emBoot-Secure

Secure boot loader

User Guide & Reference Manual

Document: UM23001
Software Version: 2.10.0
Revision: 0
Date: July 13, 2026

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: ticket_emboot-secure@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: July 13, 2026

Software	Date	By	Description
2.10.0	2026-07-13	MHA	Update to reflect configuration changes in the latest emCrypt version (optimization goals). Tools • SigServerAdmin: Add documentation for verification of keys and server restart. • KeyTool: Add documentation about how to change the passphrase used to encrypt a key.
2.00.2	2026-02-26	TISC	Clarify introduction
2.00.1	2026-02-26	TISC	Clarify introduction
2.00.0	2026-02-01	*	Initial version

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (compiler, linker, Integrated Development Environment).
- The C programming language.
- The target processor.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction

In today's connected world, the ability to provide updates to embedded devices is more important than ever. Devices need to be updated to provide new functionality, connect with new services and receive security updates. Updates are no longer only a necessity to establish a strong reputation for customer trust, but are also required for legal reasons. New security regulations for embedded devices, such as the EU's Cyber Resilience Act (CRA), require the ability to install trusted updates over long timescales.

Aside from updates, devices also need to be secured against manipulation. Embedded devices handle a large range of tasks which are often safety-critical, involve financial transactions, or provide sensitive data to end users or service providers which must not be forged. Manufacturers of embedded devices need to ensure that the software on their devices cannot be manipulated, even before they are allowed to enter certain markets at all. Due to these reasons, manufacturers of embedded devices need to integrate updateability into the core of their products right from the start of the design process.

SEGGER's emBoot-Secure, which is described in detail in this manual, is a collection of multiple components that make designing and implementing a secure update mechanism for embedded devices easy. In particular, emBoot-Secure consists of the following components:

- **Easy-to-use software tools** for creating update packages, compressing them, protecting them from manipulation and inspection, and managing the required cryptographic keys
- **A bootloader library**, which makes it easy to build a custom bootloader that is tailored to a specific embedded project and handles firmware updates efficiently and securely
- **Example bootloader implementations**, which show how the bootloader library can be used and adapted to a specific embedded project
- **The Signature Server**, a dedicated hardware device that can optionally be used to generate, store, and apply cryptographic keys
- **Comprehensive documentation** (this document)

To ease the update process, the firmware of the embedded device is divided into two software components: the bootloader, which is built with the emBoot-Secure library, and the application firmware. While the application firmware provides the core functionality of the device, the bootloader is responsible for verifying the authenticity and integrity of update packages and for their installation. Each time the embedded device is turned on, the bootloader is also responsible for verifying that the application firmware has not been manipulated.

1.1 Principle of operation

This section introduces the emBoot-Secure packaging process and the bootloader's logic at startup.

1.1.1 Packaging process

At a high level, the process of integrating emBoot-Secure into an embedded software project can be separated into the following five steps.

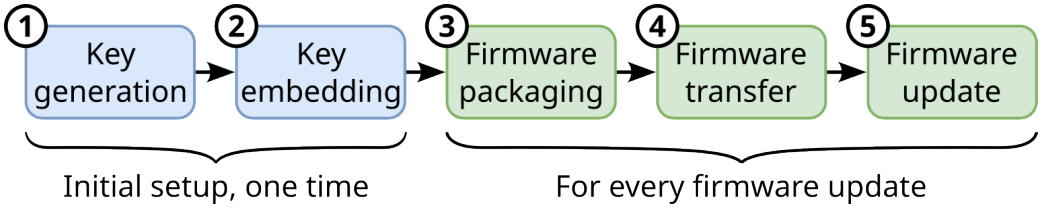


Figure: The 5-step process of setting up and using emBoot-Secure

This process involves some initial steps to generate the required cryptographic keys and build a tailored bootloader for the project that contains those keys. In addition, each firmware update needs to be packaged, transferred to the target, and installed.

Key generation

The security services that emBoot-Secure provides rely on cryptographic procedures, which require some cryptographic keys to operate. In particular, two different kinds of keys are required:

- **Signing key pair:** To implement protection against manipulation of the update package, the concept of digital signatures is used. This concept requires a pair of two keys, a public and a private key. The private key is kept in secret at the manufacturer of the embedded device and is used to produce valid firmware update packages. The public key is embedded into every device and can be used to verify whether an update package has been manipulated or not. emBoot-Secure uses the established RSA or ECDSA algorithm for this purpose.
- **Encryption key:** To implement protection against inspection of the update package, for example by malicious actors or competitors, update packages can optionally be encrypted. This requires another (single) key, which is used for both encryption (at the manufacturer) and decryption (on the embedded device). emBoot-Secure uses the established AES-GCM algorithm for this purpose.

These cryptographic keys need to be generated once and are valid for the whole lifecycle of the product.

The private signing key is a valuable asset in the process of creating cryptographically signed firmware images and update packages. If an attacker can gain access to the key, manipulated firmware images can be signed and installed into the target device. To prevent malicious actors from gaining access to the private key, emBoot-Secure provides its Signature Server. This server, a dedicated hardware appliance, can securely store private keys.

Key embedding

The generated keys are converted into C files using the key generation tool and linked into the bootloader.

Firmware packaging

Whenever a new update is to be distributed, it is packaged into an update package using the firmware preparation tool. This tool applies the signature and optionally compresses and encrypts the firmware image.

The signing process consists of two steps, hash computation and creation of the signature. In the first step, a hash over the firmware image is computed. A hash is a cryptographic checksum, which is designed such that changing a single bit in the firmware image will

change the hash result in an unpredictable fashion. This makes it practically impossible to modify the image in such a way that the modified version produces the same hash as the original version. The emBoot-Secure library supports several cryptographic hash functions such as SHA256, SHA512 or SHA1.

In the second step, the digital signature is computed. To prevent manipulation of the hash value, the manufacturer uses the private key to sign it. The bootloader can later verify the signature using the public key, which is embedded in the bootloader in the target device. The supported signature algorithms are RSA and ECDSA, with various key sizes.

The firmware preparation tool either computes the signature locally on a computer or requests it from the Signature Server. If the Signature Server is used, the firmware preparation tool sends the hash to the server and receives the signature in return, which is then embedded into the update package. The private key, however, does not leave the Signature Server and can therefore not be leaked.

In addition, the image can optionally be compressed and encrypted. Compression enables the creation of small update packages, which is especially important when they need to be distributed to the target using low-bandwidth channels such as CAN or small RF data packages. The emBoot-Secure library features compression of the update package using SEGGER's SMASH-2 algorithm, which is optimized for compression of firmware images. It has a very small footprint and is therefore ideal for use in bootloaders.

Encryption prevents update packages from inspection by third parties. emBoot-Secure optionally encrypts update packages with the AES-GCM algorithm.

The firmware preparation tool can also produce a signed firmware image for initial manufacturing, which can be written into flash memory using standard tools. This step is required because the bootloader needs to verify the authenticity of the installed firmware image on every start.

Firmware transfer

For update package distribution, the manufacturer of the embedded device is free to implement a method that fits the target ecosystem - such as Wi-Fi, Bluetooth, Ethernet, USB, SD card, or CAN.

The target's application firmware is responsible for storing the update package on the device, such that the bootloader can find it during the next reboot.

Firmware update

When the bootloader discovers a new update package, it first compares the version number of the update with version number of the currently installed firmware. If the version number of the update package is newer, the bootloader unpacks it. If needed, the update package is decrypted and/or decompressed. Then, the signature is verified and, given a valid signature, the new firmware is written to flash memory. This process is explained in detail in the next section.

1.1.2 Bootloader logic

This section describes the principle of operation of the emBoot-Secure bootloader library and its interaction with the target system as part of a bootloader. The figure below shows how the bootloader operates after a reset of the microcontroller.

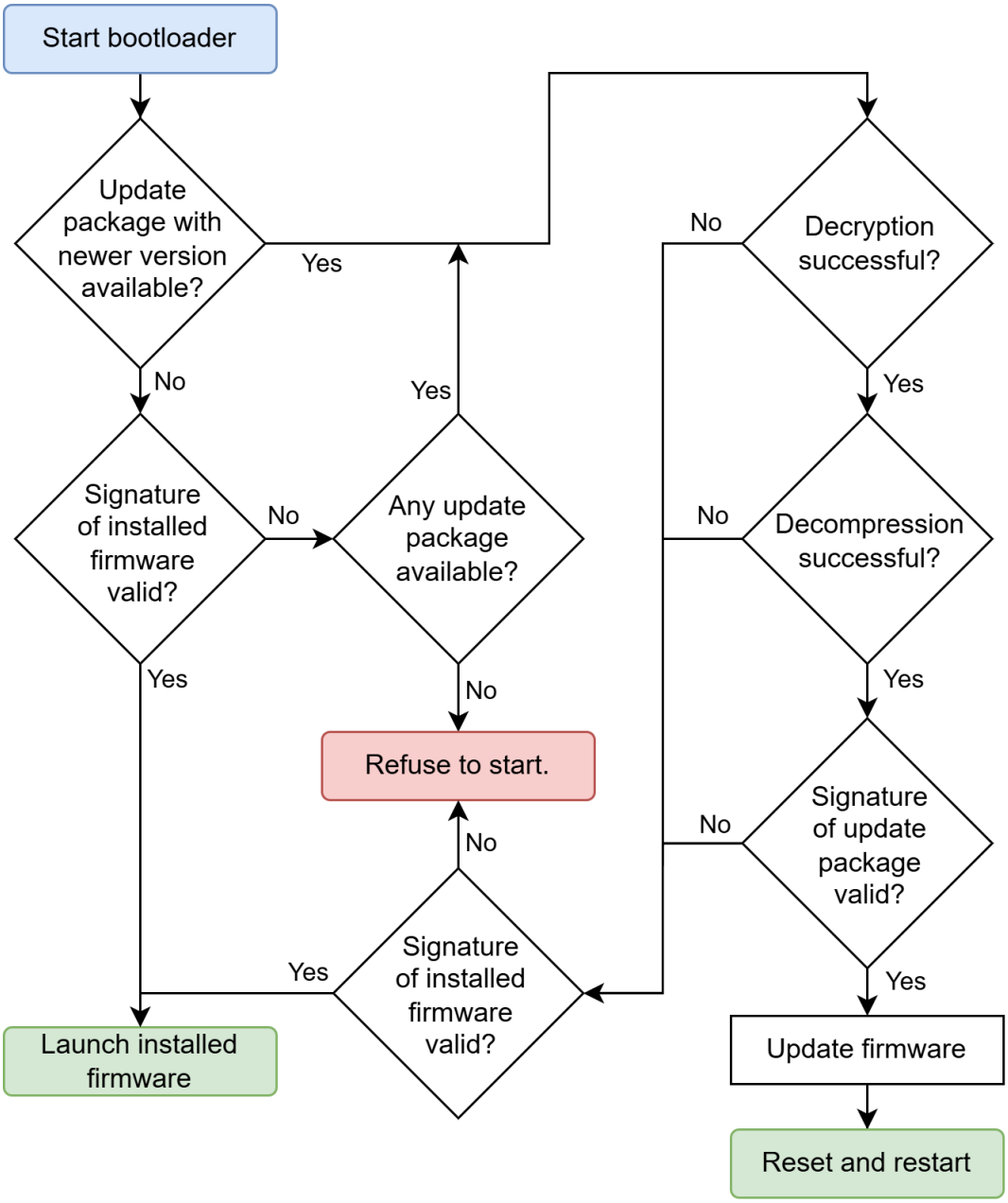


Figure: Flowchart of the update process

Searching for update packages

The bootloader starts by searching for update packages. These packages are stored in so-called areas, which are addressed using an index. During the configuration of the bootloader, the manufacturer specifies the number of areas and implements a function which allows the bootloader to read data from them. The bootloader itself is agnostic to the implementation details of these areas - they can be regions of internal flash memory, external flash memory or files in a filesystem.

Version number check

When the bootloader determines that an area contains an update package, it compares the version number of the update package with the version number of the installed firmware image. The version numbers are stored in the headers of both the update package and the firmware image. By default, the bootloader will only install update packages with higher version numbers, in order to prevent a firmware rollback, but this setting can be changed if needed. Once an update package with a suitable version number has been found, the bootloader considers it for installation.

Update package decryption

If the update package is encrypted, the bootloader will attempt to decrypt it with its embedded encryption key. The AES-GCM encryption method includes an authentication layer, which allows to detect manipulated or corrupted packages early on. If this authentication fails, the update package is rejected immediately.

Update package decompression

If the update package is compressed, it is decompressed at this stage. The update package is rejected if any bitstream errors occur.

Firmware signature verification

The emBoot-Secure library's core function is the verification of both firmware update packages and installed firmware images. The verification is needed to confirm that the application firmware was supplied by the manufacturer and has not been tampered with, neither before nor after installation. In order to verify the authenticity of an update package, the digital signature is checked in two steps: hash computation and signature verification.

In the first step, the hash of the firmware image in the update package is computed and compared with the expected hash value stored in the package's metadata. If the hash matches, it has to be checked for manipulation. Therefore, in the second step, the digital signature that has been computed by the manufacturer when the update was packaged is verified using the public key, which is embedded in the bootloader.

Update installation

If the bootloader determines that the update package is authentic, it proceeds with installation. This requires the bootloader to erase the region of flash memory which contains the previous version of the application firmware and to write the firmware image obtained from the update package into it. The process in which flash memory is erased and written to is highly dependent on the employed target hardware. Therefore, the bootloader employs a flash memory driver, which is usually part of the emBoot-Secure shipping package. The manufacturer configures the flash memory layout which the emBoot-Secure library should use via compile-time configuration flags. The bootloader uses these settings to determine the region of flash memory into which it must write the updated firmware image.

Since the bootloader was designed such that it can access the update package without help from the application firmware, it can resume the update process if it gets interrupted after the previously installed application firmware has been (partially) erased from flash memory. This ensures that the device does not get "bricked".

Booting without an update package

When the bootloader has finished the installation of the update package, it resets the microcontroller. After the reset, the bootloader again searches the regions which may contain update packages, but it should not find a newer update package any more.

At this point, the bootloader does not know that the firmware image has just been installed and therefore proceeds to check the authenticity of the installed firmware image. This step is necessary to detect accidental or intentional manipulation of the firmware image. During the installation of the firmware image, the header and the metadata which contain the version number, the hash value, and the manufacturer's cryptographic signature have been written into the flash memory along with the raw firmware image. Therefore, the bootloader can again compute the hash, compare it against the hash value stored in the metadata, and verify the signature. If the signature is valid, the bootloader reads the firmware entry point address from the header and proceeds to start the firmware image. Its work is done.

Handling invalid firmware images

If the bootloader determined that the installed firmware image is not authentic, it looks again for firmware update packages. But this time, since there is no usable firmware image installed, it will install any update package, regardless of the version number. The bootloader can also be supplied with a dedicated rescue firmware image, which is retained in a

special area independent of regular update packages. This rescue image may have reduced functionality and its only purpose may be to provide a means of obtaining a more recent update package. The manufacturer can also implement a method to determine whether a rescue image should be installed based on certain conditions, for example, when the customer presses a button or sets a jumper during start-up. Alternatively, the application could indicate a non-recoverable state to the bootloader using non-volatile, battery-powered SRAM which is available in some targets.

In the worst case scenario, the bootloader will neither find an update package nor will it be able to confirm the authenticity of the installed firmware image. In that case, it will refuse to start the application firmware and indicate an error condition, depending on the means available in the device. While this condition usually renders the device unusable, it prevents the device from starting with a damaged or tampered firmware image, which could allow an attacker to wreak havoc unnoticedly or cause dangerous malfunction of the embedded device.

1.2 Components of emBoot-Secure

emBoot-Secure consists of the following parts:

Item	Description
emBoot-Secure library	Source code of emBoot-Secure library.
Signature Server	Signature Server hardware device for secure key storage (optional).
Command line tools	Command line tools for generating keys, managing the Signature Server and creating update packages.
Documentation	The documentation.

The command line tools are provided in ELF format (without a file extension) for Linux and as `.exe` files for the Windows operating system. In this manual, the command line tools' filenames are referenced for both formats interchangeably. The command line arguments are the same for both operating systems.

1.2.1 Components of the emBoot-Secure library

emBoot-Secure is provided in source code and the exact content depends upon the versions and add-ons that you purchase. The following table shows the content of the package:

Files	Description
`Config`	Configuration header and C files.
`BOOT`	emBoot-Secure library source code.
`SEGGER`	SEGGER software component source code.
`CRYPTO`	Cryptographic library for signature validation and optional update package decryption
`COMPRESS-ToGo`	Code for optional decompression of update package
`Application`	emBoot-Secure sample implementations and configurations.
`BSP/<mcu>`	Sample configurations and flash routines for selected MCUs.
`Tool`	Command line tools
`Doc`	Documentation

Chapter 2

Getting started

This chapter explains how to write a secure bootloader which can update the application firmware and itself.

2.1 Overview

Partitioning of the non-volatile memory

Several areas in the non-volatile memory of the target device must be defined, especially:

- The area of the bootloader.
- The area of the application firmware.
- One or more areas where firmware update files can be stored.

The bootloader and the application firmware usually reside in memory that can be executed by the CPU. The memory addresses of these areas must be defined in the configuration file `BOOT_Conf.h`. For all other areas, it is sufficient that they can be read sequentially.

For details, see *Partitioning of the non-volatile memory* and `SBTL_ReadArea()`.

Bootloader project setup

A firmware project for the bootloader can be created with any toolchain or IDE. After an empty (“Hello-World”) application is created, all necessary emBoot-Secure source and configuration files can then be added to the project in order to build a bootloader, see *Setting up the bootloader project*.

Configuring the bootloader's behavior

The behavior of the parts of the bootloader supplied by the emBoot-Secure library are configured in the `BOOT_Conf.h`. This concerns both the search for update packages as well as supported cryptographic and compression algorithms.

Hardware (board) specific support

Each bootloader project requires hardware-specific support functions, especially for flash memory write and erase. These board support files are usually placed in a file named `BOOT_Config_<boardname>.c`. Depending on the purchase agreement, this copy of the emBoot-Secure library ships with board support files for the employed target boards.

The emBoot-Secure library searches for update packages by using the `SBTL_ReadArea()` callback. This callback needs to be implemented to allow hardware specific access to update packages, for example in a part of the device’s internal flash memory or by accessing a filesystem on external flash memory.

There are also some options of the CRYPTO library which can be configured to enable hardware cryptographic support if available on the target board.

Implementation of the top-level bootloader code (main() function).

emBoot-Secure comes with samples that can be taken as the main bootloader function or can be used as a starting point for creating a customized version.

Key creation

At least one firmware signing key pair must be generated. Key generation can either be done using the *key generation tool* or the Signature Server. Either RSA or ECDSA algorithms can be chosen, depending on the project’s individual requirements. The public key must be made available to the emBoot-Secure library using either the `SBTL_GetRSAPublicKey` or the `SBTL_GetECPublicKey()` function, respectively. A C file containing implementations of these functions, including the public key, can be created using the key generation tool.

Optionally, a firmware encryption key can be generated using the key generation tool. The firmware encryption key is requested by the emBoot-Secure library through the `SBTL_GetAESKey()` function, which can also be created from the AES key using the key generation tool.

The resulting C files containing the keys need to be added to the bootloader project.

Application firmware and integration

Application firmware needs to be adapted in order to be launchable by the bootloader. The linker has to be instructed to place it in its designated region of flash memory, leaving space for metadata such as a version number and the location of the digital signature.

Also, the vector table, which is used by the microcontroller to determine the entry point of the application and the location of interrupt handlers, has to be placed at a suitable position in the firmware's flash region and activated during application start-up. This process is described in the section *Adapting application firmware*.

Creating firmware for the target

Finally, the application firmware image has to be processed by emBoot-Secure's Firmware-Tool. For the device production process, a signed firmware image is created which can be flashed into the target alongside the bootloader. To update the application firmware and the bootloader in the field, update packages containing the signed (and optionally compressed and encrypted) firmware image are produced. These steps are covered in the section *Creating firmware for the target*.

2.2 Partitioning of the non-volatile memory

emBoot-Secure can handle the following types of areas in non-volatile memory:

Bootloader area

This area contains the bootloader, which is started by the device after power up. If the bootloader shall be able to update itself, the memory region must be erasable without affecting other parts of the memory.

Application firmware area

This area contains the application firmware of the target device and can be updated by the bootloader. It should be large enough to store any future versions of the firmware. The memory region must be erasable without affecting other parts of the memory. The firmware in this area contains a header and a signature that is verified on every start-up by the bootloader.

Update area

There can be one or several of these areas. The application firmware can store update packages into any of these areas which will be read by the bootloader on the next start-up of the device. If the bootloader can successfully verify the update package, it will transfer the new firmware into the application firmware area. emBoot-secure uses the `SBTL_ReadArea` callback to request data from this area.

Failsafe firmware area

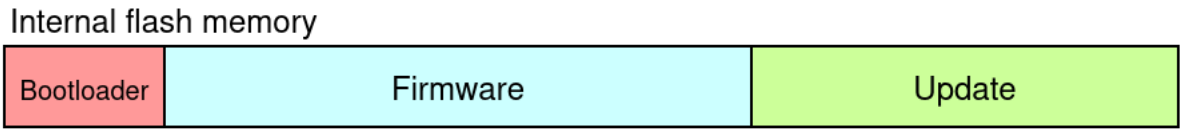
This is an optional area that contains a failsafe firmware image. The bootloader will copy this into the application firmware area in case it determines that the application firmware is corrupted and no valid update package is available. In order to get the device operational again, the failsafe firmware must contain at least the functionality to receive an update package and store it into an update area. emBoot-secure uses the `SBTL_ReadArea` callback to request data from this area.

2.2.1 Sample memory configurations

This section lists common flash memory layouts and how to configure emBoot-Secure to use these layouts. The memory layouts are described using configuration flags in `BOOT_Conf.h`. For a general explanation of these flags, see *Configuration switches for flash memory layout*. Only regions of memory which contain the bootloader and the application firmware image need to be described, since emBoot-Secure needs to be able to write into them. Areas containing update packages or failsafe firmware update packages are not described by configuration flags. Instead, emBoot-Secure will request data from these regions using the `SBTL_ReadArea()` callback, which needs to be implemented accordingly.

2.2.1.1 Simple configuration

All areas reside in internal flash of the device.



This configuration can be achieved with settings like this (in `BOOT_Conf.h`):

```
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_NORMAL

#define SBTL_BOOTLOADER_START_ADDR        0x00000000
#define SBTL_BOOTLOADER_SECTOR_SIZE           0x1000
#define SBTL_BOOTLOADER_SECTOR_COUNT            0x20
```

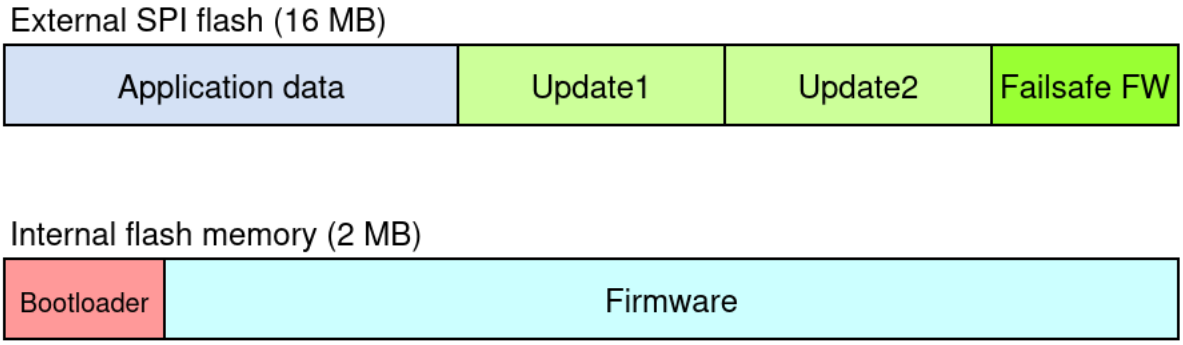


```
#define SBTL_FIRMWARE_START_ADDR          0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE             0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT             0x100
```

In this example with a 2 MB flash memory, the bootloader is in the memory area from `0x000000` to `0x01FFFF` and the firmware is in `0x020000` to `0x11FFFF`. In both areas, the smallest addressable flash blocks have a size of 4KB. Address `0x120000` denotes the starting point of the region containing firmware update packages. It can be used for one or more update packages, for example a failsafe firmware area which is not touched and an area for firmware updates.

2.2.1.2 Configuration with external SPI flash

An external flash memory can easily used to store update packages and failsafe firmware in order to fully utilize the internal flash memory for the active firmware:



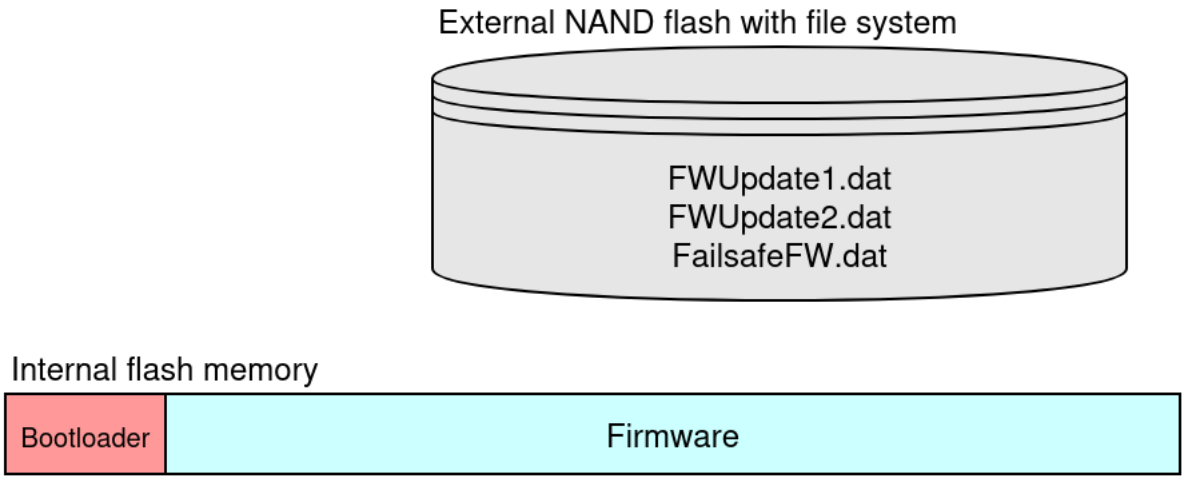
This configuration can be achieved with settings like this (in `BOOT_Conf.h`):

```
#define SBTL_NUM_AREAS_UPDATE                    2
#define SBTL_NUM_AREAS_FAILSAFE                  1
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_NORMAL

#define SBTL_BOOTLOADER_START_ADDR        0x00000000
#define SBTL_BOOTLOADER_SECTOR_SIZE           0x1000
#define SBTL_BOOTLOADER_SECTOR_COUNT            0x20

#define SBTL_FIRMWARE_START_ADDR          0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE             0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT             0x1e0
```

The bootloader is in memory area `0x000000` to `0x01FFFF` and the firmware in `0x020000` to `0x1FFFFF`.

2.2.1.3 Configuration with filesystem

If there is a filesystem on the device, it can be used to store update packages and failsafe firmware in files.

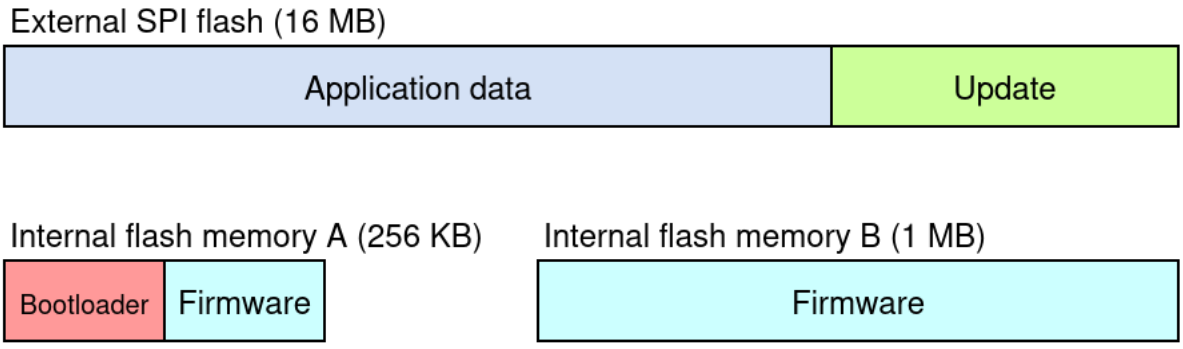


This configuration can be achieved with settings like this (in `BOOT_Conf.h`):

```
#define SBTL_NUM_AREAS_UPDATE                    2
#define SBTL_NUM_AREAS_FAILSAFE                  1
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_NORMAL

#define SBTL_BOOTLOADER_START_ADDR       0x00000000
#define SBTL_BOOTLOADER_SECTOR_SIZE          0x1000
#define SBTL_BOOTLOADER_SECTOR_COUNT           0x20

#define SBTL_FIRMWARE_START_ADDR         0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE            0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT             0x1e0
```

2.2.1.4 Scattered firmware flash

The application firmware can be scattered across multiple flash memories:



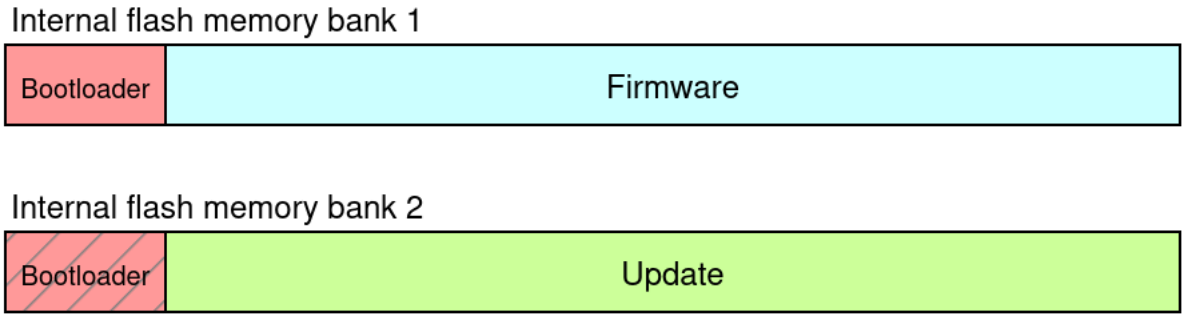
This configuration can be achieved with settings like this (in `BOOT_Conf.h`):

```
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_NORMAL

#define SBTL_BOOTLOADER_START_ADDR       0x00000000
#define SBTL_BOOTLOADER_SECTOR_SIZE          0x1000
#define SBTL_BOOTLOADER_SECTOR_COUNT           0x20

#define SBTL_FIRMWARE_START_ADDR         0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE            0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT             0x20

#define SBTL_FIRMWARE_START_ADDR1        0x08000000
#define SBTL_FIRMWARE_SECTOR_SIZE1           0x4000
#define SBTL_FIRMWARE_SECTOR_COUNT1            0x40
```


In this example there is a 256 KB flash at address `0x0` and another 1 MB flash at `0x08000000`.

2.2.1.5 Configuration with dual bank flash

If flash bank swapping is supported by the hardware, the following configuration can be used for a reliable bootloader update mechanism (for details about this mode, see section *Bootloader update in bank swap mode*):



The bootloader in flash bank 1 is the active bootloader, while the bootloader region in flash bank 2 is only active during the update process.

This configuration can be achieved with settings like this (in `BOOT_Conf.h`):

```
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_BANKSWAP

// Definition of the flash area containing the active bootloader
#define SBTL_BOOTLOADER_SOURCE_START_ADDR       0x00000000
#define SBTL_BOOTLOADER_SOURCE_SECTOR_SIZE          0x1000
#define SBTL_BOOTLOADER_SOURCE_SECTOR_COUNT           0x20

// Definition of the flash area used during bootloader updates
#define SBTL_BOOTLOADER_TARGET_START_ADDR       0x00100000
#define SBTL_BOOTLOADER_TARGET_SECTOR_SIZE          0x1000
#define SBTL_BOOTLOADER_TARGET_SECTOR_COUNT           0x20

#define SBTL_FIRMWARE_START_ADDR                0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE                   0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT                   0x1e0
```


2.3 Setting up the bootloader project

Basic familiarity with the tools required for developing embedded software applications, such as compiler, linker and project manager / build system is assumed. This includes adding source and header files to the project, setting the include search path, and so on.

The starting point is a running, empty project (“Hello-World” application) for the target device. It should contain code to initialize any hardware required for bootloader operation, for example clocks, PLLs, RAM and flash memory access. The linker must be directed to place the code into the configured address range for the bootloader, see *SBTL_BOOTLOADER_START_ADDR* and associated macros. All necessary emBoot-Secure source and configuration files can then be added to the project, as described in the next sections.

Add emBoot-Secure files

All source files from the following folders should be added to the project.

Folder	Description
`BOOT`	emBoot-Secure library core components
`SEGGER`	Supportive code shared by SEGGER’s products
`CRYPTO`	Cryptographic library for signature validation and optional update package decryption
`COMPRESS-ToGo`	Code for optional decompression of update package

The above folders must also added to the include search path of the project. Not all of these files (or functions therein) may be needed in an actual bootloader project, but the linker will usually drop everything which is not referenced. The options required for this feature to work depend on the compiler and the linker, and can usually be found in using the keyword “function sections”.

The `Config` directory contains files which are designed to be customized and are provided as templates only. It is recommended to create a copy of them in the directory of the user-specific bootloader code. This prevents the changes from being overwritten when a newer version of the emBoot-Secure library is imported. These files are:

File	Description
`Config/BOOT_Conf.h`	Configures flash memory layout and boot loader behavior.
`Config/BOOT_ConfigIO.c`	Contains functions for debug output.
`Config/CRYPTO_Conf.h`	Configuration for cryptographic library. See *Configuring the CRYPTO library*.
`BSP/<BoardName>/*`	Contains board-specific code for the bootloader, especially for reading memory areas and flash programming. The central file is `BOOT_Config_<board>.c`
`Application/*.c`	Contains example bootloader `main()` implementations.

Customization of theses files is described in the following sections.

2.4 Configuring the bootloader

The file BOOT_Conf.h contains the configuration specific to the user's project and needs to be modified as described in the following sections.

2.4.1 Partitioning the non-volatile memory for emBoot-Secure

The first step in the bootloader creation process is the partitioning of the non-volatile memory to allocate memory for the bootloader and the application firmware. In this section, an example configuration for a device with 2 MB internal flash memory is demonstrated. The flash memory is organized into sectors of 4096 bytes (4 KB). Erasing of flash memory can only be done at the sector scale.

To provide ample room for during development of the bootloader, especially for non-optimized debug builds, 128 KB of flash memory are assigned to the bootloader while the remaining flash memory is available for the application firmware. The bootloader must be placed at the beginning of the flash memory, since it contains the vector table which is read by the microcontroller after a reset.

The linker needs to know about the flash memory layout in order to place the application firmware code at the appropriate addresses, therefore separate linker scripts are needed for the bootloader and the application firmware. A suitable script for the SEGGER Linker is shown below:

```
// Linker_Bootloader.icf

define region FLASH1 = [from 0x00000000 size 0x00200000];

#define BOOTLOADER_SIZE 128k
// HEADER_SIZE must match value SBTL_FIRMWARE_HEADER_SIZE in Boot_ConfDefaults.h.
#define HEADER_SIZE     64

define region BOOTLOADER  = [from 0x00000000 size BOOTLOADER_SIZE];
define region HEADER      = [from BOOTLOADER_SIZE size HEADER_SIZE];
define region APP         = FLASH - BOOTLOADER - HEADER;

// snipped: definitions of blocks and sections

//
// FLASH Placement
//
place at start of BOOTLOADER
    { block vectors };  // Vector table section
place in BOOTLOADER with minimum size order
    {
      block tdata_load, // Thread-local-storage load image
      block exidx,      // ARM exception unwinding block
      block ctors,      // Constructors block
      block dtors,      // Destructors block
      readonly,         // Catch-all for readonly data
      readexec          // Catch-all for (readonly) executable code
    };
```

The bootloader also needs to know about the flash layout; it is configured in BOOT_Conf.h as follows:

```
#define SBTL_FIRMWARE_START_ADDR      0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE         0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT         0x1e0
```

If the bootloader should be able to also update itself, the bootloader area must be configured in BOOT_Conf.h, too:

```
#define SBTL_BOOTLOADER_TARGET_START_ADDR    0x00000000
```



```
#define SBTL_BOOTLOADER_TARGET_SECTOR_SIZE       0x1000
#define SBTL_BOOTLOADER_TARGET_SECTOR_COUNT        0x20
```

2.4.1.1 Scattered area

The application firmware can be distributed over multiple flash memories. For example a device with three flash memories:

1. Range `0x00000000 - 0x0007FFFF`
2. Range `0x80000000 - 0x800FFFFF`
3. Range `0x81000000 - 0x810FFFFF`

```
#define SBTL_FIRMWARE_START_ADDR      0x00020000
#define SBTL_FIRMWARE_SECTOR_SIZE         0x1000
#define SBTL_FIRMWARE_SECTOR_COUNT          0x60

#define SBTL_FIRMWARE_START_ADDR1     0x80000000
#define SBTL_FIRMWARE_SECTOR_SIZE1        0x4000
#define SBTL_FIRMWARE_SECTOR_COUNT1         0x40

#define SBTL_FIRMWARE_START_ADDR2     0x81000000
#define SBTL_FIRMWARE_SECTOR_SIZE2        0x4000
#define SBTL_FIRMWARE_SECTOR_COUNT2         0x40
```

If a flash memory contains areas with different erasable block sizes, it can be configured in the same way.

2.4.2 Customizing the behavior of the bootloader

Several aspects of the behavior of the bootloader need configuration in `BOOT_Conf.h`. Explicit configuration is only needed when the settings differ from the defaults defined in `BOOT_ConfDefaults.h`.

The configuration of the vendor and product names is needed by the bootloader for the derivation of a 32 bit product id number, which is used to ensure that only compatible firmware updates are installed. The product id can be obtained by calling `SBTL_GetProductID()`.

```
#define SBTL_VENDOR_NAME          "Segger"
#define SBTL_PRODUCT_NAME         "Test_Bootloader"
```

For this example, the number of areas in which the bootloader can search for updates is set to two.

```
#define SBTL_NUM_AREAS_UPDATE             2
```

Additionally, areas containing update packages with failsafe rescue firmware can be configured if available. If they are used, the bootloader has to determine whether the currently installed firmware needs to be replaced by a failsafe version, for example by reading out a push button which can be pressed by the end user during the boot process. The rescue update package can be queried for using the `SBTL_FindRescueFirmware()`.

```
#define SBTL_NUM_AREAS_FAILSAFE           0
```

By default, the emBoot-Secure library will only consider update packages for installation which have a version number higher than the firmware which is currently installed. The same is true for bootloader update packages. This provides rollback protection, i.e., it prevents the installation of older, known-vulnerable versions. If rollback protection is not desired, downgrades can be allowed using the respective flags for bootloader and application firmware update packages.

```
#define SBTL_FIRMWARE_ALLOW_DOWNGRADE             0
#define SBTL_BOOTLOADER_ALLOW_DOWNGRADE           0
```


Two different bootloader update modes can be enabled, depending on the capabilities of the target hardware.

- SBTL_BOOTLOADER_UPDATE_MODE_NONE

Update of the bootlader is prohibited.

- SBTL_BOOTLOADER_UPDATE_MODE_NORMAL

The bootloader can be updated. During update, the bootloader overwrites itself. This carries the risk of device bricking if the update process is interrupted. This should be used with care.

- SBTL_BOOTLOADER_UPDATE_MODE_BANKSWAP

On target hardware with two swappable flash banks, the bootloader can update itself in a two-step process, which can recover from interruptions or transient failures.

For this example, the normal update mode is configured:

```
#define SBTL_BOOTLOADER_UPDATE_MODE    SBTL_BOOTLOADER_UPDATE_MODE_NORMAL
```

For verification of the firmware integrity and authenticity, either RSA or ECDSA based signature algorithms can be used. Several hash algorithms like SHA1, SHA256, SHA384 and SHA512 are available. A list of relevant flags is provided in the section *Configuration switches related to algorithms*. Section *Choosing a signature scheme and its parameters* provides a more in-depth overview over the available choices and their trade-offs. In this example, RSA-based signatures with SHA256 hashes are chosen.

```
#define SBTL_ALGO_SIGN_RSA_WITH_SHA256          1
#define SBTL_ALGO_SIGN_RSA_WITH_SHA512          0
#define SBTL_ALGO_SIGN_RSA_WITH_SHA1            0
#define SBTL_ALGO_SIGN_ECDSA_WITH_SHA256        0
#define SBTL_ALGO_SIGN_ECDSA_WITH_SHA384        0
#define SBTL_ALGO_SIGN_ECDSA_WITH_SHA512        0
#define SBTL_ALGO_SIGN_ECDSA_WITH_SHA1          0
```

Update packages can optionally be compressed using SEGGER's SMASH-2 compression algorithm. If the application firmware contains a lot of ARM Thumb-2 instructions, the optimized SMASH-2-T2 variant can be used. Enabling compression can lead to smaller update packages, but increases code size of the bootloader. See section *Resource use and performance* for more information. Enabling this flag will integrate the code required to process compressed update packages, but it does not make compression mandatory. The bootloader can still process and install uncompressed update packages.

```
#define SBTL_ALGO_COMP_SMASH2                   1
#define SBTL_ALGO_COMP_SMASH2_T2                0
```

Finally, update packages can be encrypted to prevent them from being inspected and reverse-engineered. The emBoot-Secure library provides the symmetric AES encryption algorithm in Galois/Counter mode. Encryption requires the generation of an AES key, which needs to be converted to a C file which can be added to the bootloader project, see section *Adding keys to the bootloader*. Enabling this flag will integrate the code required to process encrypted update packages, but it does not make encryption mandatory. The bootloader can still process and install unencrypted update packages.

```
#define SBTL_ALGO_ENC_AES_GCM                   1
```


2.5 Adding hardware-dependent configuration

A bootloader implementation always needs hardware- and/or board-specific support functions, which act as callbacks. They are used both to obtain information as well as to modify flash memory. These functions are stored in a file called `BOOT_Config_<TargetName>.c`

Sample configurations for popular evaluation boards are supplied with the emBoot-Secure shipping package (in the folders `BSP/<BoardName>`) or can be provided by SEGGER on request.

If there is no board configuration file for the selected target device, a configuration file for a similar hardware should be taken and modified to match the hardware.

Required support functions are described in the next sections.

2.5.1 Configuring debugging output

While developing and testing a bootloader, we recommend to use the DEBUG configuration of emBoot-Secure. This is enabled by setting the preprocessor symbol `DEBUG` (or `SBTL_DEBUG`) to 1. The DEBUG configuration contains additional run-time checks and generates debug output messages which are very useful to identify problems that may occur during development. In case of a fatal problem (e.g. an invalid configuration), the program will end up in the function *`SBTL_Panic`*.

Once the application is running correctly, `DEBUG` can be set to 0. To later compile a release configuration, which has a smaller code footprint, the preprocessor symbol `DEBUG` (or `SBTL_DEBUG`) can simply be set to 0.

The file `BOOT_ConfigIO.c` found in the folder `Config` must be added to the project and configured to match the message output method used by the debugging tools (semihosting, RTT, etc.). If possible, SEGGER's Real-Time-Transfer (RTT) should be used. The output is shown in the output pane of the Embedded Studio IDE or can be viewed with the J-Link RTT Viewer. Note, however, that when the bootloader transfers control to the firmware application, which uses its own buffer for RTT, those messages will not be captured by the RTT interface. Therefore, for observing the application's debug output during development, it has to be started directly by the debugger.

Debug output can be added to custom bootloader code using the `SBTL_LOG` macro, which invokes the `SBTL_Logf()` function when debug output is enabled. Otherwise, it does not produce any code. The macro forwards messages with `printf`-compatible format specifiers and a variable number of arguments to `SBTL_Logf()`. Due to limitations in the C preprocessors, two parentheses have to be used around its arguments. A newline character at the end of the string is not needed.

```
SBTL_LOG(("Bootloader version %d starting", SBTL_GetBootloaderVersion()));
```

2.5.2 Support functions for accessing firmware images

The emBoot-Secure library needs to access different types of firmware (and possibly, bootloader) images while determining which actions to take, like installing an update or starting the application. The different accessible images are called "areas" and are addressed via indices. In order to read data from an area, the function `SBTL_ReadArea()` is used. When the emBoot-Secure library requests data from an area, it will always start reading the area from the beginning. It will always read the data sequentially and when it has either read enough data from an area or read until the end of an area, it will call `SBTL_CloseArea()`. These rules allow the mechanism to be applied to different kinds of storage mechanisms.

In this example, the two update areas which have been configured will be implemented as files on the NAND flash chip on the SEGGER emPower evaluation board. This allows the application firmware to place up to two files with firmware in the filesystem.

```
// Static variable to hold the handle for the currently open file.
static FS_FILE* AreaFile = 0;
```



```
int SBTL_ReadArea(unsigned Area, U32 Offset, unsigned BuffSize, U8 *pBuff) {
  FS_FILE*          FileHandle;
  unsigned long     nBytesRead;
  I16               ErrorCode;
  const char*       Filename = 0;

  if (Area == SBTL_MEM_AREA_FIRMWARE) {  ❶
    memcpy(pBuff, (U8*)(SBTL_FIRMWARE_START_ADDR+Offset), BuffSize);
    return BuffSize;
  }

  if (Offset == 0) {  ❷
    switch (Area) {

    case SBTL_MEM_AREA_UPDATE(0):
      Filename = "update_1.upd";
      break;

    case SBTL_MEM_AREA_UPDATE(1):
      Filename = "update_2.upd";
      break;

    default:
      return -1;
    }

    FileHandle = _OpenFirmwareFileForReading(Filename); ❸
    if (FileHandle == 0) {
      return -1;
    }
    AreaFile = FileHandle;
  }

  // At this point, areaFile is a handle to an open file.
  // Read data as requested.
  nBytesRead = FS_Read(AreaFile, pBuff, BuffSize); ❹
  if (nBytesRead != BuffSize) {

    ErrorCode = FS_FError(AreaFile);
    if (ErrorCode == FS_ERRCODE_EOF) {
      // Reached EOF, that is not an error.
      return nBytesRead;
    }

    // Other error
    return -1;
  }

  return nBytesRead;
}
```

❶ Reading from the currently installed firmware

Data was requested from the currently installed firmware image. It can simply be copied into the caller's buffer using `memcpy`.

❷ Caller starts reading from a new area

The `Offset` parameter is zero, which means the caller starts reading from a new area. The switch statement is used to determine the filename corresponding to the area.

❸ Attempt to open the file corresponding to the area

An attempt to open the file for reading is made. Details of file opening, like checking for available storage volumes, has been moved into the function `_OpenFirmwareFileForReading` which is not shown here. If opening of the file fails, an error code is returned. This

indicates to the caller that the region cannot be read. If opening succeeds, the file handle is stored in a global variable to be available for the next call.

❹ Read data from the file

Data is read from the file. If less data than requested was read, the code checks for filesystem errors. In case of end of file (`EOF`), the number of bytes read is returned. The caller must not assume that the end of the file has been reached if less data than requested has been returned. The caller may call the function again to read more data, in which case the function will return zero, indicating to the caller that the end of the file has been reached. In case of any other error, `-1` is returned.

The function `SBTL_CloseArea()` is called by the emBoot-Secure library after it has finished reading from an area. In this example, only the areas backed by files need to be closed. The `Area` parameter is therefore not needed in this case. Only the `AreaFile` variable is checked for a file handle, which is then closed.

```
void SBTL_CloseArea(unsigned Area) {

  // Mark the parameter Area as unused to avoid
  // compiler warnings.
  (void)Area;

  // If a file is open, close it.
  if (AreaFile != 0) {
    FS_FClose(AreaFile);
    AreaFile = 0;
  }
}
```

2.5.3 Writing to flash memory

When the emBoot-Secure library is instructed to update application firmware images or bootloader images in flash memory, it needs hardware specific functions which perform the writing operations. Depending on the agreements made during purchase of the emBoot-Secure license, a flash memory driver for a specific controller is provided alongside the emBoot-Secure core library. Many microcontrollers do not support writing to flash memory while code is executed from flash memory at the same time. Therefore, the functions for writing to flash memory have to be placed in RAM.

This chapter provides hints for placing functions in RAM, and discusses the inner workings of a flash memory driver at a high level.

2.5.3.1 Placing functions in RAM

The details about how to place a function in RAM depend on the compiler. Usually, the compiler has to be instructed to mark the function for placement in a certain section, which has been configured for placement in RAM in the linker script. The linker will then place the function in a section which is copied from flash memory into RAM during startup. Not all compiler/linker combinations perform the initial copying process from flash into RAM on their own and a custom implementation may be needed. For details, the manuals of the compiler and linker need to be consulted. This section explains how placing a function in RAM works with the SEGGER compiler, the GNU compiler, the IAR compilers for ARM and Renesas platforms, as well as the Renesas compiler in C99 mode.

In the emBoot-Secure library, the instruction for placing a function into RAM is implemented by the `SBTL_RAM_FUNCTION` macro. This macro has to be placed in front of the definition of each function which needs to be placed in RAM. Unfortunately, some compilers do not allow changing the placement options for a single function with the required granularity, and therefore the macro may affect all function definitions which follow it. Therefore, source code files should be divided into a section of functions with regular placement in flash, while the functions with RAM placement, each prefixed by the macro, follow below them.

The SBTL_RAM_FUNCTION implementations discussed below are activated by default in the file BOOT_ConfDefaults.h. If another compiler is used, the macro has to be appropriately defined in the file BOOT_Conf.h.

SEGGER Compiler, GNU Compiler, IAR (ARM and Renesas) compiler

For these compilers, the RAM placement is achieved using a function attribute which contains the name of the section and an instruction forbidding inlining of the function. The name of the section depends on the linker script. .fast is the default name in used in the linker scripts provided for the SEGGER linker and a common name used in scripts for other linkers.

```
#define SBTL_RAM_FUNCTION __attribute__ ((section (".fast"), noinline))
```

Renesas compiler in C99 mode

In C99 mode, the Renesas compiler enables RAM placement through a pragma, which applies to all functions which appear after the pragma. Assuming that the linker script is configured to place the section ram_text in RAM, the pragma can be implemented as follows:

```
#define SBTL_MACRO_TO_STRING(x)      #x
#define SBTL_RAM_FUNCTION _Pragma(SBTL_MACRO_TO_STRING(section P ram_text))
```

2.5.3.2 Flash memory driver functions

The SBTL_WriteFlash() function is called by the emBoot-Secure library during an update to write the new firmware or bootloader code into flash memory. The process of writing to flash memory is highly specific to the employed microcontroller model. Depending on the agreements made during purchase of the emBoot-Secure license, a driver package for a specific controller is provided alongside the emBoot-Secure core library. Since the driver package is usually acquired by customers, this guide does not discuss the details of flash memory programming for a specific controller, but explains the high level details.

```
SBTL_RAM_FUNCTION ❶
int SBTL_WriteFlash(PTR_ADDR FlashAddr, unsigned Size, U8 *pData) {
  unsigned int SectorNumber;
  int ReturnCode;

  SectorNumber = FlashAddr / SBTL_FIRMWARE_SECTOR_SIZE; ❷
  ReturnCode = 0;

  if (!PrepareFlashWriting()) { ❸
    return -1;
  }

  if (!EraseSector(SectorNumber)) { ❹
    ReturnCode = -1;
    goto end;
  }

  if (!WriteFlash(FlashAddr, Size, pData) { ❺
    ReturnCode = -1;
  }

end:

  if (!RestoreFromFlashWriting()) { ❻
    ReturnCode = -1;
  }
  return ReturnCode;
}
```


❶ Placing the function in RAM

The `SBTL_RAM_FUNCTION` macro is used to place the function in RAM.

❷ Determining the sector number

The `SBTL_WriteFlash()` function has to take care of both flash erasure and programming. The emBoot-Secure library will examine the flash memory layout defined in `BOOT_Conf.h`, see *Configuration switches for flash memory layout* for details. It will make a separate call to `SBTL_WriteFlash()` for each sector of flash memory which needs to be written. Therefore, the function can determine the sector number from the `FlashAddr` parameter, erase it, and then write the buffer pointed to by `pData` into that sector.

❸ Prepare for flash programming

On the MK66F18 processor, which drives the previously mentioned SEGGER emPower evaluation board, clock speed and supply voltages have to be in certain ranges for flash memory programming. Therefore, in this step, the clock speed and the supply voltages are set to the required values. Also, if interrupts are enabled, they need to be disabled to avoid jumps into flash memory due to interrupts while flashing is in progress.

❹ Erase flash memory

The flash sector is erased. If erasure fails, programming is skipped but the system still has to be returned into the previous state, therefore a jump to the `end` label is performed.

❺ Write to flash memory

The buffer is written to flash memory.

❻ Restore system state

The system is restored to the clock speed, supply voltage and interrupt state as it was at the beginning of the function.

2.5.4 Configuring the CRYPTO library

The emBoot-Secure library handles most of the configuration required for the crypto library, emCrypt, internally. Depending on the signature algorithm and hash function configured in `BOOT_Conf.h`, the necessary functions of the library are activated. Remaining user-specific runtime configuration is performed in the callback function `CRYPTO_X_Config()`, which is invoked when `SBTL_Init()` is called. Any remaining user-specific compile-time configuration is performed in the file `CRYPTO_Conf.h`.

2.5.4.1 CRYPTO_X_Config: Exponentiation and hardware support

For signature checking with RSA or ECDSA, a modular exponentiation algorithm has to be configured in the `CRYPTO_X_Config()` function. The default implementation of `CRYPTO_X_Config()` provided in the board-specific `BOOT_Config_<board>.c` file selects a reasonably fast modular exponentiation algorithm with low RAM requirements.

If the microcontroller or its periphery offer cryptographic accelerators, for example a hash calculator implemented in hardware, that can also be activated in this function.

More information can be found in the section *`CRYPTO_X_Config`*.

2.5.4.2 CRYPTO_Conf.h: Software optimization

For many cryptographic algorithms, a time-memory tradeoff can be made: An implementation can either be small (with respect to its code size) and more computation-intensive, or use more space to store pre-computed values that speed up computations.

Algorithms are selected based on the *optimization goal* configuration in `CRYPTO_Conf.h`. An optimization goal specifies the application developer's preference in the tradeoff between code size and performance.

By default, emCrypt selects a *balanced* configuration that provides reasonable performance at a reasonable code size for most applications. It is recommended to try the balanced default configuration first and assess its performance. If more performance is needed or the code size needs to be reduced, the configuration can be adapted as described below.

Optimization goals can be configured at two levels:

- *Algorithm-specific* optimization goals specify the preferred goal for a particular algorithm.
- The *global* optimization goal sets the general preference. It is applied to all algorithms that do not have an algorithm-specific optimization goal set (fallback).

There are seven optimization goals to choose from:

Optimization goal	Description
`SEGGER_OPT_GOAL_SIZE_MIN`	Lowest resource requirements without performance restriction. Should be used on constrained systems only.
`SEGGER_OPT_GOAL_SIZE_SMALL`	Favor low resource requirements at reasonable performance.
`SEGGER_OPT_GOAL_SIZE`	Favor lower resource requirements at slightly lower performance.
`SEGGER_OPT_GOAL_BALANCED`	Balance between performance and resource requirements. The common case default.
`SEGGER_OPT_GOAL_SPEED`	Favor performance at slightly higher resource requirements.
`SEGGER_OPT_GOAL_SPEED_HIGH`	High performance with reasonable resource requirements.
`SEGGER_OPT_GOAL_SPEED_MAX`	Maximum performance without resource restriction. May fit only larger systems.

All optimization goal settings default to `SEGGER_OPT_GOAL_BALANCED`.

An optimization goal configuration can be changed by defining a macro in the file `CRYPTO_Conf.h`. The global optimization goal can be set like this:

```
#define CRYPTO_OPT_GOAL_GLOBAL  SEGGER_OPT_GOAL_SPEED
```

The algorithm-specific optimization goals that are relevant for emBoot-Secure can be modified as follows:

```
#define CRYPTO_OPT_GOAL_SHA1    SEGGER_OPT_GOAL_SPEED
#define CRYPTO_OPT_GOAL_SHA256  SEGGER_OPT_GOAL_SPEED
#define CRYPTO_OPT_GOAL_SHA512  SEGGER_OPT_GOAL_SPEED
#define CRYPTO_OPT_GOAL_AES     SEGGER_OPT_GOAL_SPEED
```

The SHA384 hash function is also configured through the `CRYPTO_OPT_GOAL_SHA512` optimization goal. The SHA384 hash is a truncated SHA512 hash and uses the same implementation internally.

While the performance of the hash functions affects the boot time of the target device, the performance of the encryption algorithm only affects the time required to install a firmware update.

2.6 Version numbers

The emBoot-Secure library uses 32 bit unsigned integers to represent version numbers. If the 32 bit version number of an installed firmware image is lower than that of an update package, the update package will be installed. Internally, the bootloader and the application firmware can use their own way to display the version number, for example by interpreting the 32 bit integer as a number of the form `M.mm.rrr`.

The version number of the application firmware is stored inside the header which is generated during the creation of signed firmware images or update packages. The bootloader can read out this version number by requesting the `FirmwareInfo` structure for a firmware area using the `SBTL_GetFirmwareInfo()` function.

Bootloader update packages also contain the version number of the bootloader, which is independent of the firmware application's version number. When the bootloader is installed, however, the header is lost since the bootloader's memory region has to start with the vector table. While the bootloader is running, the the emBoot-Secure library needs to know the version number of the currently running bootloader, in order to compare it with version numbers of bootloader update packages. Since it can not read the version number from a header field, it uses the `SBTL_GetBootloaderVersion()` function instead to determine it.

For more information about how to embed version information into both the bootloader and the application firmware, see *Specifying version numbers*.

2.7 Adding keys to the bootloader

This section explains how to create keys for use with the emBoot-Secure library using an example. For a detailed background about which key types, sizes and algorithms to use, see *Passwords, keys, and key files*. In this example, keys are generated using the key generation tool, a command-line program that ships with emBoot-Secure. Alternatively, the SEGGER Signature Server can be used for key generation, see *Initialization of the Signature Server*.

Both ECDSA and RSA keys are supported. In this example, a 2048 bit RSA key will be used. To generate a signing key pair, the *key generation tool* is used. Using the following command, a 2048 bit RSA key is generated. The public part of the signing key is stored in the file `KeyFile1.pub`. Since the private key has to remain secret, the **`-p`** parameter is specified to enable encryption of the key. The key generation tool asks for a password to encrypt the key before writing it to the file `KeyFile1.prv`.

```
KeyTool gen-rsa -s 2048 --out KeyFile1 -p
```

The private key has to be carefully guarded (see *Security recommendations for keys and key files*). In order to provide the public key to the bootloader, it is converted to a C file using the key generation tool:

```
KeyTool rsa2c --in KeyFile1.pub --out signkey1.c
```

The resulting C file contains an implementation of `SBTL_GetRSAPublicKey()`, which returns the key data. This additional source code file has to be added to the bootloader project and linked into the final bootloader image.

If the ECDSA signature scheme is used instead of RSA, the key generation and conversion process is similar. The resulting C file contains a function `SBTL_GetECPublicKey()` in that case.

For encryption of update packages with AES, a key is required as well. It is generated using the key generation tool. Again, the parameter **`-p`** is specified to enforce encrypted storage of the key in the file. The key generation tool writes the encrypted key to the file `EncKeyFile1.sec`.

```
KeyTool gen-aes --out EncKeyFile -p
```

To include this AES key into the bootloader, it is also converted to a C file, this time implementing the `SBTL_GetAESKey()` function:

```
KeyTool aes2c --in EncKeyFile.sec --out enckey1.c
```

The file `enckey1.c` now contains the AES encryption key as plain text. This additional source code file has to be added to the bootloader project and linked into the final bootloader image as well.

2.8 Starting application firmware

When the bootloader has verified the authenticity of the application firmware image, it can launch the application firmware. Before the launch, the bootloader has to take a few preparatory steps.

Deactivating hardware

The bootloader may have activated a hardware cryptographic accelerator module or external flash memory to load the update package. If these are not needed by the application firmware, it should turn these components off again.

Clock and supply voltage configuration

The startup code of the bootloader may have reconfigured the clock and supply voltage settings. If the application firmware contains startup code which also modifies these settings, and expects the processor to be in the default state after reset, the bootloader has to revert these settings. The exact requirements depend on the way in which the clock settings and supply voltages were changed. On some processors, performing the activation steps again in the same order may not cause faults. In that case, the bootloader can leave the settings as they are. Alternatively, the application firmware's startup code can be modified to leave the clock and supply voltage settings as they were set up by the bootloader.

Turning off interrupts

The processor handles interrupts using the vector table. When the bootloader launches the application, the bootloader's vector table is still active and any interrupt would cause a jump back into bootloader code. Therefore, the bootloader must disable interrupts, giving the application firmware's startup code a chance to activate it's own vector table.

Jumping to the entry point

When the bootloader's main() function determines whether authentic application firmware is installed, it uses the `SBTL_ScanFirmware()` function, which stores information about the firmware image into a `SBTL_FIRMWARE_INFO` structure. This information also contains the address of the application firmware's entry point. The bootloader can jump to this address using the function pointer stored in the `SBTL_FIRMWARE_INFO` structure. An abridged code sample follows, see *Bootloader update in normal mode* for details about the omitted parts.

```
SBTL_FIRMWARE_INFO FirmwareInfo;
r = SBTL_ScanFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer);
if (r != 0) { /* Error handling */ }
__disable_irq();

FirmwareInfo.EntryPoint.Jump();
```


2.9 Sample bootloader implementations

This section describes how to combine the high level functions provided by the emBoot-Secure library API to create a customized bootloader. Three example bootloader implementations, which ship with emBoot, are explained in detail to understand how the bootloader is implemented. It is assumed that the required callback functions which were explained in the previous sections have been implemented and that the emBoot-Secure library has been configured by adjusting the `BOOT_Conf.h` file.

2.9.1 Bootloader update in normal mode

This section describes the simplest bootloader implementation which can update itself using normal bootloader update mode. The code is shipped with emBoot-Secure in the file `Application/BL_NormalMode.c`. The internal flash memory could be configured as described in *Simple configuration*.

```
/*********************************************************************
*                    (c) SEGGER Microcontroller GmbH                 *
*                         The Embedded Experts                       *
**********************************************************************
*                                                                    *
*       (c) 2022 - 2026     SEGGER Microcontroller GmbH              *
*                                                                    *
*       www.segger.com     Support: www.segger.com/ticket            *
*                                                                    *
**********************************************************************
*                                                                    *
*       emBoot-Secure * Secure bootloader for embedded applications  *
*                                                                    *
*       Please note: Knowledge of this file may under no             *
*       circumstances be used to write a similar product.            *
*       Thank you for your fairness !                                *
*                                                                    *
**********************************************************************
*                                                                    *
*       emBoot version: Internal                                     *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
Purpose     : Sample bootloader application for bootloader updates in
              normal mode.
--------------------------- END-OF-HEADER ----------------------------
*/

#include "BOOT.h"
#include "cmsis_compiler.h"

/*********************************************************************
*
*       Static variables
*
**********************************************************************
*/

// Work memory for SBTL_Verify and SBTL_Unwrap
static SBTL_WORK_MEMORY _workMemory; ❶

// Buffer for reading sectors to memory during flashing
static SBTL_FLASH_BUFF  _sectorBuffer;

/*********************************************************************
*
*       main()
*
* Function description
```



```
*   Application entry point
*/
int main(void) {
  SBTL_FIRMWARE_INFO  FirmwareInfo;
  int                 r;

  SBTL_Init();

  SBTL_LOG(("Bootloader version %d starting...", SBTL_GetBootloaderVersion()));

  // Look for firmware updates, and also check whether the
  // currently installed firmware can be verified.
  r = SBTL_ScanFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer); ❷
  if (r != 0) {
    SBTL_Panic("Scanning for valid firmware failed.");
  }

  if (FirmwareInfo.Area == SBTL_MEM_AREA_FIRMWARE) { ❸
    // Boot into already installed firmware

    // If applicable, de-initialize any hardware which is not needed
    // by the application firmware and also reset the clocks and
    // supply voltages to safe settings.

    // Disable interrupts
    __disable_irq();

    // Jump to firmware entry point
    FirmwareInfo.EntryPoint.Jump();
  }

  if ( (FirmwareInfo.IsBootLoaderUpdate == 0 )
    && SBTL_MEM_AREA_IS_UPDATE(FirmwareInfo.Area) ) { ❹

    // Install normal firmware update
    r = SBTL_UpdateFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer);
    if (r != 0) {
      // Error updating the firmware
      SBTL_LOG(("Error updating the firmware, code: %d", r));
      SBTL_Panic("Error updating the firmware.");
    }

    // Reboot and start firmware
    SBTL_ResetHardware();
  }

  if ( (FirmwareInfo.IsBootLoaderUpdate == 1 )
    && SBTL_MEM_AREA_IS_UPDATE(FirmwareInfo.Area) ) { ❺

    // Install bootloader update

    // Prepare for the bootloader update

  r = SBTL_PrepareBootloaderUpdate(&FirmwareInfo, &_workMemory, &_sectorBuffer);
    if (r < 0) {
      SBTL_LOG(("Error updating the bootloader, code: %d", r));
      SBTL_Panic("Error updating the bootloader.");
    }

    // Perform the bootloader update.
    // This function will never return.
    SBTL_PerformBootloaderUpdate(&_sectorBuffer);
  }

  // This place is never reached.
  SBTL_Panic("Bootloader logic failed.");
}
```


❶ Static working memory

Allocates static working memory for the unpacking of the update package and signature validation, as well as a buffer to hold parts of the unpacked update package during flash programming.

❷ Scan for update packages

`SBTL_ScanFirmware()` is called to determine which action the bootloader should take. It scans the firmware update areas for the presence of a valid update package. If a valid update package is found, it's version number is compared against the version number of the currently installed firmware. If that is the case, information about that update package is copied to the `FirmwareInfo` structure. If no suitable update package is available, the installed firmware is verified and header of the installed firmware is copied into the `FirmwareInfo` structure. If neither a valid update package can be found nor a valid installed application firmware image, an error is returned.

❸ Launch installed application firmware

If the `FirmwareInfo.Area` contains the area index of the currently installed application firmware image, that image is started. If applicable, any hardware which was configured by the bootloader may need to be deinitialized and the clock and supply voltage settings be returned to default safe settings. In any case, interrupts have to be disabled to ensure no interrupt handlers are triggered during the transition, since the vector table of the bootloader is still active. Afterwards, a jump to the firmware entry point is performed using the function pointer in the `FirmwareInfo` structure.

❹ Install application update package

If the `FirmwareInfo` structure points to an area containing an application firmware update, the update process is started. The function `SBTL_UpdateFirmware()` is called to perform the update and afterwards, the microcontroller is reset. After the reset, the bootloader will verify the newly installed application firmware image again and start it.

❺ Install bootloader update package

If the `FirmwareInfo` structure points to a bootloader update package, a two-step update process is performed. First, the bootloader update package is unpacked into RAM. Afterwards, the bootloader image is written into flash memory using the `SBTL_PerformBootloaderUpdate()` function. This function has been placed in RAM, since during the update process, the bootloader code in flash memory is overwritten and can therefore not be executed any more. The function in RAM can continue execution until flashing has finished and then restart the target hardware to boot into the updated bootloader.

2.9.2 Bootloader update in bank swap mode

Bank swap mode is only available on targets with two flash banks which can be swapped, as described in section *Configuration with dual bank flash*. The code for this example is shipped with emBoot-Secure in the file `Application/BL_BankSwapMode.c`. The diagram below shows the steps of the update process. The blue numbered dots indicate the same steps in both the diagram and the source code.

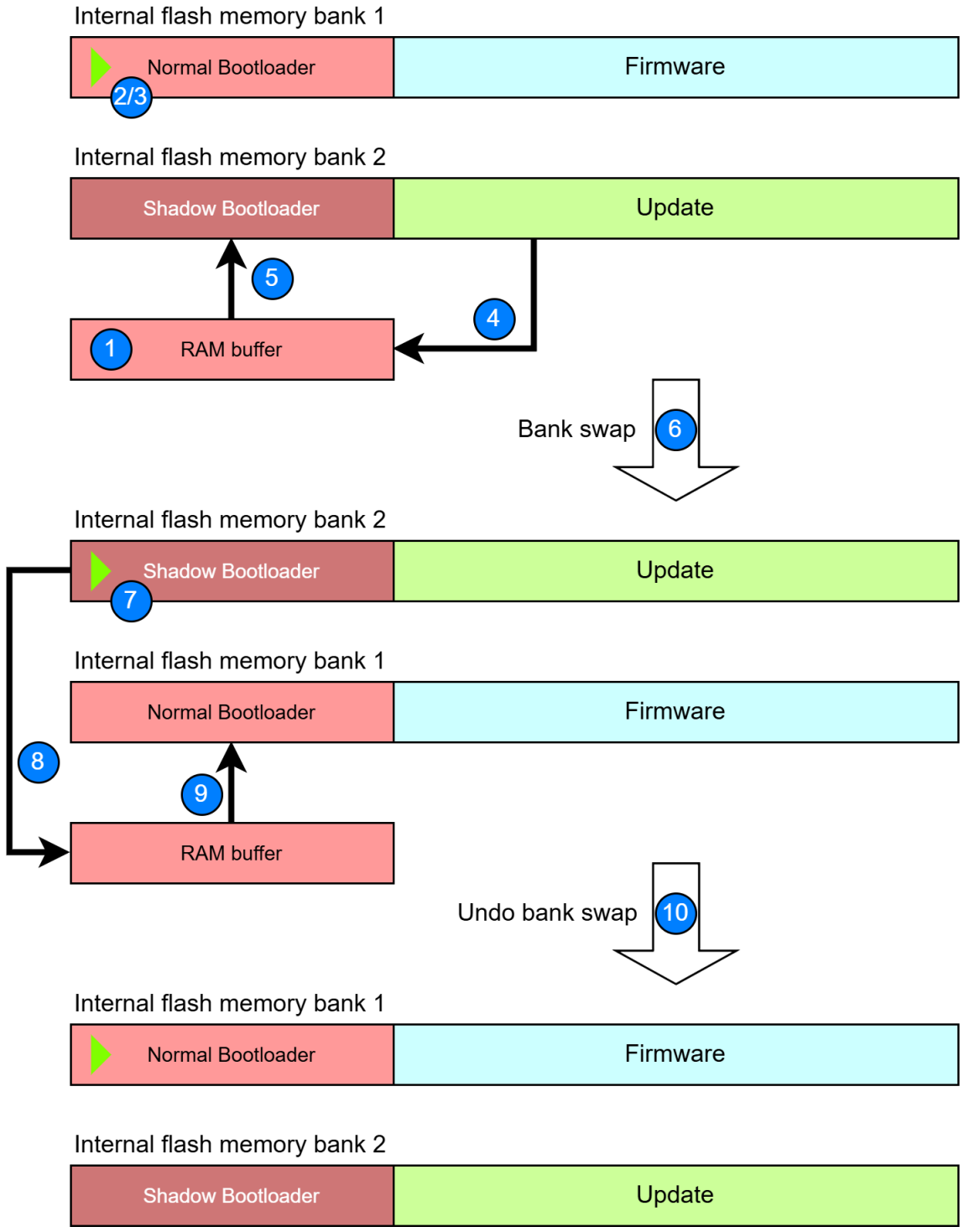


Figure: Bootloader update in bank swap mode

```
/*********************************************************************
*                    (c) SEGGER Microcontroller GmbH                 *
*                        The Embedded Experts                        *
**********************************************************************
*                                                                    *
*       (c) 2022 - 2026     SEGGER Microcontroller GmbH              *
*                                                                    *
*       www.segger.com     Support: www.segger.com/ticket            *
*                                                                    *
**********************************************************************
*                                                                    *
*       emBoot-Secure * Secure bootloader for embedded applications  *
*                                                                    *
*       Please note: Knowledge of this file may under no             *
```



```
*       circumstances be used to write a similar product.             *
*       Thank you for your fairness !                                 *
*                                                                     *
**********************************************************************
*                                                                     *
*       emBoot version: Internal                                      *
*                                                                     *
**********************************************************************
----------------------------------------------------------------------
Purpose     : Sample bootloader application for bootloader updates
              in bank swap mode.
--------------------------- END-OF-HEADER ----------------------------
*/

#include "BOOT.h"
#include "cmsis_compiler.h"

/*********************************************************************
*
*       Static variables
*
**********************************************************************
*/

// Work memory for SBTL_Verify and SBTL_Unwrap
static SBTL_WORK_MEMORY _workMemory;

// Buffer for reading sectors to memory during flashing
static SBTL_FLASH_BUFF  _sectorBuffer;  ❶

/*********************************************************************
*
*       main()
*
* Function description
*   Application entry point
*/
int main(void) {
  SBTL_FIRMWARE_INFO  FirmwareInfo;
  int                 r;

  SBTL_Init();

  SBTL_LOG(("Bootloader version %d starting...", SBTL_GetBootloaderVersion()));

  // Determine bank swap state
  r = SBTL_GetBankSwapState(); ❷
  SBTL_LOG(("Active flash bank: %d", r));

  if (r == 0) { ❸
    // Banks are currently not swapped,
    // this is the "normal bootloader".

    // Look for firmware updates, and also check whether the
    // currently installed firmware can be verified.
    r = SBTL_ScanFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer);
    if (r != 0) {
      SBTL_Panic("Scanning for valid firmware failed.");
    }

    if (FirmwareInfo.Area == SBTL_MEM_AREA_FIRMWARE) {
      // Boot into already installed firmware

      // If applicable, de-initialize any hardware which is not needed
      // by the application firmware and also reset the clocks and
      // supply voltages to safe settings.
```



```
      // Disable interrupts
      __disable_irq();

      // Jump to firmware entry point
      FirmwareInfo.EntryPoint.Jump();
    }

    if ( (FirmwareInfo.IsBootLoaderUpdate == 0 )
      && SBTL_MEM_AREA_IS_UPDATE(FirmwareInfo.Area) ) {
      // Install normal firmware update
      r = SBTL_UpdateFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer);
      if (r != 0) {
        // Error updating the firmware
        SBTL_LOG(("Error updating the firmware, code: %d", r));
        SBTL_Panic("Error updating the firmware.");
      }

      // Reboot and start firmware
      SBTL_ResetHardware();
    }

    if ( (FirmwareInfo.IsBootLoaderUpdate == 1 )
      && SBTL_MEM_AREA_IS_UPDATE(FirmwareInfo.Area) ) {
      // Install the bootloader update

      // Load the bootloader update into RAM.
      r = SBTL_PrepareBootloaderUpdate(&FirmwareInfo, &_workMemory,
                                       &_sectorBuffer); ❹
      if (r != 0) {
        goto errorEnd;
      }

      // Update the shadow bootloader from RAM.
      r = SBTL_PerformBootloaderUpdate(&_sectorBuffer); ❺
      if (r != 0) {
        goto errorEnd;
      }

      // Bootloader update has been written to bank 2.
      // Swap banks to activate bootloader in bank 2.
      r = SBTL_SetBankSwapState(1); ❻
      if (r != 0) {
        goto errorEnd;
      }

      // Banks have been swapped, reboot.
      // Function does not return.
      SBTL_ResetHardware();

errorEnd:
      // Error updating the bootloader
      SBTL_LOG(("Error updating the bootloader, code: %d", r));
      SBTL_Panic("Error updating the bootloader.");
    }

  } else { ❼
    // Banks are currently swapped,
    // this is the "shadow bootloader".

    // Load "shadow bootloader" into RAM
    SBTL_LoadActiveBootloaderIntoRAM(&_sectorBuffer); ❽

    // Overwrite the bootloader on bank 1 with the "shadow bootloader" from RAM
    r = SBTL_PerformBootloaderUpdate(&_sectorBuffer); ❾
    if (r != 0) {
      SBTL_LOG(("Writing bootloader to shadow address failed: %d", r));
```



```
      SBTL_Panic("Writing bootloader to shadow address failed.");
    }

    r = SBTL_SetBankSwapState(0); ❿
    if (r != 0) {
      SBTL_LOG(("Swapping banks after updating shadow bootloader failed:
 %d", r));
      SBTL_Panic("Swapping banks after updating shadow bootloader failed.");
    }

    SBTL_ResetHardware();
  }

  // This place is never reached.
  while (1);
}
```

❶ Static working memory

Allocates static working memory for the unpacking of the update package and signature validation, as well as a buffer to hold parts of the unpacked update package during flash programming.

❷ Determine the bank swap state

The bootloader calls `SBTL_GetBankSwapState()` to determine whether the banks are currently swapped.

❸ Banks are not swapped

The bootloader determined that the banks are not swapped. It can proceed with the same steps as the bootloader in normal update mode (see previous section) to scan for update packages, launch currently installed firmware or update the installed firmware.

❹ Load the bootloader update into RAM

The bootloader has determined that a bootloader update package is available. Just like in normal update mode, it unpacks the entire bootloader update package in RAM.

❺ Update shadow bootloader in flash memory

The `SBTL_PerformBootloaderUpdate()` function is used to write the bootloader update from RAM into the flash memory in flash bank 1, overwriting the normal bootloader.

❻ Bank swap: Activate shadow bootloader

The bootloader instructs the hardware to apply flash bank swapping after the next restart. Afterwards, it performs a hardware reset to launch the bootloader on the other flash bank.

❼ Shadow bootloader is active

The code will be executed again from the top, but this time, the call to `SBTL_GetBankSwapState()` will indicate that the flash banks are swapped. Hence, the first branch of the `if`-clause will be skipped and the update process will be continued in the next step.

❽ Load shadow bootloader into RAM

Since the authenticity of the shadow bootloader, which is currently running, has already been determined during its installation, it can now simply use itself as the source for the update of the normal bootloader. It loads itself into RAM using the `SBTL_LoadActiveBootloaderIntoRAM()` function.

❾ Update from the shadow bootloader

It then overwrites the normal bootloader using the `SBTL_PerformBootloaderUpdate()` function.

⑩ Disable bank swap

The hardware is instructed to disable the bank swap after the next reset. The shadow bootloader's work is done. It resets the hardware to restart into the normal bootloader.

2.9.3 Bootloader with rescue firmware

This section describes how to implement a bootloader which can install a rescue application firmware image in case the currently installed application firmware becomes unusable. This could be due to an incompatibility with the device hardware which is only discovered after installation, or because a severe bug is present which prevents obtaining new update packages using this software version.

In that case, the bootloader can be supplied with a rescue firmware update package, which is only installed under special circumstances. The rescue firmware image may have reduced functionality and focus on making a new update package available to the bootloader. The exact circumstances under which it is installed depend on the device and the manufacturer's precautions. A simple option would be a restore button which is pressed by the user while turning the device on.

In order for the bootloader to be able to find a rescue image, the number of areas containing rescue firmware images has to be defined in `BOOT_Conf.h`:

```
#define SBTL_NUM_AREAS_FAILSAFE   1
```

Also, the rescue firmware image has to be made accessible via the `SBTL_ReadArea()` implementation when the emBoot-Secure library is requesting data from it:

```
int SBTL_ReadArea(unsigned Area, U32 Offset, unsigned BuffSize, U8 *pBuff) {

  if (Area == SBTL_MEM_AREA_FAILSAFE(0)) {
    // Return data for rescue image

  }

  // Return data for other regions

}
```

Below, sample code for such a bootloader is provided. For simplicity, code for updating the bootloader itself has been omitted from this sample since it can be found in the previous two sections. The code is shipped with emBoot-Secure in the file `Application/BL_RescueImage.c`.

```
/*********************************************************************
*                    (c) SEGGER Microcontroller GmbH                 *
*                         The Embedded Experts                       *
**********************************************************************
*                                                                    *
*       (c) 2022 - 2026     SEGGER Microcontroller GmbH              *
*                                                                    *
*       www.segger.com     Support: www.segger.com/ticket            *
*                                                                    *
**********************************************************************
*                                                                    *
*       emBoot-Secure * Secure bootloader for embedded applications  *
*                                                                    *
*       Please note: Knowledge of this file may under no             *
*       circumstances be used to write a similar product.            *
*       Thank you for your fairness !                                *
*                                                                    *
**********************************************************************
*                                                                    *
*       emBoot version: Internal                                     *
*                                                                    *
```



```
**********************************************************************
----------------------------------------------------------------------
Purpose     : Sample bootloader application with rescue update package
              logic.
-------------------------- END-OF-HEADER -----------------------------
*/

#include "BOOT.h"
#include "cmsis_compiler.h"

/**********************************************************************
*
*       Static variables
*
**********************************************************************
*/

// Work memory for SBTL_Verify and SBTL_Unwrap
static SBTL_WORK_MEMORY _workMemory; ❶

// Buffer for reading sectors to memory during flashing
static SBTL_FLASH_BUFF  _sectorBuffer;

/**********************************************************************
*
*       RescueButtonPressed()
*
* Function description
*   Checks whether the rescue button is pressed
*
*  Return value
*    >  0: Rescue button is pressed.
*    == 0: Rescue button not pressed.
*/
static int RescueButtonPressed(void) { ❷

  // Put code for reading the state of the rescue button here
  // Return 1 if pressed

  return 0;
}

/**********************************************************************
*
*       main()
*
* Function description
*   Application entry point
*/
int main(void) {
  SBTL_FIRMWARE_INFO  FirmwareInfo;
  int                 r;

  SBTL_Init();

  SBTL_LOG(("Bootloader version %d starting...", SBTL_GetBootloaderVersion()));

  if (RescueButtonPressed() == 1) { ❸

    r = SBTL_FindRescueFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer); ❹
    if (r != 0) {
      SBTL_Panic("Error finding recovery firmware.");
    }

    r = SBTL_UpdateFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer); ❺
    if (r != 0) {
      SBTL_Panic("Error installing recovery firmware.");
```



```
    }

    // If applicable, de-initialize any hardware which is not needed
    // by the application firmware and also reset the clocks and
    // supply voltages to safe settings.


    // Disable interrupts and launch firmware
    __disable_irq();  ❻
    FirmwareInfo.EntryPoint.Jump();
  }

  // Look for firmware updates, and also check whether the
  // currently installed firmware can be verified.
  r = SBTL_ScanFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer); ❼
  if (r != 0) {
    SBTL_Panic("Scanning for valid firmware failed.");
  }

  if (FirmwareInfo.Area == SBTL_MEM_AREA_FIRMWARE) {
    // Boot into already installed firmware

    // If applicable, de-initialize any hardware which is not needed
    // by the application firmware and also reset the clocks and
    // supply voltages to safe settings.

    // Disable interrupts
    __disable_irq();

    // Jump to firmware entry point
    FirmwareInfo.EntryPoint.Jump();
  }

  if ( SBTL_MEM_AREA_IS_UPDATE(FirmwareInfo.Area) ) {
    // Install normal firmware update
    r = SBTL_UpdateFirmware(&FirmwareInfo, &_workMemory, &_sectorBuffer);
    if (r != 0) {
      // Error updating the firmware
      SBTL_LOG(("Error updating the firmware, code: %d", r));
      SBTL_Panic("Error updating the firmware.");
    }

    // Reboot and start firmware
    SBTL_ResetHardware();
  }

  // This place is never reached.
  SBTL_Panic("Bootloader logic failed.");
}
```

❶ Static working memory

Allocates static working memory for the unpacking of the update package and signature validation, as well as a buffer to hold parts of the unpacked update package during flash programming.

❷ Function for rescue condition

This function stub needs to contain code to check whether a rescue condition has occurred, for example whether the user is pressing a certain button.

❸ Check for rescue condition

The bootloader main function checks whether a rescue firmware restoration is requested.

❹ Scan for rescue update packages

The `SBTL_FindRescueFirmware()` function is used to scan for rescue update packages. It behaves differently from the `SBTL_ScanFirmware` used in previous sections, because it only checks for the presence of valid update packages, regardless of their version number.

❺ Install rescue image

The `FirmwareInfo` structure points to an area containing the rescue update package. The function `SBTL_UpdateFirmware()` is called to perform the update.

❻ Start rescue firmware image

Since the rescue condition is only valid for as long as the user presses the button, the installed rescue firmware is launched directly, without a reboot. This prevents the bootloader from reinstalling the problematic firmware image instead of launching the rescue version.

❼ Scan for normal updates

This part of the code is only reached if the rescue button was not pressed during startup. The code scans available firmware update packages and compares them against the version of the currently installed application firmware image. Since the code is the same as in the section *Bootloader update in normal mode*, the annotations are not repeated here.

2.10 Adapting application firmware

Application firmware and bootloader code can be created as independent projects, or as two compilation targets in a single project, depending on user preferences and the amount of code shared between them, for example code for accessing update packages in a filesystem. In any case, the application must be adapted to the flash memory partitioning chosen earlier. Also, the bootloader sets up its own vector table and stack region. The application has to activate its own vector table and stack region. Additionally, the bootloader may configure additional hardware elements, like system clocks, supply voltages, and additional hardware needed to install updates, like external SPI flash memory or a hardware hash acceleration module. During the transition from bootloader to application firmware, these settings may need to be reversed or adapted to the applications needs.

This chapters requires either an existing project for the application firmware or an empty (“Hello-World”) application project.

2.10.1 Adapting the linker script

The first step in adapting the application firmware to the bootloader is the adaptation of the linker script. This is required in order to make room for the bootloader and the firmware header in memory. The following example of a SEGGER Linker script extends the sample described in *Partitioning the non-volatile memory for emBoot-Secure* on page 26.

Since the APP region is placed directly behind the HEADER region, in most cases its start address is not compatible with the alignment requirements of the vector table. In order to guide the linker to place the vector table at the first available address which fulfills its alignment requirements, the instruction for placing the vector table is placed before any other placement instructions for the APP region. The linker is free to place any fitting sections in the space before the vector table.

```
// Linker_Application.icf

define region FLASH1 = [from 0x00000000 size 0x00200000];

#define BOOTLOADER_SIZE 128k
// HEADER_SIZE must match value SBTL_FIRMWARE_HEADER_SIZE in Boot_ConfDefaults.h.
#define HEADER_SIZE     64

define region BOOTLOADER  = [from 0x00000000 size BOOTLOADER_SIZE];
define region HEADER      = [from BOOTLOADER_SIZE size HEADER_SIZE];
define region APP         = FLASH - BOOTLOADER - HEADER;

// snipped: definitions of blocks and sections

place in APP
    { block vectors };    // Vector table section
place in APP with minimum size order
    {
      block tdata_load,   // Thread-local-storage load image
      block exidx,        // ARM exception unwinding block
      block ctors,        // Constructors block
      block dtors,        // Destructors block
      readonly,           // Catch-all for readonly data
      readexec            // Catch-all for (readonly) executable code
    };
```

2.10.2 Activating the vector table

For many ARM processors, the alignment of the vector table depends on its size. The alignment usually has to be to the next-larger power of two. For the MK66F18 mentioned earlier, the vector table has to have 512 byte alignment. The assembly file which defines the vector table therefore needs to include this information, provided by the `.balign` directive. Alternatively, the alignment can also be specified in linker scripts.


```
.section  .vectors, "ax"
          .code 16
          .balign 512
          .global _vectors
_vectors:
          //
          // Internal exceptions and interrupts
          //
          VECTOR __stack_end__
          VECTOR Reset_Handler
          // More vector table entries ...
```

When the bootloader activates the application, it reads its entrypoint from the header produced by the FirmwareTool and performs a jump to this address, which is usually the reset handler defined by the application. Since the microcontroller activated the bootloader's vector table during startup, the application's reset handler has to activate its own vector table. The startup code provided by the microcontroller's manufacturer, CMSIS code or the compiler vendor usually does this by referring to the vector section using its symbolic name. For some variants of startup code, this step has to be manually activated using a compiler flag, while some startup code contains a hardcoded address such as `0x0` for the location of the vector table. In these cases, either the flag has to be set or the startup code has to be modified accordingly. In case of startup files delivered with Segger's embOS, the `__VTOR_CONFIG` flag has to be defined project-wide.

```
#if defined(__VTOR_CONFIG) || defined(__VECTORS_IN_RAM)
        //
        // Configure vector table offset register
        //
#ifdef __ARM_ARCH_6M__
        ldr     R0, =0xE000ED08    // VTOR_REG
#else
        movw    R0, 0xED08         // VTOR_REG
        movt    R0, 0xE000
#endif
        ldr     R1, =_vectors
        str     R1, [R0]
#endif
```

The vector table not only contains the application's entry point, but also the starting address for the stack pointer. When the bootloader jumps to the application's entry point, the stack pointer is still pointing to the stack defined by the bootloader. Therefore, the startup code also has to read the desired starting point of the stack from the application's vector table. Again, some startup code requires a compiler flag to be set in order to perform this step, while in other code, this step may be missing. In case of startup files delivered with Segger's embOS, the `__INITIALIZE_STACKPOINTER` flag has to be defined project-wide.

```
#ifdef __INITIALIZE_STACKPOINTER
        ldr R0, =__stack_end__
        mov SP, R0
#endif
```

While adapting the application firmware to be launchable by the bootloader, proper initialization of the vector table and stack pointer addresses should be checked by setting a breakpoint into the application's `main()` function. If there is a difference from the expected settings, the startup code should be inspected for additional places in which it modifies the respective registers. The CMSIS files supplied by the microcontroller's manufacturer may contain such code, in addition to the start-up code implemented by the compiler or embedded operating system vendor.

2.10.3 Hardware reconfiguration

When a microcontroller resets, its supply voltages and clock settings are set to safe defaults. The startup code is responsible for adjusting these settings to fulfill the requirements of the firmware. The requirements can be low power consumption, high processing speed or something more balanced. To achieve this state, the startup code adapts the supply voltages to match the desired clock speed and configures the clock sources and dividers. When the bootloader hands over control to the application firmware, its startup code may have already configured non-default clock and supply voltage settings. In the best case, a repetition of the configuration steps in the startup code will not lead to faults and no adaptation is needed. In other cases, either the startup code of the application has to take the state left by the bootloader into account, or the bootloader has to de-initialize these settings.

Chapter 3

Passwords, keys, and key files

emBoot-Secure makes use of cryptographic procedures in order to protect firmware images from malicious modification and inspection. These procedures require secrets to operate, typically in form of cryptographic keys. This section gives a high-level overview of the secrets involved in the security architecture of emBoot-Secure and provides guidance on how to generate, store, protect, and use them.

3.1 Overview

3.1.1 emBoot-Secure with and without the Signature Server

There are two possible ways of using emBoot-Secure: It can be used with or without the SEGGER Signature Server. The Signature Server is a dedicated hardware device that can be used to generate and store keys securely and to generate firmware signatures on a trusted device (see chapter *The Signature Server*). Alternatively, keys and signatures can be generated on a personal computer. The following figures provide an overview of the secrets involved in both scenarios. Each secret is explained in detail in later sections in this chapter.

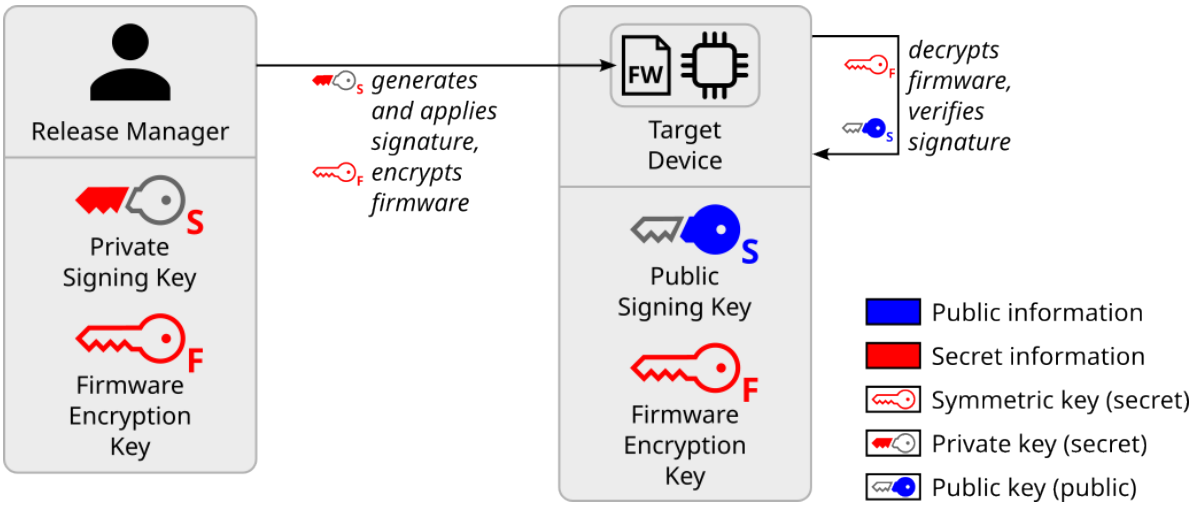


Figure: Overview of secrets in emBoot-Secure (without Signature Server)

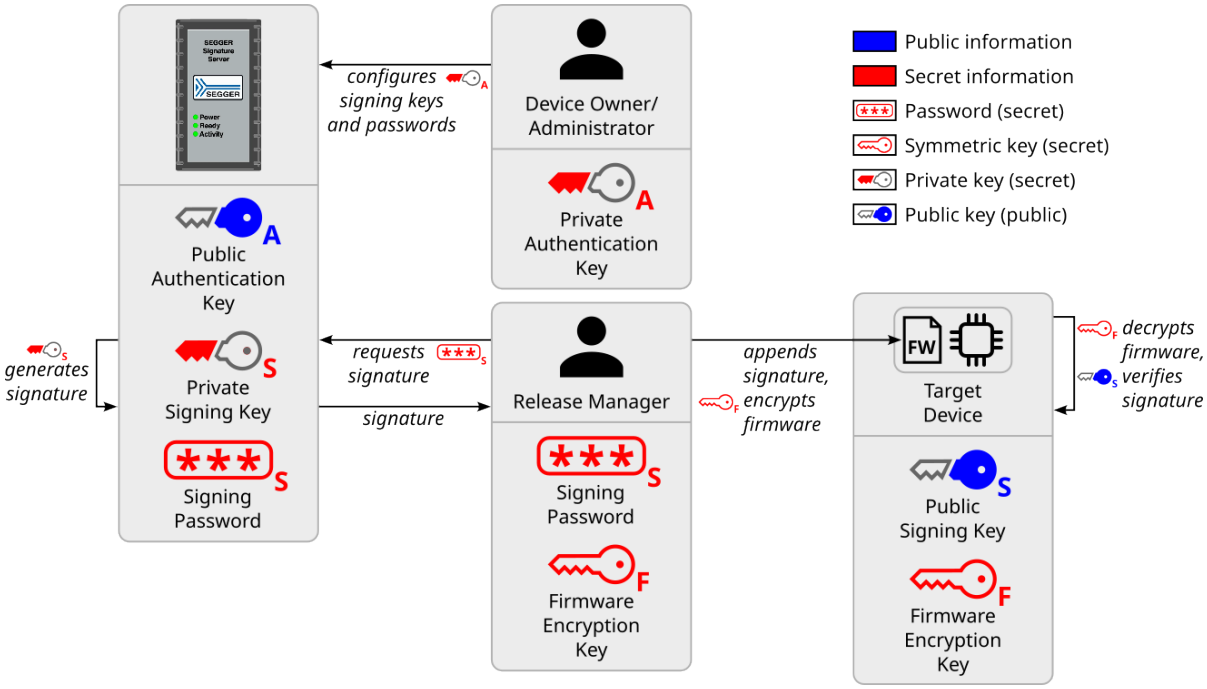


Figure: Overview of secrets in emBoot-Secure (when using the Signature Server)

Overview of user roles

The figure above depicts various user roles, which are explained in detail in the section *User roles*. In short, if emBoot-Secure is used without the Signature Server, there is only a

single user role: the Release Manager. This role is responsible for creating update packages for new firmware releases.

If the Signature Server is used, the Release Manager becomes a user of the Signature Server and uses the keys stored in the server's key store. Furthermore, two additional user roles exist for the Signature Server: Administrator and Device Owner. A Device Owner is a user with highest privilege and has access to all keys on the Signature Server. Administrators have similar privileges to Devices Owners, but can only manage a subset of keys on the Signature Server.

3.1.2 Types of secrets

emBoot-Secure uses three conceptually different kinds of secrets: passwords, symmetric keys, and asymmetric keys (also referred to as key pairs). While passwords are a well-known concept, keys need some explanation.

Cryptographic keys

A cryptographic key, or key for short, is a sequence of bits that is used to control cryptographic operations. Depending on the cryptographic algorithm in use, the key has to meet specific criteria, for example with respect to mathematical properties or its randomness. emBoot-Secure ships with a software program that generates the required keys and takes these requirements into account.

There are two common types of cryptographic keys: symmetric and asymmetric keys.

Symmetric keys

An algorithm that utilizes symmetric keys uses the same key for all operations that it performs. For example, a symmetric encryption algorithm uses the same key for both of its operations: encryption and decryption. emBoot-Secure uses a symmetric encryption algorithm to encrypt update packages.

Asymmetric keys

In contrast, asymmetric keys always appear in pairs of two: a public key and a private key. Both keys are linked together, but serve different purposes. In an asymmetric encryption algorithm, for example, the public key is used for encryption, while the corresponding private key is used for decryption. In emBoot-Secure, asymmetric keys are used to implement digital signatures and authentication.

Storing keys

Symmetric keys and private keys must be stored securely. Because of their complexity, keys cannot easily be memorized. emBoot-Secure provides two options for key storage: First, keys can be exported to a key file, a small file that can be stored on a digital medium. In order to protect the key file from unauthorized access, it is protected with a password. Second, some keys can be stored securely in the Signature Server.

3.1.3 Digital signatures

emBoot-Secure ensures that only firmware images that have been approved by the manufacturer can be executed on the target device. This functionality is implemented through the cryptographic concept of digital signatures.

Much like a handwritten signature on a document, a digital signature shows that a particular person (or company) approves of the content of the signed object. In the case of emBoot-Secure, the signed object is the firmware image.

On an abstract level, this digital signature works as follows: The signer uses the firmware image and a private key to compute a signature value. The signature value is a bit string that is appended to the firmware image. It proves that the firmware image is authentic: It is cryptographically guaranteed that only the owner of the private key (i.e., the manufacturer) can generate that bit string.

On the target device, a cryptographic procedure uses a corresponding public key to verify that the signature on the firmware image was in fact generated by the manufacturer (i.e., the owner of the private key). This verification step is performed on every boot and when installing firmware updates. Firmware images with invalid signatures are rejected.

3.2 Secrets in emBoot-Secure

This section describes the secrets that appear in the security architecture of emBoot-Secure and that the users interact with. First, we discuss the minimal set of secrets that is present in any scenario where emBoot-Secure is used. Then, we discuss additional secrets that are only required when the Signature Server is used.

3.2.1 Signing key pair

Overview: Private signing key	
Secret?	Yes
Owner	**With Signature Server:** Device Owner, Administrator **Without Signature Server:** Release Manager
Purpose	Sign new firmware images
Generation	On a PC (*key generation tool*) or on the Signature Server
Storage	Password-protected key file in multiple safe locations (see *recommendations*); on the Signature Server

Overview: Public signing key	
Secret?	No
Owner	**With Signature Server:** Device Owner, Administrator **Without Signature Server:** Release Manager
Purpose	Verify authenticity of firmware images
Generation	On a PC (*key generation tool*) or on the Signature Server
Storage	On each target device (included in bootloader code)

One goal of emBoot-Secure is to ensure the authenticity of firmware images. This means that only firmware images that are approved by the manufacturer can be executed on a target device. The manufacturer's approval is expressed through a cryptographic signature, which is attached to a firmware image when it is packaged into an update package to be released. This signature is verified by the target device during the update process and every time the device boots up. If the firmware image does not have a valid signature, the respective process (update or boot) is aborted.

To generate and verify signatures, emBoot-Secure uses an asymmetric signing key pair. When no Signature Server is used, this key pair is generated by a Release Manager on a PC. If a Signature Server is used, a Device Owner or Administrator can generate the key pair on the Signature Server.

In any case, the key pair is only generated once before the first firmware release and remains the same throughout the product's lifetime.

The signing key pair consists of a public signing key and a private signing key which serve different purposes and have to be treated differently:

- The **private signing key** is used to sign firmware images. Anyone who has access to this key can generate valid update packages that the target device will accept. Therefore, this key remains with the manufacturer and **must be kept confidential at all times**. In addition, it is recommended to **keep backup copies of this key in multiple safe locations**. The private signing key can be stored in a key file on a digital medium, such as a hard drive. More details on how to handle key files are provided in the section *Security recommendations for keys and key files*. It can also be imported into the Signature Server, where it is stored securely and can be used to create signatures through a protected interface.

> **Warning**

> The private signing key is the most important key in the security architecture of emBoot-Secure. If a malicious actor gains access to this key, they can sign arbitrary firmware images that the target device will accept as valid. If the private signing key is lost, it will be impossible to create new update packages that the target device will accept.

- The **public signing key** is used to verify signed firmware images. This key is integrated into the bootloader and ships with the target device. It does not have to be kept confidential.

3.2.2 Firmware encryption key

Overview: Firmware encryption key	
Secret?	Yes
Owner	Release Manager
Purpose	Encrypt or decrypt a firmware image
Generation	On a PC (*key generation tool*)
Storage	Password-protected key file in multiple safe locations (see *recommendations*); on each target device (included in bootloader code)

emBoot-Secure can optionally also encrypt the firmware image in the update package. Encryption provides confidentiality, i.e., it prevents third parties from inspecting the firmware. This security service is based on a symmetric firmware encryption key. The encryption is an optional feature - when the bootloader supports encrypted update packages, it can still process update packages which are not encrypted.

The firmware encryption key is generated on a PC (using the *key generation tool*) before the first firmware release and remains the same throughout the product's lifetime. It is stored in a password-protected key file. It is recommended to protect the key file with a strong password. More details on how to handle key files are provided in the section *Security recommendations for keys and key files*.

It is possible to store multiple encryption keys in a single bootloader. This provides a means to switch to another encryption key in case the first key is compromised.

3.2.3 Secrets related to the Signature Server

Two additional secrets come into play when emBoot-Secure is used together with the Signature Server: the authentication key pair and the signing password. Both secrets restrict access to the Signature Server and the key material enclosed in it.

Specifically, the Signature Server distinguishes multiple user roles: Device Owner, Administrator, and Release Manager (see *User roles*). In short, Device Owners and Administrators configure the Signature Server and manage its key storage, while Release Managers use those keys to generate signatures for new firmware releases. These user roles use different secrets to authenticate.

3.2.3.1 Authentication key pair

Overview: Private authentication key	
Secret?	Yes
Owner	Device Owner, Administrator
Purpose	Prove the Administrator's identity to the Signature Server
Generation	On the Administrator's PC (*key generation tool*)
Storage	Password-protected key file (see *recommendations*) kept by the Administrator

Overview: Public authentication key	
Secret?	No
Owner	Device Owner, Administrator
Purpose	Verify the Administrator’s identity on the Signature Server
Generation	On the Administrator’s PC (*key generation tool*)
Storage	On the Signature Server

Device Owners and Administrators are authenticated through their personal authentication key pair. This key pair is generated on the user’s PC. Similarly to a signature key pair, it consists of a public and a private key component. The private authentication key remains with the user and must be kept safe and confidential. It is stored in a key file that should be protected with a strong password. More details on how to handle key files are provided in the section *Security recommendations for keys and key files*. The public authentication key is transmitted to the Signature Server and stored there to authenticate the user in the future.

3.2.3.2 Signing password

Overview: Signing password	
Secret?	Yes
Owner	Release Manager
Purpose	Prove the Release Manager’s identity to the Signature Server; verify their identity on the Signature Server
Generation	Configured in the Signature Server by the Administrator
Storage	For proof: memorized by Release Manager or stored in secure password storage; for verification: Signature Server

Release Managers authenticate with a personal signing password. Signing passwords are set up by an Administrator.

Every signing password must be strong and stored only in safe places (e.g., memorized or stored in a secure password manager), as anyone who knows a signing password and has access to the Signature Server is able to produce valid update packages. We recommend following the password guidelines of your local cybersecurity authorities, such as the *Federal Office for Information Security (BSI) in Germany*.

3.3 Choosing a signature scheme and its parameters

emBoot-Secure supports two different signature schemes for signing firmware images, each of which can be further configured with various parameters that impact the overall level of security. Before signing the first firmware release, a signature scheme and the corresponding parameters must be selected.

Naturally, security does not come for free: A higher level of security often also increases the time and memory requirements of the underlying cryptographic algorithms, impacting the boot time of the target device as well as the duration of the update process. This section provides guidance on the selection of schemes, parameters, and algorithms, allowing the user to find a suitable trade-off for their application. Additional information on the performance impact of various parameters is provided in the section *Resource use and performance*.

Generally, emBoot-Secure supports the following variants of digital signatures:

- RSA signatures (RSASSA-PSS) with SHA-1, SHA-256, or SHA-512 hashes and key lengths up to 8192 bits
- ECDSA signatures with SHA-1, SHA-256, SHA-384, or SHA-512 hashes on the curves listed below

The following curves are supported for the ECDSA signature scheme:

- `brainpoolP256r1`: 256 bits
- `brainpoolP320r1`: 320 bits
- `brainpoolP384r1`: 384 bits
- `brainpoolP512r1`: 512 bits
- NIST P-256 (`secp256r1`): 256 bits
- NIST P-384 (`secp384r1`): 384 bits
- NIST P-521 (`secp521r1`): 521 bits

3.3.1 Precedence of any applicable regulations

If the target device falls under any cybersecurity regulation, the signature scheme and its parameters should be chosen according to the requirements and/or recommendations of the relevant regulatory bodies.

If this is not the case, the following sections provide some general guidance based on the recommendations of the German Federal Office for Information Security (*BSI TR-02102-1*) and the National Institute of Standards and Technology of the United States (*NIST SP 800-78-5* and *NIST SP 800-131A Rev. 2*).

3.3.2 Signature scheme

First, a signature scheme must be selected. emBoot-Secure supports RSA and ECDSA signatures.

RSA is an established cryptographic mechanism since the 1970s, while ECDSA was first proposed in the early 1990s. The German BSI considers cryptography based on RSA and elliptic curves (EC) equally secure, but notes that EC algorithms are more efficient for high security levels (see *BSI TR-02102-1*, Section 2.3). In particular, RSA requires larger keys and produces larger signatures compared to ECDSA at the same security level. Generating large RSA keys on the Signature Server can take up one hour, while ECDSA keys can be generated within seconds.

> **Note**
>
> If you have the choice, prefer ECDSA over RSA to get smaller signatures at the same level of security.

3.3.3 Key size and curves

Next, a key size (for RSA) or a curve (for ECDSA) has to be selected. These parameters impact the level of security of the resulting signature significantly.

For RSA: The German BSI recommends to use keys that are at least 3000 bits long (see *BSI TR-02102-1*, Table 1.2). NIST recommends at least 3072 bits for long-term applications (see *NIST SP 800-78-5*, Table 2).

For ECDSA: The key size depends on the chosen curve. The German BSI recommends using one of the "Brainpool" curves (see *BSI TR-02102-1*, Section 2.3) with a size of at least 250 bits (see *BSI TR-02102-1*, Table 1.2). All Brainpool curves supported by emBoot-Secure fulfill this key size requirement. NIST recommends using one of their own curves, at least P-256 or higher (see *NIST SP 800-78-5*, Table 2). All NIST curves supported by emBoot-Secure fulfill this requirement.

The major difference between the Brainpool and NIST curves lies in the entity that selected the cryptographic parameters for the curves. Brainpool is a working group of German institutions, universities, and companies. NIST is an institute of the United States. There is no noticeable performance difference between Brainpool curves and NIST curves of (approximately) the same length.

> **Note**
>
> If you have the choice, choose the following parameters for a reasonable level of security:
>
> **For RSA:** Use a key size of at least 3072 bits.
>
> **For ECDSA:** Use at least a 256 bit curve from the preferred standardization body.

3.3.4 Hash function

Finally, a hash function has to be selected.

The SHA-1 hash function is considered insecure and should be avoided whenever possible; it should only be used where performance is crucial (see section *Resource use and performance*) or where compatibility with legacy systems has to be maintained.

The remaining hash functions, SHA-256, SHA-384, and SHA-512, are generally considered secure (see *BSI TR-02102-1*, Section 4, and *NIST SP 800-131A Rev. 2* Rev. 2, Table 8), but require increasing computation time and storage. For most applications, SHA-256 provides an appropriate level of security.

> **Note**
>
> If you have the choice, use at least the SHA-256 hash function. Avoid SHA-1.

3.3.5 Summary and final recommendation

In summary, if no external regulations impose other requirements on the target device, the following configuration is a reasonable choice in most scenarios:

- Use the ECDSA signature scheme
- Use curve `brainpoolP256r1` or `secp256r1`
- Use the SHA-256 hash algorithm

Using ECDSA with the SHA-256 hash function is the default and can be changed in the file `BOOT_Conf.h` (see *Configuration switches related to algorithms*). The curve is selected during key generation (see *Generating an ECDSA key pair (KeyTool)* and *Generating an ECDSA key pair (Signature Server)*).

3.4 Details on firmware encryption

A firmware image can optionally be encrypted using a symmetric cipher. If encryption is enabled, the firmware image is encrypted when it is packaged into an update package. The update package is then decrypted on the target device during the update process. Afterwards, the firmware image is stored in plain in the target's flash memory.

emBoot-Secure implements this feature using a strong encryption algorithm, the Advanced Encryption Standard (AES), with a key length of 128, 192, or 256 bits. This algorithm is generally considered secure (see *BSI TR-02102-1*, Section 3.1, and *NIST SP 800-131A Rev. 2*, Table 1). It is used in Galois/Counter Mode (GCM), which is considered a secure way of using AES to decrypt larger amounts of data (see *BSI TR-02102-1*, Section 3.1.1, and *NIST SP 800-38D*). An additional benefit of AES-GCM is that an additional layer of authentication is provided, allowing the system to detect malicious or accidental modifications as early as possible. If a modification of the update package is detected at this stage, it is flagged as invalid and not considered for installation.

Note that firmware encryption has inherent limitations, even when used with strong ciphers. The encryption protects the confidentiality of the firmware image only while it is "in transit", i.e., as long as it is packaged in the update package. Malicious actors with physical access to a target device can attempt to extract the firmware encryption key or the decrypted firmware image itself directly from the target's flash or memory. Any protection of the firmware and the encryption key inside the device may be supported by the target hardware.

3.5 Security recommendations for keys and key files

There are a few things to consider when operating with files that contain private keys. This applies to private signing keys (when stored outside the Signature Server) as well as the private authentication keys of the Signature Server. Files containing public keys can be treated more leniently, as they do not contain any confidential information.

3.5.1 Use strong passwords to protect key files

It is recommended to follow the password guidelines of your local cybersecurity authorities, such as the *Federal Office for Information Security (BSI) in Germany*.

Generally, long and complex passwords are considered stronger than short and simple passwords. Dictionary words or company names, such as the name of the manufacturer of the target device, do not qualify as strong passwords. Passwords should be memorized or stored in secure locations, such as secure password managers.

3.5.2 Only use dedicated media to store key files

Whenever a private key is stored on a digital medium (e.g., in a file on a hard disk), it leaves traces on the medium that persist even if the file is removed or the medium is formatted. This is due to the way most digital media work internally: when a file is removed, the associated data structure is not immediately wiped from the medium. Instead, only an entry in the index of the file system is removed. This means that the actual private key data remain on the medium (until they are overwritten by new data eventually) and can be recovered with data recovery software or in specialized hardware labs.

Therefore, it is recommended to store private keys only on digital media that are dedicated for this purpose. Besides the Signature Server, this could be a dedicated external hard drive that is only used to keep private keys.

It should also be avoided to save a private key to a computer's local hard disk temporarily (for example, after it was generated) and then copy it over to an external medium for long-term storage. This process leaves traces of the private key on the computer's local hard disk.

3.5.3 Keep backup copies in safe places

It is further recommended to keep multiple copies of important key files, especially of the private signing key. A commonly used rule of thumb is the "3-2-1 backup strategy": store at least three copies of the key, on at least two different types of media, and keep at least one of the copies in a different location (e.g., in a safe deposit box at a bank).

It is important to choose reliable storage media for this key storage. The use of USB flash drives, SD cards, and similar flash media is not recommended due to their poor long-term reliability. As a backup of last resort, key files can also be printed out on paper.

An appropriate backup strategy is required even when the Signature Server is used. The Signature Server can be used to hold *one of the copies* of a key, but it should not hold *the only copy*. Being a hardware device, the Signature Server is not immune to failure. In addition, an Administrator could accidentally remove a key.

Although it may seem convenient, it is considered bad practice to store private key files in source code repositories, as those tend to have complex and intransparent access control structures and the storage devices that contain the repository are typically out of the user's control.

Chapter 4

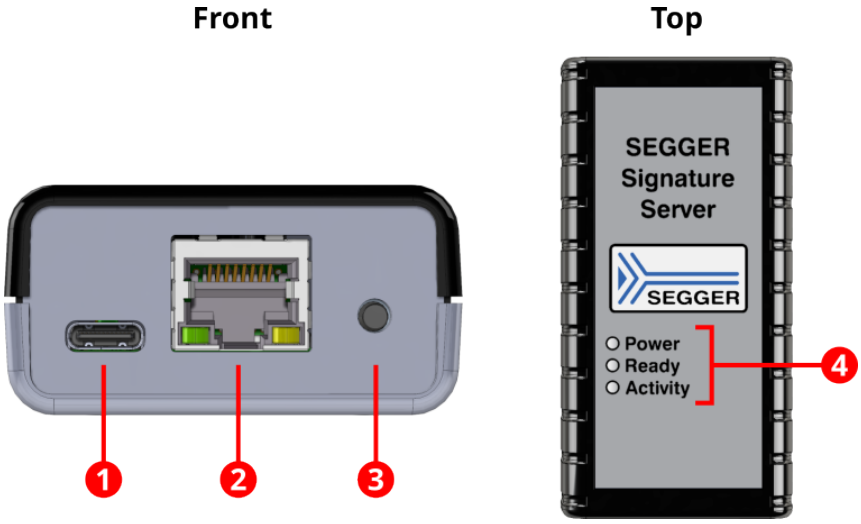
The Signature Server

The Signature Server is a hardware device that stores private signing keys securely and uses them to generate signatures for update packages. It restricts access to those signing keys through an interface that requires authentication.

This chapter explains the communication interfaces of the Signature Server, its user roles, how to connect to it, how to initialize it, and how to operate it.

4.1 Hardware overview

This section describes the external features of the Signature Server. The following figure provides an overview.



No.	Component
Front	
(1)	USB-C port (power supply and communication)
(2)	Ethernet port
(3)	Push button
Top	
(4)	Status LEDs

4.1.1 Ports and push button

There are two ports on the front of the server: a USB-C port (1) and an Ethernet port (2).

The USB port is used to power the device and as a communication interface for administrative operations, such as managing keys. The Signature Server consumes less than 1 watt of power and can be powered either by a computer or a dedicated power adapter.

The Ethernet port is used as a communication interface to request and transmit new signatures.

Furthermore, there is a push button (3) on the front of the Signature Server. This button is used to enable the USB command interface (see *Connecting to the USB interface*) and to confirm dangerous operations, such as a factory reset.

4.1.2 Status LEDs

On the top of the Signature Server is a row of three status LEDs (4): *Power*, *Ready*, and *Activity*. Each LED can light up in one of three colors, red, green, or orange, indicating different states of the device.

4.1.2.1 LED patterns in normal operation

During normal operation of the Signature Server, the Power LED lights up green. If the Power LED flashes green, the USB command interface is enabled (see *Connecting to the USB interface*).

The Ready LED indicates the Signature Server's global state (see *Signature Server global states*), and the Activity LED indicates the status of command execution.

Power	Ready	Activity	Description
Power LED			
🟢	(any)	(any)	Signature Server in normal operation
☼🟢☼	(any)	(any)	Signature Server in normal operation, USB command interface enabled (see *Connecting to the USB interface*)
Ready/Activity LEDs			
🟢 /☼🟢☼	🟠	⚪	Signature Server in factory state
🟢 /☼🟢☼	⚪	⚪	Signature Server Device Owner configured
🟢 /☼🟢☼	🟢	⚪	Signature Server in operational state
🟢 /☼🟢☼	🔴	⚪	Signature Server in “Fault” state (key store damaged)
🟢 /☼🟢☼	(any)	☼🟢☼	A command is being executed
🟢 /☼🟢☼	⚪	☼🔴☼	Waiting for factory reset confirmation (see *Resetting to factory state*)

Legend: ☼○☼: LED flashing

4.1.2.2 Diagnostic LED patterns

When the Signature Server updates its own firmware or encounters an error condition during startup, the Power LED lights up in orange. In this case, the Ready and Activity LEDs are used as status indicators.

Power	Ready	Activity	Description
🟠	⚪	⚪	Signature Server in startup phase
🟠	🟠	⚪	Starting Signature Server firmware
🟠	🔴	⚪	Invalid Signature Server firmware
🟠	⚪	🟠	Firmware update in progress
🟠	⚪	🔴	Bootloader update in progress (update phase 1)
🟠	🟠	🔴	Bootloader update in progress (update phase 2)

4.1.3 Physical security

The Signature Server should be kept in a safe and trusted environment at all times. The keys stored on the Signature Server cannot be used or exported without prior authentication. However, everyone with physical access to the device can unlock the USB command interface and, at the very least, reset the Signature Server to factory state, erasing all the keys stored on the device.

4.2 Signature Server global states

The Signature Server can be in one of the following states:

Factory state

The Signature Server does not contains any keys, either because it is a new device or a reset to factory state has been performed. In this state anybody can take ownership of the Signature Server, see *Initialization of the Signature Server*.

Owned

A Device Owner key has been set and the Signature Server is owned by this Device Owner.

Operational

The device has been configured and at least one signature key and a corresponding Release Manager is installed.

Fault

The Signature Server can’t access the keys anymore due to a problem with the key store flash memory (corrupted data, read/write access error). In this case the Signature Server may be power-cycled in order to recover from a nonrecurring flash read problem. But if the Signature Server remains in “Fault” state or shows this state multiple times, it should be taken out of service.

4.3 User roles

The Signature Server distinguishes three user roles with different privilege levels: Device Owners, Administrators and Release Managers.

In short, Device Owners are permitted to configure the Signature Server and to manage all keys stored on it. They can also set up Administrator users. Administrators have the same permissions as Device Owners, except that they can only manage a subset of the keys stored on the Signature Server. Release Managers use specific keys on the Signature Server to generate signatures for update packages.

There is a single Device Owner for each Signature Server, but there can be multiple Administrators and multiple Release Managers. It is recommended to make use of this permission scheme and set up individual user credentials for each person that needs to access the Signature Server. User credentials should not be shared.

If the same person acts as an Administrator (or Device Owner) and as a Release Manager, they are assigned two credentials.

4.3.1 Device Owner role

Overview: Device Owner role	
Tasks	Manage server configuration, create Administrators, manage signing passwords, perform firmware updates (of the Signature Server)
Interfaces	USB only
Credentials	*Authentication key pair*: User: Private authentication key Signature Server: Public authentication key
Tool	*Signature Server administration tool*

The Device Owner puts the Signature Server into operation: This role is automatically assigned to the first user who uploads a public authentication key.

A Device Owner accesses the Signature Server via the USB command interface and authenticates themselves with their personal private authentication key. To validate the Device Owner’s identity, the Signature Server stores the Device Owner’s public authentication key.

The Device Owner is permitted to manage the server’s configuration and to perform firmware updates if needed. In addition, the Device Owner is responsible for creating Administrator accounts by importing the public authentication keys of the Administrators into the device.

The Device Owner can further manage all private signing keys on the server (e.g., import or generate new keys, or export or remove existing keys), and can also manage Release Manager users if needed. If the

Signature Server is used by a very small group of people, there may be no need to set up distinct Administrator users, as the Device Owner can perform all operations that an Administrator can perform as well.

A Device Owner uses the *Signature Server administration tool* to interact with the Signature Server.

4.3.2 Administrator role

Overview: Administrator role	
Tasks	Manage server configuration, manage private signing keys (import, export, generate, remove, etc.), manage public authentication keys, manage signing passwords
Interfaces	USB only
Credentials	*Authentication key pair*:

	User: Private authentication key Signature Server: Public authentication key
Tools	*Signature Server administration tool* *Key generation tool*

An Administrator accesses the Signature Server via the USB command interface and authenticates themselves with their personal private authentication key. To validate the Administrator's identity, the Signature Server stores the Administrator's public authentication key.

The Administrator role is very similar to the Device Owner role. Administrators can also configure the Signature Server, and they can manage private signing keys that are stored in the Signature Server's key store. More precisely, they are permitted to import, export, generate, and remove private signing keys. In contrast to Device Owners, however, Administrators can be restricted to only have access to a subset of keys.

In addition, Administrators manage Release Managers (see *Signing password*) and can grant them permission to use private signing keys.

An Administrator uses the *Signature Server administration tool* to interact with the Signature Server and optionally the key generation tool to create key pairs for signing.

4.3.3 Release Manager role

Overview: Release Manager role	
Tasks	Generate signatures using private signing keys stored in the Signature Server
Interfaces	USB, Ethernet
Credentials	*Signing password*
Tool	*Firmware preparation tool*

A Release Manager accesses the Signature Server via USB or Ethernet and authenticates themselves with their personal signing password. This password is set by an Administrator or Device Owner.

When the Release Manager prepares a new firmware release, they use the *firmware preparation tool* to connect to the Signature Server and to generate a signature for the update.

Release Managers can only use private signing keys for which they were granted permission (see *Granting permission*).

4.3.4 Sample Signature Server configuration

This section shows an example of a Signature Server configuration for two products using emBoot-Secure. Responsible for the key management of product 1 is the Administrator `Admin1a` and his substitute `Admin1b`. There are two release managers for product 1 using signature keys `SignKey1a` and `SignKey1b`. Responsible for the key management of product 2 is the Administrator `Admin2`. The release manager for product 2 is `RelManager2` using signature key `SignKey2`.

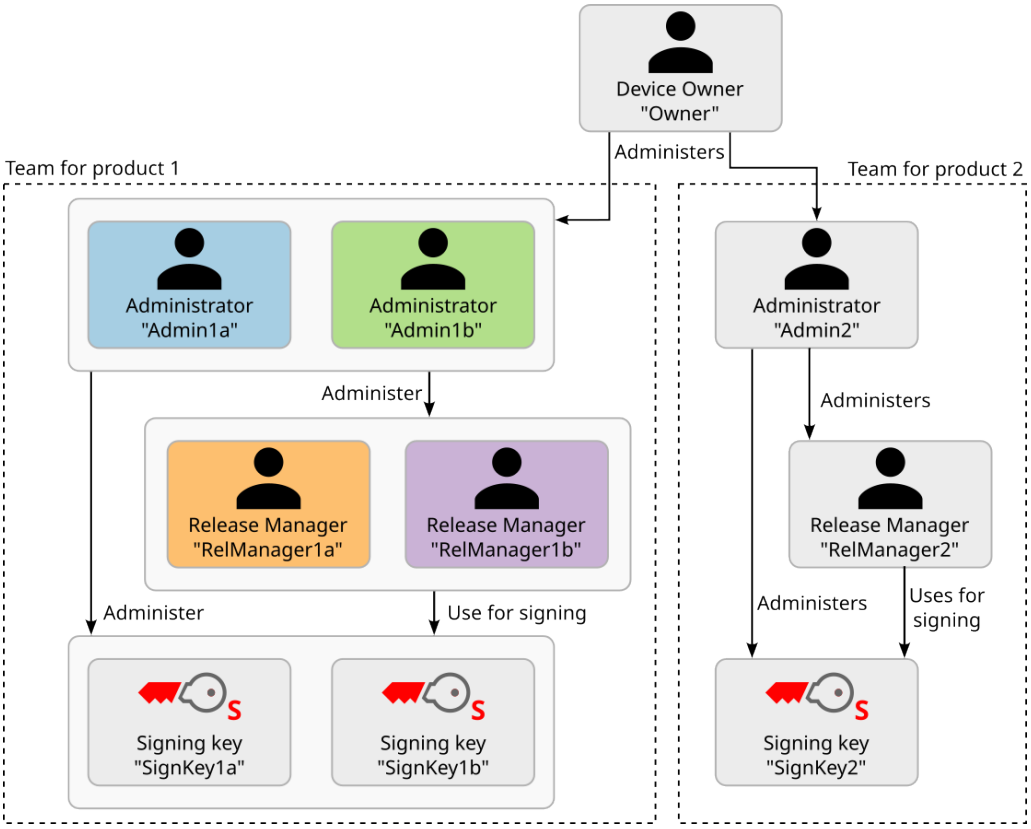


Figure: Overview of user roles on the Signature Server: Two teams working on separate products.

The output of the 'list' command of the Signature Server (see *Listing users and keys*) looks like:

```
Name              Usage  Typ Att   Accessible to
-----------------------------------------------
Owner             ECDSA  Own       Owner
Admin1a           ECDSA  Adm       Admin1a
Admin1b           ECDSA  Adm       Admin1b
SignKey1a         ECDSA  Sig noEx  Admin1a, Admin1b, RelManager1a, RelManager1b
SignKey1b         ECDSA  Sig noEx  Admin1a, Admin1b, RelManager1a, RelManager1b
RelManager1a      PWD    Usr Rem   Admin1a, RelManager1a
RelManager1b      PWD    Usr Rem   Admin1a, RelManager1b
Admin2            ECDSA  Adm       Admin2
SignKey2          RSA    Sig       Admin2, RelManager2
RelManager2       PWD    Usr Rem   Admin2, RelManager2
```


4.4 Connecting to the Signature Server

The Signature Server can be controlled via USB or Ethernet. Both interfaces can be used to request new signatures, but key management commands, such as generating, importing, and exporting keys, can only be sent via the USB interface.

4.4.1 Connecting to the USB interface

In order to connect to the Signature Server via USB, the USB command interface must first be unlocked as follows:

1. Connect the USB cable to the Signature Server and to the user's PC.
2. Wait for the Signature Server to start up. The Signature Server is ready when the Power LED lights up green and no error is indicated (see *Status LEDs*).
3. Push and hold the button on the front of the device for one second. The Power LED begins flashing green to indicate that the USB command interface is enabled. The *Signature Server administration tool* and the *firmware preparation tool* can now be used to communicate with the Signature Server.

This unlocking procedure must be repeated whenever the Signature Server was unplugged, lost power, or did not receive a command for 20 minutes.

The USB command interface does not require any custom drivers on modern operating systems. On Linux systems, however, the Signature Server's USB command interface can only be accessed by processes running with root privileges by default. In order to use the interface with regular user privileges, a udev rule has to be configured. The emBoot-Secure shipping package contains a sample configuration file `99-Segger-SigServer.rules`, which can be copied to `/etc/udev/rules.d` on the user's machine. Refer to the Linux manual page for udev for more information.

4.4.2 Configuring and connecting to the Ethernet interface

The Ethernet interface uses DHCP to acquire an IPv4 address by default. It also provides it's default hostname **`SignatureServer-<Serialnumber>`** to the DHCP server such that its name can be resolved by a local DHCP server. It also advertises its presence using mDNS. See *Specifying a Signature Server address* for more details about how to connect to the server in this case.

It's configuration can be changed once a Device Owner has been taken ownership of the Signature Server, see section *Setting up the Device Owner account*. Custom network parameters can be set up via the USB command interface and the *Signature Server administration tool*. The following properties of the Signature Server can be configured:

- Host name (see *Setting the hostname*)
- Domain name (see *Setting the domain name*)
- TCP port number (see *Setting the port number*)
- Static IPv4 address configuration: IPv4 address, subnet mask, and gateway address (see *Configuring IP settings*)

The *Signature Server administration tool* and the *firmware preparation tool* can be used to connect to the Signature Server via Ethernet. The address or hostname of the Signature Server have to be specified on the commandline as described in *Specifying a Signature Server address*.

4.5 Initialization of the Signature Server

The emBoot-Secure shipping package contains three command line tools that are required to initialize and operate the Signature Server:

- The *key generation tool* (**KeyTool**) to generate and convert keys
- The *Signature Server administration tool* (**SigServerAdmin**) to initialize and manage the key store on the Signature Server.
- The *firmware preparation tool* (**FirmwareTool**) to create and sign update packages

This section guides the user through the initialization process that brings the Signature Server from the factory state into the operational state.

4.5.1 Setting up the Device Owner account

The first setup step is to set up the Device Owner account to manage the Signature Server and the Administrators (see *Device Owner role* for more information). To this end, the Device Owner needs to generate an authentication key pair and import it into the Signature Server. The purpose of this key pair is explained in the section *Authentication key pair*. In short, it identifies and authenticates the Device Owner and has to be passed to the Signature Server administration tool whenever an administrative command is executed.

4.5.1.1 Generating an authentication key pair

The authentication key pair can be an RSA or an ECDSA key pair. It is generated with the key generation tool. For example, an ECDSA key pair on a 256-bit curve can be generated as follows:

```
KeyTool gen-ecdsa -s 256 -p --out E:\owner_auth_key
```

This command may take some time, especially if large RSA keys are generated. It creates two key files, `auth_key.prv` (containing the private authentication key) and `auth_key.pub` (containing the public authentication key). The tool will ask for a password to protect the private key file. A strong password should be chosen for this purpose.

The private key file should be stored only in safe locations and backup copies should be created. For further recommendations on how to deal with key files, see *Security recommendations for keys and key files*.

4.5.1.2 Importing the authentication key pair

While the private key file remains with the Device Owner, the public key needs to be imported into the Signature Server. The import of an authentication key is an administrative operation, which has to be performed via the USB interface as follows:

1. Use the USB cable to connect the Signature Server to the Device Owner's PC.
2. Unlock the USB command interface as explained in the section *Connecting to the USB interface*.
3. Use the Signature Server administration tool to import the public authentication key, for example:

```
SigServerAdmin import -n Owner --in E:\owner_auth_key.pub --server usb:
    --auth-key E:\owner_auth_key.prv
```

The tool will ask for the password of the private authentication key file that was selected during generation in the previous section.

Note that the **--in** and **--auth-key** parameters in the example command above refer to the same authentication key pair. This ony applies when importing the first authentication key. For any later import operations, a previously imported authentication key pair must be used. In other words: The first Device Owner can enroll themselves with their own key, but any additional Administrators have to be enrolled by an existing Device Owner.

If the import was successful, the tool confirms the operation.

4.5.2 Setting up an Administrator account

Once the Device Owner has been set up, an Administrator account can be generated. For authentication purposes, another authentication key pair has to be created. The Administrator generates this key pair using the key generation tool and provides the public authentication key to the Device Owner. Assuming the Administrator's public authentication key is stored in **`E:\admin_auth_key.pub`**, the Device Owner uses the following command to create an Administrator account named **`Admin`**:

```
SigServerAdmin import -n Admin --in E:\admin_auth_key.pub --server usb:
    --auth-key E:\owner_auth_key.prv
```

The tool will ask for the password of the Device Owner's private authentication key file and confirm the operation once completed. The tasks in the following sections can be carried out by the Administrator.

4.5.3 Generating the first signing key pair

Next, a signing key pair has to be generated and placed in the Signature Server's key store. This key pair can later be used to generate signatures for firmware images. More details can be found in the section *Signing key pair*.

There are two ways to generate a signing key pair: It can either be generated on the Administrator's PC using the key generation tool and then imported into the Signature Server, or it can be generated on the Signature Server itself. The following sections explain both approaches.

Note that the properties of the signing key pair decide about the security level of the signatures that are generated later. Refer to the section *Choosing a signature scheme and its parameters* for guidance on this matter. In the following examples, a key pair for ECDSA signatures is generated, using a 256-bit curve.

4.5.3.1 Alternative 1: Generating a signing key pair on a PC

A new signing key pair can be generated with the key generation tool, for example:

```
KeyTool gen-ecdsa -s 256 -p --out E:\sign_key
```

This command may take some time, especially if large RSA keys are generated. It creates two key files, `sign_key.prv` (containing the private signing key) and `sign_key.pub` (containing the public signing key). The tool will ask for a password to protect the private key file. A strong password should be chosen for this purpose.

The private key file should be stored only in safe locations and backup copies should be created. For further recommendations on how to deal with key files, see *Security recommendations for keys and key files*.

Next, the private signing key has to be imported into the Signature Server. The import of a signature key is an administrative operation, which has to be performed via USB and while the USB command interface is unlocked. Use the Signature Server administration tool to import the private signing key, for example:

```
SigServerAdmin import -n SignKey1 --in E:\sign_key.pub --server usb:
    --auth-key E:\auth_key.prv
```

The tool will ask for the password of the private signature key file that was selected above and for the password of the Administrator's private authentication key file.

If the import was successful, the tool confirms the operation.

4.5.3.2 Alternative 2: Generating a signing key pair on the Server

Alternatively, the signing key pair can be generated on the Signature Server itself. This operation must be triggered though the Signature Server administration tool while the Signature Server is connected via USB and the USB command interface is unlocked. For example:


```
SigServerAdmin gen-ecdsa -n SignKey1 -s 256 --out E:\sign_key.pub
   --server usb: --auth-key E:\auth_key.prv
```

The tool will ask for the password of the Administrator's private authentication key file. The command may take some time to complete, especially if large RSA keys are generated. Once completed, the public signing key is exported to a key file on the Administrator's PC. This public key will be included in the bootloader for signature verification later.

If the key pair was successfully generated, the tool confirms the operation.

Key export

It is strongly recommended to keep backup copies of the private signing key (see *Security recommendations for keys and key files*). For this purpose, the generated key can be exported from the Signature Server via the USB interface. While the USB command interface is unlocked, the Signature Server administration tool can be used to export the key, for example:

```
SigServerAdmin export -n SignKey1 --out E:\sign_key.prv --server usb:
   --auth-key E:\auth_key.prv
```

The tool will ask for a password to protect the private signature key file with (choose a strong password here) and for the password of the Administrator's private authentication key file.

If the export was successful, the tool confirms the operation.

4.5.3.3 Protecting a private signing key from being exported

Regardless of how a private signing key was generated, it can be exported from the Signature Server by Administrators later, for example for backup purposes. If this is not desired, this behavior can be changed: A key can be prevented from ever being exported again by setting its "no-export" attribute.

> **Warning**
>
> Setting the "no-export" attribute on a key is final and cannot be undone.
>
> Only set this attribute on keys that are already backed up. Once the "no-export" attribute has been set, there is no way to ever export the key again from the Signature Server.

The "no-export" attribute of a private signing key can be set through the Signature Server administration tool while the Signature Server is connected via USB and the USB command interface is unlocked. For example:

```
SigServerAdmin set-no-export -n SignKey1 --server usb:
   --auth-key E:\auth_key.prv --auth-key-pass console
```

The tool will ask for the password of the Administrator's private authentication key file.

If the command was executed successfully, the tool confirms the operation.

4.5.4 Setting up a Release Manager user

A Release Manager user can use the Signature Server to generate signatures and is authenticated with a personal signing password. Refer to the sections *Release Manager role* and *Signing password* for more information.

A Release Manager user is set up by an Administrator through the Signature Server administration tool while the Signature Server is connected via USB and the USB command interface is unlocked. For example:

```
SigServerAdmin set-password -n RelMgr1 --server usb:
   --auth-key E:\auth_key.prv
```


The tool will ask for the password of the Administrator’s private authentication key file and for the new password to be set for the Release Manager. Choose a strong password here.

If the command was executed successfully, the tool confirms the operation.

4.5.5 Granting access to a signing key

The Release Managers need to be granted permission to use a signing key for signature generation when creating update packages. For each signing key, the Release Managers who are authorized to use it for signing are configured individually. The Administrator uses the Signature Server administration tool to grant permission to one or more Release Managers at a time. For example:

```
SigServerAdmin grant-permission -n SignKey1 -u RelMgr1 --server usb:
   --auth-key E:\auth_key.prv
```

The tool will ask for the password of the Administrator’s private authentication key file and confirm the successful execution of the command.

This step completes the initialization of the Signature Server. The *Ready* LED should now light up green (see *Status LEDs*), and the Signature Server is ready to generate signatures, as described in section *Creating firmware for the target*.

4.6 Installing updates

Update packages may be provided by SEGGER Microcontroller to enhance the functionality of the Signature Server. In order to install an update on the Signature Server, the update package is uploaded using the Signature Server administration tool by Device Owners or Administrators. The update is installed when the Signature Server is rebooted. See *Uploading firmware updates* for more information about the command which is used to upload updates. The Signature Server administration tool can be used to check the version number of the application firmware and the bootloader to verify that the update was installed as expected, see *Checking the status of a Signature Server*.

The key store, which contains both the signing keys and user authentication data, is unaffected by updates. However, maintaining a backup copy of signing keys is recommended in any case.

Chapter 5

Creating firmware for the target

This chapter describes how application firmware can be prepared for flashing into the target during manufacturing, and how both application firmware and bootloader update packages are created.

It is assumed that a functioning bootloader has been implemented as described in the previous sections, and that at least a “Hello, world” application firmware has been adapted to be startable by a bootloader.

5.1 Creating initial firmware for manufacturing

During manufacturing, both the bootloader and a signed application firmware image have to be programmed into the target's flash memory. Since the bootloader and the application firmware are in different regions of flash memory, they can either be programmed one after the other, or the two files containing them have to be combined into a single file. For simplicity, a two step process is assumed here.

The bootloader can be flashed into the target unchanged. The application firmware image has to be signed, otherwise the bootloader will not start it.

5.1.1 Alternative 1: Using a signing key file

The following command uses a private signing key stored in a local file to create an update package:

```
FirmwareTool sign
    -I ../Inc
    -k file:KeyFile1.prv
    -v 10
    --in application.elf --out application_signed_V10.srec
    -c BOOT_Conf.h
```

This command assumes that both the **application.elf** and **BOOT_Conf.h** files are in the current working directory, while the emBoot-Secure include directory, which contains the `BOOT_ConfDefaults.h`, is available under the relative path **../Inc**. The version number of the update package is set to **10** (see *Version numbers*).

With this command, the *firmware preparation tool*:

- asks for the password for the key stored in **KeyFile1.prv**,
- reads the application firmware image from **application.elf** and signs it,
- and writes the signed application firmware image to **application_signed_V10.srec**.

5.1.2 Alternative 2: Using the Signature Server

In order to sign an application firmware image with the Signature Server, the device must have been initialized as described in section *Initialization of the Signature Server*.

A Release Manager user can then connect to the Signature Server via Ethernet or USB (with the USB command interface unlocked, see *Connecting to the USB interface*) and use the firmware preparation tool to request a signature for a given application firmware image, for example:

```
FirmwareTool sign
    -I ../Inc
    -k sigs:SignKey1@usb:
    -v 10
    --in application.elf --out application_signed_V10.srec
    -c BOOT_Conf.h
```

This command assumes that both the **application.elf** and **BOOT_Conf.h** files are in the current working directory, while the emBoot-Secure include directory, which contains the `BOOT_ConfDefaults.h`, is available under the relative path **../Inc**. The version number of the update package is set to **10** (see *Version numbers*).

With this command, the firmware preparation tool:

- asks for the password of a Release Manager with permissions to use **SignKey1**,
- connects to a Signature Server via USB,
- requests a signature from the Signature Server for the application firmware image **application.elf**,
- receives the signature in response,
- combines both and writes the signed application firmware image to **application_signed_V10.srec**.

The signed application **`application_signed_V10.srec`** together with the bootloader firmware image can then be flashed into the target devices.

5.2 Creating a firmware update

The following command uses a private signing key stored in a local file to create a signed and encrypted update package:

```
FirmwareTool wrap
    -I ../Inc
    -k file:KeyFile1.prv
    --enc-key EncKeyFile1.sec
    -v 11
    --in application.elf --out Application_Update_V11.upd
    -c BOOT_Conf.h
```

This command assumes that required files **application.elf**, **BOOT_Conf.h**, **EncKeyFile1.prv** and **EncFile1.sec** are in the current working directory. The emBoot-Secure include directory, which contains the `BOOT_ConfDefaults.h`, must be available under the relative path **../Inc**. The version number of the update package is set to **11** (see *Version numbers*).

With this command, the firmware preparation tool:

- asks for the password for the encryption key stored in **EncKeyFile1.sec**,
- asks for the password for the private signing key stored in **KeyFile1.prv**,
- reads the application firmware image from **application.elf** and signs it,
- compresses the application firmware image, if compression is enabled in `BOOT_Conf.h`,
- encrypts the compressed data using the encryption key stored in **EncKeyFile1.sec**,
- and writes the completed update package to **Application_Update_V11.upd**.

If a signing key stored in a Signature Server should be used for signing instead of a local file, the **-k** parameter is adjusted as in the example from the previous section:

```
-k sigs:SignKey1@usb:
```

The resulting update file **Application_Update_V11.upd** can then be distributed to the target devices.

5.3 Creating a bootloader update

The process of creating a bootloader update package is very similar to that for producing an application update package. The main difference is the specification of the **--btl** parameter to inform the firmware preparation tool that a bootloader is packaged instead of an application. The firmware preparation tool needs this information to verify that the size and memory layout of the bootloader is compatible with the configuration provided in BOOT_Conf.h.

```
FirmwareTool wrap
    --btl
    -I ../Inc
    -k file:KeyFile1.prv
    --enc-key EncKeyFile1.sec
    -v 2
    --in bootloader.elf --out Bootloader_Update_V2.upd
    -c BOOT_Conf.h
```

This command assumes that required files **bootloader.elf**, **BOOT_Conf.h**, **EncKeyFile1.prv** and **EncFile1.sec** are in the current working directory. The emBoot-Secure include directory, which contains the BOOT_ConfDefaults.h, must be available under the relative path **../Inc**. The version number of the bootloader update package is set to **2** (see *Version numbers*).

The tool will request the passwords for both the encryption key file and the private signing key. If a Signature Server should be used instead for signing, the **-k** parameter can simply adapted to:

```
-k sigs:SignKey1@usb:
```

The resulting update file **Bootloader_Update_V2.upd** can then be distributed to the target devices.

Chapter 6

emBoot-Secure configuration switches

This chapter documents all compile time options for emBoot.

All compile-time switches and their default values can be found in the file `BOOT_ConfDefaults.h`. To change the default configuration of emBoot-Secure customized values for compile-time switches can be added to `BOOT_Conf.h`. The file `BOOT_ConfDefaults.h` should not be changed for easy updates of emBoot.

6.1 Configuration switches for firmware update behavior

6.1.1 SBTL_NUM_AREAS_UPDATE

Description

Defines the number of areas which can contain update packages.

The maximum number of update package areas is 16.

Definition

```
#define SBTL_NUM_AREAS_UPDATE    1
```

6.1.2 SBTL_NUM_AREAS_FAILSAFE

Description

Defines the number of areas which contain failsafe firmware images.

The maximum number of failsafe areas is 16.

Definition

```
#define SBTL_NUM_AREAS_FAILSAFE    0
```

6.1.3 SBTL_FIRMWARE_ALLOW_DOWNGRADE

Description

Defines whether downgrades, i.e. installation of firmware with a version number lower than the installed firmware, are allowed.

Definition

```
#define SBTL_FIRMWARE_ALLOW_DOWNGRADE    0
```

6.1.4 SBTL_BOOTLOADER_UPDATE_MODE

Description

Defines the bootloader updated mode, if any.

- `SBTL_BOOTLOADER_UPDATE_MODE_NONE`:

No bootloader updates allowed.

- `SBTL_BOOTLOADER_UPDATE_MODE_NORMAL`:

Normal bootloader updates: Bootloader update is loaded into RAM and replaces the currently installed bootloader.

- `SBTL_BOOTLOADER_UPDATE_MODE_BANKSWAP`:

Bank swap mode: For controllers with two flash banks. The bootloader in the first bank is responsible for starting the application and installing updates. When it needs to update itself, it updates the bootloader in the second bank. On success, it swaps banks and the bootloader in the second bank is started. It updates the bootloader in the first bank and reverts the bank swap, relinquishing control back to the bootloader in the first bank. In case the update fails due to power loss, one of the two bootloaders is still available to continue the update process.

Definition

```
#define SBTL_BOOTLOADER_UPDATE_MODE     SBTL_BOOTLOADER_UPDATE_MODE_NONE
```

6.1.5 SBTL_BOOTLOADER_ALLOW_DOWNGRADE

Description

Defines whether downgrades, i.e. installation of a bootloader with a version number lower than the installed bootloader, are allowed.

Definition

```
#define SBTL_BOOTLOADER_ALLOW_DOWNGRADE     0
```

6.1.6 SBTL_FW_VERSION_SYMBOL

Description

Name of the symbol in the firmware, where the version number is stored. The symbol must have an address inside read-only data of the firmware. The version number must be stored as U32 in little endian. If set, the FirmwareTool can automatically retrieve the version from a firmware image in ELF format.

Definition

```
#define SBTL_FW_VERSION_SYMBOL     ""
```

6.1.7 SBTL_BTL_VERSION_SYMBOL

Description

Name of the symbol in the bootloader, where the version number is stored. The symbol must have an address inside read-only data of the bootloader. The version number must be stored as U32 in little endian. If set, the FirmwareTool can automatically retrieve the version from a bootloader image in ELF format.

Definition

```
#define SBTL_BTL_VERSION_SYMBOL     ""
```

6.1.8 SBTL_FIRMWARE_FILL_BYTE

Description

Gaps in the firmware image will be filled with this value by the FirmwareTool when wrapping / signing an image.

Definition

```
#define SBTL_FIRMWARE_FILL_BYTE     0x00
```


6.2 Configuration switches for flash memory layout

The flash memory layout needs to be communicated to emBoot-Secure using configuration flags. For information about how to apply those to sample memory layouts, see *Sample memory configurations*.

Areas are described in terms of start addresses, sector sizes and sector counts. Each area can be composed of up to 10 separate regions with different sector sizes and counts, if needed. The configuration flags listed below describe the first region in an area (region 0). To configure regions 1 through 9, the numbers need to be appended to the configuration flags, for example, for region 7, the flag `SBTL_FIRMWARE_START_ADDR` is named `SBTL_FIRMWARE_START_ADDR7`.

Three areas can be described using configuration switches:

Application firmware image area

This area contains the active application firmware image. The bootloader erases this area and subsequently writes to it when it installs update packages. The respective configuration flags are:

Flag name	Description
`SBTL_FIRMWARE_START_ADDR`	Start address of the first firmware image region.
`SBTL_FIRMWARE_SECTOR_SIZE`	Size of sectors in the first firmware image region.
`SBTL_FIRMWARE_SECTOR_COUNT`	Number of sectors in the first firmware image region.

Bootloader target image area

For bootloader updates in normal update mode, the following configuration flags describe the part of flash memory containing the active bootloader. If bank swap update mode is enabled, this area represents the "shadow bootloader", see *Bootloader update in bank swap mode* for details. In both cases, a bootloader update is written into this region:

Flag name	Description
`SBTL_BOOTLOADER_TARGET_START_ADDR`	Start address of the first bootloader target image region.
`SBTL_BOOTLOADER_TARGET_SECTOR_SIZE`	Size of sectors in the first bootloader target image region.
`SBTL_BOOTLOADER_TARGET_SECTOR_COUNT`	Number of sectors in the first bootloader target image region.

Bootloader source image area

This area is only of relevance when bootloader updates in bank swap mode are active. In this case, this area describes the part of flash memory containing the active bootloader, which is copied to the target area during the second step of the update process.

Flag name	Description
`SBTL_BOOTLOADER_SOURCE_START_ADDR`	Start address of the first bootloader source image region.
`SBTL_BOOTLOADER_SOURCE_SECTOR_SIZE`	Size of sectors in the first bootloader source image region.
`SBTL_BOOTLOADER_SOURCE_SECTOR_COUNT`	Number of sectors in the first bootloader source image region.

6.3 Configuration switches related to algorithms

These configuration switches control which signature algorithm, hashing function, encryption and compression is used. They can be enabled by setting them to one in the file `BOOT_Conf.h`. Example:

```
#define SBTL_ALGO_SIGN_RSA_WITH_SHA512  1
```

At least one signature algorithm/hash pair has to be activated. Compression and encryption can be optionally enabled. Enabling the flags for compression and encryption only enable the bootloader to process compressed and encrypted update packages, they do not enforce that compression and encryption are actually used in update packages. If several algorithms are activated at the same time, the bootloader will be able to process update packages packaged using any of them. This is especially useful for the compression techniques. If application firmware evolves over time, another compression algorithm may provide better compression than when the project was released. Therefore, if enough code and RAM space are available for the bootloader, both compression methods should be activated to be future proof.

Activating several signature/hashing or compression algorithms at the same time increases code size, since several implementations have to be included in the final binary. The size of static RAM structures used by these algorithms will be the size be determined by the algorithm of each category with the highest RAM requirements. Static RAM reservations are shared by the algorithms, since they will not be running at the same time. See *Resource use and performance* for more information.

Switch	Explanation
Signature and hashing algorithms	
`SBTL_ALGO_SIGN_RSA_WITH_SHA256`	RSA signature with SHA256 hash
`SBTL_ALGO_SIGN_RSA_WITH_SHA512`	RSA signature with SHA512 hash
`SBTL_ALGO_SIGN_RSA_WITH_SHA1`	RSA signature with SHA1 hash
`SBTL_ALGO_SIGN_ECDSA_WITH_SHA256`	ECDSA signature with SHA256 hash
`SBTL_ALGO_SIGN_ECDSA_WITH_SHA384`	ECDSA signature with SHA384 hash
`SBTL_ALGO_SIGN_ECDSA_WITH_SHA512`	ECDSA signature with SHA512 hash
`SBTL_ALGO_SIGN_ECDSA_WITH_SHA1`	ECDSA signature with SHA1 hash
Compression algorithms	
`SBTL_ALGO_COMP_SMASH2`	Use SEGGER's SMASH-2 compression algorithm.
`SBTL_ALGO_COMP_SMASH2_T2`	Use SEGGER's SMASH-2 compression algorithm for better compression of ARM Thumb-2 instructions.
Encryption	
`SBTL_ALGO_ENC_AES_GCM`	Use AES in Galois/Counter Mode (AES-GCM).

6.3.1 SBTL_RSA_MAX_KEY_LENGTH

Description

Defines the maximum supported RSA key length in bits.

Should be set no larger than the largest RSA key used, since it is used to reserve static memory.

Definition

```
#define SBTL_RSA_MAX_KEY_LENGTH    4096
```


6.3.2 SBTL_ECDSA_MAX_KEY_LENGTH

Description

Defines the maximum supported ECDSA key length in bits.

Should be set no larger than the largest ECDSA key used, since it is used to reserve static memory.

Definition

```
#define SBTL_ECDSA_MAX_KEY_LENGTH     521
```

6.3.3 SBTL_COMP_SMASH_MAX_WINDOW_SIZE

Description

Configures the maximum window size which can be handled when decompressing SMASH-2 compressed images.

During decompression, a static RAM region with the same size as the compression window has to be used for housekeeping.

Actual window size in bytes is 256 * 2^SBTL_COMP_SMASH_MAX_WINDOW_SIZE.

Valid values are from 0 (= 256 byte window size) up to 6 (= 16KB window size). Default is 4 (= 4KB window size).

Definition

```
#define SBTL_COMP_SMASH_MAX_WINDOW_SIZE     4
```


6.4 Miscellaneous configuration switches

6.4.1 SBTL_FW_VERSION_SYMBOL

Description

Name of the symbol in the firmware, where the version number is stored. The symbol must have an address inside read-only data of the firmware. The version number must be stored as U32 in little endian. If set, the FirmwareTool can automatically retrieve the version from a firmware image in ELF format.

Definition

```
#define SBTL_FW_VERSION_SYMBOL    ""
```

6.4.2 SBTL_BTL_VERSION_SYMBOL

Description

Name of the symbol in the bootloader, where the version number is stored. The symbol must have an address inside read-only data of the bootloader. The version number must be stored as U32 in little endian. If set, the FirmwareTool can automatically retrieve the version from a bootloader image in ELF format.

Definition

```
#define SBTL_BTL_VERSION_SYMBOL    ""
```

6.4.3 SBTL_DEBUG

Description

emSecure can be configured to display debug messages and warnings to locate an error or potential problems. This can be useful for debugging. In a release (production) build of a target system, they are typically not required and should be switched off.

To output the messages, emSecure uses the logging routines contained in `BOOT_ConfigIO.c` which can be customized.

Definition

```
#define SBTL_DEBUG    0
```

6.4.4 SBTL_LOG_BUFFER_SIZE

Description

Maximum size of a debug / warning message (in characters) that can be output. A buffer of this size is created on the stack when a message is output.

Definition

```
#define SBTL_LOG_BUFFER_SIZE    200
```

6.4.5 SBTL_FIRMWARE_HEADER_SIZE

Description

Defines the size of the firmware header.

Do not change without consultation with SEGGER Microcontroller GmbH.

If this field is changed, linker scripts will also need to be adjusted. Update are not compatible between compilations with different header sizes.

Definition

```
#define SBTL_FIRMWARE_HEADER_SIZE    64
```

6.4.6 SBTL_ALLOW_UNSIGNED_INSTALLED_FIRMWARE

Description

During development of a bootloader and application pair, it can be helpful to allow the bootloader to start unsigned firmware.

Setting this flag to 1 will cause the bootloader to start unsigned firmware. Care has to be taken to disable this flag before shipping. A good way to enable it is via the build-system, but only for debug builds. In this way, the flag will not make it into release builds.

Definition

```
#define SBTL_ALLOW_UNSIGNED_INSTALLED_FIRMWARE    0
```

6.4.7 SBTL_RAM_FUNCTION

Description

Places a function in RAM. This is needed by functions used for manipulating flash memory. `SBTL_RAM_FUNCTION` needs to be placed in the line above the function. Depending on the compiler, all functions below this macro may be placed in RAM. The macro must be placed before each function to ensure cross-compiler compatibility.

For details, see *Placing functions in RAM*.

Chapter 7

Bootloader API reference

This chapter documents all functions and macros that can be used to build a customized bootloader.

7.1 Overview

Routine	Explanation
Bootloader library functions	
SBTL_Init()	Initializes the emBoot component.
SBTL_GetProductID()	Returns the product ID configured in BOOT_Conf.h.
SBTL_GetFirmwareInfo()	Read information from the header of a firmware or update file.
SBTL_CompareFirmwareInfo()	Compares the relevant entries of two SBTL_FIRMWARE_INFO structures.
SBTL_ScanFirmware()	Scans through the firmware areas provided by the user and determines whether to install new firmware.
SBTL_FindRescueFirmware()	Scans through the rescue firmware images for a valid image.
SBTL_UnwrapInit()	Initializes an SBTL_UNWRAP_CTX context.
SBTL_UnwrapReturnDataOnly()	Configures SBTL_Unwrap to only return firmware data without header / signature.
SBTL_Unwrap()	Reads data from an area and verifies the area's signature it.
SBTL_Verify()	Verify data from area.
SBTL_UpdateFirmware()	Updates the firmware.
SBTL_PrepareBootloaderUpdate()	Prepares for a bootloader update.
SBTL_PerformBootloaderUpdate()	Write a previously loaded bootloader image into flash and restart.
SBTL_LoadActiveBootloaderIntoRAM()	Loads the active bootloader into RAM.
SBTL_GetErrorText()	Decode an emBoot error code.
SBTL_FIRMWARE_INFO	Contains information read from a firmware header.
Callback functions to be provided by the application	
SBTL_GetBootloaderVersion()	Returns the version number of the currently installed bootloader.
SBTL_ReadArea()	Reads data from a firmware area.
SBTL_CloseArea()	Closes a firmware area after it has been read from using SBTL_ReadArea().
SBTL_GetRSAPublicKey()	Returns an RSA public key for signature verification.
SBTL_GetECPublicKey()	Returns an ECDSA public key for signature verification.
SBTL_GetAESKey()	Returns an AES key for firmware decryption.
SBTL_WriteFlash()	Writes firmware/bootloader data to the internal flash.
SBTL_GetBankSwapState()	Returns the currently active bank swap state.
SBTL_SetBankSwapState()	Sets bank swap as active/inactive.
SBTL_ResetHardware()	Resets the hardware.
SBTL_Logf()	Displays log information messages.

Routine	Explanation
SBTL_Panic()	Is called if the bootloader encounters a fatal error.
CRYPTO_X_Config()	Performs user configurable configuration for the cryptographic component.

7.2 Bootloader library functions

7.2.1 SBTL_Init()

Description

Initializes the emBoot component.

Must be called before any other `SBTL_…` function is called.

This function initializes the cryptographic component emCrypt, which calls the user-defined `CRYPTO_X_Config()` function.

Prototype

```
void SBTL_Init(void);
```


7.2.2 SBTL_GetProductID()

Description

Returns the product ID configured in `BOOT_Conf.h`.

The product ID is a 32 bit unsigned integer computed from the `SBTL_VENDOR_NAME` and `SBTL_PRODUCT_NAME` defines.

Prototype

```
U32 SBTL_GetProductID(void);
```

Return value

Product ID.

7.2.3 SBTL_GetFirmwareInfo()

Description

Read information from the header of a firmware or update file.

This function does not check the validity of the data in the area. It is intended to be used to quickly identify areas which may contain valid firmware, for example while a bootloader is searching for updates. This allows a complete, slow signature verification to only be performed if the firmware in this area is of interest for installation or launching. Use `SBTL_Verify()` to verify the signature of the area.

Prototype

```
int SBTL_GetFirmwareInfo(unsigned             Area,
                         SBTL_FIRMWARE_INFO * pInfo);
```

Parameters

Parameter	Description
`Area`	Index of non volatile memory area to read from.
`pInfo`	Pointer to a structure, that receives the information.

Return value

= 0 Success. Information stored into `pInfo`.
≠ 0 Error. No valid header in the given area.

7.2.4 SBTL_CompareFirmwareInfo()

Description

Compares the relevant entries of two SBTL_FIRMWARE_INFO structures.

Compares the Version, ProductID and IsBootloaderUpdate fields.

Prototype

```
int SBTL_CompareFirmwareInfo(const SBTL_FIRMWARE_INFO * pInfo1,
                             const SBTL_FIRMWARE_INFO * pInfo2);
```

Parameters

Parameter	Description
pInfo1	Pointer to the first structure.
pInfo2	Pointer to the second structure.

Return value

= 0 Success. Relevant information in the two structures is identical.
≠ 0 Information in both structures does not match.

7.2.5 SBTL_ScanFirmware()

Description

Scans through the firmware areas provided by the user and determines whether to install new firmware.

If no suitable firmware update is found, this function will return information about the currently installed firmware (`pFirmwareInfo->Area = SBTL_MEM_AREA_FIRMWARE`). The caller should proceed to launch this firmware.

For more information about this function, see emBoot-Secure documentation.

Prototype

```
int SBTL_ScanFirmware(SBTL_FIRMWARE_INFO * pFirmwareInfo,
                      SBTL_WORK_MEMORY   * pWorkMemory,
                      SBTL_FLASH_BUFF    * pFlashBuffer);
```

Parameters

Parameter	Description
`pFirmwareInfo`	Pointer to an `SBTL_FIRMWARE_INFO` structure to put the information about the firmware to install into.
`pWorkMemory`	Pointer to a memory region large enough for an `SBTL_WORK_MEMORY` structure (uninitialized).
`pFlashBuffer`	Pointer to a memory region large enough for an `SBTL_FLASH_BUFF` structure (uninitialized).

Return value

= 1 Success, `pFirmwareInfo` contains information about the firmware to install.
< 0 Error, neither found an update nor is there a verifiable firmware installed.

Additional information

The scan behavior is controlled by the following macros defined in `BOOT_Conf.h`:

- *`SBTL_FIRMWARE_ALLOW_DOWNGRADE`*: Specifies whether firmware images with a version number lower than the currently installed firmware may be installed.
- *`SBTL_BOOTLOADER_UPDATE_MODE`*: If bootloader updates are enabled, this function will scan for bootloader updates.
- *`SBTL_BOOTLOADER_ALLOW_DOWNGRADE`*: Specifies whether downgrades of the bootloader are allowed.
- *`SBTL_ALLOW_UNSIGNED_INSTALLED_FIRMWARE`*: Can be set to 1 for debugging purposes, when the developer needs to flash firmware images repeatedly during development and needs to verify the transition from the bootloader to the application. USE WITH CAUTION!

7.2.6 SBTL_FindRescueFirmware()

Description

Scans through the rescue firmware images for a valid image.

The scan checks the rescue firmware images starting at the lowest index. It stops at the first valid image and fills *pFirmwareInfo with information about that image.

Prototype

```
int SBTL_FindRescueFirmware(SBTL_FIRMWARE_INFO * pFirmwareInfo,
                            SBTL_WORK_MEMORY   * pWorkMemory,
                            SBTL_FLASH_BUFF    * pFlashBuffer);
```

Parameters

Parameter	Description
pFirmwareInfo	Pointer to an SBTL_FIRMWARE_INFO structure to put the information about the firmware to install into.
pWorkMemory	Pointer to a memory region large enough for an SBTL_WORK_MEMORY structure (uninitialized).
pFlashBuffer	Pointer to a memory region large enough for an SBTL_FLASH_BUFF structure (uninitialized).

Return value

= 0 Success, pFirmwareInfo contains information about the firmware to install
≠ 0 Error, could not find a valid rescue firmware image.

7.2.7 SBTL_UnwrapInit()

Description

Initializes an SBTL_UNWRAP_CTX context.

Use this function to prepare an SBTL_UNWRAP_CTX to read data from or verify area with index Area.

Prototype

```
void SBTL_UnwrapInit(SBTL_UNWRAP_CTX    * pCtx,
                     unsigned             Area,
                     SBTL_FIRMWARE_INFO * pInfo,
                     SBTL_WORK_MEMORY   * pWorkMem);
```

Parameters

Parameter	Description
pCtx	Pointer to an SBTL_UNWRAP_CTX context to be initialized.
Area	Non volatile memory area to read from.
pInfo	Structure to store information about the firmware. Filled by the function SBTL_Unwrap().
pWorkMem	Pointer to working memory to be used by the verification / decompression / decryption functions (uninitialized).

7.2.8 SBTL_UnwrapReturnDataOnly()

Description

Configures `SBTL_Unwrap` to only return firmware data without header / signature.

`SBTL_Unwrap` will still check the signature in this mode.

Prototype

```
void SBTL_UnwrapReturnDataOnly(SBTL_UNWRAP_CTX * pCtx);
```


7.2.9 SBTL_Unwrap()

Description

Reads data from an area and verifies the area's signature it.

In order to read data from an area with this function, reserve memory for an `SBTL_UNWRAP_CTX` context. Then provide the index of the area to `SBTL_UnwrapInit()` which will initialize the `SBTL_UNWRAP_CTX`. Then call this function to read the data from the area and verify its signature.

After all data has been read, `SBTL_CloseArea()` has to be called to close the area.

If only the signature of an area is to be verified without any need for the data, use `SBTL_Verify()` instead.

Prototype

```
int SBTL_Unwrap(SBTL_UNWRAP_CTX * pCtx,
                U32               NumBytesReq,
                U8              * pData,
                U32             * pNumBytesReceived);
```

Parameters

Parameter	Description
`pCtx`	Pointer to `SBTL_UNWRAP_CTX` context, initialized with `SBTL_UnwrapInit()`.
`NumBytesReq`	Maximum number of bytes to be read. Must be ≥ `SBTL_FIRMWARE_HEADER_SIZE` on first call.
`pData`	Buffer to receive the data.
`pNumBytesReceived`	Number of bytes returned are stored here.

Return value

= 0	Success. Length of data read is stored in *`pNumBytesReceived` (may be 0). Call again for more data.
< 0	Error
> 0	End of data reached and signature verified successfully. *`pNumBytesReceived` contains the length of data read (may be 0).

7.2.10 SBTL_Verify()

Description

Verify data from area.

In order to verify an area with this function, reserve memory for an `SBTL_UNWRAP_CTX` context. Then provide the index of the area to `SBTL_UnwrapInit()` which will initialize the `SBTL_UNWRAP_CTX`. Then call this function to verify its signature.

After the signature has been verified, `SBTL_CloseArea()` has to be called to close the area.

In contrast to `SBTL_Unwrap()`, this function will not return any data from the area, but merely verify its signature.

Prototype

```
int SBTL_Verify(SBTL_UNWRAP_CTX * pCtx,
                SBTL_FLASH_BUFF * pFlashBuffer);
```

Parameters

Parameter	Description
`pCtx`	Pointer to UNWRAP context, initialized with `SBTL_UnwrapInit()`.
`pFlashBuffer`	Pointer to a buffer to store temporary data during verification process.

Return value

< 0 Error
> 0 Signature verified successfully.

7.2.11 SBTL_UpdateFirmware()

Description

Updates the firmware.

Prototype

```
int SBTL_UpdateFirmware(SBTL_FIRMWARE_INFO * pFirmwareInfo,
                        SBTL_WORK_MEMORY   * pWorkMemory,
                        SBTL_FLASH_BUFF    * pFlashBuffer);
```

Parameters

Parameter	Description
pFirmwareInfo	Pointer to an SBTL_FIRMWARE_INFO structure containing information about the firmware to be installed.
pWorkMemory	Pointer to a memory region large enough for an SBTL_WORK_MEMORY structure (uninitialized).
pFlashBuffer	Pointer to a memory region large enough for an SBTL_FLASH_BUFF structure (uninitialized).

Return value

= 0 Success, firmware has been updated.
≠ 0 Error, flashing the firmware failed.

7.2.12 SBTL_PrepareBootloaderUpdate()

Description

Prepares for a bootloader update.

Loads the bootloader image from the area indicated by `pFirmwareInfo` into RAM and verifies it. On success, the caller can call `SBTL_PerformBootloaderUpdate()` to write the image into flash memory.

Prototype

```
int SBTL_PrepareBootloaderUpdate(SBTL_FIRMWARE_INFO * pFirmwareInfo,
                                 SBTL_WORK_MEMORY   * pWorkMemory,
                                 SBTL_FLASH_BUFF    * pFlashBuffer);
```

Parameters

Parameter	Description
`pFirmwareInfo`	Pointer to an `SBTL_FIRMWARE_INFO` structure containing information about the bootloader to install.
`pWorkMemory`	Pointer to a memory region large enough for an `SBTL_WORK_MEMORY` structure (uninitialized).
`pFlashBuffer`	Pointer to a memory region large enough for an `SBTL_FLASH_BUFF` structure (uninitialized).

Return value

= 0 Bootloader loaded into RAM, caller can proceed with flashing.
≠ 0 Error while loading the bootloader into RAM.

7.2.13 SBTL_PerformBootloaderUpdate()

Description

Write a previously loaded bootloader image into flash and restart.

Before calling the function, the caller must deinitialize any hardware parts which need a graceful shutdown before a reboot.

This function has two modes of operation:

- Normal update mode Use `SBTL_PrepareBootloaderUpdate()` to load the image into an `SBTL_FLASH_BUFF` structure, which this function will write to flash memory.

After writing the bootloader update, this function calls `SBTL_ResetHardware()` in order to start the updated bootloader. If writing the bootloader image to flash fails, the system will halt.

In this mode, this function will not return.

- Bank swap update mode

Prototype

```
SBTL_RAM_FUNCTION int SBTL_PerformBootloaderUpdate(SBTL_FLASH_BUFF * pFlashBuffer);
```

Parameters

Parameter	Description
`pFlashBuffer`	Pointer to an `SBTL_FLASH_BUFF` which contains the bootloader.

Return value

= 0 Bootloader loaded into RAM, caller can proceed with flashing.
≠ 0 Error while loading the bootloader into RAM.

7.2.14 SBTL_LoadActiveBootloaderIntoRAM()

Description

Loads the active bootloader into RAM.

This function is only needed during a bootloader update in bank swap mode, after the bootloader on bank 1 has updated the shadow bootloader on bank 2 and activated it. This function then loads the updated shadow bootloader from bank 2 into RAM, such that it can be used to update the bootloader in bank 1.

Prototype

```
void SBTL_LoadActiveBootloaderIntoRAM(SBTL_FLASH_BUFF * pFlashBuffer);
```

Parameters

Parameter	Description
pFlashBuffer	Pointer to a memory region large enough for an SBTL_FLASH_BUFF structure (uninitialized).

7.2.15 SBTL_GetErrorText()

Description

Decode an emBoot error code.

Prototype

```
char *SBTL_GetErrorText(int Status);
```

Parameters

Parameter	Description
Status	Code to decode.

Return value

Non-zero pointer to status description.

7.2.16 SBTL_FIRMWARE_INFO

Description

Contains information obtained from a firmware header.

Type definition

```
typedef struct {
  U32      ProductID;
  U32      Version;
  U8       Area;
  I8       IsBootLoaderUpdate;
  union {
    PTR_ADDR Addr;
    void   (*Jump)(void);
  } EntryPoint;
} SBTL_FIRMWARE_INFO;
```

Member	Description
ProductID	Product ID the firmware / update package belongs to.
Version	Version number of the firmware / update package.
Area	Area where the firmware / update package is stored. See SBTL_MEM_AREA_... macros.
IsBootLoaderUpdate	Value 1 if the package is a bootloader update, 0 otherwise.
EntryPoint.Addr	Address of firmware entry point
EntryPoint.Jump()	Function pointer for jumping to firmware entry point.

7.3 Functions to be provided by the application

7.3.1 SBTL_GetBootloaderVersion()

Description

Returns the version number of the currently installed bootloader.

This function only has to be implemented if the bootloader is configured to update itself (see `SBTL_BOOTLOADER_UPDATE_MODE`).

If the version number which is returned here is stored in a const static variable, the firmware preparation tool can read it from the resulting ELF file. See `SBTL_BTL_VERSION_SYMBOL` for more information.

Prototype

```
U32 SBTL_GetBootloaderVersion(void);
```

Return value

Version number of the currently installed bootloader.

7.3.2 SBTL_ReadArea()

Description

Reads data from a firmware area.

Constraints:

- Only one area will be read at a time. After an area has been read, `SBTL_CloseArea()` will be called, even if the initial reading from the area failed.
- Reading of an area will be sequential and starts at offset zero.
- This function may return fewer bytes than requested, even if more data is available in the area. In that case, the caller will adjust the offset accordingly and request the remaining data until either all required data has been read or the end of the area is reached (signaled by a return value of 0).

Prototype

```
int SBTL_ReadArea(unsigned   Area,
                  U32        Offset,
                  unsigned   BuffSize,
                  U8       * pBuff);
```

Parameters

Parameter	Description
`Area`	Index of the area, compatible with the numbering scheme used by the `SBTL_MEM_AREA_...` macros.
`Offset`	`Offset` from which to obtain data.
`BuffSize`	Size of the buffer pointed to by `pBuff`.
`pBuff`	Pointer to a buffer for the obtained data.

Return value

> 0	Amount of bytes read from the area and copied into buffer.
= 0	End of file reached.
< 0	Error reading from the area.

7.3.3 SBTL_CloseArea()

Description

Closes a firmware area after it has been read from using `SBTL_ReadArea()`.

Prototype

```
void SBTL_CloseArea(unsigned Area);
```

Parameters

Parameter	Description
`Area`	Index of the area to be closed, with the same conventions as used with `SBTL_ReadArea()`.

7.3.4 SBTL_GetRSAPublicKey()

Description

Returns an RSA public key for signature verification.

Prototype

```
CRYPTO_RSA_PUBLIC_KEY *SBTL_GetRSAPublicKey(unsigned KeyIndex);
```

Parameters

Parameter	Description
KeyIndex	Index of the requested key. Can be ignored, if the bootloader only contains a single RSA key.

Return value

Pointer to a structure containing the public RSA key.

7.3.5 SBTL_GetECPublicKey()

Description

Returns an ECDSA public key for signature verification.

Prototype

```
CRYPTO_ECDSA_PUBLIC_KEY *SBTL_GetECPublicKey(unsigned KeyIndex);
```

Parameters

Parameter	Description
KeyIndex	Index of the requested key. Can be ignored, if the bootloader only contains a single ECDSA key.

Return value

Pointer to a structure containing the public ECDSA key.

7.3.6 SBTL_GetAESKey()

Description

Returns an AES key for firmware decryption.

Prototype

```
U8 *SBTL_GetAESKey(unsigned   KeyIndex,
                   unsigned * pKeySize);
```

Parameters

Parameter	Description
KeyIndex	Index of the requested key. Can be ignored, if the bootloader only contains a single AES key.
pKeySize	Returns the size of the key in bytes.

Return value

Pointer to the AES key.

7.3.7 SBTL_WriteFlash()

Description

Writes firmware/bootloader data to the internal flash.

The caller must ensure that `FlashAddr` and `Size` match the boundary conditions of the flash memory regarding sector size and alignment.

This function must take care of the erasure of flash memory, depending on the flash geometry. For example, if the flash can be erased at a sector level, it has to erase each sector as it is written. If a number of sectors have to be erased simultaneously due to hardware constraints, the function has to perform erasure when the `FlashAddr` points to the beginning of the first sector.

This function is also responsible for preparing the microcontroller for flash programming. It is strongly advised to disable interrupts during the writing process unless otherwise noted in the microcontroller's documentation. Other requirements may included adjustment of clock core/bus clock frequencies and supply voltages.

Note that this function may need to be placed in RAM because it runs while flash memory writing/erasing functions are executed. Be sure not to call any functions which reside in flash while the flash memory is being programmed!

If bootloader flashing in normal mode is enabled, this function must be placed in RAM.

Error codes can either be chosen from the list of `SBTL_ERR_…` codes or custom codes can be used for flash-algorithm specific errors.

Prototype

```
SBTL_RAM_FUNCTION int SBTL_WriteFlash(PTR_ADDR   FlashAddr,
                                      unsigned   Size,
                                      U8       * pData);
```

Parameters

Parameter	Description
`FlashAddr`	Target address in flash memory.
`Size`	`Size` of buffer pointed to by `pData`.
`pData`	Pointer to buffer to be written into flash.

Return value

= 0 Success
≠ 0 Error

7.3.8 SBTL_GetBankSwapState()

Description

Returns the currently active bank swap state.

This function is only needed when the bank swap bootloader update mode is used (`SBTL_BOOTLOADER_UPDATE_MODE_BANKSWAP`).

Prototype

```
int SBTL_GetBankSwapState(void);
```

Return value

0	Banks are not swapped
1	Banks are swapped

7.3.9 SBTL_SetBankSwapState()

Description

Sets bank swap as active/inactive.

It is expected that the changes in the bank swap state are only applied after a reboot.

This function is only needed when the bank swap bootloader update mode is used (`SBTL_BOOTLOADER_UPDATE_MODE_BANKSWAP`).

Prototype

```
int SBTL_SetBankSwapState(int EnableSwap);
```

Parameters

Parameter	Description
`EnableSwap`	Set to 1 to active bank swap.

Return value

= 0 Success
< 0 Error

7.3.10 SBTL_ResetHardware()

Description

Resets the hardware. Used after a new firmware/bootloader has been written to flash memory.

If bootloader flashing in normal mode is enabled, this function and any functions it calls must be placed in RAM.

This function must never return.

Prototype

```
SBTL_RAM_FUNCTION void SBTL_ResetHardware(void);
```


7.3.11 SBTL_Logf()

Description

Displays log information messages.

Do not call this function directly, use the `SBTL_LOG` macro instead, since it will disable logging in release mode.

This function is expected to produce a linebreak in the log after each message.

Prototype

```
void SBTL_Logf(const char * sFormat,
                           ...);
```

Parameters

Parameter	Description
`sFormat`	Message string with optional format specifiers.

7.3.12 SBTL_Panic()

Description

Is called if the bootloader encounters a fatal error.

This function must not return.

Prototype

```
void SBTL_Panic(const char * pErrorString);
```

Parameters

Parameter	Description
pErrorString	Pointer to a string holding the error message.

7.3.13 CRYPTO_X_Config()

Description

Performs user configurable configuration for the cryptographic component.

The function `SBTL_Init()` initializes the cryptographic component emCrypt, which in turn calls this function during initialization.

The modular exponentiation function to be used for public key processing has to be configured here. Since the public keys are rather small, a reasonably fast function with low memory requirements is `CRYPTO_MPI_ModExp_Basic_Fast`.

The `CRYPTO_MPI_ModExp_Basic_Fast` can be enabled like this:

```
CRYPTO_MPI_SetPublicModExp(CRYPTO_MPI_ModExp_Basic_Fast);
```

For more options, see the subsection *Exponentiation* of the *Multiprecision integers* section of the CRYPTO library documentation.

Prototype

```
void CRYPTO_X_Config(void);
```


7.4 Utility macros for area indices

In the emBoot-Secure library, update packages and installed firmware images are organized into areas, which are addressed via indices. This section explains the utility macros which can be used to represent and check region numbers both in callback functions and data returned from library functions.

SBTL_MEM_AREA_FIRMWARE

This index represents the currently installed application firmware image. When information from this area is requested from the callback function `SBTL_ReadArea()`, it can simply be copied from the flash region into the RAM buffer supplied by the caller. The `SBTL_ScanFirmware()` function will populate the supplied `SBTL_FIRMWARE_INFO` structure with information about this region when the currently installed application firmware image is to be launched by the bootloader.

SBTL_MEM_AREA_UPDATE(n)

Returns the area index for the n-th area which may contain update packages for either the application firmware or the bootloader. In the `SBTL_ReadArea()` function, it can be used to match the index supplied by the caller against the manufacturer defined update package storage locations. There can be up to 16 areas containing update packages.

SBTL_MEM_AREA_FAILSAFE(n)

Returns the area index for the n-th area which may failsafe application update packages. In the `SBTL_ReadArea()` function, it can be used to match the index supplied by the caller against the manufacturer defined update package storage locations. There can be up to 16 areas containing failsafe update packages.

SBTL_MEM_AREA_IS_UPDATE(x)

Returns 1 if the index points to an area containing an update package, otherwise 0.

SBTL_MEM_AREA_IS_FAILSAFE(x)

Returns 1 if the index points to an area containing a failsafe update package, otherwise 0.

Chapter 8

The key generation tool

The `KeyTool` can be used to generate and convert cryptographic keys needed by emBoot.

8.1 Key generation tool invocation

`KeyTool.exe`/`KeyTool` is a command line tool for PC (Windows / Linux).

General command line syntax:

`KeyTool` `<operation>` `<options>`...

The supported `<operation>`s are explained in the following sections.

General options	
`--version`	Print the version of the key generation tool and the underlying emCrypt library.
`-q`	Quiet execution. Don't output messages, except errors.
`--pass` `<pwd-opt>`	Provide a password.

8.1.1 Providing passwords

Private and secret keys may be stored password encrypted in files. To access such a file a password must be provided. By default, the key generation tool prompts for every password required and reads the password from the console. To use the key generation tool in batch scripts, passwords may be provided in other ways using the **`--pass`** option.

`<pwd-opt>`	
`pass:``<password>`	The actual password is `<password>`, given on the command line. Note that the command line is visible to other processes and users, for example the `ps` utility on Linux.
`<password>`	The actual password is `<password>`, given on the command line. Note that the command line is visible to other processes and users, for example the `ps` utility on Linux.
`console`	Prompt user to enter password on the console.
`file:``<file>`	Read the password from the file `<file>`. Only the first line of the file (until the first newline character) is used as the password.
`env:``<var>`	Obtain the password from the environment variable `<var>`. Note that environment variables may be visible to other processes and users.

8.2 Generating an ECDSA key pair (KeyTool)

The following command generates an ECDSA key pair and writes it into files. A file for the private key another one for the public key are created. The private key file can be encrypted with a password.

Either the name of the elliptic curve to be used can be specified or the key size in bits. If the key size is specified, Brainpool curves are used by default. To use curves provided by NIST, the **--NIST** parameter has to be specified.

```
KeyTool gen-ecdsa [-s <keysize> [--nist] | --curve <curve-name>]
    [--out <path>] [-p | --pass <pwd-opt>]
```

Options	
-s <keysize>	Bit size of the ECDSA key to be generated. This option determines the elliptic curve which is used. Possible values are 256, 320, 384, 512 or 521. Default is 256.
--curve <name>	Specify the elliptic curve to be used by name.
--out <path>	Pathname of the key files to be created. For the private key file `.prv` will be appended to the pathname and `.pub` for the public key file. The default file names are `emBootSecureKey.prv` and `emBootSecureKey.pub`.
-p	Shorthand for <--pass console>, i.e., encrypt the private key file with a password that is read from the console.
--pass <pwd-opt>	Encrypt the private key file with a password provided as described under *Providing passwords*.

Supported elliptic curves are:

- brainpoolP256r1
- brainpoolP320r1
- brainpoolP384r1
- brainpoolP512r1
- secp256r1
- secp384r1
- secp521r1

For an up to date list of supported curves, use:

```
KeyTool list-curves
```

> **Note**
>
> Alternatively, ECDSA key pairs can be generated on a Signature Server, see *Generating an ECDSA key pair (Signature Server)*.

8.3 Generating an RSA key pair (KeyTool)

The following command generates an RSA key pair and writes it into files. One file for the private key another one for the public key are created. The private key file can be encrypted with a password.

KeyTool gen-rsa [**-s** <keysize>] [**--out** <path>] [**-p** | **--pass** <pwd-opt>]

Options	
-s <keysize>	Bit size of the RSA key to be generated. Supported values are 2048, 3072, 4096, 6144 or 8192. Default is 3072.
--out <path>	Pathname of the key files to be created. For the private key file `.prv` will be appended to the pathname and `.pub` for the public key file. The default file names are `emBootSecureKey.prv` and `emBootSecureKey.pub`.
-p	Shorthand for <--pass console>, i.e., encrypt the private key file with a password that is read from the console.
--pass <pwd-opt>	Encrypt the private key file with a password provided as described under *Providing passwords*.

Note

RSA keys with large sizes may take a long time to generate.

Note

Alternatively, RSA key pairs can be generated on a Signature Server, see *Generating an RSA key pair (Signature Server)*.

8.4 Generating an AES key

The following command generates an AES key and writes it into a file. The key file can be encrypted with a password.

KeyTool gen-aes [**-s** <keysize>] [**--out** <path>] [**-p** | **--pass** <pwd-opt>]

Options	
-s <keysize>	Bit size of the AES key to be generated. Possible values are 128, 192 or 256. Default is 128.
--out <path>	Pathname of the key file to be created. The extension `.sec` will be appended to the file name. The default file name is `emBootSecureKey.sec`.
-p	Shorthand for <--pass console>, i.e., encrypt the key file with a password that is read from the console.
--pass <pwd-opt>	Encrypt the key file with a password provided as described under *Providing passwords*.

8.5 Converting an RSA key to C

The following command reads a public RSA key from either a public or private RSA key file and converts it into a C file, that can be compiled into the bootloader project. The resulting C file will contain the definition of the `SBTL_GetRSAPublicKey()` function, which is used by the emBoot-Secure library to obtain public keys.

If a bootloader needs to be able to work with several public keys for verification of signatures, they have to be merged into a single C file by specifying them to a single invocation of the key generation tool. The order of the keys is important, because the keys are retrieved from the resulting `SBTL_GetRSAPublicKey()` function using an index, which is based on the order in which the keys are provided to the key generation tool. When an update package is signed using the *firmware preparation tool*, that index has to be provided such that it can be recorded inside the update package. Keys must not be reordered, because otherwise, there will be a key index mismatch between the bootloader and the update package, preventing signature validation.

KeyTool rsa2c --in <key-file> [**--in** <key-file>]… [**--out** <path>]

Options	
--in <key-file>	Input file that contains either a public or private RSA key. Multiple key files can be specified. All Keys will be output to a single C file. If any encrypted private key file is given, a password is required. See *Providing passwords*.
--out <path>	Pathname of the C file to be created. If <path> does not already have a file extension, then `.c` will be appended. If this option is not specified, the C code is send to `stdout`.

8.6 Converting an ECDSA key to C

The following command reads a public ECDSA from either a public or private ECDSA key file and converts it into a C file, that can be compiled into the bootloader project. The resulting C file will contain the definition of the `SBTL_GetECPublicKey()` function, which is used by the emBoot-Secure library to obtain public keys.

If a bootloader needs to be able to work with several public keys for verification of signatures, they have to be merged into a single C file by specifying them to a single invocation of the key generation tool. The order of the keys is important, because the keys are retrieved from the resulting `SBTL_GetECPublicKey()` function using an index, which is based on the order in which the keys are provided to the key generation tool. When an update package is signed using the *firmware preparation tool*, that index has to be provided such that it can be recorded inside the update package. Keys must not be reordered, because otherwise, there will be a key index mismatch between the bootloader and the update package, preventing signature validation.

`KeyTool ecdsa2c --in` `<key-file>` [**`--in`** `<key-file>`]… [**`--out`** `<path>`]

Options	
`--in` `<key-file>`	Input file that contains either a public or private ECDSA key. Multiple key files can be specified. All Keys will be output to a single C file. If any encrypted private key file is given, a password is required. See *Providing passwords*.
`--out` `<path>`	Pathname of the C file to be created. If `<path>` does not already have a file extension, then `.c` will be appended. If this option is not specified, the C code is send to `stdout`.

8.7 Converting an AES key to C

The following command reads an AES key from a key file and converts it into a C file, that can be compiled into the bootloader project. Note that the C file will contain the AES key in plain text. The C file will contain a definition of the `SBTL_GetAESKey()` function, which is called by the emBoot-Secure library when the AES key is needed for decrypting an update package.

If a bootloader needs to be able to work with several encryption keys, they have to be merged into a single C file by specifying them to a single invocation of the key generation tool. The order of the keys is important, because the keys are retrieved from the resulting `SBTL_GetAESKey()` function using an index, which is based on the order in which the keys are provided to the key generation tool. When an update package is encrypted using the *firmware preparation tool*, that index has to be provided such that it can be recorded inside the update package. Keys must not be reordered, because otherwise, there will be a key index mismatch between the bootloader and the update package, preventing decryption of the update package.

`KeyTool aes2c --in` `<key-file>` [**`--in`** `<key-file>`]… [**`--out`** `<path>`]

Options	
`--in` `<key-file>`	Input file that contains the AES key. Multiple key files can be specified. All Keys will be output to a single C file. If any encrypted key file is given, a password is required. See *Providing passwords*.
`--out` `<path>`	Pathname of the C file to be created. If `<path>` does not already have a file extension, then `.c` will be appended. If this option is not specified, the C code is sent to `stdout`.
`--pass` `<pwd-opt>`	Password for decryption of the AES key, provided as described under *Providing passwords*.

8.8 Changing the passwords of private key files

The key generation tool can read a private key from a file and save it to another file with either a changed password or in unencrypted form using the **change-password** operation. If no **--pass**/**--newpass** arguments are specified, the respective passwords are read from the terminal, unless the **--unencrypted** parameter is supplied, which will cause the key to be written to the new file without encryption.

KeyTool change-password --in <key-in-file> **--out** <key-out-file>
[**--pass**] [**--newpass**] [**--unencrypted**]

Options	
--in <key-in-file>	Input file that contains the AES key. Multiple key files can be specified. All Keys will be output to a single C file. If any encrypted key file is given, a password is required. See *Providing passwords*.
--out <key-out-file>	Pathname of the C file to be created. If <path> does not already have a file extension, then `.c` will be appended. If this option is not specified, the C code is sent to `stdout`.
--pass <pwd-opt>	Password for decryption of the key in the input file, provided as described under *Providing passwords*.
--newpass <pwd-opt>	Password for encryption of the key in the output file, provided as described under *Providing passwords*.
--unencrypted	Store the key without encryption in the output file.

Chapter 9

The firmware preparation tool

The firmware preparation tool (`FirmwareTool`) is used to process firmware images and turn them into update packages for distribution or signed firmware images for initial flashing.

The process of creating an update package from a firmware image is called "wrapping", reflecting the separate layers of the update package (signing, compression, encryption). The resulting binary files are suitable for distribution to devices in the field.

During device manufacturing, both the bootloader and a signed application firmware image have to be flashed into the target device. Since the bootloader will only launch signed firmware, a signature has to be added to the initial application firmware as well. It is rendered into a Motorola S-Record file which can be processed as usual by the firmware flashing utilities.

9.1 Supported file formats

Usual file extension	File input	File output	Description
`.srec`	✓	✓	Motorola S-Record format
`.hex`	✓		Intel HEX file format
`.elf`	✓		Executable and Linking Format (ELF)
`.bin`	✓		Binary files
`.upd`		✓	SEGGER wrapped / signed firmware format (Update package)

Motorola S-Record file format

S-Record files produced by linkers store firmware code and data in hexadecimal format along with their designated addresses on the target hardware. They also contain the address of the firmware entry point (usually the reset handler or the main() function). The FirmwareTool can read these files to create either update packages or signed firmware images.

Signed firmware images are written to S-Record files, in order to be consumed by firmware flashing tools.

Intel HEX file format

HEX files are similar to S-Record files and also contain code and data in hexadecimal format. The FirmwareTool can only read this file format.

ELF file format

The ELF file format contains code and data in binary form, along with some symbol names. The FirmwareTool extracts the relevant parts and deduces both the firmware entry point address as well as the base address of the firmware. The FirmwareTool can only read this file format.

Binary files

Binary files contain the raw code and data of the firmware. Since there is no metadata, the application entry point and the base address of the firmware in the file have to be provided using the **`--entry-point`** and **`--binary-fw-base-addr`** arguments, respectively.

SEGGER wrapped / signed firmware files

These files are produced by the FirmwareTool during the firmware wrapping process.

Input file format detection

The FirmwareTool automatically detects the format of the input file, independent of the file extension, except for binary files. In order to process a binary file, the command line option **`--binary-fw-base-addr`** must be specified.

9.2 Firmware preparation tool option reference

`FirmwareTool.exe`/`FirmwareTool` is a command line tool for PC (Windows / Linux).

General command line syntax:

`FirmwareTool` `<operation>` `<options>`...

The supported `<operation>`s are explained in the following sections.

General options	
`--version`	Print the version of the key generation tool and the underlying emCrypt library.
`-q`	Quiet execution. Don't output messages, except errors.
`--in` `<input-file>`	Input file name, see *Supported file formats*.
`--out` `<output-file>`	Output file name, see *Supported file formats*.
`-c` `<config-file>`	Path name of the `BOOT_Conf.h` file used to configure the bootloader
`-I` `<incpath>`	Path to an include directory containing files included by `BOOT_Conf.h`. The file `BOOT_ConfDefaults.h` must be available via this path.
`-D` `<macro>`	Allows the definition of a preprocessor macro used for processing of the configuration file.
`--pass` `<pwd-opt>`	Provide a password for a signature key. See *Providing passwords* for details.

9.2.1 Specifying version numbers

As described in the section *Version numbers*, version numbers are represented by 32 bit unsigned integers in emBoot-Secure. When an update package or signed firmware image is created, the firmware preparation tool needs to know the version number of the application firmware or bootloader it is processing. The version number can either be specified as a command line parameter or be read out from an ELF file.

Specifying the version number as a command line parameter

The version number can be specified as decimal number, hexadecimal number (prefixed with `0x`) or in dot notation with up to four components (major, minor, patch, etc.), like "`1.23.5`". Each component specified in dot notation is mappend to a byte in the 32 bit version number, starting with the highest byte. For example, the version number specification

`-v 3.12.5.127`

is equivalent to specifying:

`-v 0x30C057F`

Reading the version number from an ELF file

If the application firmware or bootloader is read from an ELF file, the firmware preparation tool can be configured to read the version number from a symbol in the ELF file. The configuration flags `SBTL_FW_VERSION_SYMBOL` can be set to the name of a symbol in the application firmware image which holds the version number:

```
#define SBTL_FW_VERSION_SYMBOL    "ApplicationVersion"
```

In this case, the firmware preparation tool will search for a symbol with the name `ApplicationVersion`, which can be defined in the application firmware code like this:

```
const U32 ApplicationVersion = 101;
```


Depending on the linker optimization settings, the symbol may be removed if it is not referenced in the application. If that happens, either the symbol needs to be referenced in application code, or placed in a linker section which is marked for the linker to keep even without being referenced.

The equivalent flag for specifying the name of the symbol holding the bootloader version is `SBTL_BTL_VERSION_SYMBOL`.

9.2.2 Specifying base address and entry point

When processing firmware images in binary format, the firmware base address and the entrypoint have to be specified manually. When reading SREC, HEX or ELF files, the firmware preparation tool determines these values automatically.

The firmware base address specifies the address at which the firmware image starts relative to the flash region reserved for it. The firmware preparation tool verifies that it does not overlap with the header which needs to be in front of the firmware image, and provides for padding in case there is a gap between the header and the start of the firmware image.

`--binary-fw-base-addr` `<addr>`

The firmware preparation tool also has to be informed about the address of the entry point, which is usually the reset handler, as follows:

`--entry-point` `<address>`

9.2.3 Specifying key indices

The bootloader can be provided with multiple public signing keys to verify signatures of update packages, as well as multiple keys which can be used for update package encryption. During the conversion of the keys to C files using the *key generation tool*, they are assigned an 0-based index. The firmware preparation tool needs to record this index in the update package for the bootloader to know which key it has to use for decryption/signature validation.

The index of the signing key is provided using this option:

`--sign-key-index` `<index>`

The index of the encryption key is provided using this option:

`--enc-key-index` `<index>`

9.2.4 Specifying algorithms

The bootloader can be configured to provide update package decryption, decompression and validation using several algorithms by setting the appropriate flags in the file `BOOT_Conf.h`, see section *Configuration switches related to algorithms*. When multiple algorithms for the same operation (encryption, compression or signing) are enabled, an algorithm has to be specified during update package creation.

The signature algorithm can be specified using the following option, with `<algo>` being one of: **`rsa+sha256`**, **`rsa+sha512`**, **`rsa+sha1`**, **`ecdsa+sha256`**, **`ecdsa+sha384`**, **`ecdsa+sha512`** or **`ecdsa+sha1`**.

`--signature-algo` `<algo>`

The compression algorithm can be specified using the following option, with `<algo>` being one of: **`smash2`**, **`smash2T2`** or **`none`**.

`--compression-algo` `<algo>`

The encryption algorithm can be specified using the following option, with `<algo>` being either **`aes-gcm`** or **`none`**.

`--encryption-algo` `<algo>`

9.2.5 Providing a signing method

The firmware preparation tool is designed to support both signing update packages using the Signature Server as well as using private signing keys provided via local files.

To instruct the tool to use a private signing key from a local file, the name of the file can either be provided directly or prefixed with the **`file:`** protocol identifier. The password for the key file can be provided as well, using the `<pwd-opt>` options discussed in section *Providing passwords*:

`--key` `<file:key-file>` **`--pass`** `<pwd-opt>`

`--key` `<key-file>` **`--pass`** `<pwd-opt>`

When using a Signature Server to sign the update package, the protocol and address of the server have to be specified. The format is the same as for the **`--server`** parameter as explained in section *Specifying a Signature Server address*. Additionally, the name of the signing key on the server has to be specified.

Assuming the Signature Server can be found on the local network via its hostname **`SignatureServer-12345678`**, and the key **`SignKey1`** is to be used, the following parameters have to be provided, either directly or prefixed with the **`sigs:`** protocol identifier:

`--key` `<sigs:SignKey1@net:SignatureServer-12345678>` **`--pass`** `<pwd-opt>`

`--key` `<SignKey1@net:SignatureServer-12345678>` **`--pass`** `<pwd-opt>`

The Release Manager's password is provided by the **`--pass`** parameter, using the `<pwd-opt>` options discussed in section *Providing passwords*.

9.3 Creating an update package

An application firmware image or bootloader image is wrapped into an update package using the following command, resulting in an update file in binary format:

```
FirmwareTool wrap --in <input-file> --out <output-file>
    -c <config-file> -I <incpath> [-D <macro>]
    --key|-k <key-spec> --pass <pwd-opt>
    --enc-key <key-file> --enc-key-pass <pwd-opt>
    [-v <version-number>] [--btl]
```

Options	
-v <version-number>	Specifies the firmware version number (32 bit unsigned integer), see *Specifying version numbers*.
--btl	Specifies that a bootloader is being packaged.
--key|-k <key-spec>	Key-spec to use for signing, see *Providing a signing method*.
--pass <pwd-opt>	Provide a password for the signing method, see *Providing passwords* for details.
--enc-key <key-file>	If encryption is to be used: Input file that contains the AES key.
--enc-key-pass <pwd-opt>	Provide a password for an encryption key. See *Providing passwords* for details.

9.4 Creating a signed firmware image

A firmware image can be signed using the following command, resulting in an SREC file which can be flashed into the target system:

```
FirmwareTool sign --in <input-file> --out <output-file>
    -c <config-file> -I <incpath> [-D <macro>]
    --key|-k <key-spec> --pass <pwd-opt>
    -v <version-number>
```

Options	
-v <version-number>	Specifies the firmware version number (32 bit unsigned integer), see *Specifying version numbers*.
--key|-k <key-spec>	Key-spec to use for signing, see *Providing a signing method*
--pass <pwd-opt>	Provide a password for the signing method, see *Providing passwords* for details.

Chapter 10

The Signature Server administration tool

The `SigServerAdmin` tool is used to configure and manage the Signature Server provided for the emBoot-Secure ecosystem.

The following operations can be carried out:

Operation	Explanation
`status`	Checks the status of a Signature Server.
`list`	Lists users and keys.
`set-password`	Creates a password based user or changes the password of an existing user.
`delete`	Deletes users or keys.
`grant-permission`	Grants permission to use keys / administer users.
`revoke-permission`	Revokes permission to use keys / administer users.
`import`	Imports a signing key or creates an Administrator.
`export`	Exports the private part of a signing key.
`set-no-export`	Prevents export of a private signing key.
`verify`	Verifies that a public/private key pair works together.
`gen-ecdsa`	Generates an ECDSA key pair.
`gen-rsa`	Generates an RSA key pair.
`set-hostname`	Sets the server’s hostname.
`set-domainname`	Sets the server’s domain name.
`set-port`	Sets the port number under which the server listens for network connections.
`set-ip`	Configures IP settings.
`upload-update`	Uploads a firmware update package.
`factory-reset`	Resets the Signature Server to factory state.

10.1 Signature Server administration tool invocation

`SigServerAdmin.exe`/`SigServerAdmin` is a command line tool for PC (Windows / Linux).

General command line syntax:

`SigServerAdmin` `<operation>` `<options>`...

The supported `<operation>`s are explained in the following sections.

General options	
`--version`	Print the version of the key generation tool and the underlying emCrypt library.
`-q`	Quiet execution. Don't output messages, except errors.
`--server` `<address>`	Protocol to use for connecting to a Signature Server, see *Specifying a Signature Server address*.
`--auth-pass` `<pwd-opt>`	Password to use for authentication to a Signature Server, see *Providing passwords* for details.

10.2 Specifying a Signature Server address

The protocol and connection details to use when connecting to a Signature Server can be specified to the Signature Server administration tool in two ways:

- Using the **`--server`** `<address>` option.
- Using the environment variable **`SIGSERVER_ADDRESS`**, by setting it to the value which would otherwise be passed to the **`--server`** option.

Specification of a port number is optional. If no port number is specified, the default value of 33030 is used.

To establish a connection to a Signature Server with a known IP address, the address can be specified as follows, with an optional port number:

```
--server net:IP_address[:port]
```

A Signature Server can also be reached via a hostname if the local network supports this. When using DHCP for network configuration, the Signature Server provides a hostname to the DHCP server, which may provide this information to the local DNS server to make the server reachable via the hostname. Depending on the configuration of the local DNS server, a suffix or domain name may be appended to the hostname. In the simplest case, the Signature Server can be addressed by its hostname like this

```
--server net:hostname[:port]
```

The Signature Server also announces its hostname and IP address via mDNS. On modern operating systems, these hostnames can be resolved with the suffix **`.local`**, provided the Signature Server and the host computer are in the same network, or broadcast messages are bridged between the two networks. The default hostname is **`SignatureServer-Serialnumber`**. A Signature Server can be addressed via its hostname using mDNS like this:

```
--server net:hostname.local[:port]
```

Connections to a Signature Server via USB require that the USB interface is enabled by pressing the push button on the Signature Server, see *Ports and push button* on page 64. If more than one SignatureServer is connected to the host computer, the serial number of the target server must be specified. In case of a single server, specification of the serial number is optional.

```
--server usb:[serial_number]
```


10.3 Authentication for commands

The Signature Server provides two authentication methods: key-based authentication (similar to SSH's public key authentication) and password authentication. For administrative commands, key-based authentication is required, while for signing firmware, passwords are used.

For password-based authentication to a server, the **`--auth-pass`** `<pwd-opt>` parameter is used. See *Providing passwords* for details about how passwords can be specified with this parameter.

For key-based authentication, a public-private key pair is required, see *Generating an authentication key pair* for details. A private key should be stored encrypted, requiring a password for decryption during the authentication phase. The path to the private key can be specified in two ways:

- Using the environment variable **`SIGSERVER_AUTH_KEY`**
- Using the **`--auth-key`** `<key-file>` option:

By default, the password for the private authentication key will be read from the console. A different method can be chosen by using the **`--auth-key-pass`** `<pwd-opt>` option, following the conventions defined under *Providing passwords*.

`--auth-key` `<key-file>` **`--auth-key-pass`** `<pwd-opt>`

10.4 Checking the status of a Signature Server

The following command reports the status (see *Signature Server global states*), serial number, software version, bootloader version and the network configuration of the Signature Server. No authentication is required.

`SigServerAdmin status --server` `<address>`

Options	
`--server` `<address>`	Server address, see *Specifying a Signature Server address*.

Sample output for an operational server using DHCP:

```
Server serial number: 12345678
Server bootloader version: 1.0.1
Server software version: 1.0.0
Server state: Operational
Server IP configuration: DHCP
Server port number: 33030 (default)
Server hostname: 'SignatureServer-12345678'
Server domain name: not set
```


10.5 Listing users and keys

The following command lists the private keys and configured users (Device Owner, Administrators and Release Managers). No authentication is required.

`SigServerAdmin list --server` `<address>`

Options	
`--server` `<address>`	Server address, see *Specifying a Signature Server address*.

The SigServerAdmin tool prints a list of the following form. The meaning of the column *Accessible to* depends on the context:

- For keys, it lists which Administrators can manage each key and which Release Managers may use it for signing.
- For users, it lists which Administrators can manage the user.

```
Server contains 5 items:
  Name              Usage  Typ Att   Accessible to
  ------------------------------------------------
  Owner             RSA    Own       Owner
  Admin1            RSA    Adm       Admin1
  RelMan1           PWD    Usr Rem   Admin1, RelMan1
  key1              RSA    Sig       Admin1, RelMan1
  key2              ECDSA  Sig       Admin1, RelMan1
```

The meaning of the abbreviations used in the `Usage`, `Typ` and `Att` columns is as follows:

Usage	
`RSA`	RSA key
`ECDSA`	ECDSA key
`PWD`	Password based authentication
Typ	
`Own`	Device Owner
`Adm`	Administrator
`Usr`	Release Manager
Att	
`Rem`	Account can be used via network
`noEx`	Key can’t be exported

10.6 Managing Release Manager passwords

The **set-password** command can be used to create a Release Manager account with a password, or to change the password of an existing account.

Release Manager accounts can only be created by Device Owners and Administrators, who have to use key-based authentication. Permission to administer the Release Manager user is granted only to the Administrator who created the user. To grant permission to other Administrators, see *Granting permission*. The following command creates a new user with a password, which can be used for signing.

```
SigServerAdmin set-password -n <username> --pass <pwd-opt>
    --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

A Release Manager can change their own password using the following variant of the previous command. Note that the current password for the user needs to be specified via **--auth-pass**, while the new password is specified via **--pass**:

```
SigServerAdmin set-password -n <username> --pass <pwd-opt>
    --server <address>
    --auth-pass <pwd-opt>
```

Options	
-n <username>	Name of the new user, up to 16 characters long.
--pass <pwd-opt>	Password for the new user, see *Providing passwords*. Minimum password length is 4 characters.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.
--auth-pass <pwd-opt>	Current user password to use for authentication to a Signature Server, see *Providing passwords* for details.

10.7 Deleting users or keys

The following command deletes an entity, such as a user, a private signing key or an authentication key, from the Signature Server. Key-based authentication is required.

```
SigServerAdmin delete -n <entity-name> --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
-n <entity-name>	Name of the key or user to delete.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.8 Granting permission

The Signature Server manages permissions of both Administrators and Release Managers. When keys are generated or imported, they can only be administered by the Administrator who created them. Release Managers can also only be administered by the Administrator who created them.

- **Release Managers** can be granted permission to use signing keys to sign firmware.
- **Administrators** can be granted permission to administer Release Managers and keys by either the Device Owner or other Administrators.

Key-based authentication is required.

Given a key or Release Manager, referred to by `<entity-name>`, the following command grants permission to one or more user accounts to use or administer that entity:

```
SigServerAdmin grant-permission -n <entity-name>
    -u <username[,username2]>
    --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
-n `<entity-name>`	Name of the key or Release Manager to administer.
-u `<username[,username2]>`	One or more user names to grant permissions to use/ administer the entity.
--server `<address>`	Server address, see *Specifying a Signature Server address*.
--auth-key `<key-filename>`	Name of file containing the private authentication key.
--auth-key-pass `<pwd-opt>`	Password for the private key, see *Providing passwords*.

10.9 Revoking permission

Permissions to administer keys and Release Managers, or to use keys for signing, can be revoked. Administrators can revoke permissions for entities for which they have permissions, while the Device Owner can revoke permissions for any entity.

Key-based authentication is required.

Given a key or Release Manager, referred to by <entity-name>, the following command revokes permission to use or administer that entity from one or more user accounts:

```
SigServerAdmin revoke-permission -n <entity-name>
    -u <username[,username2]>
    --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
-n <entity-name>	Name of the key or Release Manager to administer.
-u <username[,username2]>	One or more user names to revoke permissions regarding the entity.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.10 Importing a key

The key import command has two purposes:

Creating Device Owner and Administrator accounts

Device Owner and Administrator accounts are created by importing the public part of their authentication key pair into the Signature Server. The first public authentication key which is imported defines the Device Owner.

Importing signing keys

A signing key is created by importing the private part of the signing key pair. Permission to administer the imported key is granted only to the Administrator who imported the key. To grant permission to other Administrators, see *Granting permission*. Release Managers also need to be granted permission to use the key for signing in the same way.

Keys can only be imported by the Device Owner and Administrators.

```
SigServerAdmin import -n <name> --in <path>
    --key-pass <pwd-opt>
    --server <address> --auth-key <key-filename>
    [--auth-key-pass <pwd-opt>]
```

Options	
-n <name>	Name to assign to the imported key or newly created Device Owner / Administrator. Can be up to 16 characters long.
--in <path>	Pathname of the file containing the key.
--key-pass <pwd-opt>	Password for the private signature key, see *Providing passwords*.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.11 Exporting a signing key

The private part of a signing key pair can be exported from a Signature Server, unless the key is marked as no-export. It is recommended to specify a password for encryption of the exported private key. Key-based authentication is required.

```
SigServerAdmin export -n <keyname> --out <key-file>
    --key-pass <pwd-opt> --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
-n <keyname>	Name of the key to export.
--out <key-file>	Pathname of the key file to write the private key into.
--key-pass <pwd-opt>	Password for the private signature key, see *Providing passwords*.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.12 Preventing export of a key

Export of a key from the Signature Server can be prohibited by flagging a key as "no-export". This can be achieved with the following command, which requires key-based authentication.

```
SigServerAdmin set-no-export -n <key-name> --server <address>
    --auth-key <path> [--auth-key-pass <pwd-opt>]
```

Options	
-n <key-name>	Name of the key to flag as "no-export".
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <path>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.13 Verifying a key pair

The following command verifies that a signature made with a private signing key stored in the Signature Server with name <key-name> can be verified using the public key stored in <key-file>. It can be used by Release Managers which have been granted permission to use key <key-name> using their password.

```
SigServerAdmin verify -n <key-name>
    --in <key-file> --auth-pass <pwd-opt>
    --server <address>
```

Options	
-n <key-name>	Name of the private signing key to be used for verification.
--in <key-file>	Pathname of the public key file to be used for verification.
--auth-pass <pwd-opt>	Password for authentication, see *Providing passwords*.
--server <address>	Server address, see *Specifying a Signature Server address*.

10.14 Generating an ECDSA key pair (Signature Server)

The following command generates an ECDSA key pair. The private key is stored in the Signature Server under the specified name. The "no-export" flag is not automatically set on this new key. The public key is written into a file. Key-based authentication is required.

Permission to administer the generated key is granted only to the Administrator who generated the key. To grant permission to other Administrators, see *Granting permission*. Release Managers also need to be granted permission to use the key for signing.

Either the name of the elliptic curve to be used can be specified or the key size in bits. If the key size is specified, Brainpool curves are used by default. To use curves provided by NIST, the **--NIST** parameter has to be specified.

```
SigServerAdmin gen-ecdsa -n <key-name>
    [-s <keysize> [--nist] | --curve <curve-name>] --out <key-file>
    --server <address> --auth-key <key-filename>
    [--auth-key-pass <pwd-opt>]
```

Options	
-n <keyname>	Name to assign to the newly generated key, up to 16 characters long.
-s <keysize>	Bit size of the ECDSA key to be generated. This option determines the elliptic curve which is used. Possible values are 256, 320, 384, 512 or 521. Default is 256.
--nist	Use NIST curves instead of Brainpool curves.
--curve <name>	Specify the elliptic curve to be used by name.
--out <path>	Pathname of the public key file to be created.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

Supported elliptic curves are:

- brainpoolP256r1
- brainpoolP320r1
- brainpoolP384r1
- brainpoolP512r1
- secp256r1
- secp384r1
- secp521r1

> **Note**
>
> Alternatively, ECDSA key pairs can be generated using the key generation tool, see *Generating an ECDSA key pair (KeyTool)*.

10.15 Generating an RSA key pair (Signature Server)

The following command generates an RSA key pair. The private key is stored in the Signature Server under the specified name. The "no-export" flag is not automatically set on this new key. The public key is written into a file. Key-based authentication is required.

Permission to administer the generated key is granted only to the Administrator who generated the key. To grant permission to other Administrators, see *Granting permission*. Release Managers also need to be granted permission to use the key for signing.

```
SigServerAdmin gen-rsa -n <key-name>
    [-s <keysize>] --out <key-file> --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
-n <keyname>	Name to assign to the newly generated key, up to 16 characters long.
-s <keysize>	Bit size of the RSA key to be generated. Supported values are 2048, 3072, 4096, 6144 or 8192. Default is 3072.
--out <path>	Pathname of the public key file to be created.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

> **Note**
>
> Alternatively, RSA key pairs can be generated using the key generation tool, see *Generating an RSA key pair (KeyTool)*.

10.16 Setting the hostname

The Signature Server's hostname is used when it requests an IP address from a DHCP server. It also advertises itself using mDNS using the hostname. See *Specifying a Signature Server address* on page 140 for details about how the hostname is used. Changes are applied after a restart of the Signature Server. Key-based authentication is required for setting it with the following command:

SigServerAdmin set-hostname --hostname <hostname> **--server** <address> **--auth-key** <key-filename> [**--auth-key-pass** <pwd-opt>]

Options	
--hostname <hostname>	Hostname to assign to server, up to 31 characters long.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.17 Setting the domain name

The Signature Server's domain name is used when it requests an IP address from a DHCP server. Changes are applied after a restart of the Signature Server. Key-based authentication is required for setting it with the following command:

SigServerAdmin set-domainname --domainname <domainname> **--server** <address> **--auth-key** <key-filename> [**--auth-key-pass** <pwd-opt>]

Options	
--domainname <domainname>	Domain name to assign to server, up to 31 characters long.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.18 Setting the port number

The port number on which the Signature Server listens for incoming network connections normally does not need manual changes. Changes are applied after a restart of the Signature Server. Key-based authentication is required for setting it with the following command:

```
SigServerAdmin set-port --port <port-number> --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
--port <port-number>	Port number to listen on. Default is 33030, valid range is 1 to 65535.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.19 Configuring IP settings

By default, the Signature Server is configured to use DHCP for network configuration. It can also be assigned a static IP address, along with network mask and gateway. To enable DHCP, the IP address has to be set to **0.0.0.0**. Changes are applied after a restart of the Signature Server.

When configuring the Signature Server to use DHCP, the parameters **--mask** and **--gateway** must not be specified.

Key-based authentication is required to change these settings.

```
SigServerAdmin set-ip --ip <IP-address> [--mask <mask>]
    [--gateway <GW-address>] --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
--ip <IP-address>	IP address to assign to the Signature Server. **0.0.0.0** enables DHCP instead.
--mask <mask>	Subnet mask in dot-notation (e.g. **255.255.0.0**).
--gateway <GW-address>	IP address of the gateway to use.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

10.20 Uploading firmware updates

Firmware updates for the Signature Server can be uploaded using the following command, which requires key-based authentication:

```
SigServerAdmin upload-update --in <path> --server <address>
    --auth-key <key-filename> [--auth-key-pass <pwd-opt>]
```

Options	
--in <path>	Pathname to the file containing the update package.
--server <address>	Server address, see *Specifying a Signature Server address*.
--auth-key <key-filename>	Name of file containing the private authentication key.
--auth-key-pass <pwd-opt>	Password for the private key, see *Providing passwords*.

The update is installed during the next reboot. The update installation process must not be aborted.

10.21 Resetting to factory state

The Signature Server can be reset to factory state. This function can be used to securely erase all keys from a Signature Server. Keys are securely deleted by erasing the internal flash memory area which stores them, leaving no option of key recovery. It can also be used to regain access to a Signature Server in case administrative credentials were lost, at the cost of erasure of all keys in the server. A factory reset must be initiated via a USB connection.

To perform a factory reset, the **SigServerAdmin** tool has to be invoked as follows:

```
SigServerAdmin factory-reset --server <address>
```

The activity LED on the Signature Server will start flashing red, and the factory reset has to be confirmed by pressing the push button on the Signature Server for at least three seconds within 20 seconds after the command has been initiated. Once the factory reset has been performed, the Ready-LED will turn off, indicating factory delivery state, see *Status LEDs*.

Options	
--server <address>	Server address, see *Specifying a Signature Server address*. Note that the USB protocol has to be used in this case.

Chapter 11

Resource use and performance

This chapter provides estimates on the resource requirements (in terms of RAM and ROM) and the performance of the emBoot-Secure bootloader on a target system.

The most critical performance factor is the signature verification, which takes place at every boot and therefore impacts the boot time of the device. With respect to RAM usage, the peak is reached during a firmware update and depends on the parameters that emBoot-Secure is configured with. The ROM usage depends on the selected features, algorithms, and their optimization goals.

The following table shows the hardware and the toolchain details of the real-world target system used for benchmarking.

Detail	Description
CPU	STM32H7, Cortex-M7, 400 MHz
Toolchain	Embedded Studio Release 8.26, segger-cc Version 20.1.1, ARM Thumb-2 instructions
Compiler Optimization Level	Level 2 balanced
Crypto Library	emCrypt Version 2.50.0

11.1 Runtime performance (signature verification)

The most critical operation with respect to the runtime performance of the emBoot-Secure bootloader is the verification of the signature of an application firmware image. This operation happens on every boot and whenever an update is installed. The verification is performed in two steps: First, a hash of the (decompressed, decrypted) firmware image is computed. Second, the actual verification is computed.

11.1.1 Signature verification, step 1: Hash computation

A significant fraction of the signature verification depends on the selected hash function. The following measurements describe the average performance of the supported hash functions, based on software implementations of the respective hash algorithms, with their optimization definitions set to `SEGGER_OPT_GOAL_SPEED` (see *Configuring the CRYPTO library*). The measurements were collected on the aforementioned Cortex-M7 MCU.

Hash function	Throughput
SHA-1	12.48 MB/s
SHA-256	8.92 MB/s
SHA-384	4.56 MB/s
SHA-512	4.56 MB/s

On this particular MCU, it would take around 0.22 seconds to hash a 2MB (uncompressed) firmware image with the SHA-256 hash function. The same image could be hashed with SHA-512 or SHA-384 in 0.44 seconds or with SHA-1 in 0.16 seconds.

11.1.2 Signature verification, step 2: Verification computation

After the hash of the firmware image has been computed, the signature verification has to be computed. This computation depends on the selected signature scheme.

11.1.2.1 RSA

The runtime of an RSA verification depends on the selected key length. On the aforementioned Cortex-M7 MCU, the following runtime can be expected on average.

RSA key length	Verification time
3072 bit	4.82 ms
4096 bit	7.76 ms
8192 bit	30.20 ms

11.1.2.2 ECDSA

The runtime of an ECDSA verification depends on the selected curve. On the aforementioned Cortex-M7 MCU, the following runtime can be expected on average.

ECDSA curve	Verification time
`brainpoolP256r1`	21.93 ms
`brainpoolP320r1`	32.62 ms
`brainpoolP384r1`	49.88 ms
`brainpoolP512r1`	94.20 ms
NIST P-256 (`secp256r1`)	19.75 ms
NIST P-384 (`secp384r1`)	31.68 ms
NIST P-521 (`secp521r1`)	61.49 ms

11.2 RAM usage

The RAM memory consumption of the bootloader is dominated by its internal data structure, `SBTL_WORK_MEMORY`, which is used to perform the necessary operations (signature verification, decompression, and decryption). The size of the data structure depends on the selected algorithms and their parameters. The data structure is defined as a static variable. This allows the amount of consumed memory to be determined at compilation time.

11.2.1 Signature verification

The memory footprint of the signature verification depends on the signature scheme, the selected key size, and the selected hash function. Generally, ECDSA is more RAM-efficient than RSA, and smaller key sizes require less memory. The difference between SHA-1 (smallest hash function) and SHA-512 (largest hash function) is about 200 bytes in RAM. The following table shows the approximate RAM usage of various configurations.

Signature scheme	Key size	Hash function	RAM	Comment
ECDSA	256	SHA-1	2.3 kB	Minimal configuration
ECDSA	256	SHA-256	2.3 kB	Recommended configuration
ECDSA	256	SHA-512	2.5 kB	
ECDSA	521	SHA-256	4.3 kB	
ECDSA	521	SHA-512	4.5 kB	Maximal ECDSA configuration
RSA	3072	SHA-1	3.7 kB	Minimal RSA configuration
RSA	3072	SHA-256	3.7 kB	Recommended RSA configuration
RSA	3072	SHA-512	3.8 kB	
RSA	4096	SHA-256	4.9 kB	
RSA	4096	SHA-512	5.0 kB	
RSA	8192	SHA-512	9.6 kB	Maximal configuration

11.2.2 Decompression (optional)

If the update package is compressed, it has to be decompressed during an update. The approximate RAM resource usage of the decompression step depends on the selected algorithm and its window size, as shown in the table below.

Compression algorithm	Window size	RAM
None	—	0.0 kB
`SMASH2`	1 kB	1.2 kB
`SMASH2`	4 kB	4.3 kB
`SMASH2_T2`	1 kB	1.4 kB
`SMASH2_T2`	4 kB	4.5 kB

11.2.3 Decryption (optional)

If the update package is encrypted, it has to be decrypted during an update. This step requires additional resources in RAM, which are approximated in the following table.

Encryption algorithm	RAM
None	0.0 kB
AES-GCM	0.5 kB

11.3 ROM usage

The overall ROM usage of the bootloader is composed of basic bootloader functions, MCU-specific functions, and the code and data required to run the selected algorithms for signature verification, decompression, and decryption. The following section provides an overview of the usual ROM usage of a bootloader. The sections afterwards provided more detailed information about how specialized optimizations on the cryptographic algorithms increase ROM size requirements.

11.3.1 ROM usage overview

The following table provides estimates for each individual component of the bootloader, which were collected on a Cortex-M7 MCU. The core bootloader logic encompasses the emBoot-Secure library, the board specific flash memory driver and the bootloader's `main()` function. The size of this component changes based on the additional features which are enabled. Signature verification code ranges between 5 kB for RSA and 9 kB for ECDSA. The total bootloader size ranges between 9 and 40 kB. Additional components, like the libC provided by the compiler vendor and startup code provided by the microcontroller's manufacturer, are not included in this estimate, since they are very project-specific. When compiling the bootloader using SEGGER's Embedded Studio for a typical Cortex-M7 MCU, another 3 kB of ROM size are added.

Component	ROM size range
Core bootloader logic	3 - 5 kB
RSA/ECDSA signature verification	5 - 9 kB
Hash function	1 - 8 kB
Decryption	0 - 17 kB
Decompression	0 - 1 kB
Total bootloader size	**9 - 40 kB**

In the code sample considered, the bootloader obtains the update package from a section of flash memory. If the bootloader needs to access external flash memory, optionally with a filesystem, the size will increase accordingly.

11.3.2 Hash computation optimization

The additional ROM usage due to the hash function depends on the selected algorithm and the configured optimization level, which is set at compile time by the `CRYPTO_OPT_GOAL_SHA*` macros in the `CRYPTO_Conf.h` file (see *Configuring the CRYPTO library*). The following table shows estimates for various hash functions and optimization levels.

Hash function	CRYPTO_OPT_GOAL_SHA*	ROM
size-optimized		
SHA1	SEGGER_OPT_GOAL_BALANCED (default)	0.8 kB
SHA256	SEGGER_OPT_GOAL_BALANCED (default)	1.0 kB
SHA384	SEGGER_OPT_GOAL_BALANCED (default)	2.1 kB
SHA512	SEGGER_OPT_GOAL_BALANCED (default)	2.0 kB
speed-optimized		
SHA1	SEGGER_OPT_GOAL_SPEED	4.1 kB
SHA256	SEGGER_OPT_GOAL_SPEED	8.0 kB
SHA384	SEGGER_OPT_GOAL_SPEED	7.4 kB
SHA512	SEGGER_OPT_GOAL_SPEED	7.3 kB

(Note: The SHA384 hash function is also configured through the CRYPTO_OPT_GOAL_SHA512 macro.)

11.3.3 Optimization of decryption

If the update package is encrypted, it has to be decrypted during an update. Decryption is based on the AES algorithm in GCM mode. For both components, AES and GCM, optimization can be configured independently, which influences their respective ROM footprint. The configuration is made through definitions in the file CRYPTO_Conf.h (see *Configuring the CRYPTO library*).

The following table shows the approximate additional ROM usage of the decryption function depending on the value of CRYPTO_OPT_GOAL_AES.

Encryption algorithm	CRYPTO_OPT_GOAL_AES	ROM
None	—	0.0 kB
AES-GCM	SEGGER_OPT_GOAL_SIZE	4.4 kB
AES-GCM	SEGGER_OPT_GOAL_BALANCED (default)	12.3 kB
AES-GCM	SEGGER_OPT_GOAL_SPEED	16.4 kB

Chapter 12

Support

12.1 Contacting support

Before contacting support please make sure that you are using the latest version of the emBoot-Secure package. Also please check the chapter *Configuring debugging output* on page 29 and run your application with enabled debug support.

If you are a registered emBoot-Secure user there are different ways to contact the emBoot-Secure support:

1. You can create a support ticket via email to ticket_emboot-secure@segger.com*
2. You can create a support ticket at segger.com/ticket.

Please include the following information in the email or ticket:

- The emBoot-Secure version.
- Your emBoot-Secure license number.
- If you are unsure about the above information you can also use the name of the emBoot-Secure zip file (which contains the above information).
- A detailed description of the problem
- The configuration files `BOOT_Conf*.*` and `CRYPTO_Conf.h`.
- Any error messages.

Please also take a few moments to help us improve our services by providing a short feedback once your support case has been solved.

12.1.1 Where can I find the license number?

The license number is part of the shipped zip file name.
For example `emBoot_BASE_V2.0.0_BOOT-01234_DD92169C_260224.zip` where BOOT-01234 is the license number. The license number is also part of every *.c- and *.h-file header. For example, if you open BOOT.h you should find the license number as with the example below:

```
*********************************************************************
*                                                                   *
*       emBoot-Secure version: V2.10.0                              *
*                                                                   *
```

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.


```
*********************************************************************
---------------------------------------------------------------------
Licensing information
Licensor:                 SEGGER Microcontroller GmbH
Licensed to:              Customer name
Licensed SEGGER software: emBoot-Secure
License number:           BOOT-01234
License model:            SSL
Licensed product:         -
Licensed platform:        Cortex-M, GCC
Licensed number of seats: 1
---------------------------------------------------------------------
Support and Update Agreement (SUA)
SUA period:               2023-05-30 - 2023-11-30
Contact to extend SUA:    sales@segger.com
---------------------------------------------------------------------
Purpose : emBoot-Secure library
```


Chapter 13

Glossary

AES
Advanced Encryption Standard, a symmetric encryption algorithm.

Administrator
A user role in the Signature Server's permission system. Administrators manage signing keys and Release Manager accounts.

Application, application firmware
Software which runs on the target device and provides its core functionality.

Authentication key pair
Key pair used for by an Administrator to authenticate with the SignatureServer. The public key is stored in the Signature Server and the private key is used to prove the Administrator's identity (similar to SSH).

Bootloader
Software which is started by the microcontroller during startup. It verifies the installed application firmware and installs updates.

Device Owner
The user role in the Signature Server's permission system with highest privileges. A Device Owner initializes, configures, and administers the Signature Server and creates Administrator accounts.

ECDSA
Elliptic Curve Digital Signature Algorithm. A digital signature algorithm based on (asymmetric) elliptic curve cryptography.

Firmware encryption key
Key used for symmetric encryption of the firmware image during update package creation, and later decryption during update installation.

Firmware image
A "raw" firmware image produced by the build system, which is neither signed, nor compressed, nor encrypted. It can further be distinguished between an *application firmware image*, which contains the software that delivers the target's core functionality, and a *bootloader image*, which contains the bootloader code.

Firmware preparation tool
Creates a secure update package from a firmware image. This tool is included in the shipping package as `FirmwareTool(.exe)`.

GCM
Galois/Counter Mode. A standardized way to use the AES encryption algorithm to encrypt and authenticate larger amounts of data.

Key generation tool
Software tool which generates and converts public-private key pairs for usage within the emBoot-Secure ecosystem. This tool is included in the shipping package as `KeyTool(.exe)`.

Manufacturer
Manufacturer of the target device, i.e., the company that uses emBoot-Secure in their product(s).

Release Manager
Person who builds the firmware and is responsible for signing it. Also a user role in the Signature Server's permission system.

RSA
An asymmetric cryptosystem, named after its inventors (Rivest, Shamir, Adleman). In emBoot-Secure, RSA is used as one possible implementation of digital signatures.

Signing password
Password used to authenticate the process of signing a firmware image with a private key.

Signature Server
Secure hardware device which can optionally be used for storage of private signing keys. Can generate signatures during update package creation without exposing private keys.

Signature Server administration tool
Software tool which allows to configure the Signature Server and manage its internal key store. This tool is included in the shipping package as `SigServerAdmin(.exe)`.

Signing key pair
Pair of private and public key used to sign a firmware image and to verify an update package.

Shipping package
The software package that is provided by SEGGER and comprises all necessary files to create products using the emBoot-Secure bootloader, along with examples and software utilities.

SMASH
Small Microcontroller Advanced Super-High format. SEGGER's proprietary format for compressing data.

Target device
Device on which the secure bootloader runs and starts the application software.

Update package
Signed and optionally compressed and encrypted firmware image, which can be distributed, transferred to target devices, and installed.

