.
- BL_BAUD_RATE : Define a baud rate for loading.
- BL_UART_DELAY_CYCLES : Define a delay time to allow the hardware buffers to clear.
- UPDATE_GPIO : Define the GPIO pin to be used to indicate an update is required.
- MIN : Define a shorthand to get the minimum of two values.
- MAX : Define a shorthand to get the maximum of two values.
- BITS2BYTES : Calculate the number of bytes needed to hold x bits.
- BITS2HALFWORDS : Calculate the number of 16 bit words needed to hold x bits.
- BL_VERSION_ENCODE : Define a mechanism to encode a version number as a uint16_t.
- BL_VERSION_DECODE : Define a mechanism to decode a version number from a uint16_t.
- BL_BOOT_VERSION : Define the boot version including name and ensure it is stored in an easily accessible location.
- BL_WATCHDOG_MAX_HOLD_OFF_SECONDS : Define the maximum time that can elapse before the watchdog must be refreshed.

Functions

- BL_CheckRemapAddressSpace : Determine download address based on given address which may be in bootloader or application space.
- BL_CheckGetApplicationSize : Fetch the application size from a buffer defined by base address of the application vector table.
- BL_CheckRelocatedApplicationSize : Fetch the application size from a buffer defined by base address of the application vector table.
- BL_CheckIfImageUpdateAvailable : Check for a valid update using the non-secure file format.
- BL_CheckIfSecureImageUpdateAvailable : Check for a valid update using the secure file format.
- BL_CheckFindSecondaryImageLocation : Based on a primary image address, calculate the potential location and extent of any secondary image.
- BL_ConfigIsValid : Helper function to return the configuration area status.
- BL_ConfigCertificateAddress : Fetch the address of the requested structure.
- BL_FCSInitialize : Initialize the FCS module, deriving the selected algorithm from the provided sample data.
- BL_FCSQuery : Query the currently selected FCS algorithm.
- BL_FCSAuthenticationRequired : Provides a mechanism to determine if the loading process should apply authentication to the protocol and images.
- BL_FCSSelect : Select a specific FCS algorithm.
- BL_FCSCheck : Check the validity of a buffer against a given FCS.
- BL_FCSCalculate : Calculate the FCS of a given buffer.
- BL_FCSAccumulateCRC : Helper method to accumulate a CRC given a buffer and a length.
- BL_FlashInitialize : Initialize the flash subsystem.
- BL_FlashSaveSector : Save a buffer to a specified flash address.
- BL_ImageInitialize : Initialize the image module for a specific set of image attributes.
- BL_ImageAddress : Convert an address to take into account potential offsets.
- BL_ImageAddressRange : Helper routine which allows access of the image as a contiguous block of addresses, wrapping around the reserved block.
- BL_ImageCopyMemoryRange : Copy a possibly split memory range to a contiguous buffer.
- BL_ImageSaveBlock : Save a block of data from a RAM buffer to the next block in Flash.
- BL_ImageVerify : Verify the most recently loaded image.

- BL_ImageAuthenticate : Authenticate a loaded image.
- BL_ImageAuthenticateCurrent : Authenticate the most recently loaded image.
- BL_ImageIsValid : Check if there is a valid image to start.
- BL_ImageSaveAddress : Return the download address corresponding to the requested address.
- BL_ImageStartApplication : Start the image stored in flash.
- BL_LoaderPerformFirmwareLoad : Perform a firmware update over the UART interface.
- BL_LoaderCertificateAddress : Fetch the address of the requested structure.
- BL_RecoveryInitialize : Define the initialization routine for the Debug Catch feature.
- BL_TargetInitialize : Target initialization function, loads the trim values and sets up the various clocks used in the system.
- BL_TargetReset : Reset the device using NVIC.
- BL_TickerInitialize : Initialize the timer tick.
- BL_TickerTime : Get the current timer tick value.
- SysTick_Handler : System tick interrupt handler, required by the ticker.
- BL_TraceInitialize : Initialize the trace sub-system.
- BL_UARTInitialize : Initialize the UART subsystem.
- BL_UARTReceiveAsync : Start receiving a fixed length data buffer using the UART.
- BL_UARTReceiveComplete : Complete the reception of an executing receive operation.
- BL_UARTReceive : Receiving a fixed length data buffer using the UART.
- BL_UARTSendAsync : Start sending a fixed length data buffer using the UART.
- BL_UARTSendComplete : Complete the transmission of an executing send operation.
- BL_UARTSend : Send a fixed length data buffer using the UART.
- BL_UpdateInitialize : Initialize the firmware update component.
- BL_UpdateRequested : Check if a firmware update is being requested.
- BL_UpdateProcessPendingImages : This will check for any pending images which have previously been downloaded and if any are found will copy them to the appropriate location for execution.
- BL_ImageSelectAndStartApplication : This will attempt to start any images which are available.
- BL_VersionsGetInformation : Get the version information from a suitable application.
- BL_VersionsGetHello : Fetch the hello response from the bootloader.
- BL_WatchdogInitialize : Initialise the watchdog module.
- BL_WatchdogSetHoldTime : Set the watchdog hold off time to seconds.
- WATCHDOG_IRQHandler : Define an interrupt handler for the watchdog interrupt.

8.2 Detailed Description

This reference chapter presents a detailed description of all the components included in the secure bootloader reference application. This reference application has four levels of secure operation, available as needed depending on the end product's use cases:

1. Basic bootloader (non-secure)
2. Secure bootloader (maintains authenticated Root of Trust set up by the ROM)
3. Secure bootloader with secure storage
4. Secure bootloader with secure storage and device attestation

8.3 Secure Bootloader Sample Reference Typedef Documentation

8.3.1 BL_FCS_t

```
typedef uint16_t BL_FCS_t
```

Location: bl_fcs.h:52

Define a FCS type.

8.3.2 BL_BootAppId_t

```
typedef char BL_BootAppId_t
```

Location: bl_versions.h:75

Define the application ID as a six character string.

8.4 Secure Bootloader Sample Reference Variable Documentation

8.4.1 BL_ImageWorkspace

```
BL_ImageOperation_t BL_ImageWorkspace
```

Location: bl_image.h:85

Defines a common operation buffer for handling images.

8.5 Secure Bootloader Sample Reference Enumeration Type Documentation

8.5.1 BL_UpdateType_t

Location: bl_check.h:66

Define the possible update types.

Members

- BL_UPDATE_IMAGE
- BL_UPDATE_BOOTLOADER
- BL_UPDATE_SECURE_IMAGE
- BL_UPDATE_SECURE_BOOTLOADER
- BL_UPDATE_SECONDARY_IMAGE
- BL_UPDATE_NONE

8.5.2 BL_ConfigStatus_t

Location: bl_configuration.h:57

Define the configuration status values.

Members

- BL_CONFIG_OKAY
- BL_CONFIG_CORRUPT

8.5.3 BL_FCSStatus_t

Location: bl_fcs.h:55

Define the possible FCS status values.

Members

- BL_FCS_NO_ERROR
- BL_FCS_VALID

- BL_FCS_INVALID
- BL_FCS_UNRECOGNIZED
- BL_FCS_NOT_INITIALIZED

8.5.4 BL_FCSAlgorithm_t

Location: bl_fcs.h:65

Define the possible valid FCS calculators.

Members

- BL_FCS_CCITT_FFFF = 0
- BL_FCS_MCRF4XX
- BL_FCS_NO_ALGO

8.5.5 BL_ImageType_t

Location: bl_image.h:51

Define the known image types.

Members

- BL_IMAGE_BOOTLOADER
- BL_IMAGE_APPLICATION
- BL_IMAGE_UNRECOGNIZED

8.5.6 BL_ImageStatus_t

Location: bl_image.h:58

Define the image status values.

Members

- BL_IMAGE_NO_ERROR = 0
- BL_IMAGE_ADDRESS_ERROR
- BL_IMAGE_LENGTH_ERROR
- BL_IMAGE_FLASH_ERROR
- BL_IMAGE_VERIFY_ERROR
- BL_IMAGE_AUTHENTICATE_ERROR

8.5.7 BL_LoaderCommand_t

Location: bl_loader.h:48

Enum specifying each of the valid commands the loader recognizes.

Members

- BL_LOADER_HELLO = 0
- BL_LOADER_PROGRAM
- BL_LOADER_READ
- BL_LOADER_RESTART
- BL_LOADER_ERROR
- BL_LOADER_COMMAND_MAX

8.5.8 BL_LoaderStatus_t

Location: bl_loader.h:81

Define a set of supported loader status codes.

Members

- BL_LOADER_NO_ERROR = 0
- BL_LOADER_BAD_MSG
- BL_LOADER_UNKNOWN_CMD
- BL_LOADER_INVALID_CMD
- BL_LOADER_GENERAL_FLASH_FAILURE
- BL_LOADER_WRITE_FLASH_NOT_ENABLED
- BL_LOADER_BAD_FLASH_ADDRESS
- BL_LOADER_ERASE_FLASH_FAILED
- BL_LOADER_BAD_FLASH_LENGTH
- BL_LOADER_INACCESSIBLE_FLASH
- BL_LOADER_FLASH_COPIER_BUSY
- BL_LOADER_PROG_FLASH_FAILED
- BL_LOADER_VERIFY_FLASH_FAILED
- BL_LOADER_VERIFY_IMAGE_FAILED
- BL_LOADER_NO_VALID_BOOTLOADER
- BL_LOADER_RX_FAILURE
- BL_LOADER_RX_TIMEOUT
- BL_LOADER_IMAGE_FAILURE
- BL_LOADER_VERIFICATION_FAILURE
- BL_LOADER_CERT_LOAD_FAILURE
- BL_LOADER_AUTHENTICATION_FAILURE
- BL_LOADER_AUTHENTICATE_IMAGE_FAILED
- BL_LOADER_FILE_SYSTEM_FAILURE

- BL_LOADER_ATTESTATION_FAILURE

8.5.9 BL_LoaderCertType_t

Location: bl_loader.h:110

Enum specifying the types of certificate that can be loaded.

Members

- BL_KEY1_CERT
- BL_KEY2_CERT
- BL_CONTENT_CERT
- BL_DEBUG_CERT

8.5.10 BL_LoaderStatusType_t

Location: bl_loader.h:119

Define a type for the status messages.

Members

- BL_LOADER_STATUS_TYPE_NEXT = 0x55
- BL_LOADER_STATUS_TYPE_END = 0xAA
- BL_LOADER_STATUS_TYPE_CRC = 0xCC

8.5.11 BL_UARTStatus_t

Location: bl_uart.h:79

Define a set of supported error codes.

Members

- BL_UART_NO_ERROR = 0
- BL_UART_TX_IDLE
- BL_UART_RX_IDLE
- BL_UART_TX_BUSY
- BL_UART_RX_BUSY
- BL_UART_TX_TIMEOUT
- BL_UART_RX_TIMEOUT
- BL_UART_INVALID_PARAMETER
- BL_UART_STATE_ERROR
- BL_UART_BAD_FCS
- BL_UART_RX_ERROR
- BL_UART_TX_ERROR

8.6 Secure Bootloader Sample Reference Macro Definition Documentation

8.6.1 VT_OFFSET_STACK_POINTER

```
#define VT_OFFSET_STACK_POINTER 0
```

Vector table offset for the stack pointer.

Location: bl_check.h:47

8.6.2 VT_OFFSET_RESET_VECTOR

```
#define VT_OFFSET_RESET_VECTOR 1
```

Vector table offset for the reset vector.

Location: bl_check.h:50

8.6.3 VT_OFFSET_VERSION_INFO

```
#define VT_OFFSET_VERSION_INFO 8
```

Vector table offset for the version information pointer.

Location: bl_check.h:53

8.6.4 VT_OFFSET_IMAGE_SIZE

```
#define VT_OFFSET_IMAGE_SIZE 9
```

Vector table offset for the used image size pointer.

Location: bl_check.h:56

8.6.5 VT_OFFSET_CERT_SIZE

```
#define VT_OFFSET_CERT_SIZE 10
```

Vector table offset for the certificate size.

Location: bl_check.h:59

8.6.6 BL_CONFIGURATION_BASE

```
#define BL_CONFIGURATION_BASE ((BL_AppConfiguration_t *) FLASH0_DATA_BASE)
```

Base address of the boot configuration in flash.

Location: bl_configuration.h:47

8.6.7 BL_CONFIGURATION_WORDS

```
#define BL_CONFIGURATION_WORDS (sizeof(BL_AppConfiguration_t) >> 2)
```

Define the size of the configuration area in words.

Location: bl_configuration.h:50

8.6.8 FLASH_BOND_INFO_SIZE

```
#define FLASH_BOND_INFO_SIZE 0x800
```

Location: bl_memory.h:53

8.6.9 BL_CODE_SECTOR_SIZE

```
#define BL_CODE_SECTOR_SIZE 2048
```

The image block size when loading data.

(Size of a code sector.)

Location: bl_memory.h:56

8.6.10 BL_DATA_SECTOR_SIZE

```
#define BL_DATA_SECTOR_SIZE 256
```

The image block size when loading data.

(Size of a code sector.)

Location: bl_memory.h:59

8.6.11 BL_FLASH_RESERVED_SIZE

```
#define BL_FLASH_RESERVED_SIZE (FLASH_DEU_RESERVED_SIZE + FLASH_BOND_INFO_SIZE)
```

The size of the area reserved for use by the ROM and stack.

Location: bl_memory.h:62

8.6.12 BL_SECURE_STORAGE_BASE

```
#define BL_SECURE_STORAGE_BASE (FLASH0_DATA_BASE + BL_FLASH_RESERVED_SIZE)
```

Define the base address of the secure storage area.

Location: bl_memory.h:66

8.6.13 BL_SECURE_STORAGE_SIZE

```
#define BL_SECURE_STORAGE_SIZE (FLASH0_DATA_RSL15_284_TOP - BL_SECURE_STORAGE_BASE + 1)
```

Define a size for the secure storage area.

Location: bl_memory.h:69

8.6.14 BL_SECURE_STORAGE_TOP

```
#define BL_SECURE_STORAGE_TOP (BL_SECURE_STORAGE_BASE + BL_SECURE_STORAGE_SIZE - 1)
```

Define the top of the secure storage area.

Location: bl_memory.h:73

8.6.15 BL_BOOTLOADER_BASE

```
#define BL_BOOTLOADER_BASE FLASH0_CODE_BASE
```

The base address of the bootloader flash.

Location: bl_memory.h:77

8.6.16 BL_BOOTLOADER_KB

```
#define BL_BOOTLOADER_KB (24 * 1024)
```

Define the size of the bootloader in kB.

Location: bl_memory.h:104

8.6.17 BL_BOOTLOADER_SIZE

```
#define BL_BOOTLOADER_SIZE BL_BOOTLOADER_KB
```

The size of the area reserved for use by the bootloader.

Location: bl_memory.h:110

8.6.18 BL_FLASH_CODE_BASE

```
#define BL_FLASH_CODE_BASE (BL_BOOTLOADER_BASE + BL_BOOTLOADER_SIZE)
```

The base of the code flash.

Location: bl_memory.h:113

8.6.19 BL_FLASH_DATA_BASE

```
#define BL_FLASH_DATA_BASE (BL_SECURE_STORAGE_TOP + 1)
```

The base of the data flash, offset by the reserved areas.

Location: bl_memory.h:116

8.6.20 BL_FLASH_CODE_TOP

```
#define BL_FLASH_CODE_TOP FLASH0_CODE_TOP
```

Define the top of code flash in 512K device.

Location: bl_memory.h:129

8.6.21 BL_FLASH_DATA_TOP

```
#define BL_FLASH_DATA_TOP FLASH0_DATA_TOP
```

Define the top of data flash in 512K device.

Location: bl_memory.h:132

8.6.22 BL_FLASH_CODE_SIZE

```
#define BL_FLASH_CODE_SIZE (BL_FLASH_CODE_TOP - BL_FLASH_CODE_BASE + 1)
```

Code size is derived from the base and top addresses.

Location: bl_memory.h:137

8.6.23 BL_FLASH_DATA_SIZE

```
#define BL_FLASH_DATA_SIZE (BL_FLASH_DATA_TOP - BL_FLASH_DATA_BASE + 1)
```

Data size is derived from the base and top addresses.

Location: bl_memory.h:140

8.6.24 BL_APPLICATION_BASE

```
#define BL_APPLICATION_BASE BL_FLASH_CODE_BASE
```

Define the base address of the application.

Location: bl_memory.h:143

8.6.25 BL_AVAILABLE_SIZE

```
#define BL_AVAILABLE_SIZE (BL_FLASH_CODE_SIZE + BL_FLASH_DATA_SIZE)
```

Define the total available flash for application and download.

Location: bl_memory.h:146

8.6.26 BL_APPLICATION_SIZE

```
#define BL_APPLICATION_SIZE ((BL_AVAILABLE_SIZE >> 1) & 0xFFFFF800)
```

Define the maximum size of an application.

(must be 2K aligned)

Location: bl_memory.h:149

8.6.27 BL_DOWNLOAD_BASE

```
#define BL_DOWNLOAD_BASE (BL_APPLICATION_BASE + BL_APPLICATION_SIZE)
```

Define the base address of the download area.

Location: bl_memory.h:152

8.6.28 BL_DOWNLOAD_SIZE

```
#define BL_DOWNLOAD_SIZE BL_APPLICATION_SIZE
```

Define the maximum size of the download area.

Location: bl_memory.h:155

8.6.29 BL_OPT_FEATURE_ENABLED

```
#define BL_OPT_FEATURE_ENABLED 1
```

Indicator that a given feature should be enabled.

Location: bl_options.h:47

8.6.30 BL_OPT_FEATURE_DISABLED

```
#define BL_OPT_FEATURE_DISABLED 0
```

Indicator that a given features should be disabled.

Location: bl_options.h:50

8.6.31 BL_OPT_FEATURE_BOOTLOADER

```
#define BL_OPT_FEATURE_BOOTLOADER BL_OPT_FEATURE_ENABLED
```

Marker indicating that the bootloader feature is enabled.

Location: bl_options.h:56

8.6.32 BL_OPT_FEATURE_SECURE_BOOTLOADER

```
#define BL_OPT_FEATURE_SECURE_BOOTLOADER BL_OPT_FEATURE_DISABLED
```

Marker indicating that the bootloader supports authenticated update of images.

Location: bl_options.h:62

8.6.33 BL_OPT_FEATURE_SECURE_STORAGE

```
#define BL_OPT_FEATURE_SECURE_STORAGE BL_OPT_FEATURE_DISABLED
```

Marker indicating if the secure storage feature is provided.

Location: bl_options.h:67

8.6.34 BL_OPT_FEATURE_ATTESTATION

```
#define BL_OPT_FEATURE_ATTESTATION BL_OPT_FEATURE_DISABLED
```

Marker indicating if the bootloader supports attestation protocols.

Location: bl_options.h:72

8.6.35 BL_OPT_ATTEST_KEY_AES

```
#define BL_OPT_ATTEST_KEY_AES BL_OPT_FEATURE_DISABLED
```

Marker indicating that the attestation feature supports AES keys.

Location: bl_options.h:110

8.6.36 BL_OPT_ATTEST_KEY_RSA

```
#define BL_OPT_ATTEST_KEY_RSA BL_OPT_FEATURE_DISABLED
```

Marker indicating that the attestation feature supports RSA keys.

Location: bl_options.h:116

8.6.37 BL_OPT_ATTEST_KEY_ECC

```
#define BL_OPT_ATTEST_KEY_ECC BL_OPT_FEATURE_DISABLED
```

Marker indicating that the attestation feature supports ECC keys.

Location: bl_options.h:122

8.6.38 BL_OPT_SECURE_FILE_SYSTEM_RESET

```
#define BL_OPT_SECURE_FILE_SYSTEM_RESET BL_OPT_FEATURE_DISABLED
```

Marker indicating that the attestation feature supports AES keys.

Location: bl_options.h:167

8.6.39 DEBUG_CATCH_GPIO

```
#define DEBUG_CATCH_GPIO 0
```

Location: bl_recovery.h:44

8.6.40 UART_CLK

```
#define UART_CLK 8000000
```

Set UART peripheral clock.

Location: bl_target.h:43

8.6.41 SENSOR_CLK

```
#define SENSOR_CLK 32768
```

Set sensor clock.

Location: bl_target.h:46

8.6.42 USER_CLK

```
#define USER_CLK 1000000
```

Set user clock.

Location: bl_target.h:49

8.6.43 VCC_BUCK_ENABLE

```
#define VCC_BUCK_ENABLE (1)
```

Enable or disable the buck converter.

The system allows for two methods of reducing the battery power supply from a higher voltage (1.2V-3.6V) to usable supply voltage (1.0V-1.31V). If the VBAT supply voltage is less than 1.4V, this should be disabled so that the device uses the low drop out (LDO) regulator. Otherwise, the buck (DC-DC) converter may be enabled. Set this to: => 0 to disable buck converter mode and enable linear mode => 1 to enable buck converter mode and disable linear mode

Location: bl_target.h:63

8.6.44 BL_TICKER_TIME_MS

```
#define BL_TICKER_TIME_MS 10
```

Define the time in ms for each interrupt.

Location: bl_ticker.h:45

8.6.45 BL_DEBUG

```
#define BL_DEBUG swmLogVerbose
```

Define the standard verbose/debug tracing routine.

Location: bl_trace.h:48

8.6.46 BL_TRACE

```
#define BL_TRACE swmLogInfo
```

Define the standard tracing routine.

Location: bl_trace.h:51

8.6.47 BL_WARNING

```
#define BL_WARNING swmLogWarn
```

Define the standard warning message routine.

Location: bl_trace.h:54

8.6.48 BL_ERROR

```
#define BL_ERROR swmLogError
```

Define the standard error message routine.

Location: bl_trace.h:57

8.6.49 BL_UART_RX_TIMEOUT_MS

```
#define BL_UART_RX_TIMEOUT_MS (3000)
```

Define the receive timeout in milliseconds.

Location: bl_uart.h:54

8.6.50 BL_WATCHDOG_FEED_ME_MS

```
#define BL_WATCHDOG_FEED_ME_MS (2000)
```

While waiting for UART input, ensure watch dog is fed.

Location: bl_uart.h:57

8.6.51 BL_UART_TX_TIMEOUT_MS

```
#define BL_UART_TX_TIMEOUT_MS (3000)
```

Define the send timeout in milliseconds.

Location: bl_uart.h:60

8.6.52 BL_UART_MAX_RX_LENGTH

```
#define BL_UART_MAX_RX_LENGTH (2048)
```

Define the maximum length of a single receive operation.

Location: bl_uart.h:63

8.6.53 BL_UART_MAX_TX_LENGTH

```
#define BL_UART_MAX_TX_LENGTH (2048)
```

Define the maximum length of a single send operation.

Location: bl_uart.h:66

8.6.54 BL_BAUD_RATE

```
#define BL_BAUD_RATE 115200
```

Define a baud rate for loading.

Location: bl_uart.h:69

8.6.55 BL_UART_DELAY_CYCLES

```
#define BL_UART_DELAY_CYCLES ((20 * SystemCoreClock) / BL_BAUD_RATE)
```

Define a delay time to allow the hardware buffers to clear.

Location: bl_uart.h:72

8.6.56 UPDATE_GPIO

```
#define UPDATE_GPIO 14
```

Define the GPIO pin to be used to indicate an update is required.

Location: bl_update.h:47

8.6.57 MIN

```
#define MIN ((a) < (b) ? (a) : (b))
```

Define a shorthand to get the minimum of two values.

Location: bl_util.h:46

8.6.58 MAX

```
#define MAX ((a) > (b) ? (a) : (b))
```

Define a shorthand to get the maximum of two values.

Location: bl_util.h:49

8.6.59 BITS2BYTES

```
#define BITS2BYTES ((x + 7) >> 3)
```

Calculate the number of bytes needed to hold x bits.

Location: bl_util.h:52

8.6.60 BITS2HALFWORDS

```
#define BITS2HALFWORDS ((x + 15) >> 4)
```

Calculate the number of 16 bit words needed to hold x bits.

Location: bl_util.h:55

8.6.61 BL_VERSION_ENCODE

```
#define BL_VERSION_ENCODE (((m) << 12) | ((n) << 8) | (r))
```

Define a mechanism to encode a version number as a uint16_t.

Location: bl_versions.h:54

8.6.62 BL_VERSION_DECODE

```
#define BL_VERSION_DECODE ((num >> 12) & 0xF), ((num >> 8) & 0xF), (num & 0xFF)
```

Define a mechanism to decode a version number from a uint16_t.

Location: bl_versions.h:57

8.6.63 BL_BOOT_VERSION

```
#define BL_BOOT_VERSION __attribute__ ((section(".rodata.boot.version")))    \
   const BL_BootAppVersion_t blBootAppVersion =         \
   {                                                    \
       id, BL_VERSION_ENCODE(major, minor, revision)    \
   };
```

Define the boot version including name and ensure it is stored in an easily accessible location.

Location: bl_versions.h:62

8.6.64 BL_WATCHDOG_MAX_HOLD_OFF_SECONDS

```
#define BL_WATCHDOG_MAX_HOLD_OFF_SECONDS 600
```

Define the maximum time that can elapse before the watchdog must be refreshed.

Location: bl_watchdog.h:50

8.7 Secure Bootloader Sample Reference Function Documentation

8.7.1 BL_CheckRemapAddressSpace

```
uint32_t BL_CheckRemapAddressSpace(uint32_t base, uint32_t address)
```

Determine download address based on given address which may be in bootloader or application space.

Location: bl_check.h:87

Parameters

Direction	Name	Description
	base	The base address of the application being checked.
	address	The given address in either bootloader or application space.

Return

The adjusted address.

8.7.2 BL_CheckGetApplicationSize

```
uint32_t BL_CheckGetApplicationSize(uint32_t address)
```

Fetch the application size from a buffer defined by base address of the application vector table.

Location: bl_check.h:95

Parameters

Direction	Name	Description
	address	The address of the base of the vector table.

Return

the size derived from the application or zero if invalid.

8.7.3 BL_CheckRelocatedApplicationSize

```
uint32_t BL_CheckRelocatedApplicationSize(uint32_t address)
```

Fetch the application size from a buffer defined by base address of the application vector table.

Location: bl_check.h:103

Parameters

Direction	Name	Description
	address	The address of the base of the vector table.

Return

the size derived from the application or zero if invalid.

8.7.4 BL_CheckIfImageUpdateAvailable

```
BL_UpdateType_t BL_CheckIfImageUpdateAvailable()
```

Check for a valid update using the non-secure file format.

When dealing with a non-secure image, the following checks must be made:

Location: bl_check.h:116

- The address must be properly aligned and within a sensible range.
- The stack pointer resides in RAM, is 64 bit aligned, allows 10 words.
- The reset ISR follows the vector table address The address to check for a valid image. extent The maximum extent of the area holding the image.
  Return

  Type of image update available in download area.

8.7.5 BL_CheckIfSecureImageUpdateAvailable

```
bool BL_CheckIfSecureImageUpdateAvailable()
```

Check for a valid update using the secure file format.

When dealing with a secure image, the following checks must be made:

Location: bl_check.h:132

- The address must be properly aligned and within a sensible range.
- The stack pointer resides in RAM, is 64 bit aligned, allows 10 words.
- The reset ISR follows the vector table.

- The full certificate chain must be authenticated. updateType The type of update being requested. address The address to check for a valid image. extent The maximum extent of the area holding the image.
  Return

  True if the image has security signature, false otherwise.

8.7.6 BL_CheckFindSecondaryImageLocation

```
 void BL_CheckFindSecondaryImageLocation(uint32_t primaryBase, uint32_t primaryExtent,
uint32_t * secondaryBase, uint32_t * secondaryExtent)
```

Based on a primary image address, calculate the potential location and extent of any secondary image.

Location: bl_check.h:144

Parameters

Direction	Name	Description
	primaryBase	The base address of the primary image, used to locate the secondary one.
	primaryExtent	The maximum extent of the primary application;
	secondaryBase	
	secondaryExtent	

8.7.7 BL_ConfigIsValid

```
BL_ConfigStatus_t BL_ConfigIsValid(BL_AppConfiguration_t * configBase)
```

Helper function to return the configuration area status.

Location: bl_configuration.h:85

Parameters

Direction	Name	Description
	configBase	Defines the base address of the configuration block.

Return

BL_CONFIG_OKAY if the configuration area has a valid CRC, BL_CONFIG_CORRUPT otherwise.

8.7.8 BL_ConfigCertificateAddress

```
uint32_t BL_ConfigCertificateAddress(BL_AppConfiguration_t * configBase, BL_LoaderCertType_t cert)
```

Fetch the address of the requested structure.

Location: bl_configuration.h:93

Parameters

Direction	Name	Description
	configBase	Defines the base address of the configuration block.
	cert	A requested certificate.

Return

The address of the requested certificate or zero if invalid request.

8.7.9 BL_FCSInitialize

```
BL_FCSStatus_t BL_FCSInitialize(uint8_t * buffer, size_t length, BL_FCS_t fcs)
```

Initialize the FCS module, deriving the selected algorithm from the provided sample data.

Location: bl_fcs.h:86

Parameters

Direction	Name	Description
	buffer	A buffer of bytes to be FCS'd.
	length	The number of bytes.
	fcs	The expected FCS value.

Return

BL_FCS_NO_ERROR if the FCS algorithm can be identified. BL_FCS_UNRECOGNIZED if the FCS algorithm cannot be identified.

8.7.10 BL_FCSQuery

```
BL_FCSAlgorithm_t BL_FCSQuery()
```

Query the currently selected FCS algorithm.

Location: bl_fcs.h:92

Return

The currently selected algorithm.

8.7.11 BL_FCSAuthenticationRequired

```
bool BL_FCSAuthenticationRequired()
```

Provides a mechanism to determine if the loading process should apply authentication to the protocol and images.

Location: bl_fcs.h:99

Return

True if authentication is required

8.7.12 BL_FCSSelect

```
BL_FCSStatus_t BL_FCSSelect(BL_FCSAlgorithm_t algo)
```

Select a specific FCS algorithm.

Location: bl_fcs.h:108

Parameters

Direction	Name	Description
	algo	Selected from BL_FCSAlgorithm_t.

Return

BL_FCS_NO_ERROR If the algorithm is valid. BL_FCS_UNRECOGNIZED If the algorithm is not valid.

8.7.13 BL_FCSCheck

```
BL_FCSStatus_t BL_FCSCheck(uint8_t * buffer, size_t length, BL_FCS_t fcs)
```

Check the validity of a buffer against a given FCS.

Location: bl_fcs.h:119

Parameters

Direction	Name	Description
	buffer	A buffer of bytes to calculate a FCS over.
	length	The number of bytes.
	fcs	The expected FCS value.

Return

BL_FCS_VALID if the FCS matches the data. BL_FCS_INVALID if the FCS does not match the data.

8.7.14 BL_FCSCalculate

```
BL_FCSStatus_t BL_FCSCalculate(uint8_t * buffer, size_t length, BL_FCS_t * fcs)
```

Calculate the FCS of a given buffer.

Location: bl_fcs.h:130

Parameters

Direction	Name	Description
	buffer	A buffer of bytes to calculate a FCS over.
	length	The number of bytes.
	fcs	The calculated FCS value.

Return

BL_FCS_NO_ERROR if the FCS can be calculated. BL_FCS_INVALID if an error is detected when calculating the FCS.

8.7.15 BL_FCSAccumulateCRC

```
uint32_t BL_FCSAccumulateCRC(uint8_t * buffer, size_t length)
```

Helper method to accumulate a CRC given a buffer and a length.

Location: bl_fcs.h:141

Parameters

Direction	Name	Description
	buffer	A buffer of bytes to calculate a CRC on.
	length	The number of bytes.

NOTE: This is expected to be used for RAM buffers where the use of the flash copier can't be used. The CRC engine should be initialised prior to calling this function.

8.7.16 BL_FlashInitialize

```
void BL_FlashInitialize()
```

Initialize the flash subsystem.

Location: bl_flash.h:55

8.7.17 BL_FlashSaveSector

```
FlashStatus_t BL_FlashSaveSector(uint8_t * address, size_t length, uint8_t * buffer)
```

Save a buffer to a specified flash address.

Location: bl_flash.h:67

Parameters

Direction	Name	Description
	address	The address in flash to save the buffer.
	length	The number of bytes to save.
	buffer	A pointer to a buffer of data to be written.

Return

FLASH_ERR_NONE if the operation is successful otherwise an error code the flash library.

NOTE: The start address is expected to start on a sector boundary.

8.7.18 BL_ImageInitialize

```
BL_ImageType_t BL_ImageInitialize(uint8_t * address, size_t length, uint32_t crc)
```

Initialize the image module for a specific set of image attributes.

Location: bl_image.h:98

Parameters

Direction	Name	Description
	address	The base address of the image being loaded.
	length	The length of the image in bytes.
	crc	the crc of the image being loaded.

Return

The type of image recognized.

8.7.19 BL_ImageAddress

```
uint32_t BL_ImageAddress(uint32_t address)
```

Convert an address to take into account potential offsets.

Location: bl_image.h:107

Parameters

Direction	Name	Description
	address	The address in an image.

Return

The converted address.

8.7.20 BL_ImageAddressRange

```
void BL_ImageAddressRange(uint8_t * address, size_t length, BL_ImageSplitRange_t * range)
```

Helper routine which allows access of the image as a contiguous block of addresses, wrapping around the reserved block.

Location: bl_image.h:118

Parameters

Direction	Name	Description
	address	An address within an image that may need to be adjusted.
	length	the length of the address range.
	range	A split range object that indicates where the address range needs to be split.

8.7.21 BL_ImageCopyMemoryRange

```
void BL_ImageCopyMemoryRange(uint8_t * dst, BL_ImageSplitRange_t * range)
```

Copy a possibly split memory range to a contiguous buffer.

Location: bl_image.h:127

Parameters

Direction	Name	Description
	dst	The destination buffer.
	range	The range defining the source locations.

8.7.22 BL_ImageSaveBlock

```
BL_ImageStatus_t BL_ImageSaveBlock(BL_ImageOperation_t * operation)
```

Save a block of data from a RAM buffer to the next block in Flash.

Location: bl_image.h:135

Parameters

Direction	Name	Description
	operation	Defines the address and length of the block to be saved.

Return

Status code indicating if the save operation failed

8.7.23 BL_ImageVerify

```
BL_ImageStatus_t BL_ImageVerify()
```

Verify the most recently loaded image.

Location: bl_image.h:144

Return

BL_IMAGE_NO_ERROR If the CRC matches the data. BL_IMAGE_VERIFY_ERROR If the CRC does not match the data.

8.7.24 BL_ImageAuthenticate

```
BL_ImageStatus_t BL_ImageAuthenticate(BL_ImageType_t imageType, uint32_t * address, size_
t length, bool verifyImages)
```

Authenticate a loaded image.

Location: bl_image.h:157

Parameters

Direction	Name	Description
	imageType	The type of the image being authenticated.
	address	The base address of the image to be authenticated.
	length	The size of the area in bytes.
	verifyImages	Flag indicating that the s/w images must be validated.

Return

BL_IMAGE_NO_ERROR If the CRC matches the data. BL_IMAGE_AUTHENTICATE_ERROR If the authentication fails.

8.7.25 BL_ImageAuthenticateCurrent

```
BL_ImageStatus_t BL_ImageAuthenticateCurrent()
```

Authenticate the most recently loaded image.

Location: bl_image.h:167

Return

BL_IMAGE_NO_ERROR If the CRC matches the data. BL_IMAGE_AUTHENTICATE_ERROR If the authentication fails.

8.7.26 BL_ImageIsValid

```
bool BL_ImageIsValid(uint32_t address, size_t length)
```

Check if there is a valid image to start.

Location: bl_image.h:176

Parameters

Direction	Name	Description
	address	The address of the image in flash.
	length	The length of the image in bytes.

Return

True if there is a valid application to start. False otherwise.

8.7.27 BL_ImageSaveAddress

```
uint32_t BL_ImageSaveAddress(BL_ImageType_t imageType, uint32_t address)
```

Return the download address corresponding to the requested address.

Location: bl_image.h:185

Parameters

Direction	Name	Description
	imageType	The type of the image being authenticated.
	address	The requested address

Return

The download address

8.7.28 BL_ImageStartApplication

```
void BL_ImageStartApplication(uint32_t imageBaseAddress)
```

Start the image stored in flash.

Location: bl_image.h:192

Parameters

Direction	Name	Description
	imageBaseAddress	The base address of the image to be started

8.7.29 BL_LoaderPerformFirmwareLoad

```
void BL_LoaderPerformFirmwareLoad()
```

Perform a firmware update over the UART interface.

Location: bl_loader.h:147

8.7.30 BL_LoaderCertificateAddress

```
uint32_t BL_LoaderCertificateAddress(BL_LoaderCertType_t cert)
```

Fetch the address of the requested structure.

Location: bl_loader.h:154

Parameters

Direction	Name	Description
	cert	A requested certificate.

Return

The address of the requested certificate or zero if invalid request.

8.7.31 BL_RecoveryInitialize

```
void BL_RecoveryInitialize()
```

Define the initialization routine for the Debug Catch feature.

Location: bl_recovery.h:57

8.7.32 BL_TargetInitialize

```
void BL_TargetInitialize()
```

Target initialization function, loads the trim values and sets up the various clocks used in the system.

Location: bl_target.h:77

8.7.33 BL_TargetReset

```
void BL_TargetReset()
```

Reset the device using NVIC.

Location: bl_target.h:82

8.7.34 BL_TickerInitialize

```
void BL_TickerInitialize()
```

Initialize the timer tick.

Location: bl_ticker.h:58

8.7.35 BL_TickerTime

```
uint32_t BL_TickerTime()
```

Get the current timer tick value.

Location: bl_ticker.h:64

Return

The time since the ticker was initialized in ms.

8.7.36 SysTick_Handler

```
void SysTick_Handler()
```

System tick interrupt handler, required by the ticker.

Location: bl_ticker.h:69

8.7.37 BL_TraceInitialize

```
void BL_TraceInitialize()
```

Initialize the trace sub-system.

Location: bl_trace.h:70

8.7.38 BL_UARTInitialize

```
void BL_UARTInitialize()
```

Initialize the UART subsystem.

Location: bl_uart.h:102

8.7.39 BL_UARTReceiveAsync

```
BL_UARTStatus_t BL_UARTReceiveAsync(uint8_t * buffer, size_t length)
```

Start receiving a fixed length data buffer using the UART.

Location: bl_uart.h:119

Parameters

Direction	Name	Description
	buffer	A pointer to a buffer in which to store the incoming data.
	length	The number of bytes to store in the buffer. (> 0)

Return

BL_UART_NO_ERROR if the operation is started successfully. BL_UART_INVALID_PARAMETER if the length is zero. BL_UART_RX_BUSY if another receive operation is currently active.

NOTE: No checking is performed to ensure that the buffer is big enough to hold the requested number of bytes. The calling function must ensure this is valid.

NOTE: There must be no pending receive operation pending when this is invoked.

8.7.40 BL_UARTReceiveComplete

```
BL_UARTStatus_t BL_UARTReceiveComplete(uint8_t * buffer, size_t length, BL_FCS_t * fcs)
```

Complete the reception of an executing receive operation.

Location: bl_uart.h:137

Parameters

Direction	Name	Description
	buffer	A pointer to a buffer in which to store the incoming data.
	length	The number of bytes to store in the buffer. (> 0)
	fcs	Indicating if a FCS should be calculated on the input. NULL indicates no FCS calculation needed.

Return

BL_UART_NO_ERROR if the operation completes successfully. BL_UART_RX_IDLE if there is no pending receive operation. BL_UART_RX_TIMEOUT if the receive operation timed out. BL_UART_BAD_FCS if the receive operation had an invalid FCS.

NOTE: There must be an existing receive operation pending.

NOTE: This is a blocking operation.

8.7.41 BL_UARTReceive

```
BL_UARTStatus_t BL_UARTReceive(uint8_t * buffer, size_t length, BL_FCS_t * fcs)
```

Receiving a fixed length data buffer using the UART.

Location: bl_uart.h:161

Parameters

Direction	Name	Description
	buffer	A pointer to a buffer in which to store the incoming data.
	length	The number of bytes to store in the buffer. (> 0)
	fcs	Indicating if a FCS should be calculated on the input. NULL indicates no FCS calculation needed.

Return

BL_UART_NO_ERROR if the operation is started successfully. BL_UART_INVALID_PARAMETER if the length is zero. BL_UART_RX_BUSY if another receive operation is currently active. BL_UART_RX_TIMEOUT if the receive operation timed out. BL_UART_BAD_FCS if the receive operation had an invalid FCS.

NOTE: No checking is performed to ensure that the buffer is big enough to hold the requested number of bytes. The calling function must ensure this is valid.

NOTE: There must be no pending receive operation pending when this is invoked.

NOTE: This is a blocking operation.

8.7.42 BL_UARTSendAsync

```
BL_UARTStatus_t BL_UARTSendAsync(uint8_t * buffer, size_t length, BL_FCS_t * fcs)
```

Start sending a fixed length data buffer using the UART.

Location: bl_uart.h:175

Parameters

Direction	Name	Description
	buffer	A pointer to a buffer holding the outgoing data.
	length	The number of bytes to send. (> 0)
	fcs	The FCS of the buffer to accompany the transmission.

Return

BL_UART_NO_ERROR if the operation is started successfully. BL_UART_INVALID_PARAMETER if the length is zero. BL_UART_TX_BUSY if another send operation is currently active.

NOTE: There must be no pending transmit operation pending when this is invoked.

8.7.43 BL_UARTSendComplete

```
BL_UARTStatus_t BL_UARTSendComplete()
```

Complete the transmission of an executing send operation.

Location: bl_uart.h:188

Return

BL_UART_NO_ERROR if the operation completes successfully. BL_UART_TX_IDLE if there is no pending receive operation. BL_UART_TX_TIMEOUT if the send operation timed out.

NOTE: There must be an existing transmit operation pending.

NOTE: This is a blocking operation.

8.7.44 BL_UARTSend

```
BL_UARTStatus_t BL_UARTSend(uint8_t * buffer, size_t length, BL_FCS_t * fcs)
```

Send a fixed length data buffer using the UART.

Location: bl_uart.h:205

Parameters

Direction	Name	Description
	buffer	A pointer to a buffer holding the outgoing data.
	length	The number of bytes to send. (> 0)
	fcs	The FCS of the buffer to accompany the transmission.

Return

BL_UART_NO_ERROR if the operation is started successfully. BL_UART_INVALID_PARAMETER if the length is zero. BL_UART_TX_BUSY if another send operation is currently active. BL_UART_TX_TIMEOUT if the send operation timed out.

NOTE: There must be no pending transmit operation pending when this is invoked.

NOTE: This is a blocking operation.

8.7.45 BL_UpdateInitialize

```
void BL_UpdateInitialize()
```

Initialize the firmware update component.

Location: bl_update.h:60

8.7.46 BL_UpdateRequested

```
bool BL_UpdateRequested()
```

Check if a firmware update is being requested.

Location: bl_update.h:66

Return

True if the update pin has been pulled low. False otherwise.

8.7.47 BL_UpdateProcessPendingImages

```
void BL_UpdateProcessPendingImages()
```

This will check for any pending images which have previously been downloaded and if any are found will copy them to the appropriate location for execution.

Location: bl_update.h:73

8.7.48 BL_ImageSelectAndStartApplication

```
void BL_ImageSelectAndStartApplication()
```

This will attempt to start any images which are available.

This will first try to validate and if necessary authenticate the primary image. If this is successful it will then perform similar validation and authentication on the secondary image. If both the primary and secondary image validation is successful then it will start the secondary image. If only the primary image is valid then it will be started instead. If both the primary and secondary image fail the validation steps then no image will be started and the function will return to the caller and the bootloader will enter the loading state.

Location: bl_update.h:90

8.7.49 BL_VersionsGetInformation

```
void BL_VersionsGetInformation(BL_BootAppVersion_t * version, uint32_t address)
```

Get the version information from a suitable application.

Location: bl_versions.h:101

Parameters

Direction	Name	Description
	version	The structure into which the information should be copied.
	address	The base address of the application under consideration.

8.7.50 BL_VersionsGetHello

```
void BL_VersionsGetHello(BL_HelloResponse_t * response)
```

Fetch the hello response from the bootloader.

Location: bl_versions.h:108

Parameters

Direction	Name	Description
	response	The structure into which the hello response should be copied.

8.7.51 BL_WatchdogInitialize

```
void BL_WatchdogInitialize()
```

Initialise the watchdog module.

Location: bl_watchdog.h:64

8.7.52 BL_WatchdogSetHoldTime

```
void BL_WatchdogSetHoldTime(uint32_t seconds)
```

Set the watchdog hold off time to seconds.

Location: bl_watchdog.h:76

Parameters

Direction	Name	Description
	seconds	The number of seconds to allow before the watchdog bites.

NOTE: This allows the watchdog interrupt to fire but refreshes the watchdog itself until the requested number of seconds has elapsed. This is a crude mechanism to prevent long running calculations such as RSA key generation from causing a system reset.

8.7.53 WATCHDOG_IRQHandler

```
void WATCHDOG_IRQHandler()
```

Define an interrupt handler for the watchdog interrupt.

Location: bl_watchdog.h:82

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

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

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

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

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

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

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

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

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

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

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:

Literature Distribution Center for onsemi

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

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

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

Email: orderlit@onsemi.com

N. American Technical Support:

800-282-9855 Toll Free USA/Canada

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

onsemi Website: www.onsemi.com

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

For additional information, please contact your local Sales Representative

M-20892-003