Gecko Bootloader

- About the Gecko Bootloader
- Getting Started
- Bootloader Fundamentals
  - Introduction
  - Overview
  - About the Gecko Bootloader
  - Memory Space for Bootloading
  - Design Decisions
- Developer's Guide
  - Overview
  - Developing and Debugging
    - Overview
    - Gecko Bootloader User's Guide for Series 3 and Higher
      - Introduction
      - Gecko Bootloader File Format v4
      - Gecko Bootloader Operation Application Upgrade
      - Gecko Bootloader Operation Bootloader Upgrade
      - Gecko Bootloader Operation Secure Engine Upgrade
      - Getting Started with the Gecko Bootloader
      - Configuring the Gecko Bootloader
      - Simplicity Commander and the Gecko Bootloader
      - Gecko Bootloader Security Features
      - Application Interface
    - Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher
      - Introduction
      - Overview
      - Gecko Bootloader File Format
      - Gecko Bootloader Operation Application Upgrade
      - Gecko Bootloader Operation Bootloader Upgrade
      - Gecko Bootloader Operation Secure Engine Upgrade
      - Getting Started with the Gecko Bootloader
      - Configuring the Gecko Bootloader
      - Simplicity Commander and the Gecko Bootloader
      - Gecko Bootloader Security Features
      - Gecko Bootloader and Trustzone
      - Gecko Bootloader and Delta DFU Upgrades
      - Application Interface
    - Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher
      - Overview

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Introduction
- About Projects
- Components
- Pin Tool
- Linker File
- Additional Macros
- Postbuild Steps
- Main Bootloader in Main Flash
- Callbacks
- Storage Slots
- App Properties

Simplicity Commander Reference Guide (link)

Security

Overview

Series 2 and Series 3 Secure Boot with RTSL

- Introduction
- Series 2 and Series 3 Device Security Features
- Secure Boot Process
- Examples
- Debugging on Secure Boot Enabled Device
- Failure Analysis
- Secure Boot Status Codes

Series 2 TrustZone

- Introduction
- Series 2 Device Security Features
- TrustZone Basics
- Bus Level Security (BLS)
- Secure And Privileged Programming Model
- TrustZone Implementation
- Upgrade Existing Application To TrustZone
- TrustZone Platform Examples

Protocol-Specific Information

Overview

Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

- Overview
- Introduction
- BGAPI UART Device Firmware Upgrade
- Bluetooth OTA Upgrade
- Working with Apploader and Secure Boot
- OTA Delta DFU
- Post Build Editor

Using the Gecko Bootloader with Silicon Labs Connect

- Introduction
- Overview
- Using the Gecko Standalone Bootloaders
- Using the Gecko Application Bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Using The Gecko Bootloader With Emberznet
  - Introduction
  - Overview
  - Using the Gecko Standalone Bootloaders
  - Using the Gecko Application Bootloader
- Gecko Bootloader API Reference.
  - Application Interface
    - Application Parser Interface
      - BootloaderParserCallbacks_t
    - Application Properties
      - ApplicationData_t
      - ApplicationCertificate_t
      - ApplicationProperties_t
    - Application Storage Interface
      - BootloaderStorageSlot_t
      - BootloaderStorageImplementationInformation_t
      - BootloaderStorageInformation_t
      - BootloaderEraseStatus_t
      - BootloaderStorageFunctions_t
    - Common Application Interface
      - BareBootTable_t
      - BootloaderInformation_t
      - BootloaderHeader_t
      - FirstBootloaderTable_t
      - MainBootloaderTable_t
      - Reset Information
        - BootloaderResetCause_t
  - Bootloader Core
    - Bootload
    - Flash
    - Reset
    - TrustZone
    - Upgrade
  - Components
    - Communication
      - EZSP-SPI
      - Bluetooth Apploader OTA DFU
      - UART XMODEM
      - BGAPI UART DFU
      - Utils
        - XMODEM Parser
          - XmodemPacket_t
          - Commands
    - Debug
    - Decompressor
      - LZ4 Decompressor

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Lz4Context_t
- LZMA Decompressor
- GPIO Activation
  - Button GPIO
  - EZSP GPIO
- Image Parser
  - GBL Parser
    - GblTagParsingInfo_t
    - ImageProperties_t
    - ParserContext_t
    - GblInputBuffer_t
    - Custom GBL Tags
      - GblCustomTag_t
      - LZ4 Programming Tag
        - Lz4ParserContext_t
      - LZMA Programming Tag
    - GBL Format
      - GblTagHeader_t
      - GblHeader_t
      - VersionDependency_t
      - GblApplication_t
      - GblBootloader_t
      - GblTagDelta_t
      - GblSeUpgrade_t
      - GblMetadata_t
      - GblProg_t
      - GblEnd_t
      - GblEncryptionInitAesCcm_t
      - GblEncryptionData_t
      - GblCertificateEcdsaP256_t
      - GblSignatureEcdsaP256_t
- Security
  - AES
    - AesContext_t
    - AesCtrContext_t
  - CRC16
  - CRC32
  - Decryption
    - DecryptContext_t
    - AuthContext_t
  - ECDSA
    - ECC Library
      - ECC_Point_t
      - ECC_EcdsaSignature_t
  - SHA256
    - btl_sha256_context

Copyright © 2025 Silicon Laboratories. All rights reserved.

- SHA_256
  - Sha256Context_t
- Tokens
- Storage
- BootloaderStorageLayout_t
- Flash Using JEDEC SFDP Standard
- SPI Flash
- Internal Flash
- Bootload Info
  - BootloadInfo_t
- SPI Flash Configurations
- SPI Flash Configurations using SFDP

Driver
- Delay
- SPI
- SPI Peripheral
- UART

Error Codes
- Bootloading Error Codes
- Communication Component Error Codes
- Compression Error Codes
- Error Code Base Values
- Image Parser Error Codes
- Initialization Error Codes
- Parse Error Codes
- RTOS Error Codes
- SPI Peripheral Driver Error Codes
- Security Error Codes
- Storage Driver Error Codes
- UART Driver Error Codes
- XMODEM Error Codes

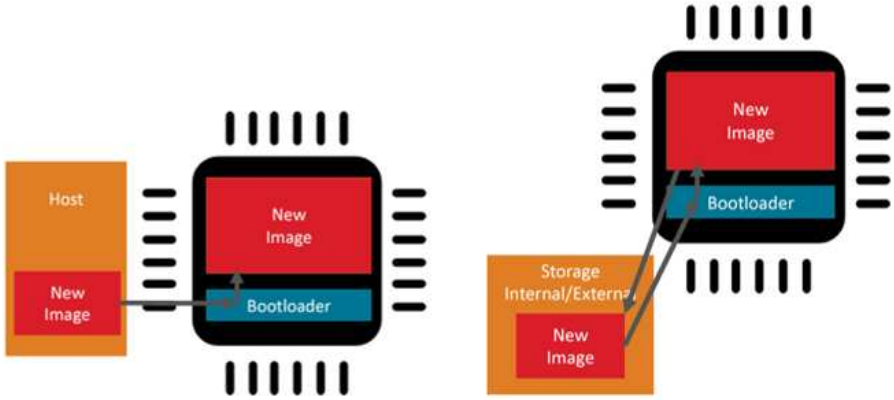
Silicon Labs Gecko Bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

About the Gecko Bootloader

Gecko Bootloader

The Gecko Bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and may also perform integrity checks. Devices are typically programmed with firmware during the product manufacturing process. A bootloader allows for reprogramming devices after production is complete, to add features or fix bugs.



The content on these pages is intended for those who want to experiment with or are already developing a bootloader for use in your devices.

For details about this release: Gecko Bootloader release notes may be found in the combined Gecko Platform release notes, available on the silabs.com Gecko SDK page.

For information about Silicon Labs' hardware or software products: See the developers pages on silabs.com.

For background about bootloading: Bootloader Fundamentals is a good place to start.

To get started working with Gecko Bootloader example applications in Simplicity Studio: See the Getting Started page.

If you are already in development with Gecko Bootloader: See the Developer's Guide for details or go directly to the API Reference.

Bootloaders are usually developed along with the firmware image development. For information about bootloading with individual protocols, see the Protocol-Specific Information pages. An extensive body of content is available for each protocol and can be accessed through the docs.silabs.com homepage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

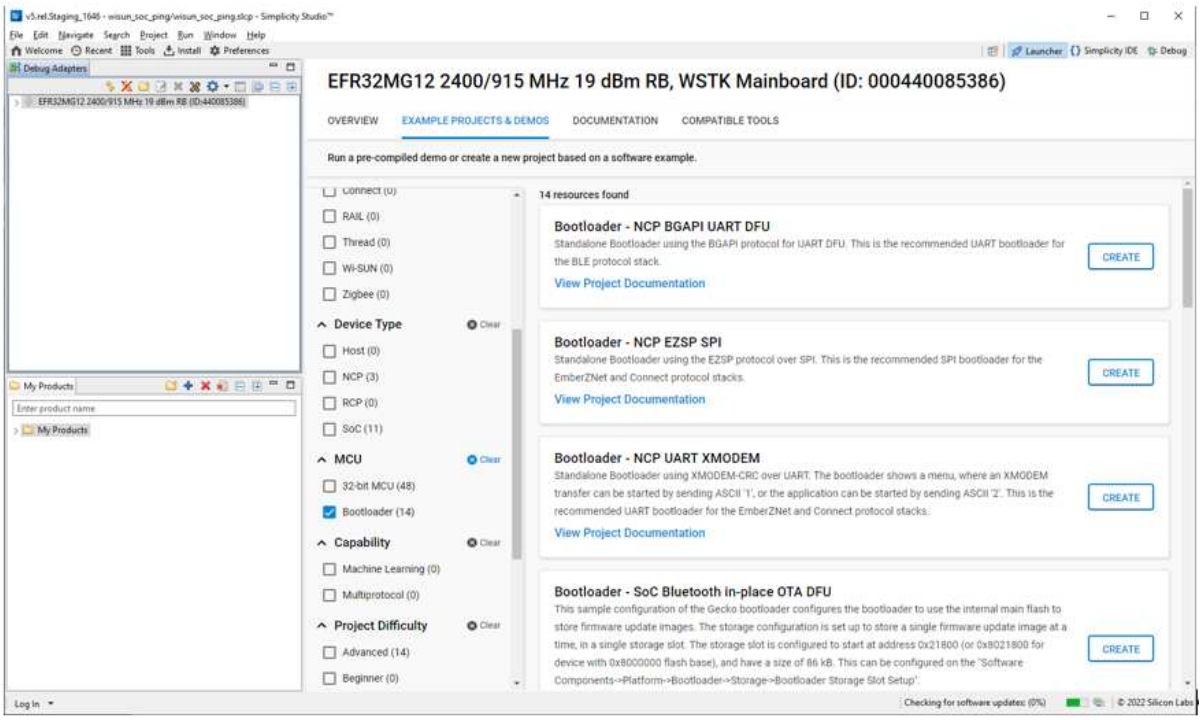
Getting Started

Getting Started with the Gecko Bootloader

The recommended environment for working with the Gecko Bootloader along with your protocol of choice is Simplicity Studio. For information about downloading Simplicity Studio and the Gecko SDK Suite (GSDK), see the Simplicity Studio v5 User's Guide. The application can be downloaded from the Simplicity Studio webpage.

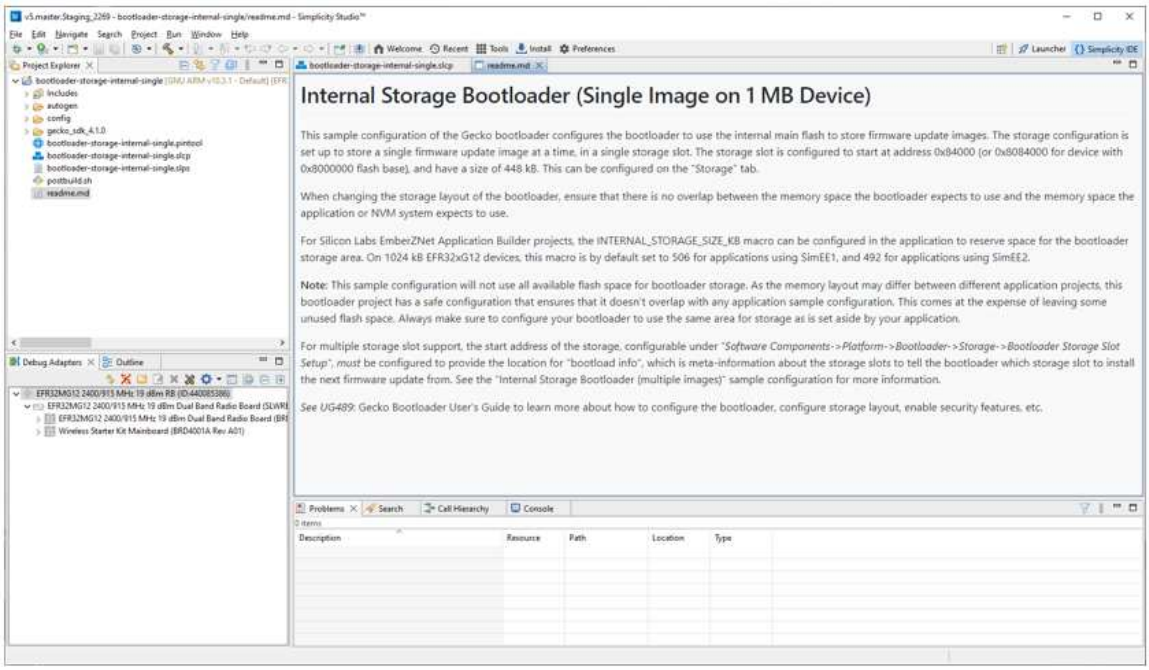
Once you have installed and familiarized yourself with Simplicity Studio and the GSDK, you can get started working with bootloader applications.

1. Create a project based on the Gecko Bootloader example of your choice.

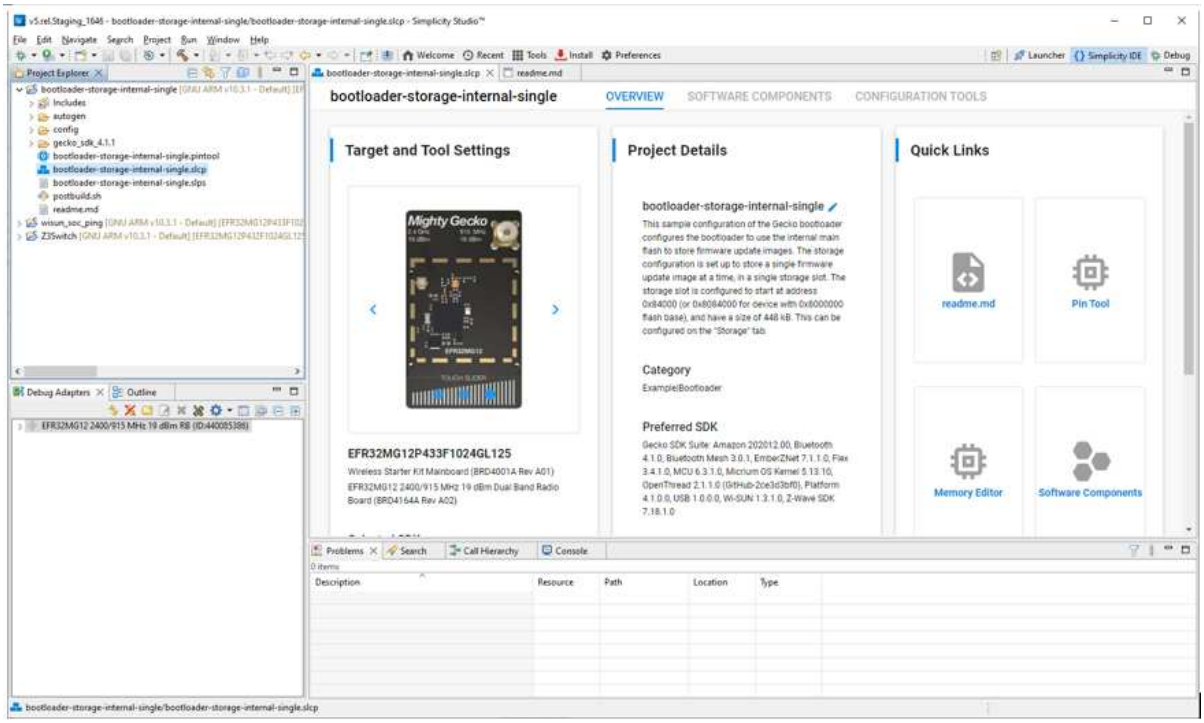


The project opens with a tab describing the example.

Copyright © 2025 Silicon Laboratories. All rights reserved.

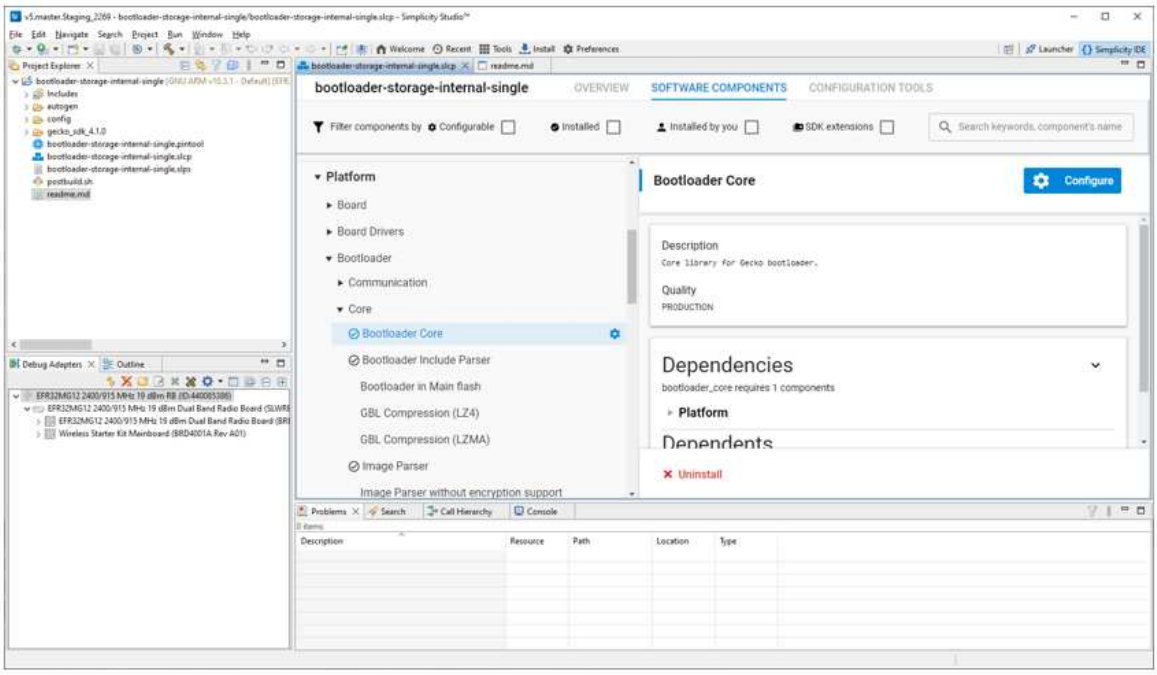


2. Click the project (*.slcp) tab to move to the Project Configurator interface.

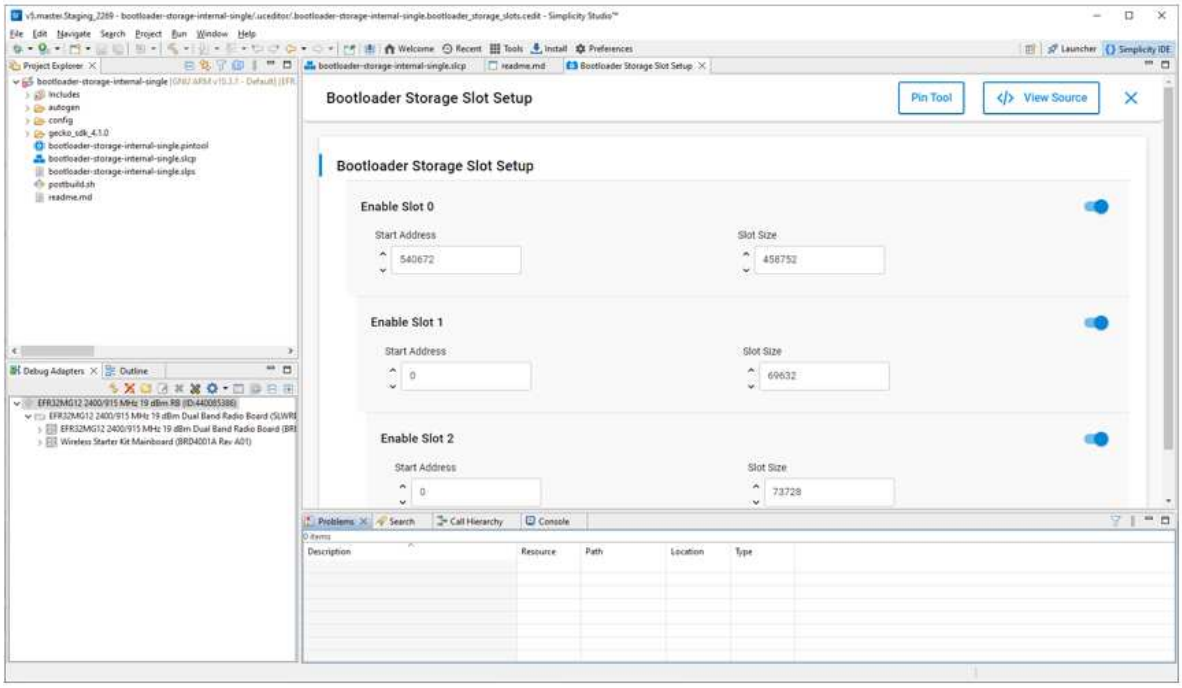


3. The Software Components tab shows the list of available components that can be installed in the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

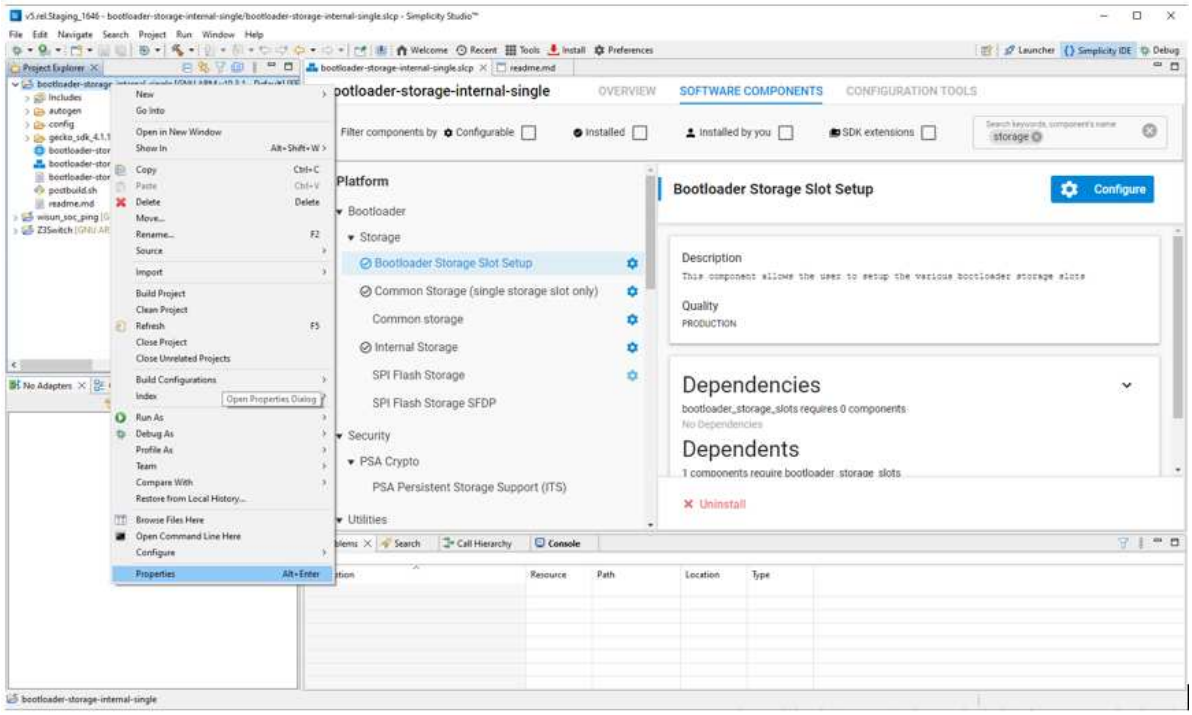


4. The Storage Slot Setup component allows you to configure storage slots to be used if a storage component is also installed. The default configuration matches the target part and bootloader type. This component supports a maximum of three storage slots.

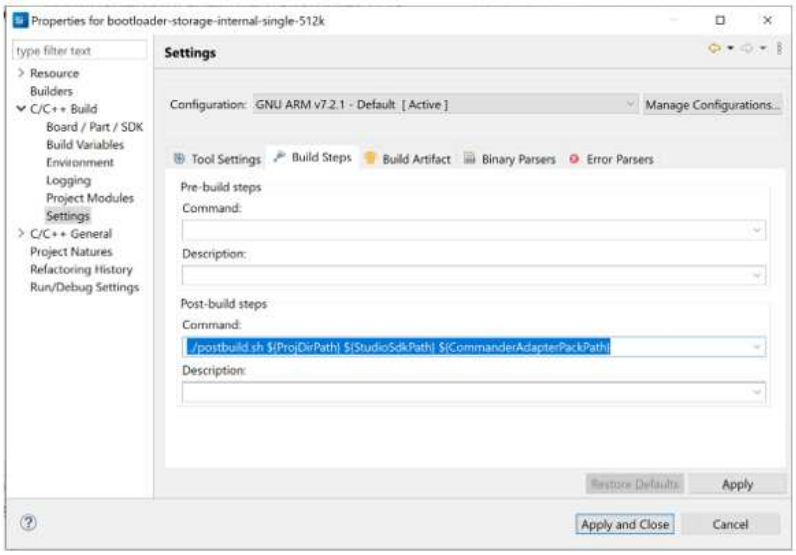


5. Right-click the project name in the Project Explorer view and select Properties.

Copyright © 2025 Silicon Laboratories. All rights reserved.



6. In the C/C++ Build group, click Settings. On the Build Steps tab, in the Post Build Steps Command field enter `../postbuild.sh "${ProjDirPath}" "${StudioSdkPath}" "${CommanderAdapterPackPath}"`. Click Apply and Close.



7. Click the Build (hammer) icon.

On Series 1 devices, three bootloader images are generated into the build directory: a main bootloader, a main bootloader with CRC32 checksum, and a combined first stage and main bootloader with CRC32 checksum. The main bootloader image is called <project-name>.s37, the main bootloader with CRC32 checksum is called <projectname>-crc.s37, while the combined first stage image + main bootloader image with a CRC32 checksum is called <projectname>-combined.s37. The first time a device is programmed, whether during development or manufacturing, the combined image needs to be programmed. For subsequent programming, when a first stage bootloader is already present on the device, it is okay to download an image containing just the main bootloader. In this case, the main bootloader with CRC32 should be used.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The requirement is that any main bootloader image that is programmed via serial wire must contain the CRC32 in the image. Files downloaded via serial wire are "s37" files. Most often, the <projectname>-combined.s37 file is the one downloaded during production programming. However, it is possible to download only the main bootloader over serial wire, in which case <projectname>-crc.s37 should be used.

Any main bootloader that is upgraded with the OTA or host method should already contain CRC32 because bootloader-initiated upgrades use GBL files (not "s37" files) and Simplicity Commander adds the CRC32 when it constructs the GBL file. The input files to Simplicity Commander can (and should) use the non-CRC "s37" file.

On Series 2 devices, the combined image is not present, since the first stage bootloader does not exist. The image containing only a main bootloader is the image that must be used to create a GBL file for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader Fundamentals

Bootloader Fundamentals

NOTE: This section replaces *UG103.06: Bootloader Fundamentals*. Further updates to this user guide will be provided here.

This guide introduces bootloading for Silicon Labs networking devices. It describes the concepts of standalone and application bootloaders and discusses their relative strengths and weaknesses. In addition, it looks at design and implementation details for each method. Finally, it describes the bootloader file format.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Introduction

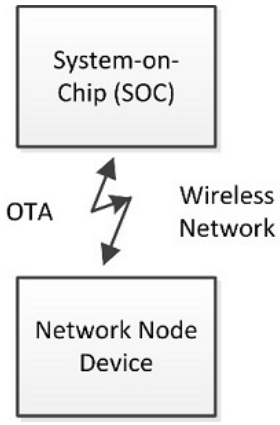
The bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and possibly perform some integrity checks. Firmware image update occurs on demand, either by serial communication or over the air. Production-level programming is typically done during the product manufacturing process yet it is desirable to be able to reprogram the system after production is complete. More importantly, it is valuable to be able to update the device's firmware with new features and bug fixes after deployment. The firmware image update capability makes that possible.

Silicon Labs supports devices that do not use a bootloader, but this requires external hardware such as a Debug Adapter (Silicon Labs ISA3 or Wireless Starter Kit (WSTK)) or third-party SerialWire/JTAG programming device to update the firmware. Devices without a bootloader have no supported way of updating the firmware once they are deployed, which is why Silicon Labs strongly advocates implementing a bootloader.

In March of 2017, Silicon Labs introduced the Gecko Bootloader, a code library configurable through Simplicity Studio's IDE to generate bootloaders that can be used with a variety of Silicon Labs protocol stacks. The Gecko Bootloader can be used with EFM32 and EFR32 Series 1 and later devices. The Gecko Bootloader was restructured into a component-based design and released with Gecko SDK Suite (GSDK) 4.0 in December of 2021. This new version is documented in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher, along with other documents. Documentation for older versions is installed with their respective SDKs.

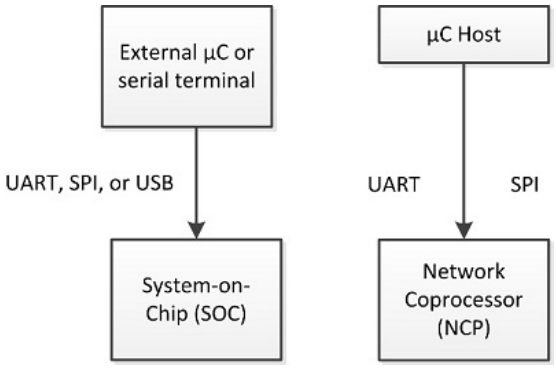
The Gecko Bootloader uses a customized update image file format. The update image file consumed by a Gecko Bootloader-generated application bootloader is a GBL (Gecko BootLoader) file, and is described in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Bootloading a firmware update image can be accomplished in two ways. The first is Over-The-Air (OTA), that is, through the wireless network, as shown in the following figure.



The second is through a hardwired link to the device. The following figure represents the serial bootloader use cases for SoCs (System on Chips) using either a UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Protocol Interface), or USB (Universal Serial Bus) interface, and for NCPs (Network Coprocessors) using either UART or SPI.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Silicon Labs networking devices use bootloaders that perform firmware updates in two different modes: standalone (also called standalone bootloaders) and application (also called application bootloaders). Application bootloaders are further divided into those that use external storage for the download update image, and those that use local storage. These bootloader types are discussed in the next two sections.

The firmware update situations described in this document assume that the source node (the device sending the firmware image to the target through a serial or OTA link) acquires the new firmware through some other means. For example, if a device on the local Zigbee network has an Ethernet gateway attached, this device could get or receive these firmware updates over the Internet. This necessary part of the firmware update process is system-dependent and beyond the scope of this document.

Standalone Bootloader

A standalone bootloader is a program that uses an external communication interface, such as UART or SPI, to get an application image. Standalone firmware update is a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. Very little interaction occurs between the standalone bootloader and the application running in flash. In general, the only time that the application interacts with the bootloader is when it requests a reboot into the bootloader. Once the bootloader is running, it receives firmware update packets containing the (new) firmware image either by physical connections such as UART or SPI, or by the radio (over-the-air).

When the firmware update process is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the application cannot be recovered, and the process must start over. For information about configuring the Gecko Bootloader as a standalone bootloader, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Application Bootloader

An application bootloader begins the firmware update process after the running application has completely downloaded the update image file. The application bootloader expects that the image either lives in external memory accessible by the bootloader or in a portion of main flash memory (if the chip has sufficient memory to support this local storage model).

The application bootloader relies on the application to acquire the new firmware image. This image can be downloaded by the application in any way that is convenient (UART, over-the-air, etc.) but it must be stored into a region referred to as the download space. The download space is typically an external memory device such as an EEPROM or dataflash, but it can also be a section of the chip's internal flash when using a local storage variant of the application bootloader. Once the new image has been stored, the application bootloader is then called to validate the new image and copy it from the download space to flash.

Since the application bootloader does not participate in acquiring the image, and the entire image is downloaded before the firmware update process is started, download errors do not adversely affect the running image. The download process can be restarted or paused to acquire the image over time. The integrity of the downloaded update image can be verified before initiating the firmware update process, to prevent a corrupt or non-functional image from being applied.

The Gecko Bootloader can be configured to accept a list of multiple upgrade images to attempt to verify and apply. This allows the Gecko Bootloader to store what is in effect a backup copy of the update image, which it can access if the first image is corrupt.

Note that the EmberZNet NCP platform does not utilize an application bootloader because the application code resides on the host rather than on the NCP directly. Instead a device acting as a serial coprocessor would utilize a standalone bootloader designed to

Copyright © 2025 Silicon Laboratories. All rights reserved.

accept code over the same serial interface as the expected NCP firmware uses. However, the host application (residing on a separate MCU from the NCP) can utilize whatever bootloading scheme is appropriate.

For more information on application bootloaders, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

About the Gecko Bootloader

About the Gecko Bootloader

The Silicon Labs Gecko Bootloader is a configurable code library that can be used with all the newer Silicon Labs Gecko MCUs and wireless MCUs. It uses a specially-formatted update image file called a GBL file. The Gecko Bootloader has a two-stage design on Series 1 devices, where a minimal first stage bootloader is used to update the main bootloader. On Series 2 devices, the first stage bootloader is replaced by a Secure Engine and the Gecko Bootloader consists only of the main bootloader. The Secure Engine may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- HSE: Hardware Secure Engine, either with or without Secure Vault if not specified
- VSE: Virtual Secure Engine
- SE: Secure Engine (either HSE or VSE, in general)

Having a first stage bootloader or SE allows for field updates of the main bootloader, including adding new capabilities, changing communication protocols, adding new security features and fixes, and so on. The Gecko Bootloader consists of three component parts:

Core: The bootloader core contains the main function of both bootloader stages. It also contains functionality to write to the internal main flash, to perform a bootloader update, and to reset into the application flagging applicable reset reasons.

Driver: Different bootloading applications require different hardware drivers for use by the other components of the bootloader.

Component/Plugin: All parts of the main bootloader that are either optional or selectable for different configurations are implemented as components (in GSDK 4.0 and higher) or previously in plugins. Each component/plugin has a generic header file, and one or more implementations. The current release contains components for functionality like UART and SPI communication protocols, SPI flash storage, internal flash storage, and different cryptographic operations.

Features

Gecko Bootloader features include:

- Field-updatable
- Secure boot
- Signed GBL firmware update image file
- Encrypted GBL firmware update image file

These features are summarized in the following sections and described in more detail in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher. Protocol-specific information about using the Gecko Bootloader may be found in the following documents:

- Using the Gecko Bootloader with EmberZNet
- Bootloading and OTA with Silicon Labs Connect SDK v3.x
- Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

Field-Updatable

Series 1

On EFM32 and EFR32 Series 1 devices, field update capability for the Gecko bootloader is provided by a two-stage design where the bootloader has a first stage and a main stage. The minimal first stage of the bootloader is not field updatable, and can only update the main bootloader by reading from and writing to fixed addresses in internal flash memory. To perform a main bootloader update, the running main bootloader verifies the integrity and authenticity of the bootloader update image, writes it to internal flash,

Copyright © 2025 Silicon Laboratories. All rights reserved.

and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader update image before copying it to the main bootloader location, completing the update

Series 2

On Series 2 devices, field update capability for the Gecko bootloader is provided by the SE. To perform a main bootloader update, the running main bootloader verifies the integrity and authenticity of the bootloader update image, writes it to internal flash, and requests that the SE installs the update. The SE optionally verifies the authenticity of the main bootloader update image before copying it to the main bootloader location, completing the update. The same mechanism can be used to update the SE itself.

Secure Boot

Secure boot is designed to prevent an untrusted application from running on the device. When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot using asymmetric cryptography. The signature algorithm used is ECDSA-P256-SHA256. The public key is written to the device during manufacturing, while the private key is kept secret. This ensures that the application was created and signed by a trusted party.

Signed GBL Update Image File

The Gecko Bootloader supports enforcing cryptographic signature verification of the update image file in addition to Secure Boot. This allows the bootloader and application to verify that the application or bootloader update comes from a trusted source before starting the update process. The signature algorithm used is ECDSA-P256-SHA256. The public key is the same key as for secure boot, written to the device during manufacturing, while the private key is never distributed. This ensures that the GBL file was created and signed by a trusted party.

Encrypted GBL Update File

The GBL update file can also be encrypted, to prevent eavesdroppers from getting hold of the plaintext firmware image. The encryption algorithm used is AES-CTR-128, and the encryption key is written to the device during manufacturing.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Memory Space for Bootloading

Memory Space for Bootloading

The first stage of the Gecko Bootloader on Series 1 devices takes up a single flash page. On devices with 2 kB flash pages, like EFR32MG1, this means that the first stage takes 2 kB.

The size of the main bootloader is dependent on the functionality required. With a typical bootloader configuration, the main bootloader for Series 1 devices takes up 14 kB of flash, bringing the total bootloader size to 16 kB.

Silicon Labs recommends reserving 16 kB for the bootloader for Series 1 and EFR32xG21 devices and 24 kB for EFR32xG22 devices.

On EFR32xG1 devices (Mighty Gecko, Flex Gecko, and Blue Gecko families), the bootloader resides in main flash.

- First stage bootloader @ 0x0
- Main bootloader @ 0x800
- Application @ 0x4000

On EFR32xG12 and later Series 1 devices, the bootloader resides in the bootloader area in the Information Block.

- Application @ 0x0
- First stage bootloader @ 0x0FE10000
- Main bootloader @ 0x0FE10800

On EFR32xG21, the main bootloader resides in main flash:

- Main bootloader @ 0x0
- Application @ 0x4000

On EFR32xG22, the main bootloader resides in main flash:

- Main bootloader @ 0x0
- Application @ 0x6000

On EFR32xG23, the main bootloader resides in main flash:

- Main bootloader @ 0x08000000
- Application @ 0x08006000

On EFR32xG24, the main bootloader resides in main flash:

- Main bootloader @ 0x08000000
- Application @ 0x08006000

Copyright © 2025 Silicon Laboratories. All rights reserved.

Design Decisions

Design Decisions

The decision of what bootloader type to deploy depends on many factors. Note that the platform type and available flash memory may limit bootloader choices.

Some questions related to this are:

- Where does the device get the new update image? Is this over-the-air via the networking protocol? Using a separate interface connected to the Internet?
- Will the device have an external memory chip to store a new update image? If not, is there enough internal flash memory to store both a current and a newly downloaded copy of the largest expected application image?
- If the device receives the new image over-the-air, will it be multiple hops away from the server holding the download image?
- What kind of image security is needed?
- What communications driver will be used (in the single protocol case)?
- Does the use case require more than one protocol?

The configurable design of the Gecko Bootloader platform means that developers can create bootloaders to fit almost any design choice. See Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Gecko Bootloader Developer's Guide

The Developer's Guide content is organized in three groups:

- Developing and Debugging: Includes a detailed Gecko Bootloader User's Guide and information about secure bootloading.
- Security: Brings together information about security considerations when bootloading.
- Protocol-Specific Information: Contains information about using the Gecko Bootloader with various protocols.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Developing and Debugging a Gecko Bootloader

These pages provide detailed information for Gecko Bootloader Developers.

- Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher: Describes the high-level implementation of the Silicon Labs Gecko Bootloader for Series 3 microcontrollers, SoCs (System on Chips), and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader.
- Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices): Describes the high-level implementation of the Silicon Labs Gecko Bootloader for EFM32 and EFR32 Series 1 and Series 2 microcontrollers, SoCs (System on Chips) and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader.
- Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher: Gecko Bootloader v2.x, introduced in GSDK 4.0, contains a number of changes compared to Gecko Bootloader v1.x. This describes the differences between the versions, including how to configure the new Gecko Bootloader in Simplicity Studio 5.
- Simplicity Commander Reference Guide: Describes how and when to use the Command-Line Interface (CLI) of Simplicity Commander. Simplicity Commander supports all EFR32 Wireless SoCs, EFR32 Wireless SoC modules (such as the MGM111 or MGM12P), EFM32 MCU families, and EM3xx Wireless SOCs.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader User's Guide for Series 3 and Higher

Silicon Labs Gecko Bootloader User’s Guide for Series 3 and Higher

This guide describes the high-level implementation of the Silicon Labs Gecko Bootloader for Series 3 microcontrollers, SoCs (System on Chips), and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader.

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. This guide is specific to Series 3 devices and above. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Usable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
- Secure Boot: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
- Signed upgrade image file: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
- Encrypted upgrade image: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

For Series 3 devices and above, Gecko Bootloader uses a proprietary format for its upgrade images, called GBL V4 (Gecko Bootloader file). These files have the file extension “.gbl”. See the *Gecko Bootloader File Format* page for more details. Series 3 and above devices do not support GBL versions lesser than v4.

On Series 3 devices, the Gecko bootloader consist of the main stage bootloader. The main bootloader is upgradable through the Secure Engine.

The Secure Engine provides functionality to install an image to code region 0 in the external flash.

To perform a bootloader upgrade, the running bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The bootloader then writes the upgrade image to the upgrade location in external flash and requests that the Secure Engine installs it. On some devices, the Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism. See the *Gecko Bootloader Operation - Secure Engine Upgrade* page for more details.

The bootloader consists of a common core, drivers, and a set of components that give the bootloader specific capabilities. The common bootloader core is provided as a full-source delivery. The common bootloader core contains functionality to parse GBL v4 files and flash their contents to the device.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be installed in and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image by a

Copyright © 2025 Silicon Laboratories. All rights reserved.

physical connection such as UART or SPI. To function as a standalone bootloader with a physical connection, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by writing the firmware upgrade image to a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, Ethernet, USB, and so on). The download space is data flash or a section of the device's flash. The Gecko Bootloader can partition the download space into multiple storage slots and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation must be configured.

Silicon Labs provides example bootloaders that come with a preconfigured set of installed components for configuration in either standalone or application mode. See the *Configuring the Gecko Bootloader* page. The following sections provide an overview of the Gecko Bootloader common core, drivers, and components. For details, including details on error codes and conditions, see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

Core

The bootloader core contains the bootloader's main functions. It also contains functionality to write to the external flash, an image parser to parse and act upon the contents of GBL v4 upgrade files, and functionality to boot the application in main flash.

A version of the GBL v4 image parser without support for encrypted upgrade images is also available. This version can be used in flash space constrained bootloader applications where encryption of the upgrade image is not required.

Shared Memory

To exchange information between the bootloader and application, a section of SRAM is used. The contents of SRAM are preserved through a software reset, making SRAM suitable as a communication channel between bootloader and application.

The shared memory has a size of 4 bytes, and is located at the first address of SRAM, 0x20000000. It is used to store a single word containing the reason for a reset. The structure of the reset cause word is defined in the Reset Information part of the Application Interface, in the file btl_reset_info.h, as 16 bits containing the reason, and 16 bits of signature indicating if the word is valid or not. If the signature reads 0xF00F, the reset reason is valid.

All 16-bit reset reasons used by Silicon Labs have the most significant bit set to zero. If custom reset reasons are desired, it is recommended to set the most significant bit to avoid conflicting definitions.

Drivers

Different applications for firmware upgrade require different hardware drivers for use by the other components of the bootloader.

Driver modules include:

- Delay: Simple delay routines for use with components that require small delays or timeouts.
- SPI Slave: Flexible SPI Slave driver implementation for use in communication components implementing SPI protocols. This driver supports both blocking and non-blocking operation, with DMA (Direct Memory Access) backing the background transfers to support non- blocking operation.
- UART: Flexible serial UART driver implementation for use in communication components implementing UART protocols. This driver supports both blocking and non-blocking operations, with DMA backing the background transfers to support non-blocking operation. Additionally, support for hardware flow control (RTS/CTS) is included.

Components

All parts of the bootloader that are either optional or that may be exchanged for different configurations are implemented as components. Each component may have a configuration header file, and one or more implementations. Components include:

- Communication
- UART: XMODEM
- UART: BGAPI
- Compression
- Debug

Copyright © 2025 Silicon Laboratories. All rights reserved.

- GPIO Activation
- Security
- Storage

Communication

Communication components provide an interface for implementing communication with a host device, such as a computer or a micro- controller. Several components implement the communication interface, using different transports and protocols.

- BGAPI UART DFU: By enabling the BGAPI communication component, the bootloader communication interface implements the UART DFU protocol using BGAPI commands.
- UART XMODEM: By enabling the UART XMODEM communication component, the bootloader communication interface implements the XMODEM-CRC protocol over UART. This component makes the bootloader compatible with the legacy serial-uart-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.

Compression

Compression components provide capability for the bootloader GBL file parser to handle compressed GBL v4 upgrade images. Each compression component provides support for one (de)compression algorithm. At the time of writing, decompression of data compressed with the LZ4 and LZMA algorithms is supported, through the *GBL Compression (LZ4)* and *GBL Compression (LZMA)* components.

Debug

This component provides the bootloader with support for debugging output. If the component is configured to enable debug prints, short debug messages will be printed over Serial Wire Output (SWO), which can be accessed in multiple ways, including using Simplicity Commander, and by connecting to port 4900 of the Wireless Starter Kit TCP/IP interface.

To turn on debug prints, enable the Debug component and select Debug prints. Select Debug asserts to enable assertions in the source code.

GPIO Activation

This component provides functionality to enter firmware upgrade mode automatically after reset if a GPIO pin is active during boot. The GPIO pin location and polarity are configurable.

- GPIO: By enabling the GPIO activation component, the firmware upgrade mode can be activated by push buttons.

Security

Security components provide implementations of cryptographic operations as well as functionality to compute checksums and to read cryptographic keys from manufacturing tokens.

Modules include:

- AES: AES decryption functionality
- CRC16: CRC16 functionality
- CRC32: CRC32 functionality
- ECDSA: ECDSA signature verification functionality
- SHA-256: SHA-256 digest functionality

Storage

These components provide the bootloader with multiple storage options for SoCs. All storage implementations must provide an API to access image files to be upgraded. This API is based on the concept of dividing the download space into storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Some storage implementations also support a raw storage API to access the underlying storage medium. This can be used by applications to store other data in parts of the storage medium that are not used for storing firmware upgrade images. Implementations include:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- External Flash: The external flash storage implementation uses the external flash of the device for upgrade image storage. Note that this storage area is only a download space and is separate from the portion of external flash used to hold the active application code.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader File Format v4

Gecko Bootloader File Format v4

The Gecko Bootloader supports GBL File format v4 for Series 3 device. This format has significant differences as compared to version 3, The file formats described in this section are generated by Simplicity Commander commands.

File Structures

The GBL v4 file format is composed of nested TLV (Type-Length-Value) elements. The format of the TLV is as follows:

Field	Length (Bytes)	Description
Type	4	Type indicates the content of the Value field
Length	4	Length of the Value field in bytes
Value	N	Length size field containing either more TLVs or data

There are two types of TLVs, data and container. Data TLVs value field contains data whose format is specified in the type field and the container TLVs value filed contains more TLVs, which in turn again may contain data or container TLVs.

TLV Descriptions

This section describes each of the TLVs.

GBLV4

This mandatory TLV is used to identify the GBLv4 file when parsing and tell the length of the file.

Type	Length	Value
0x84A617EB	Variable	Contains all the TLVs for the file

MANIFEST

MANIFEST TLV is present inside the GBLV4 TLV. It has a collection, TLVs, that describes the metadata of the GBLv4 file that is used during the update process.

Type	Length	Value
0xAA01012A	Variable	Contains metadata TLVs for applying the file.

MANIFEST_INFO

The MANIFEST_INFO TVL contains the `ManifestInfo_t` structure that describes the version dependency and the features of the GBL file.

Type	Length	Value
0x2B03032B	Variable	manifest_info structure

The `ManifestInfo_t` structure is as follows:

Data Type	Field	Purpose
uint32_t	version	Minimum version of the GLBv4 parser required, used for future extensions.
uint32_t	features	Features requested from the parser (encryption, compression etc.)

Copyright © 2025 Silicon Laboratories. All rights reserved.

BUNDLE_VERSION

This is the child TVL of MANIFEST TLV, required to identify the bundle/file and is used for versioning and rejection of downgrades.

Type	Length	Value
0x2B04042B	Fixed	bundle_version structure

The `BundleVersion_t` structure is as follows:

Data Type	Field	Purpose
Uint8_t	productId	128 bit unique product ID
uint32_t	bundleVersion	Reject if smaller than this exists
Uint32_t	minVersion	The minimum previous version to which this update file can be applied

CONTENT_HASH

An optional TLV that provides integrity checking on the remainder of the file. This TLV is applicable to batch updates only.

Type	Length	Value
0x0x2B05052BUL	Fixed	content_hash structure

The `content_hash` structure is as follows:

Data Type	Field	Purpose
uint32_t	hashType	The hash function used to validate (1=SHA, etc.)
uint8_t	sha[]	The hash of all contents following the manifest

MANIFEST_CERTIFICATE

Type	Length	Value
0x2B01012B	Fixed	ApplicationCertificate_t

The `ApplicationCertificate_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	structVersion	Vesion of the certificate structure
uint8_t	flags	Reserved flags
uint8_t	key	Public key used to verify GBL file
uint32_t	version	Version number of this certificate
uint8_t	signature	The signature of the certificate itself (not the GBL file)

MANIFEST_SIGNATURE

Type	Length	Value
0x2B02022BUL	Variable	Manifest signature structure

The `Manifest_signature` structure is as follows:

Data Type	Field	Purpose
uint32_t	Signature function	The function used for signing

Copyright © 2025 Silicon Laboratories. All rights reserved.

Data Type	Field	Purpose
uint8_t[]	Signature	The signature for authentication

UPDATE_PROCESS

This is a collection of TLVs that describes the update process. Parsing this TLV does not happen until the MANIFEST_TLV is authenticated. Each TLV contained in this is parsed and processed one at a time in the order they are listed.

Type	Length	Value
0xAB06062BUL	Variable	TLVs describing the steps to update the system

UPDATE_SE

This is an optional TLV that is used to update the SE.

Type	Length	Value
0x2C02022CUL	Fixed	UpdateSe_t structure

The `UpdateSe_t` structure is as follows:

Data Type	Field	Purpose
uint32_t	version	Version number of the SE update image.
uint32_t	tlvPosition	The absolute position of the SE update image Blob TLV

UPDATE_MEMORY_SECTION

The memory section's content is present in the file and needs to be updated.

Type	Length	Value
0x2C03032CUL	Fixed	UpdateMemorySection_t structure

The `UpdateMemorySection_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	targetMemory	Which memory this definition is for (0=main memory, 0x1-0x7f = reserved, 0x80-0xff=defined by the user)
uint8_t	plainImageSize	Represents the size of the plain image in 3 bytes
uint32_t	targetAddr	The target memory address for the update
uint32_t	type	Bitfield representing the type of application
uint32_t	version	Version number for this application (customer-defined)
uint32_t	capabilities	Bitfield representing the capabilities of this application
uint32_t	memorySectionPos	The absolute position of the MEMORY_SECTION_TLV in the file
HashVaule_t	memSecHash	A hash value for verifying the integrity of the memory section

MANIFEST_FINISH

TLV to indicate the end of an update.

Type	Length	Value
0x2C04042CUL	Fixed	Empty

MEMORY_SECTION

Copyright © 2025 Silicon Laboratories. All rights reserved.

This TLV contains TLVs for a memory section update.

Type	Length	Value
0xBA01013AUL	Variable	TLVs for the region

MEMORY_SECTION_INFO

This TLV contains information about how the blob data is read and where the data is to be written to. This is a common wrapper for all data as it allows the data to be compressed, verified and processed in a unified manner.

The data can be divided into blocks, with each block having its own hash. This can then be compared with the hashes in the memory section info to authenticate the block. This is useful during streaming updates to avoid writing unauthenticated data to flash.

Type	Length	Value
0x3B01013BUL	Fixed	MemSectionPartialInfo_t

The `MemSectionPartialInfo_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	compressionScheme	0 = No compression 1 = LZ4 compression 2 = LMZA compression
uint8_t	encryptionScheme	0 = No encryption 1 = AES CCM
uint8_t	secureBootScheme	Reserved for future use.
uint8_t	reserved	
uint16_t	signBlockSize	Block size if DFU authentication is done for each block.
uint16_t	numBlocks	Number of blocks inside the BLOB.
uint8_t[]	nonce	Nonce for encryption
uint8_t[]	finalImageHash	Reserved for future use.
uint8_t[]	secureBootSign	Reserved for future use.

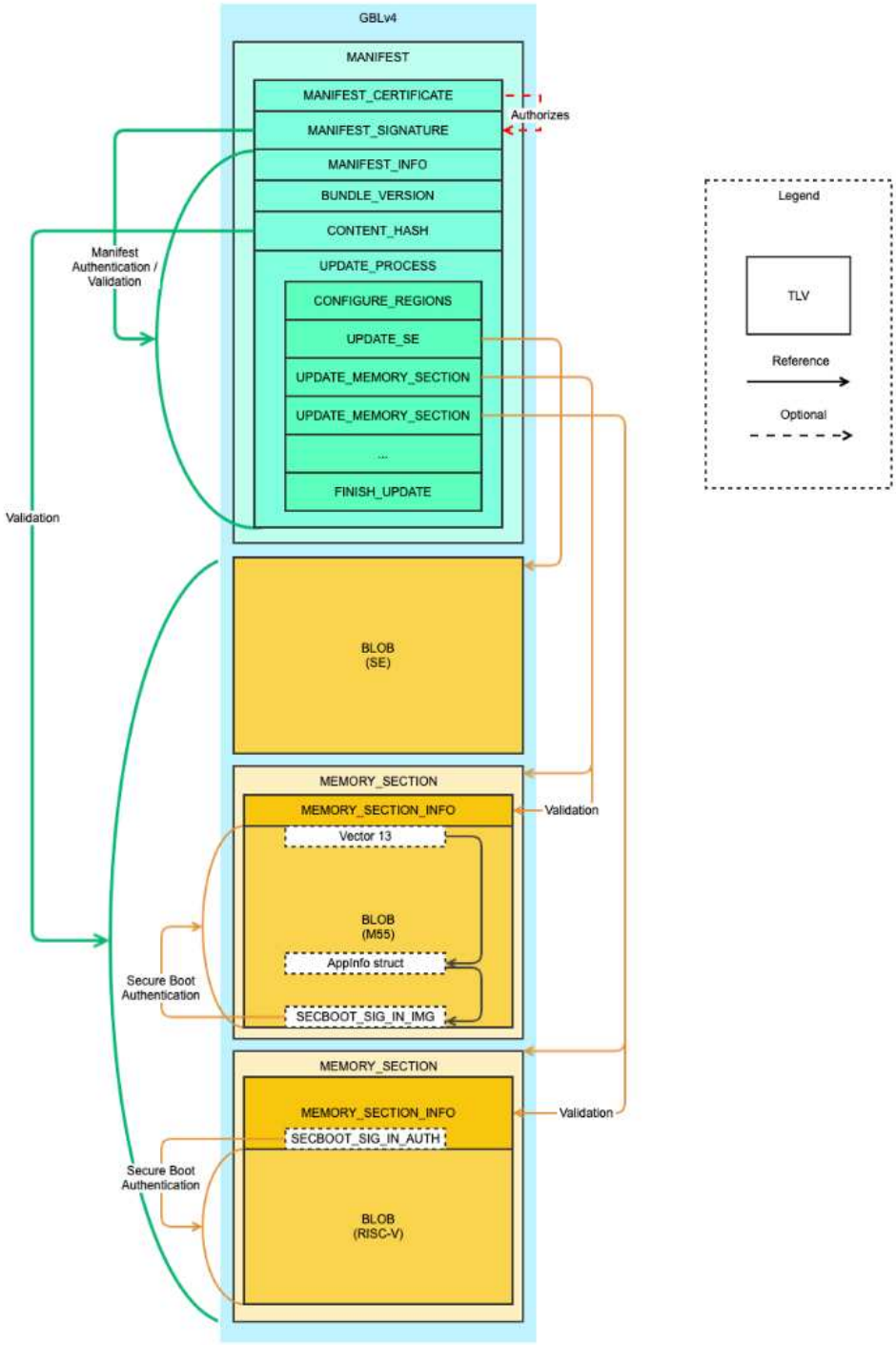
BLOB

This TLV contains variable length binary data.

Type	Length	Value
0x3B02023BUL	Variable	Binary Data

The figure below shows the structure of a GBL v4 file.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

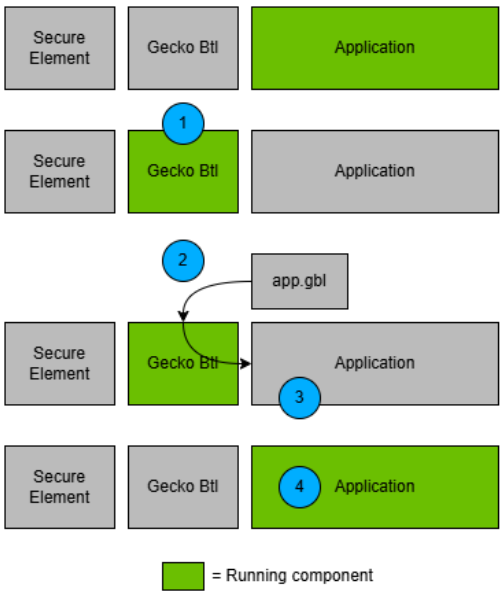
Gecko Bootloader Operation Application Upgrade

Gecko Bootloader Operation - Application Upgrade

This section summarizes Gecko Bootloader operation for updating application firmware, first if the Gecko Bootloader is configured in standalone mode and then if it is configured in application mode. The *Gecko Bootloader Operation - Bootloader Upgrade* provides the same information for updating the bootloader firmware.

Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing an application image is transmitted from the host to the device. If image encryption is enabled in the bootloader and the image is encrypted, decryption is performed during the process of receiving and parsing the GBL file.
3. The bootloader applies the application upgrade from the GBL upgrade file on-the-fly. If image authentication is enabled in the bootloader and the GBL file contains a signature, the authenticity of the image is verified before completing the process.
4. The device boots into the application. Application upgrade is complete.

Rebooting Into the Bootloader

The Gecko Bootloader supports multiple mechanisms for triggering the bootloader. If the GPIO activation component is installed, the host device can keep this pin low/high (depending on configuration) through reset to make the device enter the bootloader. The bootloader can also be entered through software. The `bootloader_rebootAndInstall` API first signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it will enter firmware upgrade mode instead of booting the existing application.

Downloading and Applying a GBL Upgrade File

When the bootloader enters firmware upgrade mode, it enters a receive loop waiting for data from the host device. The specifics of the receive loop depend on the protocol. Received packets are passed to the image parser, a state machine that parses the data and returns a callback containing any data that should be acted upon. The bootloader core implements this callback and flashes the data to external flash at the specified address. If GBL file authentication or encryption is enabled, the image parser will enforce

Copyright © 2025 Silicon Laboratories. All rights reserved.

this, and abort the image upgrade if the authentication fails The bootloader prevents a newly uploaded image from being bootable by holding back parts of the application vector table until the GBL file hash and GBL signature (if required) have been verified.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

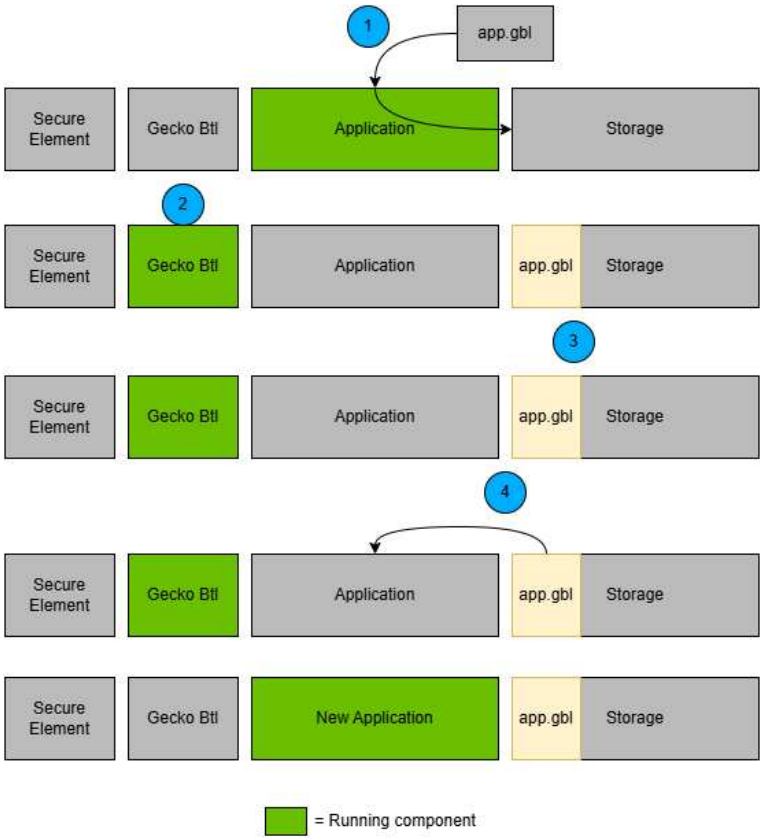
Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid and also if the lock bit is set. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Error Handling

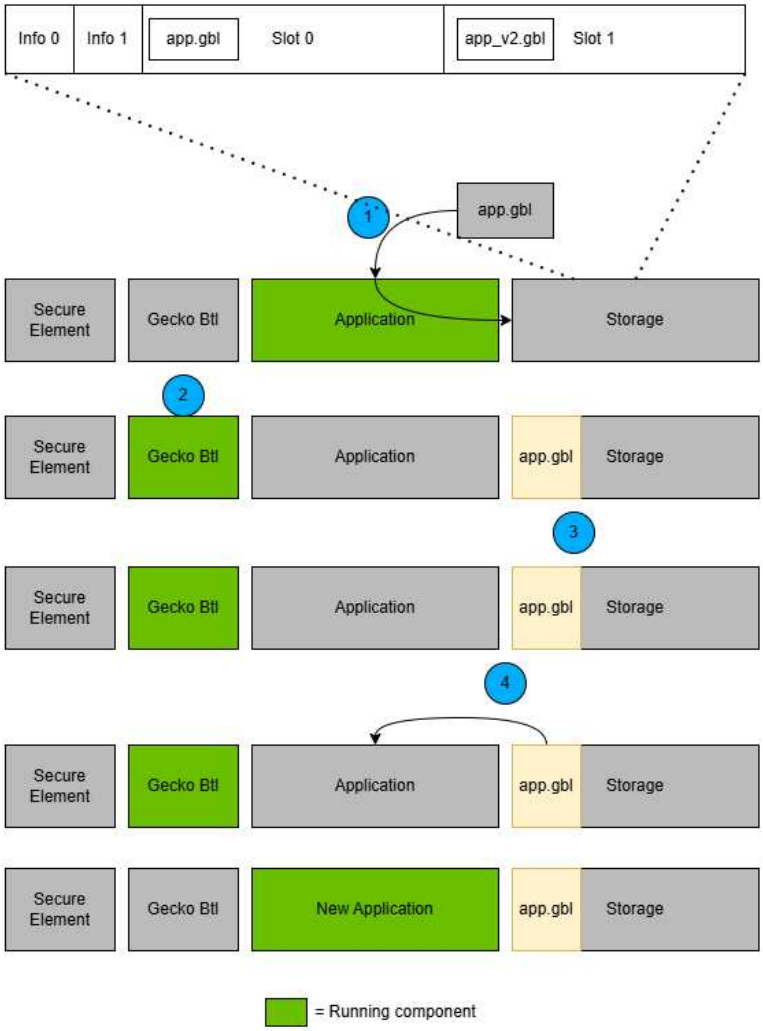
If the application upgrade is interrupted at any time, the device will be without a working application. The bootloader then resets the device, and re-enters firmware upgrade mode. The host device can easily restart the application upgrade process, to try loading the upgrade image again.

Application Bootloader Operation

The following figure illustrates the application bootloader operation both for a single image/single storage slot, and multiple images/ multiple storage slots.



Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A GBL file is downloaded onto the storage medium of the device, as described below, and the presence of an upgrade image is indicated.
2. The device reboots into the bootloader, and the bootloader enters firmware upgrade mode.
3. The device applies the application upgrade.
4. The device boots into the application. Application upgrade is complete.

Downloading and Storing a GBL Upgrade Image File

To prepare for receiving an upgrade image, the application finds an available storage slot or erases an existing one using `bootloader_eraseStorageSlot`. If the bootloader only supports a single storage slot, a value of 0 should be used for the slot ID.

The application then receives a GBL file using an applicable protocol, such as Ethernet, USB, Zigbee, OpenThread or Bluetooth, and stores it in the slot by calling `bootloader_writeStorage`.

When the download is complete, the application can optionally verify the integrity of the GBL file by calling `bootloader_verifyImage`. This is also done by the bootloader before applying the image but can be done from the application to avoid rebooting into the bootloader if the received image was corrupt.

If multiple storage slots are supported, the application should write a bootload list by calling `bootloader_setImageToBootload`. The list is written to the two bootload info pages as shown in the figure above. The bootload list is a prioritized list of slots indicating the order the bootloader should use when attempting to perform a firmware upgrade. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader treats the entire download space as a single storage slot.

Rebooting and Applying a GBL Upgrade File

Copyright © 2025 Silicon Laboratories. All rights reserved.

The bootloader can be entered through software. The `bootloader_rebootAndInstall` API signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it enters firmware upgrade mode instead of booting the existing application.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the image stored in each. Once it finds a valid GBL upgrade file, firmware upgrade is attempted from this GBL file. If the upgrade fails, the bootloader moves to the next image in the list. If no images pass verification, the bootloader reboots back into the existing application with a message in the shared memory location in SRAM indicating that no good upgrade images were found.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid as well as if the lock bit is set. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Bootloader Upgrade

Gecko Bootloader Operation - Bootloader Upgrade

Bootloader upgrade functionality is provided by the Secure Engine on Series 3 devices. The Secure Engine itself is also upgradable. For more details, see the *Gecko Bootloader Operation - Secure Engine Upgrade* page.

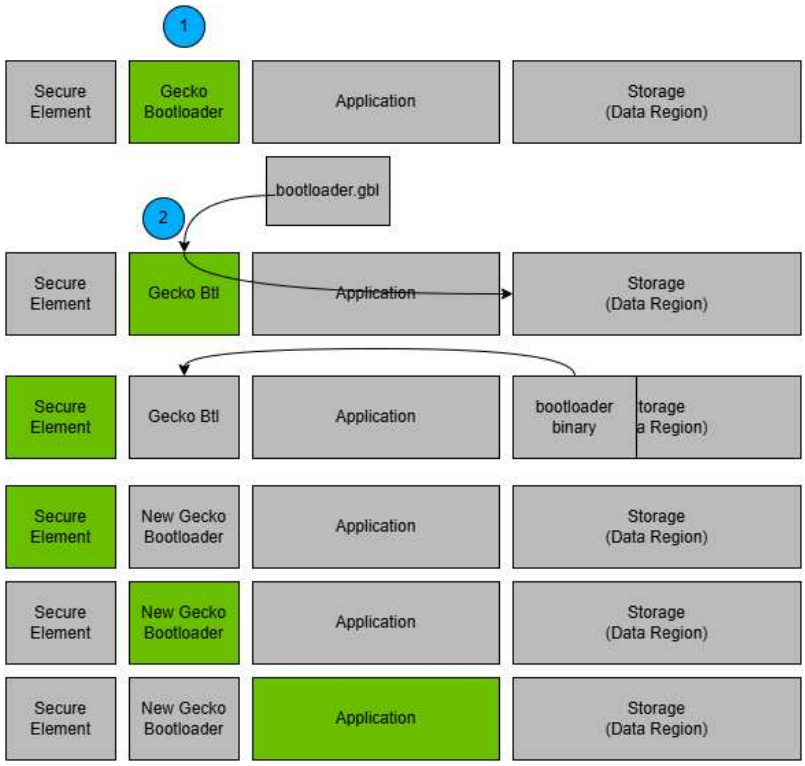
Requirements for upgrading the main bootloader vary depending on the bootloader configuration:

- Application bootloader with storage: Upgrading the main bootloader requires a single GBL file containing both bootloader and application upgrade images.
- Standalone bootloader with communication interface: Upgrading the bootloader requires a GBL file with only the bootloader upgrade image.

Security of the bootloader upgrade process is provided by signing the GBL file. See *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* on the *Gecko Bootloader Security Features* page.

Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing only a bootloader upgrade image is transmitted from the host to the device.
3. The contents of the GBL Bootloader tag are written to the data region in flash
4. The device reboots into the Secure Engine.
5. The Secure Engine replaces the main bootloader with the new version found in the bootloader upgrade GBL.
6. The device boots into the new main bootloader. Bootloader upgrade is complete.

Copyright © 2025 Silicon Laboratories. All rights reserved.

A bootloader upgrade is started in the same way as an application upgrade.

Downloading and Applying a Bootloader GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a bootloader upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns bootloader upgrade data in a callback. The bootloader core implements this callback and flashes the data to the flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file hash and GBL signature (if required) have been verified.

When a complete bootloader upgrade image is received, the bootloader signals the Secure Engine that it should enter firmware upgrade mode. Secure Engine communication is used to signal that bootloader upgrade is ready to be performed.

The authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See *Setting a Version Number* on the *Configuring the Gecko Bootloader* page for more information about versioning bootloader images.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature, using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
    - Turn on secure boot in Simplicity Studio by going to the Bootloader Core component and selecting the Enable secure boot option.
    - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. Key locations are defined in the bootloader project file btl_security_tokens.c.
4. Create a GBL file using the Gecko Bootloader image. The GBL file needs to be signed/unsigned depending on the current configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
5. Upload the GBL file. For more details on the upgrade process, see the *Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)* section.

Enabling Secure Boot RTSL on Series 3 Devices

Secure Boot RTSL (Root of Trust and Secure Loader) can be enabled using the following procedure:

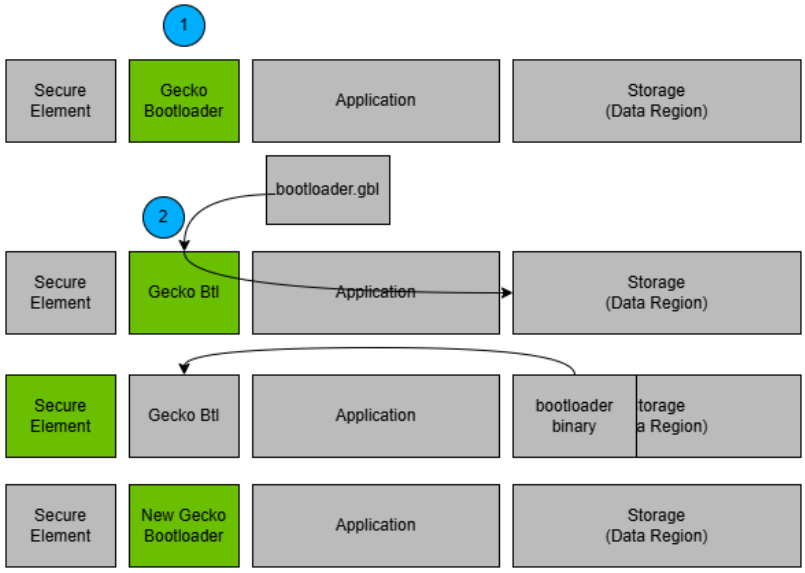
1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
    - Turn on secure boot in Simplicity Studio by going to the Bootloader Core component and selecting the Enable secure boot option.
    - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in the VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an unsigned GBL file from this application and upload it. For more information on installing public keys, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2.

Copyright © 2025 Silicon Laboratories. All rights reserved.

6. Create a GBL file using the Gecko Bootloader image from step 4.
7. Create a GBL file using the application from step 5. The GBL file need to be signed if the Bootloader Core component option Require signed firmware upgrade files was selected in step 1.
8. Upload the GBL file containing the Gecko Bootloader image.
9. Upload the GBL file containing the application.

Bootloader Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



1. A single GBL file containing a bootloader upgrade image is downloaded onto the storage medium of the device.
2. The device reboots into the main bootloader.
3. The main bootloader verifies the integrity of the upgrade image and then resets the device with reset reason BOOTLOADER_RESET_REASON_UPGRADE to apply the upgrade.
4. The device reboots into the Secure Engine.
5. The Secure Engine replaces the main bootloader with the new version.
6. The device boots into the new main bootloader.

A bootloader upgrade is started in the same way as an Application Upgrade. A GBL file containing a bootloader is written to storage by the application, and the bootloader is entered.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the GBL file stored within. Verification returns information about whether the GBL file contains a bootloader, or both a bootloader and an application. The image parser parses the file. If the GBL file contains a bootloader, the bootloader upgrade data is returned in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file hash and GBL signature (if required) have been verified.

Secure Engine communication interface is used to signal the Secure Engine that a bootloader upgrade is ready to be performed.

On Series 3 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See the *Setting a Version Number* section on the *Configuring the Gecko Bootloader* page for more information about versioning bootloader images.

The new main bootloader is entered, and the images in the list of storage slots marked for bootload are verified. When the image parser parses the slot containing the GBL file with the bootloader + application upgrade, the version number of the bootloader upgrade is equal to the running main bootloader version, so another bootloader upgrade will not be performed. Instead, the

Copyright © 2025 Silicon Laboratories. All rights reserved.

application upgrade data are returned in a callback. Bootloading of the new application proceeds as described in the *Application Bootloader Operation* section on the *Gecko Bootloader Operation Application Upgrade* page.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature. The procedure is as follows:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot from the Bootloader Core component in Simplicity Studio by selecting the Enable secure boot option.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device. The Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c`.
4. Prepare a signed application image using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing an application.
5. Create a GBL file using the Gecko Bootloader image and the signed application image. The GBL file needs to be signed/unsigned depending on the configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
6. Upload the GBL file. For more details on the upgrade process, see the *Bootloader Upgrade on Application Bootloaders with Storage* section.

Enabling Secure Boot RTSL on Series 3 Devices

Secure Boot RTSL can be enabled by using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot from the Bootloader Core component in Simplicity Studio by selecting the Enable secure boot option.
   - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an un-signed GBL file from this application and upload it. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine.
6. Create a GBL file using the Gecko Bootloader image from step 4 and the application from step 5. The GBL file must be signed if the Bootloader Core component option Require signed firmware upgrade files was selected in step 1. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
7. Upload the GBL file containing the Gecko Bootloader image and the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Secure Engine Upgrade

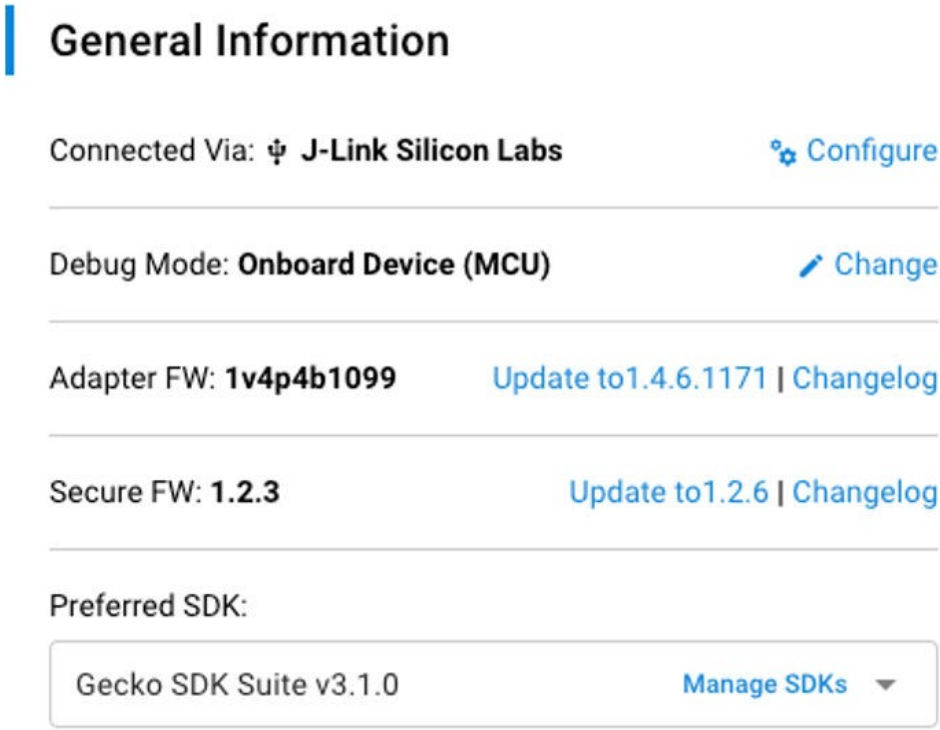
Gecko Bootloader Operation - Secure Engine Upgrade

The Secure Engine is upgradable and for both application and standalone type of bootloader a GBL file containing the Secure Engine Upgrade image has to be flashed or sent to the bootloader.

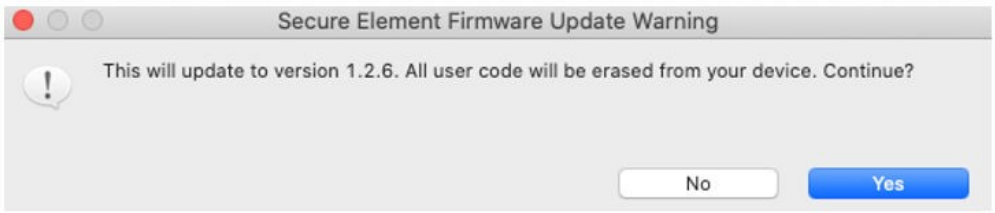
A bootloader upgrade can also be included in the same GBL file in application mode, or as a second GBL file in standalone mode. The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the *Memory Space for Bootloading* section in Bootloader Fundamentals.

Signed and encrypted Secure Engine upgrade images are provided by Silicon Labs through Simplicity Studio. Upgrade images with the same or lower version number than the running Secure Engine will be ignored.

To download Secure Engine firmware images, connect a Series 3 device and select a preferred SDK. The Secure Firmware Update to x.x.x link appears in the Launcher Perspective, as shown in the following figure.



Click Update to x.x.x next to Secure FW: x.x.x. A warning dialog box appears. Click Yes to continue.



Copyright © 2025 Silicon Laboratories. All rights reserved.

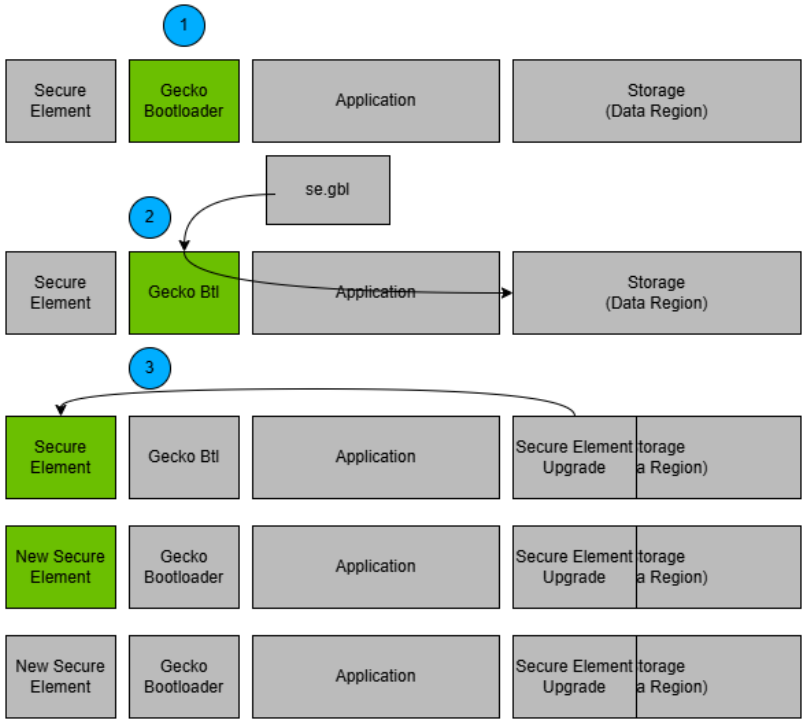
The Launcher Perspective is then updated so that the current Secure Firmware version and link version are the same.



The Secure Engine firmware images can be found in the *util/se_release/public* directory under the Gecko SDK. Simplicity Studio displays the SE firmware version available in the Gecko SDK selected.

Secure Engine Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure.



1. The device reboots into the bootloader.
2. A GBL file containing only a Secure Engine upgrade image is transmitted from the host to the device.
3. The device reboots into the Secure Engine.
4. The Secure Engine is replaced by the new version found in the pre-configured upgrade location.
5. The device boots into the bootloader.

Downloading and Applying a Secure Engine GBL Upgrade File

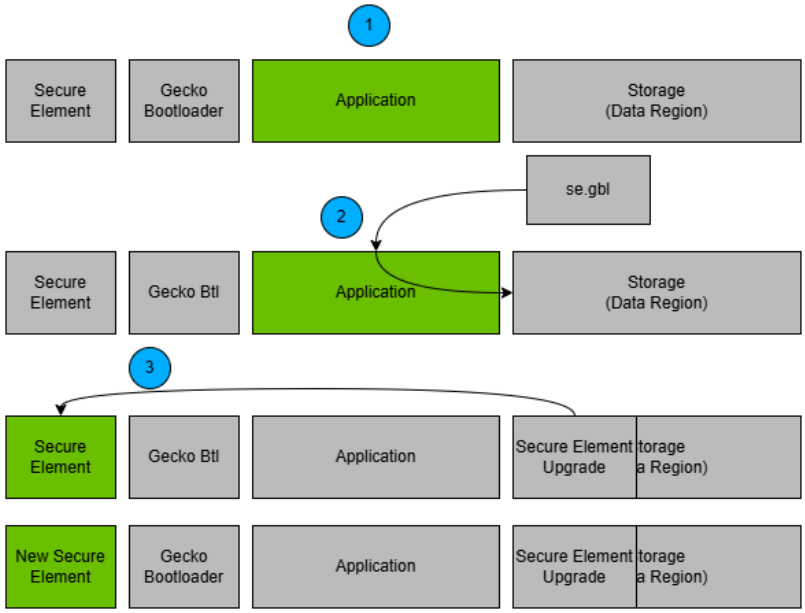
When the bootloader has entered the receive loop, a GBL upgrade file containing a Secure Engine upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns Secure Engine upgrade data in a callback. The bootloader core implements this callback and flashes the data to flash at the pre-configured storage data region.

Copyright © 2025 Silicon Laboratories. All rights reserved.

When a complete Secure Engine upgrade image is received, the bootloader signals the Secure Engine that it should enter firmware upgrade mode. This is done by the Secure Engine communication interface that is used to signal that bootloader upgrade is ready to be performed.

Secure Engine Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



1. A single GBL file containing a Secure Engine upgrade image is downloaded onto the storage medium of the device.
2. The device reboots into the bootloader.
3. The upgrade image will be fetched directly from the GBL file in storage instead of first copying the image to the pre-configured upgrade location.
4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location (or directly from storage, ref. 3b).
6. The device boots into the main bootloader.
7. The bootloader applies the application image from the GBL upgrade file.
8. The device boots into the application. Secure Engine upgrade is complete.

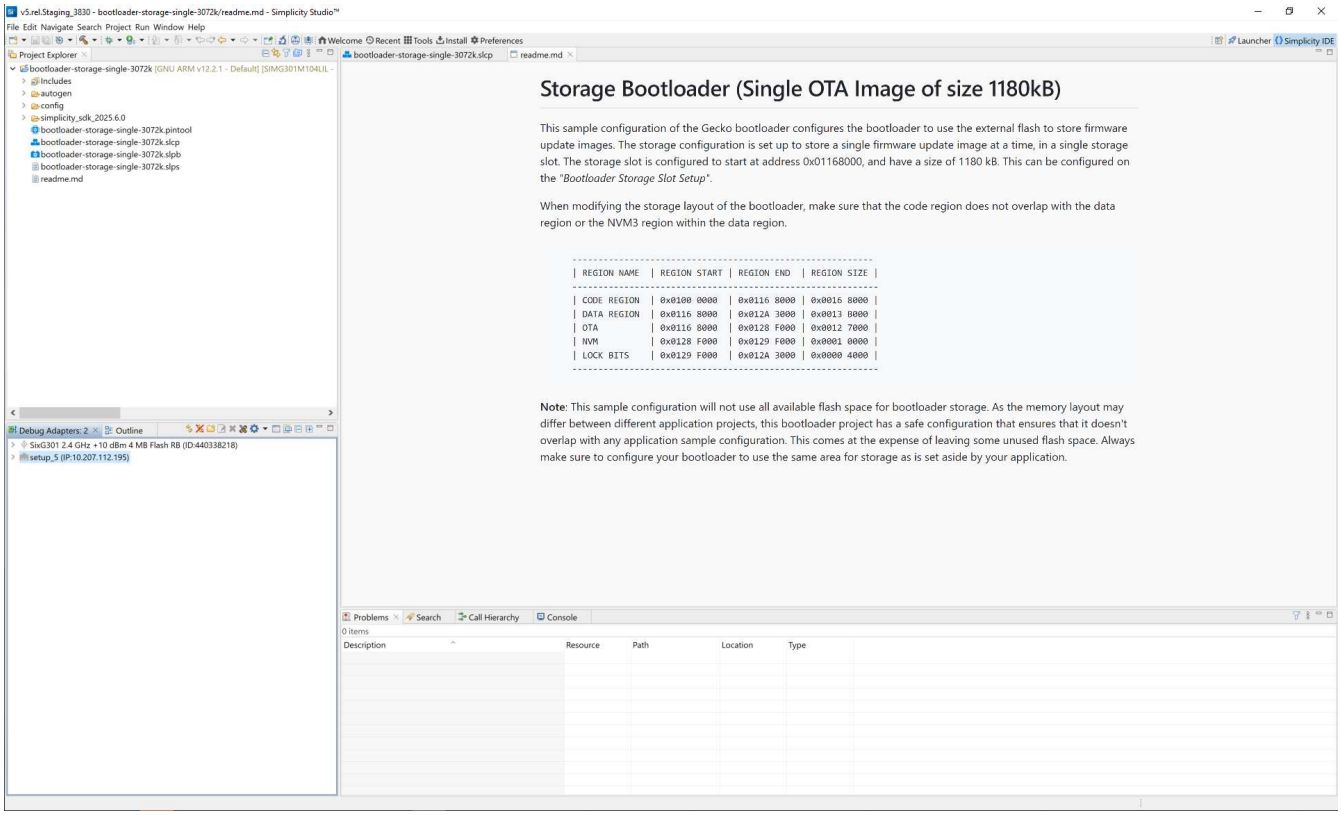
Copyright © 2025 Silicon Laboratories. All rights reserved.

Getting Started with the Gecko Bootloader

Getting Started with the Gecko Bootloader

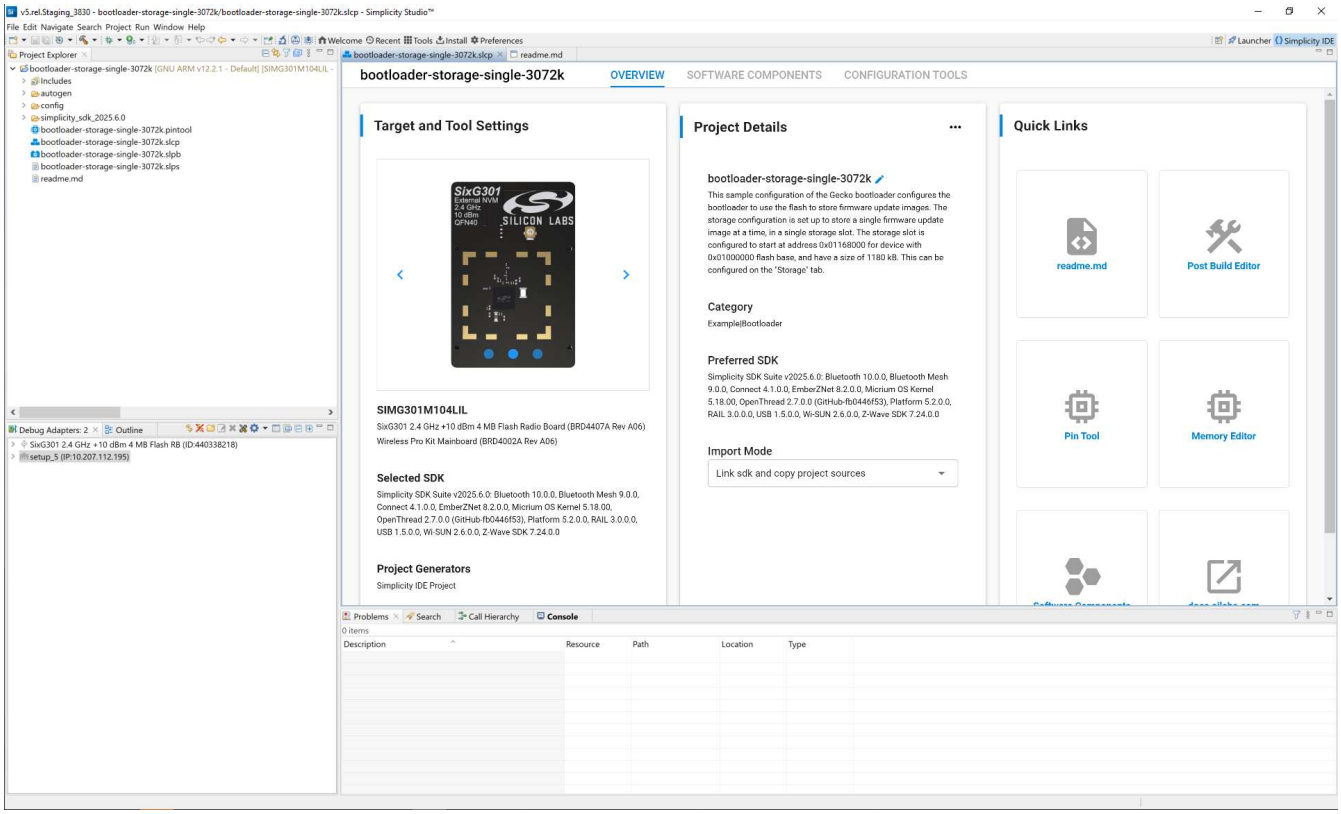
This section describes how to build a Gecko Bootloader from one of the provided examples. These instructions assume that you have installed Simplicity Studio 5, the SiSDK and associated utilities as described in the SDK's quick start guide, and that you are familiar with generating, compiling, and flashing an example application in the relevant version.

1. Create a project based on the Gecko Bootloader example of your choice. The project opens with a tab describing the example.

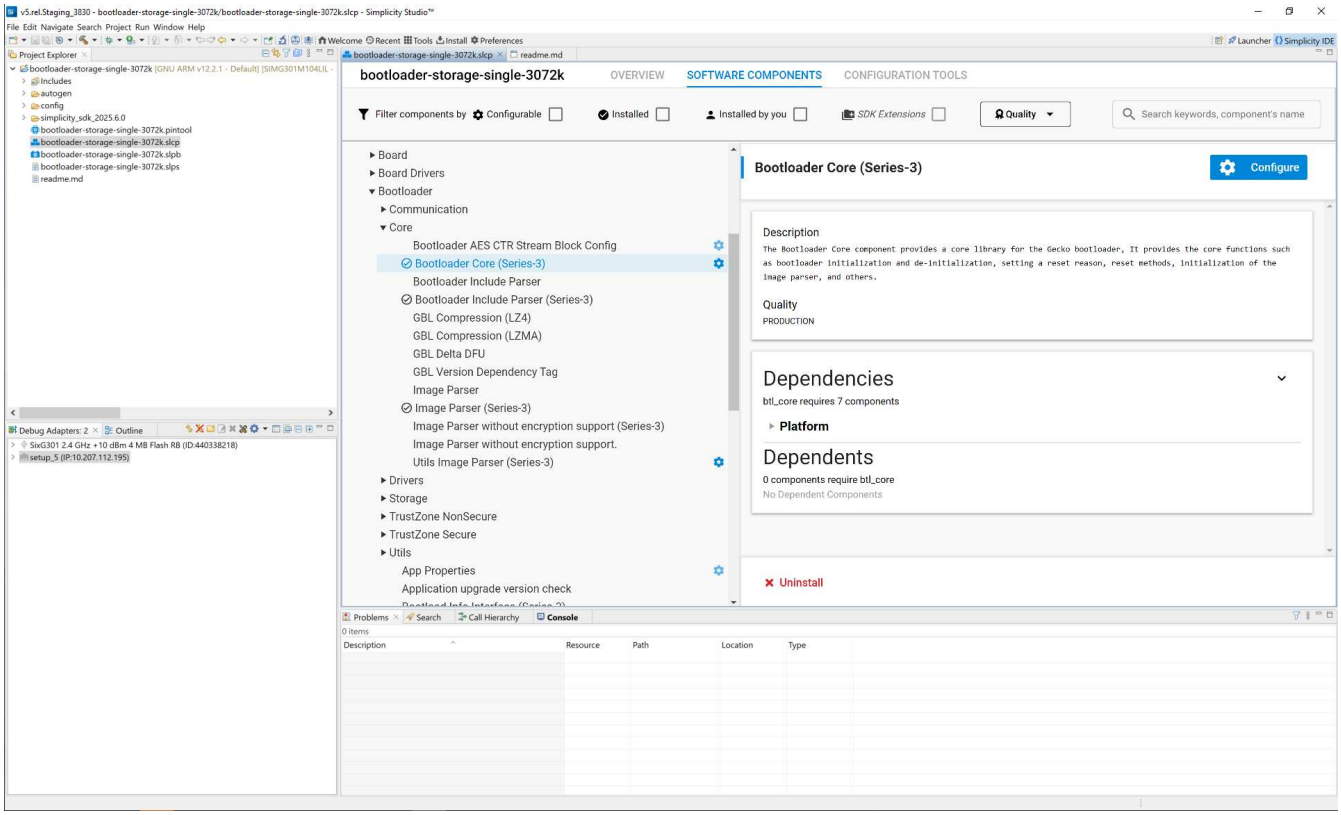


2. Click the project (*.slcp) tab to move to the Project Configurator interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

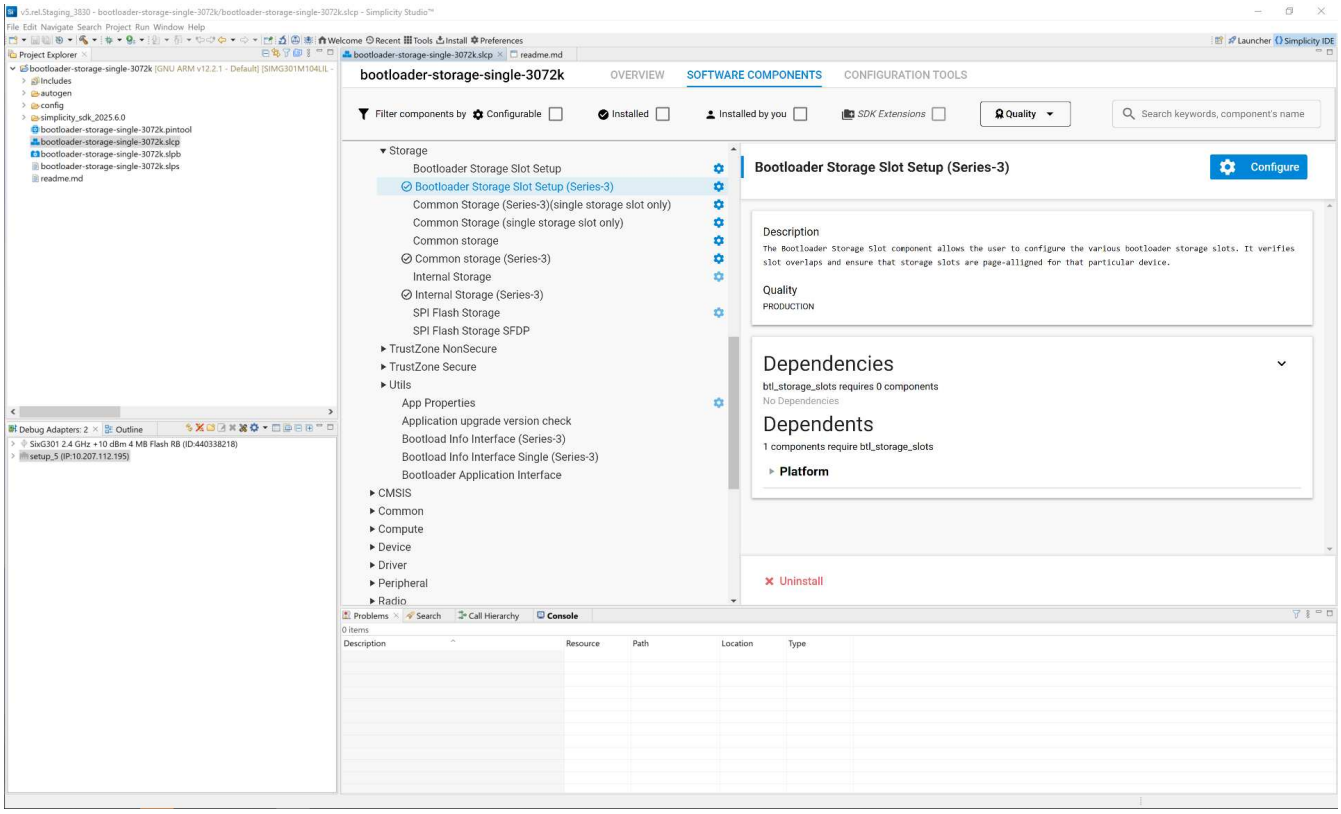


3. The Software Components tab shows the list of available components that can be installed in the project.

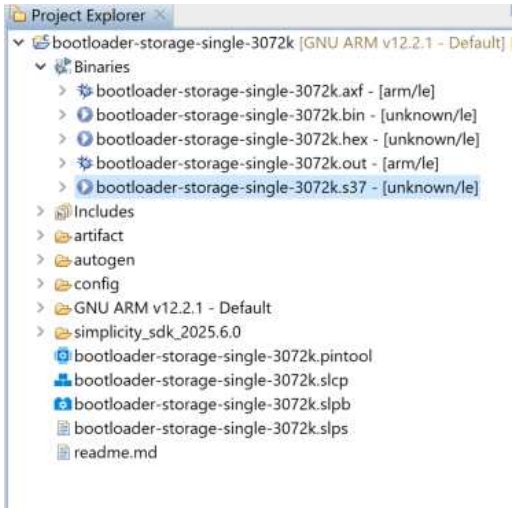


Copyright © 2025 Silicon Laboratories. All rights reserved.

4. The Storage Slot Setup component allows you to configure storage slots to be used if a storage component is also installed. The default configuration matches the target part and bootloader type. This component supports a maximum of three storage slots.



5. Click the Build (hammer) icon.
6. After the build is complete, the bootloader binaries are available in the artifact folder as depicted in the image below.



The image containing only a bootloader must be used to create a GBL file for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuring the Gecko Bootloader

Configuring the Gecko Bootloader

Configuring Storage

Gecko Bootloaders configured as application bootloaders must include an API to store and access image files. This API is based on the concept of storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Slots are configured in the Bootloader Storage Slot Setup component.

When multiple storage slots are configured, a bootload list is used to indicate the order in which the bootloader should access slots to find upgrade images. If multiple storage slots are supported, the application should write the bootload list by calling bootloader_setImageToBootload before rebooting into the bootloader to initiate a firmware upgrade process. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader uses this slot implicitly. A maximum of three slots may be configured in the Bootloader Storage Slot Setup component.

Storage Configuration

When configuring a Gecko Bootloader to obtain images fromflash, modify the following.

The base address of the storage area should be configured in the Common Storage component. This is the address at which the bootloader will place the prioritized list of storage slots to attempt to bootload from, if more than one storage slot is configured. In the default configuration, only a single storage slot is configured, so this value is set to 0, and isn't used. If more than one storage slot is configured, this value needs to be configured too.

The location and size of the storage slots can be configured using the Bootloader Storage Slot Setup Component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, enough space must be reserved between the base address as configured in the Common Storage component and the first storage slot configured in the Bootloader Storage Slot Setup component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. The following figure illustrates how the storage area can be partitioned.

Storage with one storage slot

0	512kB
Application	Storage Slot 0 Size = 512kB

Storage with two storage slot

0	340kB	344kB	348kB	652kB
Application	Bootload list	Bootload list	Storage slot 0 Size = 340kB	Storage slot 1 Size = 340kB

> Note: The storage area partitioning in the example for two storage slots above does not take any NVM system into account. If using an NVM3 , take care to place and size the storage area in such a way that bootloader storage does not overlap with NVM3.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Compressed Upgrade Images

The Gecko Bootloader optionally supports compressed GBL files. In a compressed GBL file, only the application upgrade data and bootloader upgrade data is compressed, any metadata (if present) stays uncompressed. Signature and encryption operations on a compressed GBL work identically to on an uncompressed GBL.

To be able to use compressed upgrade images, a decompressor for the relevant compression algorithm must be added to the Gecko bootloader. The following table shows which compression algorithms are supported by the Gecko Bootloader, and which Bootloader component should be added to enable the feature. The table also shows how much space the decompressor takes up in the bootloader, and how big of a size reduction to expect for the compressed application upgrade image. Be aware of the bootloader size requirement. The bootloader space might be too small to fit the decompressors, depending on the device and enabled components.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZ4	GBL Compression (LZ4)	< 1 kB	~ 10%
LZMA	Bootloader Compression (LZMA)	~5 kB flash, 18 kB RAM	~ 30%

It is important to note that the compressed GBL file stays compressed while being transferred to the device, and while it is stored in the upgrade area. It is decompressed by the bootloader when the upgrade is applied. This means that the running application in main flash will be identical to one that was installed using an uncompressed (normal) GBL file.

Compressed GBL files can only be decompressed by the bootloader when running standalone, not through the Application Interface. This means that upgrade image verification performed by the application prior to reboot will not attempt to decompress the application upgrade, it will only verify the signature of the compressed payload. After rebooting into the bootloader, it will decompress the image as part of the upgrade process.

LZMA Compression Settings

LZMA decompression is only supported for images compressed with certain compression settings. Simplicity Commander automatically uses these settings during gbl4 creation.

- Probability model counters: lp + lc <= 2. Simplicity Commander uses lp=1, lc=1.
- Dictionary size no greater than 8 kB. Simplicity Commander uses 8 kB.

Together, these settings cause the decompressor to require 18 kB of RAM for decompression – 10 kB for the counters and 8 kB for the dictionary.

The Gecko bootloader only supports decompressing payloads that contain the end mark as the last 8 bytes of the compressed stream.

Bootloader Example Configurations

The following sections describe the key configuration options for the example bootloader applications.

> Note: Security features are disabled for all example configurations. In development, Silicon Labs strongly recommends enabling security features to prevent unauthorized parties from uploading untrusted program code. See the *Using Application Image Security Features* section on the *Gecko Bootloader Security Features* page to learn how to configure the security features of the Gecko Bootloader.

UART XMODEM Bootloader

Standalone bootloader for Series 3 devices running the EmberZNet PRO and Silicon Labs Connect protocol stacks, using XMODEM-CRC over UART.

Copyright © 2025 Silicon Laboratories. All rights reserved.

In this configuration, the UART XMODEM communication component, XMODEM Parser component, and Bootloader UART Driver component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the Bootloader UART Driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

BGAPI UART DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using the BGAPI protocol for UART DFU. This bootloader should be used for all NCP-mode Bluetooth applications.

In this configuration, the BGAPI UART DFU communication component and Bootloader UART Driver component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the Bootloader UART Driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation component is also installed by default, allowing bootloader entry by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

Storage Bootloader

Application bootloader for all wireless protocol stacks, using flash to store upgrade images received over the air by the application.

Multiple examples are provided, including configurations for 2 MB, 3 MB, 4 MB & 8 MB flash memory devices . The storage layout should be modified before running the bootloader on any other devices. In this configuration, the flash and common storage components are installed. The base address of the storage area is configured in the Common Storage component. The location and size of the storage slots can be configured using the Bootloader Storage Slot Setup component (provides up to three configurable storage slots). Default example applications are provided with configurations for both single storage slot and multiple storage slots.

The default storage slot configurations provided by the Gecko Bootloader must be configured to match the use-case-specific application configurations, as shown below.

Sample Applications	Storage Offset	Storage Size
Storage Bootloader (Single OTA Image of size 792kB)	0x010E0000	0x000C6000 (811,008 bytes)
Storage Bootloader (Single OTA Image of size 1180kB)	0x0128F000	0x00127000 (1,210,368 bytes)
Storage Bootloader (Single OTA Image of size 1540kB)	0x01379000	0x00181000 (1,577,984 bytes)

Image Acquisition Application Example Configuration

These examples illustrate applications that acquire and store a GBL upload image for an application bootloader. For the running bootloader to accept an application upgrade, the new application version must be higher than the existing version.

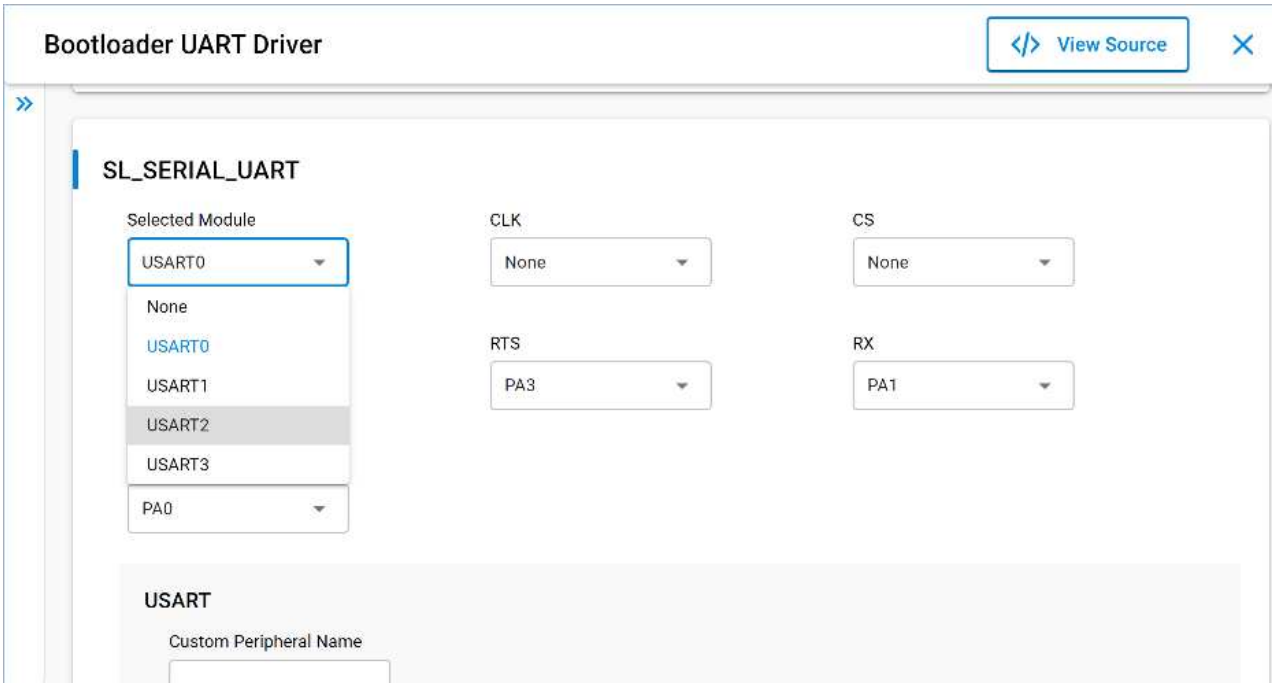
Setting a Version Number

To distinguish between different builds of the Gecko Bootloader, it is useful to set a version number. To perform a bootloader upgrade, not only must the running bootloader pass its integrity checks (see the *Downloading and Applying a Bootloader GBL Upgrade File* section on the *Gecko Bootloader Operation Bootloader Upgrade* page), but the bootloader upgrade image must also have a higher version number than the running bootloader image. A version number can be set using Simplicity Studio by configuring the Bootloader Version Main Customer option of the Bootloader Core component. This macro will be picked up by the config file btl_config.h, where it is combined with the version number of the Gecko Bootloader files provided by Silicon Labs.

Hardware Configuration

The Gecko Bootloader uses the Pin Tool for configuration of pinout and other hardware-related settings. When Pin Tool configuration is available for a bootloader component, the relevant settings are shown in the Component Editor for that component.

Copyright © 2025 Silicon Laboratories. All rights reserved.



The standalone Pin Tool User Interface can also be used to configure settings for Gecko Bootloader if desired.

While the Pin Tool provides configuration for many different peripherals, the Gecko Bootloader uses only the following Pin Tool modules:

- SERIAL is used by the UART Driver component to configure baud rate, flow control mode and pinout.
- VCOM is used by the UART Driver component to enable the serial interface if necessary (only on Silicon Labs Wireless Starter Kits).
- BTL_BUTTON is used by the GPIO Activation component.
- CMU HFXO frequency setting is used by the delay driver to calibrate timing if the core is running from the HFXO.

Other settings, like CMU oscillator configuration or DCDC configuration, are not taken into consideration by the default bootloader code. If using these configuration settings is desired, the required code must be added in btl_main.c.

> Note: While the delay driver uses the HFXO frequency setting from Pin Tool, the HFXO enable setting is not used to initialize the HFXO on startup. This setting is only used when calling the bootloader through the Application Interface, and the application has switched to the HFXO prior to calling the Bootloader Application Interface API.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Simplicity Commander and the Gecko Bootloader

Simplicity Commander and the Gecko Bootloader

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface) that is also scriptable. You can use Simplicity Commander to perform these essential tasks:

- Generating key files for signing and encryption
- Signing application images for Secure Boot
- Creating GBL images (encrypted or unencrypted, signed, or unsigned)
- Parsing GBL images

Simplicity Commander is used throughout the examples in the following sections. For more information on executing commands to complete these tasks, see the Simplicity Commander User Guide.

> Note: Simplicity Commander also offers a GUI (Graphical User Interface) that can be used in the lab for typical tasks such as flashing device images. The functions described in this User Guide are performed from the CLI.

Creating GBL V4 Files Using Simplicity Commander

To create an unsigned GBL V4 file from an application myapp.s37, execute `commander gbl4 create myapp.gbl --config configfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:“00000000000000000000000000000000”
updates:
  -data:myapp.s37
   block_size: 0
```

To create an unsigned GBL V4 file from a bootloader upgrade mybootloader.s37, execute `commander gbl4 create mybootloader.gbl --config configfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:“00000000000000000000000000000000”
updates:
  -data:mybootloader.s37
   block_size: 0
```

This file can be used with the standalone bootloader configurations of the Gecko Bootloader.

To create an unsigned GBL file from a Secure Engine, upgrade mySecureElement.seuv2, and execute `commander gbl4 create mySecureElement.gbl –config configfile.yaml`.

Example of a config file is as follows:

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
manifest:
 product_id:“00000000000000000000000000000000”
se_upgrade:
  se_file:mysecureElement.seuv2
```

The Secure Engine images, .seuv2, are provided by Silicon Labs and can be found through Simplicity Studio. See the *Gecko Bootloader Operation - Secure Engine Upgrade* section on the *Gecko Bootloader Operation Secure Engine Upgrade* page.

The command can also be used to create a single upgrade image, suitable for use with application bootloader configurations of the Gecko Bootloader: `commander gbl4 create myupgrade.gbl –config configfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:“00000000000000000000000000000000”
updates:
  -data:myapp.s37
  -data:mybootloader.s37

se_upgrade:
  se_file: mysecureElement.seuv2
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Security Features

Gecko Bootloader Security Features

About Bootloader Image Security

Secure Boot and Secure Firmware Upgrade, discussed in the following sections, enables Gecko Bootloader to provide authenticity and integrity checks on the Application image, which provides a sufficient level of security for many applications. However, in systems without a hardware root of trust, no process checks the authenticity or integrity of the Gecko Bootloader itself. Its security is provided solely by the device hardware and the robustness of the software running on the device.

The native behavior of Firmware Upgrade will prevent accidental version rollback of Gecko Bootloader under normal usage conditions.

About Application Image Security

The Gecko Bootloader can enforce security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted.

Secure Boot Procedure

When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA (elliptical curve digital signature algorithm) of the SHA-256 digest of the application firmware image, using the NIST P-256 (secp256r1) curve.

Summary of Operation

1. On boot, the bootloader checks the application image for information about whether it is signed.
2. The type of signature and signature location is determined.
3. If the type of signature does not match the requirements of the bootloader, the bootloader enters device firmware upgrade mode and prevents the application from running.
4. According to the chosen signature algorithm, the signature of the contents of flash from the beginning of the application to the location of the signature is compared to the signature at the signature location.
5. If the signatures do not match, the bootloader enters device firmware upgrade mode and prevents the application from running.

Secure Boot using ECDSA-P256-SHA256

For an image to be signed for Secure Boot, the application needs to contain a copy of the ApplicationProperties_t struct. This struct contains information about which signature algorithm is used, and where to find the signature.

On every boot, the bootloader calculates the SHA-256 digest of the application image, from the beginning of the application to the start of the signature. The signature of the SHA-256 digest is then verified using ECDSA-P256.

If the signature is valid, the application is allowed to boot. Else, the bootloader is entered, and an application upgrade is attempted if one is available.

Simplicity Commander can be used to generate a key pair and write the public key to the device. See the Simplicity Commander User Guide for more information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Boot with Application Rollback Protection

On every boot, the application version in the ApplicationData_t struct is stored at the end of the bootloader area in flash, which is used to prevent applications from being downgraded. The application version can remain the same for upgrades.

If application rollback prevention component is installed in the bootloader project, then before applying the upgrade image, the bootloader will check the application version in the application properties structure that resides inside the signed/encrypted GBL file and will only apply the OTA image to application area if the application version in ApplicationData structure is equal or higher than the highest application version last seen.

The application rollback prevention feature can be enabled in the Bootloader Core component by selecting the Enable application rollback protection option. The Minimum application version allowed option can be used to configure the minimum application version that should be allowed to boot.

The application versions are stored in the SE OTP area.

Secure Boot Using a Certificate

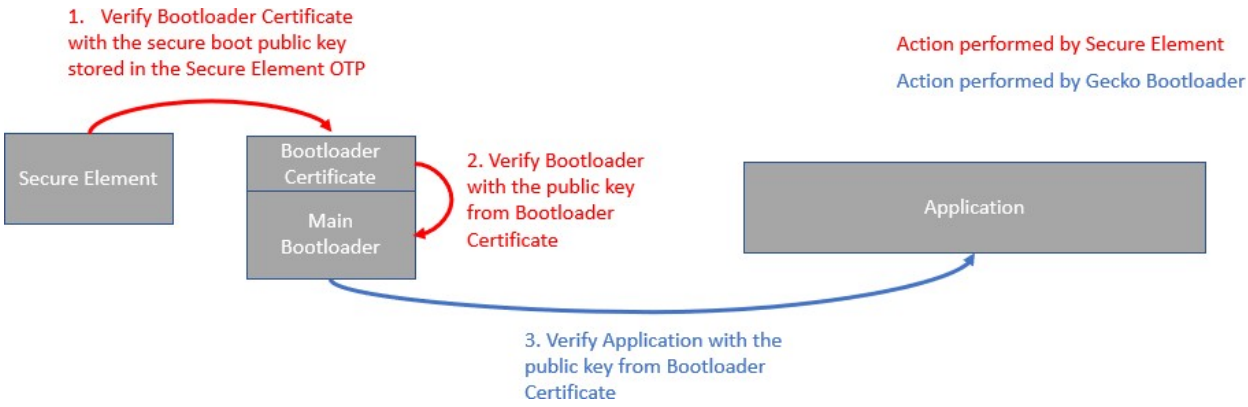
On Series 3 devices, a certificate-based secure boot operation is supported. The Certificate contains:

- Struct version: The version of the certificate structure.
- Public key: ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
- Certificate version: The version of the running certificate.
- Signature: ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

The definition of the certificate struct can be found in `api/application_properties.h`.

To utilize certificate-based secure boot, configure Secure Engine to authenticate the bootloader image by configuring the certificate- based secure boot option in the Secure Engine OTP. Configure the Gecko Bootloader to enable certificate-based secure boot in the Bootloader Core component by selecting the Enable certificate support option. The Gecko Bootloader certificate must be signed by the private key pair of the public key stored in the Secure Engine OTP. For more information on the key storage, see the *Key Storage* section.

The certificate-based secure boot procedure is illustrated in the following figure.

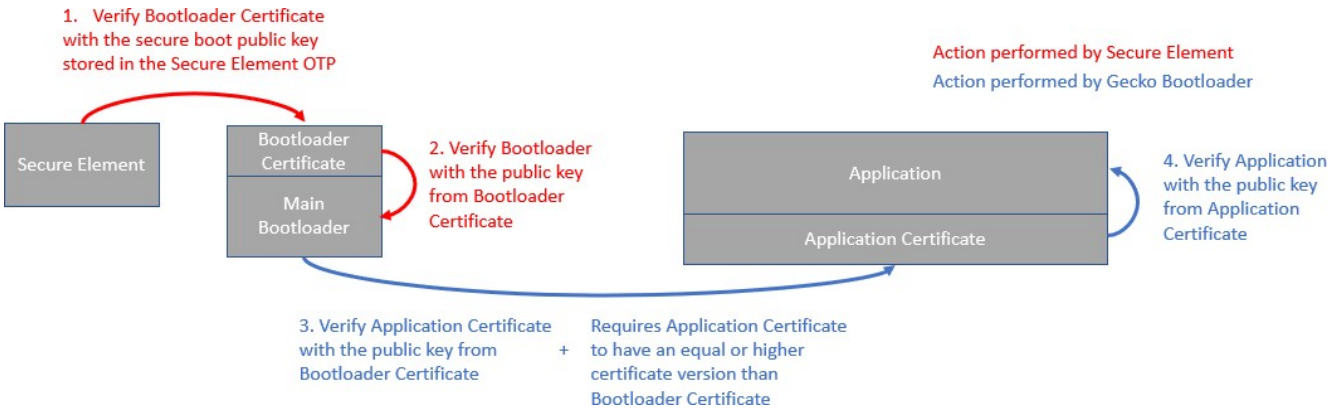


Once the certificate-based secure boot option on Secure Engine is turned on, Secure Engine verifies the Gecko Bootloader certificate. The public key stored in the certificate is used to validate the signature of the Gecko Bootloader. Secure Engine will not accept bootloader images without a certificate.

If only the secure boot option is enabled (not certificate-based) on Secure Engine, and Secure Engine identifies a certificate, the certificate will be used to validate the bootloader image. If the certificate version from the bootloader image is higher than 0 and it gets verified once, Secure Engine will never again accept direct signed bootloader images without a certificate.

The Gecko Bootloader will authenticate the direct signed application using the public key stored in the Gecko Bootloader certificate. If the application contains a certificate, Gecko Bootloader will authenticate it. The procedure is illustrated in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.



After authentication of the application certificate, Gecko Bootloader verifies the signature of the application using the public key from the application certificate. In addition, Gecko Bootloader compares the Gecko Bootloader certificate version against the application certificate version. All application images with certificate version lower than the certificate version of the Gecko Bootloader will be rejected. Gecko Bootloader can be configured to only allow applications with certificates to boot by configuring the Bootloader Core component by selecting the Reject direct signed images option.

The ApplicationProperties_t struct contains the certificate struct ApplicationCertificate_t. The certificate struct can be injected to images that contain an ApplicationProperties_t with ApplicationCertificate_t. To inject a certificate to an image, issue the following command from Simplicity Commander:

```
commander convert <image file> --secureboot --keyfile <signing key> --certificate <certificate>
--outfile <signed image file with certificate>
```

Secure Firmware Upgrade

The Gecko Bootloader supports a secure firmware upgrade process. This is achieved by using symmetric encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image. Symmetric encryption provides confidentiality, and asymmetric cryptography provides integrity and authenticity. Note that encryption alone is not enough to provide authenticity.

Encryption Algorithms

The Gecko Bootloader supports the AES-CTR-128 encryption algorithm. The GBL upgrade file is encrypted using 128-bit AES in Counter mode with a random nonce as the initial counter value.

To make use of the OTA decryption key stored in the Secure Engine OTP, the Use symmetric key stored in Secure Engine storage option in the Bootloader Core component must be selected. Simplicity Commander can be used to generate an OTA decryption key and write the key to the device. For more information on storing the OTA decryption key on Series 3 devices, see Simplicity Commander User Guide.

The Secure Engine OTP key support depends on the SE Manager component, which is enabled by default.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA signature of the SHA-256 digest of the GBL upgrade file, using the NIST P-256 (secp256r1) curve.

Summary of Operation

Before starting a firmware upgrade process, the application can verify an image in storage by calling into the bootloader verification functions. For bootloaders with Communication Interface, the host device should verify the image before sending it to the NCP or RCP.

During firmware upgrade, the GBL file is parsed, and if encrypted, decrypted on-the-fly. A GBL Signature Tag in the GBL file indicates to the bootloader that the file is signed, and the signature is verified. If signature verification fails, the firmware upgrade process is aborted.

Copyright © 2025 Silicon Laboratories. All rights reserved.

On Series 3 devices, Gecko Bootloader will authenticate the GBL signature tag using the public key stored in the bootloader certificate if the Enable certificate support option is selected in the Bootloader Core component. A GBL Certificate tag in the GBL file indicates to the bootloader that the GBL certificate tag needs to be authenticated using the public key stored in the bootloader certificate. The certificate version in the GBL certificate tag is compared with the bootloader certificate and only a version equal or higher than the bootloader certificate is accepted. Once the GBL certificate tag is authenticated, the GBL file's signature is verified using the authenticated public key from the GBL certificate tag.

Using Application Image Security Features

This example assumes that a bootloader called bootloader-uart-xmodem has been built in Simplicity Studio.The relevant version can be flashed to the device using the Flash Programmer in Simplicity Studio or using Simplicity Commander.

This example provides two ways of signing the upgrade images. The first option uses Simplicity Commander to generate key material and sign data. This is suitable for development. The second option uses an external signer, such as a dedicated Hardware Security Module (HSM) to protect private key material and perform signing operations. Silicon Labs recommends using an HSM to safeguard private keys.

Generating Keys

To use the security features of the Gecko Bootloader, encryption and signing keys need to be generated. These keys must then be written to the device. The encryption key is used with the GBL file for secure firmware upgrade. The signing keys are used both with the GBL file for secure firmware upgrade and to sign the application image for Secure Boot.

Generating a Signing Key Using Simplicity Commander

```
commander util genkey --type ecc-p256 --privkey signing-key --pubkey signing-key.pub
```

This creates an ECDSA-P256 key pair for signing; `signing-key` contains the private key in PEM format and must be kept secret from third parties. This key will later be used to sign images and GBL files. `signing-key.pub` contains the public key in PEM format and can be used to verify GBL files using commander gbl4 info.

Generating a Signing Key Using a Hardware Security Module

When using a Hardware Security Module, the private key is kept secret inside the HSM. According to the instructions from your HSM vendor, have it generate an ECDSA-P256 key pair and export the public key in PEM format to the file signing-key.pub. Then use Simplicity Commander to convert the key to token format, suitable for writing to the Series 3 device.

```
commander gbl keyconvert --type ecc-p256 signing-key.pub
```

Generating an Encryption Key

```
commander util genkey --type aes-ccm --outfile encryption-key
```

This creates an AES-128 key for encryption in the file encryption-key.

Writing Keys to the Device

> Note: Refer to the sections, *Writing the AES Decryption Key* and *Writing the Public Key to the Device* in the Simplicity Commander User Guide.

Signing an Application Image for Secure Boot

Copyright © 2025 Silicon Laboratories. All rights reserved.

If the bootloader enforces Secure Boot, the application needs to be signed to pass verification. On every boot, a SHA-256 digest of the application is calculated. The signature is verified using ECDSA-P256, with the same public key as for the GBL file signing. Signature verification failure prevents the application from booting.

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. If ApplicationProperties_t is missing, the application image cannot be signed. For more details on adding ApplicationProperties_t, see the *Application Properties* section on the *Application Interface* page.

Using Simplicity Commander

Signing the application can be done with the command:

```
commander convert myapp.s37 --secureboot --keyfile signing-key --outfile myapp-signed.s37
```

Using a Hardware Security Module

The application can be prepared for signing by issuing the command:

```
commander convert myapp.s37 --secureboot --extsign --outfile myapp-for-signing.s37
```

Using an HSM, sign the output file myapp-for-signing.s37, and supply the resulting DER-formatted signature file signature.der back to Simplicity Commander:

```
commander convert myapp-for-signing.s37 --secureboot --signature signature.der --verify signing-key.pub

--outfile myapp-signed.s37
```

Creating a Signed and Encrypted GBL Upgrade Image File from an Application

To create a GBL file from an application, use `commander gbl4 create`.

Note that, as of this writing, secure application images can only be constructed through Simplicity Commander, not through the configuration options available through Simplicity Studio.

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. If ApplicationProperties_t is missing, the application image cannot be signed. For more details on adding ApplicationProperties_t, see the *Application Properties* section on the *Application Interface* page.

Using Simplicity Commander to Sign

For an application called myapp.s37, use:

```
commander gbl4 create myapp.gbl -–config configfile.yaml
```

Example of a config file is as follows:

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
manifest:
 product_id: "[16 B ID] (default from app or 00000000000000000000000000000000)"
 bundle_version: "[version] (default 0x00000000)"
 min_version: "[version] (default 0x00000000)"
 hash_function: "[none | sha256] (default sha256)"
 certificate: "[path to file] (optional)"
 signing_key: "[path to file] (optional)"
se_upgrade:
 se_file: "[path to file] (optional)"
updates:
 - data: <path to file>
  app_type: "[uint32] (default from app or 0x00000000)"
  app_capabilities: "[uint32] (default from app or 0x00000000)"
  version: "[uint32] (default from app or 0x00000000)"
  compression_scheme: "[none | lz4 | lzma] (default none)"
  encryption_scheme: "[none | aes-ctr-128] (default none)"
  block_size: "[size in bytes] (default 0/disabled)"
```

This single command performs three actions:

- Creates a GBL file
- Encrypts the GBL file
- Signs the GBL file

If Secure Boot is also desired, the application must be signed using `commander convert --secureboot` prior to creating the GBL.

System Security Considerations

The Gecko bootloader security features can be used to create a secure device, but do not create a secure system by themselves. This section goes over considerations that need to be taken when designing a secure system where the Gecko Bootloader is a component.

Key Storage

On Series 3 devices, the decryption key and the sign key used by the Gecko Bootloader in the Secure Engine OTP. The decryption key can be provisioned in the Secure Engine OTP using Simplicity Commander or using the Secure Engine Mailbox interface. Once a key value has been programmed into the Secure Engine OTP, it cannot be changed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Interface

Application Interface

The bootloader has an application interface exposed through a function table in the bootloader. The application interface provides APIs to use bootloader functions for storing and retrieving upgrade images and verifying their integrity. APIs to reboot into the bootloader are also provided. For details see the Gecko Bootloader API Reference, at https://docs.silabs.com/gecko-platform/5.1.2/platform-bootloading-overview/folder.

If you are not using a protocol stack from Silicon Labs, the api/btl_interface.h header provides the bootloader application interface API. If you are using a protocol stack from Silicon Labs, the recommended bootloader interface API for the specific protocol stack should be used instead. The following files provide the implementation of the bootloader interface:

api/btl_interface.c (common interface)

api/btl_interface_storage.c (interface to storage functionality)

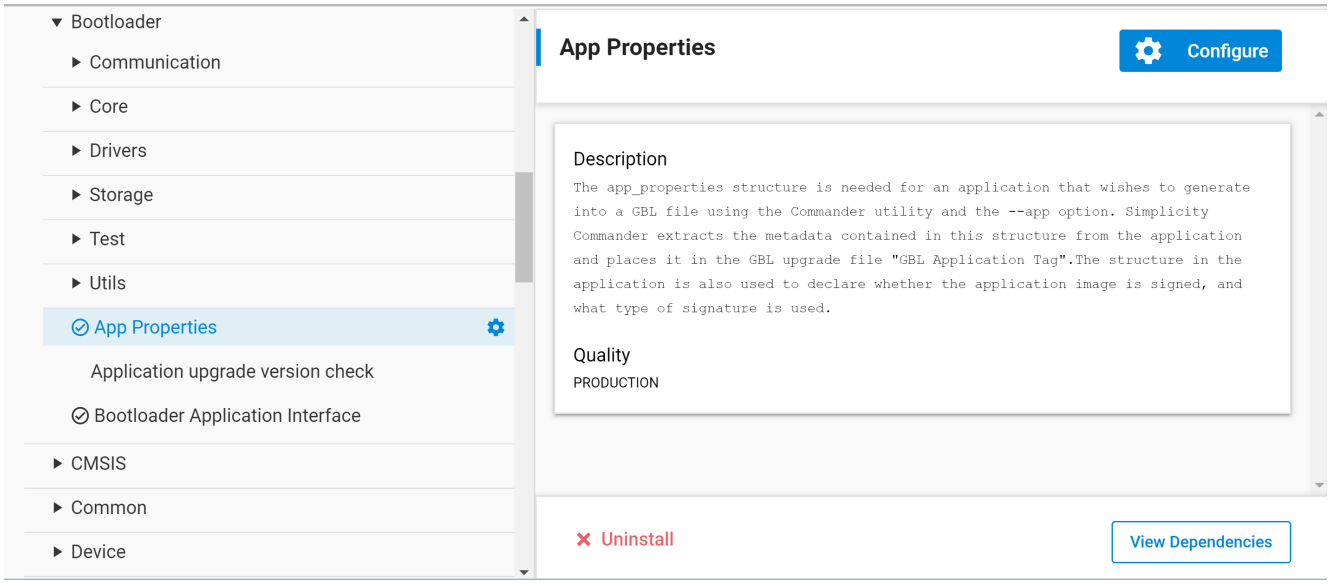
The application interface consists of functions that can be included in the customer application, and that communicate with the bootloader through the MainBootloaderTable_t. This table contains function pointers into the bootloader. The 10th word of the bootloader contains a pointer to this structure, allowing any application to easily locate it. Using the wrapper functions provided in the Bootloader Interface API is preferred over accessing the bootloader table directly. Modules include:

- Application Storage Interface: Application interface for interfacing with bootloader storage. The Storage Interface is only available on bootloaders that support the storage interface.
- Common Application Interface: Generic application interface available on all versions of the bootloader, independently of which components are present.

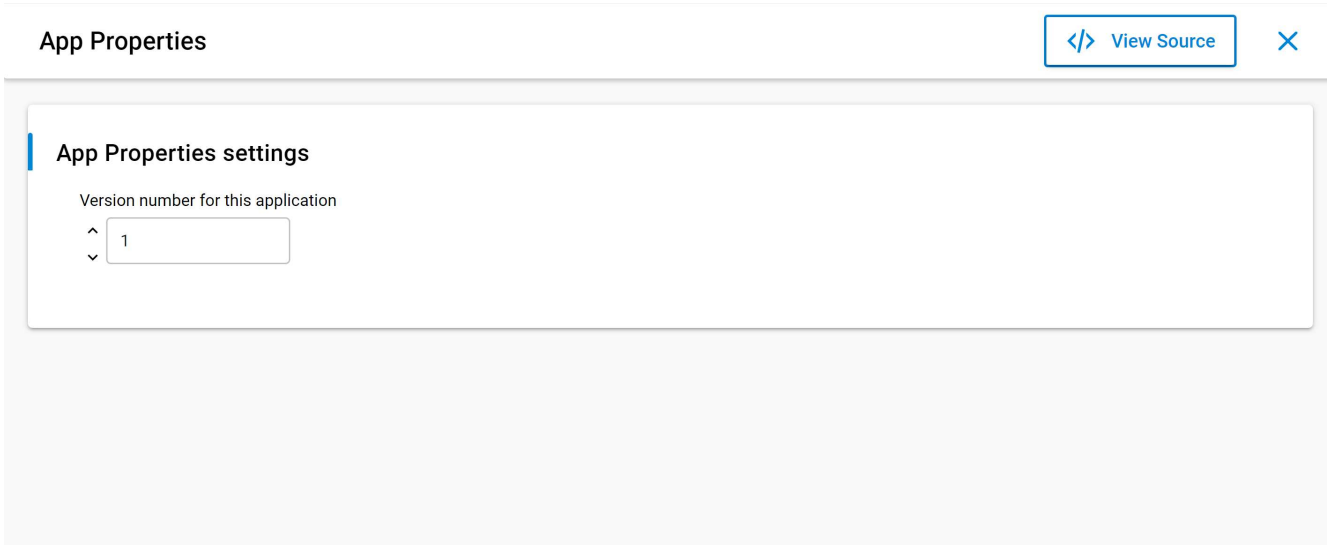
Application Properties

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. The Simplicity Commander extracts the metadata contained in this structure from the application and places it in the UPDATE_MEMORY_SECTION tag in the GBL upgrade file. If the structure is not present in the application, Simplicity Commander will raise an error. The ApplicationProperties_t struct is added to the application on installing the bootloader_interface component to the application. The bootloader_interface component installs bootloader_app_properties component which adds an instance of ApplicationProperties_t named sl_app_properties to the project. The component adds a source file named app_properties.c and a configuration file named app_properties_config.h. This configuration file allows users to configure the application version via Simplicity Studio's Component Editor. To open the Component Editor, locate the App Properties component under Platform > Bootloader as shown below.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Click Configure to open the Component Editor.



The application type is automatically populated based on the wireless stack used in building the project. The value of the application type is automatically set during the project generation step and can be found in autogen/sl_application_type.h file.

The contents of the GBL Application Tag can be extracted from a GBL file by a running application using the Application Storage interface. Note that the GBL Application Tag will only be added if the GBL file contains an application image, not if the GBL file only contains a bootloader upgrade or metadata.

The structure in the application is also used to declare whether the application image is signed, and what type of signature is used. This information is added by Simplicity Commander when signing the image using `commander convert (--secureboot, --extsign or -- signature)`. For the bootloader to locate the ApplicationProperties_t struct, if not already done by the linker, Simplicity Commander modifies word 13 of the application to insert a pointer to the ApplicationProperties_t struct when signing the application image for Secure Boot.

Error Codes

Most Gecko bootloader APIs return error codes. The following table lists the groups of error codes that may be returned. The full list of error codes within each group can be found in *api/btl_errorcode.h* in the platform/bootloader directory of the SDK, as well as

Copyright © 2025 Silicon Laboratories. All rights reserved.

in the API Reference.

ID	Description
0x0	OK
0x01xx	Initialization error
0x02xx	Image verification error
0x04xx	Storage error
0x05xx	Bootload error
0x06xx	Security error
0x07xx	Communication error
0x09xx	XMODEM parser error
0x10xx	GBL file parser error
0x11xx	SPI slave driver error
0x12xx	UART driver error
0x13xx	Compression error
0x14xx	RTOS error

Copyright © 2025 Silicon Laboratories. All rights reserved.

Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher

Silicon Labs Gecko Bootloader User’s Guide for GSDK 4.0 and Higher

Note: This section replaces *UG489: Silicon Labs Gecko Bootloader User’s Guide for GSDK 4.0 and Higher*. Further updates to this user guide will be provided here.

This document describes the high-level implementation of the Silicon Labs Gecko Bootloader for EFM32 and EFR32 Series 1 and Series 2 microcontrollers, SoCs (System on Chips) and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to *UG103.6: Bootloader Fundamentals*. For more information on using the Gecko Bootloader with different wireless stacks, see the following:

- Using the Gecko Bootloader with EmberZNet
- Bootloading and OTA with Silicon Labs Connect v3.x
- Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

For more information on Series 2 device security, see:

- Series 2 and Series 3 Secure Boot with RTSL
- Series 2 Secure Debug
- Production Programming of Series 2 and Series 3 Devices

For more information on security using Series 2 devices with Secure Vault, see:

- Anti-Tamper Protection Configuration and Use
- Authenticating Silicon Labs Devices using Device Certificates
- Secure Key Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Useable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
  - Secure Boot: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
  - Signed upgrade image file: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
  - Encrypted upgrade image file: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader file). These files have the file extension ".gbl". See section Gecko Bootloader File Format for more details.

On Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

On Series 2 devices, the Gecko bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the Secure Engine. The Secure Engine may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- HSE - Hardware Secure Engine, either with or without Secure Vault if not specified
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE, in general)

The Secure Engine provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. This makes it possible to have a 2-stage design, where the main bootloader is not present. However, the presence of a main bootloader is assumed throughout this document.

To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Engine installs it. On some devices, the Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism. See Gecko Bootloader Operation - Secure Engine Upgrade for more details.

The main bootloader consists of a common core, drivers, and a set of components that give the bootloader specific capabilities. The common bootloader core is now provided as a full-source delivery, as opposed to previously being a combination of compiled libraries and source code while the components continue to be delivered as source code. The common bootloader core contains functionality to parse GBL files and flash their contents to the device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be installed in and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image over-the-air using Bluetooth or by a physical connection such as UART or SPI. To function as a standalone bootloader with a physical connection, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by writing the firmware upgrade image to a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, Ethernet, USB, and so on). The download space is either an external memory device such as an EEPROM or dataflash or a section of the device's internal flash. The Gecko Bootloader can partition the download space into multiple storage slots and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation has to be configured.

Silicon Labs provides example bootloaders that come with a preconfigured set of installed components for configuration in either standalone or application mode. See section Configuring the Gecko Bootloader. The Silicon Labs Gecko SDK Suite also includes precompiled bootloader images for several different EFR32 devices. As of this writing, the images shown in the following table are provided.

> Note: The bootloader security features are not enabled in these precompiled images.

Table Prebuilt Bootloader Images

Use	Wireless Stack	Image Name	Mode	Interface
SoC	EmberZNet PRO	SPI Flash Storage Bootloader	Application	SPI Serial Flash
SoC	Bluetooth	Bluetooth In-Place OTA DFU Bootloader	Application	OTA/internal flash
NCP	EmberZNet PRO	UART XMODEM Bootloader	Standalone	UART (EZSP)
NCP	Bluetooth	BGAPI UART DFU Bootloader	Standalone	UART (BGAPI)

Note that on devices with a dedicated bootloader area (EFR32xG12 and later Series 1 devices), if the device is configured to boot to the bootloader area (that is, if bit 1 of the Config Lock Word 0 CLW0[1] is set), an image always must be present in the bootloader area. The device is factory-programmed with a dummy bootloader that simply jumps directly to the application in main flash. This means that when flashing a bootloader to a device with a dedicated bootloader area, this dummy bootloader is replaced. If later during development using the bootloader is no longer desired, CLW0[1] must be cleared or the dummy bootloader needs to be re-flashed. Platform-specific prebuilt dummy bootloader images are located in *./platform/bootloader/util/bin/*. Note that since the dummy bootloader only consists of a few instructions and doesn't pad out the remainder of the bootloader area, only the first flash page (where the first-stage bootloader resides) is overwritten, so the main stage bootloader would likely remain intact after programming the dummy bootloader. If desired, the rest of the flash pages in the bootloader area can then be erased separately.

On devices that do not have a dedicated bootloader area (EFR32xG1 and EFR32 Series 2), a dummy bootloader is not needed.

The following sections provide an overview of the Gecko Bootloader common core, drivers, and components. For details, including details on error codes and conditions, see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

The bootloader area can be fully erased using the `commander device pageerase --region @bootloader` command with Simplicity Commander. In this state, the device will not boot until CLW0[1] is cleared or the dummy bootloader is flashed. For more information on how to use Simplicity Commander with Gecko bootloader, see section Simplicity Commander and the Gecko Bootloader.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Core

The bootloader core contains the bootloader's main functions. It also contains functionality to write to the internal flash, an image parser to parse and act upon the contents of GBL upgrade files, and functionality to boot the application in main flash.

The image parser can also optionally support the legacy Ember Bootloader (EBL) file format, but none of the security features offered by the Gecko Bootloader are supported if support for legacy EBL files is enabled.

A version of the GBL image parser without support for encrypted upgrade images is also available. This version can be used in flash space constrained bootloader applications where encryption of the upgrade image is not required.

Shared Memory

To exchange information between the bootloader and application, a section of SRAM is used. The contents of SRAM are preserved through a software reset, making the SRAM suitable as a communication channel between bootloader and application.

The shared memory has a size of 4 bytes, and is located at the first address of SRAM, 0x20000000. It is used to store a single word containing the reason for a reset. The structure of the reset cause word is defined in the Reset Information part of the Application Interface, in the file btl_reset_info.h, as 16 bits containing the reason, and 16 bits of signature indicating if the word is valid or not. If the signature reads 0xF00F, the reset reason is valid.

All 16-bit reset reasons used by Silicon Labs have the most significant bit set to zero. If custom reset reasons are desired, it is recommended to set the most significant bit to avoid conflicting definitions.

In addition to the reset causes defined in the Reset Information documentation, the bootloader will enter firmware upgrade mode if the shared memory contains the value 0x00000001. This value is supported for compatibility with certain legacy Bluetooth applications.

Drivers

Different applications of firmware upgrade require different hardware drivers for use by the other components of the bootloader.

Driver modules include:

- Delay: Simple delay routines for use with components that require small delays or timeouts.
- SPI: Simple, blocking SPI master implementation for communication with external devices such as SPI flashes.
- SPI Slave: Flexible SPI Slave driver implementation for use in communication components implementing SPI protocols. This driver supports both blocking and non-blocking operation, with DMA (Direct Memory Access) backing the background transfers to support non- blocking operation.
- UART: Flexible serial UART driver implementation for use in communication components implementing UART protocols. This driver supports both blocking and non-blocking operation, with DMA backing the background transfers to support non-blocking operation. Additionally, support for hardware flow control (RTS/CTS) is included.

Components

All parts of the bootloader that are either optional or that may be exchanged for different configurations are implemented as components. Each component may have a configuration header file, and one or more implementations. Components include:

- Communication
  - UART: XMODEM
  - UART: BGAPI
  - SPI: EZSP
  - Bluetooth: AppLoader
- Compression
- Debug
- GPIO Activation
- Security
- Storage
  - Internal flash
  - External SPI flash

Copyright © 2025 Silicon Laboratories. All rights reserved.

Communication

The Communication components provide an interface for implementing communication with a host device, such as a computer or a micro- controller. Several components implement the communication interface, using different transports and protocols.

- BGAPI UART DFU: By enabling the BGAPI communication component, the bootloader communication interface implements the UART DFU protocol using BGAPI commands. See *AN1053: Bluetooth® Device Firmware Update over UART for EFR32xG1 and BGM11x Series Products* for more information about this legacy bootloader.
- Bluetooth:AppLoader: By enabling the Bluetooth AppLoader communication component, the bootloader communication interface implements over-the-air device firmware upgrade functionality using Bluetooth. See Using the Gecko Bootloader with the Silicon Labs Bluetooth Applications for more information.
- EZSP-SPI: By enabling the EZSP-SPI communication component, the bootloader communication interface implements the EZSP protocol over SPI. This component makes the bootloader compatible with the legacy ezsp-spi-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.
- UART XMODEM: By enabling the UART XMODEM communication component, the bootloader communication interface implements the XMODEM-CRC protocol over UART. This component makes the bootloader compatible with the legacy serial-uart-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.

Compression

The Compression components provide capability for the bootloader GBL file parser to handle compressed GBL upgrade images. Each compression component provides support for one (de)compression algorithm. At the time of writing, decompression of data compressed with the LZ4 and LZMA algorithms is supported, through the *GBL Compression (LZ4)* and *GBL Compression (LZMA)* components.

Debug

This component provides the bootloader with support for debugging output. If the component is configured to enable debug prints, short debug messages will be printed over Serial Wire Output (SWO), which can be accessed in multiple ways, including using Simplicity Commander, and by connecting to port 4900 of the Wireless Starter Kit TCP/IP interface.

To turn on debug prints, enable the Debug component and select Debug prints. Select Debug asserts to enable assertions in the source code.

On Series 1 devices, also select the GPIO peripheral in the Pin Tool.

GPIO Activation

This component provides functionality to enter firmware upgrade mode automatically after reset if a GPIO pin is active during boot. The GPIO pin location and polarity are configurable.

- GPIO: By enabling the GPIO activation component, the firmware upgrade mode can be activated by the push buttons.
- EZSP GPIO: The EZSP communication protocol over SPI can be used together with this component. By enabling the EZSP GPIO component, the firmware upgrade mode can be entered by activating the *nWake* pin.

Security

Security components provide implementations of cryptographic operations as well as functionality to compute checksums and to read cryptographic keys from manufacturing tokens.

Modules include:

- AES: AES decryption functionality
- CRC16: CRC16 functionality
- CRC32: CRC32 functionality
- ECDSA: ECDSA signature verification functionality
- SHA-256: SHA-256 digest functionality

Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

These components provide the bootloader with multiple storage options for SoCs. All storage implementations have to provide an API to access image files to be upgraded. This API is based on the concept of dividing the download space into storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Some storage implementations also support a raw storage API to access the underlying storage medium. This can be used by applications to store other data in parts of the storage medium that are not used for storing firmware upgrade images. Implementations include:

- Internal Flash: The internal flash storage implementation uses the internal flash of the device for upgrade image storage. Note that this storage area is only a download space and is separate from the portion of internal flash used to hold the active application code.
- SPI Flash: Two components are available for SPI Flash storage implementation.
    1. SPI Flash Storage: The SPI flash storage implementation supports a variety of SPI flash parts. The subset of devices supported can be configured in this component in Gecko Bootloader. (The default configuration if no devices are selected is to include drivers for all supported parts.) Including support for multiple devices requires more flash space in the bootloader. The SPI flash storage implementation does not support any write protection functionality. Supported SPI flash parts are shown in Table 1.2.
    2. SPI Flash Storage SFDP: The SPI Flash storage SFDP implementation supports all the SPI flash parts that support SFDP (Serial Flash Description Parameter) Standard from JEDEC. The SPI Flash type is detected automatically by querying the SFDP Parameter table present within the flash memory. All the properties of the SPI Flash are accessed from this parameter table. This component is not configurable since the flash is automatically detected. Any flash that supports the SFDP standard can be used with the Gecko Bootloader. However, performance delays might occur when compared to using the SPI Flash Storage component.

> Note: Low power devices are recommended for battery-operated applications. Use of the other listed devices will decrease battery life due to higher quiescent current, but this can be mitigated with external shutdown FET circuitry, if desired.

Table Supported Serial Dataflash/EEPROM External Memory Parts

Manufacturer Part Number	Size (kB)	Quiescent Current (µA Typical)*
Macronix MX25R8035F (low power)	1024	0.007
Macronix MX25R6435SF (low power)	8192	0.007
Macronix MX25R3235F (low power)	4096	0.007
Spansion S25FL208K	1024	15
Winbond W25X20BVSNIG (W25X20CVSNJG for high-temperature support)	256	1
Winbond W25Q80BVSNIG (W25Q80BVSNJG for high-temperature support)	1024	1
Macronix MX25L2006EM1I-12G (MX25L2006EM1R-12G for high-temperature support)	256	2
Macronix MX25L4006E	512	2
Macronix MX25L8006EM1I-12G (MX25L8006EM1R-12G for high-temperature support)	1024	2
Macronix MX25L1606E	2048	2
Macronix MX25U1635E (2V)	2048	2
Atmel/Adesto AT25DF041A	512	15
Atmel/Adesto AT25DF081A	1024	5
Atmel/Adesto AT25SF041	512	2
Micron (Numonyx) M25P20	256	1
Micron (Numonyx) M25P40	512	1

Copyright © 2025 Silicon Laboratories. All rights reserved.

Manufacturer Part Number	Size (kB)	Quiescent Current (µA Typical)*
Micron (Numonyx) M25P80	1024	1
Micron (Numonyx) M25P16	2048	1
ISSI IS25LQ025B	32	8
ISSI IS25LQ512B	64	8
ISSI IS25LQ010B	126	8
ISSI IS25LQ020B	256	8
ISSI IS25LQ040B	512	8

* Quiescent current values are as of December 2017; check the latest part specifications for any changes.

Compatibility

1. Silicon Labs recommends building the Gecko Bootloader and the application using the same GSDK. This implies that, if the Gecko Bootloader has been built using GSDK v4.0, it is recommended that the application also be built using GSDK v4.0.
2. Backward compatibility is supported wherein the Gecko Bootloader is built using a previous SDK version and the application is built using a newer SDK version.
3. In general, Silicon Labs does not recommend building the Gecko Bootloader using a newer SDK version with the application built using an older SDK version.

The below table attempts to explain the above-mentioned guidelines:

SDK version X < SDK version Y	Application built using SDK version X	Application built using SDK version Y
Gecko Bootloader built using SDK version X	Recommended	Compatible
Gecko Bootloader built using SDK version Y	Not recommended	Recommended

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader File Format

Gecko Bootloader File Format

The GBL file format is used by the Gecko Bootloader. File formats described in this section are generated by Simplicity Commander commands. For more information, see Simplicity Commander Reference Guide.

File Structures

The GBL file format is composed of several tags that indicate a format of the subsequent data and the length of the entire tag. The format of a tag is as follows:

Tag ID	Tag Length	Tag Payload
4 bytes	4 bytes	Variable (according to tag length)

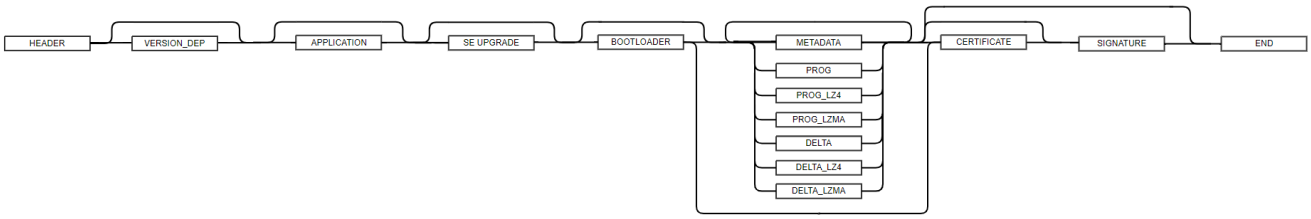
Plaintext Tag Description

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	This must be the first tag in the file. The header tag contains the version number of the GBL file specification, and flags indicating the type of GBL file - whether it is signed or encrypted.
GBL Version Dependency Tag	0x76A617EB	This optional tag contains encoded version dependencies that the software currently running on the device must satisfy before an upgrade can be attempted. Only available on Series 2 devices.
GBL Application Info Tag	0xF40A0AF4	This tag contains information about the application update image that is contained in this GBL file
GBL SE Upgrade Tag	0x5EA617EB	This tag contains a complete encrypted Secure Element update image. Only applicable on Series 2 devices.
GBL Bootloader Tag	0xF50909F5	This tag contains a complete bootloader update image.
GBL Program Data Tag	0xFE0101FE / 0xFD0303FD	This tag contains information about what application data to program at a specific address into the main flash memory.
GBL Delta Tag	0xF80A0AF8UL	This tag contains the information about the delta patch that should be used to create the new app.
GBL Program LZ4 Compressed Data Tag	0xFD0505FD	This tag contains LZ4 compressed information about what application data to program at a specific address into the main flash memory.
GBL Delta LZ4 Compressed Data Tag	0xF80B0BF8UL	This tag contains LZ4 compressed information about the delta patch that should be used to create the new app.
GBL Program LZMA Compressed Data Tag	0xFD0707FD	This tag contains LZMA compressed information about what application data to program at a specific address into the main flash memory.
GBL Delta LZMA Compressed Data Tag	0xF80C0CF8UL	This tag contains LZMA compressed information about the delta patch that should be used to create the new app.
GBL Metadata Tag	0xF60808F6	This tag contains metadata that the bootloader does not parse but can be returned to the application through a callback.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tag Name	ID	Description
GBL Certificate Tag	0xF30B0BF3	This tag contains a certificate that will be used to verify the authenticity of the GBL file.
GBL Signature Tag	0xF70A0AF7	This tag contains the ECDSA-P256 signature of all preceding data in the file.
GBL End Tag	0xFC0404FC	This tag indicates the end of the GBL file. It contains a 32-bit CRC for the entire file as an integrity check. The CRC is a non-cryptographic check. This must be the last tag.

The allowed sequence of GBL tags in a GBL file is shown in the following figure.

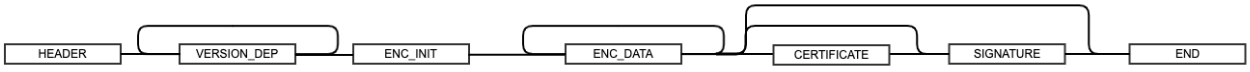


Encrypted Tag Descriptions

The encrypted GBL file format is like the unencrypted version. It introduces several new tags.

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	The GBL header is the same as for a plaintext GBL file, but the flag indicating that the GBL file is encrypted must be set.
GBL Encryption Init Header	0xFA0606FA	This contains information about the image encryption such as the Nonce and the amount of encrypted data.
GBL Encrypted Program Data	0xF90707F9	This contains an encrypted payload containing a plaintext GBL tag, one of Application Info, Bootloader, Metadata or Program Data. The data is encrypted using AES-CTR-128.

The allowed sequence of GBL tags in an encrypted GBL file is shown in the following figure.



GBL Tag Data Structures and Definitions

GBL Tag Header

```
typedef struct {
uint32_t  tagId;  // Tag ID representing the type of tag (GBL Header Tag, GBL Bootloader Tag, etc.).
uint32_t  length; // Length of the subsequent tag data in bytes.
} GblTagHeader_t;
```

GBL Header Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {

GblTagHeader_t header;  // Tag header.

uint32_t       version; // Version of the GBL file format specification. E.g. 0x03000000.

uint32_t       type;    // Flags indicating whether the GBL file is encrypted and/or signed.

// See definitions below.

} GblHeader_t;
```

```
// GBL types

#define GBL_TYPE_NONE                0x00000000UL

#define GBL_TYPE_ENCRYPTION_AESCCM   0x00000001UL

#define GBL_TYPE_SIGNATURE_ECDSA     0x00000100UL
```

GBL Version Dependency Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
uint8_t  imageType;  // Type of image (application, bootloader, SE)
uint8_t  statement;  // Encoded dependency statement (ex. appVersion > (0).1.2.3)
uint16_t reserved;   // Reserved
uint32_t version;    // The version number used in the statement (ex. (0).1.2.3)
} VersionDependency_t;
// Image types
#define GBL_VERSION_DEPENDENCY_TYPE_APPLICATION             0x01U
#define GBL_VERSION_DEPENDENCY_TYPE_BOOTLOADER              0x02U
#define GBL_VERSION_DEPENDENCY_TYPE_SE                      0x03U
// Operator encoding
#define GBL_VERSION_DEPENDENCY_OPERATOR_MASK                0x0FU
#define GBL_VERSION_DEPENDENCY_OPERATOR_SHIFT               0x00U
#define GBL_VERSION_DEPENDENCY_OPERATOR_TYPE_MASK           0x0EU
#define GBL_VERSION_DEPENDENCY_OPERATOR_NEGATOR_BIT_MASK    0x01U
#define GBL_VERSION_DEPENDENCY_OPERATOR_LT                  0x00U
#define GBL_VERSION_DEPENDENCY_OPERATOR_LEQ                 0x02U
#define GBL_VERSION_DEPENDENCY_OPERATOR_EQ                  0x04U
#define GBL_VERSION_DEPENDENCY_OPERATOR_GEQ                 0x06U
#define GBL_VERSION_DEPENDENCY_OPERATOR_GT                  0x08U
// Connective encoding
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_MASK              0xF0U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_SHIFT             0x04U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_TYPE_MASK         0x0EU
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_NEGATOR_BIT_MASK  0x01U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_AND               0x00U
// SE version mask for ignoring the compatibility byte when comparing versions
#define GBL_VERSION_DEPENDENCY_SE_VERSION_MASK              0x00FFFFFFUL
```

GBL Application Info Tag

```
typedef struct {
GblTagHeader_t    header;  // Tag header.
ApplicationData_t appInfo; // Application information structure. See definition below.
} GblApplication_t;
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct ApplicationData {
uint32_t type;           // Bitfield representing the type of application.
// See definitions below for possible values.
uint32_t version;        // Version number for this application (customer-defined).
uint32_t capabilities;   // Bitfield representing the capabilities of this application.
uint8_t  productId[16];  // Unique ID (UUID or GUID) for the product this application is built for.
} ApplicationData_t;
```

```
// Application types
#define APPLICATION_TYPE_ZIGBEE          (1UL << 0UL)
#define APPLICATION_TYPE_THREAD          (1UL << 1UL)
#define APPLICATION_TYPE_FLEX            (1UL << 2UL)
#define APPLICATION_TYPE_BLUETOOTH       (1UL << 3UL)
#define APPLICATION_TYPE_MCU             (1UL << 4UL)
#define APPLICATION_TYPE_BLUETOOTH_APP   (1UL << 5UL)
#define APPLICATION_TYPE_BOOTLOADER      (1UL << 6UL)
#define APPLICATION_TYPE_ZWAVE           (1UL << 7UL)
```

GBL SE Upgrade Tag

```
typedef struct {
GblTagHeader_t header;   // Tag header.
uint32_t       blobSize; // Size of the SE upgrade data blob.
uint32_t       version;  // Version of the SE image.
uint8_t        data[];   // Array of data containing the SE upgrade blob.
} GblSeUpgrade_t;
```

GBL Bootloader Tag

```
typedef struct {
GblTagHeader_t header;            // Tag header.
uint32_t       bootloaderVersion; // Version number of the bootloader.
uint32_t       address;           // Base address of the bootloader image.
uint8_t        data[];            // Array of data containing the bootloader upgrade image.
} GblBootloader_t;
```

GBL Program Data Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
GblTagHeader_t header;            // Tag header.
uint32_t       flashStartAddress; // Address at which to start flashing data.
uint8_t        data[];            // Array of data to be flashed
// (compressed data in the LZ4 and LZMA variants of the tag).
} GblProg_t;
```

GBL Metadata Tag

```
typedef struct {
GblTagHeader_t header;     // Tag header.
uint8_t        metadata[]; // Array containing the metadata.
} GblMetadata_t;
```

GBL Certificate Tag

```
typedef struct {
GblTagHeader_t           header;      // Tag header.
ApplicationCertificate_t certificate; // Certificate used to verify the GBL file. See definition below.
} GblCertificateEcdsaP256_t;
typedef struct ApplicationCertificate {
uint8_t  structVersion;     // Version of the certificate structure.
uint8_t  flags[3];          // Reserved flags.
uint8_t  key[64];           // Public key used to verify the GBL file.
uint32_t version;           // Version number of this certificate.
uint8_t  signature[64];     // The signature of the certificate itself (not the GBL file).
} ApplicationCertificate_t;
```

GBL Signature Tag

```
typedef struct {
GblTagHeader_t header; // Tag header.
uint8_t        r[32];  // The r value of the ECDSA signature.
uint8_t        s[32];  // The s value of the ECDSA signature.
} GblSignatureEcdsaP256_t;
```

GBL End Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
GblTagHeader_t header; // Tag header.
uint32_t       gblCrc; // CRC32 checksum of the entire GBL file.
} GblEnd_t;
```

GBL Encryption Init Header Tag

```
typedef struct {
GblTagHeader_t header;    // Tag header.
uint32_t       msgLen;    // Length of the ciphertext in bytes.
uint8_t        nonce[12]; // Random AES-CTR nonce.
} GblEncryptionInitAesCcm_t;
```

GBL Encrypted Data Tag

```
typedef struct {
GblTagHeader_t header;             // Tag header.
uint8_t        encryptedGblData[]; // Array containing the ciphertext (encrypted GBL data).
// Note that the corresponding plaintext must contain one or
// more complete GBL tags. Put differently: A single plaintext GBL
// tag cannot be split across multiple encrypted tags.
} GblEncryptionData_t;
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Application Upgrade

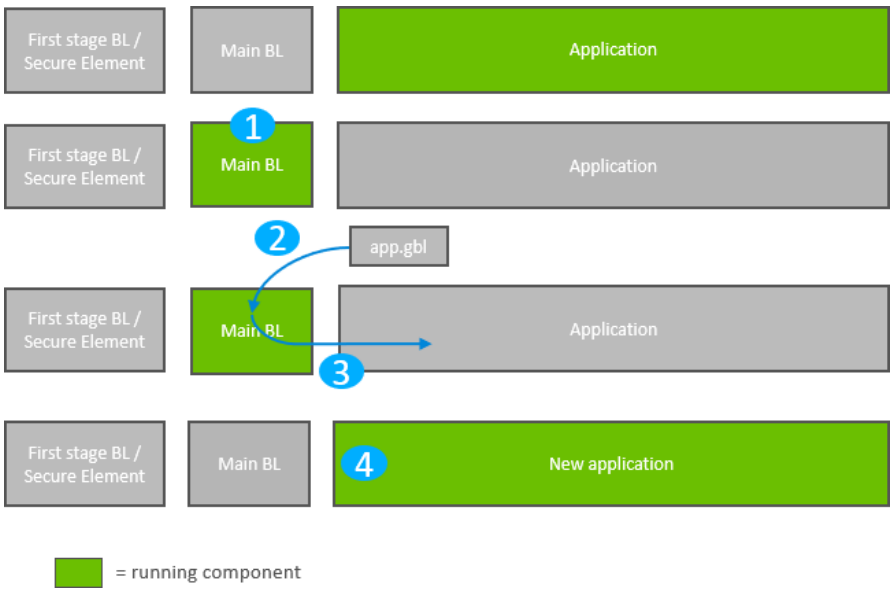
Gecko Bootloader Operation - Application Upgrade

This section summarizes Gecko Bootloader operation for updating application firmware, first if the Gecko Bootloader is configured in standalone mode and then if it is configured in application mode. Section Gecko Bootloader Operation - Bootloader Upgrade provides the same information for updating the bootloader firmware.

The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the section "Memory Space for Bootloading" in *UG103.6: Bootloader Fundamentals*.

Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing an application image is transmitted from the host to the device. If image encryption is enabled in the main stage bootloader and the image is encrypted, decryption is performed during the process of receiving and parsing the GBL file.
3. The bootloader applies the application upgrade from the GBL upgrade file on-the-fly. If image authentication is enabled in the main stage bootloader and the GBL file contains a signature, the authenticity of the image is verified before completing the process.
4. The device boots into the application. Application upgrade is complete.

Rebooting Into the Bootloader

The Gecko Bootloader supports multiple mechanisms for triggering the bootloader. If the GPIO activation component is installed, the host device can keep this pin low/high (depending on configuration) through reset to make the device enter the bootloader. The bootloader can also be entered through software. The `bootloader_rebootAndInstall` API first signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it will enter firmware upgrade mode instead of booting the existing application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Downloading and Applying a GBL Upgrade File

When the bootloader enters firmware upgrade mode, it enters a receive loop waiting for data from the host device. The specifics of the receive loop depend on the protocol. Received packets are passed to the image parser, a state machine that parses the data and returns a callback containing any data that should be acted upon. The bootloader core implements this callback and flashes the data to internal flash at the address specified. If GBL file authentication or encryption is enabled, the image parser will enforce this, and abort the image upgrade if the authentication fails.

The bootloader prevents a newly uploaded image from being bootable by holding back parts of the application vector table until the GBL file CRC and GBL signature (if required) have been verified.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If a valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

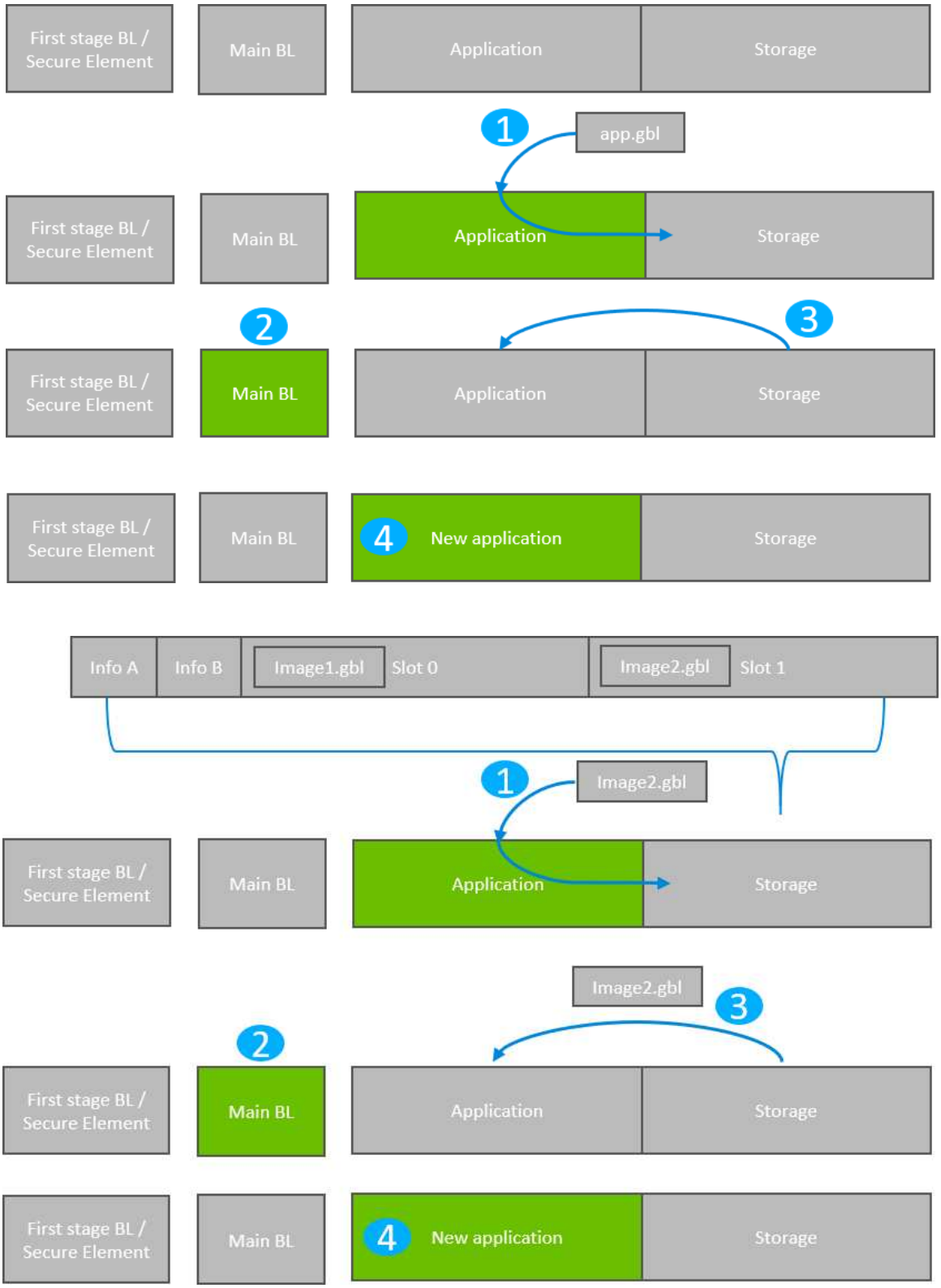
Error Handling

If the application upgrade is interrupted at any time, the device will be without a working application. The bootloader then resets the device, and re-enters firmware upgrade mode. The host device can easily restart the application upgrade process, to try loading the upgrade image again.

Application Bootloader Operation

The following figure illustrates the application bootloader operation both for a single image/single storage slot, and multiple images/multiple storage slots.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A GBL file is downloaded onto the storage medium of the device (internal flash or external dataflash), as described below, and the presence of an upgrade image is indicated.
2. The device reboots into the bootloader, and the bootloader enters firmware upgrade mode.
3. The bootloader applies the application upgrade from the GBL upgrade file.
4. The device boots into the application. Application upgrade is complete.

Downloading and Storing a GBL Upgrade Image File

Copyright © 2025 Silicon Laboratories. All rights reserved.

To prepare for receiving an upgrade image, the application finds an available storage slot, or erases an existing one using `bootloader_eraseStorageSlot`. If the bootloader only supports a single storage slot, a value of 0 should be used for the slot ID.

The application then receives a GBL file using an applicable protocol, such as Ethernet, USB, Zigbee, OpenThread, or Bluetooth, and stores it in the slot by calling `bootloader_writeStorage`.

When download is complete, the application can optionally verify the integrity of the GBL file by calling `bootloader_verifyImage`. This is also done by the bootloader before applying the image but can be done from the application in order to avoid rebooting into the bootloader if the received image was corrupt.

If multiple storage slots are supported, the application should write a bootload list by calling `bootloader_setImageToBootload`. The list is written to the two bootload info pages as shown in Figure The bootload list is a prioritized list of slots indicating the order the bootloader should use when attempting to perform a firmware upgrade. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader treats the entire download space as a single storage slot.

Rebooting and Applying a GBL Upgrade File

The bootloader can be entered through software. The `bootloader_rebootAndInstall` API signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it enters firmware upgrade mode instead of booting the existing application.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the image stored in each. Once it finds a valid GBL upgrade file, firmware upgrade is attempted from this GBL file. If the upgrade fails, the bootloader moves to the next image in the list. If no images pass verification, the bootloader reboots back into the existing application with a message in the shared memory location in SRAM indicating that no good upgrade images were found.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Bootloader Upgrade

Gecko Bootloader Operation - Bootloader Upgrade

Bootloader upgrade functionality is provided by the first stage bootloader on Series 1 devices, or the Secure Engine on Series 2 devices. The Secure Engine itself is also upgradable. For more details, see section Gecko Bootloader Operation - Secure Engine Upgrade. On Series 1 devices, the first stage bootloader is not upgradable.

Requirements for upgrading the main bootloader vary depending on the bootloader configuration:

- Application bootloader with storage: Upgrading the main bootloader requires a single GBL file containing both bootloader and application upgrade images.
- Standalone bootloader with communication interface: Upgrading the bootloader requires two GBL files, one with only the bootloader upgrade image, and one with only the application upgrade image.

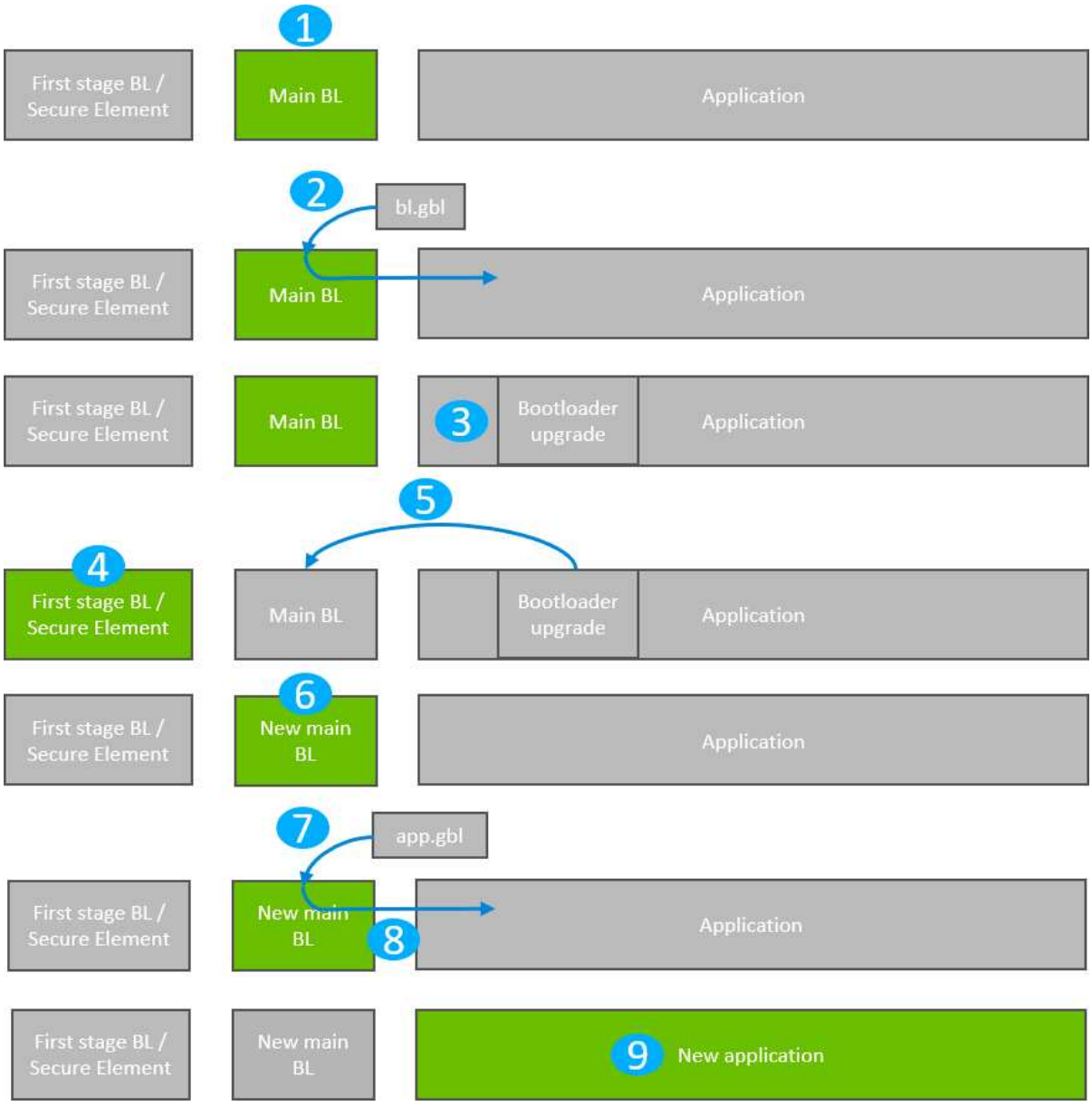
Security of the bootloader upgrade process is provided by signing the GBL file. See section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.

The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the “Memory Space for Bootloading” section in *UG103.6: Bootloader Fundamentals*. For convenience, the figures do not distinguish between SE and VSE.

Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. The device reboots into the bootloader.
2. A GBL file containing only a bootloader upgrade image is transmitted from the host to the device.
3. The contents of the GBL Bootloader tag are written to the bootloader upgrade location in internal flash, overwriting the existing application.
4. The device reboots into the first stage bootloader / Secure Engine.
5. The first stage bootloader / Secure Engine replaces the main bootloader with the new version found in the bootloader upgrade location.
6. The device boots into the new main bootloader.
7. A GBL file containing only an application image is transmitted from the host to the device.
8. The bootloader applies the application image from the GBL upgrade file on-the-fly.
9. The device boots into the application. Bootloader upgrade is complete.

A bootloader upgrade is started in the same way as an application upgrade.

Downloading and Applying a Bootloader GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a bootloader upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

upgrade data in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file CRC and GBL signature (if required) have been verified.

When a complete bootloader upgrade image is received, the main bootloader signals the first stage bootloader / Secure Engine that it should enter firmware upgrade mode. On Series 1 devices, this is done by writing a command to the shared memory location at the bottom of SRAM, and then performing a software reset. On Series 2 devices, Secure Engine communication is used to signal that bootloader upgrade is ready to be performed.

On Series 1 devices, the first stage bootloader verifies the CRC on the current main bootloader and verifies the CRC of the bootloader upgrade present in the bootloader upgrade location in internal flash.

- If the CRC in the upgrade image location and the CRC in the current main bootloader are both valid, then the upgrade image will be copied over the main bootloader if the version number of the upgrade is higher than the current main bootloader version.
- If the CRC in the upgrade image location is valid and the CRC in the current main bootloader location is not valid, the upgrade image will be copied over the main bootloader regardless of the version. This is because the version of the main bootloader cannot be relied upon if the main bootloader image is corrupt.
- If the CRC in the upgrade location is not valid, the upgrade will not be applied.

On Series 2 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See section Setting a Version Number for more information about versioning bootloader images.

Upgrading Bootloaders Without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature, using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
    - Turn on secure boot in Simplicity Studio by going to the Bootloader Core component and selecting the Enable secure boot option.
    - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device as long as the Lock Bits page is configured to allow flash writes. If the Lock Bits page is locked, it can only be erased by the debugger. Therefore, signing/decryption keys residing in the Lock Bits page cannot be erased from firmware. This means that, for devices in the field, those areas in flash cannot be replaced with new ones. However, the Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c`.
4. Create a GBL file using the Gecko Bootloader image. The GBL file needs to be signed/unsigned depending on the current configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
5. Upload the GBL file. For more details on the upgrade process, see section Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders).

Enabling Secure Boot RTSL on Series 2 Devices

Secure Boot RTSL (Root of Trust and Secure Loader) can be enabled using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
    - Turn on secure boot in Simplicity Studio by going to the Bootloader Core component and selecting the Enable secure boot option.
    - For EFR32xG21, in the Bootloader Core component, disable the Allow use of public key from manufacturing token storage option. This means that the Gecko Bootloader will never make use of the public key stored in the last page of the main flash
    - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.

Copyright © 2025 Silicon Laboratories. All rights reserved.

2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in the VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an unsigned GBL file from this application and upload it. For more information on installing public keys, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine, see Series 2 and Series 3 Secure Boot with RTSL.
6. Create a GBL file using the Gecko Bootloader image from step 4.
7. Create a GBL file using the application from step 5. The GBL file need to be signed if the Bootloader Core component option Require signed firmware upgrade files was selected in step 1.
8. Upload the GBL file containing the Gecko Bootloader image.
9. Upload the GBL file containing the application.

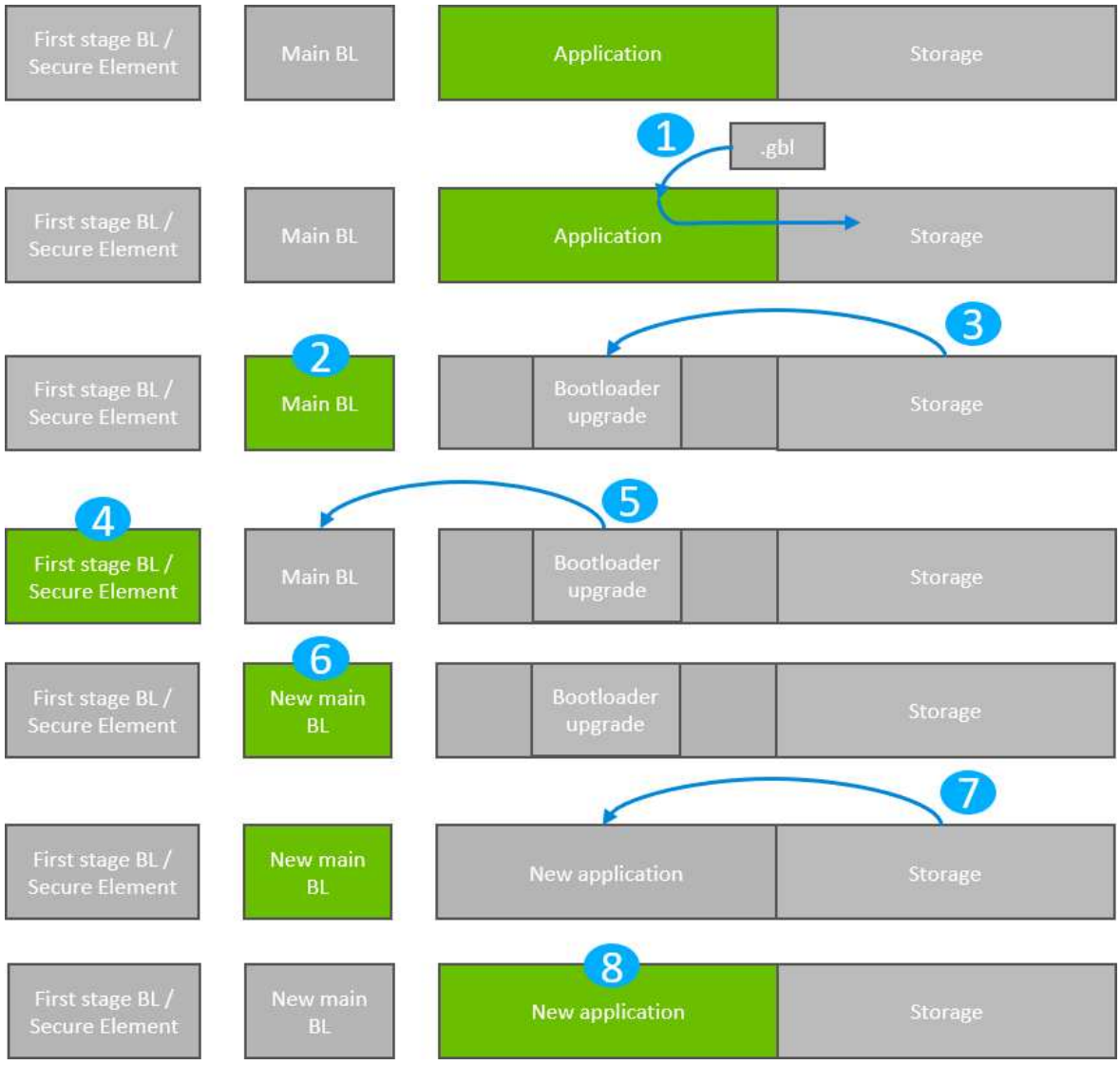
Downloading and Applying an Application GBL Upgrade File

Once the bootloader upgrade is completed, the existing application is rendered invalid, since the bootloader upgrade location overlaps with the application. A GBL upgrade file containing an application upgrade is transmitted to the bootloader. The application upgrade process follows that in section Standalone Bootloader Operation.

Bootloader Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A single GBL file containing both a bootloader upgrade image and an application image is downloaded onto the storage medium of the device (internal flash or external SPI flash).
2. The device reboots into the main bootloader.
3. The main bootloader parses its upgrade image into internal flash at the bootloader upgrade location. Depending on the application size as well as how the “scratch space” is configured for the bootloader on Series 2 devices, the existing application may be overwritten.
4. The main bootloader verifies the integrity of the upgrade image and then resets the device with reset reason BOOTLOADER_RESET_REASON_UPGRADE to apply the upgrade.
5. The device reboots into the first stage bootloader / Secure Engine.
6. The first stage bootloader / Secure Engine replaces the main bootloader with the new version.
7. The device boots into the new main bootloader.
8. The bootloader applies the application image from the GBL upgrade file.
9. The device boots into the application. Bootloader upgrade is complete.

A bootloader upgrade is started in the same way as an Application Upgrade. A single GBL file containing both a bootloader and an application upgrade is written to storage by the application, and the bootloader is entered.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the GBL file stored within. Verification returns information about whether the GBL file contains an application, or both a bootloader and an application. The

Copyright © 2025 Silicon Laboratories. All rights reserved.

image parser parses the file. If the GBL file contains a bootloader, the bootloader upgrade data is returned in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file CRC and GBL signature (if required) have been verified.

On Series 1 devices, the main bootloader signals the first stage bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performing a software reset. On Series 2 devices, Secure Engine communication interface is used to signal the Secure Engine that a bootloader upgrade is ready to be performed.

On Series 1 devices, the first stage bootloader verifies the CRC of the bootloader upgrade present in the bootloader upgrade location in internal flash and copies the bootloader upgrade over the main bootloader if the version number of the upgrade is higher than the version number of the existing main bootloader. On Series 2 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See section Setting a Version Number for more information about versioning bootloader images.

The new main bootloader is entered, and the images in the list of storage slots marked for bootload are verified. When the image parser parses the slot containing the GBL file with the bootloader + application upgrade, the version number of the bootloader upgrade is equal to the running main bootloader version, so another bootloader upgrade will not be performed. Instead, the application upgrade data are returned in a callback. Bootloading of the new application proceeds as described in section Application Bootloader Operation.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images. Depending on the configuration, the bootloader size can vary. For size requirements of the bootloader, see section Size Requirements for Different Bootloader Configurations for Series 1 Devices.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature. The procedure is as follows:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot from the Bootloader Core component in Simplicity Studio by selecting the Enable secure boot option.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device as long as the Lock Bits page is configured to allow flash writes. If the Lock Bits page is locked, it can only be erased by the debugger. Therefore, signing/decryption keys residing in the Lock Bits page cannot be erased from firmware. This means that, for devices in the field, those areas in flash cannot be replaced with new ones. However, the Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c`.
4. Prepare a signed application image using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing an application.
5. Create a GBL file using the Gecko Bootloader image and the signed application image. The GBL file needs to be signed/unsigned depending on the configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
6. Upload the GBL file. For more details on the upgrade process, see section Bootloader Upgrade on Application Bootloaders with Storage.

Enabling Secure Boot RTSL on Series 2 Devices

Secure Boot RTSL can be enabled by using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot from the Bootloader Core component in Simplicity Studio by selecting the Enable secure boot option.
   - For EFR32xG21, in the Bootloader Core component, disable the Allow use of public key from manufacturing token storage option. This means that the Gecko Bootloader will never make use of the public key stored in the last page of the main flash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

    - (Optional) In the Bootloader Core component, select the Require signed firmware upgrade files option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an un-signed GBL file from this application and upload it. For more information on installing public keys, see Series 2 and Series 3 Secure Boot with RTSL. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine, see Series 2 and Series 3 Secure Boot with RTSL.
6. Create a GBL file using the Gecko Bootloader image from step 4 and the application from step 5. The GBL file must be signed if the Bootloader Core component option Require signed firmware upgrade files was selected in step 1. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
7. Upload the GBL file containing the Gecko Bootloader image and the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Secure Engine Upgrade

Gecko Bootloader Operation - Secure Engine Upgrade

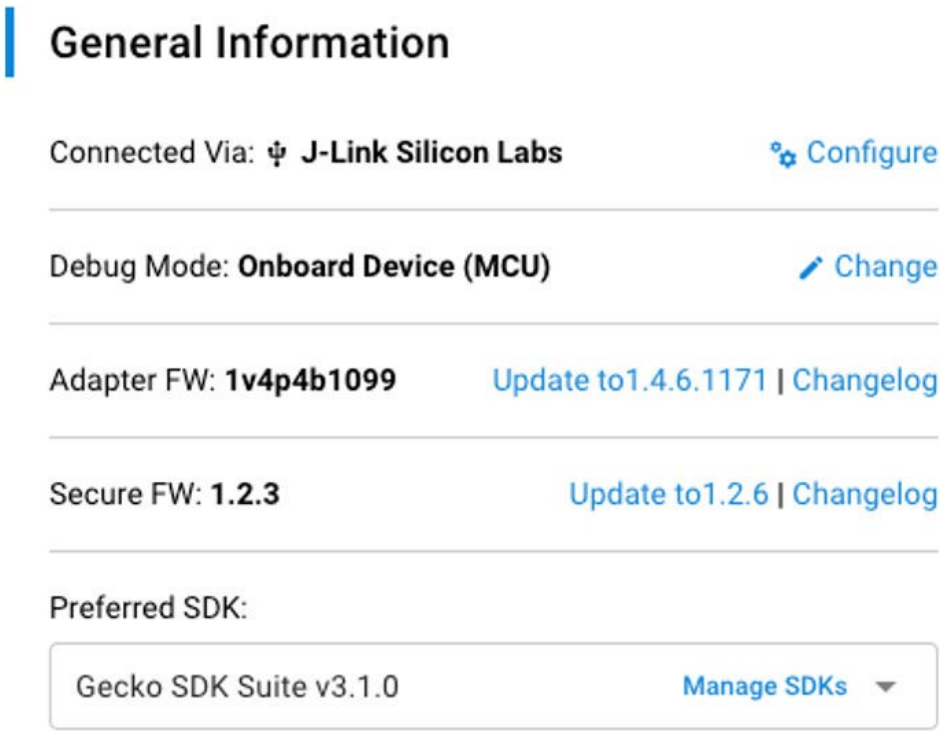
The Secure Engine is upgradable and requirements for upgrading the Secure Engine vary depending on the bootloader configuration:

- Application bootloader with storage: Upgrading the Secure Engine requires a single GBL file containing both Secure Engine and application upgrade images.
- Standalone bootloader with communication interface: Upgrading the Secure Engine requires two GBL files, one with only the Secure Engine upgrade image, and one with only the application upgrade image and optionally a third image containing only the Main bootloader upgrade.

A bootloader upgrade can also be included in the same GBL file in application mode, or as a third GBL file in standalone mode. The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the section “Memory Space for Bootloading” in *UG103.6: Bootloader Fundamentals*.

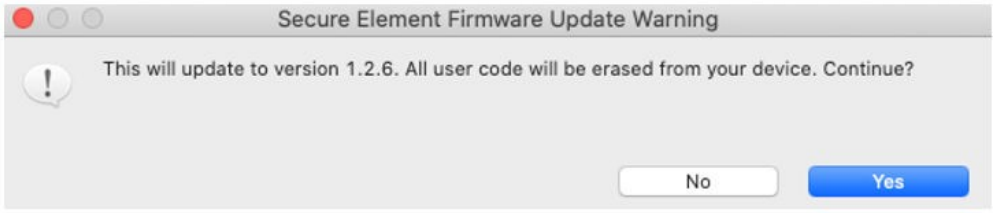
Signed and encrypted Secure Engine upgrade images are provided by Silicon Labs through Simplicity Studio. Upgrade images with the same or lower version number than the running Secure Engine will be ignored.

To download Secure Engine firmware images, connect a Series 2 device and select a preferred SDK. The Secure Firmware Update to x.x.x link appears in the Launcher Perspective, as shown in the following figure.



Click Update to x.x.x next to Secure FW: x.x.x. A warning dialog box appears. Click [Yes] to continue.

Copyright © 2025 Silicon Laboratories. All rights reserved.



The Launcher Perspective is then updated so that the current Secure Firmware version and link version are the same.

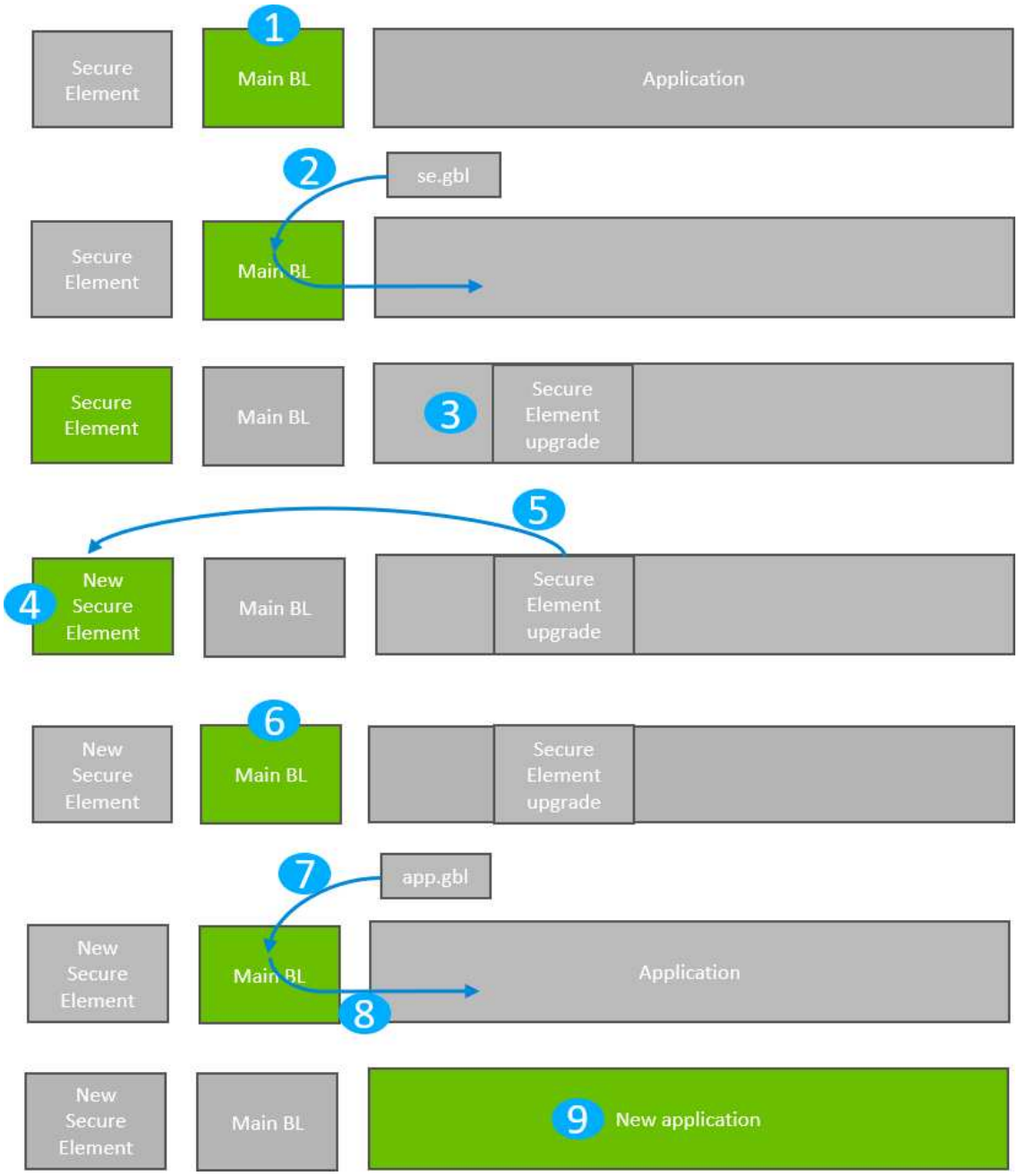


The Secure Engine firmware images can be found in the *util/se_release/public* directory under the Gecko SDK. Simplicity Studio displays the SE firmware version available in the Gecko SDK selected.

Secure Engine Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. The device reboots into the bootloader.
2. A GBL file containing only a Secure Engine upgrade image is transmitted from the host to the device.
3. The contents of the GBL Secure Engine tag are written to the pre-configured upgrade location in internal flash, overwriting the existing application.
4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location.
6. The device boots into the main bootloader.
7. A GBL file containing only an application image is transmitted from the host to the device.
8. The bootloader applies the application image from the GBL upgrade on the fly.
9. The device boots into the application. Secure Engine upgrade is complete.

Downloading and Applying a Secure Engine GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a Secure Engine upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns Secure

Copyright © 2025 Silicon Laboratories. All rights reserved.

Engine upgrade data in a callback. The bootloader core implements this callback and flashes the data to internal flash at the pre-configured bootloader upgrade location.

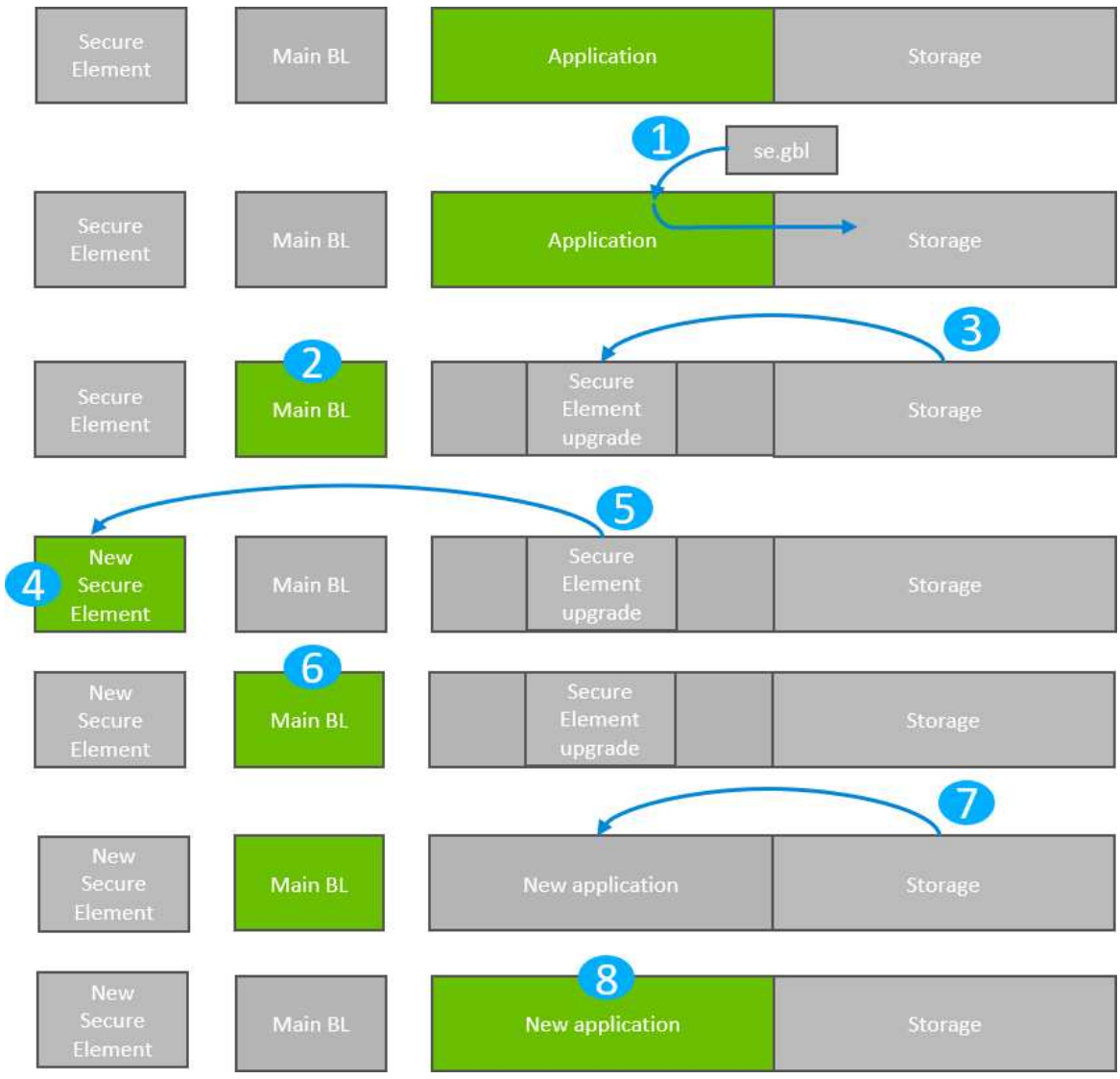
When a complete Secure Engine upgrade image is received, the main bootloader signals the Secure Engine that it should enter firmware upgrade mode. This is done by the Secure Engine communication interface that is used to signal that bootloader upgrade is ready to be performed.

Downloading and Applying an Application GBL Upgrade File

Once the Secure Engine upgrade is completed, the existing application is rendered invalid if the Secure Engine upgrade location overlaps with the application. A GBL upgrade file containing an application upgrade is transmitted to the bootloader. The application upgrade process follows that. For more information, see section Standalone Bootloader Operation.

Secure Engine Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



1. A single GBL file containing both a Secure Engine upgrade image and an application image is downloaded onto the storage medium of the device (internal flash or external SPI flash).
2. The device reboots into the bootloader.
3. a) The main bootloader copies its upgrade image into internal flash at the pre-configured upgrade location.
   b) Alternatively, if the no-staging Secure Engine upgrade option has been enabled, the upgrade image will be fetched directly from the GBL file in storage instead of first copying the image to the pre-configured upgrade location.

Copyright © 2025 Silicon Laboratories. All rights reserved.

4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location (or directly from storage, ref. 3b).
6. The device boots into the main bootloader.
7. The bootloader applies the application image from the GBL upgrade file.
8. The device boots into the application. Secure Engine upgrade is complete.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images. The following table shows the reserved SE upgrade image sizes.

Device Family	Reserved Flash for SE Upgrade Image
EFR32xG21	48 kB
EFR32xG22	24 kB
EFR32xG23	96 kB

Depending on the configuration, the bootloader size can vary. For size requirements of the bootloader, see section Size Requirements for Different Bootloader Configurations for Series 1 Devices. The bootloader size for EFR32xG21 devices can be up to 16 kB and for EFR32xG22, EFR32xG23, and EFR32xG24 devices the bootloader size can be up to 24 kB. For more details, see *UG103.6: Bootloader Fundamentals*.

Copyright © 2025 Silicon Laboratories. All rights reserved.

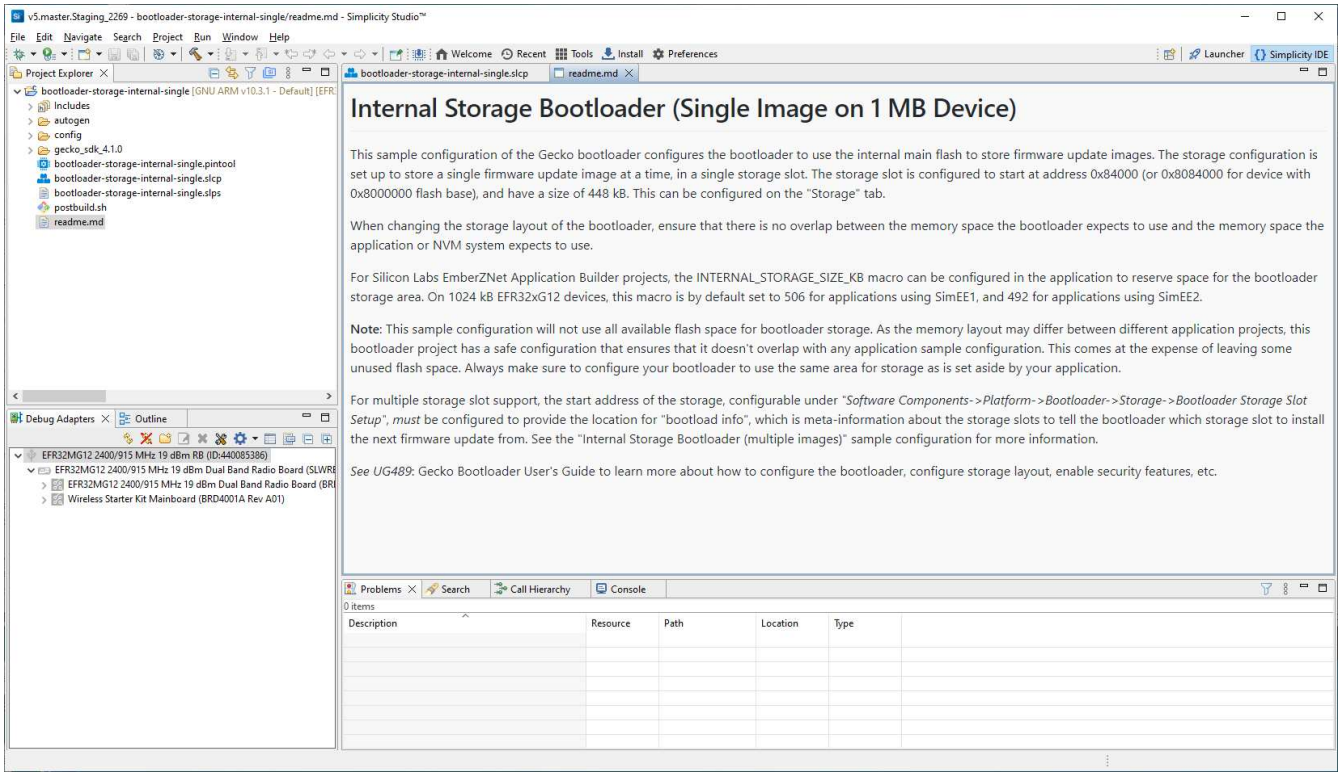
Getting Started with the Gecko Bootloader

Getting Started with the Gecko Bootloader

This section describes how to build a Gecko Bootloader from one of the provided examples. Simplicity Studio 5, used with Gecko SDK Suite (GSDK) v3.x and higher, differs from Simplicity Studio 4, used with GSDK v2.x, in how sample applications are selected and how projects are created. Refer to the documentation provided with your SDK for details. These instructions assume that you have installed Simplicity Studio 5, the GSDK and associated utilities as described in the SDK's quick start guide, and that you are familiar with generating, compiling, and flashing an example application in the relevant version.

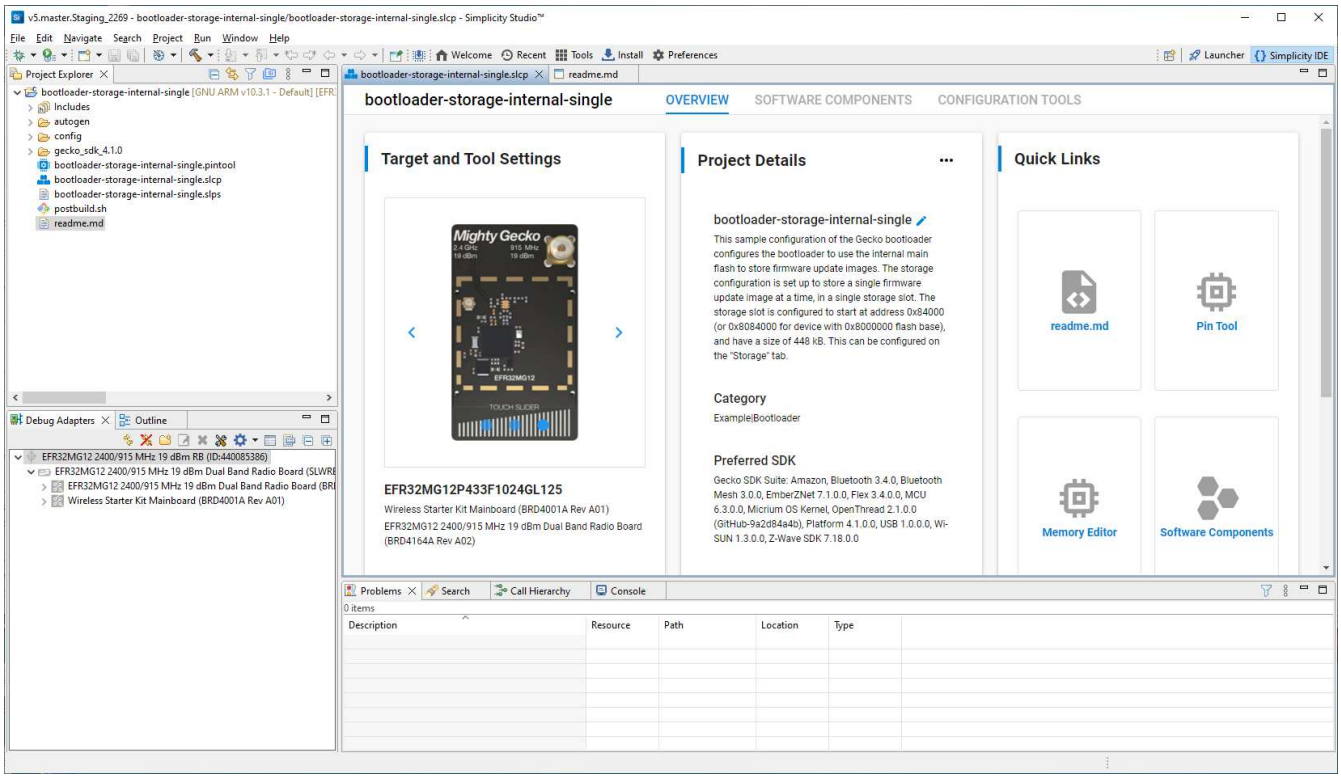
- *QSG169: Bluetooth® SDK v3.x Quick-Start Guide*
- Proprietary Flex SDK v3.x Quick-Start Guide
- *QSG170: Silicon Labs OpenThread SDK Quick-Start Guide*
- *QSG180: Zigbee EmberZNet Quick-Start Guide for SDK 7.0 and Higher*

1. Create a project based on the Gecko Bootloader example of your choice. The project opens with a tab describing the example.

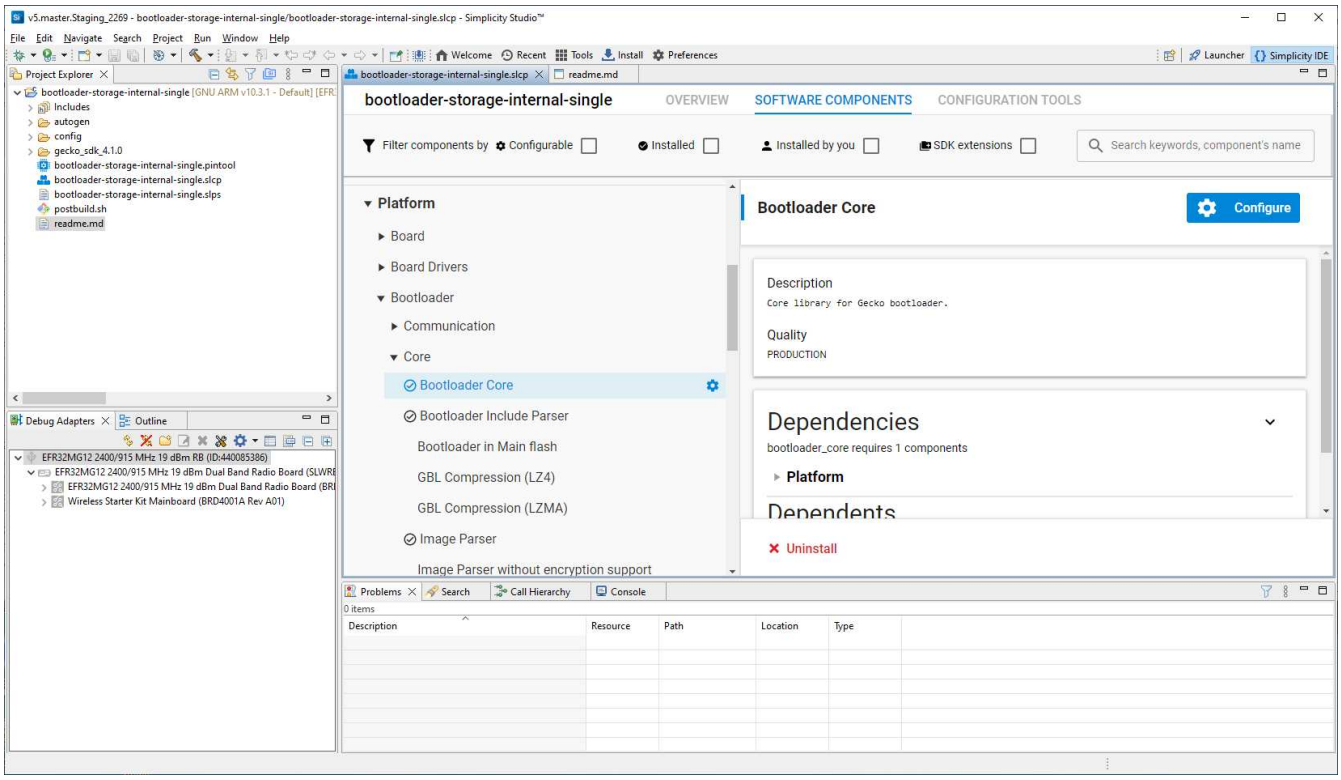


2. Click the project (*.slcp) tab to move to the Project Configurator interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

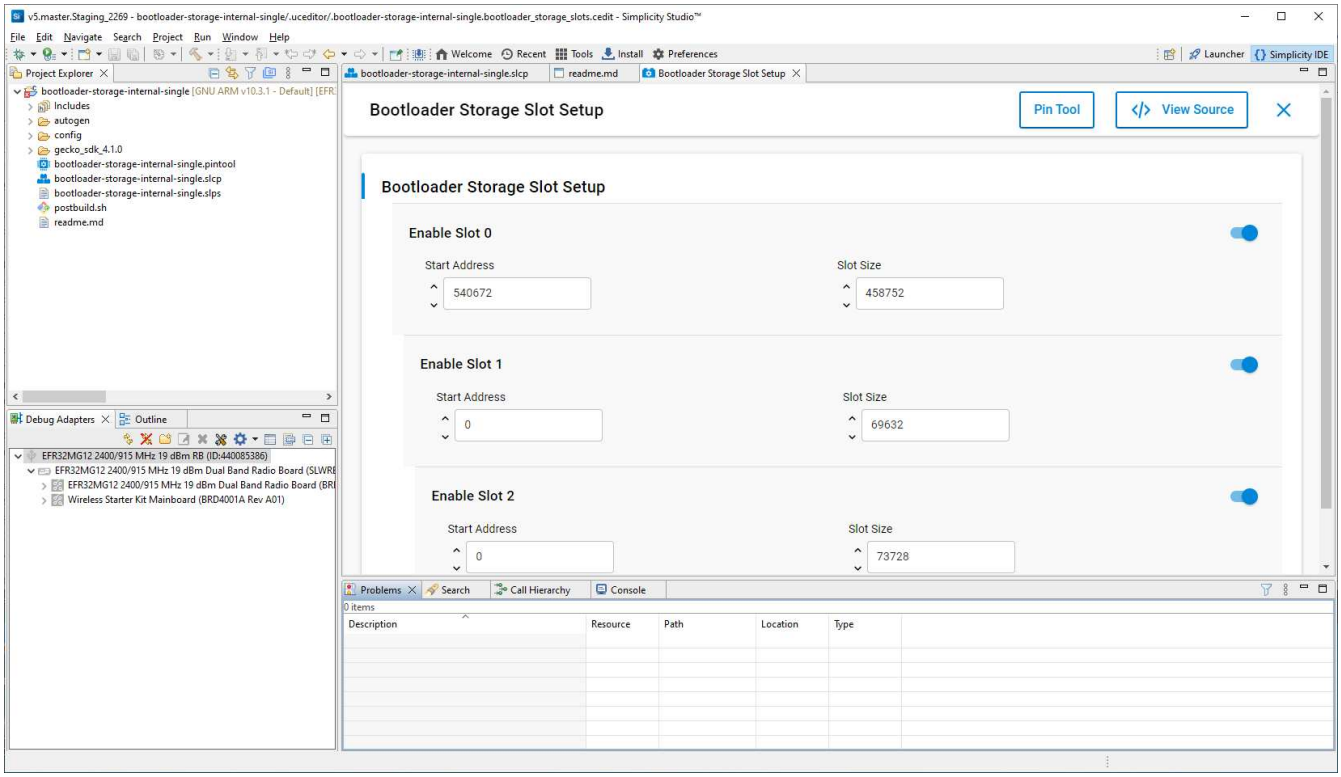


3. The Software Components tab shows the list of available components that can be installed in the project.

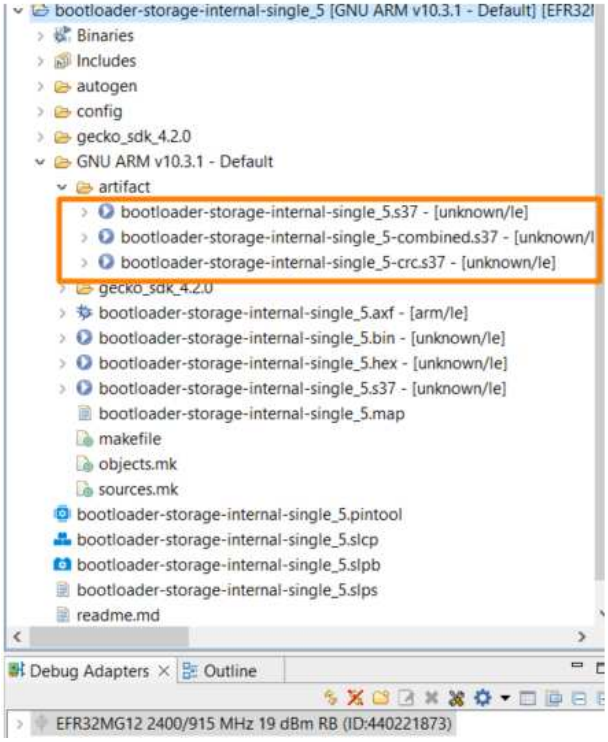


4. The Storage Slot Setup component allows you to configure storage slots to be used if a storage component is also installed. The default configuration matches the target part and bootloader type. This component supports a maximum of three storage slots.

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. Click the Build (hammer) icon.
6. After the build is complete, the bootloader binaries are available in the artifact folder as depicted in the image below.



On Series 1 devices, three bootloader images are generated into the build directory: a main bootloader, a main bootloader with CRC32 checksum, and a combined first stage and main bootloader with CRC32 checksum. The main bootloader image is called <project- name>.s37, the main bootloader with CRC32 checksum is called <projectname>-crc.s37, while the combined first stage

Copyright © 2025 Silicon Laboratories. All rights reserved.

image + main bootloader image with a CRC32 checksum is called <projectname>-combined.s37. The first time a device is programmed, whether during development or manufacturing, the combined image needs to be programmed. For subsequent programming, when a first stage bootloader is already present on the device, it is okay to download an image containing just the main bootloader. In this case, the main bootloader with CRC32 should be used.

The requirement is that any main bootloader image that is programmed via serial wire must contain the CRC32 in the image. Files downloaded via serial wire are “s37” files. Most often, the <projectname>-combined.s37 file is the one downloaded during production programming. However, it is possible to download only the main bootloader over serial wire, in which case <projectname>-crc.s37 should be used.

Any main bootloader that is upgraded with the OTA or host method should already contain CRC32 because bootloader-initiated upgrades use GBL files (not “s37” files) and Simplicity Commander adds the CRC32 when it constructs the GBL file. The input files to Simplicity Commander can (and should) use the non-CRC “s37” file.

On Series 2 devices, the combined image is not present, since the first stage bootloader does not exist. The image containing only a main bootloader is the image that must be used to create a GBL file for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuring the Gecko Bootloader

Configuring the Gecko Bootloader

Configuring Storage

Gecko Bootloaders configured as application bootloaders must include an API to store and access image files. This API is based on the concept of storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Slots are configured in the Bootloader Storage Slot Setup component.

When multiple storage slots are configured, a bootload list is used to indicate the order in which the bootloader should access slots to find upgrade images. If multiple storage slots are supported, the application should write the bootload list by calling bootloader_setImageToBootload before rebooting into the bootloader to initiate a firmware upgrade process. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader uses this slot implicitly. A maximum of three slots may be configured in the Bootloader Storage Slot Setup component.

SPI Flash Storage Configuration

When configuring a Gecko Bootloader to obtain images from SPI flash, modify the following.

The base address of the storage area should be configured in the Common Storage component. This is the address at which the bootloader places the bootload list, if more than one storage slot is configured. In the default configuration, this address is set to 0. If only a single storage slot is configured, the bootload list is not used, so configuring it may be omitted.

The location and size of the storage slots can be configured using the Bootloader Storage Slot Setup component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, space must be reserved between the base address as configured in the Common Storage component and the first storage slot configured in the Bootloader Storage Slot Setup component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. In the default example application, a SPI flash part with 4 kB flash sectors is used. This means that 8 kB must be reserved before the first storage slot. The following figure illustrates how the storage area can be partitioned, where the numbers in the top row represent the starting addresses.

External storage with two storage slots

0	4 kB	8 kB	260 kB
Bootload list	Bootload list	Storage slot 0 Size = 252 kB	Storage Slot 1 Size = 252 kB

External storage with one storage slot

0
Storage slot 0 Size = 512 kB

Internal Storage Configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

When configuring a Gecko Bootloader to obtain images from internal flash, modify the following.

The base address of the storage area should be configured in the Common Storage component. This is the address at which the bootloader will place the prioritized list of storage slots to attempt to bootload from, if more than one storage slot is configured. In the default configuration, only a single storage slot is configured, so this value is set to 0, and isn't used. If more than one storage slot is configured, this value needs to be configured too.

The location and size of the storage slots can be configured using the Bootloader Storage Slot Setup Component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, enough space must be reserved between the base address as configured in the Common Storage component and the first storage slot configured in the Bootloader Storage Slot Setup component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. The following figure illustrates how the storage area can be partitioned.

Internal storage with one storage slot

0	512 kB
Application	Storage Slot 0 Size = 512 kB

Internal storage with two storage slots

0	340 kB	342 kB	344 kB	648 kB
Application	Bootload list	Bootload list	Storage slot 0 Size = 340 kB	Storage slot 1 Size = 340 kB

> Note: The storage area partitioning in the example for two storage slots above does not take any NVM system into account. If using an NVM system like SimEE or PS Store, take care to place and size the storage area in such a way that bootloader storage does not overlap with NVM.

Compressed Upgrade Images

The Gecko Bootloader optionally supports compressed GBL files. In a compressed GBL file, only the application upgrade data is compressed, any metadata and bootloader upgrade data (if present) stay uncompressed. This means that a compressed GBL file is identical to a normal (uncompressed) GBL file, except that the GBL Programming Tag containing the application upgrade image (as described in *UG103.6: Bootloader Fundamentals*) has been replaced by a GBL LZ4 Compressed Programming Tag or GBL LZMA Compressed Programming Tag. Signature and encryption operations on a compressed GBL work identically to on an uncompressed GBL.

To be able to use compressed upgrade images, a decompressor for the relevant compression algorithm must be added to the Gecko bootloader. The following table shows which compression algorithms are supported by the Gecko Bootloader, and which Bootloader component should be added to enable the feature. The table also shows how much space the decompressor takes up in the bootloader, and how big of a size reduction to expect for the compressed application upgrade image. Be aware of the bootloader size requirement. The bootloader space might be too small to fit the decompressors, depending on the device and enabled components.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZ4	GBL Compression (LZ4)	< 1 kB	~ 10%

Copyright © 2025 Silicon Laboratories. All rights reserved.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZMA	Bootloader Compression (LZMA)	~5 kB flash, 18 kB RAM	~ 30%

It is important to note that the compressed GBL file stays compressed while being transferred to the device, and while it is stored in the upgrade area. It is decompressed by the bootloader when the upgrade is applied. This means that the running application in main flash will be identical to one that was installed using an uncompressed (normal) GBL file.

Compressed GBL files can only be decompressed by the bootloader when running standalone, not through the Application Interface. This means that upgrade image verification performed by the application prior to reboot will not attempt to decompress the application upgrade, it will only verify the signature of the compressed payload. After rebooting into the bootloader, it will decompress the image as part of the upgrade process.

> Note: The above means that Bluetooth in-place application upgrades cannot be compressed, as they are processed by the Bluetooth Supervisor or AppLoader using functionality in the bootloader through the Application Interface. Supervisor/stack and AppLoader updates can be compressed, but the user application cannot.

LZMA Compression Settings

LZMA decompression is only supported for images compressed with certain compression settings. Simplicity Commander automatically uses these settings when using the `commander gbl create --compress lzma` command.

- Probability model counters: lp + lc <= 2. Simplicity Commander uses lp=1, lc=1.
- Dictionary size no greater than 8 kB. Simplicity Commander uses 8 kB.

Together, these settings cause the decompressor to require 18 kB of RAM for decompression – 10 kB for the counters and 8 kB for the dictionary.

The GBL LZMA Compressed Programming Tag contains a full LZMA file, containing the LZMA header, raw stream, and end mark. The Gecko bootloader only supports decompressing payloads that contain the end mark as the last 8 bytes of the compressed stream.

Bootloader Example Configurations

The following sections describe the key configuration options for the example bootloader applications.

> Note: Security features are disabled for all example configurations. In development, Silicon Labs strongly recommends enabling security features to prevent unauthorized parties from uploading untrusted program code. See section Using Application Image Security Features to learn how to configure the security features of the Gecko Bootloader.

UART XMODEM Bootloader

Standalone bootloader for EFM32 and EFR32 devices running the EmberZNet PRO and Silicon Labs Connect protocol stacks, using XMODEM-CRC over UART.

In this configuration, the UART XMODEM communication component, XMODEM Parser component, and Bootloader UART Driver component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the Bootloader UART Driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The GPIO activation component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

BGAPI UART DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using the BGAPI protocol for UART DFU. This bootloader should be used for all NCP-mode Bluetooth applications.

In this configuration, the BGAPI UART DFU communication component and Bootloader UART Driver component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the Bootloader UART Driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation component is also installed by default, allowing bootloader entry by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

EZSP SPI Bootloader

Standalone bootloader for EmberZNet PRO and Silicon Labs Connect protocol stacks using EZSP for SPI.

In this configuration, the EZSP-SPI communication component, XMODEM Parser component, and Bootloader SPI Peripheral Driver component are installed. For the example application to run on a custom board, the GPIO ports and pins used for SPI and EZSP signaling need to be configured in the Bootloader SPI Peripheral Driver and EZSP-SPI components, respectively.

The EZSP GPIO activation component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

Bluetooth AppLoader OTA DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using Bluetooth for Over-The-Air DFU.

In this configuration, the Bluetooth AppLoader communication component is installed.

SPI Flash Storage Bootloader

Application bootloader for all wireless protocol stacks, using an external SPI flash to store upgrade images received over the air by the application.

In this configuration, the SPI Flash Storage and Common Storage components, as well as the Bootloader SPI Controller Driver component, are installed. For the example application to run on a custom board, the GPIO ports and pins used for SPI communication with the external flash need to be configured in the Bootloader SPI Controller Driver component, and the type of SPI flash needs to be configured in the SPI Flash Storage component. The base address of the storage area can be configured in the Common Storage component. The location and size of the storage slots themselves can be configured using the Bootloader Storage Slot Setup component (supports up to three configurable storage slots).

The SPI Flash storage bootloader provides customizable callback functions, which can be found in *bootloader-callback-stubs.c*. The callback functions *storage_customInit* and *storage_customShutdown*, for example, can be used for custom hardware setups. Those callback functions can be interfaced from the applications using the bootloader interface functions such as *bootloader_init* and *bootloader_deinit* from *api/btl_interface.h*.

Currently, Silicon Labs supports the following SPI Flash storage bootloaders:

1. SPI Flash storage bootloader (multiple images)
2. SPI Flash storage bootloader (single image)
3. SPI Flash storage bootloader (single image with slot size of 1MB). Note: This bootloader should only be used for systems with an external SPI Flash size >= 1 MB.

SPI Flash Storage Bootloader SFDP

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application bootloader for all wireless protocol stacks, using an external SPI flash to store upgrade images received over the air by the application.

In this configuration, the SPI Flash Storage SFDP and Common Storage components, as well as the Bootloader SPI Controller Driver component, are installed. For the example application to run on a custom board, the GPIO ports and pins used for SPI communication with the external flash need to be configured in the Bootloader SPI Controller Driver component. The type of SPI flash is detected at runtime. The base address of the storage area can be configured in the Common Storage component. The location and size of the storage slots themselves can be configured using the Bootloader Storage Slot Setup component (supports up to three configurable storage slots).

The SPI Flash storage bootloader SFDP provides customizable callback functions, which can be found in *bootloader-callback-stubs.c*. The callback functions *storage_customInit* and *storage_customShutdown*, for example, can be used for custom hardware setups. Those callback functions can be interfaced from the applications using the bootloader interface functions such as *bootloader_init* and *bootloader_deinit* from *api/btl_interface.h*.

Internal Storage Bootloader

Application bootloader for all wireless protocol stacks, using internal flash to store upgrade images received over the air by the application.

Multiple examples are provided, including configurations for 512 kB flash memory devices like EFR32xG13, 1024 kB flash memory devices like EFR32xG12, and 2048 kB flash memory devices like EFM32GG11. The storage layout should be modified before running the bootloader on any other devices. In this configuration, the internal flash and common storage components are installed. The base address of the storage area is configured in the Common Storage component. The location and size of the storage slots can be configured using the Bootloader Storage Slot Setup component (provides up to three configurable storage slots). Default example applications are provided with configurations for both single storage slot and multiple storage slots.

The default storage slot configurations provided by the Gecko Bootloader must be configured to match the use-case-specific application configurations.

Table: Internal Storage Bootloader Default Storage Configurations

Sample Applications	Storage Offset	Storage Size	Additional Notes
Internal Storage Bootloader (single image on 256 kB device)	0x21800 (137216)	88064	This configuration is available for devices where flash base address is 0x00
'	0x8021800 (134354944)	88064	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 352 kB device)	0x28000 (163840)	147456	This configuration is available for devices where flash base address is 0x00
'	0x8028000 (134381568)	147456	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 512 kB device)	0x44000 (278528)	196608	This configuration is available for devices where flash base address is 0x00
'	0x8044000 (134496256)	196608	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 1 MB device)	0x84000 (540672)	458752	This configuration is available for devices where flash base address is 0x00

Copyright © 2025 Silicon Laboratories. All rights reserved.

Sample Applications	Storage Offset	Storage Size	Additional Notes
'	0x8084000 (134758400)	458752	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 2 MB device)	0x10000 (1048576)	1011712	This configuration is available for devices where flash base address is 0x00
'	0x8100000 (135266304)	1011712	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (multiple images on 1 MB device)	0x5A000 (368640)	337920	This configuration is available for devices where flash base address is 0x00
'	0xAC800 (706560)	337920	This configuration is available for devices where flash base address is 0x00
Internal Storage Bootloader (single image on 1536kB device)	0xC0000 (786432)	737280	This configuration is available for devices where flash base address is 0x00
'	0x80C0000 (135004160)	737280	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 1920kB device)	0x80E8000 (135168000)	966656	This example application is available for device families with flash size of 1920kB or higher (for example xG25 device family)

Image Acquisition Application Example Configuration

These examples illustrate applications that acquire and store a GBL upload image for an application bootloader. For the running bootloader to accept an application upgrade, the new application version must be higher than the existing version. The version number can be set in Appbuilder by defining the macro “APP_PROPERTIES_VERSION” in the Additional Macros field on the Other tab.

Setting a Version Number

To distinguish between different builds of the Gecko Bootloader, it is useful to set a version number. To perform a bootloader upgrade, not only must the running bootloader pass its integrity checks (see section Downloading and Applying a Bootloader GBL Upgrade File), but the bootloader upgrade image must also have a higher version number than the running bootloader image. A version number can be set using Simplicity Studio by configuring the Bootloader Version Main Customer option of the Bootloader Core component. This macro will be picked up by the config file btl_config.h, where it is combined with the version number of the Gecko Bootloader files provided by Silicon Labs.

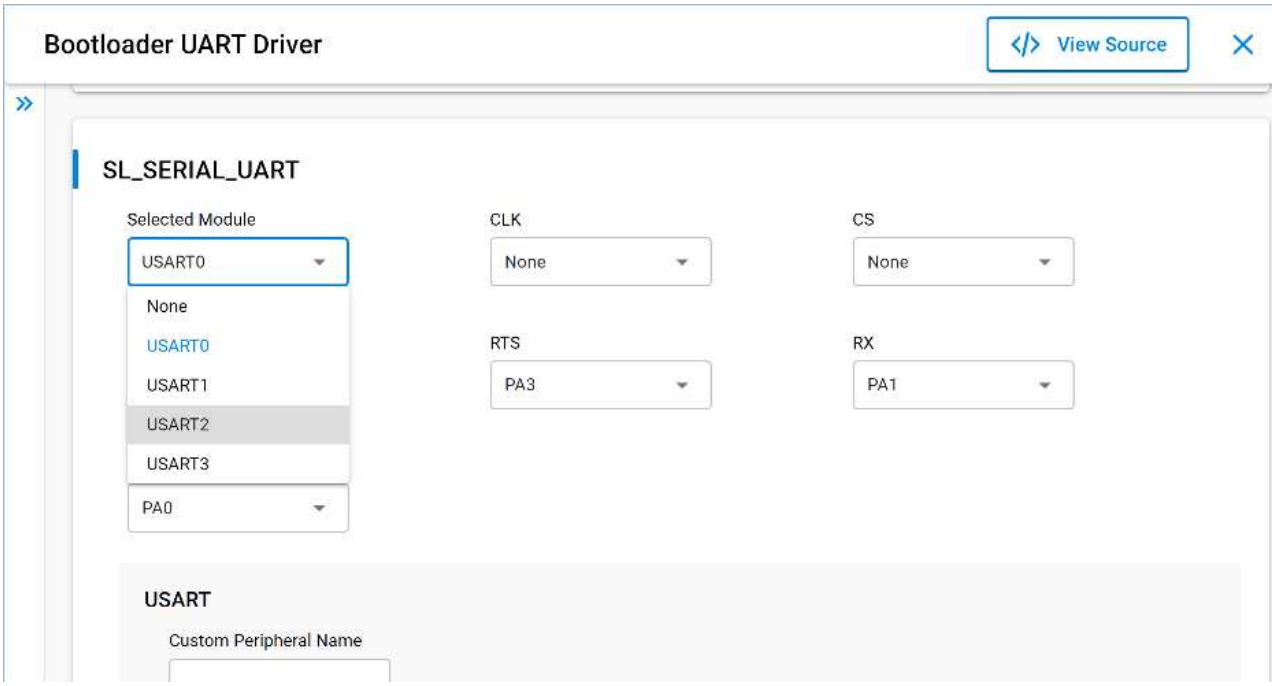
Placing Bootloader in Main Flash

For device families xG13 and xG14, the entire main stage bootloader might not fit into the bootloader flash if the user installs some extra components. In such scenarios, the main stage bootloader can be placed in the main flash by installing the Bootloader in Main Flash core component. This component internally provides a MACRO named MAIN_BOOTLOADER_IN_MAIN_FLASH which is used to set the appropriate address required to place the main bootloader in main flash. The MACRO is used to generate the linker file from the common linker template with the appropriate addresses to place the main stage bootloader in the main flash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hardware Configuration

The Gecko Bootloader uses the Pin Tool for configuration of pinout and other hardware-related settings. When Pin Tool configuration is available for a bootloader component, the relevant settings are shown in the Component Editor for that component.



The standalone Pin Tool User Interface can also be used to configure settings for Gecko Bootloader if desired.

While the Pin Tool provides configuration for many different peripherals, the Gecko Bootloader uses only the following Pin Tool modules:

- SERIAL is used by the UART Driver component to configure baud rate, flow control mode and pinout.
- VCOM is used by the UART Driver component to enable the serial interface if necessary (only on Silicon Labs Wireless Starter Kits).
- EXTFLASH is used by the SPI Driver to configure frequency and pinout.
- SPINCP is used by the SPI Slave Driver to configure pinout.
- BTL_BUTTON is used by the GPIO Activation component.
- CMU HFXO frequency setting is used by the delay driver to calibrate timing if the core is running from the HFXO.

Other settings, like CMU oscillator configuration or DCDC configuration, are not taken into consideration by the default bootloader code. If using these configuration settings is desired, the required code must be added in btl_main.c.

> Note: While the delay driver uses the HFXO frequency setting from Pin Tool, the HFXO enable setting is not used to initialize the HFXO on startup. This setting is only used when calling the bootloader through the Application Interface (see section Gecko Bootloader and TrustZone), and the application has switched to the HFXO prior to calling the Bootloader Application Interface API.

Size Requirements for Different Bootloader Configurations for Series 1 Devices

Enabling different configuration options for the Gecko Bootloader changes the size of the resulting image. The following table shows a list of example bootloader configurations and the resulting approximate size of the main bootloader. Note that any size above 14 kB will be too large to fit in the bootloader area of flash of EFR32xG13, and that any size above 16 kB will be too large to fit in the bootloader area of flash on EFR32xG14 and EFM32TG11. This table was last updated for Gecko SDK Suite 4.0 (Gecko

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader 2.0), and no guarantees are made that a configuration of a specific size with one SDK version will maintain that size in future releases.

Table: Bootloader Size Requirements

Base Configuration	Enabled Options	Size (kB)
XMODEM UART	Default configuration	12.8
'	Secure boot, signed and encrypted upgrade	12.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.7
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.9
Internal storage, single slot	Default configuration	11.5
'	Secure boot, signed and encrypted upgrade	11.5
'	Secure boot, signed and encrypted upgrade, LZ4 compression	12.4
'	Secure boot, signed and encrypted upgrade, LZMA compression	16.6
Internal storage, multiple slots	Default configuration	12.1
'	Secure boot, signed and encrypted upgrade	12.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.1
SPI flash, single slot	Default configuration	12.6
'	Secure boot, signed and encrypted upgrade	12.6
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.5
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.7
SPI flash, multiple slots	Default configuration	13.1
'	Secure boot, signed and encrypted upgrade	13.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	14
'	Secure boot, signed and encrypted upgrade, LZMA compression	18
SPI flash using SFDP, single slot	Default configuration	12.8
'	Secure boot, signed and encrypted upgrade	12.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.8
'	Secure boot, signed and encrypted upgrade, LZMA compression	18
BGAPI UART	Default configuration	11.8
'	Secure boot, signed and encrypted upgrade	11.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	12.7
'	Secure boot, signed and encrypted upgrade, LZMA compression	16.9
EZSP SPI	Default configuration	13.1
'	Secure boot, signed and encrypted upgrade	13.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.9
'	Secure boot, signed and encrypted upgrade, LZMA compression	18.1

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Simplicity Commander and the Gecko Bootloader

Simplicity Commander and the Gecko Bootloader

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface that is also scriptable. You can use Simplicity Commander to perform these essential tasks:

- Generating key files for signing and encryption
- Signing application images for Secure Boot
- Creating GBL images (encrypted or unencrypted, signed, or unsigned)
- Parsing GBL images

Simplicity Commander is used throughout the examples in the following sections. For more information on executing the commands to complete these tasks, see *UG162: Simplicity Commander Reference Guide*.

> Note: Simplicity Commander also offers a GUI (Graphical User Interface) that can be used in the lab for typical tasks such as flashing device images. The functions described in this User Guide are performed from the CLI.

Creating GBL Files Using Simplicity Commander

To create an unsigned GBL file from an application myapp.s37, execute `commander gbl create myapp.gbl --app myapp.s37`.

To create an unsigned GBL file from a main bootloader upgrade mybootloader.s37, execute `commander gbl create mybootloader.gbl --bootloader mybootloader.s37`. This file can be used with the standalone bootloader configurations of the Gecko Bootloader.

To create an unsigned GBL file from a Secure Engine, upgrade mySecureElement.seu, and execute `commander gbl create mySecureElement.gbl --seupgrade mySecureElement.seu`. The Secure Engine images, .seu, are provided by Silicon Labs and can be found through Simplicity Studio. See section Gecko Bootloader Operation - Secure Engine Upgrade.

These commands can also be combined to create a single upgrade image, suitable for use with application bootloader configurations of the Gecko Bootloader: `commander gbl create myupgrade.gbl --app myapp.s37 --bootloader mybootloader.s37 --seupgrade mySecureElement.seu`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Security Features

Gecko Bootloader Security Features

About Bootloader Image Security

Secure Boot and Secure Firmware Upgrade, discussed in the following sections, enable Gecko Bootloader to provide authenticity and integrity checks on the Application image, which provides a sufficient level of security for many applications. However, in systems without a hardware root of trust, no process checks the authenticity or integrity of the Gecko Bootloader itself. Its security is provided solely by the device hardware and the robustness of the software running on the device.

The native behavior of Firmware Upgrade will prevent accidental version rollback of Gecko Bootloader under normal usage conditions. However, without a hardware root of trust, intentional downgrade attacks may be feasible. If a higher level of security assurance is required by the application, using a Series 2 device with hardware root of trust and anti-rollback protection is recommended. For more information, refer to Series 2 and Series 3 Secure Boot with RTSL.

About Application Image Security

The Gecko Bootloader can enforce security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted.

Secure Boot Procedure

When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA (elliptical curve digital signature algorithm) of the SHA-256 digest of the application firmware image, using the NIST P-256 (secp256r1) curve.

Summary of Operation

1. On boot, the bootloader checks the application image for information about whether it is signed.
2. The type of signature and signature location is determined.
3. If the type of signature does not match the requirements of the bootloader, the bootloader enters device firmware upgrade mode and prevents the application from running.
4. According to the chosen signature algorithm, the signature of the contents of flash from the beginning of the application to the location of the signature is compared to the signature at the signature location.
5. If the signatures do not match, the bootloader enters device firmware upgrade mode and prevents the application from running.

Secure Boot using ECDSA-P256-SHA256

For an image to be signed for Secure Boot, the application needs to contain a copy of the ApplicationProperties_t struct. This struct contains information about which signature algorithm is used, and where to find the signature.

On every boot, the bootloader calculates the SHA-256 digest of the application image, from the beginning of the application to the start of the signature. The signature of the SHA-256 digest is then verified using ECDSA-P256.

If the signature is valid, the application is allowed to boot. Else, the bootloader is entered, and an application upgrade is attempted if one is available.

The public key used for signature verification is stored as a manufacturing token on Series 1 and EFR32xG22 devices. OnEFR32xG21 devices, the public key can either be stored as a manufacturing token or stored in the Secure Engine One-time

Copyright © 2025 Silicon Laboratories. All rights reserved.

Programmable memory (OTP). Simplicity Commander can be used to generate a key pair and write the public key to the device. See *UG162: Simplicity Commander Reference Guide* for more information.

Secure Boot with Application Rollback Protection

On every boot, the application version in the ApplicationData_t struct is stored at the end of the bootloader area in flash, which is used to prevent applications from being downgraded. The application version can remain the same for upgrades. The bootloader will only allow applications to increment its version 6 times by default. The bootloader can be upgraded after that to reset the stored application versions and to allow application upgrades again. The allowed number of upgrades can be increased by modifying linkerfiles to reserve an extra 4 bytes for each version and by increasing SL_GBL_APPLICATION_VERSION_STORAGE_CAPACITY from *btl_bootload.c*. For Series 2 devices, this option is not applicable on devices with Secure Engine configured to perform full page lock. If the bootloader area in flash is locked by Secure Engine, the bootloader will not be able to store the application versions in flash, and it would continue without performing that function.

When the application upgrade version check option is enabled in the Bootloader Core component, the bootloader checks where App properties points into flash. If app properties struct is present, then it will check compatibility of the application properties struct and check if the upgrade version is greater than the installed app; otherwise, it will check if product ID is correct before applying the upgrade image.

If app properties struct is absent at the app properties location, it is assumed that the previous upgrade was interrupted, and allows the upgrade to continue.

If application rollback prevention component is installed in the bootloader project, then before applying the upgrade image, the bootloader will check the application version in the application properties structure that resides inside the signed/encrypted GBL file and will only apply the OTA image to application area if the application version in ApplicationData structure is equal or higher than the highest application version last seen.

The application rollback prevention feature can be enabled in the Bootloader Core component by selecting the Enable application rollback protection option. The Minimum application version allowed option can be used to configure the minimum application version that should be allowed to boot.

The application versions are stored in the bootloader area in flash. To properly protect the stored application versions, it is recommended to lock the bootloader flash by selecting the Prevent bootloader write/erase option in the Bootloader Core component.

Secure Boot Using a Certificate

On Series 2 devices, a certificate-based secure boot operation is supported. The Certificate contains:

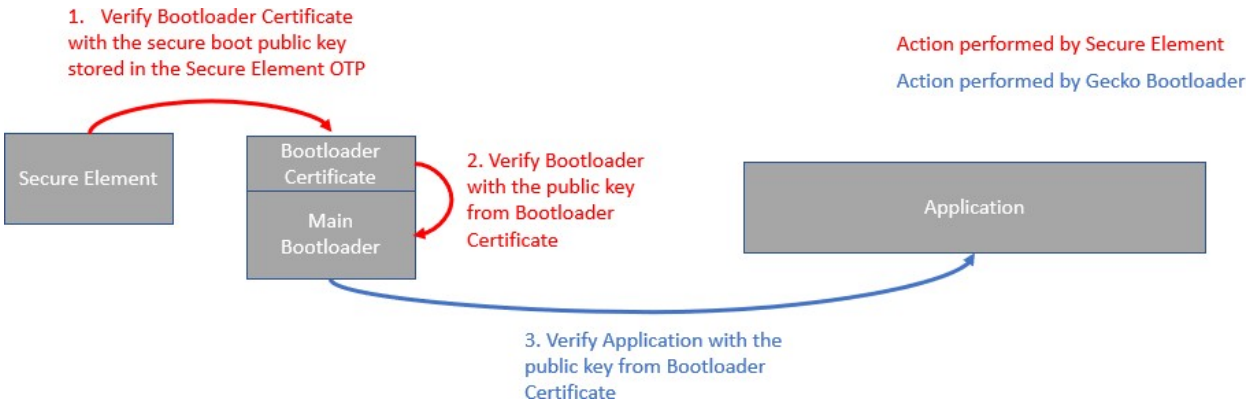
- Struct version: The version of the certificate structure.
- Public key: ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
- Certificate version: The version of the running certificate.
- Signature: ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

The definition of the certificate struct can be found in api/application_properties.h.

To utilize certificate-based secure boot, configure Secure Engine to authenticate the bootloader image by configuring the certificate- based secure boot option in the Secure Engine OTP. Configure the Gecko Bootloader to enable certificate-based secure boot in the Bootloader Core component by selecting the Enable certificate support option. The Gecko Bootloader certificate must be signed by the private key pair of the public key stored in the Secure Engine OTP. For more information on the key storage, see section Key Storage.

The certificate-based secure boot procedure is illustrated in the following figure.

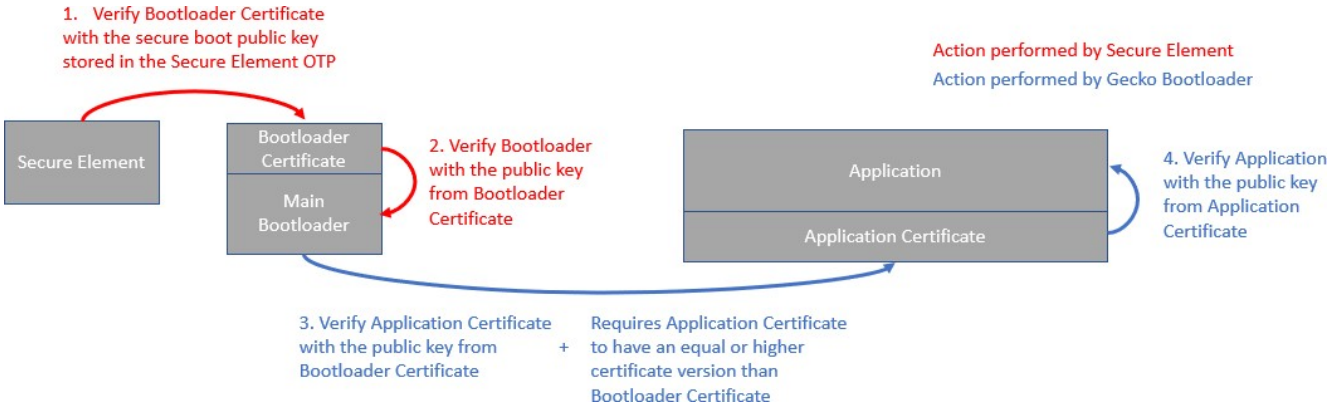
Copyright © 2025 Silicon Laboratories. All rights reserved.



Once the certificate-based secure boot option on Secure Engine is turned on, Secure Engine verifies the Gecko Bootloader certificate. The public key stored in the certificate is used to validate the signature of the Gecko Bootloader. Secure Engine will not accept bootloader images without a certificate.

If only the secure boot option is enabled (not certificate-based) on Secure Engine, and Secure Engine identifies a certificate, the certificate will be used to validate the bootloader image. If the certificate version from the bootloader image is higher than 0 and it gets verified once, Secure Engine will never again accept direct signed bootloader images without a certificate.

The Gecko Bootloader will authenticate the direct signed application using the public key stored in the Gecko Bootloader certificate. If the application contains a certificate, Gecko Bootloader will authenticate it. The procedure is illustrated in the following figure.



After authentication of the application certificate, Gecko Bootloader verifies the signature of the application using the public key from the application certificate. In addition, Gecko Bootloader compares the Gecko Bootloader certificate version against the application certificate version. All application images with a certificate version lower than the certificate version of the Gecko Bootloader will be rejected. Gecko Bootloader can be configured to only allow applications with certificate to boot by configuring the Bootloader Core component by selecting the Reject direct signed images option.

The ApplicationProperties_t struct contains the certificate struct ApplicationCertificate_t. The certificate struct can be injected to images that contain an ApplicationProperties_t with ApplicationCertificate_t. To inject a certificate to an image, issue the following command from Simplicity Commander:

```
commander convert <image file> --secureboot --keyfile <signing key> --certificate <certificate>--outfile
<signed image file with certificate>
```

Secure Firmware Upgrade

The Gecko Bootloader supports a secure firmware upgrade process. This is achieved by using symmetric encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image. Symmetric encryption provides confidentiality and asymmetric cryptography provides integrity and authenticity. Note that encryption alone is not enough to provide authenticity.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Encryption Algorithms

The Gecko Bootloader supports the AES-CTR-128 encryption algorithm. The GBL upgrade file is encrypted using 128-bit AES in Counter mode with a random nonce as the initial counter value. Through configuration, the GBL file's decryption duration can be improved by increasing the counter's block size. The block size is configurable with options 1, 2, 4, or 8 blocks. The configuration is available as part of the AES CTR Stream Block Config component.

The AES-128 key used for decryption, more commonly called the OTA decryption key, is stored as a manufacturing token on Series 1 and EFR32xG22 devices. On EFR32xG21 devices, the OTA decryption key can either be stored as a manufacturing token or stored in the Secure Engine OTP. To make use of the OTA decryption key stored in the Secure Engine OTP, the Use symmetric key stored in Secure Engine storage option in the Bootloader Core component must be selected. Simplicity Commander can be used to generate an OTA decryption key and write the key to the device. See *UG162: Simplicity Commander Reference Guide* for more information. For more information on storing the OTA decryption key on Series 2 devices, see Production Programming of Series 2 and Series 3 Devices.

The Secure Engine OTP key support depends on the SE Manager component, which is enabled by default.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA signature of the SHA-256 digest of the GBL upgrade file, using the NIST P-256 (secp256r1) curve.

Summary of Operation

Before starting a firmware upgrade process, the application can verify an image in storage by calling into the bootloader verification functions. For bootloaders with Communication Interface, the host device should verify the image before sending it to the NCP or RCP.

During firmware upgrade, the GBL file is parsed, and if encrypted, decrypted on-the-fly. A GBL Signature Tag in the GBL file indicates to the bootloader that the file is signed, and the signature is verified. If signature verification fails, the firmware upgrade process is aborted.

On Series 2 devices, Gecko Bootloader will authenticate the GBL signature tag using the public key stored in the bootloader certificate if the Enable certificate support option is selected in the Bootloader Core component. A GBL Certificate tag in the GBL file indicates to the bootloader that the GBL certificate tag needs to be authenticated using the public key stored in the bootloader certificate. The certificate version in the GBL certificate tag is compared with the bootloader certificate and only a version equal or higher than the bootloader certificate is accepted. Once the GBL certificate tag is authenticated, the GBL file's signature is verified using the authenticated public key from the GBL certificate tag. On EFRxG21 devices, the GBL certificate tag can also be signed by the private key pair of the public key stored in the Secure Engine OTP.

Using Application Image Security Features

This example assumes that a bootloader called bootloader-uart-xmodem has been built in Simplicity Studio. For a Series 1 device, three files of interest have been generated in the output directory:

- bootloader-uart-xmodem.s37 – This file contains the main bootloader. Can be used for bootloader upgrade.
- bootloader-uart-xmodem-crc.s37 - This file contains the main bootloader with CRC32 checksum.
- bootloader-uart-xmodem-combined.s37 – This file contains the first stage and main bootloader with a CRC32 checksum in a single image. Can be used for manufacturing and initial deployment of the bootloader.

The relevant version can be flashed to the EFR32 using the Flash Programmer in Simplicity Studio or using Simplicity Commander. For Series 2 devices, only the main bootloader binary is generated.

This example provides two ways of signing the upgrade images. The first option uses Simplicity Commander to generate key material and sign data. This is suitable for development. The second option uses an external signer, such as a dedicated Hardware Security Module (HSM) to protect private key material and perform signing operations. Silicon Labs recommends using an HSM to safeguard private keys.

Generating Keys

Copyright © 2025 Silicon Laboratories. All rights reserved.

To use the security features of the Gecko Bootloader, encryption and signing keys need to be generated. These keys must then be written to the EFR32 device. The encryption key is used with the GBL file for secure firmware upgrade. The signing keys are used both with the GBL file for secure firmware upgrade and to sign the application image for Secure Boot.

Generating a Signing Key Using Simplicity Commander

```
commander util genkey --type ecc-p256 --privkey signing-key --pubkey signing-key.pub --tokenfile signing-key-tokens.txt
```

This creates an ECDSA-P256 key pair for signing; signing-key contains the private key in PEM format and must be kept secret from third parties. This key will later be used to sign images and GBL files. signing-key.pub contains the public key in PEM format and can be used to verify GBL files using commander gbl parse. signing-key-tokens.txt contains the public key in token format, suitable for writing to the EFR32 device.

Generating a Signing Key Using a Hardware Security Module

When using a Hardware Security Module, the private key is kept secret inside the HSM. According to the instructions from your HSM vendor, have it generate an ECDSA-P256 key pair and export the public key in PEM format to the file signing-key.pub. Then use Simplicity Commander to convert the key to token format, suitable for writing to the EFR32 device.

```
commander gbl keyconvert --type ecc-p256 signing-key.pub --outfile signing-key-tokens.txt
```

Generating an Encryption Key

```
commander util genkey --type aes-ccm --outfile encryption-key
```

This creates an AES-128 key for encryption in the file encryption-key. The file has token format, making it suitable to write to the EFR32 device using `commander flash --tokenfile`.

Writing Keys to the Device

To write the two token files containing the encryption key and public key as manufacturing tokens to the device, issue the following command:

```
commander flash --tokengroup znet --tokenfile encryption-key --tokenfile signing-key-tokens.txt
```

Signing an Application Image for Secure Boot

If the bootloader enforces Secure Boot, the application needs to be signed to pass verification. On every boot, a SHA-256 digest of the application is calculated. The signature is verified using ECDSA-P256, with the same public key as for the GBL file signing. Signature verification failure prevents the application from booting.

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. If ApplicationProperties_t is missing, the application image cannot be signed. For more details on adding ApplicationProperties_t, see section Application Properties.

Using Simplicity Commander

Signing the application can be done with the command:

```
commander convert myapp.s37 --secureboot --keyfile signing-key --outfile myapp-signed.s37
```

Using a Hardware Security Module

The application can be prepared for signing by issuing the command:

```
commander convert myapp.s37 --secureboot --extsign --outfile myapp-for-signing.s37
```

Creating a Signed and Encrypted GBL Upgrade Image File from an Application

To create a GBL file from an application, use `commander gbl create`.

Note that, as of this writing, secure application images can only be constructed through Simplicity Commander, not through the configuration options available through Simplicity Studio.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. If ApplicationProperties_t is missing, the application image cannot be signed. For more details on adding ApplicationProperties_t, see section Application Properties.

Using Simplicity Commander to Sign

For an application called myapp.s37, use:

```
commander gbl create myapp.gbl --app myapp.s37 --sign signing-key --encrypt  encryption-key
```

This single command performs three actions:

- Creates a GBL file
- Encrypts the GBL file
- Signs the GBL file

If Secure Boot is also desired, the application must be signed using `commander convert --secureboot` prior to creating the GBL.

Using a Hardware Security Module to Sign

For an application called myapp-signed.s37, which has previously been signed for Secure Boot, use:

```
commander gbl create myapp-for-signing.gbl --app myapp-signed.s37 --extsign --encrypt  encryption-key
```

This command performs the following actions:

- Creates a partial GBL file
- Encrypts the partial GBL file
- Prepares the partial GBL file for signing by an external signer

Using an HSM, sign the output file myapp-for-signing.gbl.extsign, and supply the resulting DER-formatted signature file signature.der back to Simplicity Commander:

```
commander gbl sign myapp-for-signing.gbl.extsign --signature signature.der --verify signing-key.pub --outfile myapp.gbl
```

The GBL file is not valid until the signature has been applied using gbl sign.

System Security Considerations

The Gecko bootloader security features can be used to create a secure device, but do not create a secure system by themselves. This section goes over considerations that need to be taken when designing a secure system where the Gecko Bootloader is a component.

Key Storage

On Series 1 devices, the decryption and public sign keys used by the Gecko bootloader are stored in the Lockbits page in flash. To prevent a rogue application from being able to change or wipe the keys, the Lockbits page should be write protected after the keys have been written in manufacturing.

On Series 2 devices, the decryption key and the sign key used by the Gecko Bootloader can either be stored in the topmost page of the main flash or in the Secure Engine OTP. The decryption key can be provisioned in the Secure Engine OTP using Simplicity Commander or using the Secure Engine Mailbox interface. The keys are not accessible from the user applications on EFR32xG22 devices, which means that the keys used by Gecko Bootloader need to be stored in the topmost page of the main flash. Once a key value has been programmed into the Secure Engine OTP, it cannot be changed. For more details, refer to Series 2 and Series 3 Secure Boot with RTSL and Simplicity Commander Reference Guide.

The default behavior of the Gecko Bootloader is to use the sign key stored in the Secure Engine OTP to perform the signature verification. If the sign key is not provisioned, the Gecko Bootloader will try to use the public sign key stored in the topmost page of the main flash. This fallback mechanism can be disabled by disabling the “Allow use of public key from manufacturing token storage” component option from the Bootloader Core component in Simplicity Studio.

Write-Protecting the Bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

By default, the region in flash used by the Gecko bootloader is readable and writeable. The region needs to stay readable for the Application Interface to be able to interact with the bootloader. Immediately after reset, the region also needs to be writable to be able to perform bootloader upgrades.

Series 1

It is possible to write-protect the bootloader region on EFR32xG12 and newer. This is done by setting the write-once MSC_BOOTLOADERCTRL_BLWDIS bit on every bootup on Series 1 devices. The pages are write-protected until the next reboot. This is done by the Gecko bootloader main stage if the “Prevent bootloader write/erase” component option is enabled in the Bootloader Core component in Simplicity Studio.

On EFR32xG1, where the bootloader resides in main flash rather than the information block, the BLWDIS option does not exist. On EFR32xG1, the first flash page containing the first stage bootloader can be write-protected with a Page Lock word, using commander `device protect --write --range 0x0:+0x800`. If bootloader upgrades are to be supported, the pages containing the main bootloader need to stay writeable.

> Note: Setting MSC_BOOTLOADERCTRL_BLWDIS will also prevent debuggers from flashing a new bootloader. This means that debug tools that do not completely halt and reset the target device before re-flashing might fail to flash the new bootloader, as the flash is write-protected. If you are unsure whether your debug tools will handle this gracefully, Silicon Labs recommends keeping this setting disabled during development, and enabling it before going into production. If your debug tools halt and reset the target device before flashing, this is not an issue, and Silicon Labs recommends enabling this setting early in the development cycle.

Series 2

On Series 2 devices, the write-once MSC_PAGELOCKx register is used to write-protect the pages used by the bootloader if the “Prevent bootloader write/erase” component option is enabled in the Bootloader Core component in Simplicity Studio. The pages are write-protected until the next reboot.

Write-Protecting the Application

On Series 2 devices, it is also possible to write-protect the application. When the component option “Prevent write/erase of verified application” is enabled, the bootloader will write-protect the pages used by the application after successfully verifying the application signature, before allowing the application to start. This option is only available when Secure Boot of the application is enabled. The MSC_PAGE- LOCKx register is used to protect the pages, and the write protection lasts until the next reboot.

Debug Access

Series 1

To prevent debugger access to the Series 1 devices, the Debug Lock word needs to be written. Device recovery after enabling the Debug Lock is possible but will erase the main flash and the Lockbits page. This will erase the main application and the key material stored in the Lockbits page. The Userdata page and bootloader area will survive Debug Unlock, so secrets should not be stored in these locations. To debug lock the device, issue `commander device lock --debug enable`. The AAP Lock can be used to permanently lock the debug port. This also prevents Debug Unlock. To AAP lock the device, please see the reference manual for your device for the address location of the AAP lock word, and use `commander flash --patch` to write the appropriate value to this address.

Series 2

On Series 2 devices, debugger access can be prevented through Secure Engine. Debugger access can be re-enabled after enabling the Debug Lock by issuing device erase. To debug-unlock the device, issue commander device lock --debug disable from Simplicity Commander. This also erases the main flash, which results in the top page of main flash that stores the encryption and signing keys being erased. To permanently lock the debug port, the device erase disable command can first be issued through

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Engine. Thereafter, the debug lock command can be issued to the Secure Engine. For more details, refer to Series 2 and Series 3 Secure Debug.

Disabling Debug Access from Software

The software APIs `DBG_DisableDebugAccess(dbgLockModeAllowErase)` and `DBG_DisableDebugAccess(dbgLockModePermanent)` from the EMLIB DBG module can be used from the application to lock the debug interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader and Trustzone

Gecko Bootloader and TrustZone

With GSDK v4.2.2, Gecko Bootloader comes with TrustZone support to build TrustZone aware solutions. This section outlines the major points that should be considered when using a TrustZone aware bootloader.

Gecko Bootloader Operation

A TrustZone aware application can have the following configurations:

- Secure app only
- Non-secure app only
- Secure and Non-secure pair

With TrustZone enabled, the application bootloaders are entirely configured as Secure applications whereas the communication bootloaders are split into Secure and Non-secure application pairs. The communication interfaces are typically in the Non-secure part of the application and the GBL parser, and other functionalities are in the Secure part of the application. TrustZone aware applications are generated and built using pre-defined workspace applications. Refer to Series 2 TrustZone and Software Project Generation and Configuration with SLC-CLI for more details regarding TrustZone workspaces and the SLC CLI tool.

For the communication bootloaders to function properly, a combined bootloader binary image (containing both the Secure and Non-secure binary images) needs to be programmed onto the device. A combined bootloader binary can be obtained by either using the post-build steps defined for the workspace or using the Commander tool. Refer to *UG162: Simplicity Commander Reference Guide* for more details on how to use the Commander tool to combine multiple binaries into a single binary. Application bootloaders need to be built with the Bootloader TrustZone Secure component when building a TrustZone aware solution. The Bootloader TrustZone Secure component is required to set up the necessary configuration for the application to function correctly within a TrustZone aware solution.

To create a bootloader upgrade file, the combined bootloader binary (Secure and Non-secure) should be used. Refer to Series 2 TrustZone for more details on upgrading an existing application to TrustZone. Special care must be taken while building the TrustZone aware communication bootloaders for the xG21 family as the bootloader size increases from 16kB (non TrustZone aware solution) to 24kB (TrustZone aware solution). Similarly, the Bootloader – SoC Bluetooth Apploader OTA DFU application's size increases from 72kB to 80kB with TrustZone enabled.

Gecko Bootloader Configuration

The following table provides three of the configuration options available as part of the Bootloader Interface with TrustZone enabled.

Configuration Option	Description
BOOTLOADER_DISABLE_OLD_BOOTLOADER_MITIGATION	Disables multi-tiered fallback logic. The fault handling logic as well as the USART auto-detection logic will be disabled. Default value is set to 0.
BOOTLOADER_DISABLE_NVM3_FAULT_HANDLING	Disables peripheral access fault handling. The fault handling triggered by an erroneous access of peripherals will be disabled. This should be disabled if all the peripherals that are in use by the bootloader have been properly configured by the Manually override security state of peripherals option. Default value is set to 0.
BOOTLOADER_MANUAL_OVERRIDE_SECURITY_STATE	Manually override the security state of peripherals in use by the bootloader. This option can be used to manually set a peripheral's access state before calling into the bootloader. Default value is set to 0.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader and Delta DFU Upgrades

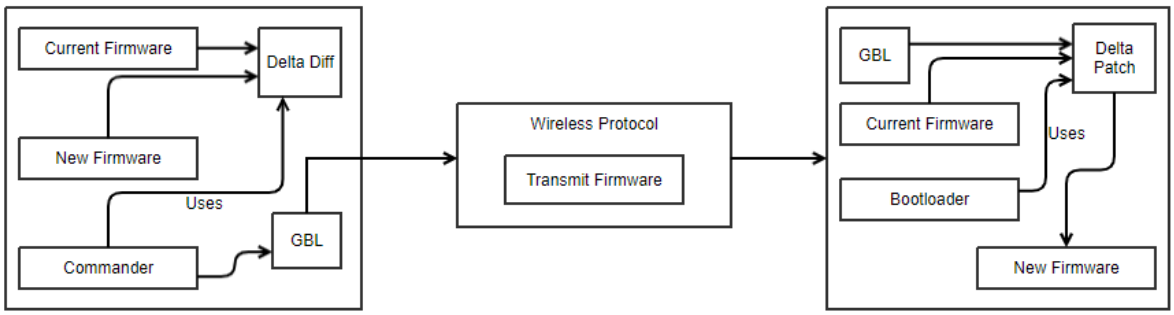
Gecko Bootloader and Delta DFU Upgrades

With GSDK v2024.6.1, Gecko Bootloader support comes with support for delta DFU upgrades. This section provides an introduction to Delta DFU upgrades and describes the major points that should be considered while using the Delta DFU component.

Introduction to Delta DFU

The main function of the image delta DFU is to calculate the difference between the current firmware stored on the device and the new firmware to be updated. This difference is used to create a patch file that can be applied to the current firmware, resulting in the new firmware. The patch file is created using the Simplicity Commander tool, transferred to the device with any wireless protocol as a GBL, and applied on top of the current firmware by the Gecko Bootloader. The patch file creation and application are both implemented in a library that is shared between these tools.

The following diagram illustrates the complete process.



Gecko Bootloader Operation

The Delta DFU upgrades can be enabled by including the Delta DFU component in the Gecko Bootloader. Simplicity Commander supports the creation of a Delta DFU GBL file.

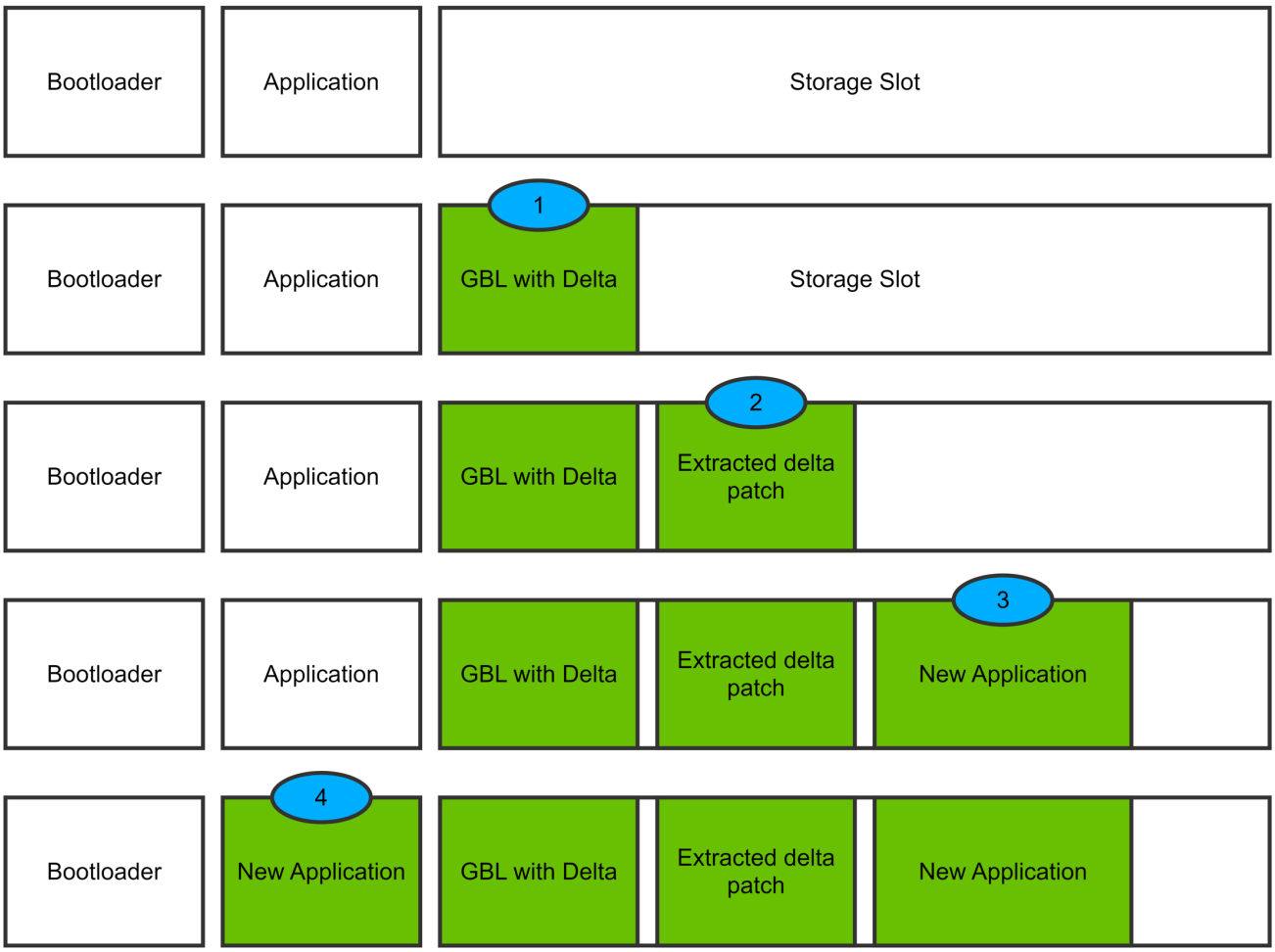
With Delta DFU enabled, the application bootloaders are equipped to accept Delta DFU GBL files. These GBL files contain a patch file that is extracted into the slot. The Gecko Bootloader creates the new firmware using this patch file and the currently running firmware within the slot. The slot size should have enough number of pages to hold the GBL file, patch file, and the new firmware binary in the slot. Each file starts on a new page.

The Delta DFU upgrades are triggered in the same way as in an application bootloader. Once the bootloader finds a GBL that contains Delta Patch, it extracts the patch file and then re-constructs the new firmware file within the slot. If the new firmware is created correctly, then the bootloader copies this new firmware to the application area, completing the upgrade.

To create the GBL file, Commander needs two input files, one of the firmware that is current and one that is the upgrade. For Delta DFU upgrades to work, it is essential that the firmware on the device and the firmware used to create the GBL are the same. Also, the version dependency tag is included in Delta DFU and should be correctly set by the user.

Delta DFU operation is illustrated in the following figure:

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Interface

Application Interface

The bootloader has an application interface exposed through a function table in the bootloader. The application interface provides APIs to use bootloader functions for storing and retrieving upgrade images and verifying their integrity. APIs to reboot into the bootloader are also provided. For details see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

If you are not using a protocol stack from Silicon Labs, the api/btl_interface.h header provides the bootloader application interface API. If you are using a protocol stack from Silicon Labs, the recommended bootloader interface API for the specific protocol stack should be used instead. The following files provide the implementation of the bootloader interface:

api/btl_interface.c (common interface)

api/btl_interface_storage.c (interface to storage functionality)

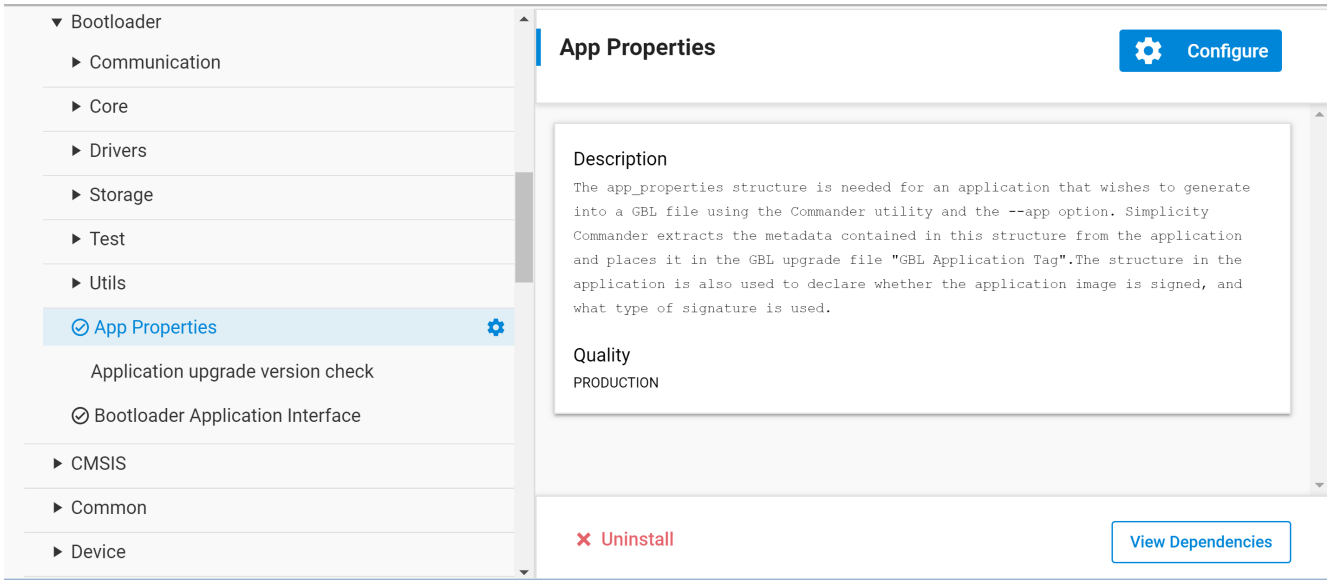
The application interface consists of functions that can be included into the customer application, and that communicate with the bootloader through the MainBootloaderTable_t. This table contains function pointers into the bootloader. The 10th word of the bootloader contains a pointer to this structure, allowing any application to easily locate it. Using the wrapper functions provided in the Bootloader Interface API is preferred over accessing the bootloader table directly. Modules include:

- Application Parser Interface: Application interface for interfacing with the bootloader image parser.
- Application Storage Interface: Application interface for interfacing with the bootloader storage. The Storage Interface is only available on bootloaders that support the storage interface.
- Common Application Interface: Generic application interface available on all versions of the bootloader, independently of which components are present.

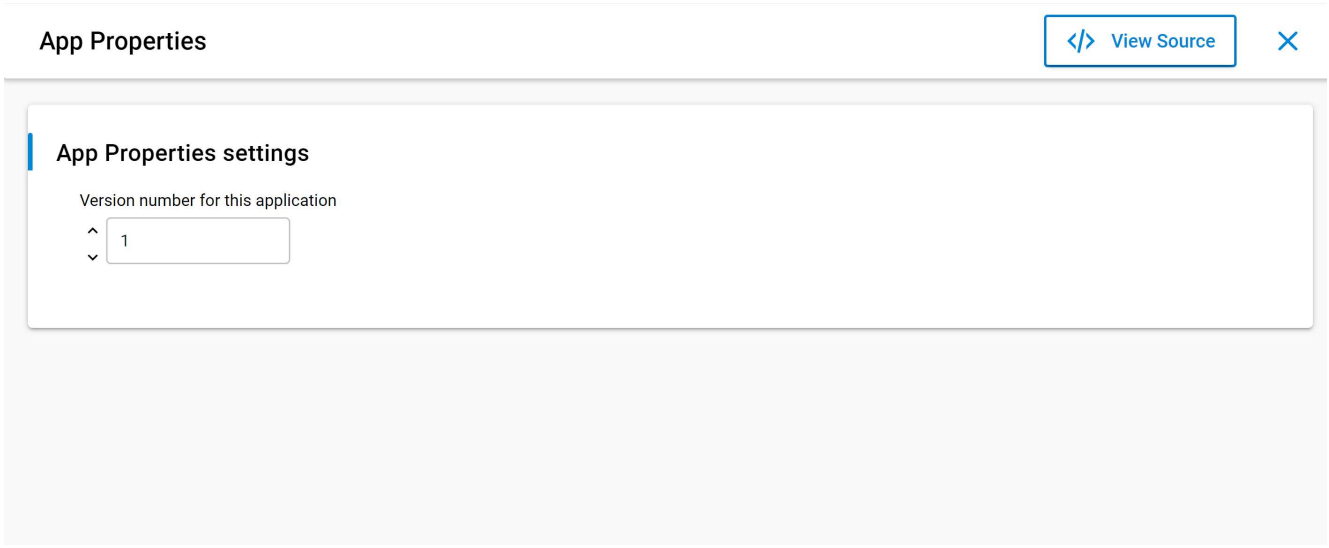
Application Properties

Application images should contain an ApplicationProperties_t struct declaring the application version, capabilities, and other metadata. Simplicity Commander extracts the metadata contained in this structure from the application and places it in the GBL upgrade file GBL Application Tag. If the structure is not present in the application, Simplicity Commander will raise an error. The ApplicationProperties_t struct is added to the application on installing the bootloader_interface component to the application. The bootloader_interface component installs bootloader_app_properties component which adds an instance of ApplicationProperties_t named sl_app_properties to the project. The component adds a source file named app_properties.c and a configuration file named app_properties_config.h. This configuration file allows users to configure the application version via Simplicity Studio's Component Editor. To open the Component Editor, locate the App Properties component under Platform > Bootloader as shown below.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Click Configure to open the Component Editor.



The application type is automatically populated based on the wireless stack used in building the project. The value of the application type is automatically set during the project generation step and can be found in autogen/sl_application_type.h file.

The contents of the GBL Application Tag can be extracted from a GBL file by a running application using the Application Storage interface. Note that the GBL Application Tag will only be added if the GBL file contains an application image, not if the GBL file only contains a bootloader upgrade or metadata.

The structure in the application is also used to declare whether the application image is signed, and what type of signature is used. This information is added by Simplicity Commander when signing the image using `commander convert (--secureboot, --extsign or -- signature)`. For the bootloader to locate the ApplicationProperties_t struct, if not already done by the linker, Simplicity Commander modifies word 13 of the application to insert a pointer to the ApplicationProperties_t struct when signing the application image for Secure Boot.

Error Codes

Most Gecko bootloader APIs return error codes. The following table lists the groups of error codes that may be returned. The full list of error codes within each group can be found in *api/btl_errorcode.h* in the platform/bootloader directory of the SDK, as well as

Copyright © 2025 Silicon Laboratories. All rights reserved.

in the API Reference.

ID	Description
0x0	OK
0x01xx	Initialization error
0x02xx	Image verification error
0x04xx	Storage error
0x05xx	Bootload error
0x06xx	Security error
0x07xx	Communication error
0x09xx	XMODEM parser error
0x10xx	GBL file parser error
0x11xx	SPI slave driver error
0x12xx	UART driver error
0x13xx	Compression error

Copyright © 2025 Silicon Laboratories. All rights reserved.

Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher

Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher

NOTE: This section replaces *AN1326: Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher*. Further updates to this application note will be provided here

Gecko Bootloader v2.x, introduced in GSDK 4.0, contains a number of changes compared to Gecko Bootloader v1.x. Many of these changes are due to an underlying framework redesign that results in an improved developer experience in Simplicity Studio 5 as well as enhanced compatibility within the GSDK. This document describes the differences between the versions, including how to configure the new Gecko Bootloader within Simplicity Studio 5.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

Gecko Software Development Kit suite (GSDK) version 3 (GSDK v3.0) introduced a new underlying Gecko Platform architecture based on components. Beginning with GSDK 4.0, the Gecko Bootloader now uses this underlying architecture. With Simplicity Studio 5 (SSv5) and GSDK 4.0, developers working with Gecko Bootloader will benefit from the following component-based project configuration features:

- Search and filter to find and discover software components that work with the target device
- Automatically pull in all component dependencies and initialization code
- Configurable software components including peripheral inits, drivers, middleware, and stacks
- All configuration settings are in C header files for usage outside of Simplicity Studio
- Configuration validation to alert developers to errors or issues
- Easily manage all project source via git or other SCM tools
- Managed migration to future component and SDK versions
- Simplified transitions from Silicon Labs development kits to custom hardware

Other features of the SSv5/GSDK 4.x development environment include:

- Project source management options (link to SDK sources or copy all contents to user folder)
- Graphical pin configuration through the Pin Tool
- Redesigned Radio Configurator with a fresh UI that's more intuitive for single- and multi-PHY customization
- Iterative development (configure components, edit sources, compile, debug) using SSv5 configuration tools and third-party IDEs
- GNU makefiles as a build option

This document summarizes the differences between the Gecko Bootloader v2.x in GSDK 4.0 and earlier AppBuilder-based versions. These differences include:

- Differences between AppBuilder and the new component-based Project Configurator
- Comparison of the new Project Configurator components with AppBuilder plugins
- Features of the Pin Tool, now used instead of Hardware Configurator
- Other differences including:
  - Linker file
  - Additional Macros
  - Postbuild steps
  - Main bootloader in main flash
  - Callbacks
  - Storage Slots
  - App Properties

Copyright © 2025 Silicon Laboratories. All rights reserved.

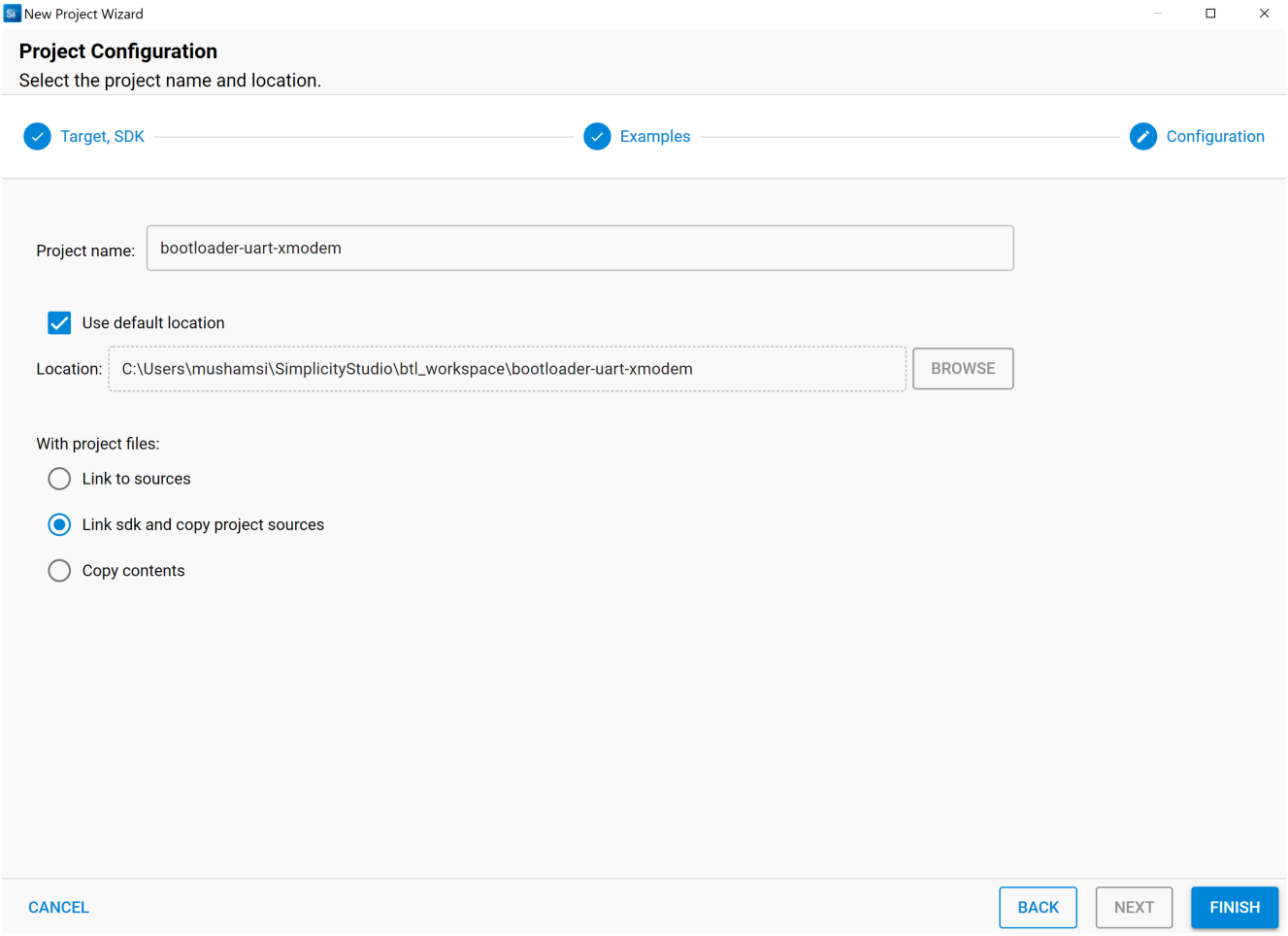
About Projects

About Projects

Creating a Project

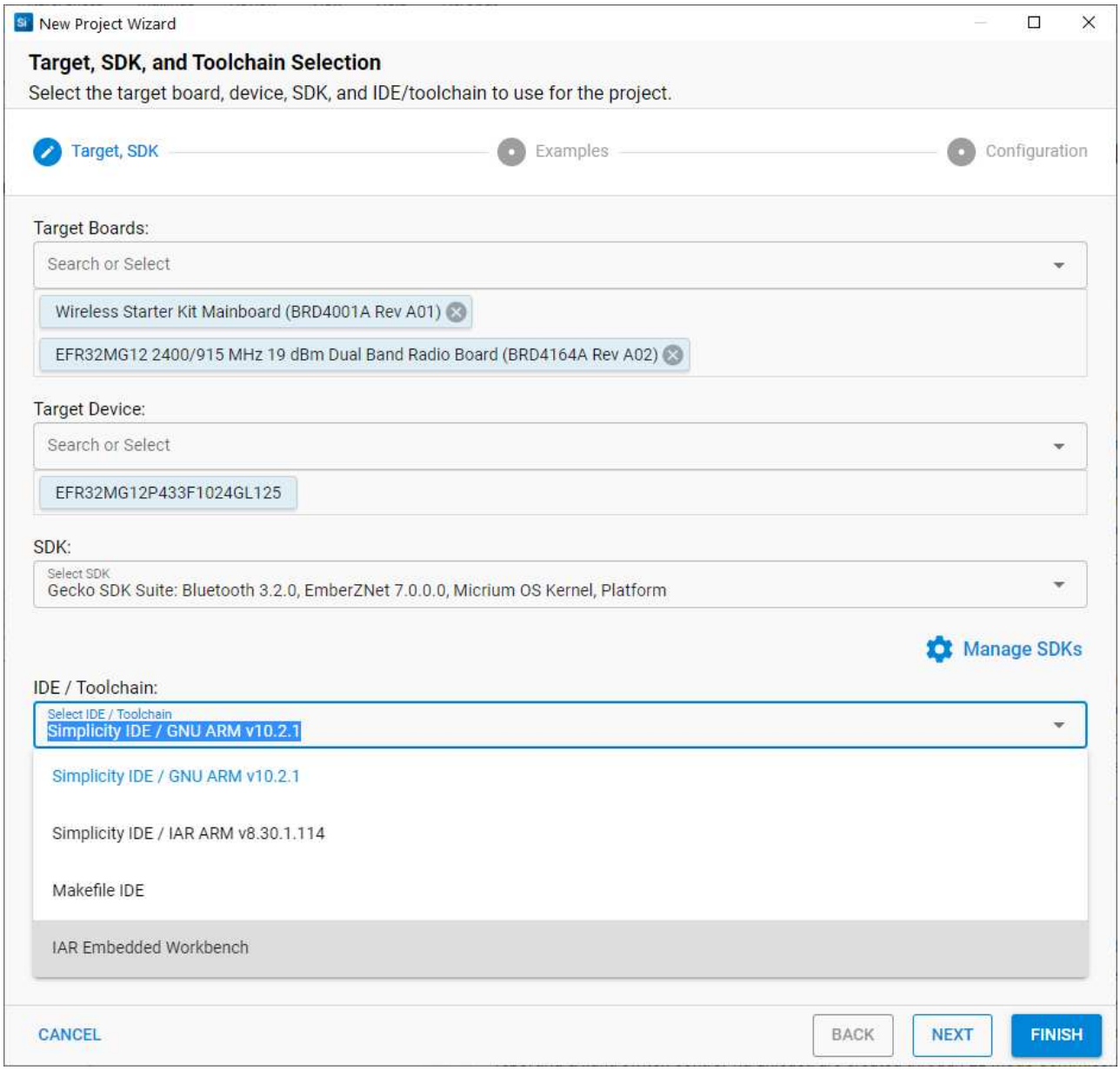
If you have used earlier versions of SSv5, initial project creation is unchanged. If you are unfamiliar with project creation in SSv5, see the Simplicity Studio 5 User's Guide. This section highlights the differences with AppBuilder project creation and configuration.

Once you have selected an example project, an initial configuration dialog opens. Note that the Project Location, which in AppBuilder is shown on the General tab, is defined here.



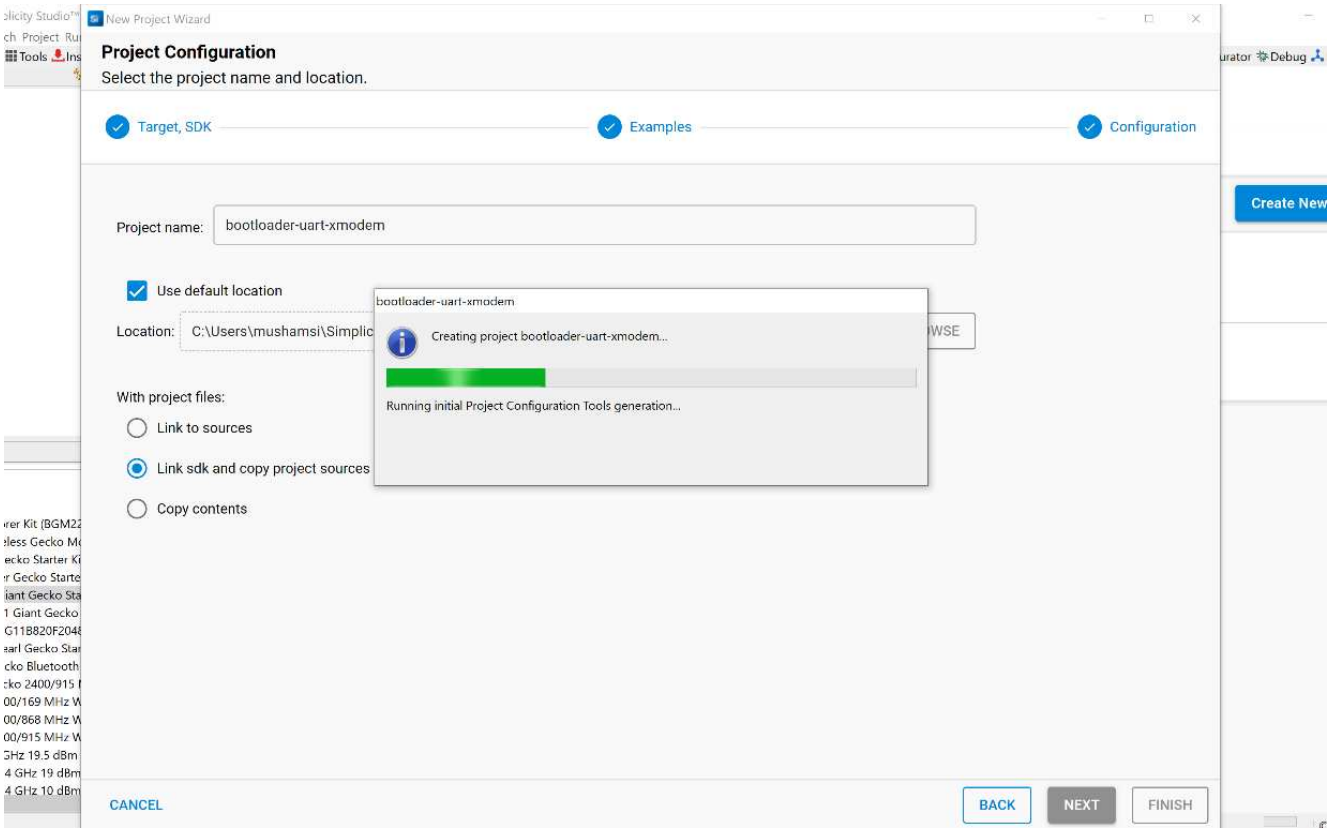
In SSv5, the compiler to use is specified when you create a project and, after that, is difficult to change (see Simplicity Studio 5 User's Guide Tips & Tricks for a procedure to convert an existing project from GCC to IAR). If you do not want to use the default compiler (GCC unless otherwise specified), click BACK, and change it on the first Project Configuration dialog.

Copyright © 2025 Silicon Laboratories. All rights reserved.



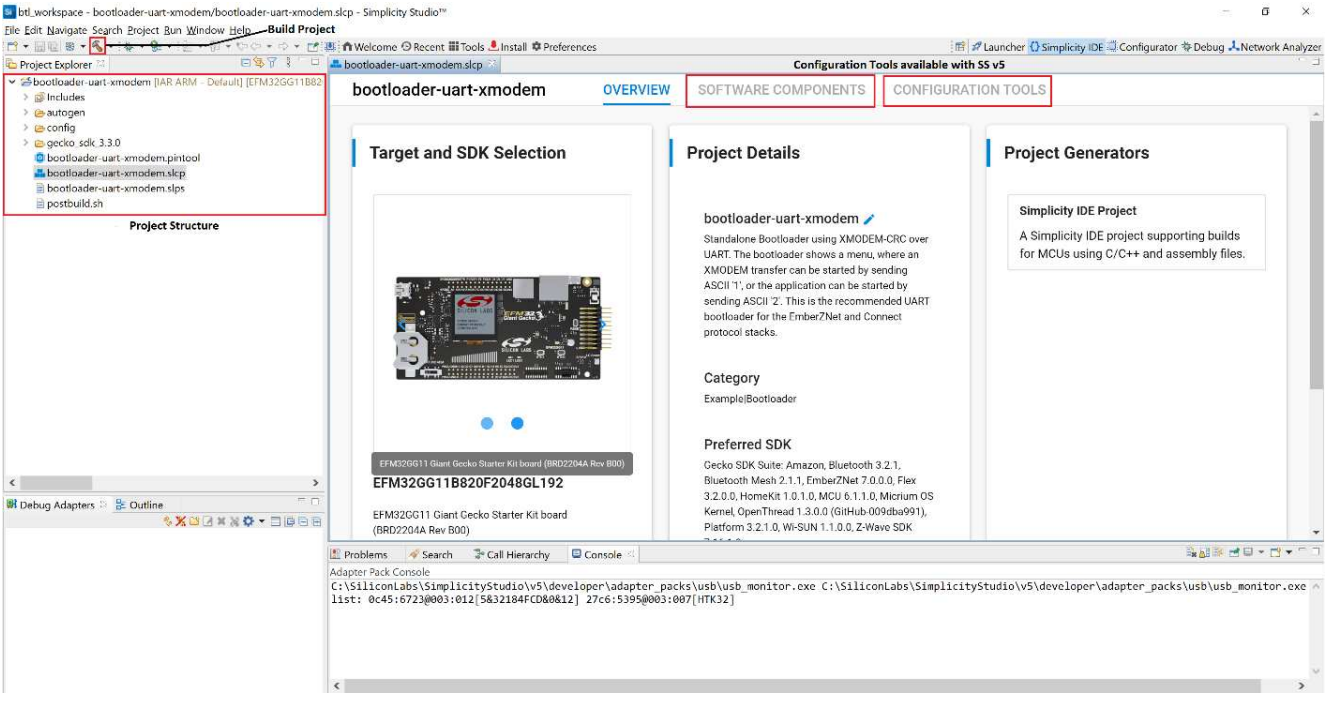
Click FINISH and the project will be generated.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Project File Structure

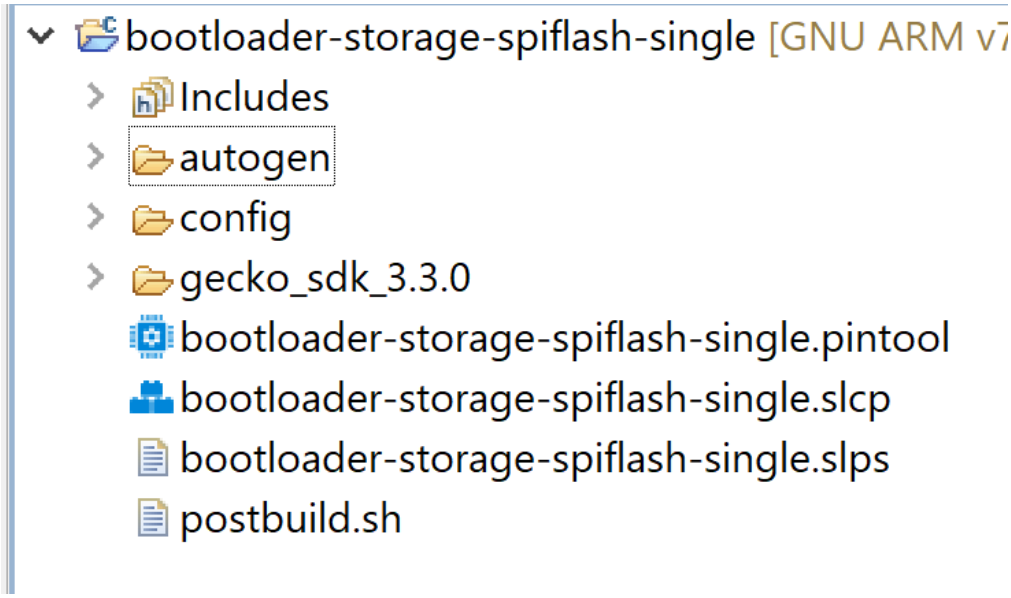
On project generation, the required project files and folders are created.



Projects always include the following parts:

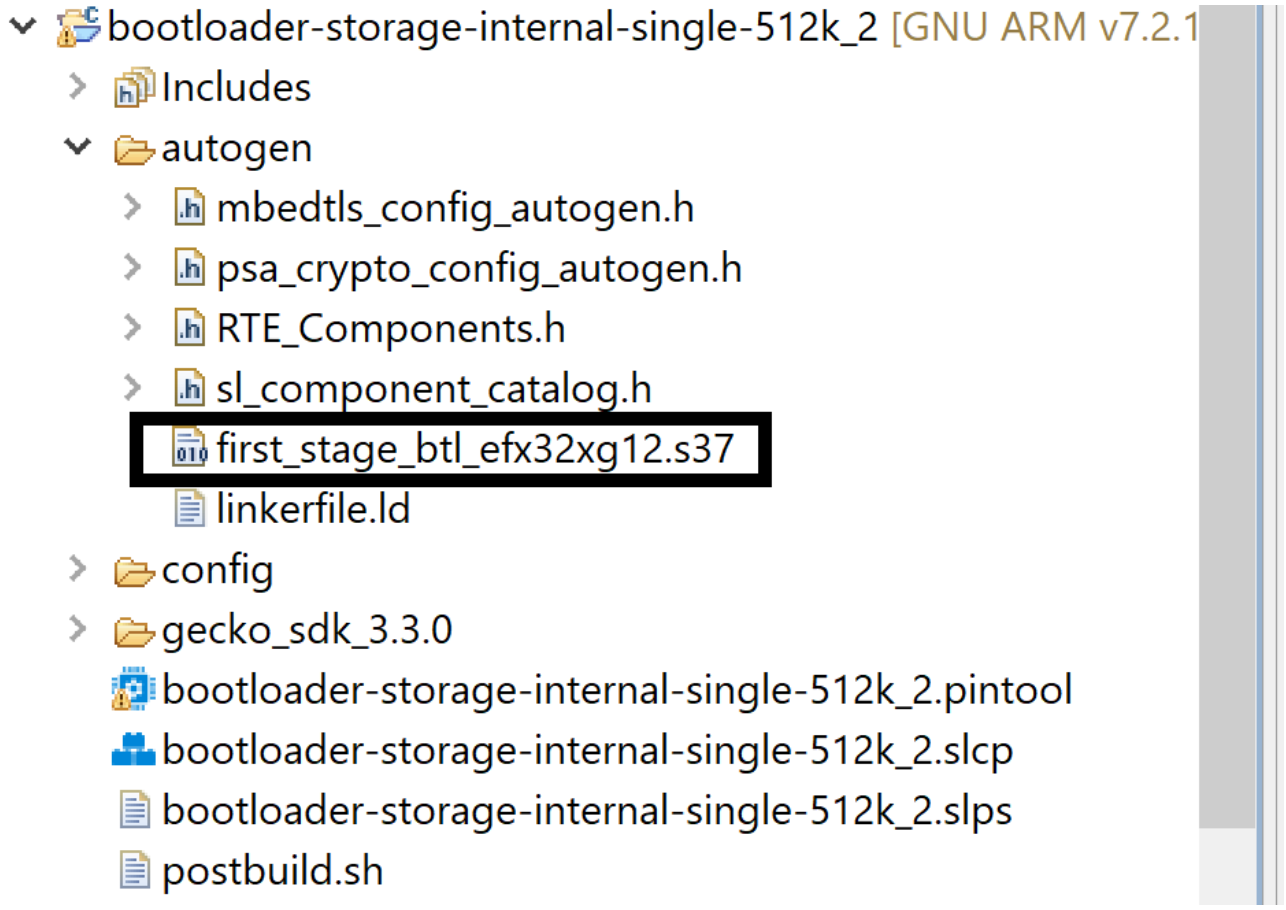
Copyright © 2025 Silicon Laboratories. All rights reserved.

- autogen folder: Only the autogen folder includes generated code. It includes configuration (mbedtls_config_autogen.h), init code, the linker script, and other generated code used by components, like the command descriptors for the CLI interface.
- config folder: Component configuration headers are located in this folder. These can be edited with the Simplicity IDE Component Editor, but directly editing the header file is also possible. The Component Editor is available through the Project Configurator's Configure control, available only for configurable components.
- gecko_sdk folder (with version number): Contains source and binary files added by components.
- files in the root folder: Only the application specific files should be in the root folder, including source files, the project configurator (*.slcp*) file and the Pin Tool *(.pintool*) file. For more details, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.



The first stage bootloader binary file is copied to the autogen folder at the time of project generation. The correct first stage binary file is located in accordance with the target hardware and placed in the autogen, which is later used by the postbuild script to generate a combined bootloader binary. The autogen folder is shown below with the first stage bootloader binary.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Configuring a Project

Most project configuration can be done through tools in Simplicity Studio. These tools provide the functionality previously provided by AppBuilder.

- Project Configurator: the top-level project configuration tool through which you install and uninstall components and change other project parameters. It also provides access to other configuration tools.
- Component Editor: provides access to the configurable parameters of an individual component.
- Pin Tool: configures peripherals.

Once you create the project, the Project Configurator tabbed interface is presented, including:

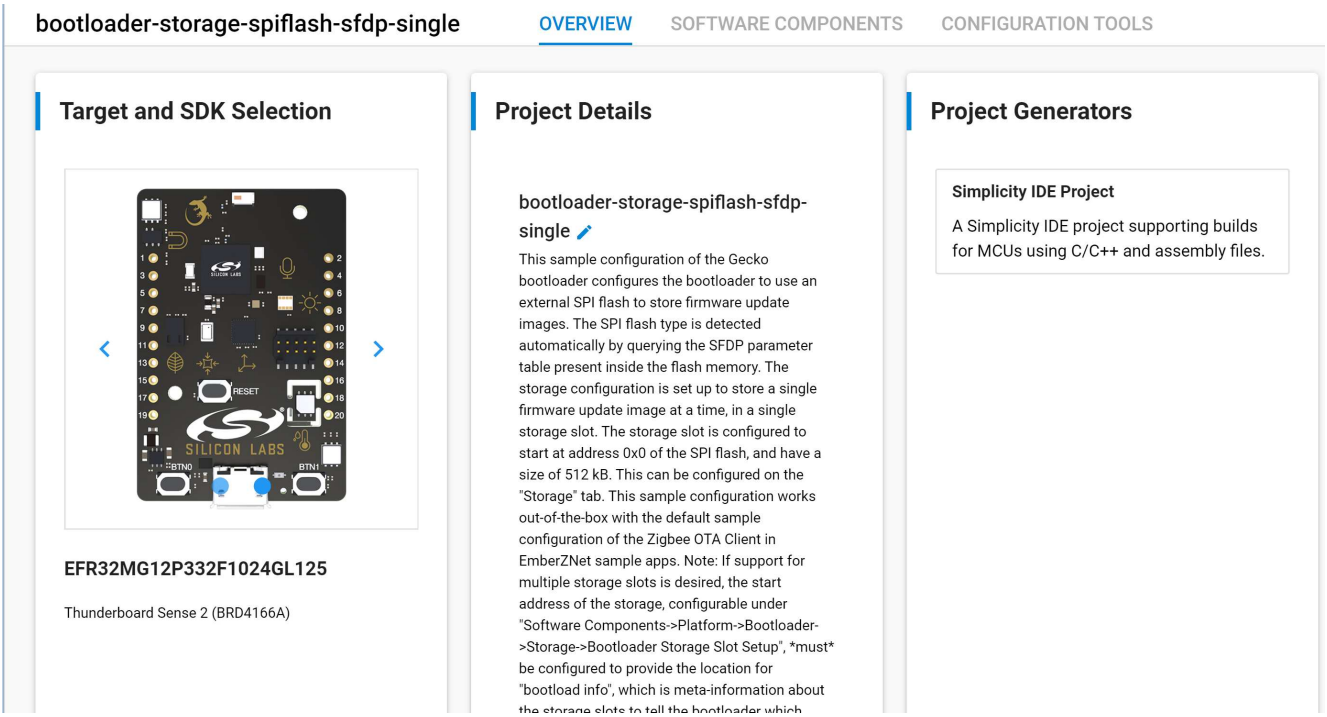
- Overview
- Software Components
- Configuration Tools

You can always re-open the Project Configurator by double-clicking the <projectname>.slcp file. For details on using the Project Configurator and Component Editor, see the Simplicity Studio 5 User's Guide.

OVERVIEW Tab

The OVERVIEW tab, like AppBuilder's General tab, shows the target part information, and a description of the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

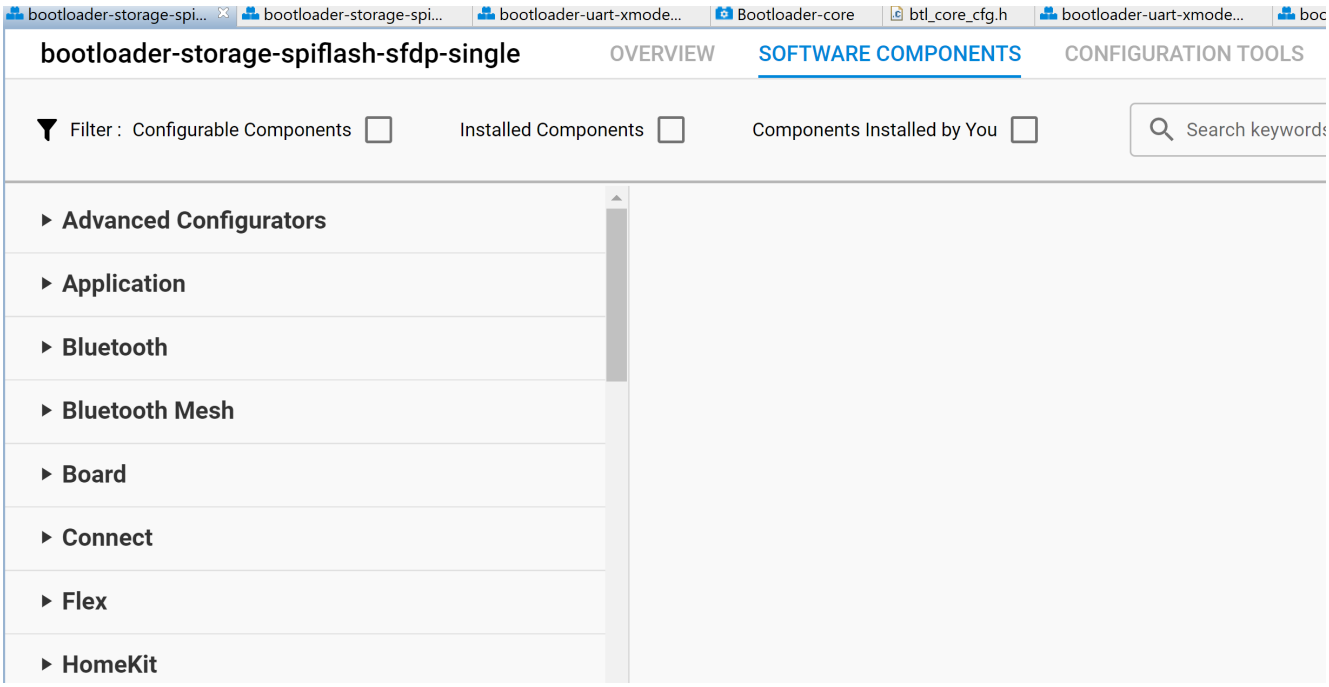


Note that there is no Generate control. Changes made through Project Configurator are auto-saved and the project files are auto-generated. The Force Generation control on the Project Details card is provided for use only in the event of a system problem, where auto-generation fails.

The Project Generators interface controls generation of additional files for import into other IDEs. It does not change the compiler used to build the application image.

SOFTWARE COMPONENTS Tab

The SOFTWARE COMPONENTS tab shows the available components and those that are already installed in the example code.

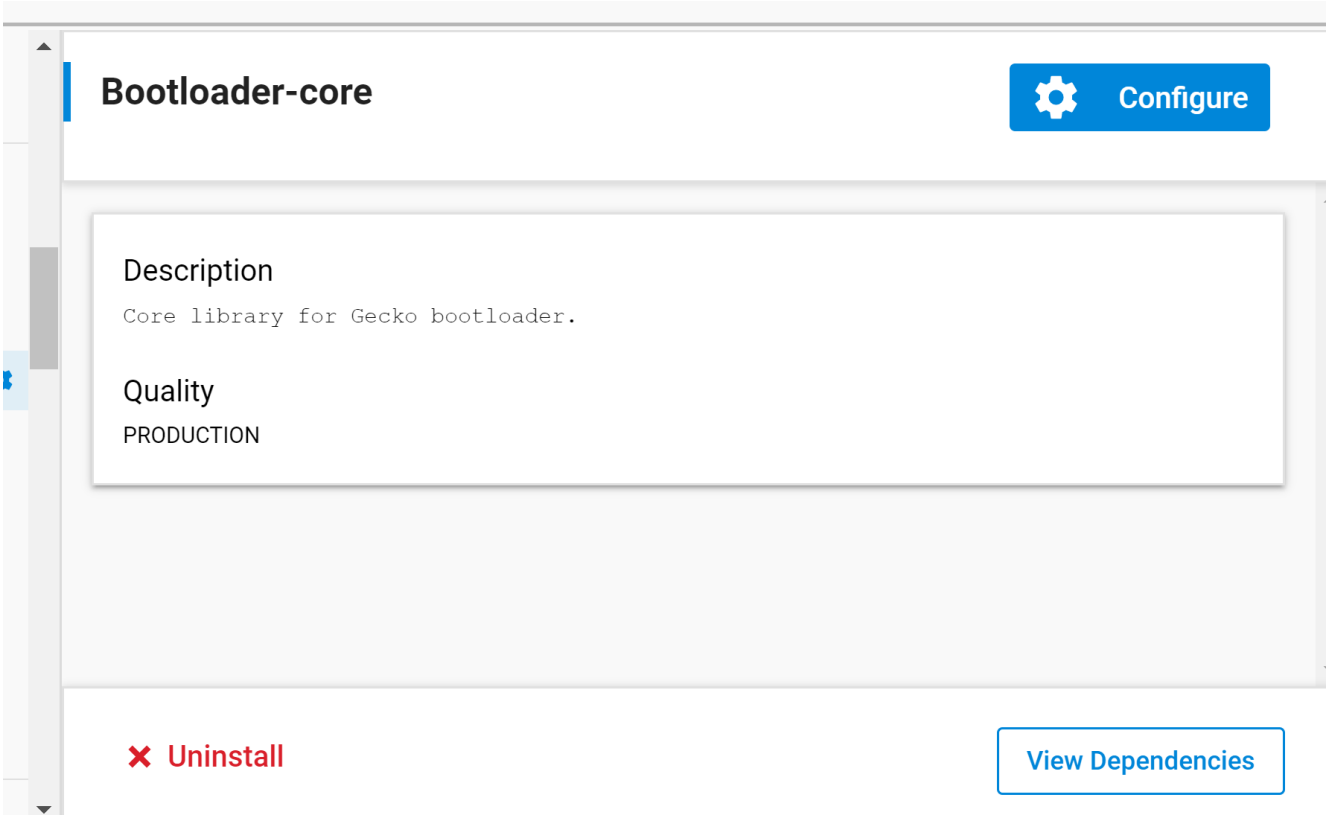


Copyright © 2025 Silicon Laboratories. All rights reserved.

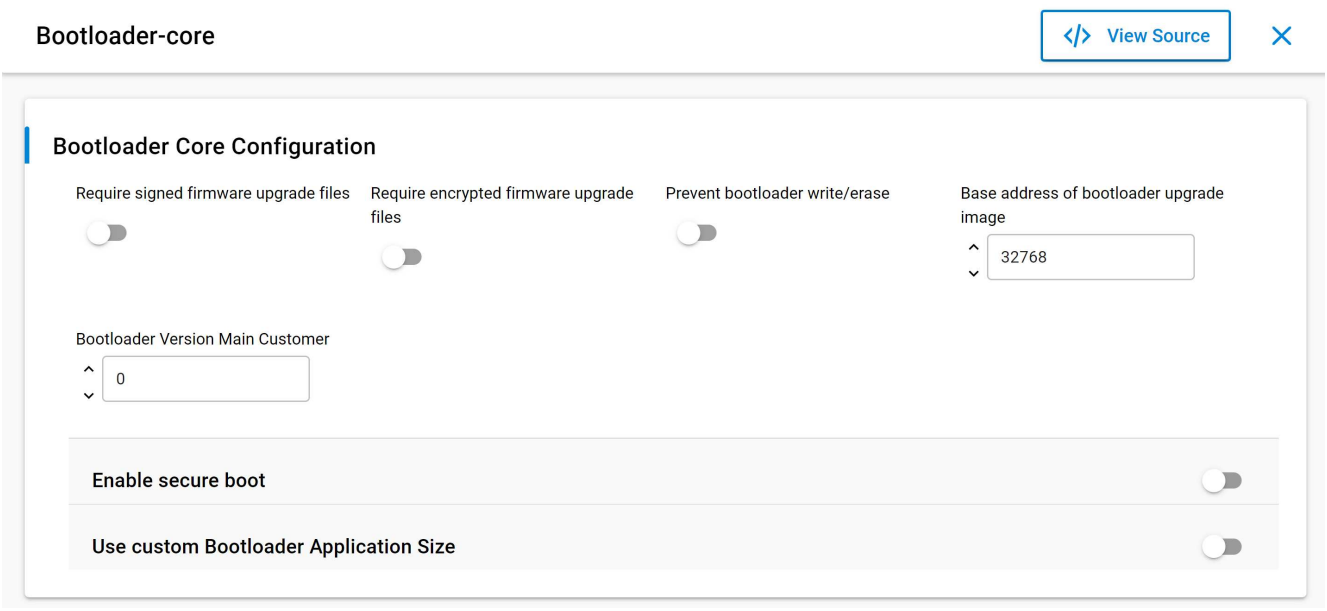
Component groupings are subject to change in SDK releases, as new components are added and organization is updated. The most reliable way to find a specific component is to search for it. You can also filter the displayed components using the checkboxes at the top.

Installed components have a circled checkmark to the left. Configurable components have a gear icon to the right.

Select a component to see details about it. Click Install to install the component or Uninstall to remove it. If it is configurable, a Configure control is shown in the upper right.



Click that control or the gear icon next to an installed component name to open the Component Editor in a new tab.

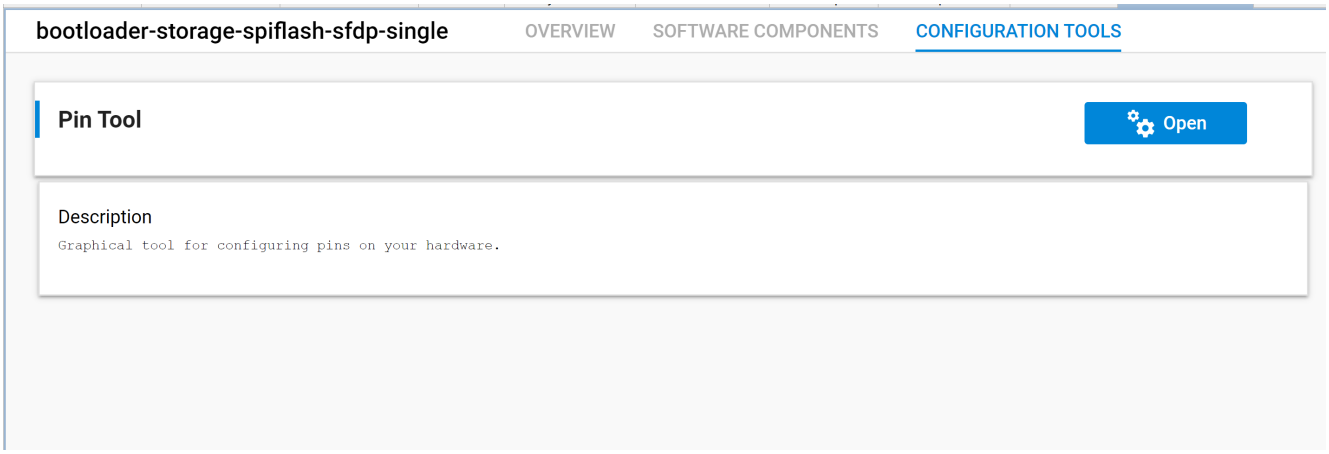


Copyright © 2025 Silicon Laboratories. All rights reserved.

Save any changes to trigger file generation.

CONFIGURATION TOOLS Tab

The CONFIGURATION TOOLS tab is a quick way of opening useful tools for project development.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Components

Components

This section maps the various Gecko Bootloader components to the previously available plugins (in the AppBuilder workflow). Although the basic subgroups might remain the same, specific component locations are subject to change across various GSDK versions. Therefore, it is good practice to search for individual components by their names rather than the specific subgroups that they belong to.

AppBuilder Plugin	Simplicity Studio v5 Component	Additional Comments
Communication		
BGAPI UART DFU	BGAPI UART DFU	
EZSP-SPI	EZSP-SPI	
UART XMODEM	UART XMODEM	
XMODEM Parser	XMODEM Parser	
Core		
Application Upgrade Version Check	Application Upgrade Version Check	
Bootloader Core	Bootloader Core	
GBL Compression (LZ4)	GBL Compression (LZ4)	
GBL Compression (LZMA)	GBL Compression (LZMA)	
Image Parser	Image Parser	
Image Parser with legacy EBL support	Image Parser with legacy EBL support	
Image Parser without encryption support	Image Parser without encryption support	
“	Bootloader Include Parser	New component that includes common Image Parser header files
“	Bootloader in Main Flash	New component that enables the user to place bootloader in main flash for xG13 and xG14 devices
Drivers		
Delay	Bootloader Delay Driver	
SPI Master	Bootloader SPI Controller USART Driver	
SPI Slave	Bootloader SPI Peripheral USART Driver	
UART	Bootloader UART Driver	
EUART	Bootloader EUART Driver	

Copyright © 2025 Silicon Laboratories. All rights reserved.

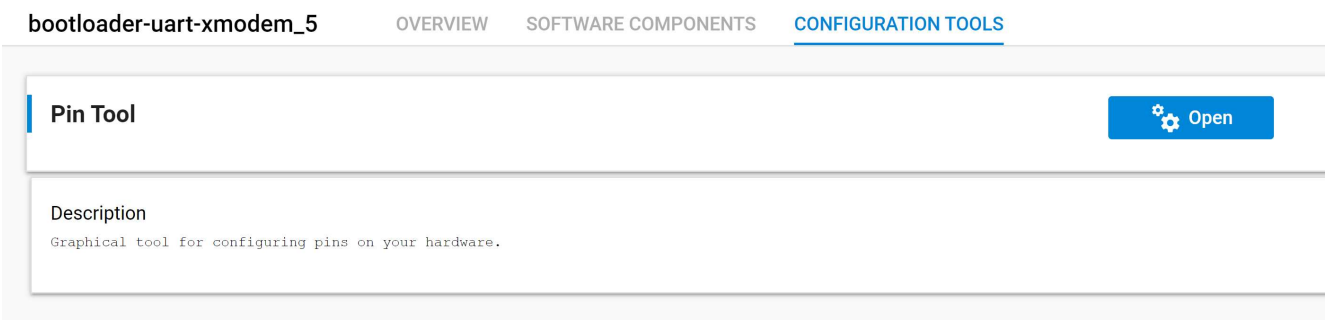
AppBuilder Plugin	Simplicity Studio v5 Component	Additional Comments
“	Bootloader SPI Controller EUSART Driver	New component supporting SPI EUSART driver. This component can be used with devices supporting EUSART interface.
“	Bootloader SPI Peripheral EUSART Driver	New component supporting SPI EUSART driver. This component can be used with devices supporting EUSART interface.
Storage		
Common Storage	Common Storage	
Common Storage (single storage slot only)	Common Storage (single storage slot only)	
Internal Storage	Internal Storage	
SPI Flash Storage	SPI Flash Storage	
“	Bootloader Storage Slot Setup	New component that facilitates configuring storage slots for Internal and SPI Flash based bootloaders
Utils		
Crypto	Crypto	
Cyclic Redundancy Check	Cyclic Redundancy Check	
Debug	Debug	
EMLIB	EMLIB Peripheral HAL	This component is no longer part of the Bootloader module. EMLIB component(s) can be found under Platform > Peripheral
EZSP GPIO Activation	EZSP GPIO Activation	
GPIO Activation	GPIO Activation	
SE Manager	SE Manager	This component is now available under Platform > Security. This component is only applicable for Series 2 devices.
Token Management	Token Management	
mbed TLS	Mbed TLS …	These components are available under Platform > Security
Bootloader Interface		
	App Properties	A new component to configure application properties (version), which is used during creation of a GBL file using Simplicity Commander.
Bootloader-interface	Bootloader Application Interface	

Copyright © 2025 Silicon Laboratories. All rights reserved.

Pin Tool

Pin Tool

In contrast to the AppBuilder workflow, Simplicity Studio v5 offers the Pin Tool configuration tool, which can be used to configure the various pins of the MCU used in the project. Pin Tool can be used in standalone mode through the tile on the Configuration Tools tab.



Pin Tool-related configuration is also available through the Component Editor of all applicable configurable components, making it a one-stop solution for configuring pin details and setting user configuration details. Following is the list of configurable components that also include Pin Tool configuration (wherever applicable) in their configuration wizard.

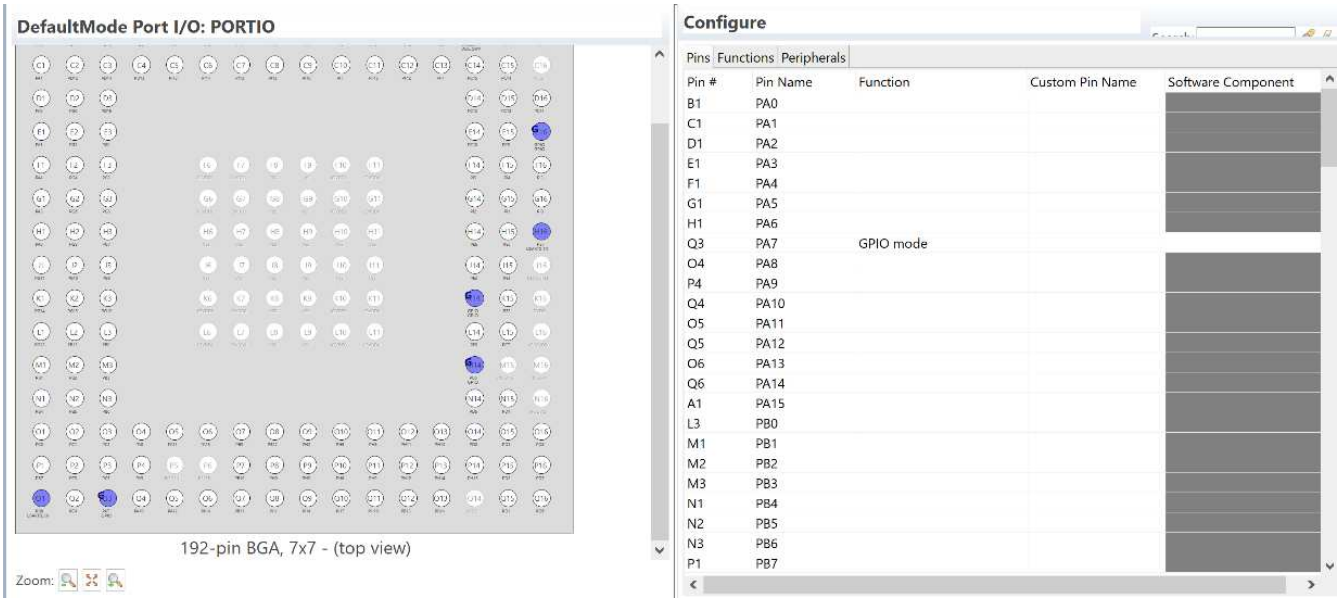
Configurable Component	Configuration Options
UART XMODEM	Menu Idle Timeout
Bootloader Core	All options that were available for the bootloader plugin are now available in the Bootloader Core component. Options include: Enable Secure Boot, Enable certificate support, Prevent bootloader write/erase, etc
Bootloader UART/EUART Driver	Configuration options include: USART settings, USART peripheral, Serial VCOM Enable/Disable
Bootloader SPI Controller USART/EUSART Driver	Configuration options include: Frequency, Peripheral and SPI Port Settings
Bootloader SPI Peripheral USART/EUSART Driver	Configuration options include: Tx Buffer Size, Rx Buffer Size, Peripheral and Port Settings
Bootloader Storage Slot Setup	Supports configuration of storage slot (start addresses and slot lengths) up to a maximum of 3 slots
Common Storage/Common Storage (single storage slot only)	Configuration options include: Start address of bootload info
Internal Storage	Configuration options include: Enable DMA based MSC Write, DMA channel to reserve
SPI Flash Storage	Enables configuration of the list of SPI flash devices that must be enabled in the bootloader image
Debug	Enable/disable Debug prints/asserts
EZSP GPIO Activation	Configuration options include: Properties of SPI NCP, WAKE INT Pin, HOST INT Pin
GPIO Activation	Configuration options include: Properties of Bootloader entry, Button Pin configuration

The configuration options of any installed configurable component can be accessed by clicking Configure. Refer to section 2 Project Creation for more details on the Component Editor.

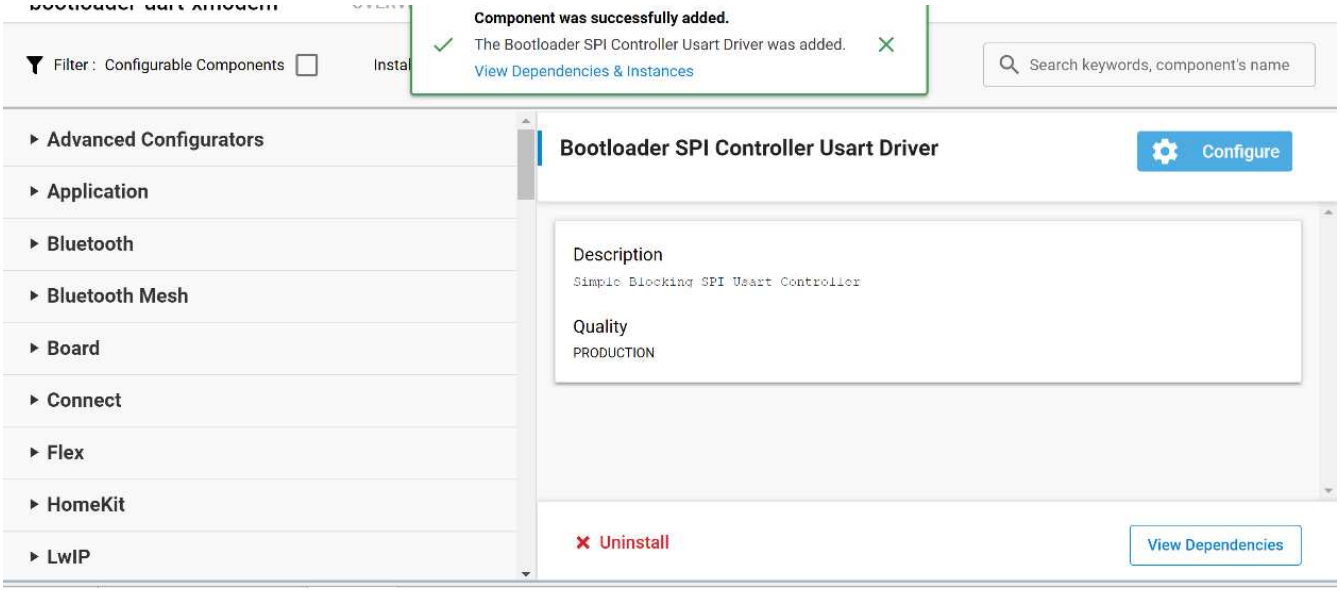
Copyright © 2025 Silicon Laboratories. All rights reserved.

As an example, in the following procedure the Bootloader SPI Controller USART Driver is configured using the Component Editor and the pin configuration is verified using the Pin Tool.

1. Pin Tool configuration before installing the SPI Controller component. The device used for this example is the EFM32GG11 Giant Gecko Starter Kit board (BRD2204A Rev B00).

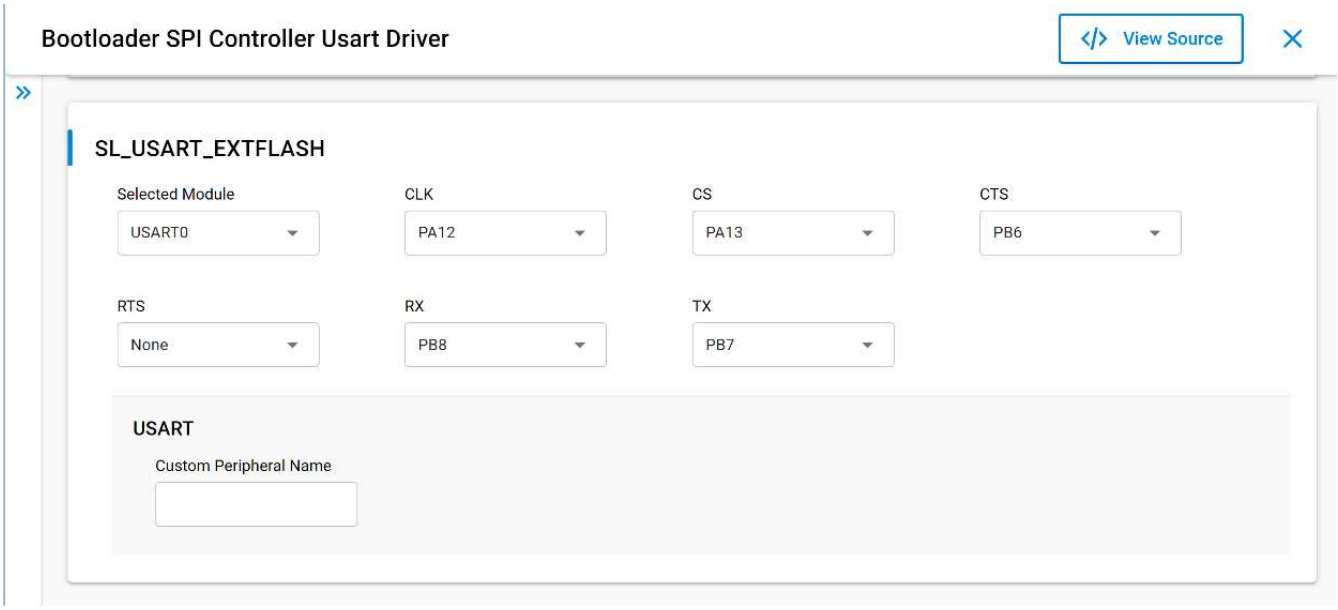


2. Install the Bootloader SPI Controller USART driver component.

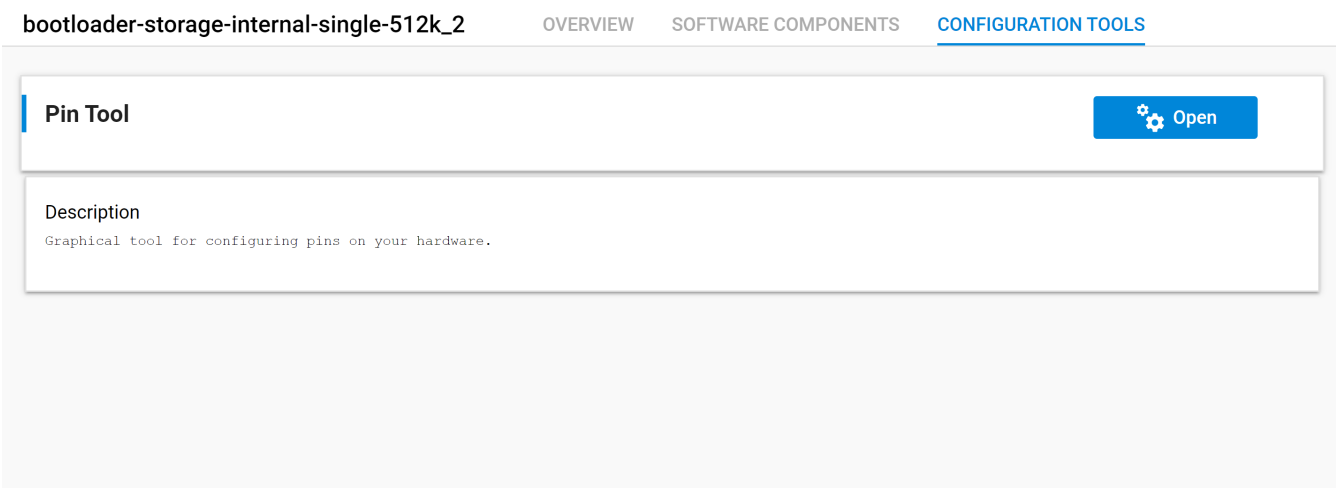


3. Click Configure to open the Component Editor and configure the USART peripheral as shown.

Copyright © 2025 Silicon Laboratories. All rights reserved.

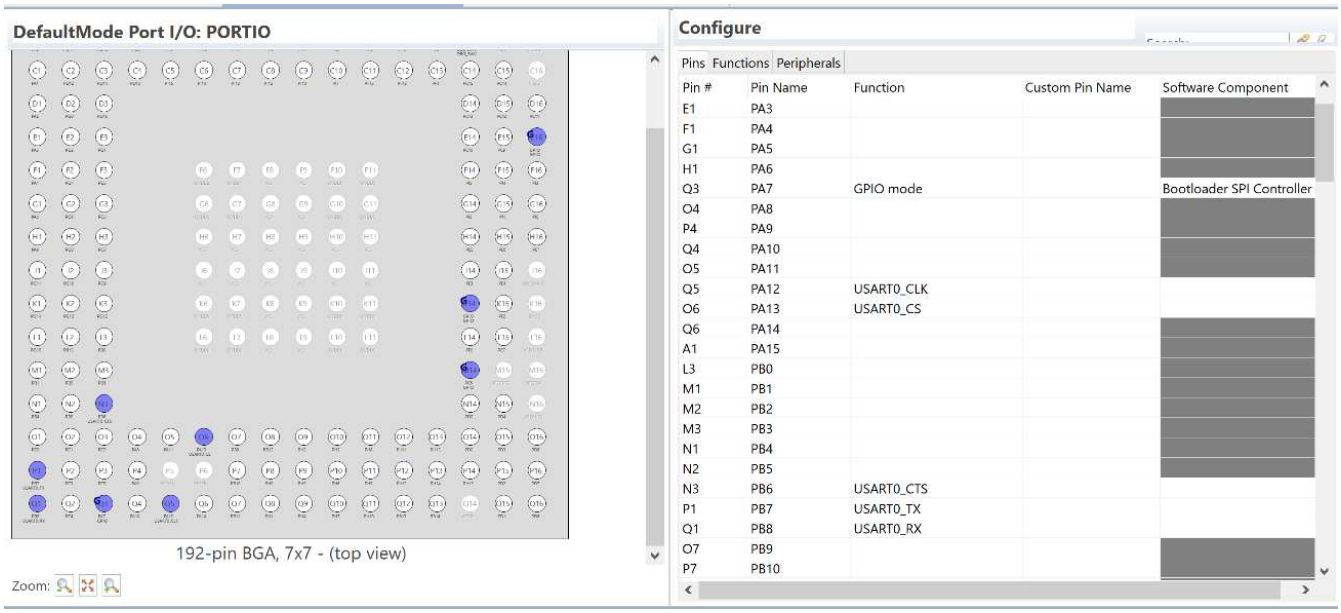


4. Open the Pin Tool.



5. The pin configurations are shown. Notice that pins PA12, PA13, PB6, PB7, and PB8 are reserved for USART0.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Linker File

Linker File

The linker files for GCC and IAR are now generated during the project generation step. The linker files for the bootloader are generated from template files using the Jinja template engine. The template files are included with theGecko SDK Suite (GSDK), which is installed through Simplicity Studio. The jinja template files are located at <path-to-simplicity-studio-installation>|v5|developer|sdks|gecko_sdk_suite|<gecko_sdk_version>|platform|common|toolchain|.This path contains folders for both gcc and iar.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Additional Macros

Additional Macros

In the AppBuilder workflow, the user could define additional compiler macros used during compilation of the bootloader application. In Gecko Bootloader v2.x, these additional macros are now part of the various components. These components have these macros defined internally, and those macros which require user intervention are available to the user as configuration options through the Component Editor.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Postbuild Steps

Postbuild Steps

For Series 1 devices, the main stage bootloader needs to be combined with the first stage bootloader, which results in a combined bootloader binary. To accomplish this, a postbuild script is made available with the GSDK, which combines the appropriate first stage bootloader with the main stage bootloader to produce a combined bootloader binary. A postbuild step must be explicitly added through the Project's settings so that the postbuild script can be run after the bootloader build is complete. For more details, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

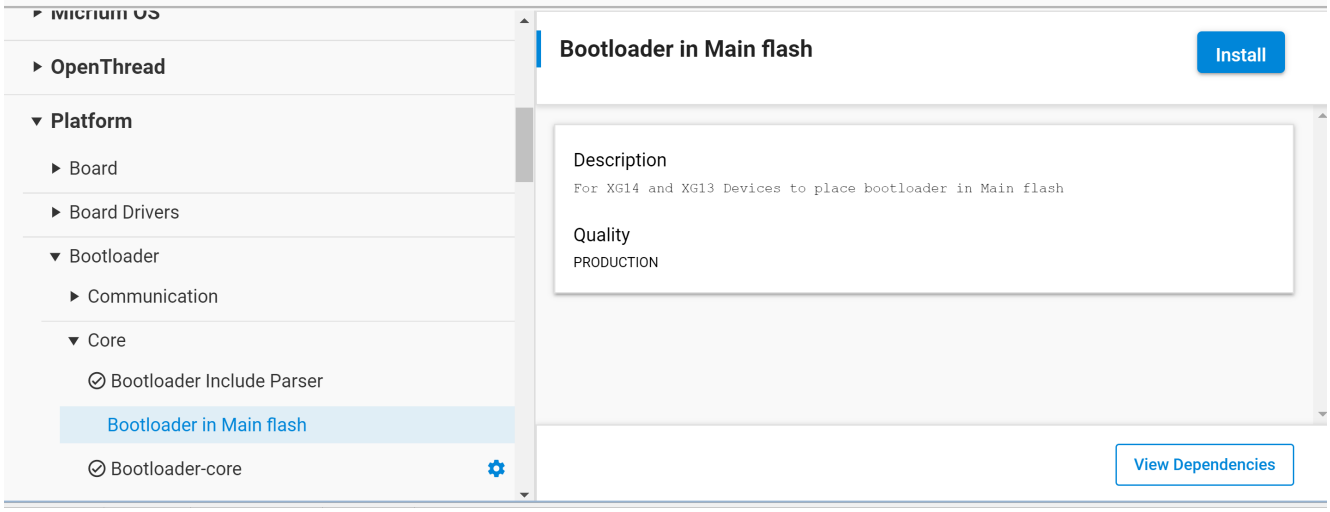
> Note: Adding the postbuild steps is mandatory when building bootloader binaries for Series 1 devices. If the postbuild step is not configured correctly, Simplicity Studio cannot build a combined bootloader binary image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Main Bootloader in Main Flash

Main Bootloader in Main Flash

For xG13 and xG14 devices, the entire main stage bootloader might not fit into the bootloader flash if the user installs some extra components. In such scenarios, the main stage bootloader can be placed in the main flash by installing the Bootloader in Main Flash core component in the bootloader project.



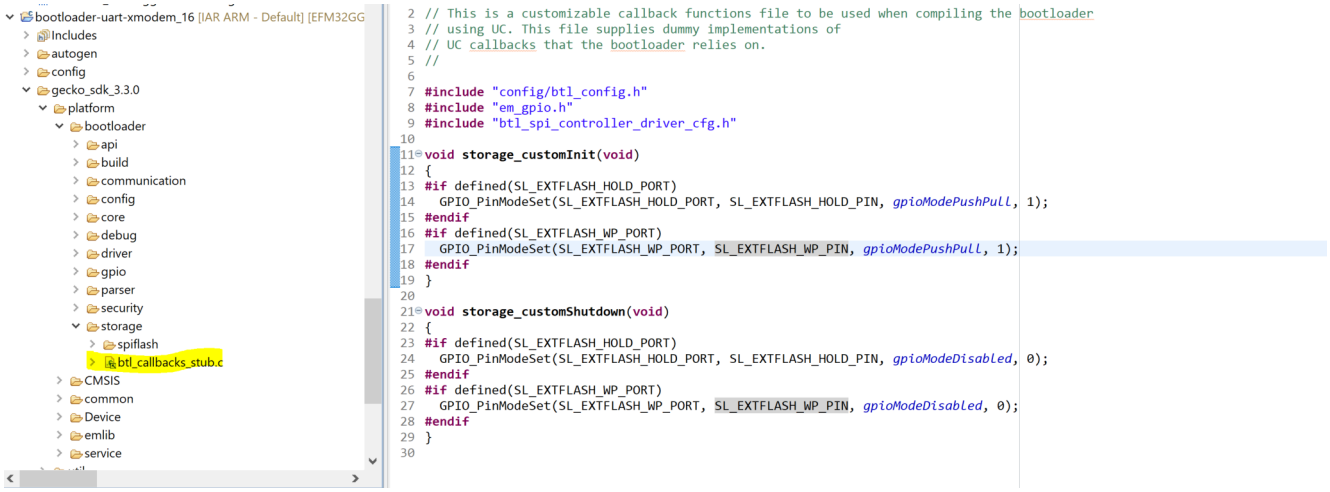
For more information on this component, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Callbacks

Callbacks

In contrast to the AppBuilder workflow, callbacks are now part of btl_callbacks_stub.c. This file is added to the project when the SPI Flash Storage component is installed. This file contains dummy implementation of callbacks that the bootloader relies on. This file can be found in gecko_sdk_<version> > platform > bootloader > storage > btl_callbacks_stub.c as shown.

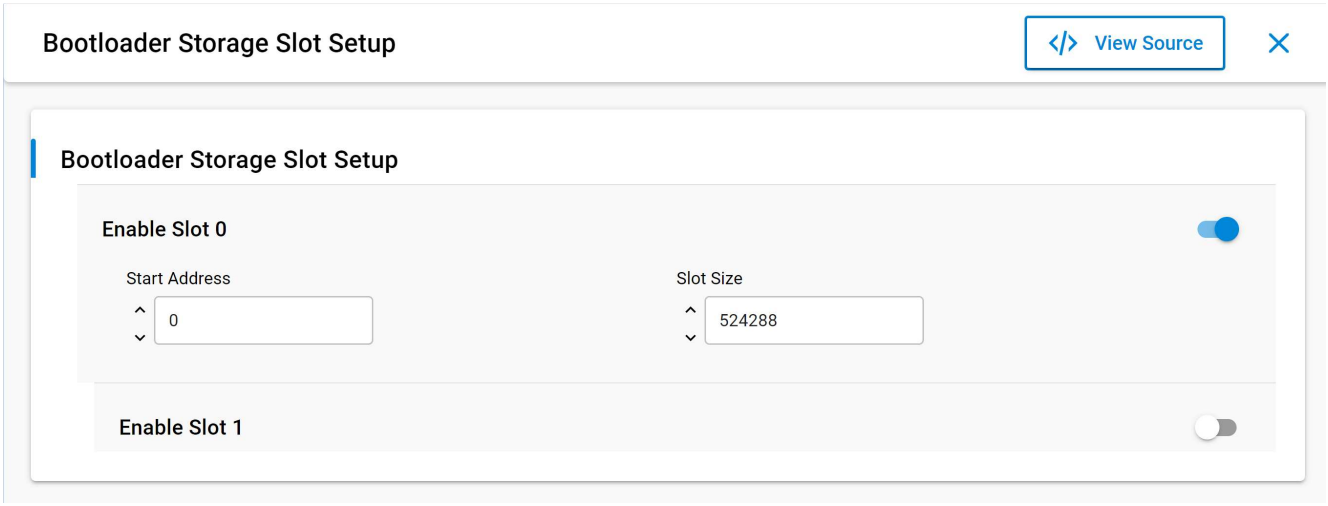


Copyright © 2025 Silicon Laboratories. All rights reserved.

Storage Slots

Storage Slots

Gecko Bootloader supports the Application Bootloader mode. This mode supports single/multiple storage slots that can be configured at compile time. In Gecko Bootloader v2.x, the storage slots can now be configured in the Bootloader Storage Slot Setup component using the Component Editor. A maximum of 3 storage slots can be configured.



Copyright © 2025 Silicon Laboratories. All rights reserved.

App Properties

App Properties

Previously, in the AppBuilder workflow, a variable of type AppProperties_t had to be configured to add application properties to the application. This was then later used by Simplicity Commander during gbl file creation. Beginning with Gecko Bootloader version 2.x, a new configurable component named App Properties can be used to configure the application properties. This component can be installed using the Component editor which is available as part of Simplicity Studio. It is also installed automatically as a dependency on installing the Bootloader Application Interface component in the project. The component allows the user to configure the application version using Simplicity Studio's Component Editor as shown below.



For more details on how to configure this component, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Security

The following documents provide additional information about Gecko Bootloader security.

- Series 2 Secure Boot with RTSL: Contains detailed information on configuring and using the Secure Boot with hardware Root of Trust and Secure Loader on Series 2 devices, including how to provision the signing key.
- Series 2 TrustZone: Provides background on ARMv8-M TrustZone, a technology that provides a foundation for improved system security in embedded applications, and discusses its implementation on Series 2 devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 and Series 3 Secure Boot with RTSL

Series 2 and Series 3 Secure Boot with RTSL

NOTE: This section replaces *AN1218: Series 2 and Series 3 Secure Boot with RTSL*. Further updates to this application note will be provided here.

This application note describes the design of Secure Boot with RTSL (Root of Trust and Secure Loader) on Series 2 and Series 3 devices. It also provides examples of how to implement the Secure Boot process.

For more information on using the Gecko Bootloader with Series 2 and Series 3 devices, see the following:

- Bootloader Fundamentals
- UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower
- Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher
- Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher

Key Points

- Compares the Secure Boot process in Series 1, Series 2, and Series 3 devices
- Describes the Series 2 and Series 3 Secure Boot with RTSL components and process
- Provides examples of configuring Series 2 and Series 3 devices for the Secure Boot process
- Recovers secure boot failure devices

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 and Series 3 Device Security Features

Series 2 and Series 3 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 and Series 3 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 and Series 3 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- Series 2 devices: EFR32xx2xx devices
- Series 3 devices: SiG3xxx devices
- HSE: Hardware Secure Engine
- VSE: Virtual Secure Engine
- SE: Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support
Secure Vault High (SVH) and Series 3 Secure Vault	HSE only (HSE-SVH)
Secure Vault Mid (SVM)	HSE (HSE-SVM), VSE (VSE-SVM)
Secure Vault Base (SVB)	N/A

Notes:

1. The features of different Secure Vault levels can be found in Security.
2. Refer to IoT Endpoint Security Fundamentals for details on supporting devices.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High and Series 3 Secure Vault devices offer additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

> Note: Series 3 Secure Vault devices do not include a device certificate for the host MCU.

Copyright © 2025 Silicon Laboratories. All rights reserved.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

In support of these products, Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 and Series 3 Secure Debug	How to lock and unlock Series 2 and Series 3 debug access, including background information about the SE	Secure Vault Mid, High, and Series 3 devices
Series 2 and Series 3 Secure Boot with RTSL (this application note)	Describes the secure boot process on Series 2 and Series 3 devices using SE	Secure Vault Mid, High, and Series 3 devices
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High, and Series 3 devices
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High, and Series 3 devices
Secure Key Storage	How to securely "wrap" keys so they can be stored in non-volatile storage.	Secure Vault High, and Series 3 devices
Production Programming of Series 2 and Series 3 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid, High, and Series 3 devices
AN1504: Series 3 Security Overview	High level overview of the security features included in Series 3 devices	Series 3
AN1509: Series 3 AXiP	How to encrypt and authenticate memory contents	Series 3

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity
AXiP Key	No	Used on Series 3 devices for encryption/ decryption and authentication of firmware placed in flash
EXiP Key	No	Used on Series 3 devices for encryption/ decryption of firmware placed in flash

SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 and Series 3 devices to support the required security features. Refer to Production Programming of Series 2 and Series 3 Devices for the procedure to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 and Series 3 devices and modules.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Boot Process

Secure Boot Process

Introduction

Secure Boot is a foundational component of platform security, as the absence of it can allow malicious code to subvert protections like secure storage, secure data transport, secure identity, and data confidentiality.

Secure Boot works as a process by which each piece of firmware is validated for authenticity and integrity before it is allowed to run. Each authenticated module can also validate additional modules before executing them, forming a chain of trust. If any module fails its security check, it is not allowed to run, and program control will typically stall in the validating module. In most lightweight IoT systems, the behavior of a Secure Boot failure is to cause the device to stop working until an authentically signed image can be loaded onto it. Whereas this may seem extreme, it is a better outcome than a smart light bulb being repurposed to mine crypto-currency, or a smart speaker being repurposed as a surveillance device on the end user's private conversations.

The first link in the chain of trust is the root of trust. This is often the weakest link in the Secure Boot chain because the root of trust itself is not checked for authenticity or integrity. The security strength of the root of trust lies in its immutability. The strongest roots of trust have their firmware origin in ROM and use a Public Sign Key that is also located in ROM.

Series 1, Series 2, and Series 3 devices use a two-stage boot design consisting of a non-upgradable first stage root of trust followed by an upgradable second stage. In Series 1 devices, the root of trust (also called the first-stage bootloader) is in flash rather than ROM, and the upgradable portion (the main bootloader) is checked for integrity using a CRC32 checksum but is not checked for authenticity using a Public Sign Key. In Series 2 and Series 3 devices, the root of trust is in ROM, and the upgradable portion is checked both for integrity and authenticity.

The Secure Boot with RTSL is implemented by Root code executed by the Hardware Secure Engine (HSE) or the Cortex-M33 operating in Root Mode (VSE). For more information about SE, see section *Secure Engine Subsystem* in Series 2 and Series 3 Secure Debug.

In Series 3 devices, once firmware is programmed to the device, the code regions must be closed. Closing a code region locks it to prevent writes, as well as further configuration, and indicates to the SE that it can now be accessed by the SE. After flashing the firmware image, commander automatically closes the code region. To open a closed code region, an erase must be performed. If a code region is not closed, the SE will return a fault when attempting to access the open code region during secure boot verification of the host firmware. Each time a code region is closed, the SE's MTP (Multiple Times Programmable) memory section is updated, which causes an SE OTP bit to be used. More details on SE OTP bits is covered in the following section.

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to customize the users' security features and settings. This application note uses the following abbreviations:

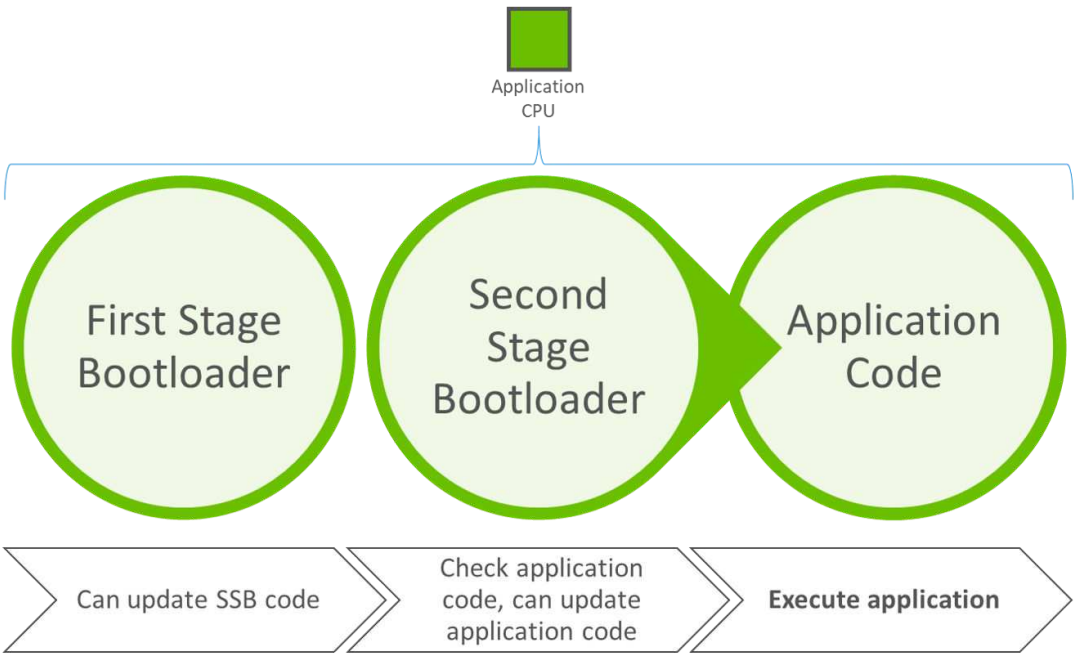
- FSB: First Stage Bootloader
- SSB: Second Stage Bootloader
- GBL: Gecko Bootloader
- RTSL: Root of Trust and Secure Loader
- HSM: Hardware Security Module
- OTP: One-Time Programmable
- WSTK: Wireless Starter Kit
- GSDK: Gecko Software Development Kit. For more information, refer to GSDK
- SiSDK: Simplicity Software Development Kit. For more information, refer to SiSDK
- ECDSA-P256-SHA256: Elliptic Curve Digital Signature Algorithm aka ECDSA using a P-256 curve and a SHA256 hash
- PEM (.pem): Privacy Enhanced Mail
- DER (.der): Distinguished Encoding Rules

ECDSA-P256-SHA256 Secure Boot in Series 1 Devices

The Secure Boot process for Series 1 (SVB) devices originates in flash, typically with the execution of the first stage of GBL. The first stage of GBL checks to see if an upgrade is pending for the second stage of GBL. If so, it processes the upgrade of the

Copyright © 2025 Silicon Laboratories. All rights reserved.

second stage and then executes it. Otherwise, it just executes the second stage. If Secure Boot is enabled, the second stage of GBL checks the integrity and authenticity of the application image before executing it. If the integrity check fails, program control remains in the SSB. The following figure illustrates the Secure Boot process on Series 1 devices.



See UG266/Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher for more information to generate and download signed firmware images using Simplicity Commander.

ECDSA-P256-SHA256 Secure Boot in Series 2 and Series 3 Devices

For Series 2 and Series 3 devices, the Secure Engine (SE) implements the FSB to authenticate and upgrade the SSB. The GBL implements the SSB (aka Main Bootloader in UG266/Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher) to authenticate and upgrade the application firmware.

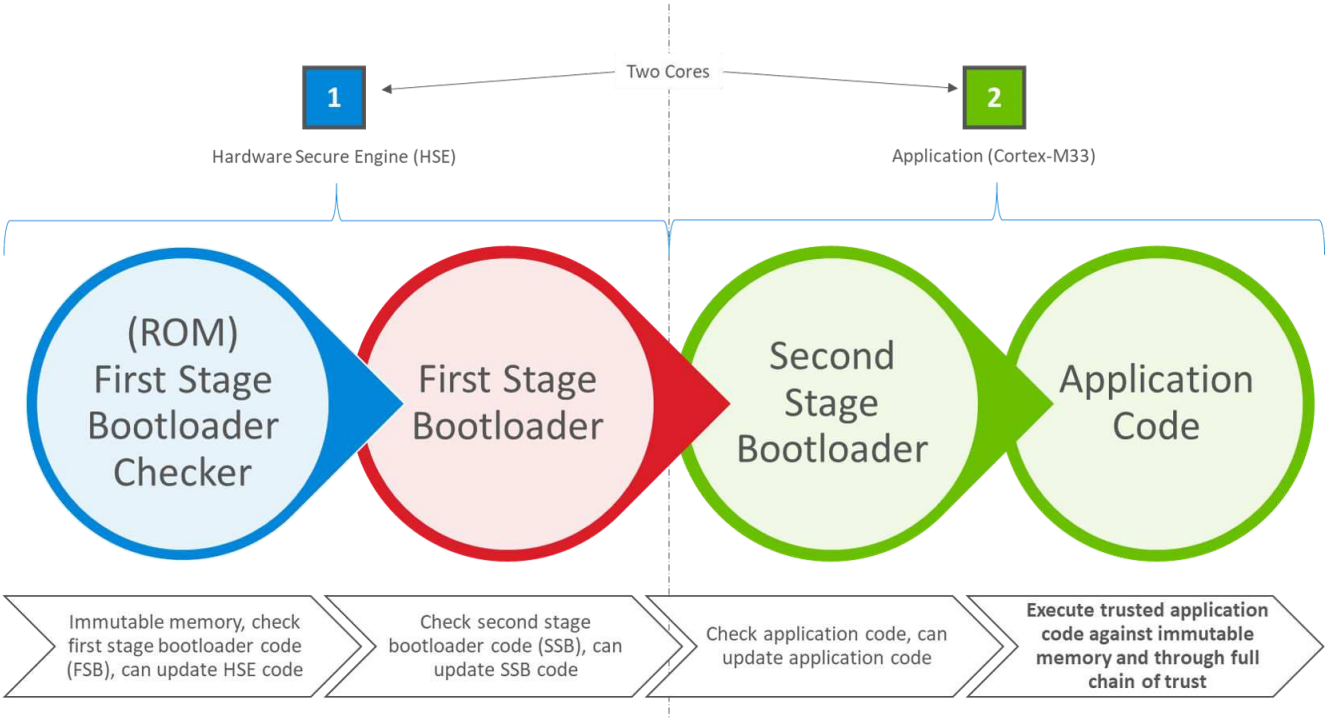
Refer to the *Gecko Bootloader Security Features* section in UG266/UG489 and ECDSA-P256-SHA256 Secure Boot example for more information about the ECDSA-P256-SHA256 secure boot process in Series 2 devices and Series 3.

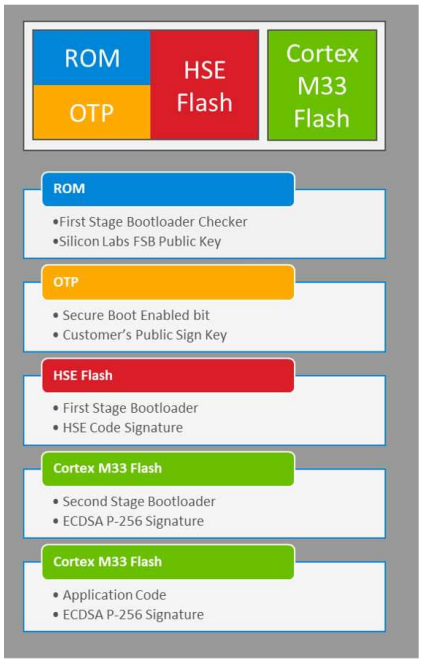
> Note: It is possible to have a 2-stage design to skip the SSB between FSB and application. However, the application cannot be upgraded if discarding the SSB, and this application note assumes the SSB is present.

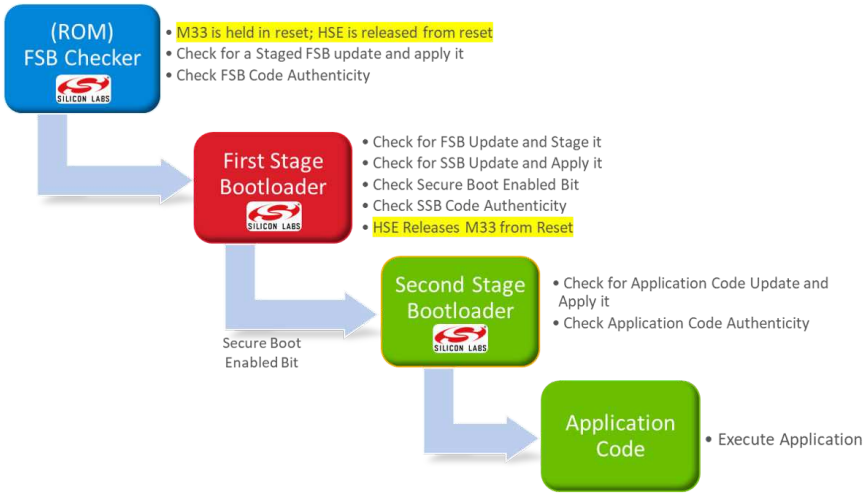
HSE

In HSE-SVM, HSE-SVH, and Series 3 Secure Vault devices, the Secure Boot process originates in ROM contained in the security co-processor (HSE). The following figures illustrate the Secure Boot process and flow on Series 2 HSE and Series 3 devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.





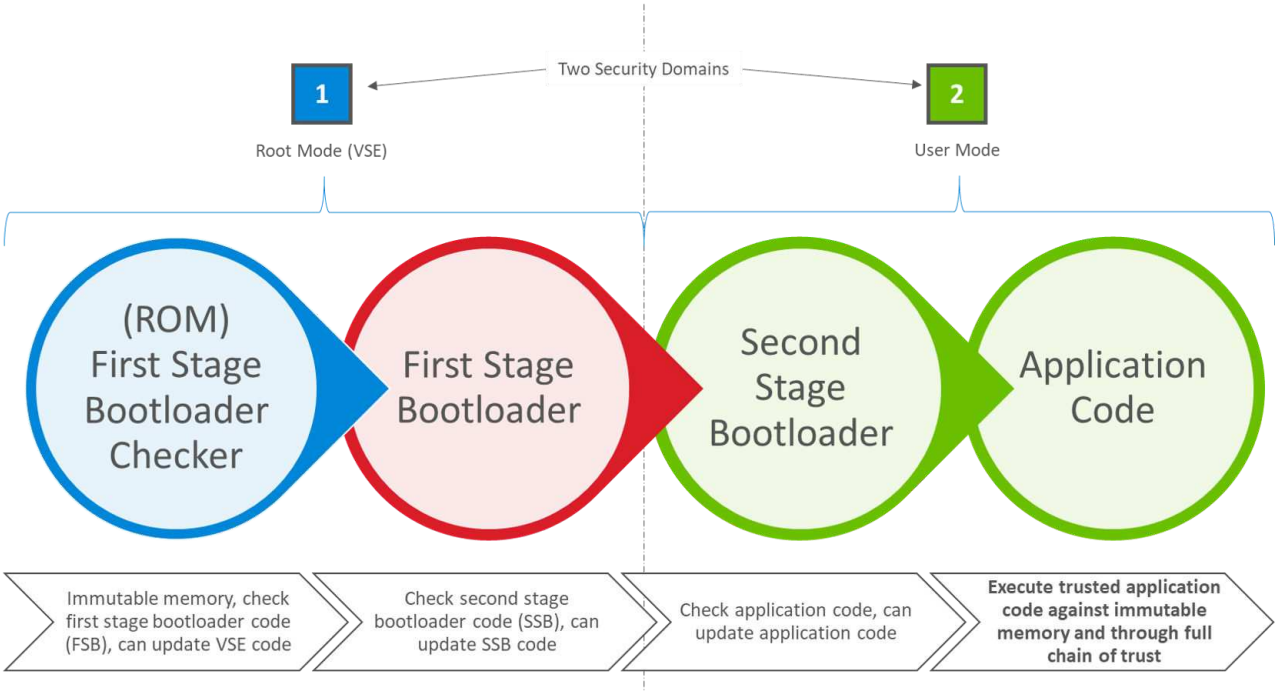


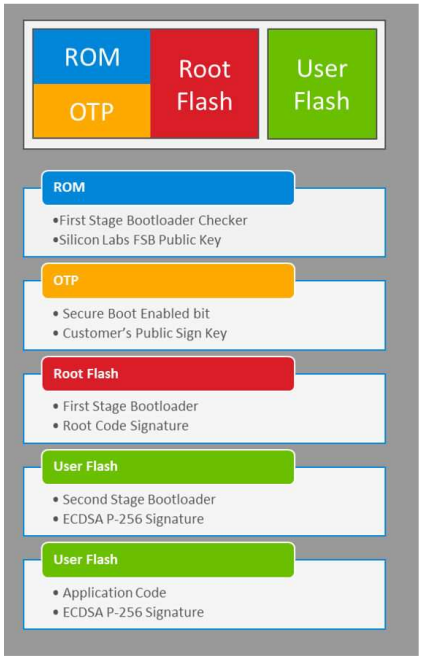
3

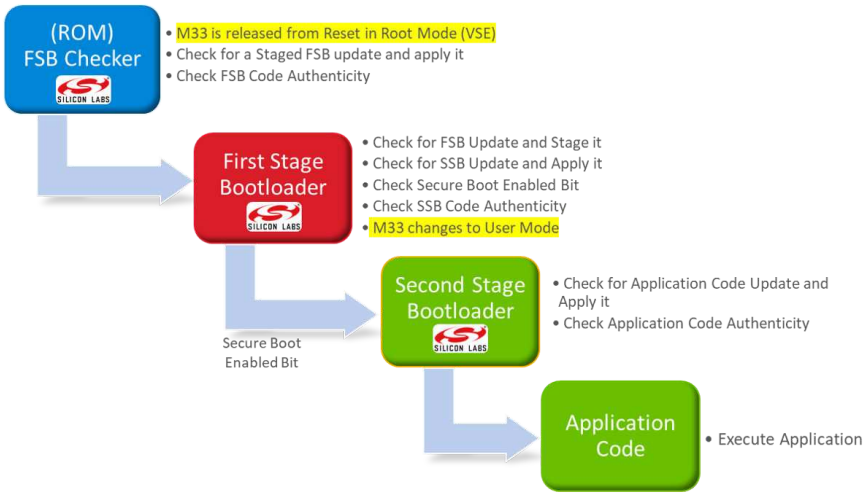
VSE

In VSE-SVM devices, the host MCU (Cortex-M33) assumes an elevated security state out of reset and securely boots itself from code that originates in ROM. The following figures illustrate the Secure Boot process and flow on Series 2 VSE devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.







Certificate-based Secure Boot in Series 2 and Series 3 Devices

Refer to the *Gecko Bootloader Security Features* section in UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) and Certificate-Based Secure Boot example for details about the certificate-based Secure Boot process in Series 2 and Series 3 devices.

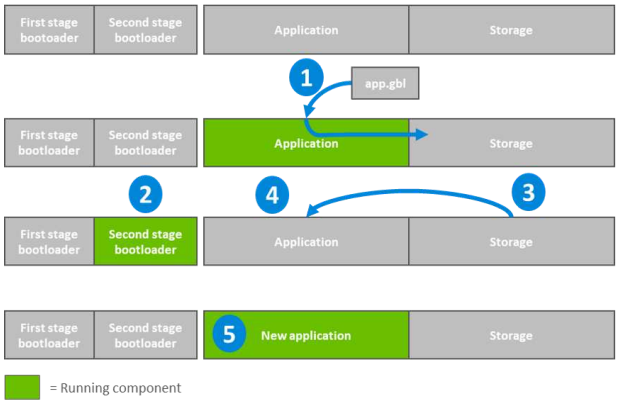
The certificate-based Secure Boot process uses key delegation to minimize the exposure of the Private Sign Key, thereby reducing the likelihood that the corresponding Public Sign Key would need to be revoked.

If the certificate's private key is leaked, all devices that have been programmed with that certificate are at risk until they can be updated with an image containing a certificate with a [higher version](./03-examples.md#certificate-revocation(03-examples.md#certificate-revocation).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Loader

In Series 2 and Series 3 devices, the Secure Loader is firmware pre-loaded into the chip. Silicon Labs maintains the Secure Loader and deploys through secure upgrade packages. It is the functional equivalent of the first-stage GBL on Series 1 devices (refer to UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) for more information). The Secure Loader validates the authenticity and integrity of a staged image before performing an upgrade operation. The Secure Loader requires the staged image to reside on-chip and the staged image must not overlap with the target destination address range. Firmware images that originate from off-chip, either off-chip storage, external NCP/ RCP host interface, or through an OTA update procedure are expected to be staged either by the application or by SSB before calling the Secure Loader to perform the upgrade.



1. The existing application downloads the upgrade image onto the storage medium of the device (for example, the app.gbl file through OTA).
2. The device reboots into the second stage bootloader, and the bootloader enters firmware upgrade mode.
3. The secure loader in the second stage bootloader verifies the integrity and authenticity of the staged upgrade image in storage, prior to performing the upgrade.
4. The second stage bootloader applies the application upgrade from the upgrade image.
5. The device boots into the application. Application upgrade is complete.

Secure Boot Time

Secure boot extends the recovery time from all sources of device reset. The duration of each authentication operation depends on the factors below:

- Computation of the SHA-256 digest (32 bytes) of the associated image, which is proportional to the size of the firmware image.
- Verification of the ECDSA-P256 signature of the SHA-256 digest above, which is independent of image size.
- The clock frequency of the Crypto Engine, which is the HSE or CRYPTOACC in VSE devices.

Authentication	Enable/Disable	Duration
FSB code	Enable (cannot disable)	FSB code size dependent
SSB code	Disable (by default)	SSB code size and SE firmware version dependent
Application code	Disable (by default)	Application code size and SSB firmware (GBL) version dependent

Notes:

- It will extend the boot time for certificates authentication if using Certificate-Based Secure Boot.
- Refer to device-specific datasheets (like EFR32MG21B) for data about the boot timing of Series 2 devices.
- Refer to *Secure Boot Configuration* below on how to enable the SSB and application code authentication.

Secure Boot Configuration

The following sections describe how to configure the Secure Boot of the SSB (GBL) and application firmware.

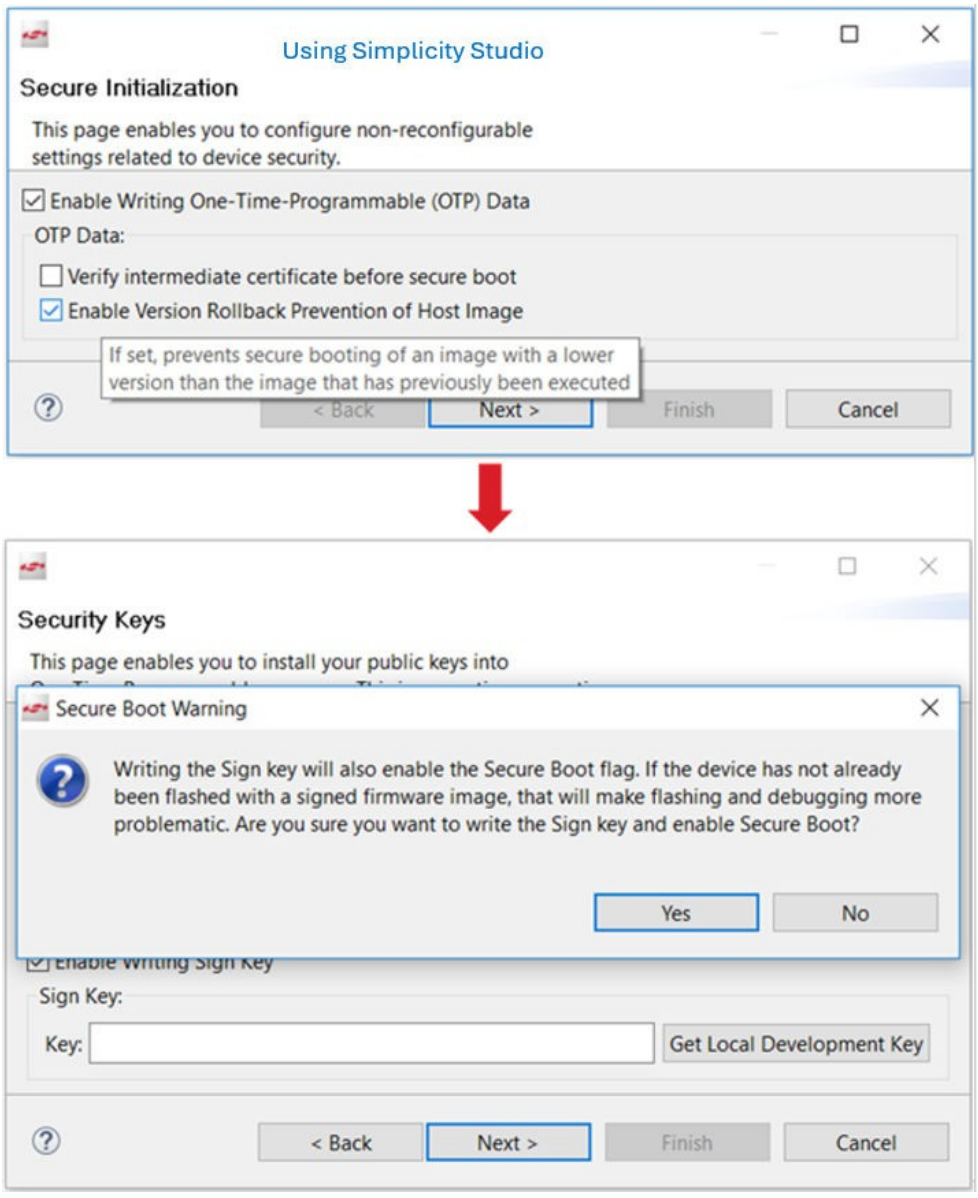
SSB

In Series 2 and Series 3 devices, the immutable OTP memory stores the Public Sign Key, Secure Boot Enable flag, and Anti-rollback flag. The user cannot change its respective value once either is programmed. Once the Public Sign Key is provisioned, it remains provisioned to that key value for the life of the device. Once Secure Boot and Anti-rollback is enabled, it remains enabled for the life of the device. Both of these assignment operations are IRREVOCABLE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Public Sign Key used for Series 2 and Series 3 devices is the public portion of an ECDSA key pair over the NIST prime curve P-256. The Public Sign Key is a customer key and is typically provisioned during the initial product manufacturing and device programming phase. It is common for all products that share the same firmware image to be loaded with the same Public Sign Key. The key loaded into the device is a public key and has no confidentiality requirements. The private key associated with that public key, which will be used to sign firmware images or certificates, should be tightly held, ideally secured in the HSM or equivalent key storage instrument.

You can use Simplicity Commander, SE Manager, or Simplicity Studio to program the Public Sign Key and configure the SSB Secure Boot with anti-rollback in SE OTP. The following figures show the Simplicity Studio and Simplicity Commander implementation, for detailed steps refer to the Example section:



Copyright © 2025 Silicon Laboratories. All rights reserved.



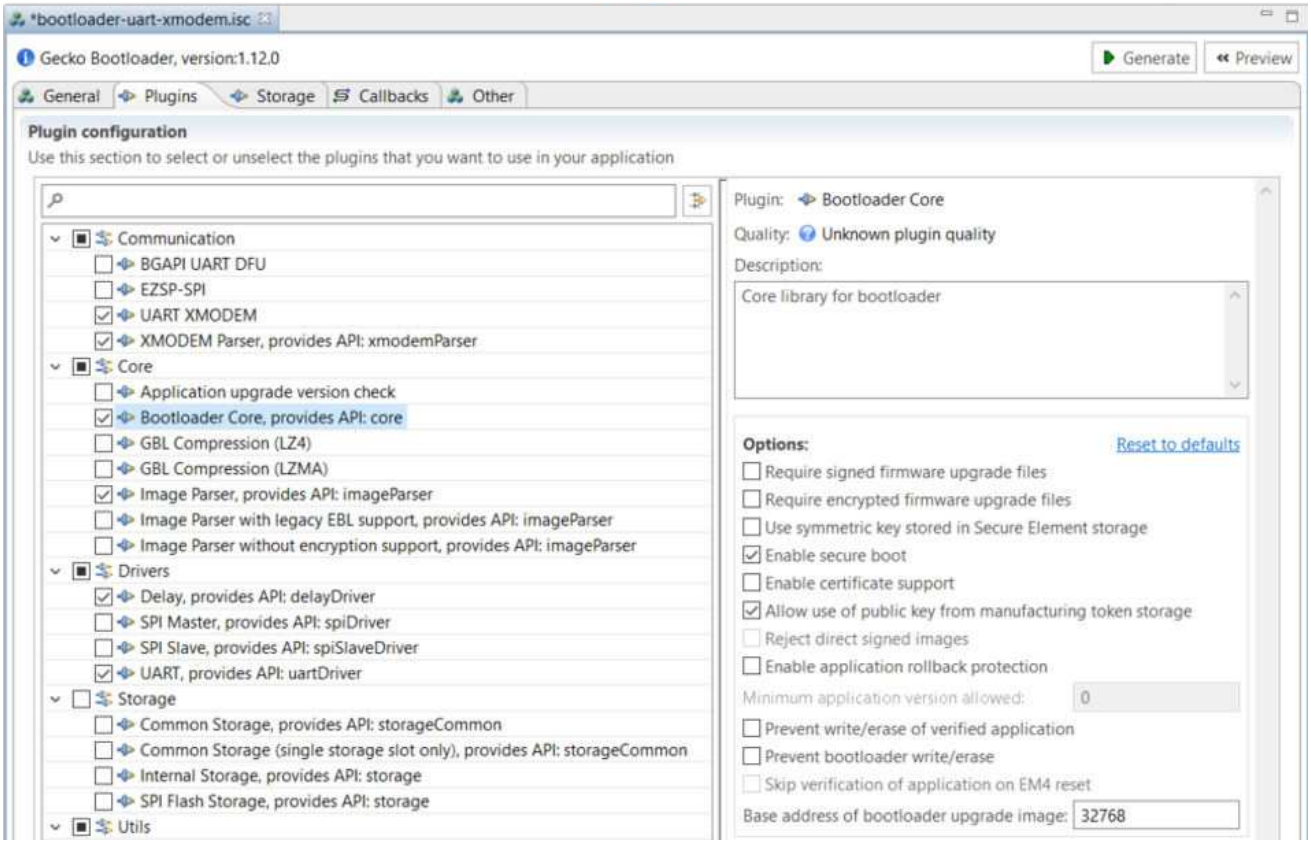
Application Firmware

You can use the AppBuilder or Bootloader-core software component in the GBL project to configure the security options of the application firmware.

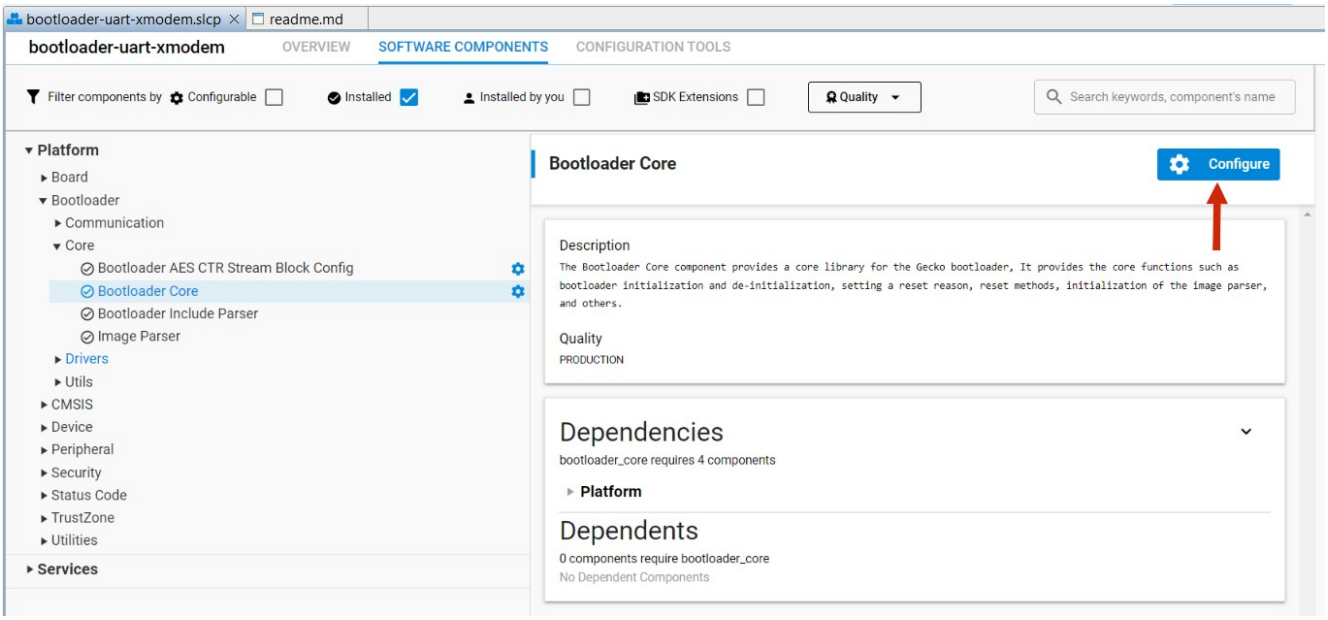
You can reconfigure the Secure Boot configuration of the application firmware by upgrading the GBL with the new custom settings.

Copyright © 2025 Silicon Laboratories. All rights reserved.

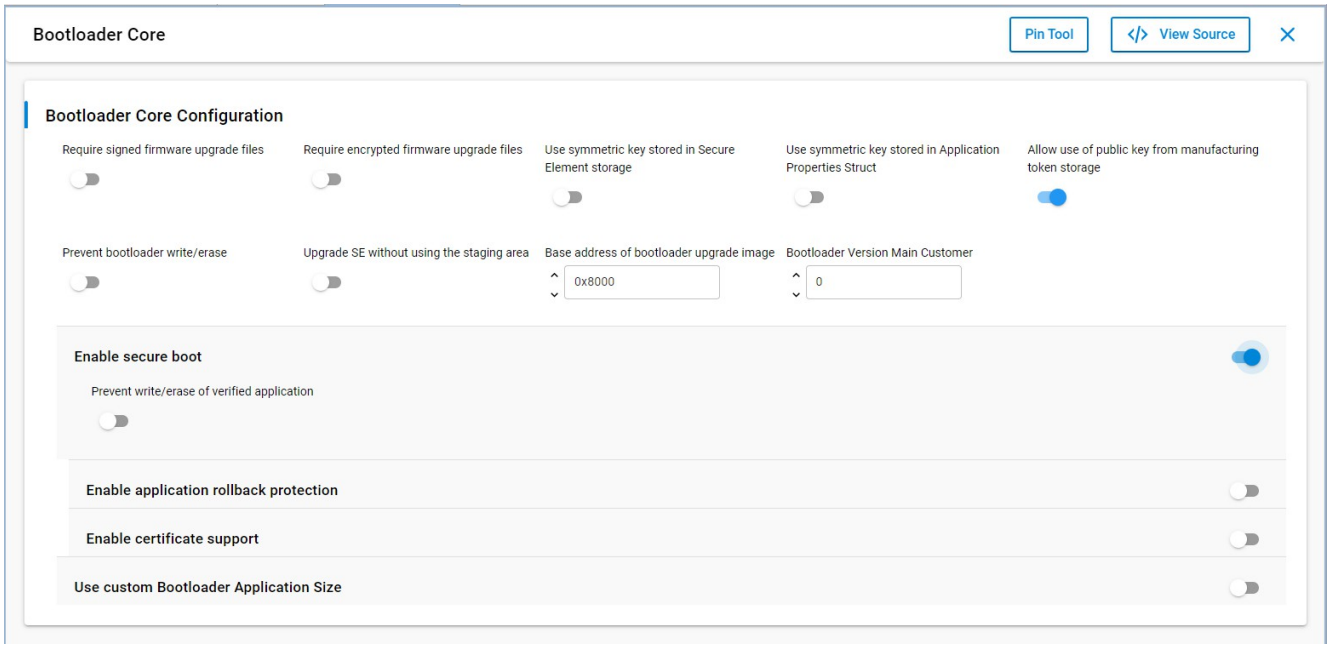
Using AppBuilder



Using Bootloader core compoent



Copyright © 2025 Silicon Laboratories. All rights reserved.



> Note: Install "Bootloader Core (Series-3)" for series 3 devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Overview

The following table describes the examples for Series 2 and Series 3 Secure Boot.

> Note: Series 3 devices require SE firmware Version 3.1.0 or later to support Secure Boot. However, version 3.3.4 or newer is recommended for full feature support and latest security enhancements.

Example	Device (Radio Board)	Minimum SE Firmware	Tool
Provision Public Sign Key and Secure Boot Enabling	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Provision GBL Decryption Key	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
Signing for ECDSA-P256-SHA256 Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Signing for Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Generate a GBL Upgrade Image File	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Certificate Revocation	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Secure Boot with RTSL	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager & Simplicity Commander
Recover Devices when Secure Boot Fails	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander

> Note: Unless specified in the example, these examples can be applied to other Series 2 and Series 3 devices.

Using Simplicity Commander

1.

Copyright © 2025 Silicon Laboratories. All rights reserved.

This application note uses Simplicity Commander v1.19.2. The procedures and console output may be different for the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from MCU Programming options.

> Note: It is recommended to upgrade to the latest version of Simplicity Commander.

```
commander --version
```

```
Simplicity Commander 1v19p2b1907

JLink DLL version: 7.52d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p19b0
mbed TLS version: 2.16.6

Emulator found with SN=440048205 USBAddr=0

DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of commander.exe to the system PATH in Windows.
3. If more than one WSTK is connected via USB, the target WSTK must be specified using the `--serialno <J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device <device name>` option. For more information about Simplicity Commander, see Simplicity Commander Reference Guide.

Using an External Tool

The Secure Boot examples use the OpenSSL to sign the image files and certificates. The Windows version of OpenSSL can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. This application note uses OpenSSL Version 3.5.0 (Win64).

```
openssl version
```

```
OpenSSL 3.5.0 8 Apr 2025 (Library: OpenSSL 3.5.0 8 Apr 2025)
```

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

Using a Platform Example

Simplicity Studio 5 includes the SE Manager platform example for key provisioning and Secure Boot enabling. This application note uses platform example of SiSDK v2025.6.1. The console output may be different on other versions.

Refer to the corresponding readme file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

Generate Key and Signing

This section describes how to generate a key to sign an image file or certificate for Secure Boot using Simplicity Commander or using an HSM with Simplicity Commander (Openssl). For production environments, it is strongly recommended to generate keys with an HSM.

Using Simplicity Commander

1. Run the `util genkey` command to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem)` and Public Sign Key token file (`sign_pubkey.txt`). The Simplicity Commander can program the Public Sign Key in token file

Copyright © 2025 Silicon Laboratories. All rights reserved.

(`sign_pubkey.txt`) to the top page of the main flash.

```
commander util genkey --type ecc-p256 --privkey sign_key.pem --pubkey sign_pubkey.pem
--tokenfile sign_pubkey.txt
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to sign_key.pem
Writing public key file in PEM format to sign_pubkey.pem
Writing EC tokens to sign_pubkey.txt...
DONE
```

> Note: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the convert command with the Private Key (like sign_key.pem) from step 1 to sign an image file or certificate. Refer to Signing for ECDSA-P256-SHA256 Secure Boot and Signing for Certificate-Based Secure Boot for more information about the Simplicity Commander signing process.

Using an HSM and Simplicity Commander

1. You can use HSM to generate the ECDSA-P256 Sign Key pair. The Private Sign Key is securely held in HSM and the Public Sign Key can be exported in a specific format (like `sign_pubkey.pem`).

> Note: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the `util keytotoken` command to convert the Public Sign Key from step 1 to token format (`sign_pubkey.txt`). The Simplicity Commander can program the Public Sign Key in token file (`sign_pubkey.txt`) to the top page of the main flash.

```
commander util keytotoken sign_pubkey.pem --outfile sign_pubkey.txt
```

```
Writing EC tokens to sign_pubkey.txt...
DONE
```

3. Use the `convert` command with `--extsign` option to prepare an unsigned image or certificate for HSM.
4. Use the Private Key from step 1 to generate a signature for the unsigned image or certificate from step 3.
5. Use the `convert` command with the signature from step 4 to generate a signed image or certificate. Refer to Signing for ECDSA-P256-SHA256 Secure Boot and Signing for Certificate-Based Secure Boot for more information about the HSM and Simplicity Commander signing process.

> Note: The Simplicity Commander v1.11.0 or above supports signature in DER format. The older version of Simplicity Commander can only handle signatures in Raw format.

Provision Public Sign Key and Secure Boot Enabling

The Public Sign Key in SE OTP is used to verify the host image signature or certificate during Secure Boot. You should provision this key before setting the Secure Boot enabled flag in SE OTP. On HSE-SVH devices, when anti-tamper functionality is needed, you need to provision the anti-tamper protection configuration with Secure Boot settings.

If you set the SECURE_BOOT_ANTI_ROLLBACK option with Secure Boot, the SE will store the version counter (4 bytes) for anti-rollback of GBL (SSB) to SE flash and check the bootloader version during an upgrade and on every boot. The version counter will

Copyright © 2025 Silicon Laboratories. All rights reserved.

not roll to 0 if it reaches the maximum value (bootloader cannot be upgraded anymore). The anti-rollback does not prevent flashing an older signed GBL hex image to the device.

The following table describes the anti-rollback protection on signed GBL when `SECURE_BOOT_ANTI_ROLLBACK` is enabled or disabled.

- The GBL handles the anti-rollback protection when upgrading the GBL through the GBL upgrade image file (`.gbl`).
- The SE handles the anti-rollback protection (if `SECURE_BOOT_ANTI_ROLLBACK` enabled) when booting the image.

Action	SECURE_BOOT_ANTI_ROLLBACK Disable	SECURE_BOOT_ANTI_ROLLBACK Enable
Use a GBL upgrade image (`.gbl`) file (e.g., OTA)	Gecko Bootloader rejects upgrade if an equal or lower GBL version is detected.	Gecko Bootloader rejects upgrade if an equal or lower GBL version is detected.
Flash and boot a firmware image (`.s32` , `.hex` , `.bin`) directly (e.g., Simplicity Commander)	SE accepts the image to flash and boot regardless of the firmware image version.	The SE accepts and boots the firmware image if its version is equal to or higher than the stored version. A lower version will be blocked at boot.

> Note: [Series 2 devices only] The device needs to execute a mass erase (`commander device masserase` or `commander security erasedevice` then reset) before flashing a GBL hex image (`.s37`) to the device if `SECURE_BOOT_PAGE_LOCK_NARROW` or `SECURE_BOOT_PAGE_LOCK_FULL` option in SE OTP is enabled.

For simplicity, the Secure Boot examples in this application note do not enable the following options for Secure Boot.

- `SECURE_BOOT_PAGE_LOCK_NARROW`
- `SECURE_BOOT_PAGE_LOCK_FULL`

> Note: `SECURE_BOOT_PAGE_LOCK_NARROW` and `SECURE_BOOT_PAGE_LOCK_FULL` are only available on Series 2 devices.

Simplicity Commander

The following procedures assume the required files are in the same folder.

1. Follow the procedures in Generate Key and Signing to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) and Public Sign Key token file (`sign_pubkey.txt`).
2. Run the `security writekey` command to provision the Public Sign Key (`sign_pubkey.pem`) to the SE OTP slot. The Public Sign Key cannot be changed once written.

```
commander security writekey --sign sign_pubkey.pem --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device has serial number 000000000000000014b457fffe045b21

================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, all code to be run on the device must be signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

3. Run the `security readkey` command to verify the Public Sign Key with the Public Sign Key in the token file (`sign_pubkey.txt`).

```
commander security readkey --sign --device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
C4AF4AC69AAB9512DB50F7A26AE5B4801183D85417E729A56DA974F4E08A562C
DE6019DEA9411332DC1A743372D170B436238A34597C410EA177024DE20FC819
DONE
```

4. For Series 2 VSE devices (like EFR32MG22C224F512IM40), run the `flash` command to program the Public Sign Key in the token file (`sign_pubkey.txt`) to the top page of the main flash for ECDSA-P256-SHA256 Secure Boot. It is optional on Series 2 HSE and Series 3 devices.

```
commander flash --tokengroup znet --tokenfile sign_pubkey.txt --device EFR32MG22C224F512IM40 --serialno
440048205
```

```
Writing 8192 bytes starting at address 0x0007e000
Comparing range 0x0007E000 - 0x0007FFFF (8 KiB)
Programming range 0x0007E000 - 0x0007FFFF (8 KiB)
DONE
```

> Note: The MCU Series 2 devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the `flash --tokengroup znet` command.

5. Run the `security genconfig` command to generate the `user_configuration.json` file for secure boot.

```
commander security genconfig --nostore --outfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
DONE
```

Name	Description
SECURE_BOOT_ENABLE	If set, verifies the host image on the Cortex-M33 before releasing the Cortex-M33 from reset.
SECURE_BOOT_VERIFY_CERTIFICATE	If set, requires certificate-based signing of the host image.
SECURE_BOOT_ANTI_ROLLBACK	If set, prevents secure upgrading to a host image with an equal or lower version than the image that is currently stored in flash.
SECURE_BOOT_PAGE_LOCK_NARROW	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, not including the last page if the signature is not on a page boundary.
SECURE_BOOT_PAGE_LOCK_FULL	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, including the last page if the signature is not on a page boundary.

> Notes: The host image is the firmware in the device's flash starting address. It is usually the GBL.
>
> SECURE_BOOT_PAGE_LOCK_NARROW and SECURE_BOOT_PAGE_LOCK_FULL are not available in Series 3 devices.

6. Use a text editor to modify the default secure boot settings to the desired configurations used in this application note.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
"mcu_flags": {
        "SECURE_BOOT_ANTI_ROLLBACK": true,
        "SECURE_BOOT_ENABLE": true,
        "SECURE_BOOT_PAGE_LOCK_FULL": false,
        "SECURE_BOOT_PAGE_LOCK_NARROW": false,
        "SECURE_BOOT_VERIFY_CERTIFICATE": false
    }
```

ECDSA-P256-SHA256 Secure Boot

```
"mcu_flags": {
        "SECURE_BOOT_ANTI_ROLLBACK": true,
        "SECURE_BOOT_ENABLE": true,
        "SECURE_BOOT_PAGE_LOCK_FULL": false,
        "SECURE_BOOT_PAGE_LOCK_NARROW": false,
        "SECURE_BOOT_VERIFY_CERTIFICATE": true
    }
```

Certificate-based Secure Boot

> Note: For devices prior to EFR32xG23, if `SECURE_BOOT_ENABLE` is set to false, the SE ignores the other four options regardless of their values, whereas on EFR32xG23 and future Series 2 devices, this configuration cannot be programmed to the SE OTP.

7. Follow the procedure in Signing for ECDSA-P256-SHA256 Secure Boot to flash a bootloader image for signature based secure boot.
   OR
   Follow the procedure in Signing for Certificate-Based Secure Boot to flash a bootloader image for certificate based secure boot.
8. Run the `security writeconfig` command to program the secure boot configuration to the SE OTP. You can execute this command once per device.

```
commander security writeconfig --configfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
```

```
================================================================================

THIS IS A ONE-TIME configuration: Please inspect file before confirming: user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

9. Run the `security readconfig` command to check the secure boot configuration of the device.

```
commander security readconfig --serialno 440048205
```

```
MCU Flags
Secure Boot                      : Enabled
Secure Boot Verify Certificate   : Disabled
Secure Boot Anti Rollback        : Enabled
Secure Boot Page Lock Narrow     : Disabled
Secure Boot Page Lock Full       : Disabled
DONE
```

ECDSA-P256-SHA256 Secure Boot

```
MCU Flags
Secure Boot                      : Enabled
Secure Boot Verify Certificate   : Enabled
Secure Boot Anti Rollback        : Enabled
Secure Boot Page Lock Narrow     : Disabled
Secure Boot Page Lock Full       : Disabled
DONE
```

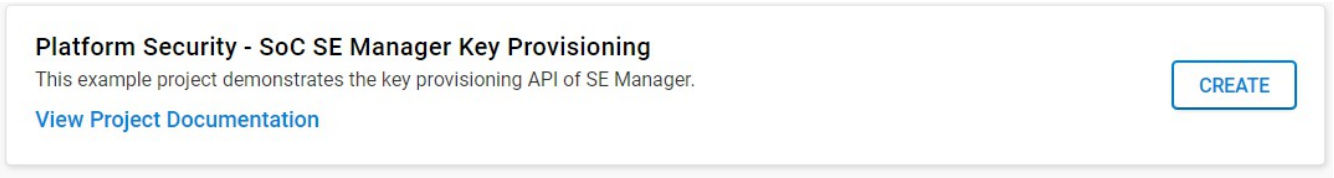
Certificate-based Secure Boot

> Note: Enabling Anti-rollback during development involves updating the firmware version every time you decide to flash a new image, this adds complexity. Therefore, it is recommended to enable Anti-rollback during production.

SE Manager Key Provisioning Platform Example

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. Click the View Project Documentation link to open the readme file.



2. Modify the default Public Sign Key in `public_sign_key[PUB_KEY_SIZE]` array in `app_process.c` to the desired values.

```
/// Public sign key
SL_ALIGN(4) static const uint8_t public_sign_key[PUB_KEY_SIZE] = {
  0xc4, 0xaf, 0x4a, 0xc6, 0x9a, 0xab, 0x95, 0x12,
0xdb, 0x50, 0xf7, 0xa2, 0x6a, 0xe5, 0xb4, 0x80,
0x11, 0x83, 0xd8, 0x54, 0x17, 0xe7, 0x29, 0xa5,
0x6d, 0xa9, 0x74, 0xf4, 0xe0, 0x8a, 0x56, 0x2c,
0xde, 0x60, 0x19, 0xde, 0xa9, 0x41, 0x13, 0x32,
0xdc, 0x1a, 0x74, 0x33, 0x72, 0xd1, 0x70, 0xb4,
0x36, 0x23, 0x8a, 0x34, 0x59, 0x7c, 0x41, 0x0e,
0xa1, 0x77, 0x02, 0x4d, 0xe2, 0x0f, 0xc8, 0x19
};
```

3. Modify the default secure boot settings in `init_se_otp_conf()` function in `app_se_manager_key_provisioning.c` to the desired configuration.

```
// Overwrite secure boot options in SL_SE_OTP_INIT_DEFAULT if necessary.
otp_init.enable_secure_boot = true;
otp_init.verify_secure_boot_certificate = false;
otp_init.enable_anti_rollback = true;
otp_init.secure_boot_page_lock_narrow = false;
otp_init.secure_boot_page_lock_full = false;
```

> Note: If `enable_secure_boot` is false, the SE will ignore the other four options regardless of whether they are true or false. The EFR32xG23 and higher Series 2 devices do not allow this setting to program to the SE OTP.

4. Follow the procedures in Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
5. Build the project and run the application. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot if a signed application image is required.
6. Then press SPACE to skip the programming of the AES-128 key (HSE devices only). Optional to press ENTER to program the hard-coded GBL Decryption Key to HSE OTP.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)

. Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
+ Current SE firmware version (MSB..LSB): 00010209

. Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)

. Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.

. Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_FAIL (cycles: 4627 time: 121 us)

. Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

7. Press ENTER to program the hard-coded Public Sign Key to SE OTP.

```
+ Warning: The public sign key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

8. Press ENTER to confirm the operation. The SE returns `SL_STATUS_INVALID_PARAMETER` if the Public Sign Key is present in SE OTP.

```
. Initialize public sign key... SL_STATUS_OK (cycles: 56052 time: 1475 us)

. Get public sign key... SL_STATUS_OK (cycles: 8450 time: 222 us)
+ The public sign key (64 bytes):
C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

```
. Initialize public sign key... SL_STATUS_INVALID_PARAMETER (cycles: 4375 time: 115 us)

. Get public sign key... SL_STATUS_OK (cycles: 8435 time: 221 us)
+ The public sign key (64 bytes):
C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

9. Press SPACE to skip the programming of the Public Command Key. Optional to press ENTER to program the hard-coded Public Command Key to SE OTP.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

10. Press ENTER to program the secure boot configuration.

```
+ Warning: The SE OTP configuration cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

11. Press ENTER to confirm the operation. The SE returns `SL_STATUS_COMMAND_IS_INVALID` if an invalid setting from step 2 or the secure boot configuration has been programmed in SE OTP.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Initialize SE OTP... SL_STATUS_OK (cycles: 267256 time: 7033 us)
+ Read SE OTP configuration... SL_STATUS_OK (cycles: 6865 time: 180 us)
+ Secure boot                   : Enabled
+ Secure boot verify certificate : Disabled
+ Secure boot anti-rollback      : Enabled
+ Secure boot page lock narrow   : Disabled
+ Secure boot page lock full     : Disabled

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

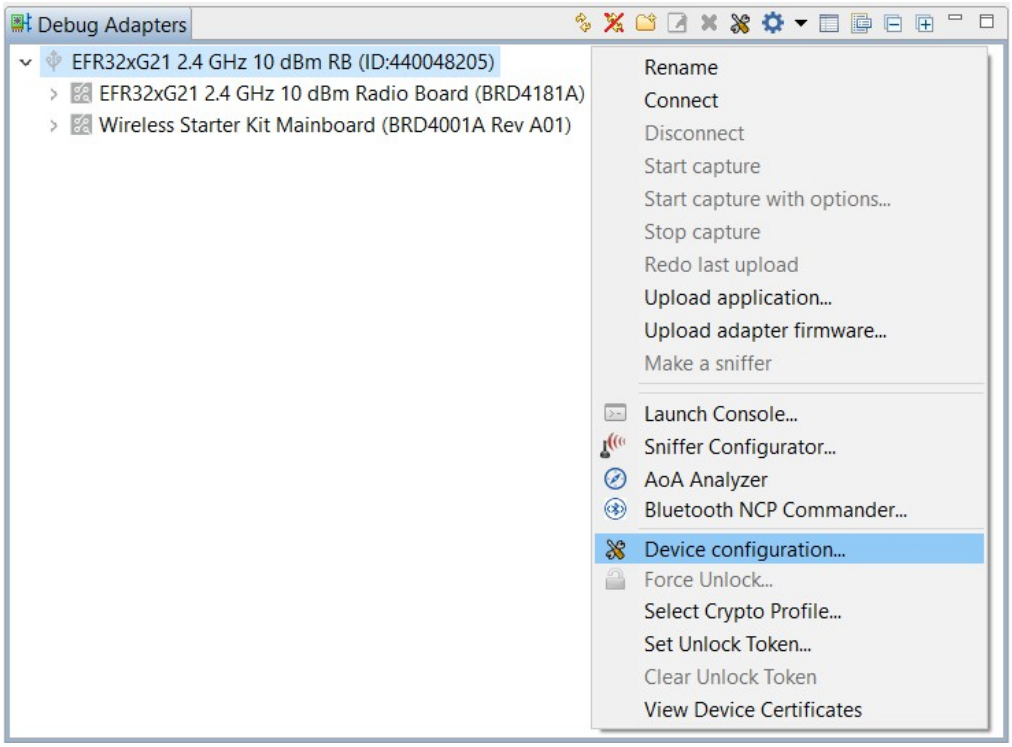
```
. Initialize SE OTP... SL_STATUS_COMMAND_IS_INVALID (cycles: 3989 time: 104 us)

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

Simplicity Studio

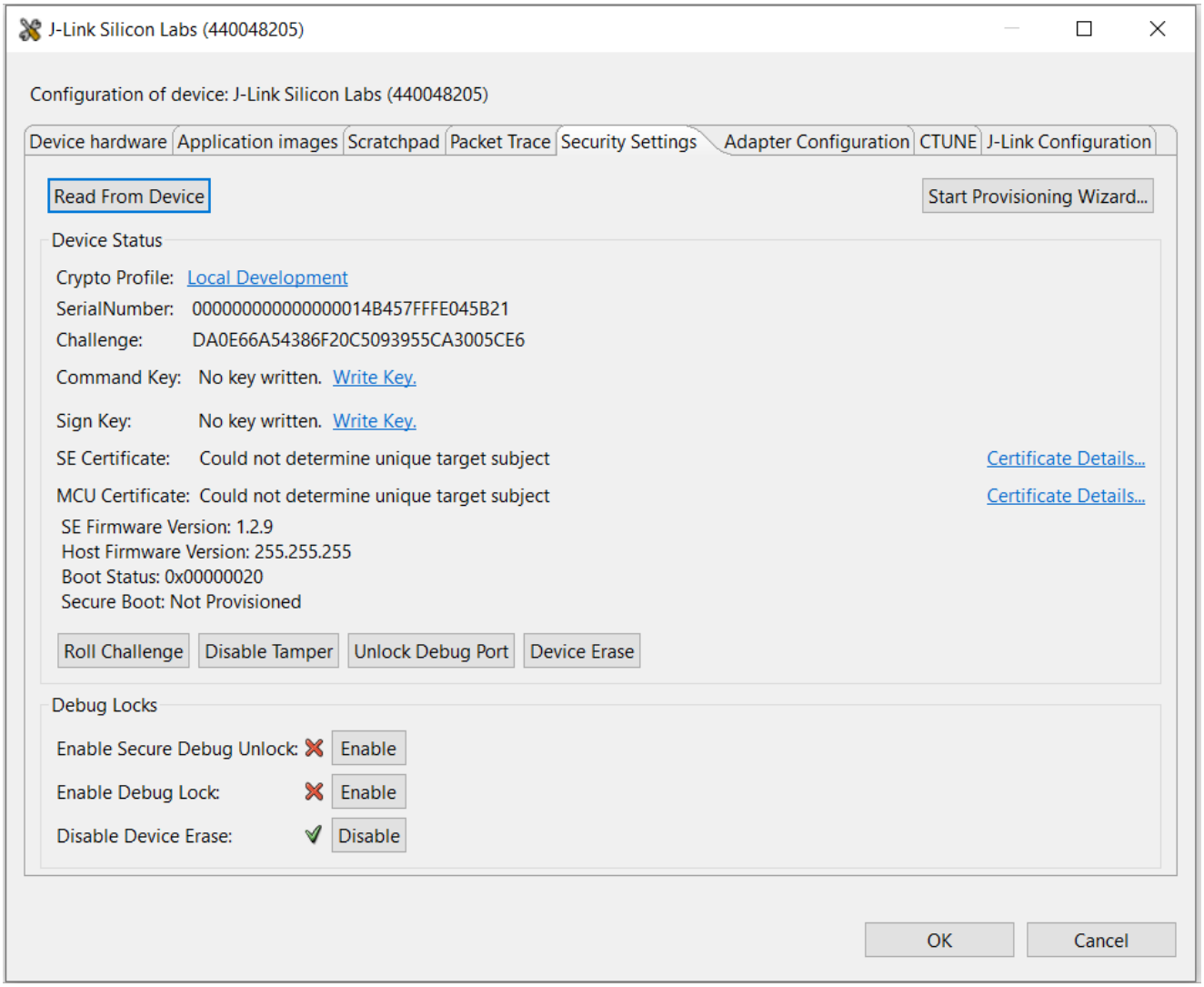
The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.2.3.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. Right-click the selected debug adapter RB (ID:J-Link serial number) to display the context menu.



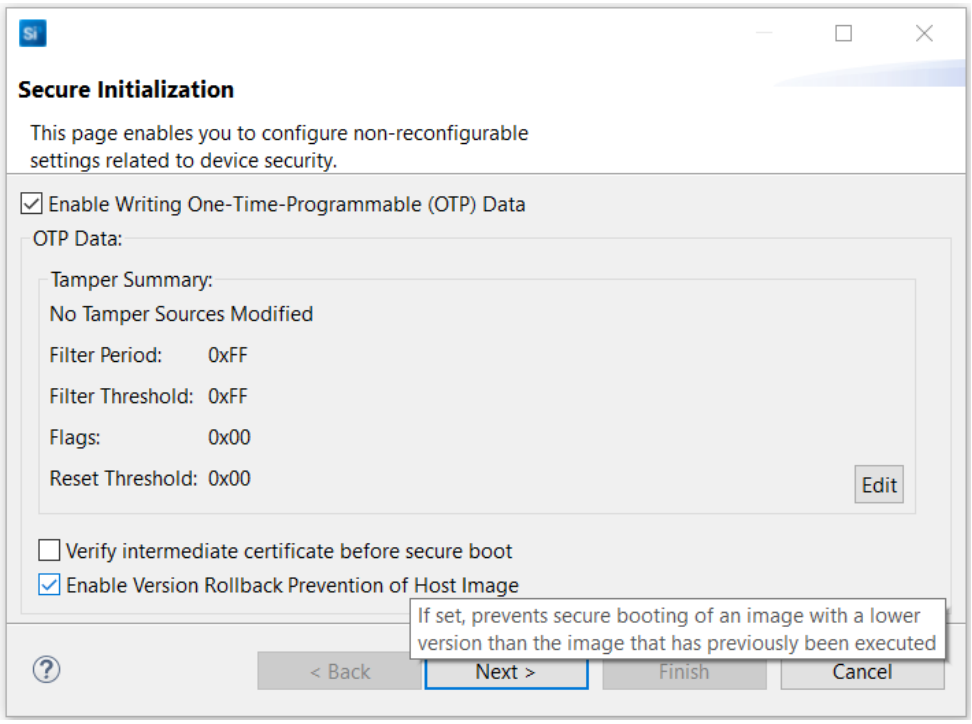
2. Click Device configuration... to open the Configuration of device: J-Link Silicon Labs (serial number) dialog box. Click the Security Settings tab to get the selected device configuration.

Copyright © 2025 Silicon Laboratories. All rights reserved.



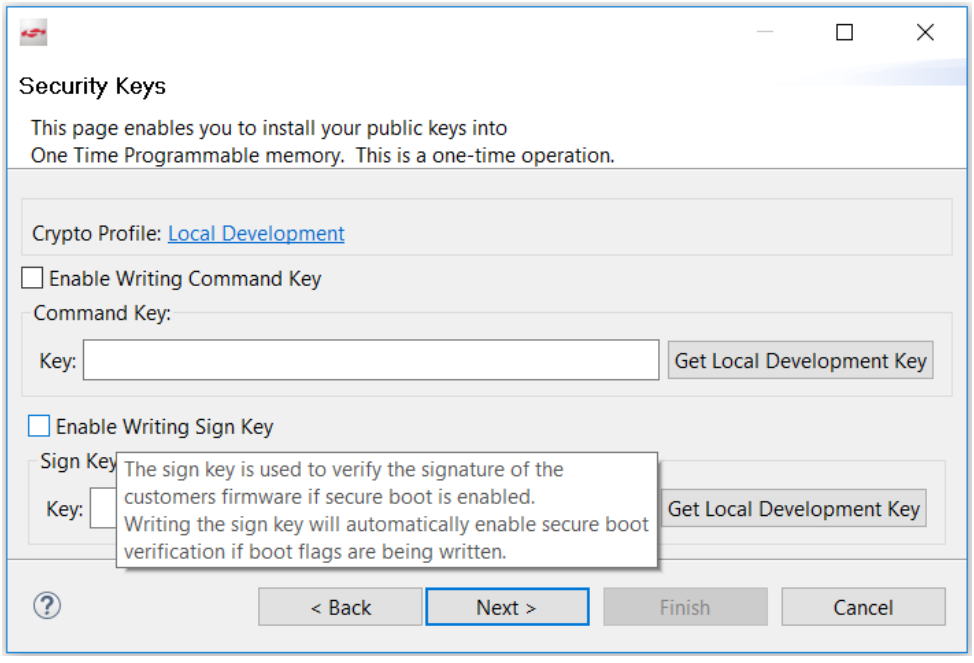
3. Click [Start Provisioning Wizard…] in the upper right corner to display the Secure Initialization dialog box. Checking the Enable Version Rollback Prevention of Host Image option is recommended. The Verify intermediate certificate before secure boot option is for Certificate-based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.



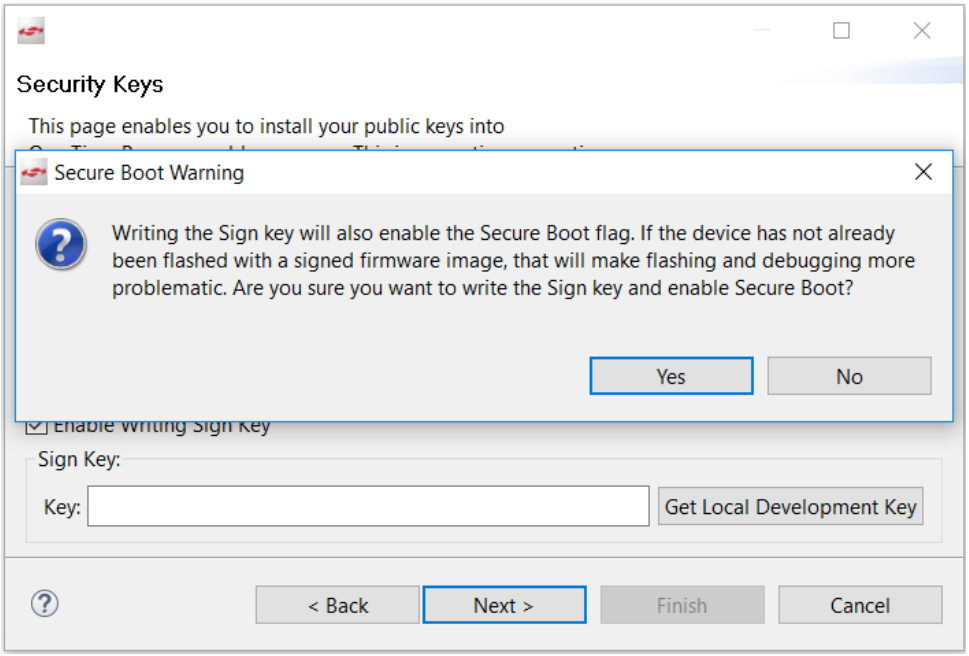
> Note: The SECURE_BOOT_PAGE_LOCK_NARROW and SECURE_BOOT_PAGE_LOCK_FULL options are not yet available in Simplicity Studio.

4. Click [Next >]. The Security Keys dialog box is displayed.



5. Checking the Enable Writing Sign Key checkbox will automatically enable Secure Boot. The following Secure Boot Warning is displayed. Click [Yes] to confirm.

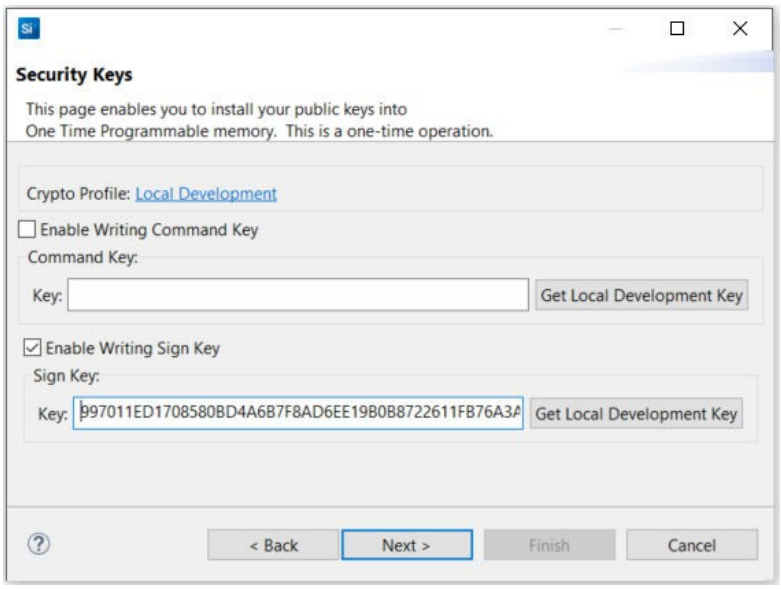
Copyright © 2025 Silicon Laboratories. All rights reserved.



6. Open the Public Sign Key token file (`sign_pubkey.txt`).

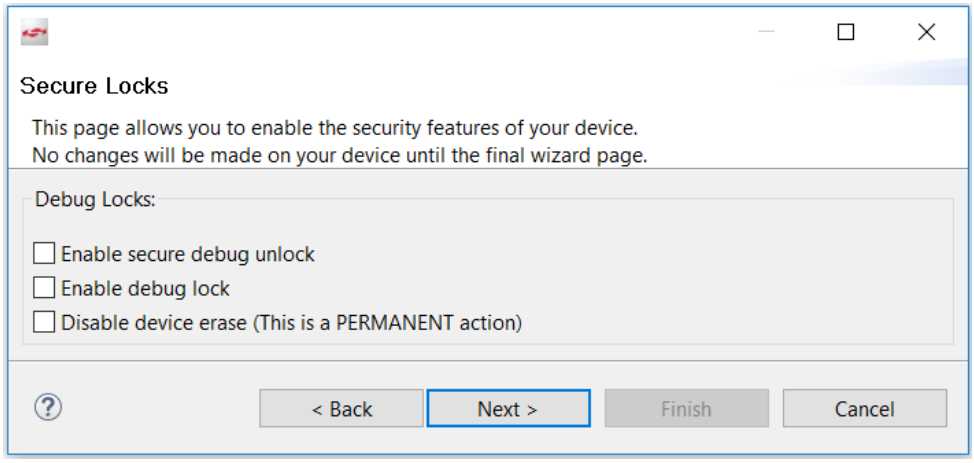
```
MFG_SIGNED_BOOTLOADER_KEY_X : 997011ED1708580BD4A6B7F8AD6EE19B0B8722611FB76A3A5702D5141180E101
MFG_SIGNED_BOOTLOADER_KEY_Y : 0AC8673C8ACC26EE2B534C004F4A4B7EBBC23D04506DD66E3EF0DDC81E3CA55E
```

7. Copy Public Sign Key (X-point `9970...` first, then Y-point `0AC8...`) to Key: box under Sign Key:.

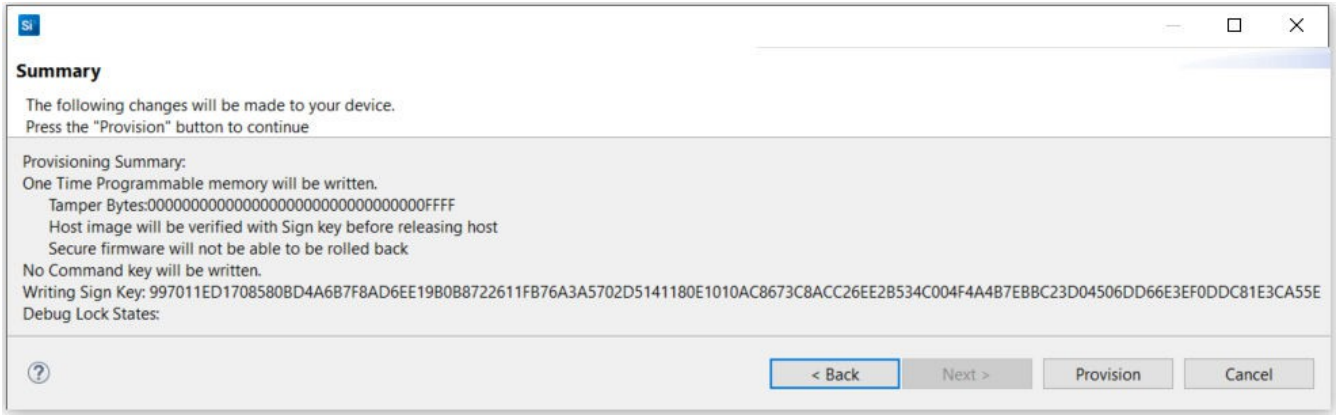


8. Click [Next >]. The Secure Locks dialog box is displayed. When Secure Boot is enabled, the Debug Locks are not set by default. Refer to Series 2 and Series 3 Secure Debug for more information about these locks.

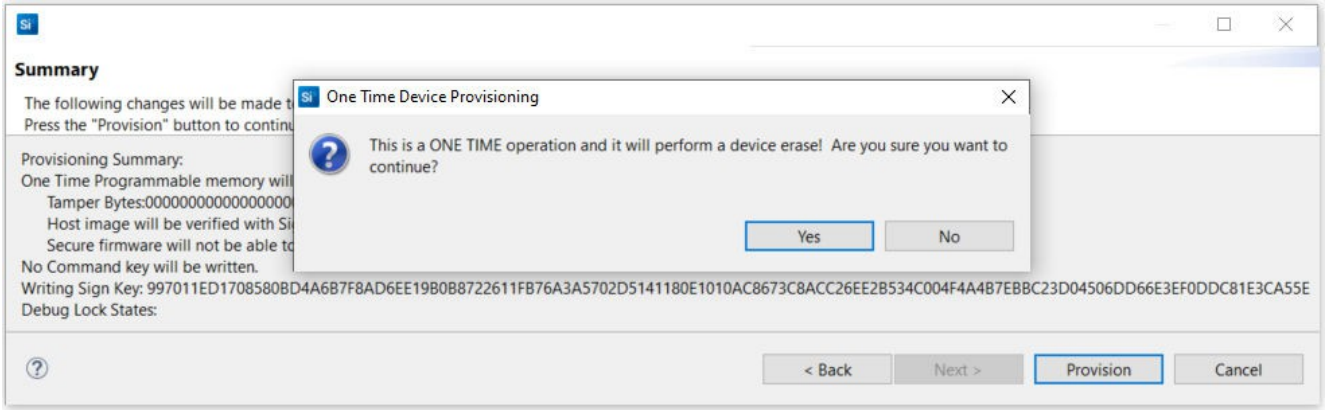
Copyright © 2025 Silicon Laboratories. All rights reserved.



9. Click [Next >] to display the Summary dialog box.

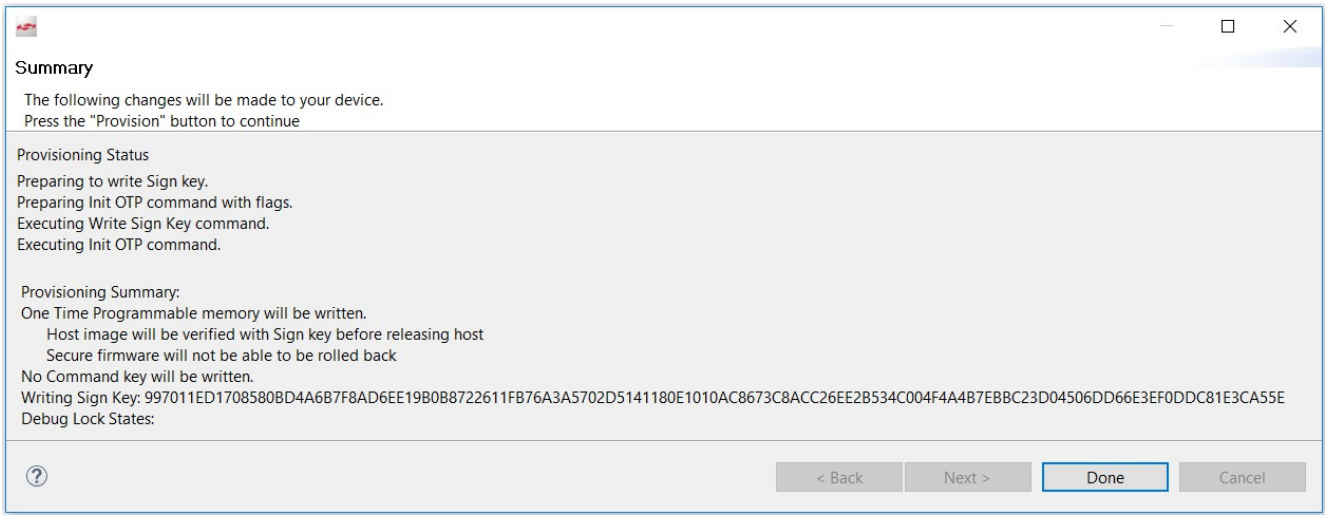


10. If the information displayed is correct, click [Provision]. Click [Yes] to confirm. The Public Sign Key and Secure Boot configuration cannot be changed once written.



1. The Summary dialog box displays the Provisioning Status.

Copyright © 2025 Silicon Laboratories. All rights reserved.

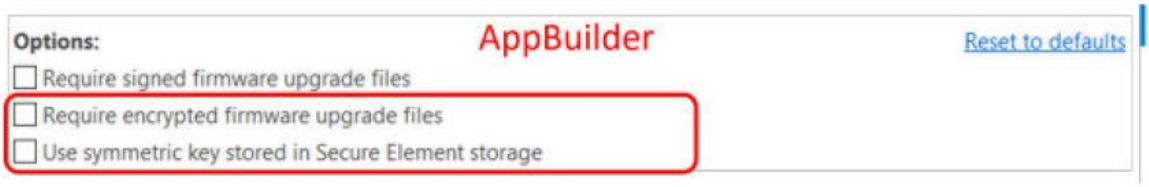


1. Click [Done] to exit the provisioning process. The device configuration is updated.

Provision GBL Decryption Key

The GBL Decryption Key is used to decrypt the GBL upgrade image file payloads during firmware upgrade. You should provision this key before enabling the Require encrypted firmware upgrade files option in AppBuilder plugin and 3.4.1.2 Bootloader-core Software Component.

The following figures and table show the available options, two for VSE and three for HSE, for selecting the GBL Decryption Key used to decrypt the GBL upgrade image (bootloader, SE, or application).





Option for GBL Decryption Key Selection	GBL Decryption Key Storage
1. Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0).	The 128-bit symmetric key stored in HSE OTP is used for GBL upgrade image file decryption.
2. Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1).	The 128-bit symmetric key stored in the GBL Application Properties Struct is used for GBL upgrade image file decryption. The key is stored in the Secure flash if TrustZone is implemented.
3. Default storage if none of the above options are selected.	The 128-bit symmetric key stored on the top page of the main flash is used for GBL upgrade image file decryption. The key is stored in the Non-secure flash if TrustZone is implemented.

Note:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Option 2 requires `ApplicationProperties_t` struct v1.2 or higher (defined in `application_properties.h` in the Windows folder `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`) in GSDK v4.1 or higher to store the GBL Decryption Key.

```
/// Major version number of the ApplicationProperties_t struct #define APPLICATION_PROPERTIES_VERSION_MAJOR (1UL)
/// Minor version number of the ApplicationProperties_t struct #define APPLICATION_PROPERTIES_VERSION_MINOR (2UL)
```

```
/// Application Properties struct typedef struct {
/// @brief Magic value indicating this is an ApplicationProperties_t struct.
/// Must equal @ref APPLICATION_PROPERTIES_MAGIC uint8_t magic[16];
/// Version number of this struct uint32_t structVersion;
/// Type of signature this application is signed with uint32_t signatureType;
/// Location of the signature. Typically points to the end of the application uint32_t signatureLocation;
/// Information about the application ApplicationData_t app;
/// Pointer to information about the certificate ApplicationCertificate_t *cert;
/// Pointer to Long Token Data Section uint8_t *longTokenSectionAddress;
/// Parser Decryption Key const uint8_t decryptKey[16];
} ApplicationProperties_t;
```

- Option 2 must be implemented before signing the GBL image for ECDSA-P256-SHA256 or certificate-based Secure Boot.
- The options for the GBL Decryption Key are mutually exclusive. Either one of the two (VSE) or three (HSE) key storages will be selected for decryption.
- From a security point of view, it is highly recommended to use or upgrade to option 1 for HSE devices and option 2 for VSE devices.
- If the GBL Decryption Key in the selected option is compromised, the simple way is to upgrade the GBL to option 2 (if the existing option is 1 or 3) with the new GBL Decryption Key.

Simplicity Commander

The following procedures describe how to program the GBL Decryption Key for the options below. All procedures assume the required files are in the same folder.

- Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0)
- Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1)
- Default Storage on the Top Page of the Main Flash

1. Generate a 128-bit Symmetric Key.
   Run the `util genkey` command to generate the token file for the GBL Decryption Key.

   ```
   commander util genkey --type aes-ccm --outfile aes_key.txt
   ```

   ```
   Using Windows' Cryptographic random number generator
   DONE
   ```

2. (Use symmetric key stored in Secure Element storage) Run the `security writekey` command to provision the GBL Decryption Key to the SE OTP slot. The GBL Decryption Key cannot be changed once written.

   ```
   commander security writekey --decrypt aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
   ```

   ```
   Device has serial number 00000000000000000d6ffffead3d94

   ================================================================================
   Please look through any warnings before proceeding.
   THIS IS A ONE-TIME command, any encrypting of GBL files must be done with this key.
   Type 'continue' and hit enter to proceed or Ctrl-C to abort:
   ================================================================================
   continue
   DONE
   ```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note: It cannot read back the GBL Decryption Key from the HSE OTP.

3. (Use symmetric key stored in Application Properties Struct) Run the `convert` command to program the GBL Decryption Key to the `Application Properties Struct` of the GBL.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

Notes:

- The `--aeskey` option in the `convert` command requires Simplicity Commander v1.12.3 or above.
- The GBL Decryption Key can only be added to the GBL with `Application Properties Struct` v1.2 or higher.

4. (Default Storage on the Top Page of the Main Flash) Run the `flash` command to program the GBL Decryption Key in the token file to the top page of the main flash.

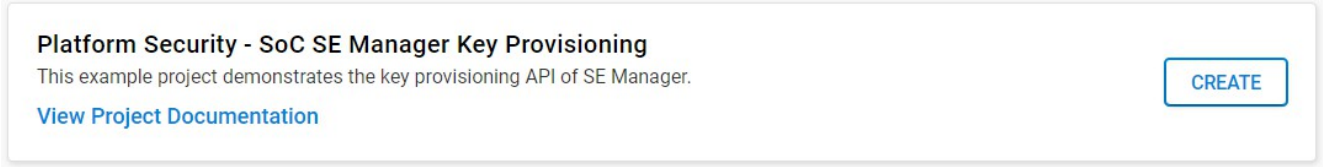
```
commander flash --tokengroup znet --tokenfile aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
```

```
Writing 8192 bytes starting at address 0x000fe000
Comparing range 0x000FE000 - 0x000FFFFF (8 KB)
Programming range 0x000FE000 - 0x000FFFFF (8 KB)
DONE
```

> Note: The MCU Series 2 VSE devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the flash --tokengroup znet command.

SE Manager Key Provisioning Platform Example

This example only applies to option 1 for Series 2 and Series 3 HSE devices. Click the View Project Documentation link to open the readme file.



1. Modify the default GBL Decryption Key in the `aes_key[16]` array in `app_process.c` to the desired values.

```
/// 128-bit AES key
SL_ALIGN(4) static const uint8_t aes_key[16] = {
0x81, 0xa5, 0xe2, 0x1f, 0xa1, 0x52, 0x86, 0xf1,
0xdf, 0x44, 0x5c, 0x2c, 0xc1, 0x20, 0xfa, 0x3f
};
```

2. Modify the `ciphertext[16]` array in `app_process.c` to the expected ciphertext for AES ECB on 16 bytes zero plaintext to verify the GBL Decryption Key in step 1.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
/// Ciphertext to verify 128-bit AES key static const uint8_t ciphertext[16] = {
0x66, 0xd2, 0x0f, 0x99, 0x65, 0x3e, 0xa8, 0xd0, 0x83, 0x05, 0xa6, 0x39, 0xd4, 0x4e, 0x98, 0xa6
};
```

3. Follow the procedures in Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
4. Build the project and run the application. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot if a signed application image is required.
5. Then press ENTER to program the hard-coded GBL Decryption Key to HSE OTP.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
. SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)

. Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
+ Current SE firmware version (MSB..LSB): 00010209

. Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)

. Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.
+ Warning: The 128-bit AES key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

6. Press ENTER to confirm the operation. The program either returns `SL_STATUS_OK` or `SL_STATUS_INVALID_PARAMETER` (already present) and performs AES ECB encryption to verify the GBL Decryption Key in HSE OTP.

```
. Initialize 128-bit AES key... SL_STATUS_OK (cycles: 39059 time: 1027 us)

. Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11013 time: 289 us)
+ Compare encrypted message with expected ciphertext... OK

. Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

```
. Initialize 128-bit AES key... SL_STATUS_INVALID_PARAMETER (cycles: 4474 time: 117 us)

. Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11001 time: 289 us)
+ Compare encrypted message with expected ciphertext... OK

. Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

7. Press SPACE to skip the programming of the Public Sign Key.

```
. Get public sign key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

8. Press SPACE to skip the programming of the Public Command Key.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)

. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

9. Press SPACE to skip the programming of the secure boot configuration.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 10 time: 0 us)
```

Secure Boot

You should usually not enable Secure Boot during the development phase to avoid a clash on debugging. The Secure Boot feature is enabled near firmware release and uses the following sections to validate the configuration and system functionality.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Generate an Unsigned GBL Image

There are two ways to configure the application firmware through a GBL project.

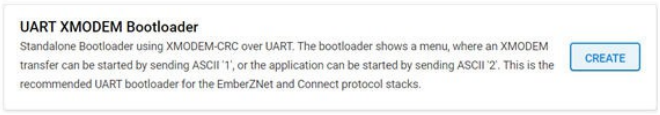
1. Use AppBuilder (`.isc file`) in GSDK v3.2 and lower.
2. Use Bootloader-core software component (`.slcp` file) in GSDK v4.0 and higher, and SiSDK v2024.6.0 and higher.

The following notes apply to the AppBuilder and Bootloader-core software component.

- Enabling the Allow use of public key from manufacturing token storage option is mandatory on VSE devices (cannot be disabled in AppBuilder and is discarded in Bootloader-core software component) for ECDSA-P256-SHA256 Secure Boot.
- The HSE device ignores this default enabled option if the Public Sign Key has been provisioned in OTP.
- When the Enable application rollback protection option (GSDK ≥ v3.0) is enabled, the GBL stores an application version counter at the end of the bootloader space in flash. The GBL checks this version during upgrades and at every boot, but it does not prevent flashing an older image onto the device.
- The Minimum application version allowed option (GSDK ≥ v3.0) sets the lowest application version that can boot. This option maintains the version counter, which resets to 0 after upgrading the GBL.
- The Enable application rollback protection option is not applicable if the SECURE_BOOT_PAGE_LOCK_FULL in SE OTP is enabled. See section "*Secure Boot with Application Rollback Protection*" in UG266 (for GSDK v3.2 and lower), Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices), or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher for details about the application rollback protection.
- The GBL size and starting address are device-dependent. For more information about the bootloader size and starting address on Series 2, see section "Memory Space For Bootloading" in UG103.6: Bootloader Fundamentals.

Refer to Generate a GBL Upgrade Image File to create GBL upgrade image file for the Require signed firmware upgrade files and Require encrypted firmware upgrade files options. For simplicity, the Secure Boot examples in this application note do not enable these options. Refer to UG266, Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices), or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher for information about these options.

The following sections describe how to build the unsigned GBL image from the UART XMODEM Bootloader (GSDK < v4.1) or Bootloader - NCP UART XMODEM (GSDK ≥ v4.1).





AppBuilder

This application note uses UART XMODEM Bootloader example v1.12.0 in GSDK v3.2.3. The procedures and pictures may be different for the other versions of this example.

1. Create a UART XMODEM Bootloader project.
2. The Plugins tab in AppBuilder (bootloader-uart-xmodem.isc) shows the default configurations for the UART XMODEM Bootloader example.
3. Use Bootloader Core, provides API: core in the Plugins tab to set up the application firmware configurations.
   a. This application note uses the configuration below for ECDSA-P256-SHA256 Secure Boot.

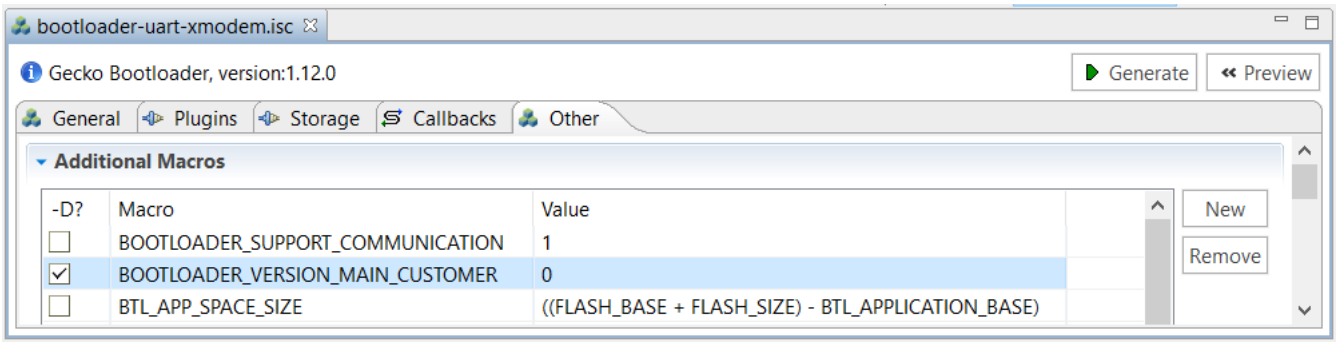
   ```
   ![ECDSA-P256-SHA256 Secure Boot configuration](./resources/sld794-image42.jpeg)
   ```

   b. This application note uses the configuration below for Certificate-based Secure Boot.

   ```
   ![Certificate-based Secure Boot configuration](./resources/sld794-image43.jpeg)
   ```

4. Enter a higher version number (default is 0) to the macro BOOTLOADER_VERSION_MAIN_CUSTOMER → Value in the Additional Macros field on the Other tab for anti-rollback protection of GBL.

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the AppBuilder settings below to keep the existing application image when upgrading the GBL or SE.
   a. Enter the required application image size to the macro BTL_APP_SPACE_SIZE → Value in the Additional Macros field on the Other tab. Check the -D? checkbox to add this definition to the project.

   ```
   This application note uses 507904 (496 kB) to replace the default value of ((FLASH_BASE + FLASH_SIZE) - BTL_APPLICATION_BASE).

   ![Bootloader application size macro](./resources/sld794-image45.jpeg)
   ```

   b. The Base address of bootloader upgrade image ≥ (BTL_APP_SPACE_SIZE + size of the GBL). The example in this application note uses EFR32MG21A010F1024:

   ```
   Base address of bootloader upgrade image = 507904 (496 kB) + 16384 (16 kB for GBL) = 524288 (512 kB)

   ![Base address of bootlader upgrade image](./resources/sld794-image46.png)

   > **Note**: The default value of Base address of bootloader upgrade image is 32768 (32 kB).
   ```

   c. The (Base address of bootloader upgrade image + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

   ```
   The example in this application note uses EFR32MG21A010F1024:

   512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes \< 1024 kB (size of main flash)

   For more information about the size of the GBL and SE, see section *Storage Space Size Configuration* in [UG266](https://www.silabs.com/documents/public/user-guides/ug266-gecko-bootloader-user-guide.pdf).

   > **Note**: It requires GBL v1.11.0 or above to support this feature.
   ```

6. Click [Generate] in the right upper corner.
7. In the Generation Successful dialog, click [OK].
8. Build the project to generate the unsigned GBL image file (`bootloader-uart-xmodem.s37`).
9. (Optional) Run the `util appinfo` command to check all available information (application properties) in an unsigned GBL image. The `App version` is the GBL version for the SECURE_BOOT_ANTI_ROLLBACK option.

```
commander util appinfo bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b1c
Signature location              : 0x00002d08
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

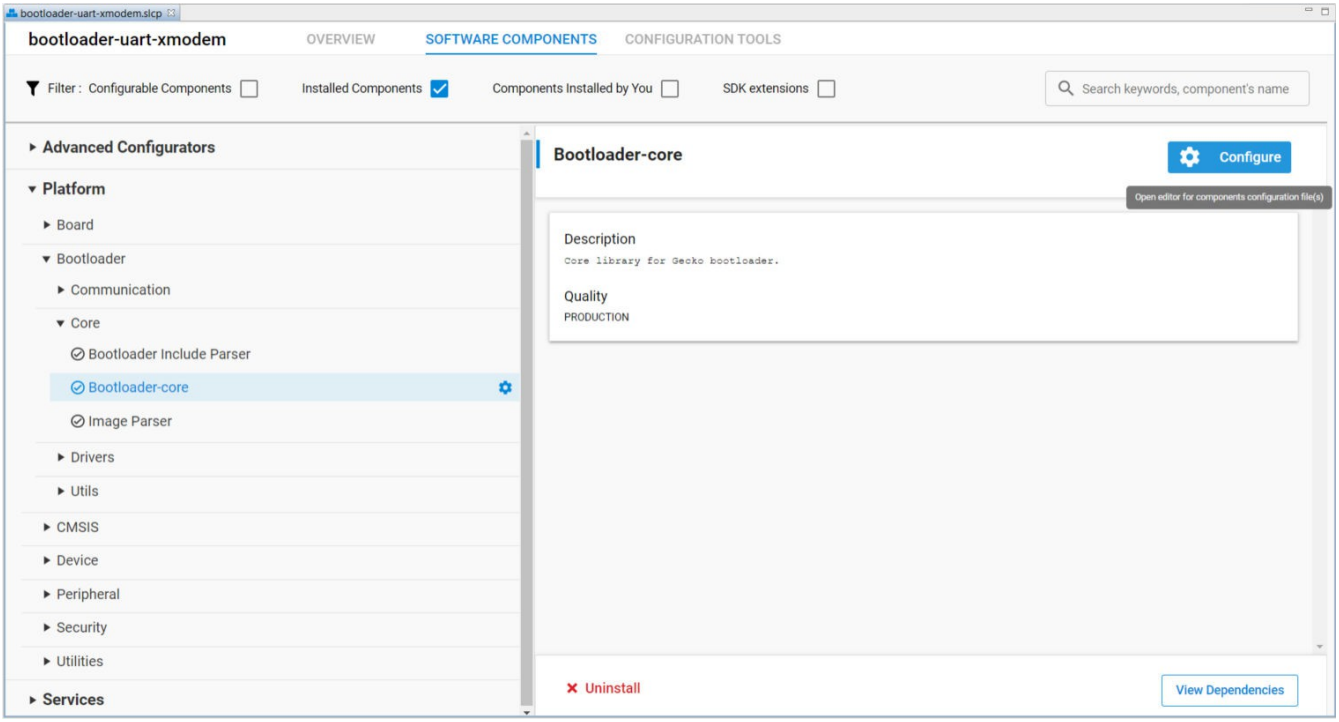
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the
device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x010c0000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen
certificate based signing, it will not accept direct signing.
DONE
```

Bootloader-core Software Component

This application note uses UART XMODEM Bootloader example v2.0.0 in GSDK v4.0. The procedures and pictures may be different for the other versions of this example.

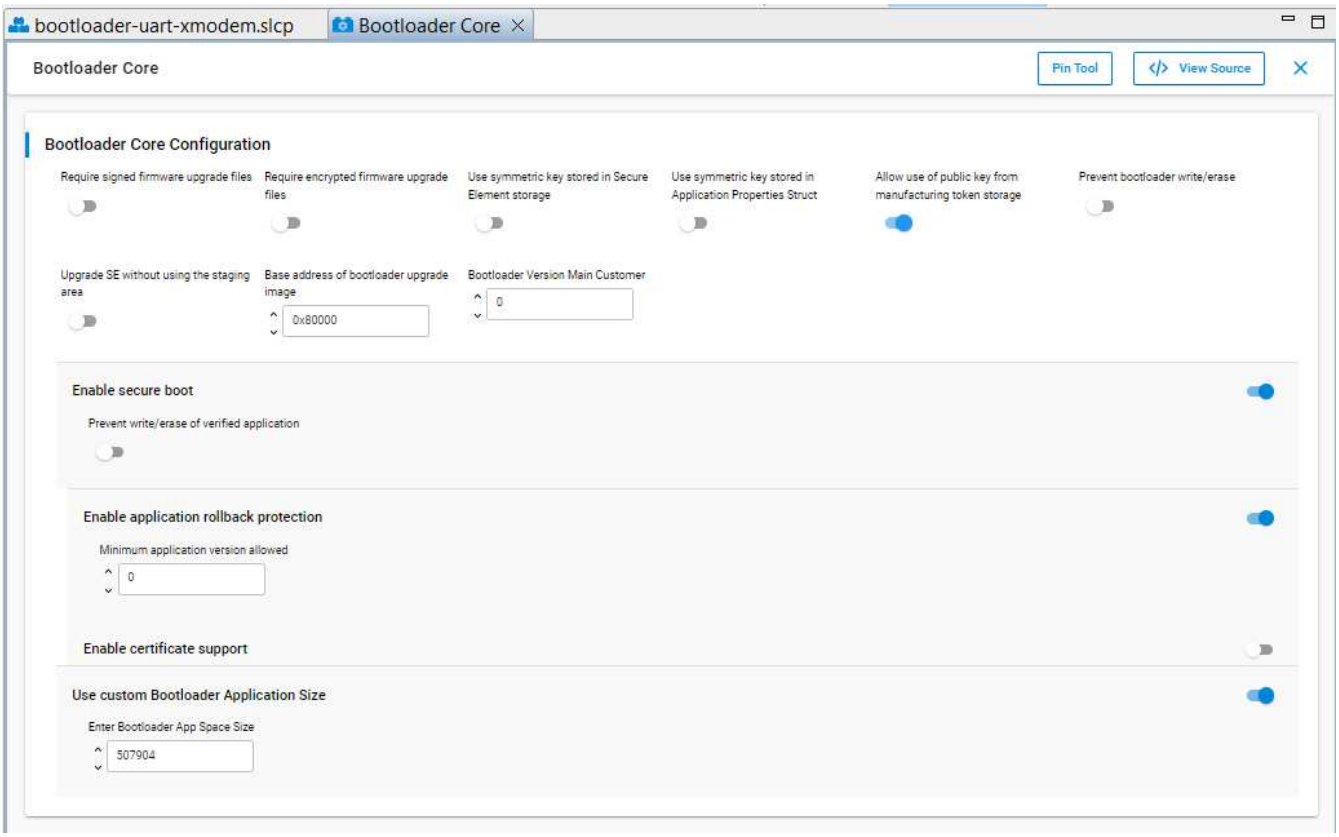
1. Create a UART XMODEM Bootloader project.
2. Checking the Installed Components under the SOFTWARE COMPONENTS tab shows the list of installed components (`bootloader-uart-xmodem.slcp`) in the UART XMODEM Bootloader example.
3. Click [Configure] in the Bootloader-core component to open the Bootloader Core Configuration.



> Note: Install "Bootloader Core (Series-3)" for series 3 devices.

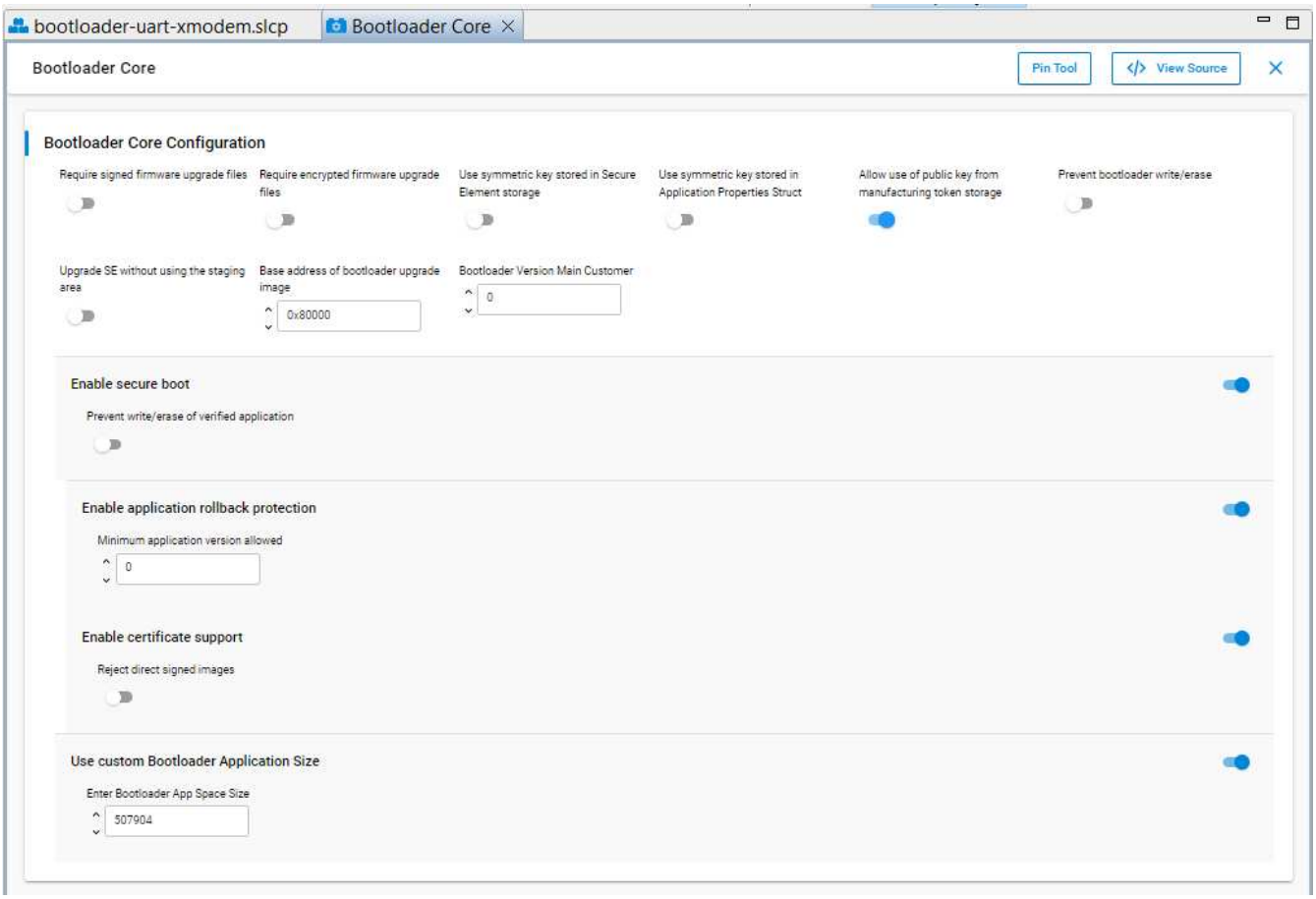
4. Use Bootloader Core Configuration in Bootloader-core to set up the application firmware configurations.

Copyright © 2025 Silicon Laboratories. All rights reserved.



6. This application note uses the configuration below for Certificate-based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.



7. [For series 2 devices only] Enter a higher version number (default is 0) to BOOTLOADER VERSION MAIN CUSTOMER for anti-rollback protection of GBL.



8. [For series 2 devices only] The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the Bootloader-core settings below to keep the existing application image when upgrading the GBL or SE.
   a. Enter the required application image size to the Enter Bootloader App Space Size dialog box after enabling the Use custom Bootloader Application Size option.

```
This application note uses 507904 (496 kB) to replace the default value of 0.

![screenshot](./resources/sld794-image51.jpeg)
```

   b. The Base address of bootloader upgrade image ≥ (Enter Bootloader App Space Size + size of the GBL). The example in this application note uses EFR32MG21A010F1024:

```
Base address of bootloader upgrade image = 507904 (496 kB) + 16384 (16 kB for GBL) = 524288 or 0x80000 (512 kB)

![screenshot](./resources/sld794-image52.png)

> **Note**: The default value of Base address of bootloader upgrade image is 32768 or 0x8000 (32 kB).
```

   c. The (Base address of bootloader upgrade image + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

Copyright © 2025 Silicon Laboratories. All rights reserved.

> The example in this application note uses EFR32MG21A010F1024:
>
> 512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes \< 1024 kB (size of main flash)
>
> For more information about the size of the GBL and SE, see section *Storage Space Size Configuration* in [Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices)](https://docs.silabs.com/mcu-bootloader/latest/bootloader-user-guide-gsdk-4/).

9. Click [X] in the right upper corner to exit the Bootloader Core Configuration.
10. Build the project to generate the unsigned GBL image file (bootloader-uart-xmodem.s37).
11. (Optional) Run the `convert` command to program the GBL Decryption Key to the Application Properties Struct if this GBL Decryption Key option in GBL (GSDK ≥ v4.1) is selected.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

1. (Optional) Run the `util appinfo` command to check all available information (application properties) in an unsigned GBL image. The App version is the GBL version for the SECURE_BOOT_ANTI_ROLLBACK option.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the
device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen
certificate based signing, it will not accept direct signing.
DONE
```

> Note: For TrustZone-aware bootloaders, the unsigned GBL image is the combined image of Secure and Non-secure bootloaders. The Bootloader-core component is installed in the Secure bootloader.

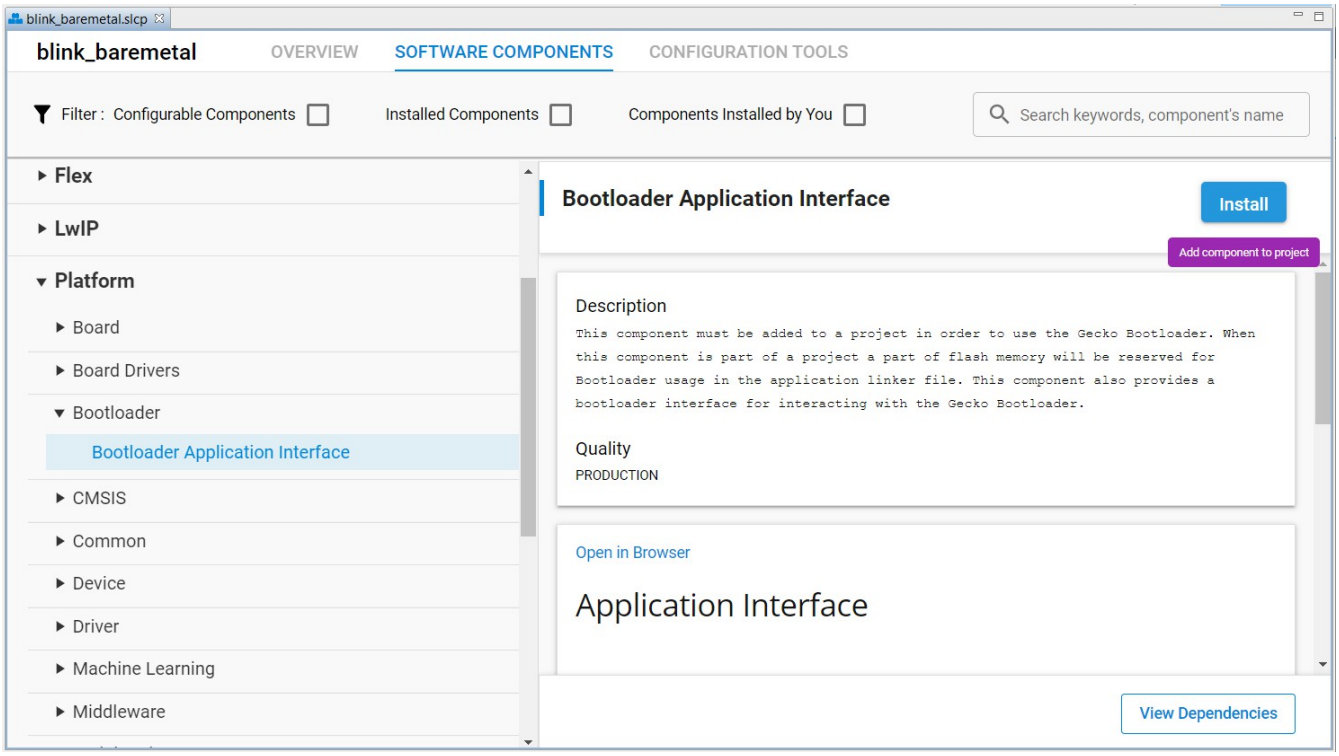
Generate an Unsigned Application Image

This section describes how to generate an unsigned application image for the GBL.

1. For Series 2 devices, the application image should be placed on the main flash page after the GBL. For more information about the application starting address, see section *Memory Space For Bootloading* in Bootloader Fundamentals.

Copyright © 2025 Silicon Laboratories. All rights reserved.

2. For Series 3 devices, the application image should be placed in the code region 1 next to the bootloader which is in region 0, see section *External Flash Architecture in Series 3 Devices* in Platform Memory Model.
3. (Simplicity Studio 5) You can use the Bootloader Application Interface component to set up the start address of the application image. This application note uses Platform - Blink Bare-metal example in GSDK v3.2.3. The procedures and pictures may be different on other versions of the GSDK. The following steps can apply to other platform examples in GSDK and SiSDK.
   a. Create a Platform - Blink Bare-metal project.
   b. The Software Components tab shows the list of available components (`blink_baremetal.slcp`) that you can install in the project.
   c. Select Platform > Bootloader > Bootloader Application Interface.
   d. Click [Install].



> Note: For the wireless protocol stack example, the Bootloader Application Interface component is already present in the project.

4. The application image should contain an `ApplicationProperties_t` struct (defined in `application_properties.h` in the Windows folder below) declaring the application version, capabilities, and other metadata.
   For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`
   For SiSDK v2024.6.0 and higher, or GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`
   Below is an example source file `app_properties.c` with `ApplicationProperties_t` struct for Secure Boot on GSDK v3.2 and lower.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#include <stddef.h>
#include "application_properties.h"

const ApplicationProperties_t sl_app_properties = {
.magic = APPLICATION_PROPERTIES_MAGIC,
.structVersion = APPLICATION_PROPERTIES_VERSION,
.signatureType = APPLICATION_SIGNATURE_NONE,
.signatureLocation = 0,
.app = {
    .type = APPLICATION_TYPE_MCU,
    .version = 1UL,
    .capabilities = 0UL,
    .productId = {0U},
},
};
```

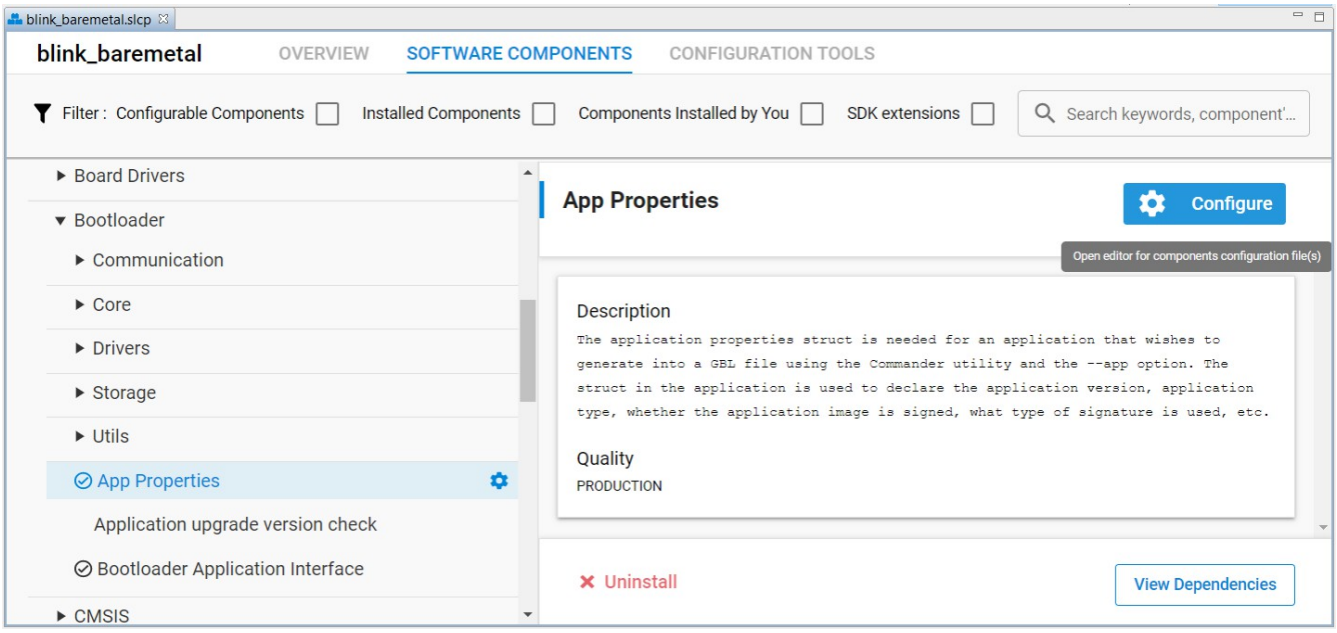
The `signatureType` and `signatureLocation` are filled by Simplicity Commander when signing the application image using the `convert` command.

5. The following table describes how to add the `app_properties.c` file in step 4 to Platform - Blink Bare-metal project. For the wireless protocol stack example, the `app_properties.c` file with `ApplicationProperties_t` struct is already present in the project.
   App properties for older Simplicity Studio and GSDK versions

Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower	Simplicity Studio 5 with GSDK v4.0 and higher
Manually added	Automatically added after installing the Bootloader Application Interface component in step 3 to the project.

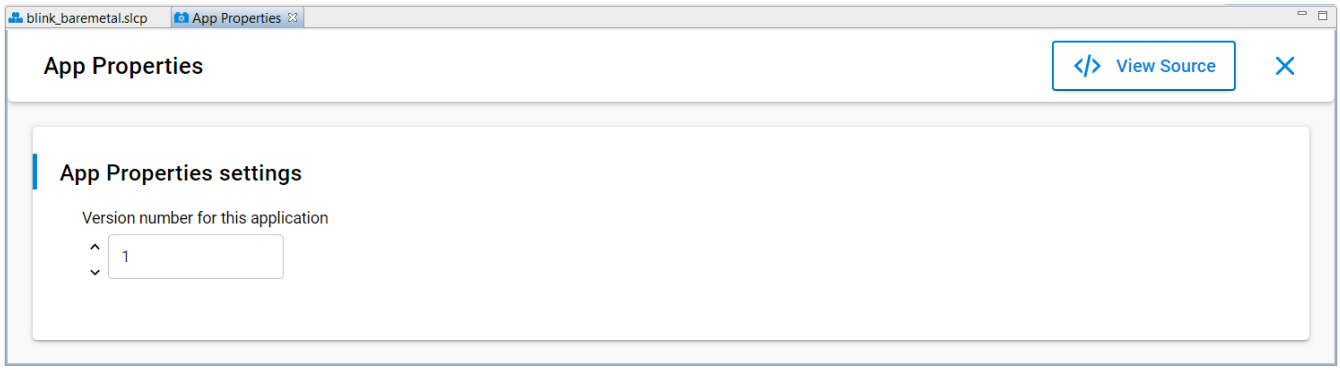
> Note: Refer to the Knowledge Article in Silicon Labs Community to add `app_properties.c` to the project in Simplicity Studio 4.

6. (Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower) Enter a higher version number to `.version` in `app_properties.c` for anti-rollback protection (if enabled) of the application.
7. (Simplicity Studio 5 with GSDK v4.0 and higher) Click [Configure] in the App Properties component under Platform > Bootloader to open the App Properties configuration. The example below uses GSDK v4.0. The procedures and pictures may be different on other versions of the GSDK.



Copyright © 2025 Silicon Laboratories. All rights reserved.

8. Enter a higher version number to Version number for this application dialog box in App Properties settings for anti-rollback protection (if enabled) of the application.



> Note: The `app_properties.c` is in the Windows folder below.
>
> `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\app_properties`

9. Build the project to generate the unsigned application image file (`blink_baremetal.s37`).
10. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in an un-signed application image. The App version is for the Enable application rollback protection option in the [AppBuilder](#appbuilder or Bootloader-core software component.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image. Application properties info:
Application properties location : 0x00006198
Signature location              : Not set (0x00000000)
Signature type                  : No signature
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the
device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

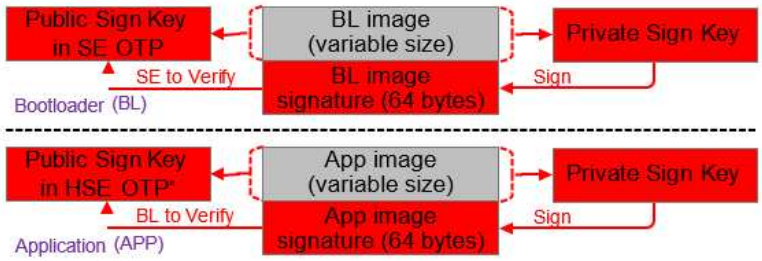
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen
certificate based signing, it will not accept direct signing.
DONE
```

Note: For the TrustZone-aware applications, the unsigned application image is the combined image of Secure and Non-secure applications. The `ApplicationProperties_t` struct is located in the Secure application.

Signing for ECDSA-P256-SHA256 Secure Boot

The following figure describes the signing and verification for ECDSA-P256-SHA256 Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.



> Note: The bootloader cannot access the Public Sign Key in VSE OTP to verify the application image. Therefore VSE devices need to store a Public Sign Key copy on the top page of the main flash (see section *Key Storage* in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices)).

Public Sign Key usage for VSE and HSE Devices

Device	FSB to Verify the Bootloader Image	SSB to Verify the Application Image
HSE	Use the Public Sign Key in HSE OTP	Use the Public Sign Key in HSE OTP
VSE	Use the Public Sign Key in VSE OTP	Use the Public Sign Key on the top page of the main flash

The HSE device ignores the default enabled Allow use of public key from manufacturing token storage option once the Public Sign Key has been provisioned.

To have better protection on the Public Sign Key, the certificate-based Secure Boot is strongly recommended on VSE devices since the SSB does not require accessing the Public Sign Key to verify the application signature.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If Secure Boot flag is not enabled in SE OTP, follow the procedures in Simplicity Commander to set up the ECDSA-P256-SHA256 Secure Boot configuration for the bootloader.
2. Follow the procedures in AppBuilder or Bootloader-core Software Component to set up the ECDSA-P256-SHA256 Secure Boot configuration for the user application to generate an unsigned bootloader image.
3. The following steps show two methods to generate a signed bootloader image—either directly with Simplicity Commander (3a) or using an HSM with Commander (3b).
   a. (Using Simplicity Commander) Run the `convert` command with Private Sign Key to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```sh
commander convert bootloader-uart-xmodem.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile bootloader-uart-xmodem.s37
```

```sh
Parsing file bootloader-uart-xmodem.s37... Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Image SHA256: c53bb8a3fd88a5071bfb71444324bb136b276160318488ff89011bbd269e114e
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
S = 9C7C7AF2624165FD90FB3B114E3FA6FE4F4C5625B15C9F3D50DCB04DD06A7B19

 Verifying signed image...
 Writing to bootloader-uart-xmodem.s37...
 Overwriting file: bootloader-uart-xmodem.s37...
 DONE
 ```
```

b. (Using an HSM and Simplicity Commander) Run the `convert` command with `--extsign` option to generate an external signing bootloader image file ( `bootloader-uart-xmodem.extsign` ).

```
 ```sh
 commander convert bootloader-uart-xmodem.s37 --secureboot --extsign --outfile bootloader-uart-xmodem
 ```

 ```sh
 Parsing file bootloader-uart-xmodem.s37... Found Application Properties at 0x00002b1c
 Writing Application Properties signature pointer to point to 0x00002d08
 Setting signature type in Application Properties: 0x00000001
 Writing to bootloader-uart-xmodem.extsign...
 DONE
 ```

 Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader image.
This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** to simulate this process.
The signature is in the `bl_signature.der`.

 ```sh
 openssl dgst -sha256 -binary -sign sign_key.pem -out bl_signature.der bootloader-uart-xmodem.extsign
 ```

 Run the `convert` command with the **bootloader image signature** to overwrite the unsigned bootloader image
file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

 ```sh
 commander convert bootloader-uart-xmodem.s37 --secureboot --signature bl_signature.der
 --verify sign_pubkey.pem --outfile bootloader-uart-xmodem.s37
 ```

 ```sh
 Parsing file bootloader-uart-xmodem.s37... Parsing signature file bl_signature.der...
 R = 0E9FC64F41B55367894908D3ADAC40E8D145E33224C4BAA8151EC3EFD107A154
 S = F56230AA6484E55270F22A4D164377CA918F66A367656AB6E10CB3F58641CE84
 Found Application Properties at 0x00002b1c
 Writing Application Properties signature pointer to point to 0x00002d08
 Setting signature type in Application Properties: 0x00000001

 Verifying signed image...
 Writing to bootloader-uart-xmodem.s37...
 Overwriting file: bootloader-uart-xmodem.s37...
 DONE
 ```
```

4. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if
the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously
seen certificate based signing, it will not accept direct signing.
DONE
```

5. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
6. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Simplicity Commander output for Series 3 devices will look as follows:

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 14280 bytes starting at address 0x01000000
Erasing range 0x01000000 - 0x01007FFF (1 sector, 32 KB)
Programming range 0x01000000 - 0x01000FFF (4 KB)
Programming range 0x01001000 - 0x01001FFF (4 KB)
Programming range 0x01002000 - 0x01002FFF (4 KB)
Programming range 0x01003000 - 0x01003FFF (4 KB)
Programming range 0x01004000 - 0x01004FFF (4 KB)
Programming range 0x01005000 - 0x01005FFF (4 KB)
Programming range 0x01006000 - 0x01006FFF (4 KB)
Programming range 0x01007000 - 0x01007FFF (4 KB)
JLinkError: Failed to halt CPU.
Closing region 0 (this consumes one OTP bit, consider --noclose on development/testing devices)
Flashing completed successfully!
DONE
```

Application Image File

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. The following steps show two methods to generate a signed application image, either directly with Simplicity Commander (2a) or using an HSM with Commander and OpenSSL (2b).
   a. (Using Simplicity Commander) Run the `convert` command with Private Sign Key to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

````
```sh
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

```sh
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 056E3AA36BD882B5467D44A56DB7CC1AEE44D45BC9B98FAB05BE2C032573A1F7
S = BE1D27CE7877D0BC761C0F02690CC74251EBE3A458474C573C21B3A738A03577

Verifying signed image...
Writing to blink_baremetal.s37... Overwriting file: blink_baremetal.s37...
DONE
```
````

b. (Using an HSM and Simplicity Commander) Run the `convert` command with `--extsign` option to generate an external signing application image file (`blink_baremetal.extsign`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

```sh
Parsing file blink_baremetal.s37...
Found Application Properties at 0x00006198
Writing Application Properties signature pointer to point to 0x0000643c
Setting signature type in Application Properties: 0x00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing application image. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** to simulate this process. The signature is in the `app_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```sh
commander  convert blink_baremetal.s37 --secureboot    --signature app_signature.der   --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

```sh
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = BD5BDC866CE67DA104B1E7B686C45B7BF96F2643154D37ACC63DACDF69C27E89
S = 2DD3BFFAC857A5B0BD8C9B4DDB23D21944D062F8E431D36541B84EF411C1CC92
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to blink_baremetal.s37... Overwriting file: blink_baremetal.s37...
DONE
```

3. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties in image. Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x000064d8
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

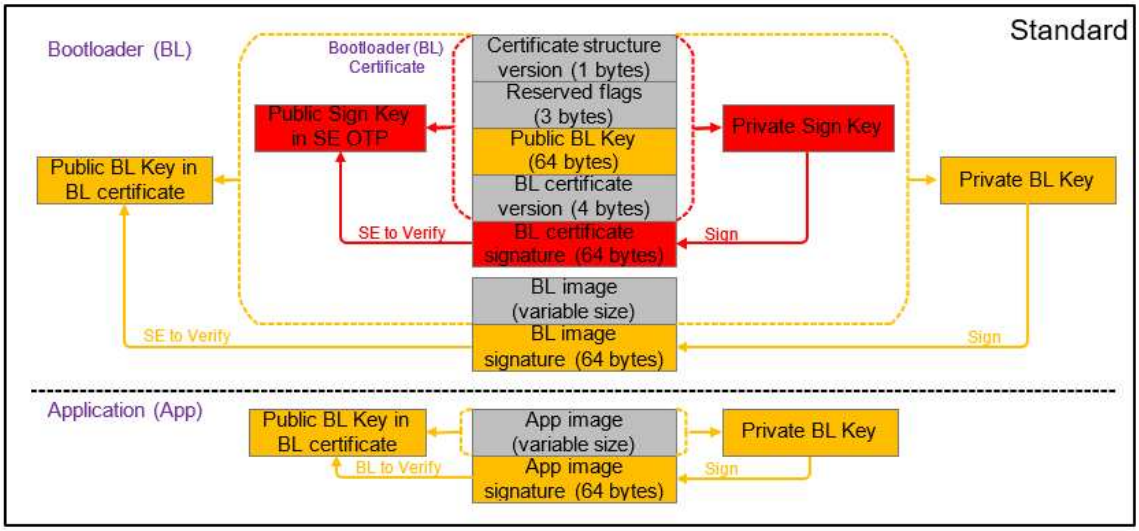
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if
the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously
seen certificate based signing, it will not accept direct signing.
DONE
```

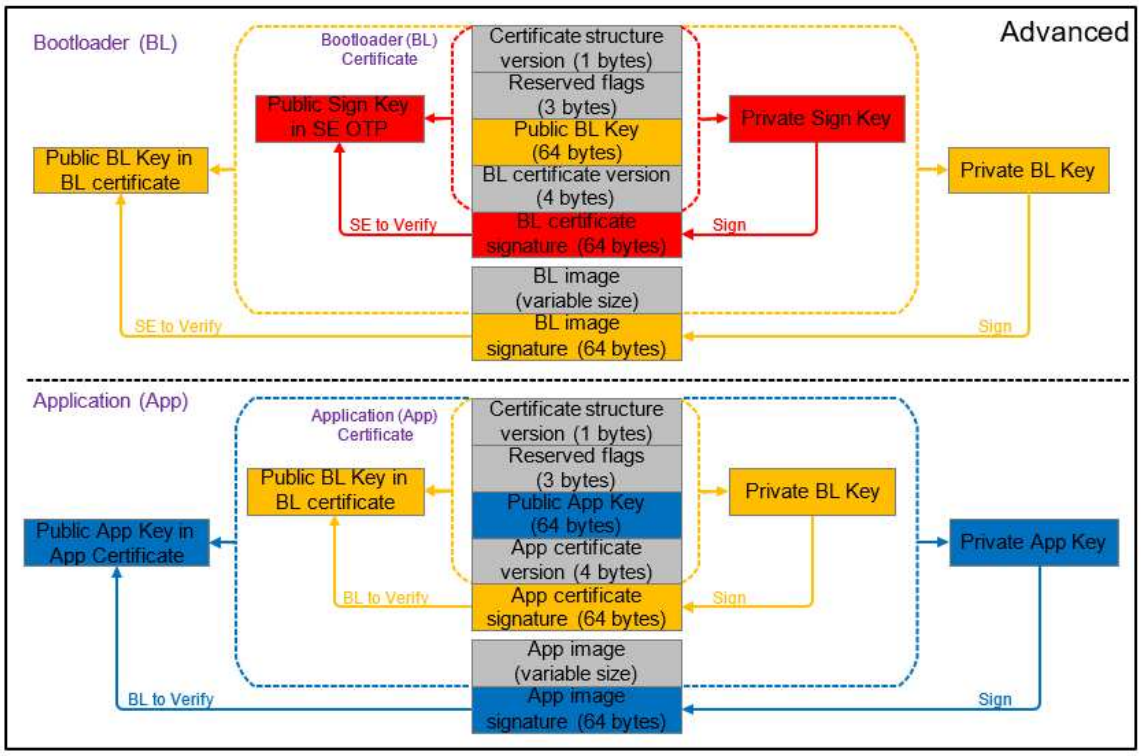
4. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Signing for Certificate-Based Secure Boot

The following figure describes the signing and verification for certificate-based Secure Boot. You can freely switch between standard and advanced certificate-based Secure Boot by upgrading the application firmware without and with the application certificate.



Copyright © 2025 Silicon Laboratories. All rights reserved.



Certificate

The following table describes the elements of a certificate.

Element	Description
Certificate structure version	The version of the certificate structure.
Reserved flags	Reserved in the current certificate structure version.
Certificate public key	ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
Certificate version	The version of the running certificate.
Certificate signature	ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

Notes:

- The `application_properties.h` file in the Windows folder below defines the parameters of the certificate structure (`ApplicationCertificate_t`).
  For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`
  For GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`
- The certificate is not in X.509 format.

Private/Public Key Pair

The following table describes two Private/Public Key pairs used in certificates for certificate-based Secure Boot. You can use Simplicity Commander or HSM to generate these key pairs.

Certificate	Private Key	Public Key	Description
Bootloader (bl_cert.bin) (1)	bl_cert_key.pem (Private Bootloader Key)	bl_cert_pubkey.pem (Public Bootloader Key)	The bootloader certificate is signed by the Private Sign Key corresponding to the Public Sign Key in SE OTP.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Certificate	Private Key	Public Key	Description
Application (app_cert.bin) (2)	app_cert_key.pem (Private Application Key)	app_cert_pubkey.pem (Public Application Key)	The application certificate is signed by the Private Bootloader Key corresponding to the Public Bootloader Key in the bootloader certificate.

Notes:

1. a. Certificate version in the bootloader certificate < certificate version in SE flash - the certificate is rejected.
   b. Certificate version in the bootloader certificate = certificate version in SE flash - the certificate is accepted.
   c. Certificate version in the bootloader certificate > certificate version in SE flash - the certificate is accepted. The certificate version in SE flash is updated to match (revocation mechanism).
2. The certificate version in the application certificate is compared with the certificate version in the bootloader certificate. The application certificate is accepted if its version is equal to or higher than the certificate version in the bootloader certificate.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If Certificated based Secure Boot flag is not enabled in SE OTP, follow the procedures in Simplicity Commander to set up the certificate-based Secure Boot configuration for the bootloader.
2. Follow the procedures in AppBuilder or Bootloader-core Software Component to set up the certificate-based Secure Boot configuration for the user application to generate an unsigned bootloader image.
3. The following steps show two methods to generate a bootloader certificate, either directly with Simplicity Commander (3a) or using an HSM with Commander (3b).
   a. (Using Simplicity Commander) Run the `util gencert` command with Public Bootloader Key and Private Sign Key to generate the bootloader certificate (`bl_cert.bin`). Refer to the table abovefor details about the `--cert-version` for bootloader certificate.

````
 ```sh
 commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem
 --sign sign_key.pem --outfile bl_cert.bin
 ```

 ```sh
 Successfully signed certificate
 DONE
 ```

 Run the `convert` command with **Bootloader Certificate** and **Private Bootloader Key** to overwrite the
unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

 ```sh
 commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin
 --keyfile bl_cert_key.pem --outfile bootloader-uart-xmodem.s37
 ```

 ```sh
 Parsing file bootloader-uart-xmodem.s37...
 Writing certificate to location 0x00002cf0
 Private key matches public key in certificate.
 Found Application Properties at 0x00002d78
 Writing Application Properties signature pointer to point to 0x00002f64
 Setting signature type in Application Properties: 0x00000001
 Image SHA256: 3cf574b688853a801e8dc98687414db27f886c60c55dbf7fea2d47633df94e8d
 R = C866592B4CB7BAD9EFC35985F1B9D52C65C26453D4808597EEEFFB16DC4AA962
 S = 94CAA21ED5D7772F96BBF4D24A0711A94DCCB6D4D38DFA45182876B9BE2A8DE3

 Verifying signed image...
 Writing to bootloader-uart-xmodem.s37... Overwriting file: bootloader-uart-xmodem.s37...
 DONE
 ```
````

Copyright © 2025 Silicon Laboratories. All rights reserved.

b. (Using an HSM and Simplicity Commander) Run the `util gencert` command with Public Bootloader Key and `--extsign` option to generate an external signing bootloader certificate (`bl_cert.extsign`). Refer to the table above for details about the `--cert-version` for bootloader certificate.

````
 ```sh
 commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem
 --extsign --outfile bl_cert
 ```

 ```sh
 DONE
 ```

 Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader
certificate. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** to simulate
this process. The signature is in the `bl_cert_signature.der`.

 ```sh
 openssl dgst -sha256 -binary -sign sign_key.pem -out bl_cert_signature.der bl_cert.extsign
 ```

 Run the `util signcert` command with the **bootloader certificate signature** to generate the bootloader
certificate (`bl_cert.bin`).

 ```sh
 commander util signcert bl_cert.extsign --cert-type secureboot --signature bl_cert_signature.der
 --verify sign_pubkey.pem --outfile bl_cert.bin
 ```

 ```sh
 R = 065A58EA6CE6BBA44F3C59C6D255A901DBBC55FA97F261658B2026ABC8CD9680
 S = 8A0011AA6393BC284B13C8313EE6772030DE07E213E74CA0FEA740F3D33E6518
 Successfully verified signature Successfully signed certificate
 DONE
 ```

 Run the `convert` command with **Bootloader Certificate** and `--extsign` option to generate an external
signing bootloader image file (`bootloader-uart-xmodem.extsign`).

 ```sh
 commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin --extsign
 --outfile bootloader-uart-xmodem
 ```
````

4. (Optional) Run the `util verifysign` command with Public Sign Key to verify that the Bootloader Certificate and image were correctly signed.

```
commander util verifysign bootloader-uart-xmodem.s37 --verify sign_pubkey.pem
```

```
Parsing file bootloader-uart-xmodem.s37... Found application properties at 0x00002d78 Found certificate at
0x00002cf0
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

5. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002d00
Signature location              : 0x00002e14
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if
the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)
App version                     : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)

Found certificate at 0x00002c78
Application certificate info:
Certificate located at          : 0x00002c78
Certificate version             : 0x00000001
Certificate key                 : 0xb1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                                  7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
Certificate signature           : 0xef3b53368d4cd7821eb30a96140bbde8840378cfea30687a8c10642e1c7728fd
                                  309f976adf46e4eac62a2233f0c1f08f4e58344bdec61775b5282ceb351bb3d0
DONE
```

6. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
7. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Simplicity Commander output for Series 3 devices will look as follows:

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 14280 bytes starting at address 0x01000000
Erasing range 0x01000000 - 0x01007FFF (1 sector, 32 KB)
Programming range 0x01000000 - 0x01000FFF (4 KB)
Programming range 0x01001000 - 0x01001FFF (4 KB)
Programming range 0x01002000 - 0x01002FFF (4 KB)
Programming range 0x01003000 - 0x01003FFF (4 KB)
Programming range 0x01004000 - 0x01004FFF (4 KB)
Programming range 0x01005000 - 0x01005FFF (4 KB)
Programming range 0x01006000 - 0x01006FFF (4 KB)
Programming range 0x01007000 - 0x01007FFF (4 KB)
JLinkError: Failed to halt CPU.
Closing region 0 (this consumes one OTP bit, consider --noclose on development/testing devices)
Flashing completed successfully!
DONE
```

Application Image File (Standard Certificate-Based)

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. a. (Using Simplicity Commander) Run the convert command with Private Bootloader Key to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

```
commander   convert blink_baremetal.s37 --secureboot    --keyfile   bl_cert_key.pem --verify bl_cert_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 994739A26AB520A88A5550F1643AE263D88A952F185F96EE7021FA43DEA6138C
S = 65B7112715E2F999A6B216C32D3331AB63B2D31A0A1311DF36EEE62269F8D6AA

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

b. (Using an HSM and Simplicity Commander) Run the `convert` command with `--extsign` option to generate an external signing application image file (`blink_baremetal.extsign`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

```sh
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application image. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Bootloader Key** to simulate this process. The signature is in the `app_signature.der`.

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```sh
commander convert blink_baremetal.s37 --secureboot --signature app_signature.der
--verify bl_cert_pubkey.pem --outfile blink_baremetal.s37
```

```sh
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = 8DA79B020E954D24C23423D80627E046E44052736F6546902F016D64464E82DE
S = 9D5A1CC424E97A5AD0352A4EEA6BBF565FED5FC61FF99E63AA73DFFEAD9EE399
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. (Optional) Run the `util verifysign` command with Public Bootloader Key to verify that the application image file was correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

```
Parsing file blink_baremetal.s37...
Found application properties at 0x000061bc
Did not find application certificate in file
If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate
based signing, it will not accept direct signing.
Successfully verified application signature.
DONE
```

4. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x000064d8
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if
the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously
seen certificate based signing, it will not accept direct signing.
DONE
```

5. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Application Image File (Advanced Certificate-Based)

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. a. (Using Simplicity Commander) Run the `util gencert` command with Public Application Key and Private Bootloader Key to generate the application certificate (`app_cert.bin`). Refer to Private/Public Key Pair for details about the `--cert-version` for application certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem
--sign bl_cert_key.pem --outfile app_cert.bin
```

```
Successfully signed certificate
DONE
```

Run the `convert` command with Application Certificate and Private Application Key to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`). This command will inject the application certificate into the application image before signing.

```
commander   convert blink_baremetal.s37 --secureboot    --certificate   app_cert.bin    --keyfile
app_cert_key.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
Private key matches public key in certificate.
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x00006560
Setting signature type in Application Properties: 0x00000001
Image SHA256: 38fd11214c36abf3bb4c4eeda8cfdd2ca2ac2ff1e07072d555a06c74700a23f5
R = 6B4E3BB454513CAA4569415AE8F79453973AAC7FD1FC4914284B65010F3790A6
S = 1657CAAABED5798801872610383 58C83B1780A67CC41475370D94ED4445A5557

Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

b. (Using an HSM and Simplicity Commander) Run the `util gencert` command with Public Application Key and `--extsign` option to generate an external signing application certificate (`app_cert.extsign`). Refer to Private/Public Key Pair for details about the `--cert-version` for application certificate.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem
--extsign --outfile app_cert
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application certificate. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Bootloader Key** to simulate this process. The signature is in the `app_cert_signature.der`.

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_cert_signature.der app_cert.extsign
```

Run the `util signcert` command with the **application certificate signature** to generate the application certificate (`app_cert.bin`).

```sh
commander util signcert app_cert.extsign --cert-type secureboot --signature app_cert_signature.der
--verify bl_cert_pubkey.pem --outfile app_cert.bin
```

```sh
R = 279D4FA1B801D108F82E30B0CF1164BF597549287290BD3883C5847B91095CCE
S = 567F0E219D2089EF4D79C3D94E43D2FADFE1899B71492ED358E6A1B46AE8162F
Successfully verified signature Successfully signed certificate
DONE
```

Run the `convert` command with the **Application Certificate** and `--extsign` option to generate an external signing application image file (`blink_baremetal.extsign`).

```sh
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --extsign --outfile
blink_baremetal
```

```sh
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
```

3. (Optional) Run the `util verifysign` command with Public Bootloader Key to verify that the Application Certificate and image were correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

```
Parsing file blink_baremetal.s37...
Found application properties at 0x000061bc Found certificate at 0x000064d8
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

4. (Optional) Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x00006560
Signature type                  : ECDSA-P256
Long token section address      : Not set (0x00000000)

Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if
the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)

Found certificate at 0x000064d8
Application certificate info:
Certificate located at          : 0x000064d8
Certificate version             : 0x00000001
Certificate key                 : 0xe562003cd86e225decfd35712e431a19ecd5031a079b06c1d473620a6be9f57a
                                879820100fee074f28b5885fd6759f480b62aaa0717f96e245aab6635cfb1e11
Certificate signature           : 0x039aaba62b5258e68d16e167c3a611c719c542bb3483f5d4b522472b06adf30f
                                8cfcc484bf8551a208256e3d2d8c9194a7d2ac551e2cac659a99822308a40aa6
DONE
```

5. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Generate a GBL Upgrade Image File

This section provides steps to create GBL files used to upgrade bootloader, application and/or SE firmware on Series 2 and Series 3 devices.

> Note: Series 3 devices support only GBLv4 files for firmware upgrade process.

- Follow the procedures in AppBuilder or Bootloader-core Software Component to avoid overwriting the existing application image (if necessary) when upgrading the bootloader or SE.
- For a standalone bootloader with communication interface, you can only generate three separate GBL upgrade image files containing bootloader, SE, and application images.
- (Series 2 devices) For an application bootloader with storage, you can generate a single GBL upgrade image file (see example below) with a combination of bootloader, SE, and application images.

```
commander gbl create all.gbl --app app.s37 --bootloader bl.s37 --seupgrade se.seu
```

- A signed GBL upgrade image file is required if you enable the Require signed firmware upgrade files option in AppBuilder or Bootloader-core Software Component. The following table shows which private key(s) can be used to sign the GBL upgrade image file (bootloader, SE, or application) on VSE, HSE and Series 3 devices. The VSE devices store a Public Sign Key copy on the top page of the main flash to verify the GBL upgrade image file for ECDSA-P256-SHA256 Secure Boot.

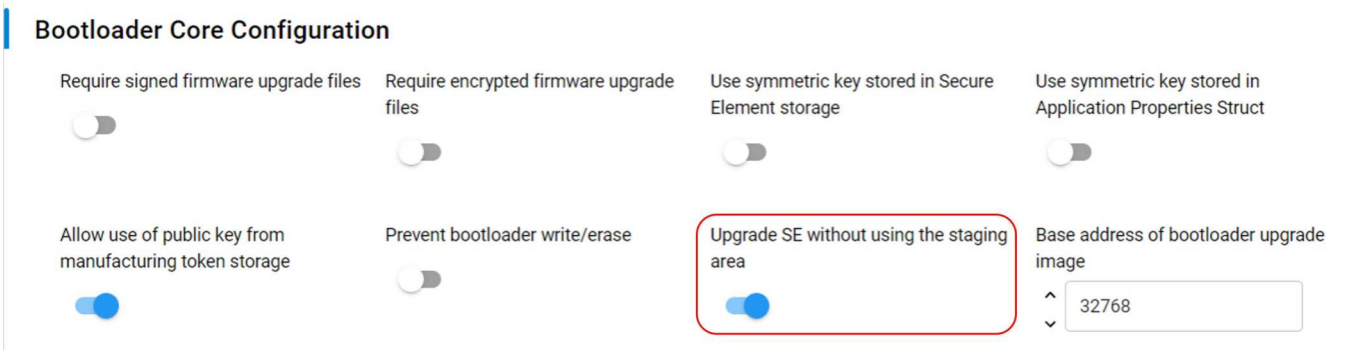
Private Key(s) usage to sign GBL upgrade image file

Secure Boot	HSE and Series 3	VSE
ECDSA-P256-SHA256	Private Sign Key	Private Sign Key (Public Sign Key in main flash)
Certificate-Based	Private Sign Key or Private Bootloader Key	Private Bootloader Key

- An encrypted GBL upgrade image file is required if you enable the Require encrypted firmware upgrade files option in AppBuilder or Bootloader-core Software Component. Refer to Provision GBL Decryption Key on how to provision the GBL Decryption Key for this option.
-

Copyright © 2025 Silicon Laboratories. All rights reserved.

[Only applicable for Series 2 devices] For an application bootloader with storage, you can enable the Upgrade SE without using the staging area option in GSDK v4.1.1 or higher to directly fetch the SE image from the GBL upgrade image file in storage instead of copying the image to the pre-configured upgrade location.



To use the above option, the SE image cannot be in the encrypted part of the GBL upgrade image file if the Require encrypted firmware upgrade files option is enabled. Use the `--seunencrypted` option in Simplicity Commander v1.13.0 or higher (see example below) to generate an encrypted GBL upgrade image file with a SE image outside the encrypted part of the file.

```
commander gbl create se-upgrade.gbl --seupgrade secure-element.seu --seunencrypted --app myapp.s37
--encrypt aes_key.txt
```

- In Series 3 devices, the storage bootloader supports direct SE upgrades from the GBL file without the need for staging.

The following sections provide two methods to sign the bootloader, Secure Engine, and application upgrade image files if the Require signed firmware upgrade files option is enabled.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

The sections also include encryption examples with an AES-128 key (like `aes_key.txt`) for the Require encrypted firmware upgrade files option. All procedures assume the required files are in the same folder.

Series 3 devices require GBLv4. GBLv4 is a new format that supports more command line options than the upgrade file for series 2 devices which do not support GBLv4.

- To compress an upgrade file, append `--compress <compression algorithm>` to the command, supported algorithms are `lz4` and `lzma`.
- To add the product id, append `--productid <16-byte long identifier>` to the command.
- To add the bundle version and minimum version, append `--bundleversion <version number> --minversion <version number>` to the command.

Read more about GBLv4 files and the configurations supported in Gecko Bootloader File Format v4 section of Gecko Bootloader User's Guide for Series 3 and Higher.

- All `gbl create` and `gbl4 create` commands support both encrypted and unencrypted images. To create an unencrypted image, omit the `--encrypt` option. To create an encrypted image, append `--encrypt aes_key.txt` to the command.

Bootloader Upgrade

1. Create an unsigned GBL upgrade image:
   a. Series 2 devices: Run the `gbl create` command with `--bootloader` option to generate the bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
```

```sh
Initializing GBL file...
Adding bootloader to GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

b. Series 3 devices: Run the `gbl4 create` command with `--data` option to generate the bootloader GBL upgrade image file (`bootloader.gbl`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

```sh
commander gbl4 create bootloader.gbl --data bootloader-uart-xmodem.s37 --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bootloader.s37...
Writing GBL file bootloader.gbl...
DONE
```

2. Create a signed GBL upgrade image using Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--bootloader` option to generate the signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with Private Sign Key or Private Bootloader Key and the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

```sh
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37 --sign
sign_key.pem
```

```sh
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--sign bl_cert_key.pem
```

```sh
Initializing GBL file...
Adding bootloader to GBL...
Signing GBL...
Image SHA256: 3eb09993ffca5f9b34df3f38b65ab9d2f6619b828b014a186516016d4bbd80f7
R = C21E0C19254AC4F62374BBCA65DEBB42C7349384F5527330CD030A51DC2170F7
S = E1680C3670DE68D731086845E2726EF3BF07B96EB54AA2DB2F390F60BDB6DAB2
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

b. Series 3 devices: Run the `gbl4 create` command with `--data` option to generate the signed bootloader GBL upgrade image file (`bootloader.gbl`) with Private Sign Key or Private Bootloader Key and the signed bootloader image file (`bootloader-u art-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

````
```sh
commander gbl4 create bootloader.gbl --data bootloader-uart-xmodem.s37 --sign sign_key.pem
--device SIMG301LIL
```

```sh
commander gbl4 create bootloader.gbl --data bootloader-uart-xmodem.s37 --sign bl_cert_key.pem
--device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bootloader.s37...
Writing GBL file bootloader.gbl...
Image SHA256: 5ba67b656103a3539f0f3802a4398f0babb3ff182fe9890962e678052571d42b
R = 6845219257E660918B7BAE26FFF3DC7BBE8150DA3281CE8881B7088E9E9736EC
S = B8C5B58987FA3C7E44444B7C43096A41F48D245ED148C4E0863F0F910133811E
DONE
```
````

3. Creating a signed GBL upgrade image using an HSM and Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--bootloader` and `--extsign` options to generate an external signing bootloader GBL upgrade image file (`bootloader-uart-xmodem.extsign`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl create bootloader-uart-xmodem --bootloader bootloader-uart-xmodem.s37 --extsign
```

```sh
Initializing GBL file...
Adding bootloader to GBL...
Preparing GBL for external signing...
Writing GBL file bootloader-uart-xmodem.extsign...
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing bootloader GBL upgrade image file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the gbl_signature.der.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

Run the `gbl sign` command with the **signature** above to generate a signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`).

```sh
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der --verify
sign_pubkey.pem
--outfile bootloader-uart-xmodem.gbl
```

```sh
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile bootloader-uart-xmodem.gbl
```

```sh
Reading GBL data from bootloader-uart-xmodem.extsign...
Parsing signature file gbl_signature.der...
R = 90F0A3C0D5D9ED2DC10EB3F55595FF21AB31307DC6283E3F3B7494A30FB741D4
S = 2765041F515A960F048CA250BFAB92031D4D1E569FB3F917C9329E7362C17B51
Writing signature to GBL... Verifying GBL...
Successfully verified GBL signature
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

b. Series 3 devices: Run the `gbl4 create` command with `--data` and `--extsign` options to generate an unsigned and manifest files (`bootloader.unsigned` and `bootloader.manifest`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl4 create bootloader --data bootloader-uart-xmodem.s37 --extsign --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bootloader-uart-xmodem.s37...
Writing manifest data (for external signing) to bootloader.manifest
Writing unsigned GBL file (for later use in 'gbl4 sign') to bootloader.unsigned...
After computing the signature of bootloader.manifest, assemble the GBL4 using:
commander gbl4 sign bootloader.unsigned --outfile bootloader --signature <signature.der>
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the manifest file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der bootloader.manifest
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der bootloader.manifest
```

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the bootloader with the bootloader GBL upgrade image file.

Secure Engine Upgrade

1. Creating an unsigned GBL upgrade image:
   a. Series 2 devices: Run the `gbl create` command with `--seupgrade` option to generate the SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

```sh
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
```

```sh
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

   b. Series 3 devices: Run the `gbl4 create` command with `--seupgrade` option to generate the SE GBL upgrade image file (`se-upgrade.gbl`) with the SE image file (`x301_se_fw_upgrade_3v3p2.seuv2`).

```sh
commander gbl4 create se-upgrade.gbl --seupgrade x301_se_fw_upgrade_3v3p2.seuv2 --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Reading SE upgrade file from x301_se_fw_upgrade_3v3p2.seuv2
Writing GBL file se-upgrade...
DONE
```

2. Creating a signed GBL upgrade image using Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--seupgrade` option to generate the signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with Private Sign Key or Private Bootloader Key and the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign sign_key.pem
```

```sh
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign bl_cert_key.pem
```

```sh
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Signing GBL...
Image SHA256: 599d7fc35996b4715441b642709ed262525d09d811d4726e423c0d605ec0f0bf
R = EF8EC2DDEDDF44DF88FEAD4ED0A9FDC6351B4D745D5A05BFB87204791871A525
S = FCB26EF005D97E8C5341153A210AE9927E1CF646A3E473FFB90DA8C857E6421F
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

b. Series 3 devices: Run the `gbl4 create` command with `--seupgrade` option to generate the signed SE GBL upgrade image file (`se-upgrade.gbl`) with Private Sign Key or Private Bootloader Key and the SE image file (`x301_se_fw_upgrade_3v3p2.seuv2`).

```sh
commander gbl4 create se-upgrade.gbl --seupgrade x301_se_fw_upgrade_3v3p2.seuv2
--sign sign_key.pem --device SIMG301LIL
```

```sh
commander gbl4 create se-upgrade.gbl --seupgrade x301_se_fw_upgrade_3v3p2.seuv2
--sign bl_cert_key.pem --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Reading SE upgrade file from x301_se_fw_upgrade_3v3p2.seuv2
Writing GBL file se-upgrade.gbl...
Image SHA256: e6462d8810f9afee36974b82b3790af55c94f559c532ee0e264617d705a1fd6b
R = 095A9D64BD490FCA444D49DADC53C666B717006D9E7C91C196B259F6E27E5C9C
S = 0BFC9B86BA5B3C2A4C98AE5EC8264D905659AE8528433832FAD225D51E9899FD
DONE
```

3. Creating a signed GBL upgrade image using an HSM and Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--seupgrade` and `--extsign` options to generate an external signing SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.extsign`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl create s2c1_se_fw_upgrade_1v2p9 --seupgrade s2c1_se_fw_upgrade_1v2p9.seu --extsign
```

```sh
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Preparing GBL for external signing...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.extsign...
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing SE GBL upgrade image file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

Run the `gbl sign` command with the **signature** above to generate a signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`).

```sh
commander  gbl sign    s2c1_se_fw_upgrade_1v2p9.extsign    --signature gbl_signature.der   --verify
sign_pubkey.pem
--outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

```sh
commander gbl sign s2c1_se_fw_upgrade_1v2p9.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

```sh
Reading GBL data from s2c1_se_fw_upgrade_1v2p9.extsign...
Parsing signature file gbl_signature.der...
R = 2798B98194EE02717C738B5866ABD8D234D0F0E096E90495D371D2507D8E1C67
S = 19F2586E2C6177D6B4EEC708E006F67334C989D0398D4233C686C98ECB6992FB
Writing signature to GBL... Verifying GBL...
Successfully verified GBL signature
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

b. Series 3 devices: Run the `gbl4 create` command with `--seupgrade` and `--extsign` options to generate unsigned and manifest files (`se-upgrade.unsigned` and `se-upgrade.manifest`) with the SE image file (`x301_se_fw_upgrade_3v3p2.seuv2`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl4 create se-upgrade --seupgrade x301_se_fw_upgrade_3v3p2.seuv2 --extsign
--device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Reading SE upgrade file from x301_se_fw_upgrade_3v3p2.seuv2
Writing manifest data (for external signing) to se-upgrade.manifest
Writing unsigned GBL file (for later use in 'gbl4 sign') to se-upgrade.unsigned...
After computing the signature of se-upgrade.manifest, assemble the GBL4 using:
commander gbl4 sign se-upgrade.unsigned --outfile se-upgrade --signature <signature.der>
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the manifest file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der se-upgrade.manifest
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der se-upgrade.manifest
```

Run the `gbl4 sign` command with the **signature** above to generate a signed SE GBL upgrade image file (`se-upgrade.gbl`).

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the SE with the SE GBL upgrade image file.

Notes:

- The `sign_key.pem` / `sign_pubkey.pem` key pair is for Signing for ECDSA-P256-SHA256 Secure Boot, and the `bl_cert_key.pem` / `bl_cert_pubkey.pem` key pair is for Signing for Certificate-Based Secure Boot.
- Trying to apply a lower version of the SE image file (`.seu`) to the device will be ignored.

Application Upgrade

1. Creating an unsigned GBL upgrade image:
   a. Series 2 devices: Run the `gbl create` command with `--app` option to generate the application GBL upgrade image file (`blink_baremetal.gbl`) with the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

```sh
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37
```

```sh
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Writing GBL file blink_baremetal.gbl...
DONE
```

   b. Series 3 devices: Run the `gbl4 create` command with `--data` option to generate the bootloader GBL upgrade image file (`app-upgrade.gbl`) with the signed bootloader image file (`bt_soc_blinky_freertos.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl4 create app-upgrade.gbl --data bt_soc_blinky_freertos.s37 --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bt_soc_blinky_freertos.s37...
Writing GBL file app-upgrade.gbl...
DONE
```

2. Creating a signed GBL upgrade image using Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--app` option to generate the signed application GBL upgrade image file (`blink_baremetal.gbl`) with Private Sign Key or Private Bootloader Key and the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

```sh
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign sign_key.pem
```

```sh
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign bl_cert_key.pem
```

```sh
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Signing GBL...
Image SHA256: 116c1be47d799ab75afc7b3f4c9a8023e5cd031103b1d28c578eebfaf1ad73d2
R = CE4D85C058301A2437440E00385D97E496F1D8B5CAFFB8C184F8A88B5266E3E9
S = 90BBF754EBC0AB343CC32AA06ADED85F9D12D1A67CA6608F9085137142000A40
Writing GBL file blink_baremetal.gbl...
DONE
```

   b. Series 3 devices: Run the `gbl4 create` command with `--data` option to generate the signed bootloader GBL upgrade image file (`app-upgrade-signed.gbl`) with Private Sign Key or Private Bootloader Key and the signed bootloader image file (`bt_soc_blinky_freertos.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

```sh
commander gbl4 create app-upgrade-signed.gbl --data bt_soc_blinky_freertos.s37
--sign sign_key.pem --device SIMG301LIL
```

```sh
commander gbl4 create app-upgrade-signed.gbl --data bt_soc_blinky_freertos.s37
--sign bl_cert_key.pem --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bt_soc_blinky_freertos.s37...
Writing GBL file app-upgrade-signed.gbl...
Image SHA256: 665ee8c703bc57ee59078d48183d39087e6a52e564ee3bd73971f33acbd0b009
R = 5C72EDFA23B947BAB1827A566C14BDF4542C24CE62F30DE928216AC10FE8AA5B
S = 22A4912CA192274681FB000F882D0CE500FC3D29C54CB354430553EC4A61B11A
DONE
```

3. Creating a signed GBL upgrade image using an HSM and Simplicity Commander:
   a. Series 2 devices: Run the `gbl create` command with `--app` and `--extsign` options to generate an external signing application GBL upgrade image file (`blink_baremetal.extsign`) with the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

````
```sh
commander gbl create blink_baremetal --app blink_baremetal.s37 --extsign
```

```sh
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Preparing GBL for external signing...
Writing GBL file blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing application GBL upgrade image file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der blink_baremetal.extsign
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der blink_baremetal.extsign
```

Run the `gbl sign` command with the **signature** above to generate a signed application GBL upgrade image file (`blink_baremetal.gbl`).

```sh
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile blink_baremetal.gbl
```

```sh
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify bl_cert_pubkey.pem
--outfile blink_baremetal.gbl
```

```sh
Reading GBL data from blink_baremetal.extsign...
Parsing signature file gbl_signature.der...
R = 533499660E24F1620EF25D862FB607F46E9E4ECC41CBDECBE77C64EF1970D96A
S = FA8901878218F5F1DB0FAF8B074CE98A27C63FFDE63730CD49EE47E847B9811D
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signature
Writing GBL file blink_baremetal.gbl...
DONE
```
````

b. Series 3 devices: Run the `gbl4 create` command with `--data` and `--extsign` options to generate an unsigned and manifest files (`app-upgrade.unsigned` and `app-upgrade.manifest`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```sh
commander gbl4 create app-upgrade --data bt_soc_blinky_freertos.s37 --extsign --device SIMG301LIL
```

```sh
Initializing GBLV4 file...
Parsing file bt_soc_blinky_freertos.s37...
Writing manifest data (for external signing) to app-upgrade.manifest
Writing unsigned GBL file (for later use in 'gbl4 sign') to app-upgrade.unsigned...
After computing the signature of app-upgrade.manifest, assemble the GBL4 using:
commander gbl4 sign app-upgrade.unsigned --outfile app-upgrade --signature <signature.der>
DONE
```

Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the manifest file. This example uses the [OpenSSL](#using-an-external-tool) with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der`.

```sh
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der app-upgrade.manifest
```

```sh
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der app-upgrade.manifest
```

Run the `gbl4 sign` command with the **signature** above to generate a signed bootloader GBL upgrade image file (`bootloader.gbl`).

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the application with the application GBL upgrade image file.

Notes:

- [Series 2 devices] The Simplicity Commander v1.11.0 or above supports GBL upgrade image file in `util verifysign` command.

```
commander util verifysign blink_baremetal.gbl --verify sign_pubkey.pem
```

```
Successfully verified GBL signature
DONE
```

- [Series 2 devices] The Simplicity Commander v1.12.0 or above fixes a bug introduced in v1.11.0 when using the `--extsign` option on the GBL upgrade image file.

Upload a GBL Upgrade Image File

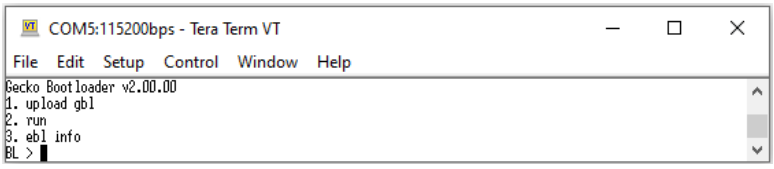
This section describes how to use UART XMODEM Bootloader v2.0.0 in GSDK v4.0 and higher to upload a GBL upgrade image file (`.gbl`) to the device. The procedures and pictures may be different for the other versions of this example.

The GBL upgrade image file uses a proprietary format to store the upgrade image for a firmware upgrade. Use the gbl create command to generate the GBL upgrade image file for bootloader, application, and Secure Engine. Refer to UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) and Generate a GBL Upgrade Image File for more information about GBL upgrade image file creation.

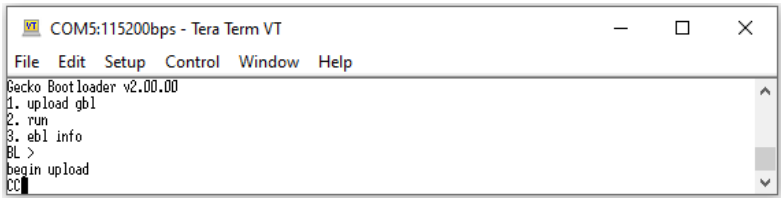
You can use any terminal software that supports the XMODEM-CRC protocol for file transfer. This application note uses Tera Term as terminal software. The default serial port setting is 115200 bps 8-N-1.

1. Assume the UART XMODEM Bootloader and application firmware had already flashed to the radio board on WSTK.
2. Press the RESET and PB0 push buttons on the WSTK.
3. Release the RESET push button to run the UART XMODEM Bootloader.

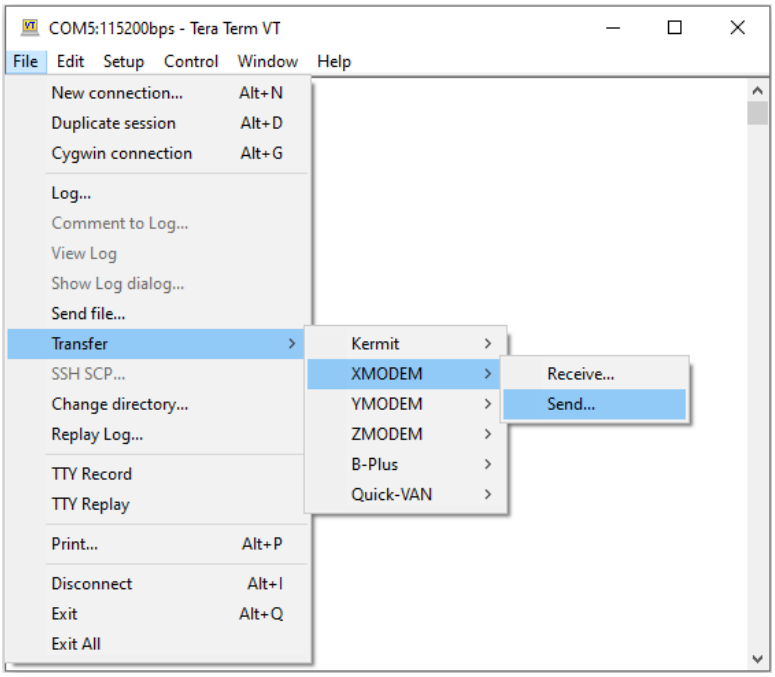
Copyright © 2025 Silicon Laboratories. All rights reserved.



4. Release the PB0 push button. Press 1 (`upload gbl`) in Tera Term to upload a GBL upgrade image file.

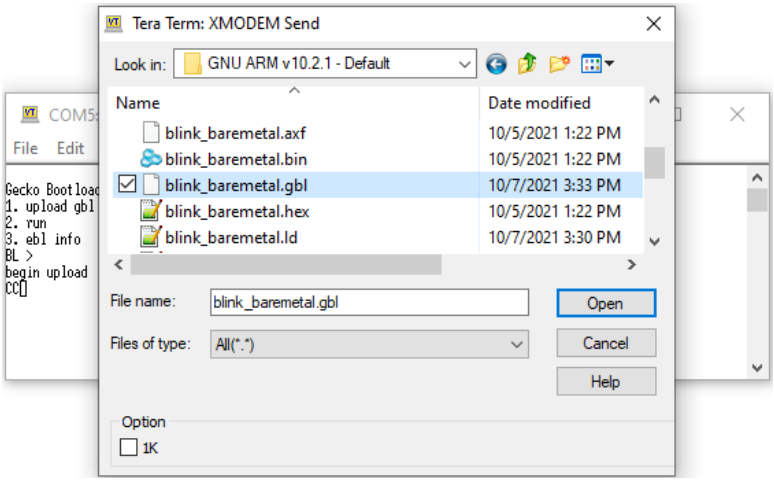


5. Transfer a file through XMODEM-CRC in Tera Term, navigate to File > Transfer > XMODEM > Send....

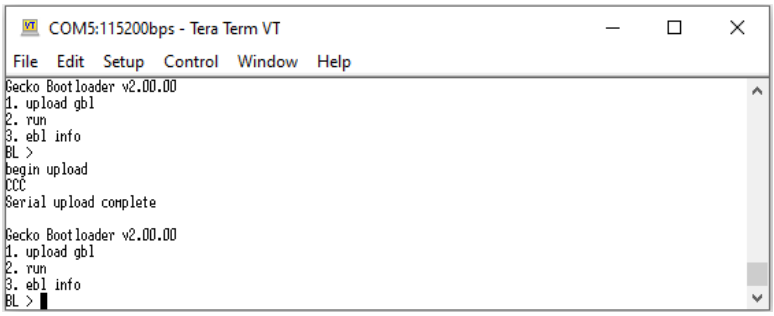


6. Select the target GBL upgrade image file. Click [Open] to upload.

Copyright © 2025 Silicon Laboratories. All rights reserved.



7. If no error occurs, press 2 (`run`) to start a firmware upgrade.



Upgrade to Certificate-Based Secure Boot

You can upgrade the Series 2 and Series 3 devices deployed in the field from ECDSA-P256-SHA256 Secure Boot to certificate-based Secure Boot even if the SECURE_BOOT_VERIFY_CERTIFICATE option in SE OTP is disabled.

```
commander security readconfig --serialno 440048205
```

```
MCU Flags
Secure Boot                    : Enabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback      : Enabled
Secure Boot Page Lock Narrow   : Disabled
Secure Boot Page Lock Full     : Disabled
```

The following procedures for the upgrade to certificate-based Secure Boot is an IRREVERSIBLE process.

1. Follow the procedures in Generate Key and Signing to generate an ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version `(--cert-version` in the `util gencert` command) in this signed GBL image file must be equal to or higher than one (≥ 1).
3. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the bootloader to certificate-based Secure Boot. Use the Private Sign Key for ECDSA-P256-SHA256 Secure Boot to sign the bootloader GBL upgrade image file if required.
   SE will use the Public Bootloader Key to validate the bootloader image once SE identifies a bootloader certificate in the bootloader image. If the bootloader certificate version from step 2 is higher than zero (> 0) and gets verified once, SE will never again accept the ECDSA-P256-SHA256 Secure Boot signed bootloader image. Refer to the "*Secure Boot Procedure*" section in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) for more information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045a2d
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x18 - Failed: Secure Boot requires cert, but none found
DONE
```

4. (Standard Certificate-Based) Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. (Advanced Certificate-Based) Follow the procedures in Generate Key and Signing to generate an ECDSA-P256 application certificate key pair.
   Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the application certificate key pair in this step and the Private Bootloader Key in step 1. The application certificate version (`--cert-version` in the `util gencert` command) in this signed application image file must be equal to or higher than the bootloader certificate version in step 2 (Private/Public Key Pair).
6. Follow the procedures in Generate a GBL Upgrade Image File to upgrade application with the signed image from step 4 or 5 for certificate-based Secure Boot. Use the Private Sign Key or Private Bootloader key in step 1 for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

Certificate Revocation

The certificate revocation is the act of invalidating a certificate when its private key shows signs of being compromised. The following procedures describe how to revoke the Series 2 and Series 3 devices' bootloader certificates deployed in the field.

1. Follow the procedures in Generate Key and Signing to generate a new ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version (`--cert-version` in the `util gencert` command) in this signed GBL image file must be higher than the certificate version in SE flash (see Private/Public Key Pair).
3. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the bootloader with the signed image from step 2. Use the Private Sign Key or existing Private Bootloader Key for certificate-based Secure Boot to sign the bootloader GBL upgrade image file if required.
4. (Standard Certificate-Based) Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. (Advanced Certificate-Based) Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1. The application certificate version (--cert-version in the util gencert command) in this signed application image file must be equal to or higher than the bootloader certificate version in step 2 (see Private/Public Key Pair).
   You should generate a new ECDSA-P256 application certificate key pair if the Private Application Key for the application certificate is compromised.
6. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the application with the signed image from step 4 or 5. Use the Private Sign Key or Private Bootloader key in step 1 for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

Upgrade to Secure Boot with RTSL

The following procedures describe upgrading Series 2 and Series 3 devices deployed in the field without Secure Boot to Secure Boot with RTSL.

1. (Recommended) Upgrade SE firmware to the latest version if available. See the *Gecko Bootloader Operation - Secure Engine Upgrade* section in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices).
2. Follow the procedures in AppBuilder or 3.4.1.2 Bootloader-core Software Component to prepare an unsigned GBL image with the required Secure Boot configuration for the application firmware.

Copyright © 2025 Silicon Laboratories. All rights reserved.

3. Follow the procedures in Generate Key and Signing to generate the ECDSA-P256 Sign Key pair for Secure Boot. The key pairs for the bootloader certificate and application certificate (advanced) are required if using Certificate-Based Secure Boot.
4. Follow steps 1 to 2 in SE Manager Key Provisioning Platform Example if this HSE GBL Decryption Key option is selected. Use the Public Sign Key in step 3 and follow steps 1 to 3 in SE Manager Key Provisioning Platform Example to generate an unsigned image. Use this image to create an application GBL upgrade image file.
5. The original GBL (application Secure Boot is disabled) boots into the unsigned SE Manager Key Provisioning Platform Example after upgrading the application with the image file in step 4.
6. Follow steps 5 to 8 in SE Manager Key Provisioning Platform Example to install the Public Sign Key to SE OTP and GBL Decryption Key (optional) to HSE OTP. Press SPACE instead of ENTER in step 9 to BYPASS the programming of the Secure Boot configuration in SE OTP.

```
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.

. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

   Notes:
   - Programming the Public Sign Key to the top page of the main flash (not included in this example) is required for the VSE device ECDSA-P256-SHA256 Secure Boot.
   - Programming the GBL Decryption Key to the top page of the main flash (not included in this example) is required if the default storage option for GBL Decryption Key is selected and the Require encrypted firmware upgrade files option is enabled in step 2.
7. Follow the signing procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot (Bootloader Image File section, skip the Secure Boot configuration for the bootloader) with the required key(s) generated in step 3 to sign the unsigned GBL image generated from step 2. Use this signed image to create a bootloader GBL upgrade image file.
8. Follow the signing procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot (Application Image File section) with the required key(s) generated in step 3 to sign the unsigned application image generated from step 4. Use this signed image to create an application GBL upgrade image file.

> Note: For the application bootloader with storage, you can generate a single GBL upgrade image file for signed images from steps 7 and 8.

9. The Secure Boot in SE OTP is not yet enabled, so FSB does not verify the signature when upgrading to the signed GBL in step 7. The updated GBL (application Secure Boot enabled) verifies the signature when upgrading or booting to the signed SE Manager Key Provisioning Platform Example in step 8.
10. Follow steps 9 to 10 (use SPACE to skip previous steps for OTP key programming) in SE Manager Key Provisioning Platform Example to program the required Secure Boot configuration in SE OTP for signed GBL.
11. Update a signed custom application firmware to replace the signed SE Manager Key Provisioning Platform Example used for Secure Boot with RTSL upgrade.

Notes:

- Refer to the "*Enabling Secure Boot RTSL on Series 2 and Series 3 Devices*" section (either Standalone Bootloaders or Application Bootloaders with Storage) in UG266/UG489 and Bootloader User Guide Series3 and higher for details.
- The SE Manager Key Provisioning Platform Example used here is just for reference. You can modify or write a new application to automate the processes for the Secure Boot with RTSL upgrade.
- If the Require signed firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be signed.
- If the Require encrypted firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be encrypted. And the GBL Decryption Key for the corresponding option in GBL must be in place.

Recover Devices when Secure Boot Fails

If a Secure Boot process fails (meaning firmware image at device starting address validation fails), the only way to recover is to flash a correctly signed image.

There are two scenarios to recover the device from a Secure Boot failure:

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. Debug Lock
2. BUSLOCK

DEBUG LOCK

The following table describes the different debug lock scenarios on recovering the Secure Boot failure device.

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Enabled	Disabled	Unlock	Flash a correctly signed image.
Disabled	Enabled	Enabled	Standard debug lock	Flash a correctly signed image after standard debug unlocking the device.
Disabled	Disabled	Enabled	Permanent debug lock	There is no way to recover the device. Make sure the programmed image is correctly signed before locking the device.
Enabled	Disabled	Enabled	Secure debug lock	Flash a correctly signed image after secure debug unlocking the device.

> Note: The error code in the Boot status of examples below depends on boot failure caused by the host image (GBL).

The following procedures describe how to recover the Secure Boot failure device from the lock states below.

- Unlocked
- Standard debug locked
- Secure debug locked

1. Follow the procedure in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to generate a correctly signed GBL.
2. (Unlocked) Run the `security status` command to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

3. (Standard debug locked) Run the `security status` command to get the boot status.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045afd
Debug lock          : Enabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `security erasedevice` command to unlock the device.

```
commander security erasedevice --device EFR32MG21A010F1024 --serialno 440048205
```

```
Successfully erased device
DONE
```

> Note: Issue a power-on or pin reset to complete the unlock process.

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

4. (Secure debug locked) Run the `security status` command to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000000d6ffffe0a3a5f
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Enabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `security unlock` command to unlock the device with the debug unlock token.

```
commander security unlock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Unlocking with unlock payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000000d6ffffe0a3a5f/chali

Secure debug successfully unlocked
DONE
```

Run the `flash` command with the `--noreset` option to flash the correctly signed image (like `bootloader-uart-xmodem.s37`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander flash --noreset bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Erasing range 0x00000000 - 0x00003FFF (2 sectors, 16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

> Note: The `--noreset` option prevents the device from returning to the secure debug lock state before flashing.

5. Run the `security status` command to check the boot status. The example below is an unlocked device.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Enabled
Boot status         : 0x20 - OK
DONE
```

BUSLOCK

When secure boot is enabled, the SE enforces the secure boot process through a hardware mechanism called BUSLOCK. This BUS-LOCK will halt the host-side (M33) bus so that it will not start the executing code. The BUSLOCK is enabled out of reset when secure boot is enabled.

- BUSLOCK status for secure boot success: BUSLOCK is enabled out of reset and is released only when the Secure boot process is successful and returns 0x20 - OK status code.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Enabled
Boot status         : 0x20 - OK
DONE
```

- BUSLOCK status for secure boot failure: BUSLOCK remains applied when the Secure Boot process fails and returns anything but 0x20 - OK status code. This keeps the BUSLOCK intact, and the host CPU is frozen, resulting in halting firmware execution on the M33 core. The device can be recovered from the BUSLOCK state by following these steps:

1. Issue a `security erase` command to the SE:

```
commander security erasedevice
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Successfully erased device
DONE
```

2. After issuing the `security erase` command to the SE, flash the correctly signed image to recover from a secure boot failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debugging on Secure Boot Enabled Device

Debugging on Secure Boot Enabled Device

Assume a correctly signed GBL image has been programmed to the device. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the GBL.

The Windows environment variable PATH should include the folder (`C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`) that locates the `commander.exe` of Simplicity Commander.

The following sections describe how to debug an application firmware with Simplicity IDE, or IAR on a Secure Boot enabled device.

Simplicity IDE

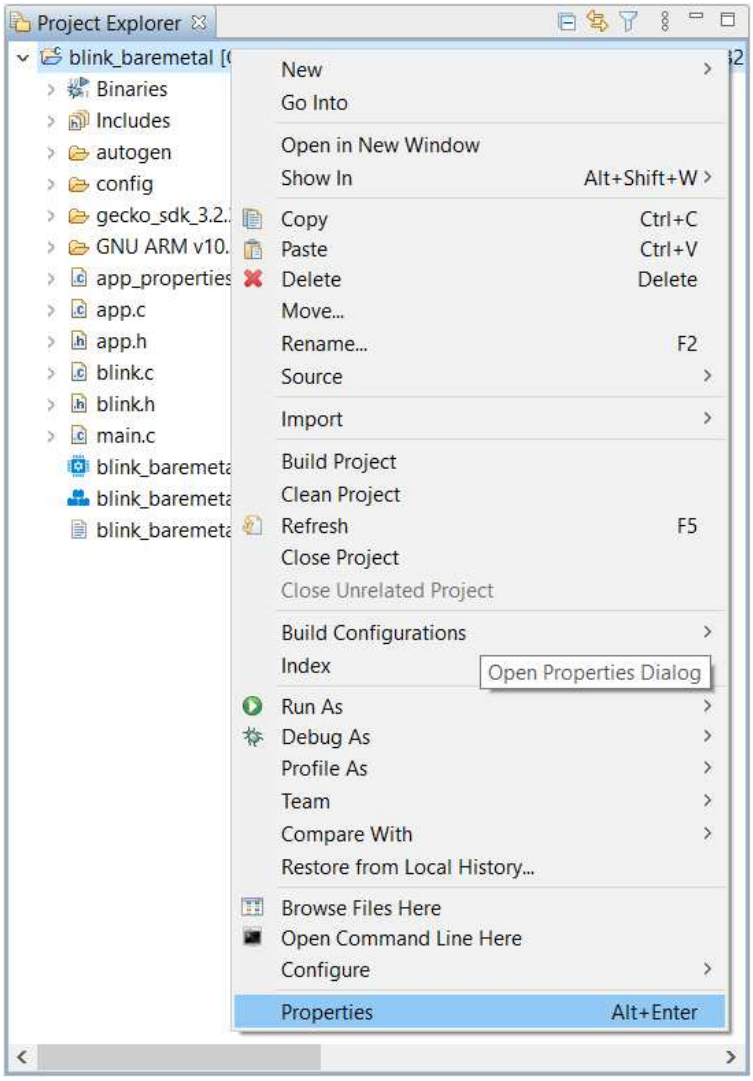
This application note uses Simplicity Studio v5.2.3.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. The Simplicity IDE creates a folder below (`<NAME>` is the Windows User Name on PC) in Windows when building the unsigned application image.
   `C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\GNU ARM v10.2.1 - Default`
2. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image and then flash it to the device. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a `secure_boot_debug.bat` file below.

   ```
   commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
   --outfile blink_baremetal.s37
   commander flash blink_baremetal.s37
   ```

3. Copy the batch file in step 2 and files (`sign_key.pem` and `sign_pubkey.pem` in this example) specified in `secure_boot_debug.bat` to the folder in step 1.
4. Right-click the project in the Project Explorer window, and then click Properties to open the properties dialog.

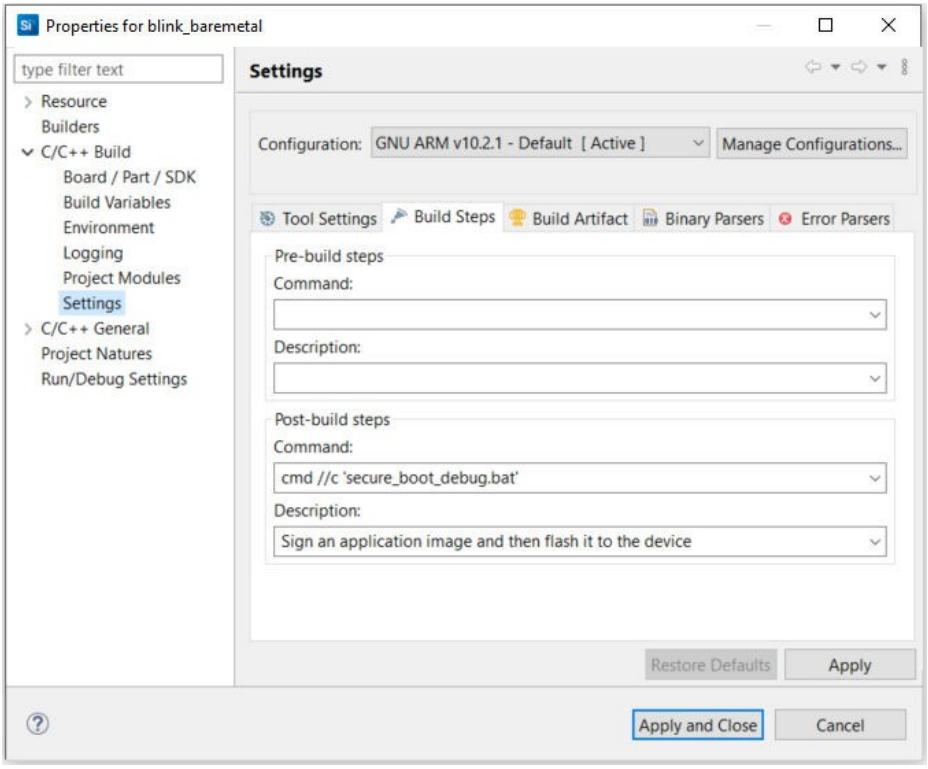
Copyright © 2025 Silicon Laboratories. All rights reserved.



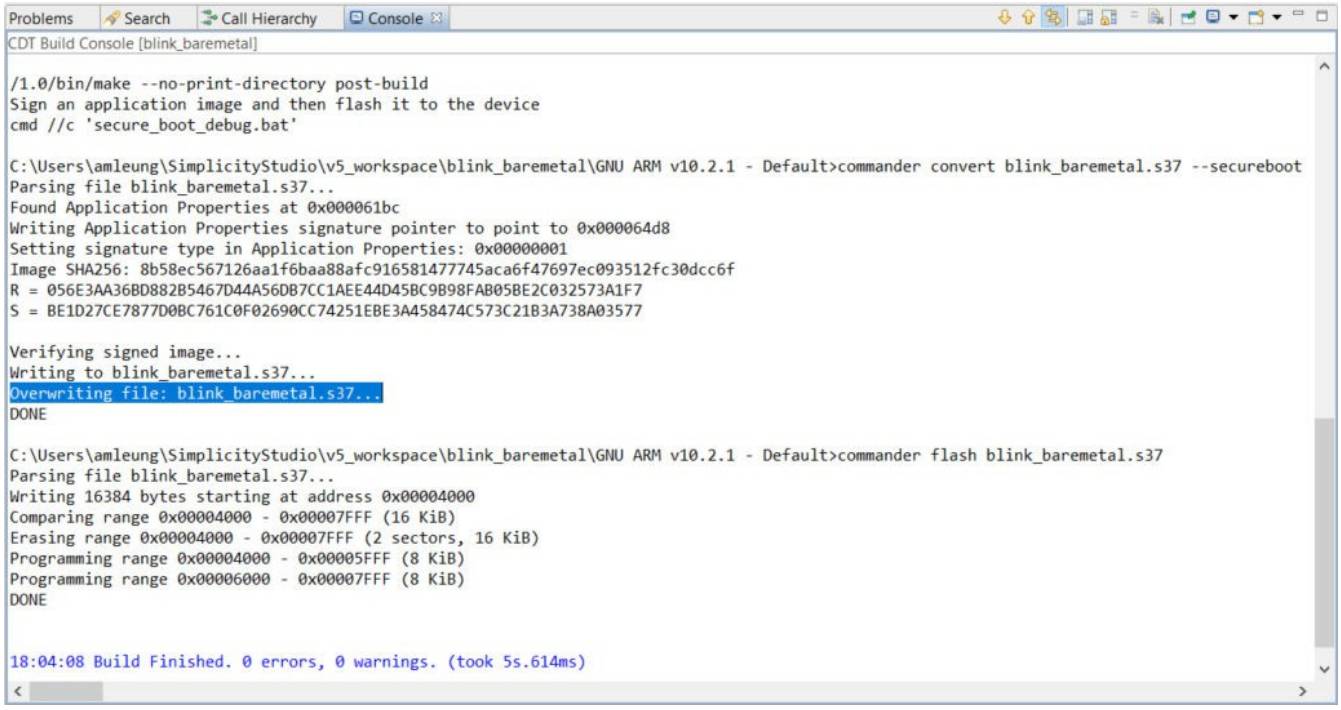
5. Select C/C++ Build > Settings > Build Steps. Enter the phrase below to the Command: box under the Post-build steps (enter text to Description: box is optional) to run the batch file as a post-build action. Click [Apply and Close] to exit.

```
cmd //c 'secure_boot_debug.bat'
```

Copyright © 2025 Silicon Laboratories. All rights reserved.



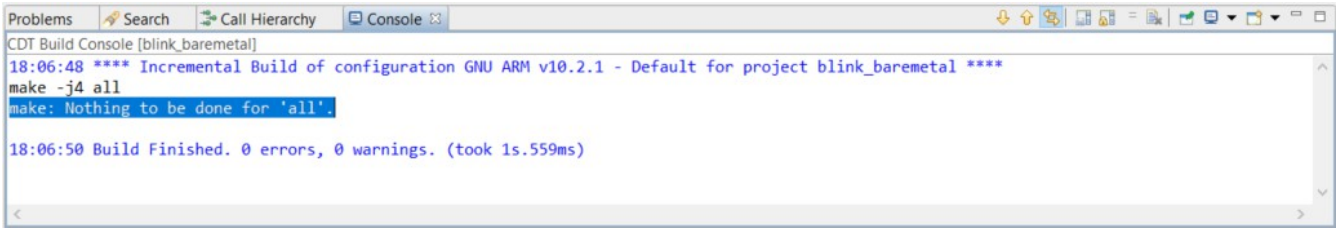
6. After building the project, the batch file in the Post-build steps overwrites the unsigned application image with the signed application image.



> Note: If the project is already up-to-date, it will not invoke the Post-build steps in step 5 to run the batch file. Use a dummy edit (add space or newline) on one of

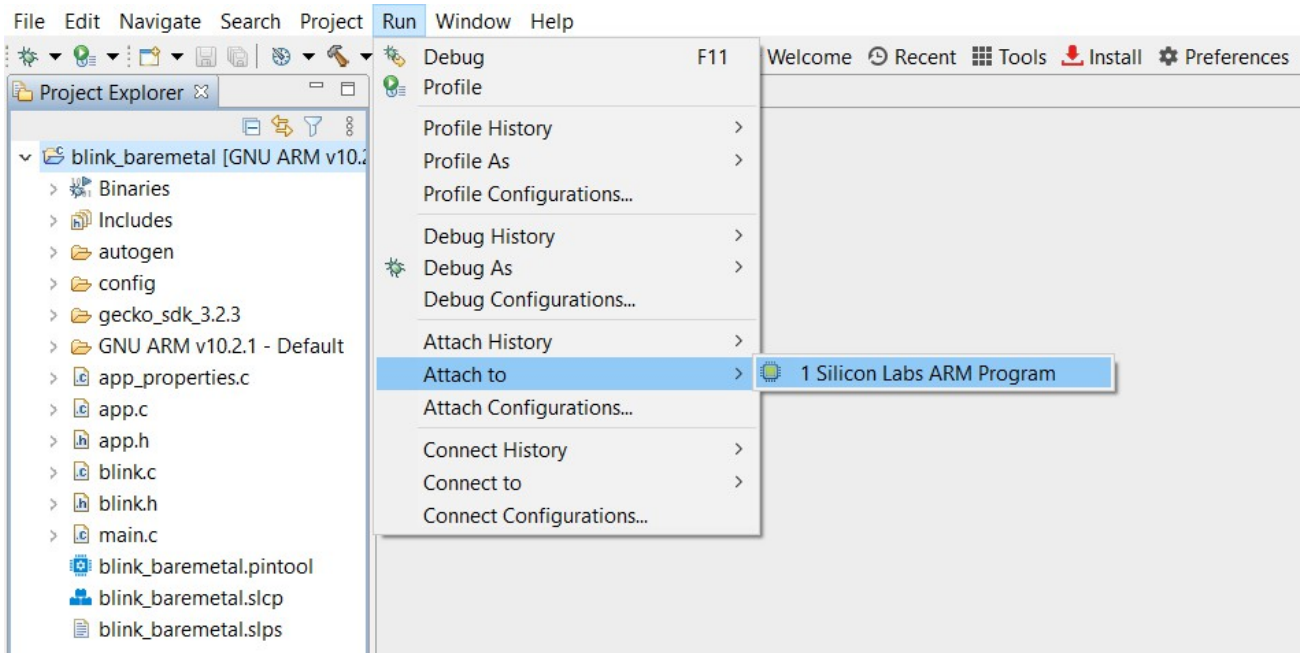
Copyright © 2025 Silicon Laboratories. All rights reserved.

```
    the source files in the project to trigger the build action.
```



The application starts to run if no error in step 6.

7. Select the project in the Project Explorer window, click Run→Attach to→1 Silicon Labs ARM Program to attach to the running target for debugging on the signed application image.

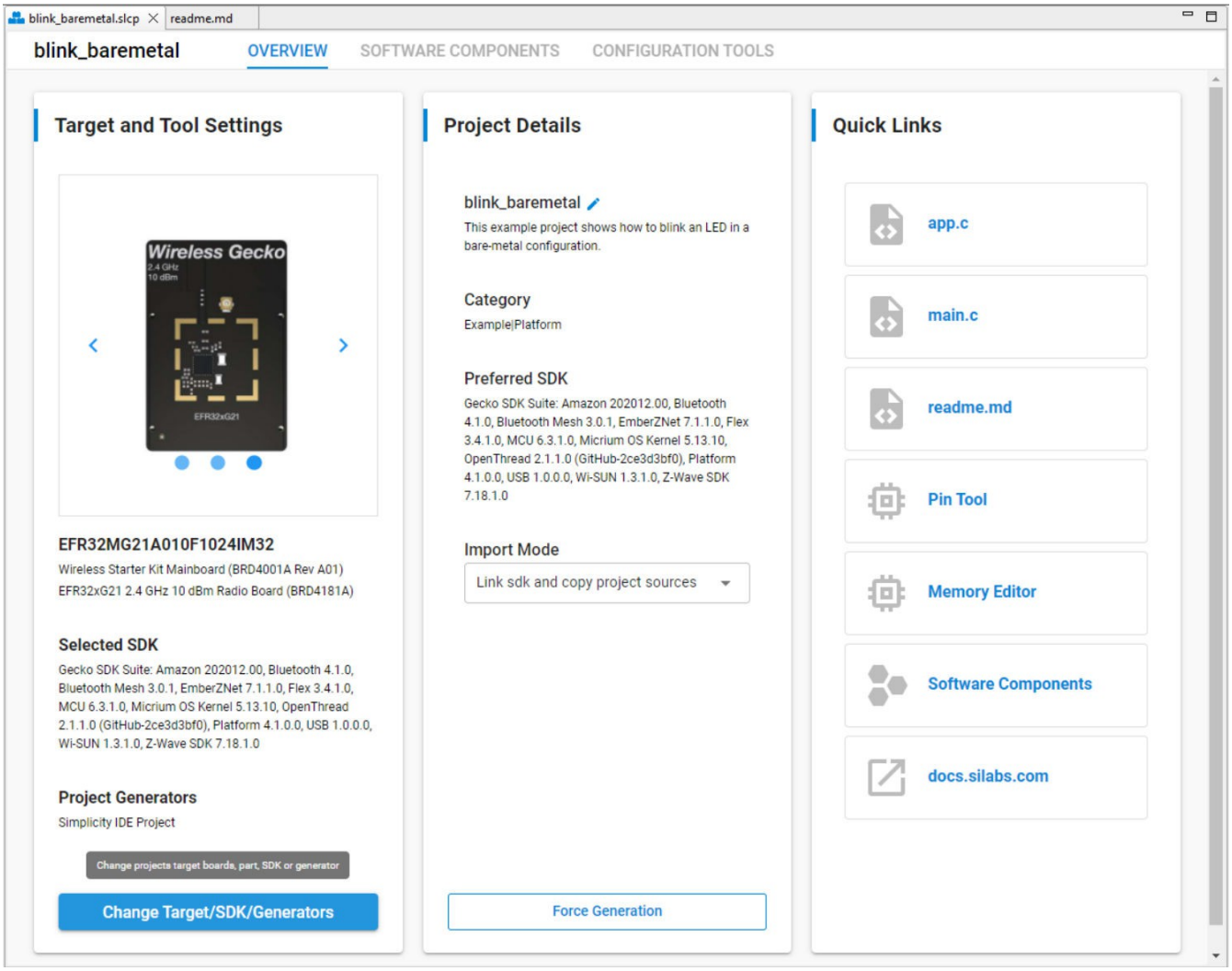


IAR

This section uses Simplicity Studio v5.4.2.0 and IAR v9.20.4. The procedures and pictures may be different for the other versions of Simplicity Studio 5 and IAR.

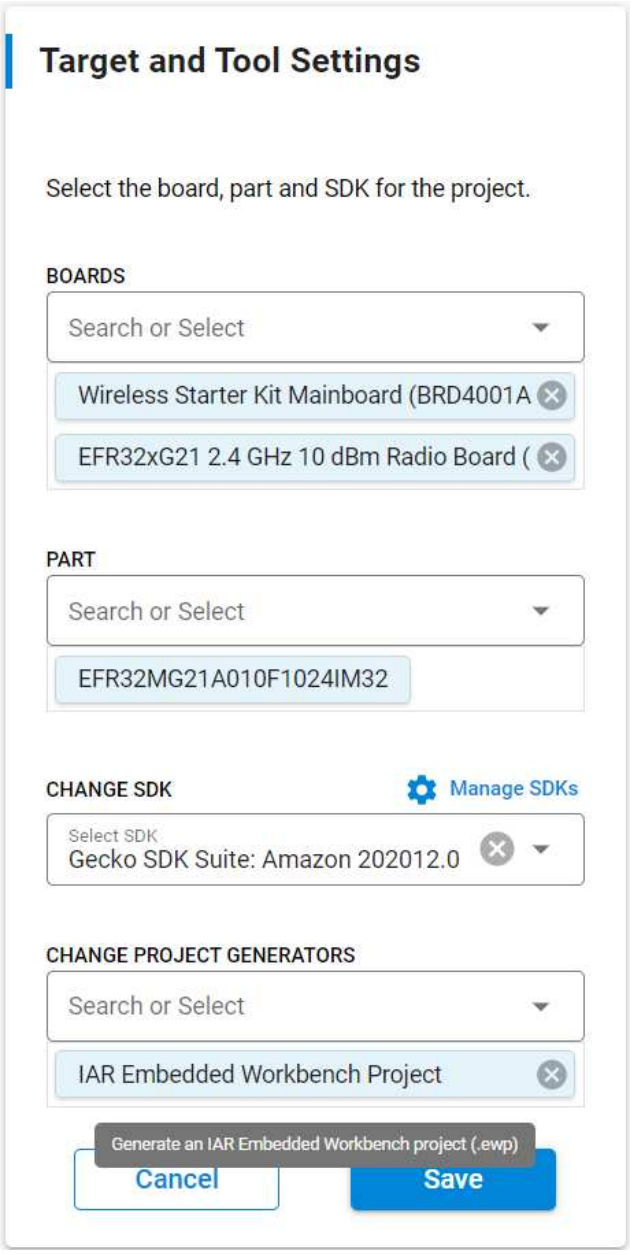
1. The Overview tab shows the Target and Tool Settings card on the left side. Scroll down if necessary and click [Change Target/SDK/Generators].

Copyright © 2025 Silicon Laboratories. All rights reserved.



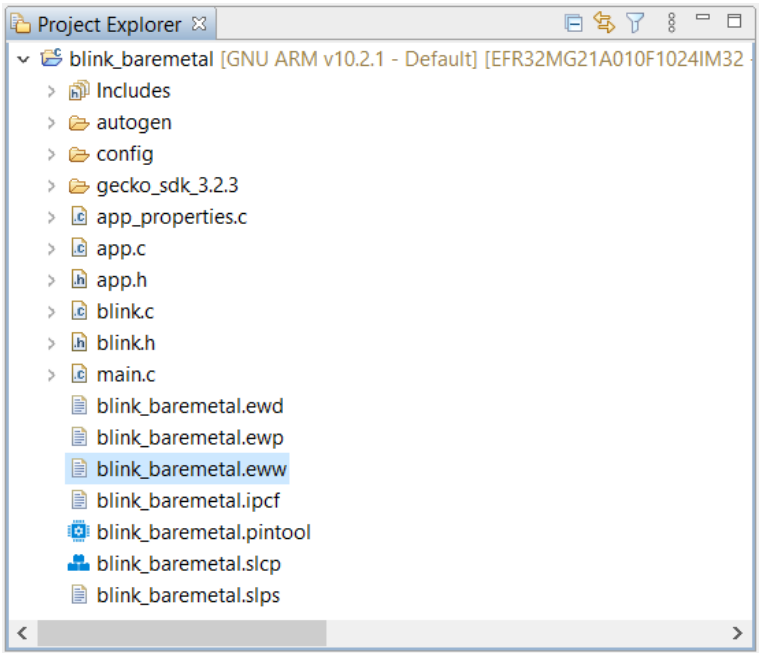
2. Drop down the CHANGE PROJECT GENERATORS list and select IAR Embedded Workbench Project. Click [Save] to generate an IAR project.

Copyright © 2025 Silicon Laboratories. All rights reserved.



3. Double click the IAR workspace file (`blink_baremetal.eww`) in the Project Explorer window to open the IAR project. The IAR creates a folder below (`<NAME>` is the Windows User Name on PC) in Windows to store the compiled image.
   `C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe`

Copyright © 2025 Silicon Laboratories. All rights reserved.

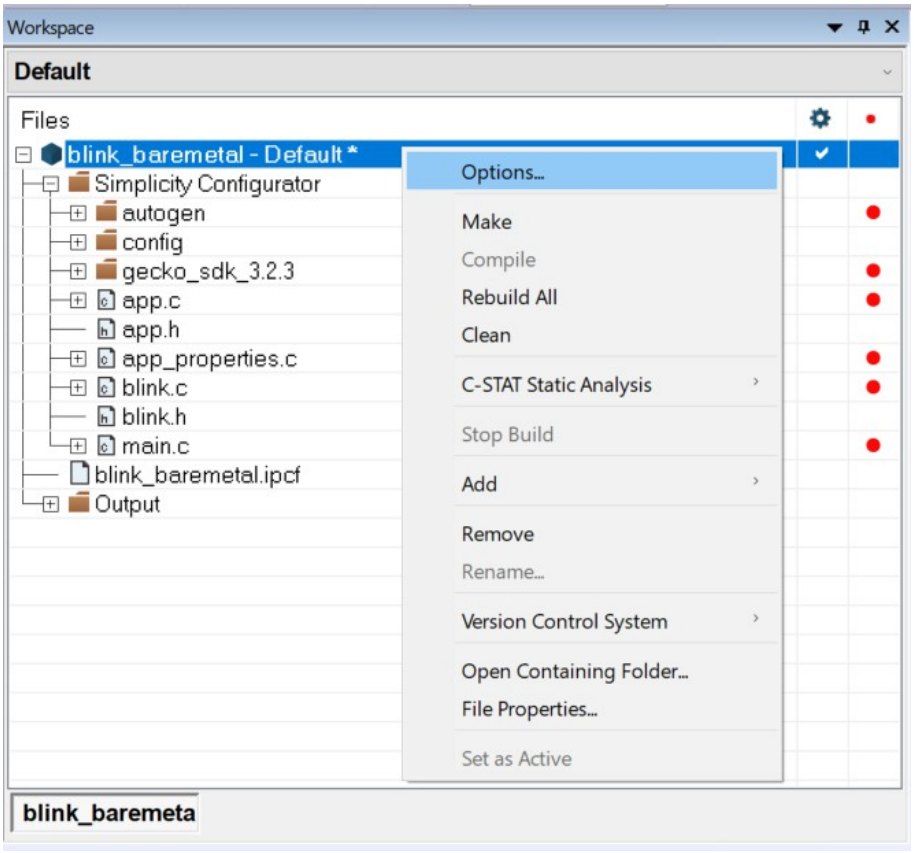


4. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a `secure_boot_debug.bat` file below.

```
cd C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

5. Copy the batch file in step 4 and files (`sign_key.pem` and `sign_pubkey.pem` in this example) specified in `secure_boot_debug.bat` to the folder in step 3.
6. Right-click the project in the workspace, and then click Options....

Copyright © 2025 Silicon Laboratories. All rights reserved.

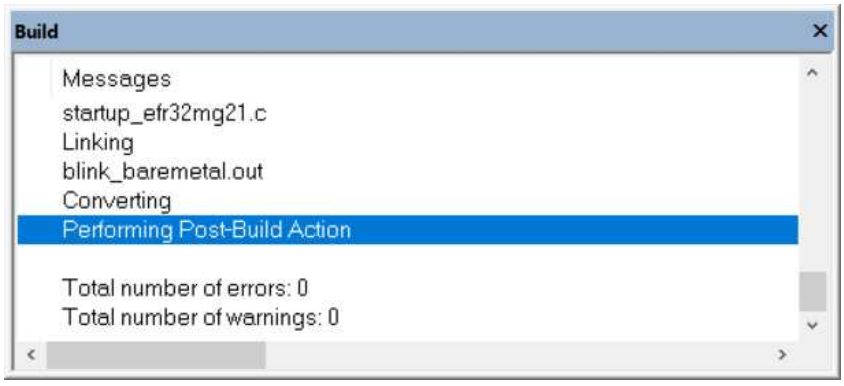


> Note: For GSDK v3.2 and lower, the `app_properties.c` is manually added to the IAR project.

7. Click Build Actions to open the Build Actions Configuration dialog box. Enter the phrase below to the Post-build command line: box to run the batch file as a post-build action. Click [OK] to exit.

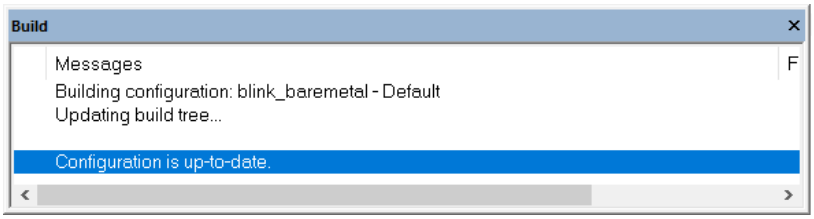
```
cmd /c "$PROJ_DIR$\ewarm-iar\exe\secure_boot_debug.bat > $PROJ_DIR$\log.txt 2>&1"
```

8. After building the project, the batch file in the Post-build command overwrites the unsigned application image with the signed application image.

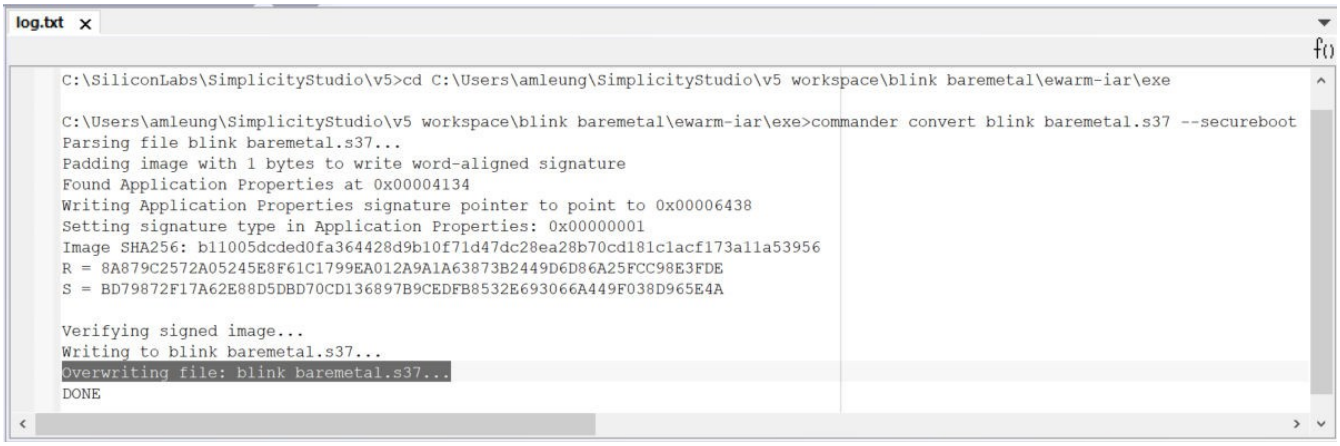


Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note: If the project is already up-to-date, it will not invoke the Post-build command in step 7 to run the batch file. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build action.



9. The `> $PROJ_DIR$\log.txt 2>&1` redirects the batch file output to the `log.txt` file in the IAR project folder.



10. If no error in step 8, click the  icon to start debugging on the signed application image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Failure Analysis

Failure Analysis

The following table describes the different scenarios when returning a Series 2 or Series 3 device to Silicon Labs for failure analysis.

State	Secure Boot Disabled	Secure Boot Enabled (2)
Standard debug unlock	Device erase is not necessary for failure analysis.	Device erase is not necessary, but a correctly signed image is required to perform failure analysis.
Standard debug lock	Device erase is required to perform failure analysis.	Require device erase and correctly signed image to perform failure analysis.
Permanent debug lock	Cannot perform failure analysis.	Cannot perform failure analysis.
Secure debug lock (1)	Require debug unlock token to perform failure analysis.	Require debug unlock token and correctly signed image to perform failure analysis.

Notes:

1. Follow the procedures in Series 2 and Series 3 Secure Debug section *Secure Debug Unlock and Roll Challenge - Simplicity Commander* to generate a valid debug unlock token for each device returned to Silicon Labs for failure analysis.
2. Secure boot enabled devices, especially with secure boot failure, may limit Silicon Labs' ability to determine the root cause of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Boot Status Codes

Secure Boot Status Codes

Boot status codes can be used to know the status of the boot mechanism. The `security status` command can be used to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number : 00000000000000014b457fffe045afd
Debug lock : Disabled
Device erase : Enabled
Secure debug unlock : Disabled
Tamper status : OK
Secure boot : Enabled
Boot status : 0x20 - OK
DONE
```

The following table shows Status codes in Secure Boot mechanism and their description:

STATUS CODE	DESCRIPTION
0x00	Start PUF
0x01	Fetch OTP for bootloader
0x02	Tamper test
0x03	Self-tests
0x04	TRNG Initialization failed
0x05	NVM Initialization failed
0x0B	Jump to main loop
0x0C	Fetch OTP for host boot
0x0D	Fetch pointers host firmware
0x0E	Fetch header host firmware
0x0F	Fetch host firmware
0x10	Check version host firmware
0x11	Check signature on certificate for host firmware
0x12	Check signature host firmware
0x13	Failed to get data from internal NVM
0x14	Finding host application properties pointer
0x15	Validating host application properties structure
0x16	Validating host application signature pointer
0x17	Getting SecureBoot key
0x18	SecureBoot requires cert, but none found
0x19	Updating required certificate version failed
0x1A	Certificate is of an older version as the last cert we validated
0x1B	Certificate structure version is not supported by this firmware

Copyright © 2025 Silicon Laboratories. All rights reserved.

STATUS CODE	DESCRIPTION
0x1C	Certificate pointer is out of range
0x1D	Region 0 is not closed
0x20	Main loop entered
0x80	PUF AC was somehow cleared
0x81	PUF failed to reconstruct after the longest delay
0x90	ESEC aborted booting due to catching too many successive tamper resets
0xFF	Finished verifying host app

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 TrustZone

Series 2 TrustZone

NOTE: This section replaces *AN1374: TrustZone*. Further updates to this user guide will be provided here.

ARMv8-M TrustZone is a technology that provides a foundation for improved system security in embedded applications. It allows the ARMv8-M to be aware of the security states of the system. Series 2 devices use the Cortex-M33 core to implement the ARMv8-M TrustZone security extension, which provides the ability to restrict access to peripherals and memory regions based on the processor security attribute. TrustZone works with the MPU, which controls privileged/unprivileged execution of code to provide a complete security solution.

ARMv8-M TrustZone is an extensive topic. The references below are publicly available on the ARM Developer Documentation website.

- ARMv8-M Architecture Reference Manual
- ARMv8-M Architecture Technical Overview
- ARM Cortex-M33 Processor Technical Reference Manual
- System Design with ARMv8-M
- TrustZone technology for ARMv8-M Architecture
- ARM Cortex-M33 Devices Generic User Guide
- Secure software guidelines for ARMv8-M
- Software Development in ARMv8-M ArchitectureReading guides:
- Beginner
- Minimal experience with TrustZone, starting with TrustZone Basics
- Intermediate - Have a basic understanding of the TrustZone technology, starting with Bus Level Security
- Advanced - Developed experience on TrustZone, starting with TrustZone Implementation-Demo - Starting with TrustZone Platform Examples

Key Points

- TrustZone Basics
- Bus Level Security (BLS)
- Secure and Privileged Programming Model
- TrustZone Implementation
- Upgrade Existing Application to TrustZone
- TrustZone Platform Examples

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure communication paths to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
Secure Vault Mid (SVM)	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

> Note: 1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug Access Control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

Series 2 devices require a specific SE firmware version to support the TrustZone implementation. Refer to Production Programming of Series 2 and Series 3 Devices to learn how to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

Series 2 devices use Cortex-M33 core to implement the ARMv8-M Mainline TrustZone security extension and refer to TrustZone as Bus Level Security. The following table lists the configuration of TrustZone related components in the Series 2 Cortex-M33 core.

Copyright © 2025 Silicon Laboratories. All rights reserved.

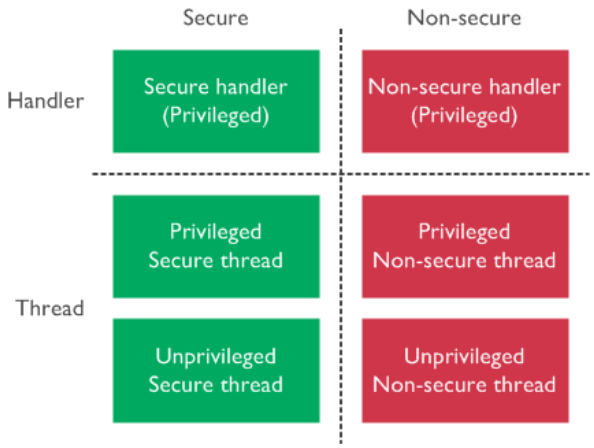
Component	Series 2 Configuration	Description
Security Extension (TrustZone)	Enabled	The security extension cannot be disabled, and the entire memory after RESET is Secure by default.
Memory Protection Unit (MPU)	16 regions (maximum)	The MPU regions for both Secure and Non-secure MPUs.
Security Attribution Unit (SAU)	8 regions (maximum)	The SAU regions for Non-secure and Non-secure Callable.

Copyright © 2025 Silicon Laboratories. All rights reserved.

TrustZone Basics

Introduction

TrustZone for ARMv8-M adds extra states to the Cortex-M processor operations to ensure there is a Secure and Non-secure state. These security states are orthogonal to the existing Thread and Handler modes, thereby having both a Thread and Handler mode in both Secure and Non-secure states. The Thread mode can also be either Privileged or Unprivileged.



Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

TrustZone for ARMv8-M is an optional architecture extension. By default, the system starts up in a Secure state if the processor implements the TrustZone security extension. The division of Secure and Non-secure worlds is memory-map based (security state depends on the address of the fetched instruction), and the transitions happen automatically. It is also possible to leave the Non-secure state unused and execute the whole application in the Secure state.

Memory Security Attributes

TrustZone classifies memory into four security attributes as described in the following table.

Security Attribute	Processor State	Description
Non-secure (NS)	Non-secure	Non-secure and Secure software can access these memory regions.
Secure (S)	Secure	Secure software can access these memory regions. Non-secure software cannot gain access to the Secure memory.
Non-secure Callable (NSC)	Secure	Secure memory with an NSC attribute provides entry points for Secure APIs that can be called from a Non-secure space. It is a region of memory that contains the Secure Gateway (SG) veneers that allow Non-secure code to call secure functions that exist in Secure code. Non-secure software cannot read/write to an NSC memory but can branch into it if the branch target is an SG instruction.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Security Attribute	Processor State	Description
Exempted	Secure/Non-secure	Non-secure and Secure software can access these memory regions (exempted from security checking). Exempted regions are typically used by debugging components that do not pose any security risk (e.g., system ROM table) when accessed by the Non-secure software.

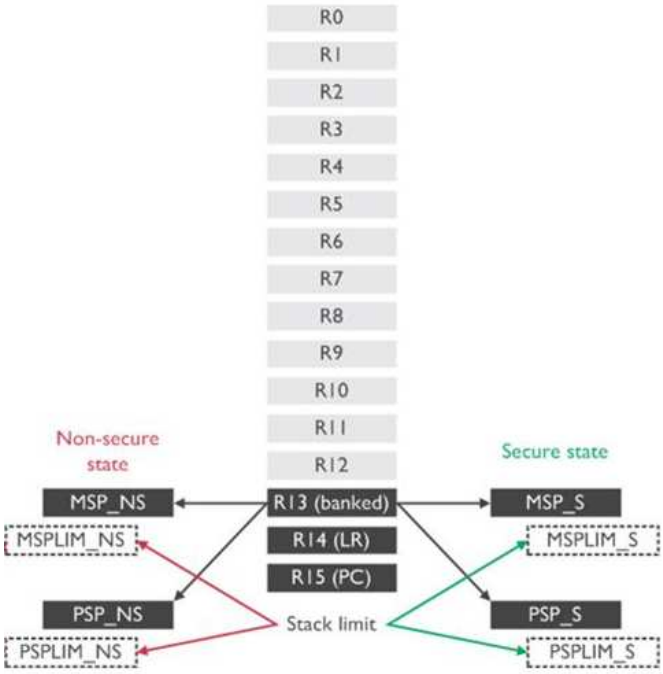
> Note: The Non-secure Callable is also known as Secure Non-secure Callable (Secure NSC) to declare that this region resides in Secure memory.

Banked Register

The concept of a banked register in ARMv8-M between Secure and Non-secure states means that there are two copies of the register, and the core automatically uses the copy that belongs to the current security state. When a register is banked, the `_S` and `_NS` suffixes are used in the ARMv8-M architecture to identify whether the resource is for the Secure state or Non-secure state.

General-Purpose Registers

The Cortex-M processors have 16 general-purpose registers (R0 - R15) for data processing (R0 - R12) and control. The following figure shows the general-purpose register view of the ARMv8-M system with TrustZone. Refer to the ARM Cortex-M33 Devices Generic User Guide for details about these registers.

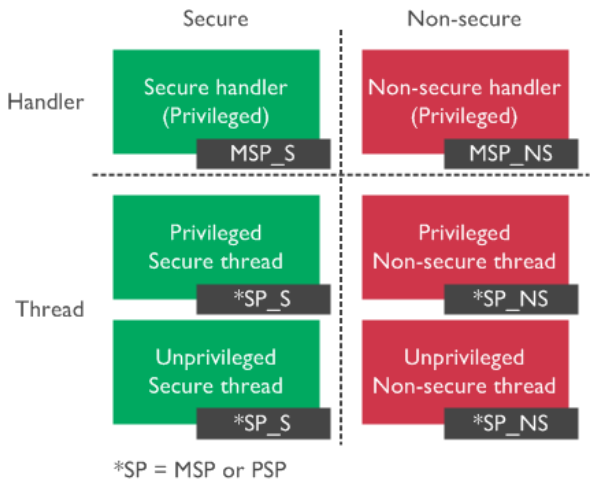


Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

The Secure or Non-secure state can access the data processing registers R0 - R12 and special usage registers R13 - R15. The register R13 (banked SP) is the stack pointer alias, and the actual stack pointer (`MSP_NS` , `PSP_NS` , `MSP_S` , `PSP_S`) accessed depends on the state (Secure or Non-secure) and mode (Handler or Thread) as described in the following figure.

In addition, stack limit registers (special registers) enable hardware to detect stack overflow conditions. Two pairs of stack limit registers (`MSPLIM_NS` and `PSPLIM_NS` , `MSPLIM_S` and `PSPLIM_S`) are implemented, one per security state, to protect the Main Stack Pointer (MSP) and Process Stack Pointer (PSP).

Copyright © 2025 Silicon Laboratories. All rights reserved.



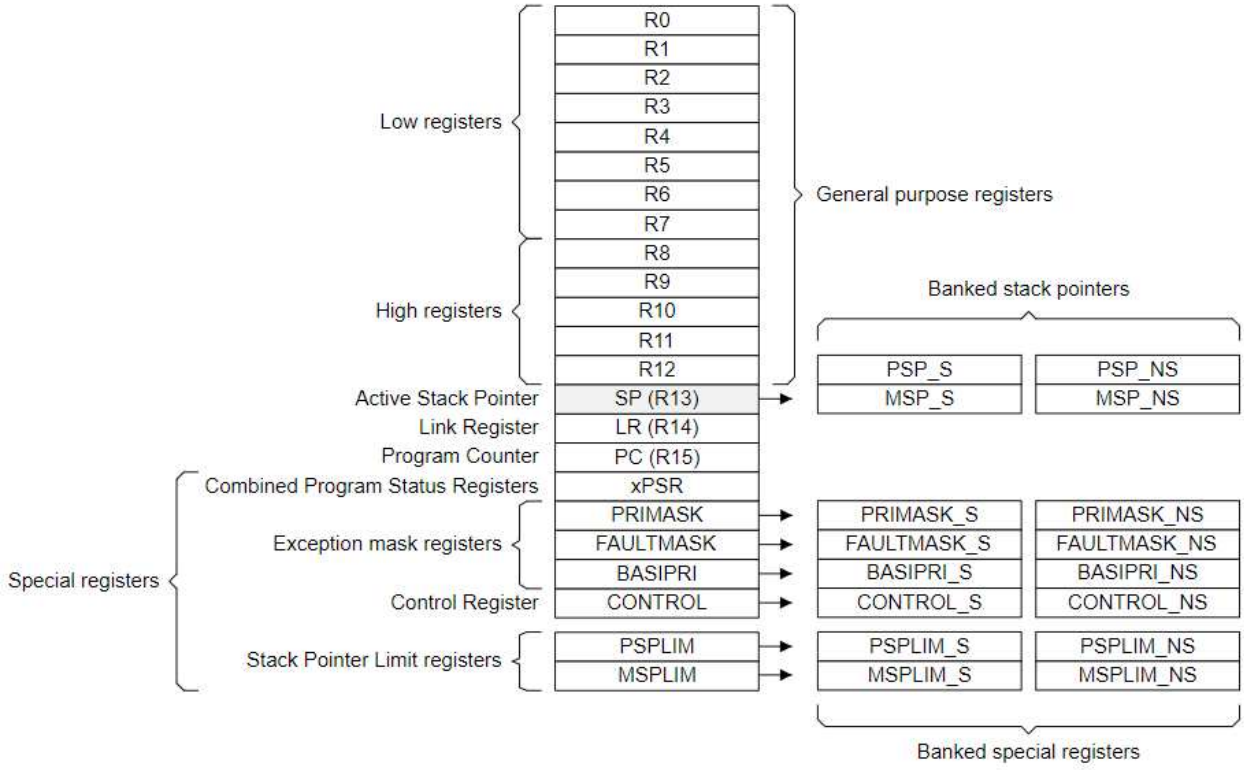
Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.

In Thread mode, execution can be privileged or unprivileged. The stack pointer used can be the MSP or PSP, depending on the `SPSEL` bit in the CONTROL register. When in Handler mode, the processor is Privileged. The stack pointer is always MSP.

It is possible to directly access the stack pointers (MSP and PSP) and stack limit registers (MSPLIM and PSPLIM), providing that the processor is in a privileged state. If the processor is in a Secure privileged state, the software can also access the Non-secure stack pointers (`MSP_NS` and `PSP_NS`) through Core Register Access Functions in CMSIS-Core.

Special-Purpose Registers

Except for the general-purpose registers, there are several special-purpose registers for conditional flags, interrupt masking, control, and stack pointer limit. The following figure shows the special-purpose registers view of the ARMv8-M system with TrustZone. Refer to the ARM Cortex-M33 Devices Generic User Guide for details about these registers.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Image:*https://documentation-service.arm.com/.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

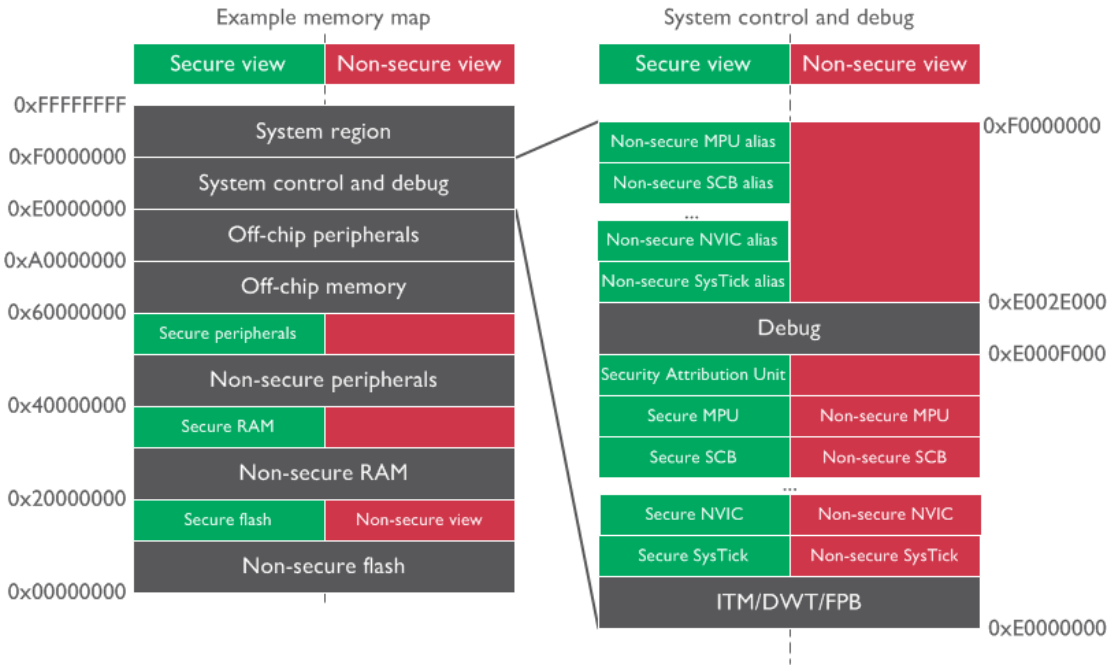
The Combined Program Status Register (xPSR) consists of the Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and Execution Program Status Register (EPSR).

Some of the special-purpose registers are banked between Secure and Non-secure states. Special-purpose registers are not memory-mapped and can be accessed using Core Register Access Functions in CMSIS-Core (except for EPSR in xPSR).

Secure privileged software can also access the Non-secure interrupt masking registers (`PRIMASK_NS` , `FAULTMASK_NS` , and `BASEPRI_NS`), CONTROL register (`CONTROL_NS`), and stack limit registers (`MSPLIM_NS` and `PSPLIM_NS`) through Core Register Access Functions in CMSIS-Core.

System Private Peripheral Bus (PPB)

The banking of registers is usually used to separate the Secure and Non-secure information of the system components inside the processor. The following figure shows the System Private Peripheral Bus (PPB) registers view of the ARMv8-M system with TrustZone. Refer to the ARM Cortex-M33 Devices Generic User Guide for details about the System PPB registers.



System components for debugging and trace operations (`0xE0000000` to `0xE0002FFF`):

- Instrumentation Trace Macrocell (ITM)
- Data Watch point and Trace unit (DWT)
- Flash Patch and Breakpoint unit (FPB)

System Control Space (SCS):

- The registers in SCS address spaces are memory-mapped and can be accessed using pointers in software
- Secure SCS (`0xE000E000` to `0xE000EFFF`) - Secure software using this address space to access the banked Secure SCS registers (e.g., `SCB->CPUID`)
- Non-secure SCS (`0xE000E000` to `0xE000EFFF`) - Non-secure software using this address space to access the banked Non-secure SCS registers (e.g., `SCB->CPUID`)
- Non-secure alias SCS (`0xE002E000` to `0xE002EFFF`) - Secure software using this address space to access the Non-secure SCS registers (e.g., `SCB_NS->CPUID`)

Copyright © 2025 Silicon Laboratories. All rights reserved.

The following table describes some core peripherals in the SCS and corresponding data structures defined in the CMSIS-Core header file to access the registers of core peripherals in two SCS address spaces.

Core Peripheral	Data Structure for Secure and NS SCS	Data Structure for NS Alias SCS
Implementation Control Block	SCnSCB (0xE000E004)	SCnSCB_NS (0xE002E004)
SysTick Timer	SysTick (0xE000E010)	SysTick_NS (0xE002E010)
Nested Vectored Interrupt Controller	NVIC (0xE000E100)	NVIC_NS (0xE002E100)
System Control Block	SCB (0xE000ECFC)	SCB_NS (0xE002ECFC)
Memory Protection Unit	MPU (0xE000ED90)	MPU_NS (0xE002ED90)
Security Attribution Unit	SAU (0xE000EDD0)	-
Debug Control Block	CoreDebug (0xE000EDF0)	CoreDebug_NS (0xE002EDF0)
Software Interrupt Generation	STIR (0xE000EF00)	STIR_NS (0xE002EF00)
Floating-Point Extension	FPU (0xE000EF34)	FPU_NS (0xE002EF34)

Notes:

- The SCB is a group of system control registers for the various usages below.
  - System Control Register (SCR) to configure processor low power mode
  - Fault Status Register (xFSR) to provide fault status information
  - Vector Table Offset Register (VTOR) for vector table relocation
- The SAU register is accessible from the Secure state only.
- The STIR register is not physically banked.
- Core peripherals such as SysTick, SCB, and MPU are duplicated. One instance is Secure and the other one is Non-secure.
- Secure software can use the corresponding functions for ARMv8-M in CMSIS-Core to configure the Non-secure NVIC and SysTick through the Non-secure alias SCS.

Debug or vendor specific components (`0xE0040000` to `0xE00FFFFF`):

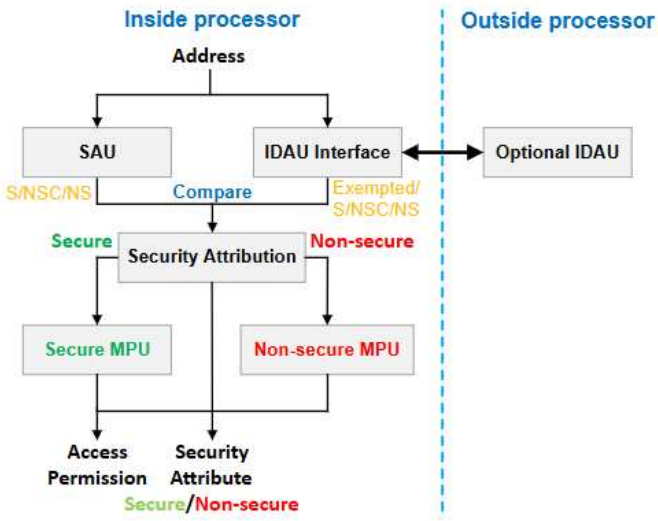
- Optional debug components (e.g., ETM)
- External Private Peripheral Bus (EPPB) allows designers to add their own debug or vendor-specific components
- System ROM Table is a simple lookup table that enables debug tools to extract the addresses of debug and trace components

Secure Attribution Unit (SAU), Implementation Defined Attribution Unit (IDAU), and Memory Protection Unit (MPU)

Two units determine the security attribute of an address:

1. The internal programmable Secure Attribution Unit (SAU).
2. The external Implementation Defined Attribution Unit (IDAU), through the IDAU interface, returns the security attribute and region number of an address.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Three possible configurations to define the security attribute of an address:

1. Internal SAU only
2. External IDAU only
3. A combination of the internal SAU and external IDAU

Notes:

- Series 2 devices use configuration 3.
- IDAU in Series 2 devices is the External Secure Attribution Unit (ESAU).

The Memory Protection Unit (MPU) is a programmable unit that allows privileged software to define memory access permission. If the TrustZone is enabled, there can be up to two MPUs, one for Secure and one for Non-secure.

- The number of MPU regions for the Secure and the Non-secure MPU can be different.
- The MPU registers are memory-mapped and are placed in the System Control Space (SCS).
- Secure software can use the MPU Functions for ARMv8-M in CMSIS-Core to configure the Non-secure MPU through the Non-secure alias SCS (`0xE002ED90` - `0xE002EDC4`).

Software	Non-secure MPU Registers	Secure MPU Registers	MemManage Fault
Non-secure privileged	0xE000ED90 - 0xE000EDC4	-	Non-secure MPU violation
Secure privileged	0xE002ED90 - 0xE002EDC4	0xE000ED90 - 0xE000EDC4	Secure MPU violation

Exceptions and Interrupts

Type of Exceptions

The following table describes the types of exceptions in the TrustZone implemented system.

Section	Guidance	Type	Default State
1 (-)	Reset	Secure only	Secure
2 (-14)	NMI	Configurable	Secure
3 (-13)	HardFault	Configurable	Secure
4 (-12)	MemManage Fault	Banked	Banked
5 (-11)	BusFault	Configurable	Secure
6 (-10)	UsageFault	Banked	Banked
7 (-9)	SecureFault	Secure only	Secure

Copyright © 2025 Silicon Laboratories. All rights reserved.

Section	Guidance	Type	Default State
11 (-5)	SVCall	Banked	Banked
12 (-4)	DebugMonitor	Configurable	Secure
14 (-2)	PendSV	Banked	Banked
15 (-1)	SysTick	Banked	Banked
16 - 495 (0 - 479)	IRQ0 - IRQ479	Configurable	Secure

Notes:

- "Secure only" means the system exceptions can only trigger in the Secure state.
- "Configurable" means the system exceptions and interrupts can be configured to target either the Secure state or the Non-secure state.
- Banked means the system exceptions can have Secure and Non-secure versions. Both can be triggered and executed independently and have different priority level settings.

Exception Priorities

It may cause a security issue if the Non-secure software uses high priority levels to mask the Secure interrupts. To avoid this issue, TrustZone introduces a programmable bit in the `AIRCR` register called `PRIS` (Prioritize Secure exception) for Secure software to prioritize, if needed, Secure exceptions and interrupts.

The `AIRCR.PRIS` is set to 0 out of reset, which means Secure and Non-secure exceptions/interrupts share the same configurable programmable priority level space (columns 2 and 3 in the following table). When the `AIRCR.PRIS` is set to 1, all Non-secure configurable exceptions/interrupts are placed in the lower half of the priority level space so that Secure exceptions/interrupts can potentially have higher priorities (columns 2 and 4 in the following table).

Priority Value	Secure Priority	Non-secure Priority (PRIS = 0)	Non-secure Priority (PRIS = 1)
0	0	0 (0x00)	128 (0x80)
1	32	32 (0x20)	144 (0x90)
2	64	64 (0x40)	160 (0xA0)
3	96	96 (0x60)	176 (0xB0)
4	128	128 (0x80)	192 (0xC0)
5	160	160 (0xA0)	208 (0xD0)
6	192	192 (0xC0)	224 (0xE0)
7	224	224 (0xE0)	240 (0xF0)

> Note: This table uses three bits (Bit [7:5]) of the group priority level (`AIRCR.PRIGROUP`) to limit the maximum number of preemption levels to 8. A lower priority value indicates a higher priority.

Vector Tables

The following figure shows two vector tables for Secure and Non-secure exceptions and interrupts. The vector table offset is defined by a Vector Table Offset Register (`VTOR` at `0xE000ED08`), which can only be programmed in the privileged state.

Copyright © 2025 Silicon Laboratories. All rights reserved.

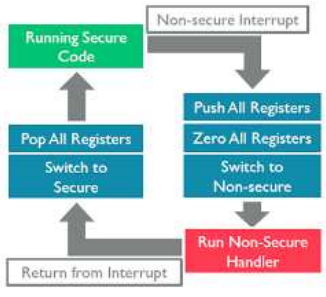
Exception number	IRQ number	Secure Vector	Non-secure Vector	Offset
495	479	IRQ479	IRQ479	0x7BC
. . .			. . .	. . .
18	2	IRQ2	IRQ2	0x48
17	1	IRQ1	IRQ1	0x44
16	0	IRQ0	IRQ0	0x40
15	-1	SysTick _S	SysTick_NS	0x3C
14	-2	PendSV _S	PendSV_NS	0x38
13		Reserved	Reserved	0x30
12	-3	DebugMonitor	DebugMonitor	
11	-5	SVCall _S	SVCall_NS	0x2C
10 9 8		Reserved	Reserved	
7	-9	SecureFault		0x1C
6	-10	UsageFault_S	UsageFault_NS	0x18
5	-11	BusFault_S	BusFault_NS	0x14
4	-12	MemManage_S	MemManage_NS	0x10
3	-13	HardFault _S	HardFault_NS	0x0C
2	-14	NMI _S	NMI_NS	0x08
1		Reset		0x04
		Initial SP value		0x00

Image: *Vector Table.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

Notes:

- The `VTOR_S` defines the address of the Secure vector table in Secure memory, and the Secure Main Stack Pointer (`MSP_S`) is the default stack for the Secure exception handler.
- The `VTOR_NS` defines the address of the Non-secure vector table in Non-secure memory, and the Non-secure Main Stack Pointer (`MSP_NS`) is the default stack for the Non-secure exception handler.
- Secure privileged software can access the `VTOR_NS` using the Non-secure SCB alias (`0xE002ED08`).
- The System Control Space contains registers for the SysTick timer, NVIC, and SCB.
- The interrupt masking registers (PRIMASK, FAULTMASK, and BASEPRI) are banked between security states. The priority level space is shared between the Secure and the Non-secure world, setting an interrupt mask register on one side can block some, or all, of the exceptions on the other side.
- Interrupts (`IRQ0` - `IRQ479`) are defined as Secure by default. Each interrupt can be configured as Secure or Non-secure and is determined by the Interrupt Target Non-secure (`NVIC_ITNS`) register, which is only programmable in the Secure software.

State Transitions in Exceptions and Interrupts



The following figure shows transitions between the processor states in ARMv8-M TrustZone.

Image (left): *Switching-between-Secure-and-Non-secure-states.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

1. Secure Thread → Secure Handler or Non-secure Thread to Non-secure Handler
   - No security state transition.
   - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.

Copyright © 2025 Silicon Laboratories. All rights reserved.

- The Interrupt Service Routine (ISR) is executed in the current security state (either Secure or Non-secure).

2. Non-secure Thread → Secure Handler or Non-secure Handler → Secure Handler
   - The transition from Non-secure to Secure state.
   - The exception sequence is almost identical to the exception stacking mechanism of current Cortex-M processors.
   - The ISR is executed in a Secure state.
3. Secure Thread → Non-secure Handler or Secure Handler → Non-secure Handler
   - The transition from Secure to Non-secure state.
   - To avoid an information leak when transitioning from the Secure to Non-secure state. The processor automatically pushes all general-purpose registers into the Secure stack and erases the contents of all general-purpose registers before executing the Non-secure ISR. The processor pops the contents of all general-purpose registers from the Secure stack when returning from the Non-secure ISR (right side in Figure 2.6 State Transitions on page 12). It incurs a slightly longer interrupt latency.
   - The ISR is executed in a Non-secure state.
4. Secure Privileged Thread ↔ Non-secure Privileged Thread or Secure Unprivileged Thread ↔ Non-secure Unprivileged Thread
   - The transition from Secure to Non-secure state or Non-secure to Secure state.
   - The Function calls and returns can be used when the privileged level remains the same.

> Note: Subject to interrupt priority, there are no restrictions regarding whether a Non-secure or Secure interrupt can occur when the processor runs Non-secure or Secure code.

Switching Between Secure and Non-secure States

The TrustZone allows direct calling between Secure and Non-secure software. The following figure shows how to use an API function call to trigger security state transitions. The state transitions can also happen because of exceptions and interrupts.

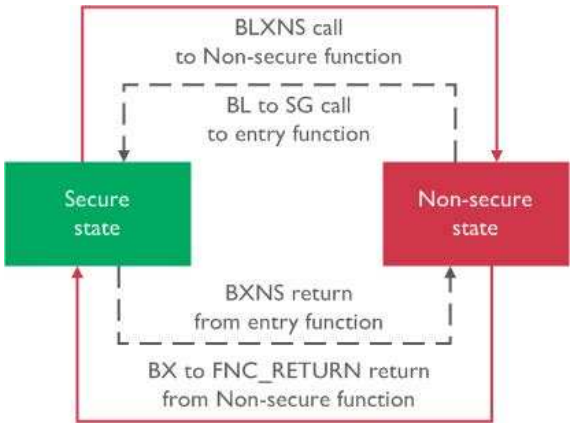


Image: *Switching-between-Secure-and-Non-secure-states.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

Switching from Non-secure to Secure State

When the Non-secure program calls a Secure software, the first instruction must be a Secure Gateway (`SG`) instruction residing in Non-secure Callable memory. The Secure Gateway entry points (veneers) decouple the address of the `SG` instructions in the Non-secure Callable memory region from the rest of the Secure code. It can eliminate the risk of having inadvertent entry points when the Secure software contains a pattern that matches the opcode of the `SG` instruction.

Copyright © 2025 Silicon Laboratories. All rights reserved.

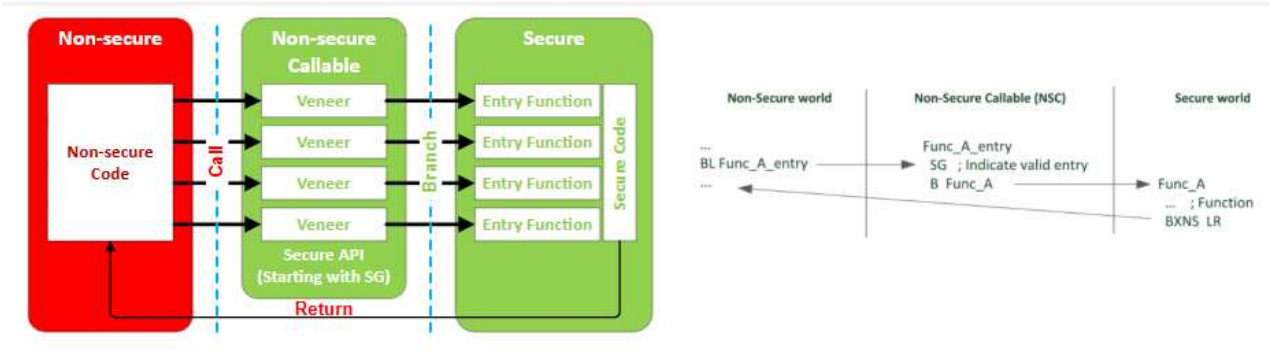


Image (right): *Whitepaper - ARMv8-M Architecture Technical Overview. Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

The bit 0 of the Link Register (LR) is cleared to zero by `SG` instruction to indicate that returning from this function transits from Secure to Non-secure. The processor is still in the Non-secure state when the `SG` instruction is executed. The `BXNS LR` instruction is used when returning since a normal `BX LR` instruction interprets it as an unsupported execution mode change. A SecureFault exception is triggered if the processor returns to a Secure address. It prevents a hacker from calling a Secure API with a fake return address pointing to a Secure program location. If bit 0 of LR is 1, the `BXNS LR` instruction behaves like a normal `BX LR`. Therefore, Secure code can call a Secure API in the NSC region even it is not a usual practice.

Program	Call Instruction	SG Instruction	Return Instruction
Non-secure call Non-secure	BL or BLX	-	BX LR (Return to Non-secure state)
Non-secure call Secure	BL or BLX	Clear bit 0 of LR	BXNS LR (Return to Non-secure state)
Secure call Secure	BL or BLX	Set bit 0 of LR	BXNS LR (Return to Secure state)

To help software developers create Secure APIs in C/C++, the Cortex-M Security Extension (CMSE) defines a C function attribute called `cmse_nonsecure_entry`.

- GCC — `__attribute__((cmse_nonsecure_entry))`
- IAR — `__cmse_nonsecure_entry`

Test Target (TT) Instruction

The software can use an ARMv8-M instruction called Test Target (TT) and the region number generated by the SAU or the IDAU to determine if a contiguous range of memory shares common security attributes and privilege levels.

The TT instruction returns the SAU/IDAU region number, security attributes (S/NS), and MPU region number after passing the start and end addresses of the memory range to the TT instruction. The software can determine whether the memory range has required security attributes and resides in the same region number.

Copyright © 2025 Silicon Laboratories. All rights reserved.

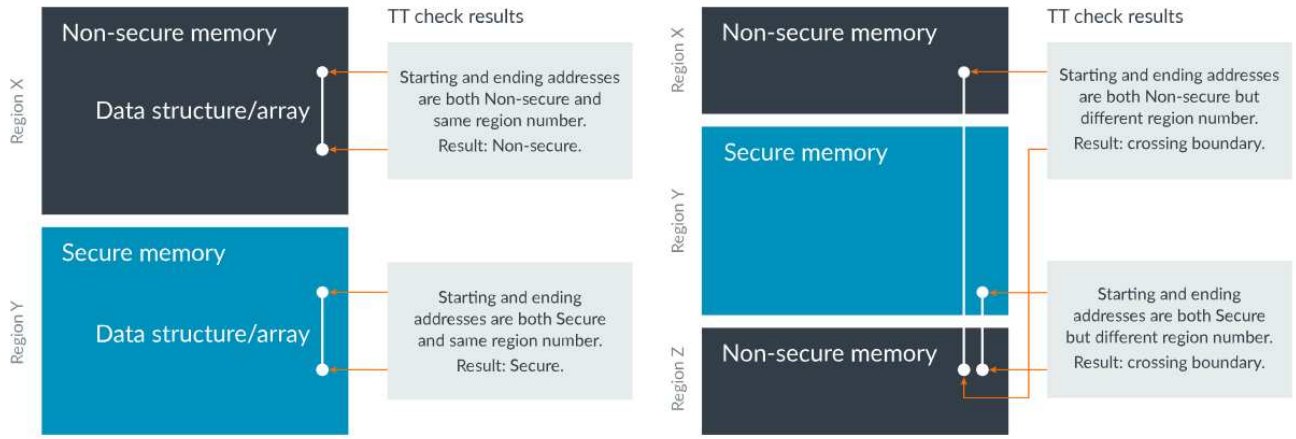


Image: *Test-target-instruction.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

This mechanism allows security checking at the beginning of the API service (instead of during the operation) to determine if the memory referenced by a pointer from Non-secure software points to the Non-secure address. It prevents Non-secure software from using those APIs in Secure software to access or modify Secure data.

To make these operations easier in a C/C++ programming environment, the Cortex-M Security Extension (CMSE) has defined a range of intrinsic functions for dealing with pointer checks with the TT instructions.

Switching from Secure to Non-secure State

When the Secure program calls a Non-secure software, the Secure program must use a `BLXNS <reg>` instruction to invoke the process. If bit 0 of the `<reg>` is 0, the processor must switch to the Non-secure state when branching to the target address. During the state transition, the return address and some processor state information are pushed onto the Secure stack, while the return address on the Link Register (LR) is set to a special value called `FNC_RETURN` (`0xFEFFFFFF`).

The Non-secure function completes by performing a branch (`BX LR`) to the `FNC_RETURN` address (bit 0 is 1 to indicate the function was called from the Secure state). It automatically triggers the unstacking of the actual return address from the Secure stack and returns to the calling function. The `FNC_RETURN` hides the return address of the Secure program from the Non-secure software to avoid the leakage of any secret information. It also prevents Non-secure software from modifying the Secure return address stored in the Secure stack.

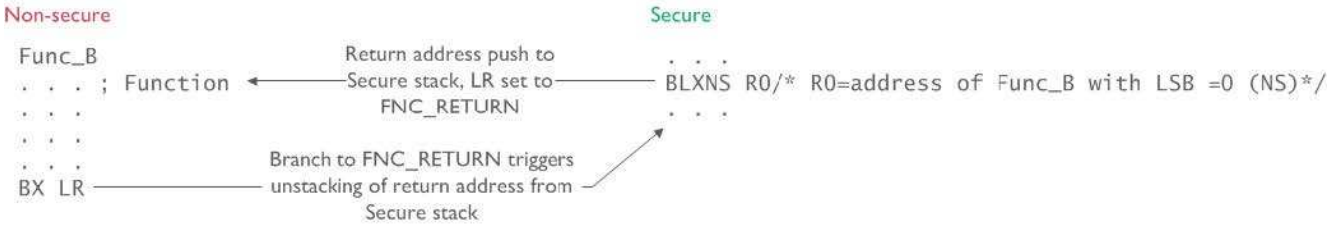


Image: *Switching-between-Secure-and-Non-secure-states.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

To help software developers declare Non-secure function pointers in C/C++, the Cortex-M Security Extension (CMSE) defines a C function attribute called `cmse_nonsecure_call` .

- GCC: `__attribute__((cmse_nonsecure_call))`
- IAR: `__cmse_nonsecure_call`

Copyright © 2025 Silicon Laboratories. All rights reserved.

Software Flow

The following figure describes a software flow example in a TrustZone implemented system.

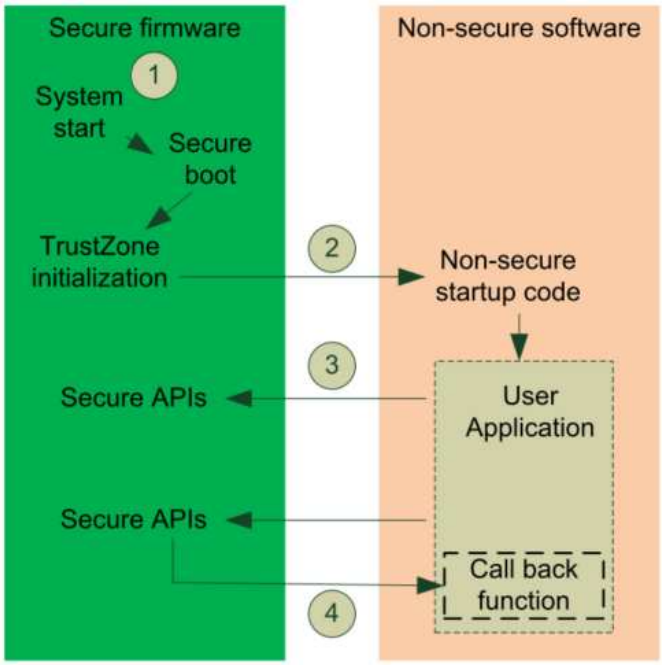


Image: *Software Development in ARMv8-M Architecture.Copyright © 1995-2022 Arm Limited (or its affiliates). All rights reserved.*

1. The system starts executing code in the Secure state after a power-on or reset (Secure boot).
   - The Secure stack pointer (`MSP_S`) is set from the address of the Secure vector table (`VTOR_S`).
   - The Secure Reset Handler pointed by the `VTOR_S` is called.
   - Perform various initialization tasks such as C startup code.
   - Place peripherals and associated interrupts in either Secure or Non-secure applications.
   - Program SAU/IDAU to partition the entire memory into Secure, Non-secure Callable, and Non-secure regions.
   - Program the address of the Non-secure vector table (`VTOR_NS`).
   - Initialize the two first entries of the table for the Non-secure stack pointer (`MSP_NS`) and Reset Handler to emulate a Non-secure reset.
2. The Secure firmware branches to the entry point (Reset Handler pointed by the `VTOR_NS`) of the Non-secure application.
   - The Non-secure software has its Reset Handler.
   - Perform various initialization tasks such as C startup code and hardware initialization (e.g., Non-secure peripherals).
   - It does not conflict with initialization from the Secure code as the stack and heap spaces of Secure and Non-secure code are separated.
3. During the execution of Non-secure applications, the application could call Secure APIs through the Secure Gateway (SG) veneer in the Non-secure Callable region.
4. In some cases, Secure APIs might need to call Non-secure call-back functions (e.g., a hardware driver).

Copyright © 2025 Silicon Laboratories. All rights reserved.

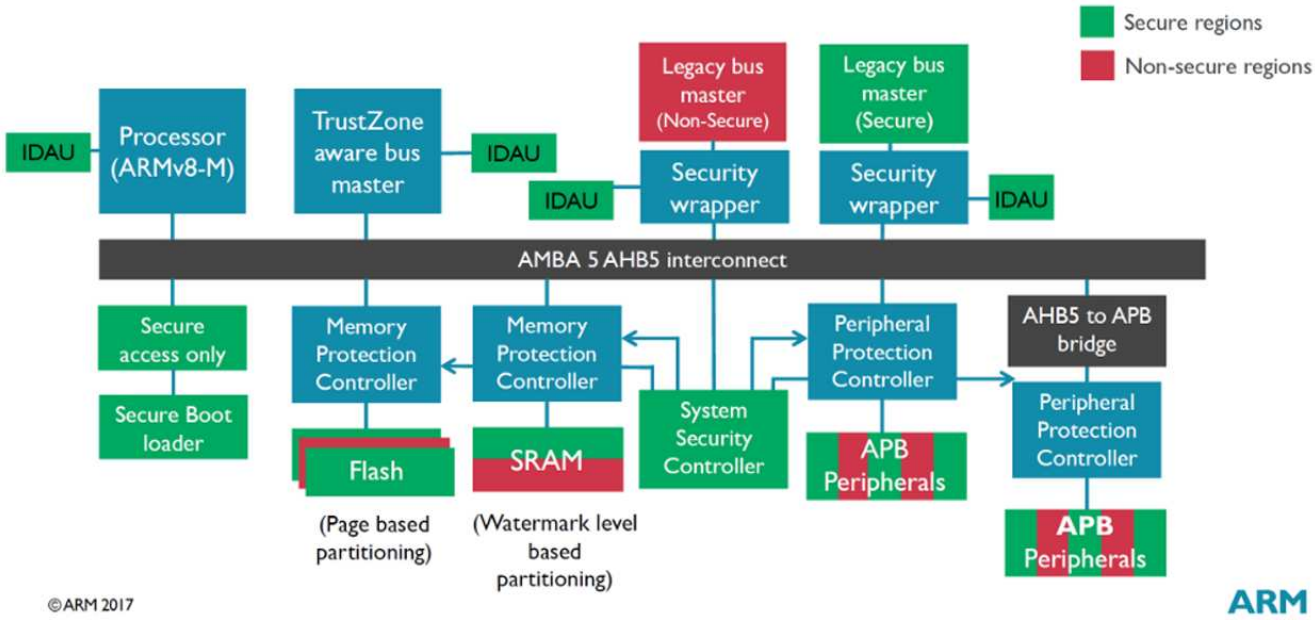
Bus Level Security (BLS)

Bus Level Security (BLS)

System Design

The following figure shows two system designs:

- The sample system contains an ARMv8-M processor and the required components to support TrustZone.
- Bus Level Security (BLS) on Series 2 devices implements the concepts introduced in the ARM TrustZone sample system. BLS enforces Secure and privileged programming models and uses security components (colored blocks) to configure the security attribute and privileged level of peripherals and Bus Masters.



ARMv8-M Processor

The ARMv8M processor is TrustZone capable of Secure and Non-secure states. It has a dedicated internal SAU that is fully programmable up to 8 different memory regions. Out of reset, the processor is in a Secure state and every transaction is a Secure transaction.

ARMv8-M Processor in Series 2 devices is the Cortex-M33.

System Security Controller

The system security controller is the central location for all security settings in the system. Each type of controller, IDAU, and wrapper receives its security configuration and bus response configuration from this block.

System Security Controller in Series 2 devices is the Security Management Unit (SMU).

Implementation Defined Attribution Unit (IDAU)

Copyright © 2025 Silicon Laboratories. All rights reserved.

The IDAU generates the security attribute for a given address. All IDAUs in the system have the same memory partitioning. The IDAU is intended only for ARMv8-M cores and utilizes the entire IDAU interface for the core. The lite IDAU uses only the Secure and Non-secure interface from the IDAU and is intended for Non-ARMv8-M Bus Masters.

IDAU in Series 2 devices is the External Secure Attribution Unit (ESAU).

Security Wrapper

The Security Wrapper gives a legacy Bus Master the ability to drive security attribution. The security wrapper outputs the transaction address to the lite IDAU which returns the security attribute of the address. If the wrapper is configured as Non-secure, any transactions to a Secure address are blocked.

Security Wrapper in Series 2 devices is the Bus Master Protect Unit (BMPU).

Memory Protection Controller (MPC)

MPC has a security configuration for a per block of memory or memory above and below the watermark. If the security attribute of the block or memory region does not match the security attribute of the address, the transaction is blocked. This controller is used in a system that alias RAM or flash memory locations. This controller is not needed when the memory region size is programmable in an IDAU.

Series 2 devices have a programmable flash and RAM region in the ESAU (equivalent to IDAU) and are not implementing this block.

Peripheral Protection Controller (PPC)

PPC has a security configuration for every peripheral. If the security attribute of the selected peripheral does not match the security attribute of the address, the transaction is blocked. This controller is used in systems that alias the peripheral memory locations.

PPC in Series 2 devices is the Peripheral Protection Unit (PPU).

Hardware security is now extended to the peripheral bus system of the processor. Each component on the bus can verify and propagate the security level for each bus operation. The following sections describe the individual security component for BLS on Series 2 devices.

Security Management Unit (SMU)

The SMU is the only user-facing block in the BLS architecture and houses all the configuration and status for the ESAUs, BMPUs, and PPUs.

- Thirteen memory regions (ESAU)
- Per Bus Master privileged and security attribute (BMPU)
- Interrupt flag for Bus Master security fault (fault table in BMPU section)
- Per peripheral privileged and security attribute (PPU)
- Interrupt flags for privileged, security, and instruction peripheral access faults (fault tables in PPU section)
- Separate Secure and Privileged IRQ

The SMU configurations can be locked down and protected from runaway code. The `SMU_LOCK` register resets to UNLOCK. Any write other than the unlock code (`0xACCE55`) locks all SMU registers from further updates. The `SMU_STATUS` register contains a `SMULOCK` bitfield with the current lock state of the SMU.

The `SMU_M33CTRL` register can lock down internal security and privileged configurations below.

- Cortex-M33 SAU
- Non-secure MPU
- Secure MPU
- Non-secure Vector Table Offset Register (VTOR)
- Secure AIRCR register

Copyright © 2025 Silicon Laboratories. All rights reserved.

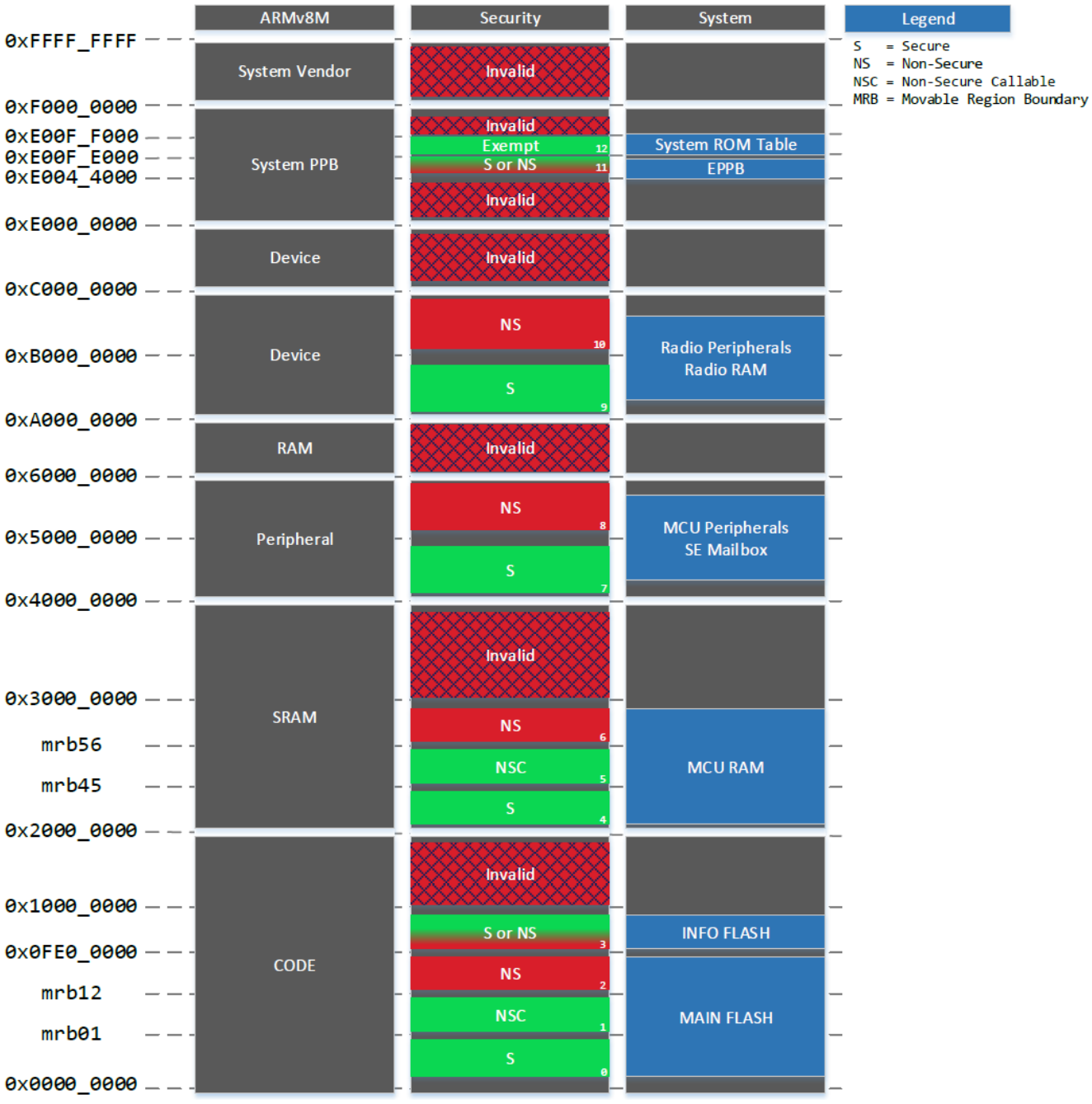
Interrupt flags in the `SMU_IF` register can generate a Secure or Privileged interrupt in the table below when its corresponding interrupt enable bit in the `SMU_IEN` register is set and `IRQn` is enabled.

Enable Bit in SMU_IEN Register	IRQn	Interrupt Handler
BMPUSEC, PPUSEC	SMU_SECURE_IRQn	SMU_SECURE_IRQHandler()
PPUPRIV, PPUINST	SMU_PRIVILEGED_IRQn	SMU_PRIVILEGED_IRQHandler()

Each interrupt flag in the `SMU_IF` register can be cleared by writing 1 to the corresponding bit of the `SMU_IF_CLR` register.

External Secure Attribution Unit (ESAU)

The ESAU is responsible for determining the memory region and security attribute of a given address. Referring to ARMv8-M TrustZone Implementation on page 16, the Cortex-M33 interfaces with an ESAU and the BMPUs of other Bus Masters interface with lite ESAUs to determine the security attribute of all transactions. The following figure describes the security attributes of different memory regions defined by the ESAU on Series 2 devices.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Notes:

- For Series 2 devices with base address `0x08000000` in region 0, the memory address from `0x0` to `0x07FFFFFF` is an invalid region.
- The invalid regions are deemed as Secure.
- The NSC and Exempted attributes are only available to the ESAU, and all lite ESAUs in the system view these attributes as Secure.

The ESAU divides the memory map into 13 memory regions and has a maximum of 6 Non-secure regions.

- Four Movable Region Boundaries (MRBs) determine the size of 6 regions.
- Two regions have configurable security attributes.
- Each memory region consists of a base address that specifies the start of the region and a limit address that specifies the end of the region plus one (+ 1).
- The address is valid if it falls between the base (≥ base) and limit (< limit) of a region.
- If the memory region is not defined, it is deemed invalid and Secure.

The MRBs distinguish the Secure, Non-secure Callable, and Non-secure regions in flash and RAM. The two configurable regions determine if the Info flash and Cortex-M33 EPPB regions are Secure or Non-secure. The MRBs have a specific programming sequence. Any misprogramming results in a `SMUPRGERR` in the `SMU_STATUS` register.

ARMv8-M CODE Regions

- Regions 0, 1, and 2 are in the Main space of flash. Region 3 is the info space of flash.
- The mrb01 (`ESAUMRB01` in `SMU_ESAURMBR01` register) determines the end of region 0 and the start of region 1.
- The mrb12 (`ESAUMRB12` in `SMU_ESAURMBR12` register) determines the end of region 1 and the start of region 2.
- The size of region 3 is device-dependent.
- Three regions' security attributes are static, and one region is configurable. Region 0 is always Secure, region 1 is always Non-secure Callable, and region 2 is always Non-secure. Region 3 is configurable as either Secure or Non-secure (`ESAUR3NS` in `SMU_ESAURTYPES0` register, default is secure after reset).
- Sizes of regions 0, 1, and 2 are adjusted in 4 kB increments with the lower 12 bits of `ESAUMRB##` in `SMU_ESAURMBR##` ignored.
  - The Secure region can be set to size 0 when mbr01 = base address of region 0.
  - The Non-secure Callable regions can be set to size 0 when mbr01 = mbr12.
- The default value of mbr01 is equal to base address + `0x02000000` , so the size of region 0 is 32 MB. Out of reset, all flash is Secure since all Series 2 devices have less than 32 MB of flash.

Region	Memory	Base Address	Limit Address	Security Attribute
0	Main flash	0x00000000 or 0x08000000	(0x00000000 or 0x08000000) | mbr01	Secure
1	Main flash	(0x00000000 or 0x08000000) | mbr01	(0x00000000 or 0x08000000) | mbr12	Non-secure Callable
2	Main flash	(0x00000000 or 0x08000000) | mbr12	0x0FE00000	Non-secure
3	Info flash	0x0FE0000	0x10000000	Secure or Non-secure

ARMv8-M RAM Regions

- Regions 4, 5, and 6 cover the entire available RAM in the device.
- The mrb45 (`ESAUMRB45` in `SMU_ESAURMBR45` register) determines the end of region 4 and the start of region 5.
- The mrb56 (`ESAUMRB56` in `SMU_ESAURMBR56` register) determines the end of region 5 and the start of region 6.
- All three regions' security attributes are static. Region 4 is always Secure, region 5 is always Non-secure Callable, and region 6 is always Non-secure.
- Sizes of all three regions are adjusted in 4 kB increments with the lower 12 bits of `ESAUMRB##` in `SMU_ESAURMBR##` ignored.
  - The Secure region can be set to size 0 when mbr45 = base address of region 4.
  - The Non-secure Callable region can be set to size 0 when mbr45 = mbr56.
- The default value of mbr45 is equal to `0x02000000` , so the size of region 4 is 32 MB. Out of reset, all RAM is Secure since all Series 2 devices have less than 32 MB of RAM.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Region	Memory	Base Address	Limit Address	Security Attribute
4	SRAM	0x20000000	0x20000000 | mbr45	Secure
5	SRAM	0x20000000 | mbr45	0x20000000 | mbr56	Non-secure Callable
6	SRAM	0x20000000 | mbr56	0x30000000	Non-secure

ARMv8-M Peripheral Regions

- These regions are aliases to the chip peripherals and SE mailbox (a device with HSE).
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 7 is always Secure, and region 8 is always Non-secure.

Region	Memory	Base Address	Limit Address	Security Attribute
7	Chip Peripherals	0x40000000	0x50000000	Secure
8	Chip Peripherals	0x50000000	0x60000000	Non-secure

ARMv8-M Device Regions

- These regions are aliases to all radio peripherals and radio RAM.
- Both regions have a fixed size.
- Both regions' security attributes are static. Region 9 is always Secure, and region 10 is always Non-secure.
- From the perspective of the device bus system, the radio is one peripheral that is either Secure or Non-secure. So any Bus Master accessing the radio needs to know the security attribute of the radio. From the perspective of the radio, all of its radio bus peripherals are accessible regardless of the security attribute. However, the radio needs to know the security attribute of chip bus peripherals to access them through the correct alias.

Region	Memory	Base Address	Limit Address	Security Attribute
9	Radio Peripherals	0xA0000000	0xB0000000	Secure
10	Radio Peripherals	0xB0000000	0xC0000000	Non-secure

ARMv8-M System Private Peripheral Bus (PPB) Regions

- Both regions have a fixed size.
- Region 11 is the Cortex-M33 EPPB memory region and is configurable as either Secure or Non-secure (`ESAUR11NS` in `SMU_ESAURTYPES1` register, default is secure after reset). It is important to note that the Cortex-M33 core is the only Bus Master that sees these memory regions. All other Bus Masters in the system do not have access to the System PPB, and it is an invalid region.
- Region 12 has a static security attribute of Exempted. It means that the Cortex-M33 core allows the transaction in all cases. It permits debuggers to read the system ROM Table regardless of the state of the Cortex-M33 core.

Region	Memory	Base Address	Limit Address	Security Attribute
11	EPPB	0xE0044000	0xE00FE000	Secure or Non-secure
12	System ROM Table	0xE00FE000	0xE00FF000	Exempted

Notes:

- The regions in flash (0/1/2) and RAM (4/5/6) can only create in the order of Secure, Non-secure Callable, and Non-secure.
- The ESAU and lite ESAUs handle the transactions of Bus Masters and must have consistent security attribute mapping. Therefore, configurations in the SMU registers apply to ESAU and lite ESAUs.
- Unlike other Bus Masters using BMPU and lite ESAU, merging the address lookup results from the internal SAU and ESAU determines the security attribute of the Cortex-M33 transaction.

Bus Master	Security Attribution
Cortex-M33	SAU and ESAU
Other	Lite ESAU

Copyright © 2025 Silicon Laboratories. All rights reserved.

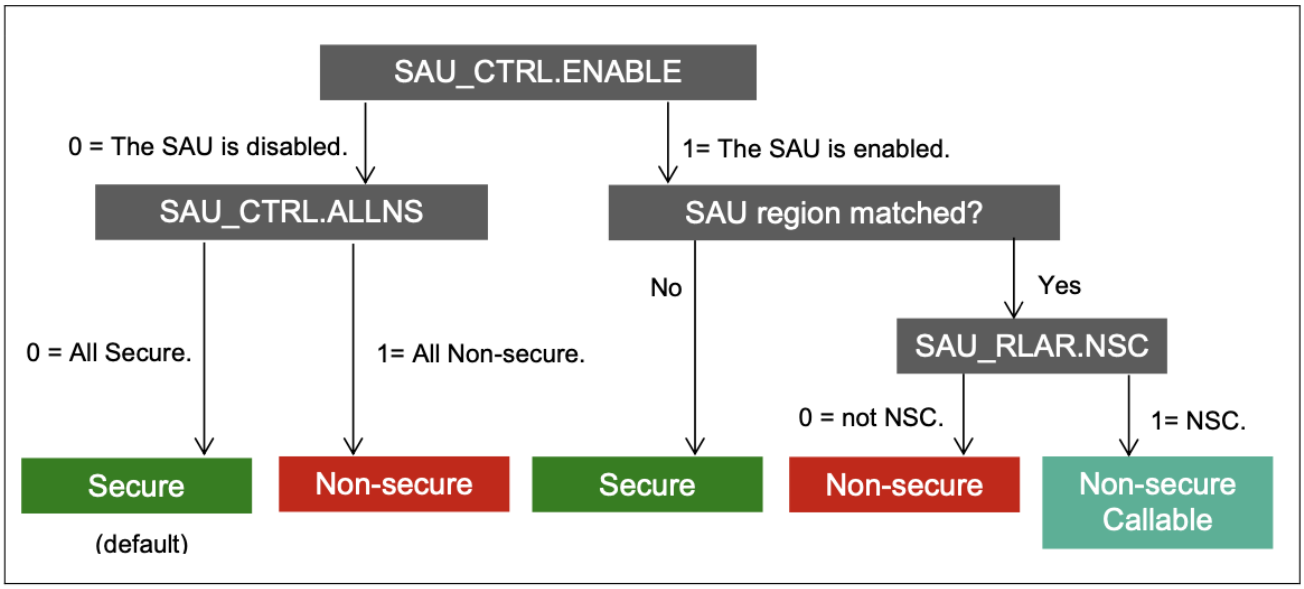
Security Attribution Unit

In Series 2 devices, the combination of the integrated SAU in the Cortex-M33 processor and an ESAU determine the security attribute of a Cortex-M33 transaction.

The SAU consists of several programmable registers. These registers are placed in the System Control Space (SCS) and are only accessible from the Secure privileged state.

- SAU Control Register (`SAU_CTRL`) — The SAU is disabled after RESET
- SAU Type Register (`SAU_TYPE`) — Indicates the number of available regions (read-only)
- SAU Region Number Register (`SMU_RNR`) — Assigns a region number
- SAU Region Base Address Register (`SAU->RBAR`) — Configures selected region base address
- SAU Region Limit Address Register (`SAU->RLAR`) — Configures selected region limit address and security attribute (NSC or NS), enable or disable the region

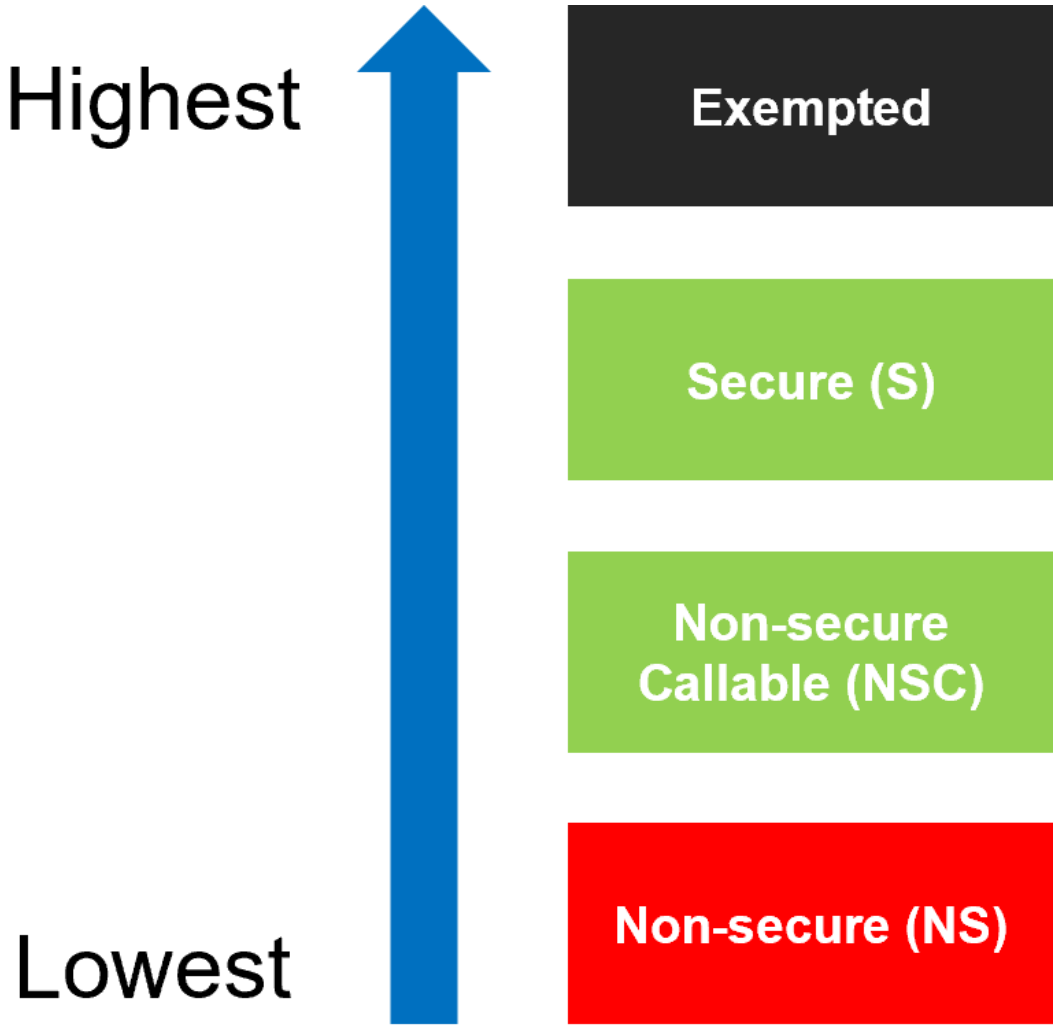
The following figure shows three different SAU configurations for determining the security attribute of a Cortex-M33 transaction.



Notes:

- All address ranges after RESET in SAU are Secure by default.
- The SAU can configure a 32 bytes aligned region as Non-secure or Non-secure Callable. Any address not defined in the SAU defaults to Secure.
- An ESAU can configure or hard-code a region as Secure, Non-secure Callable, Non-secure, or Exempted. An Exempted region enables Non-secure debuggers to access debugging components and establish a debug connection to the processor before the SAU is configured.
- The processor determines the final attribute of the address based on the higher security attribute (Exempted > S > NSC > NS) from either the SAU or the ESAU.

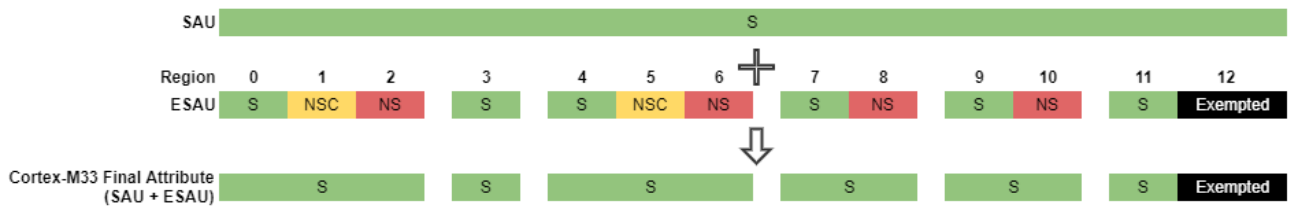
Copyright © 2025 Silicon Laboratories. All rights reserved.



All Secure Configuration

Highlights:

- SAU is disabled.
- `ALLNS` bit in the SAU Control register is clear.
- The whole memory is in a Secure state (highest security attribute apart from Exempted).
- All Cortex-M33 transactions in this configuration are Secure or Exempted and give the Cortex-M33 access to all memory locations through either the Secure or Non-secure alias after RESET.



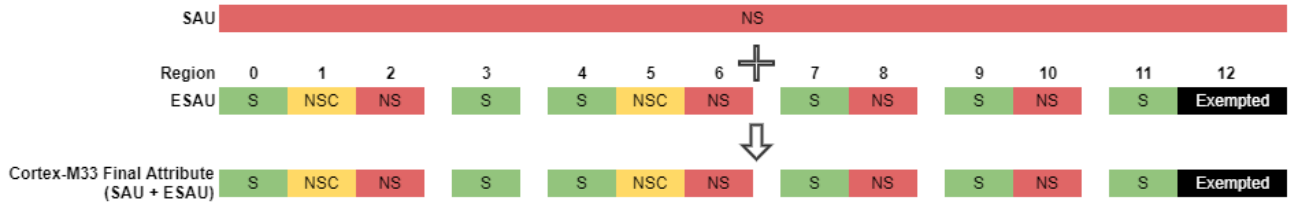
- It is up to the boot procedure in a Secure state to keep the current configuration or use other configurations once the boot process is complete.

Copyright © 2025 Silicon Laboratories. All rights reserved.

All Non-secure Configuration

Highlights:

- SAU is disabled.
- `ALLNS` bit in the SAU Control register is set.
- The whole memory is in a Non-secure state (lowest security attribute).
- Therefore the ESAU configuration determines the security attribute of all Cortex-M33 transactions.

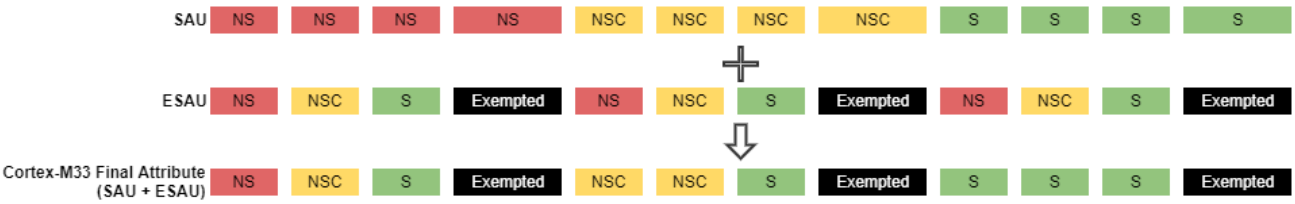


- Except for the `SAU_CTRL` register, this configuration does not require programming on other SAU registers.

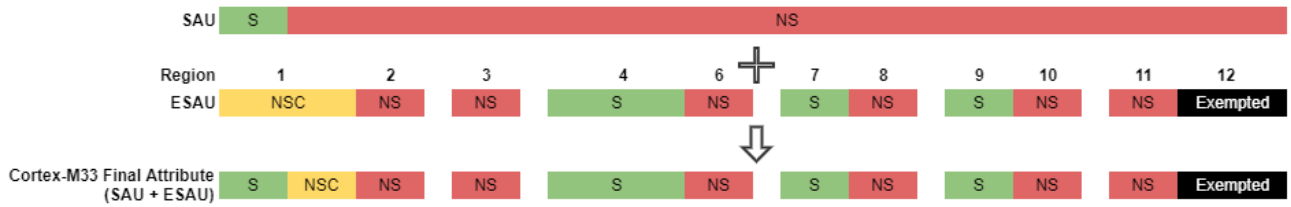
Configurable Configuration

Highlights:

- SAU is enabled.
- `ALLNS` bit in the SAU Control register can be 0 or 1 (do not care).
- The `NSC` bit on the `SAU_RLAR` register determines the security attribute of an address as Non-secure or Non-secure Callable if an address matches an SAU region.
- The security attribute of an address is Secure by default if the address does not match any SAU region.
- This configuration programs `SAU_RNR`, `SAU_RBAR`, and `SAU_RLAR` registers to correlate the Non-secure regions in ESAU.
- The SAU or ESAU overrides the attribute to a higher security level if any security attribute mismatch occurs in a memory region.



- The following figure is an example of a configurable configuration with the size of ESAU regions 0 and 5 are set to zero.



Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note: The Cortex-M33 has an internal SAU that defaults all undefined addresses to Secure if enabled. If the Secure regions do not align between the Cortex-M33 (SAU + ESAU) and other Bus Masters (lite ESAU), the Cortex-M33 treats a memory region as Secure while other Bus Masters treat it as Non-secure. It can lead to the leaking of secure data if the Cortex-M33 stores secure data in what other Bus Masters think is a Non-secure area (Main Flash Layout on page 34).

Bus Master Protection Unit (BMPU)

The BMPU is a security wrapper used for assigning a Bus Master specific security and privileged states. Referring to Figure 3.1 ARMv8-M TrustZone Implementation on page 16, the BMPU generally lies between the Bus Master and the Advanced High-performance Bus (AHB) Matrix. BMPU interfaces with a lite ESAU to determine the security attribute of all Bus Master transactions.

The registers below in SMU configure the security and privileged state of a Bus Master. The Bus Masters in group 0 are device-dependent. Out of reset, each Bus Master is Secure and privileged.

Register	Description
SMU_BMPUPATD0	Bitfields (privileged if set) for privileged attribute configuration on Bus Master group 0
SMU_BMPUSATD0	Bitfields (Secure if set) for security attribute configuration on Bus Master group 0

> Note: The Bus Master privileged attribute only applies to peripheral accesses. Flash and RAM accesses ignore the privileged attribute of the Bus Master.

The BMPU generates a security fault when the security attribute of the bus transaction is Secure, and the security attribute (`SMU_BMPUSATD0`) for the BMPU is configured as Non-secure.

Below is the security fault table that shows how the security attribute on the bus is driven based on the lite ESAU attribute and the BMPU security configuration. The interrupt is triggered if `BMPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Lite ESAU Attribute	Secure Bus Master	Non-secure Bus Master
Non-secure	Non-secure	Non-secure
Secure	Secure	FAULT

Upon a BMPU fault, the registers in SMU below notify that a BMPU security fault occurred and on which Bus Master. The registers also identify the offending fault address. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and the corresponding interrupt flag is set in the `SMU_IF` register. The values in `SMU_BMPUFS` and `SMU_BMPUFSADDR` do not change until the BMPU fault (`BMPUSEC`) in the `SMU_IF` register is cleared by software.

Register	Bitfield	Fault
SMU_IF	BMPUSEC	Security Fault if set
SMU_BMPUFS	BMPUFSMASTERID	ID of the Bus Master that triggered the fault
SMU_BMPUFSADDR	BMPUFSADDR	Access address that triggered the fault

> Note: No privileged fault is generated because all the other Bus Masters in the system do not drive the privileged attribute.

Peripheral Protection Unit (PPU)

Copyright © 2025 Silicon Laboratories. All rights reserved.

The PPU is a security wrapper used for assigning a Bus Slave peripheral specific security and privileged states. Referring to Figure 3.1 ARMv8-M TrustZone Implementation on page 16, the PPU comes in the form of a PPU in Advanced High-performance Bus (AHB) and a PPU in Advanced Peripheral Bus (APB).

- The PPU AHB generally lies between the Bus Matrix and an AHB Bus Slave peripheral.
- The PPU APB lies between the output of an AHB to APB bridge and all of the APB Slaves on that APB bus.

The registers below in SMU configure the security and privileged state of a peripheral. The peripherals in groups 0 and 1 are device-dependent. Out of reset, each peripheral is Secure and privileged. While each peripheral in address `0x40000000` (region 7) or `0x50000000` (region 8) can be configured independently, the radio subsystem in `0xA0000000` (region 9) or `0xB0000000` (region 10) is configured as a unit.

Register	Description
SMU_PPUPATD0	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUPATD1	Bitfields (privileged if set) for privileged access configuration on peripheral group 0
SMU_PPUSATD0	Bitfields (Secure if set) for security access configuration on peripheral group 0
SMU_PPUSATD1	Bitfields (Secure if set) for security access configuration on peripheral group 1

The PPU can generate three types of faults:

1. Privileged faults occur on unprivileged transactions to privileged peripherals. Below is the privileged fault table that shows when a privileged fault occurs based on the PPU peripheral privileged configuration and the bus transaction privileged attribute. The interrupt is triggered if `PPUPRIV` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	Privileged Peripheral	Unprivileged Peripheral
Privileged	SUCCESS	SUCCESS
Unprivileged	FAULT	SUCCESS

2. Security faults occur on Secure transactions to Non-secure peripherals and Non-secure transactions to Secure peripherals. Below is the security fault table that shows when a security fault occurs based on the PPU Peripheral security configuration and the bus transaction security attribute. The interrupt is triggered if `PPUSEC` in `SMU_IEN` is set and the `SMU_SECURE_IRQn` is enabled.

Bus Attribute	Secure Peripheral	Non-secure Peripheral
Secure	SUCCESS	FAULT
Non-secure	FAULT	SUCCESS

3. Instruction faults occur on any transaction marked as an instruction fetch. Below is the instruction fault table that shows when a PPU instruction fault occurs based on the bus transaction instruction attribute. The interrupt is triggered if `PPUINST` in `SMU_IEN` is set and the `SMU_PRIVILEGED_IRQn` is enabled.

Bus Attribute	Secure Peripheral	Non-secure Peripheral
Secure	SUCCESS	FAULT
Non-secure	FAULT	SUCCESS

Upon a PPU fault, the registers below in SMU notifies which PPU fault occurred and on which peripheral. If a fault is detected, the response is Read As Zero (RAZ) or Write Ignored (WI) and set the corresponding interrupt flag in the `SMU_IF` register. The values in `SMU_IF` and `SMU_PPUFS` do not change until all PPU faults in the `SMU_IF` register are cleared by software.

Register	Bitfield	Fault
SMU_IF	PPUPRIV	Privilege Fault if set
SMU_IF	PPUSEC	Security Fault if set
SMU_IF	PPUINST	Instruction Fault if set
SMU_PPUFS	PPUFSPERIPHID	ID of the peripheral that caused the fault

Compatibility

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure software usually controls the SYSCFG and SMU peripherals to prevent Non-secure software from changing critical configurations in the Secure domain. It requires switching between Secure and Non-secure states when Non-secure software wants to update the registers in these peripherals. Therefore dedicated registers for Non-secure access are added to SYSCFG and SMU peripherals on newer Series 2 devices.

System Configuration (SYSCFG)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table: Dedicated Bitfield to Configure Access for Non-secure SYSCFG Registers

Bitfield (Register)	Description
SYSCFGCFGNS (SMU_PPUPATD0)	Bitfields (privileged if set) for privileged access configuration on NS SYSCFG registers
SYSCFGCFGNS (SMU_PPUSATD0)	Bitfields (Secure if set) for security access configuration on NS SYSCFG registers

> Note: Reset `SYSCFGCFGNS` bit in `SMU_PPUSATD0` to allow Non-secure software to access NS SYSCFG registers.

Table: Dedicated SYSCFG Registers for Non-secure State

SYSCFG Non-secure Registers	Description
SYSCFG_CFGNS_CFGNSTCALIB	NS SysTick calibration value register
SYSCFG_CFGNS_ROOTNSDATA0	NS root data register 0
SYSCFG_CFGNS_ROOTNSDATA0	NS root data register 1

Security Management Unit (SMU)

Except for EFR32xG21 devices, the following tables apply to all Series 2 devices.

Table: Dedicated Bitfield to Configure Access for Non-secure SMU Registers

Bitfield (Register)	Description
SMUCFGNS (SMU_PPUPATD1)	Bitfields (privileged if set) for privileged access configuration on NS SMU registers
SMUCFGNS (SMU_PPUSATD1)	Bitfields (Secure if set) for security access configuration on NS registers

> Note: Reset `SMUCFGNS` bit in `SMU_PPUSATD1` to allow Non-secure software to access NS SMU registers.

The SMU_CFGNS register file is for the TrustZone Non-secure state and has its register lock (`NSLOCK`). It allows hardware to maintain the privileged assignments for the NS state. The privileged configuration within the NS state is the same as the Secure state, except it has an "NS" to differentiate the registers.

Table: Dedicated SMU Registers for Non-secure State

SMU Non-secure Registers	Description
SMU_CFGNS_NSSTATUS	Lock status of SMU_CFGNS registers
SMU_CFGNS_NSLCOK	Lock and unlock the SMU_CFGNS registers

Copyright © 2025 Silicon Laboratories. All rights reserved.

SMU Non-secure Registers	Description
SMU_CFGNS_NSIF	Interrupt flags for NS privilege (PPUNSPRIVIF) and instruction (PPUNSINSTIF) faults
SMU_CFGNS_NSIEN	Interrupt enable flags for NS privilege (PPUNSPRIVIEN) and instruction (PPUNSINSTIEN) faults
SMU_CFGNS_PPUNSPATD0	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 0
SMU_CFGNS_PPUNSPATD1	Bitfields (privileged if set) for NS privileged access configuration on peripheral group 1
SMU_CFGNS_PPUNSFS	ID (PPUFSPERIPHID) of the NS peripheral that caused the fault
SMU_CFGNS_BMPUNSPATD0	Bitfields (privileged if set) for privileged attribute configuration on NS Bus Master group 0

Table: Fault Statuses Only for Secure State

Bitfield (Register)	Description
PPUPRIV (SMU_IF)	Fault status now limited only to Secure state
PPUINST (SMU_IF)	Fault status now limited only to Secure state
PPUPRIV (SMU_IEN)	Fault status now limited only to Secure state
PPUINST (SMU_IEN)	Fault status now limited only to Secure state
PPUFSPERIPHID (SMU_PPUFS)	Fault status now limited only to Secure state

Table Dedicated SMU Interrupt for Non-secure State

Interrupt	Description
SMU_NS_PRIVILEGED_IRQHandler()	An interrupt flag in the SMU_CFGNS_NSIF register can generate an NS privileged interrupt when its corresponding interrupt enable bit in the SMU_CFGNS_NSIEN register is set and SMU_NS_PRIVILEGED_IRQn is enabled, and in which the peripheral (ID) that triggers the fault is in the SMU_CFGNS_PPUNSFS register.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure And Privileged Programming Model

Secure and Privileged Programming Model

The implementation of BLS on Series 2 devices, both flash and RAM, use a programmable watermark to delineate Secure, Non-secure Callable, and Non-secure regions. On the other hand, peripherals exist in both a Secure and Non-secure alias of memory.

BLS SMU Programming

Enabling SMU Clock

Except for the EFR32xG21 devices, all Series 2 devices enable the SMU clock in CMU before programming the SMU registers.

```
#if (_SILICON_LABS_32B_SERIES_2_CONFIG > 1)
  CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
#endif
```

Cortex-M33 Lock Control

The Cortex-M33 security and privileged configurations can be locked by programming the `SMU_M33CTRL` register.

```
// Lock Secure MPU configuration
SMU->M33CTRL |= SMU_M33CTRL_LOCKSMPU;
```

Locking SMU Configuration

Th entire SMU configuration can be locked down to avoid runaway code. Below is an example of how to lock and unlock the SMU.

```
uint32_t lock_status;
// Lock Down SMU
SMU->LOCK = ~SMU_LOCK_SMULOCKKEY_UNLOCK;
// Grab Lock Status
lock_status = (SMU->STATUS & _SMU_STATUS_SMULOCK_MASK) >> _SMU_STATUS_SMULOCK_SHIFT;
// Unlock SMU
SMU->LOCK = SMU_LOCK_SMULOCKKEY_UNLOCK;
```

Interrupt Control

Each interrupt flag in `SMU_IF` can generate an interrupt when its corresponding interrupt enable flag in the `SMU_IEN` register is set. Each interrupt flag can be cleared by writing the clear alias of the `SMU_IF` register.

```
// Clear and enable the SMU PPUSEC and BMPUSEC interrupt
NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
NVIC_EnableIRQ(SMU_SECURE_IRQn);
SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
```

BLS ESAU Programming

Copyright © 2025 Silicon Laboratories. All rights reserved.

Region Types

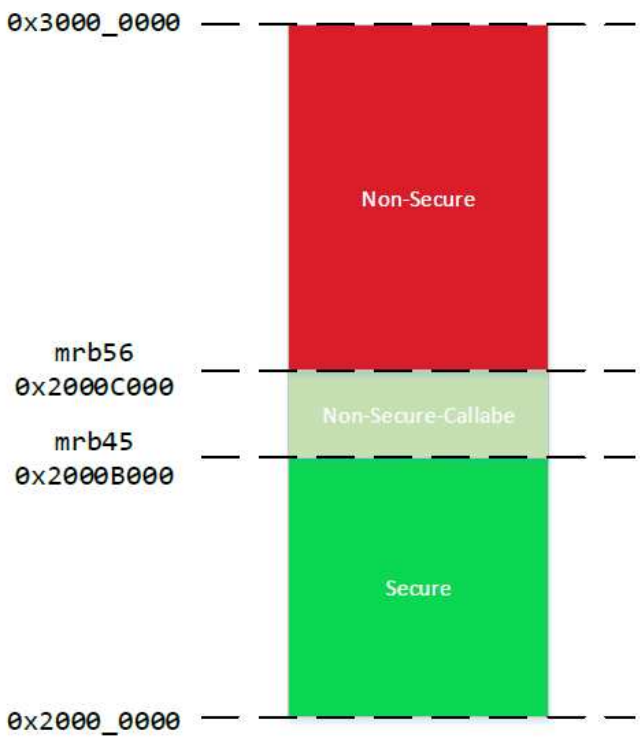
The `SMU_ESAURTYESn` registers are used to configure memory regions with a specific security attribute. All configurable memory regions reset to Secure. Below is an example of programming regions 3 and 11 to Non-secure.

```
// Region 3 (Info flash) is Non-secure
SMU->ESAURTYPES0 = SMU_ESAURTYPES0_ESAUR3NS;
// Region 11 (EPPB) is Non-secure
SMU->ESAURTYPES1 = SMU_ESAURTYPES1_ESAUR11NS;
```

Region Sizes

The code and figure below highlight how to program the Movable Region Boundaries (MRBs) of ESAU.

```
// ESAU region 0/1/2 programming
// Boundary01 at 252kB and  Boundary12 at 256kB
SMU->ESAUMRB01 = 0x0003F000U & _SMU_ESAUMRB01_MASK;
SMU->ESAUMRB12 = 0x00040000U & _SMU_ESAUMRB12_MASK;
// ESAU region 4/5/6 programming
// Boundary45 at 44kB and Boundary56 at 44kB (region 5 size = 0)
SMU->ESAUMRB45 = 0x0000B000U & _SMU_ESAUMRB45_MASK;
SMU->ESAUMRB56 = 0x0000B000U & _SMU_ESAUMRB56_MASK;
```



Notes:

- The mrb12 (`ESAUMRB12` in `SMU_ESAURMBR12`) has to be greater than or equal to mrb01 (`ESAUMRB12` in `SMU_ESAURMBR12`).
- The mrb56 (`ESAUMRB56` in `SMU_ESAURMBR562`) has to be greater than or equal to mrb45 (`ESAUMRB45` in `SMU_ESAURMBR45`).
- If one of the rules above is violated, the `SMU_STATUS.SMUPRGERR` is asserted.
- When mrb01 and mrb12 are equal, region 1 (NSC) is a size of 0 and is not seen by the system.
- When mrb45 and mrb56 are equal, region 5 (NSC) is a size of 0 and is not seen by the system.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BLS SAU Programming

All Secure Configuration

All secure configuration is the default state after reset. It clears the `SAU_CTRL.ENABLE` and the `SAU_CTRL.ALLNS` bits in SAU, and the entire memory is in a Secure attribute.

All Non-secure Configuration

All Non-secure Configuration occurs when the `SAU_CTRL.ENABLE` bit is cleared, and the `SAU_CTRL.ALLNS` bit is set. The ESAU controls the security attribute of a Cortex-M33 transaction.

```
// Disable SAU (ALLNS = 1) and clear data and instruction pipe
SAU->CTRL = SAU_CTRL_ALLNS_Msk;
__DSB();
__ISB();
```

Configurable Configuration

Configurable configuration occurs when the `SAU_CTRL.ENABLE` bit is set (`SAU_CTRL.ALLNS` is irrelevant). Both SAU and ESAU determine the security attribute of a Cortex-M33 transaction. The code and figure below highlight how to program the SAU regions.

```
// Define all Non-secure (NS) and Non-secure Callable (NSC) Regions
#define REGION0_BASE 0x0001E000UL
#define REGION1_BASE 0x00020000UL
#define REGION2_BASE 0x20004000UL
#define REGION0_LIMIT 0x0001FFFFUL
#define REGION1_LIMIT 0x000FFFFFUL
#define REGION2_LIMIT 0x20017FFFUL
// CMSIS calls to enable SAU Regions
// SAU region 0 - Flash NSC at 120 kB to 128 kB (0x0001E000 - 0x0001FFFF)
SAU->RNR = (0UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION0_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION0_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_NSC_Msk | SAU_RLAR_ENABLE_Msk;
// SAU region 1 - Flash NS at 128 KB to 1024 kB (0x00020000 - 0x000FFFFF)
SAU->RNR = (1UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION1_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION1_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;
// SAU region 2 - RAM NS at 16 kB to 96 kB (0x20004000 - 0x20017FFF)
SAU->RNR = (2UL & SAU_RNR_REGION_Msk);
SAU->RBAR = (REGION2_BASE & SAU_RBAR_BADDR_Msk);
SAU->RLAR = (REGION2_LIMIT & SAU_RLAR_LADDR_Msk) | SAU_RLAR_ENABLE_Msk;
// CMSIS functions to enable SAU and clear data and instruction pipe
TZ_SAU_Enable();
__DSB();
__ISB();
```

Copyright © 2025 Silicon Laboratories. All rights reserved.



BLS BMPU Programming

Bus Master Privileged Attribute

A Bus Master can be configured as either privileged (default) or unprivileged by programming the corresponding index in the `SMU_BMPUPATDn` register.

```
// Configure all odd Bus Masters unprivileged
for (i = 0; i < SMU_NUM_BMPUS; i++) {
  if (i & 0x01) {
    SMU->BMPUPATD0 &= ~(1 << i);
  }
}
```

Bus Master Security Attribute

A Bus Master can be configured as either Secure (default) or Non-secure by programming the corresponding index in the `SMU_BMPUPATDn` register. Configure a Bus Master as Non-secure results in the Bus Master only being able to access Non-secure addresses.

```
// Configure all odd Bus Masters Non-secure
for (i = 0; i < SMU_NUM_BMPUS; i++) {
  if (i & 0x01) {
    SMU->BMPUSATD0 &= ~(1 << i);
  }
}
```

Bus Master Fault Status

The Bus Master ID and the address that triggered the fault can be read from the `SMU_BMPUFS` and `SMU_BMPUFSADDR` registers.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint32_t fs_bmpu_id;
uint32_t fs_bmpu_addr;
uint32_t fs_bmpu_secfault;
// Read Bus Master fault status
fs_bmpu_id = SMU->BMPUFS;
fs_bmpu_addr = SMU->BMPUFSADDR;
fs_bmpu_secfault = (SMU->IF & _SMU_IF_BMPUSEC_MASK) >> _SMU_IF_BMPUSEC_SHIFT;
// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_BMPUSEC;
```

BLS PPU Programming

Peripheral Privileged Attributes

A peripheral can be configured as either privileged (default) or unprivileged by programming the corresponding index in the `SMU_PPUPATDn` register.

```
// Configure all odd peripherals unprivileged
for (i = 0; i < SMU_NUM_PPU_PERIPHS; i++) {
  if (i & 0x01) {
    if (i >= 32){
      SMU->PPUPATD1 &= ~(1 << (i-32));
    } else {
      SMU->PPUPATD0 &= ~(1 << i);
    }
  }
}
```

Notes:

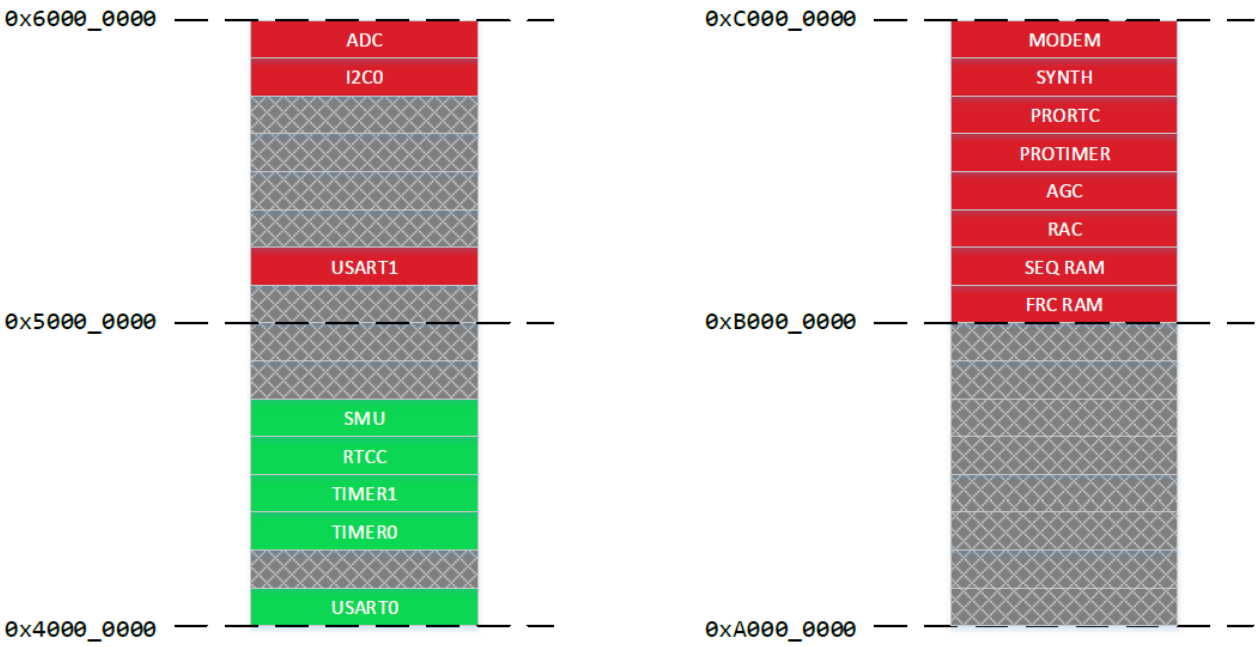
- The peripherals in `SMU_PPUPATD0` and `SMU_PPUPATD0` are device-dependent.
- The privileged attribute of the radio subsystem (`AHBRADIO` index) is configured as a unit.

Peripheral Security Attributes

A peripheral can be configured as either Secure (default) or Non-secure by programming the corresponding index in the `SMU_PPUSATDn` register. The figure below shows the memory map when the ADC, I2C0, USART1, and RADIO are configured as Non-secure and other peripherals (e.g., SMU, RTCC, TIMER1, TIMER0, USART0...) as Secure.

```
// Configure all the Non-secure peripherals
SMU->PPUSATD0 &= ~SMU_PPUSATD0_USART1;
SMU->PPUSATD1 &= ~(SMU_PPUSATD1_I2C0 | SMU_PPUSATD1_IADC0 | SMU_PPUSATD1_AHBRADIO);
```

Copyright © 2025 Silicon Laboratories. All rights reserved.



Notes:

- The peripherals in `SMU_PPUSATD0` and `SMU->PPUSATD1` are device-dependent.
- The security attribute of the radio subsystem (`AHBRADIO` index) is configured as a unit.

Peripheral Fault Status

The peripheral ID that triggered the fault can be read from the `SMU_PPUFS` register.

```
uint32_t fs_ppu_periph_id;
uint32_t fs_sec_fault;
uint32_t fs_priv_fault;
uint32_t fs_inst_fault;
// Read peripheral fault status
fs_ppu_periph_id = SMU->PPUFS;
fs_sec_fault = (SMU->IF & _SMU_IF_PPUSEC_MASK) >> _SMU_IF_PPUSEC_SHIFT;
fs_priv_fault = (SMU->IF & _SMU_IF_PPUPRIV_MASK) >> _SMU_IF_PPUPRIV_SHIFT;
fs_inst_fault = (SMU->IF & _SMU_IF_PPUINST_MASK) >> _SMU_IF_PPUINST_SHIFT;
// Clear the IF to capture a new fault
SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_PPUPRIV | SMU_IF_PPUINST;
```

Floating Point Unit (FPU) Programming

If the Non-secure application enables the FPU at initialization, the Secure software needs to set up the `NSACR` register in SCB to grant the FPU access for Non-secure software.

```
// Enable Non-secure access to the FPU
SCB->NSACR |= SCB_NSACR_CP10_Msk + SCB_NSACR_CP11_Msk;
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

TrustZone Implementation

TrustZone Implementation

The goal of TrustZone implementation is to provide Secure Key Storage that can keep access to keys limited to Secure applications while at the same time allowing Non-secure applications to exercise the keys. It is an added feature for the SVM devices that do not have dedicated hardware for Secure Key Storage as in SVH devices.

The PSA Crypto is placed in a Secure region to keep key material hidden from the Non-secure application. The exposed PSA Crypto APIs stay the same while the backend provides persistent key encryption and decryption similar to the key wrapping and unwrapping functionality of the SVH device.

The following items need to be considered when upgrading the existing system for Secure Key Storage with TrustZone.

- System Configuration
- Gecko Bootloader
- Secure Library
- TrustZone Secure Key Storage
- PSA Attestation
- SE Manager
- Common Vulnerabilities and Exposures (CVE)

System Configuration

The system configuration includes the following items:

- Enable system exceptions in the Secure state.
- Set the security attributes of different regions in the SAU and ESAU.
- Place peripherals and associated interrupts in either Secure or Non-secure applications.
- Assign the Bus Masters' security attributes.
- The system has two Secure/Non-secure pairs for the bootloader and application. The Secure part of each pair is responsible for properly configuring the split in its Secure application before branching to the Non-secure application.

> Note: The secure application will issue a software reset at startup (fatal error) if the device's SE firmware version is lower than the first version that supports TrustZone.

System Exceptions

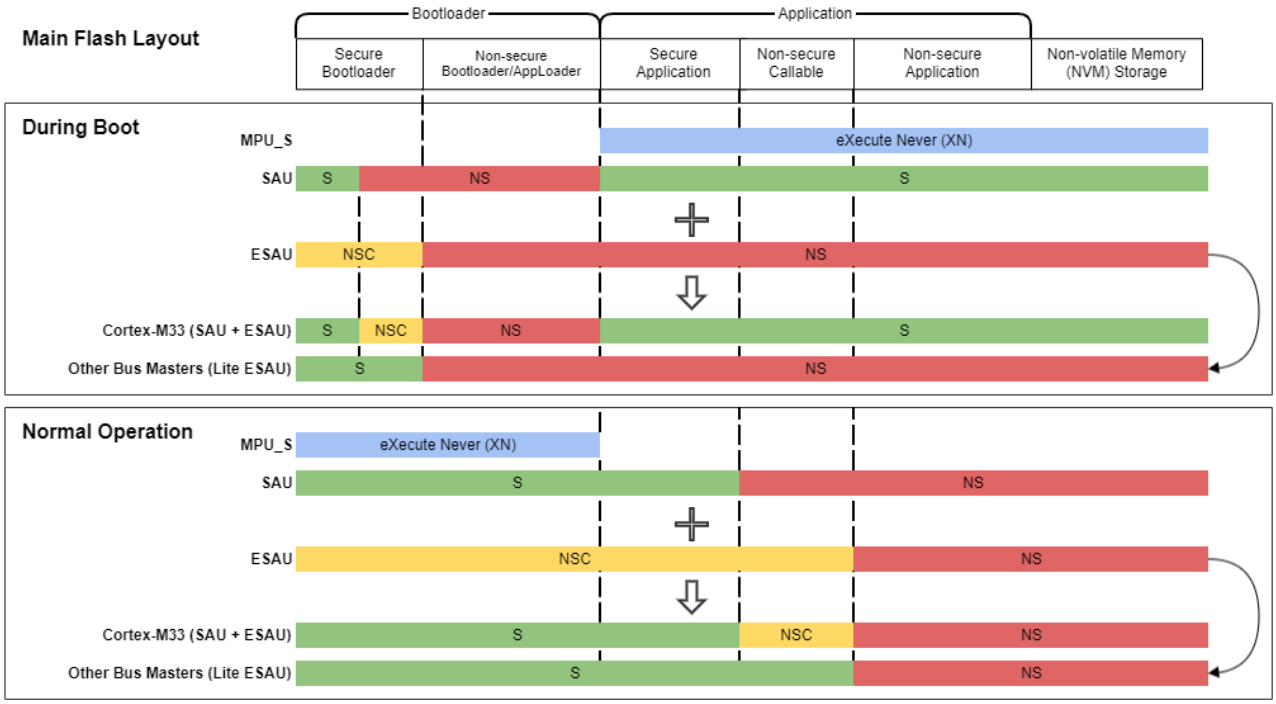
The following system exceptions are enabled in the Secure state for the bootloader and application.

- MemManage Fault
- BusFault
- UsageFault
- SecureFault

Main Flash Layout

The following figure is an overview of the main flash layout that covers the isolation requirements for the Secure Key Storage solution. The SAU and ESAU configurations provide the required security to the Cortex-M33 and other Bus Masters during boot and normal operation.

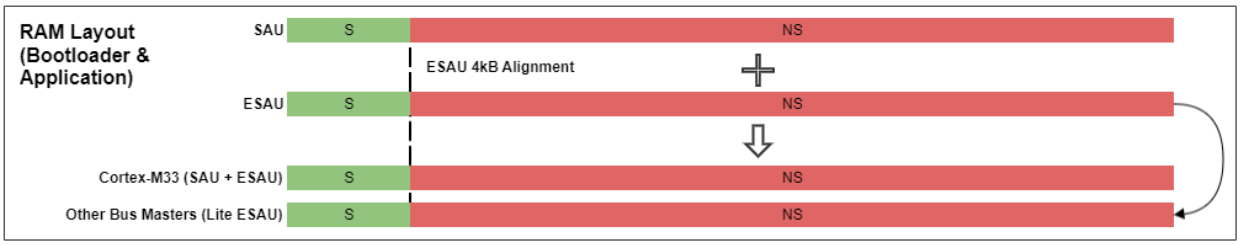
Copyright © 2025 Silicon Laboratories. All rights reserved.



1. Settings:
   - The application is set to non-executable (XN) by Secure MPU to avoid any code execution in this area during boot.
   - The bootloader is set to non-executable (XN) by Secure MPU to avoid any code execution in this area during normal operation.
2. The ESAU configuration only uses the NSC section by setting mrb01 to the base address of region 0. The reason is that lite ESAU in other Bus Masters treats both S and NSC as a Secure attribute. For the Cortex-M33, the SAU upgrades the NSC in the ESAU to Secure. The 32 bytes region alignment of SAU also relaxes the 4 kB alignment restriction on the start address of the NSC in ESAU.
3. The whole application area is set to Secure in SAU for Cortex-M33 during boot to hide details from the bootloader NS part.
4. The ESAU cannot mark any region that comes after a Non-secure section as Secure (must be in the order of S/NSC/NS). Therefore the Secure application area does not align between the Cortex-M33 (SAU + ESAU) and other Bus Masters (lite ESAU) during boot. The secrets stored in that Secure region expose as Non-secure for other Bus Masters during boot (no such issue in normal operations). So the application must not save any plaintext secrets in that Secure region to overcome this limitation during boot.
5. The NVM storage is in the Non-secure region, so the application must encrypt the persistent keys before storing them in this area.

RAM Layout

The following figure is an overview of the RAM layout used for the bootloader and application. The SAU and ESAU are used to split the RAM into a Secure and Non-secure region (Non-secure Callable is not required).



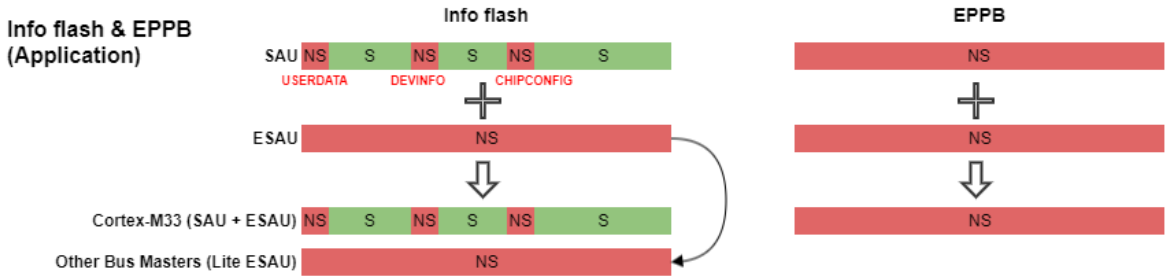
In practice, the Secure part (bootloader or application) takes ownership of the amount of RAM it needs from the beginning of RAM and leaves the rest (up to the ESAU 4 kB alignment requirement) configured as Non-secure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The bootloader does not know how the application partitions the RAM between Secure and Non-secure. So bootloader removes any secrets from RAM before handing control to the application.

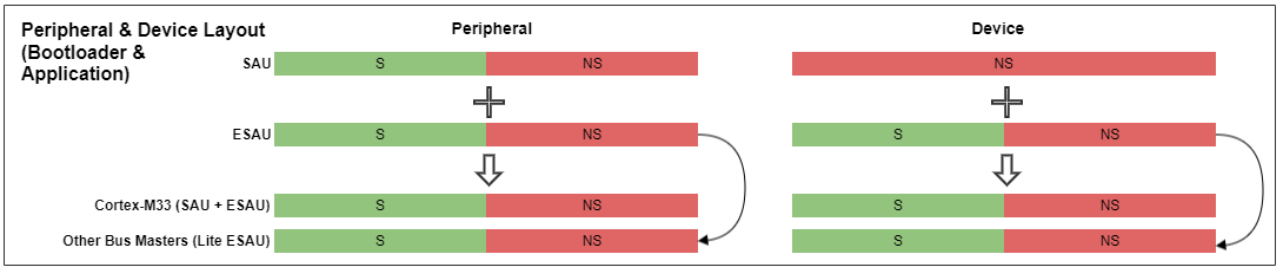
Info Flash and EPPB

The following figure is an overview of the Info flash and EPPB layout for the application. The Cortex-M33 core is the only Bus Master that can access the EPPB region.



Peripheral and Device

The following figure is an overview of the peripheral and device layout for the bootloader and application. The SAU and ESAU are used to split the peripheral and device into a Secure and Non-secure region.



The Secure software is responsible for moving all peripherals and associated interrupts to the Non-secure state at startup, except for the peripherals and interrupts that need to be Secure. The Non-secure software must not include code that attempts to directly access any peripheral that is used by the Secure software.

Peripherals owned by the Secure software (application)

1. Security Management Unit (SMU)
   - It prevents Non-secure software from changing the configuration for the ESAUs, BMPUs, and PPUs.
   - Except for EFR32xG21 devices, some features are also available in the dedicated Non-secure version of SMU registers (`SMU_CFGNS`).
2. CRYPTOACC (VSE devices) or SEMAILBOX (HSE devices)
   - The crypto engine is placed in the Secure domain for Secure library.
3. System Configuration (SYSCFG)
   - It prevents Non-secure software from changing system configurations for Secure software.
   - Except for EFR32xG21 devices, some features are also available in the dedicated Non-secure version of SYSCFG registers (`SYSCFG_CFGNS`).
4. Memory System Controller (MSC)
   - It prevents Non-secure software from writing to Secure flash.

Peripheral interrupts owned by the Secure software:

Copyright © 2025 Silicon Laboratories. All rights reserved.

Table: Secure Peripheral Interrupts (Application)

VSE Device	HSE Device
SMU_SECURE_IRQn	SMU_SECURE_IRQn
SYSCFG_IRQn	SYSCFG_IRQn
MSC_IRQn	MSC_IRQn
CRYTOACC_IRQn	SEMBRX_IRQn
TRNG_IRQn	SEMBTX_IRQn
PKE_IRQn	

The `PRIS` bit in the `AIRCR` register is set to 1 to place all Non-secure exceptions/interrupts in lower priority level space. Therefore any Secure exceptions/interrupts can be programmed with higher priority than Non-secure ones.

The `BMPUSEC` and `PPUSEC` interrupt enable flags in the `SMU_IEN` register are set to enable the SMU security fault interrupts (`SMU_SECURE_IRQn`) on Bus Masters and peripherals.

Floating Point Unit (FPU):

The Secure application does not use the FPU. But the Secure startup code also enables the FPU for use by the Non-secure application.

Bus Masters

To keep all secrets from the Non-secure world, only the Bus Masters in the table below can access data in the Secure world. For the Bus Masters living in the Secure world, the secure application must configure their corresponding control interfaces in the peripheral space to Secure. The Cortex-M33 core as a Bus Master is split to run in Secure and Non-secure contexts.

Table: Secure Bus Masters (Application)

Device	Secure Bus Master	Control Interface of Bus Master
VSE	CRYPTOACC	CRYTPOACC
HSE	SEDMA or SEEXTDMA	SEMAILBOX

Notes:

- Use `SMU_BMPUSATD0` register to configure the security attribute of a Bus Master.
- Use `SMU_PPUSATDn` register to configure the control interface of Bus Master as a Secure peripheral.
- LDMA is set as a Non-secure Bus Master to make sure it cannot be used to copy out data from the Secure memory.

Application Transitions

The system contains two Secure/Non-secure pairs.

1. The [bootloader pair](#bootloader pair) has a Secure bootloader and a Non-secure bootloader containing the communication interfaces.
2. The [application pair](#application pair) has a Secure application and a Non-secure application consisting of the wireless stacks (if applicable) and application layers.

As described in the preceding sections, the Secure part of these pairs is responsible for setting the security configurations of the system during startup. For the handover between Secure/Non-secure pairs, the software must restore the system so the Secure part of the other pair can execute and reconfigure the system.

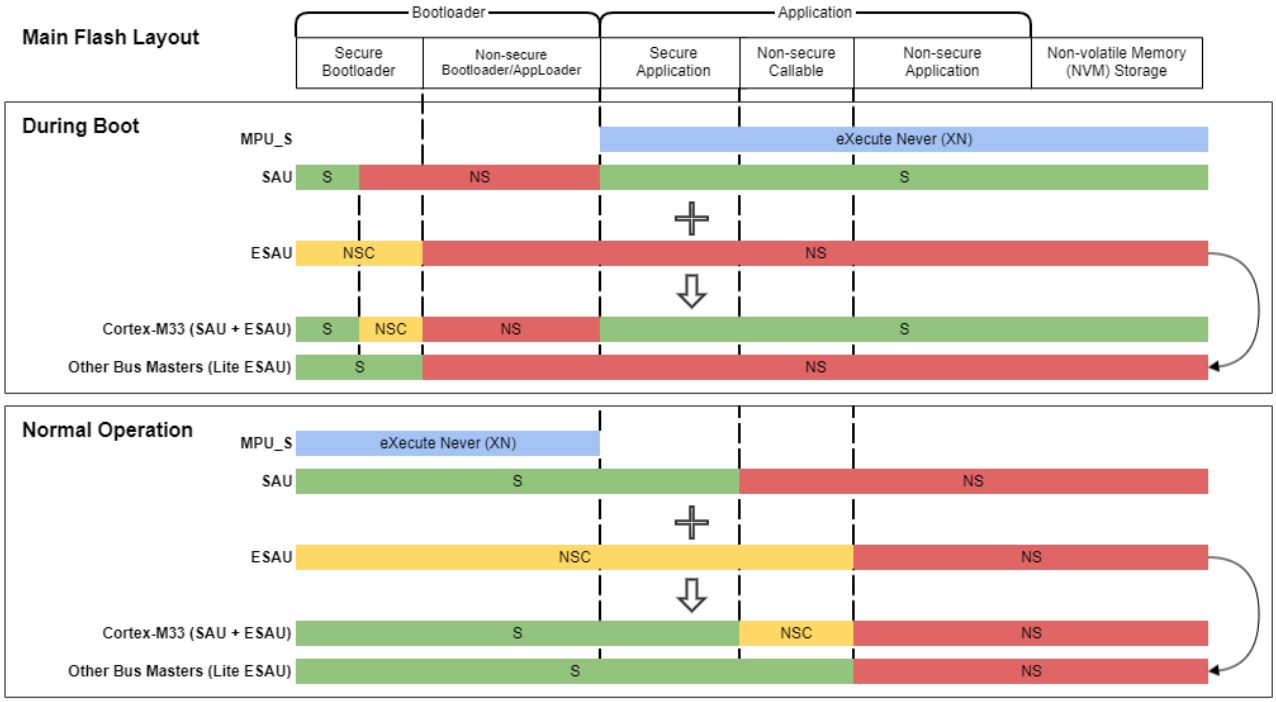
The software must reconfigure the following items before transitioning to the next Secure/Non-secure pair:

- Restored all peripherals and interrupts to Secure
- Reset ESAU to default configuration (all configurable regions to Secure)
- Reset SAU to default configuration (Secure for everything)
- Reset MPU to default configuration (removes any XN)

Gecko Bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Gecko bootloader ensures the Secure assets are protected during the boot flow and normal operation.



1. The SAU and Secure MPU mark all the flash for application and NVM as Secure and non-executable (XN) during boot. It guards against bootloader NS code execution branching into the application code.
2. The bootloader needs to split into Secure and Non-secure software to protect secrets in the system. Secure code can access the entire flash to validate or upgrade the system.
3. For VSE devices, the GBL Decryption Key (AES-128 key) is moved from the NS memory (last page of the main flash) to the Secure part of the bootloader. The Simplicity Commander v1.13 or higher provides a feature to inject the AES-128 key to the bootloader binary file.

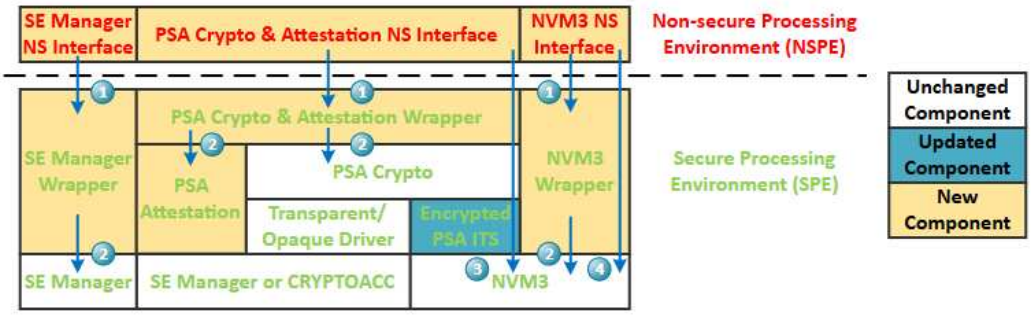
```
commander convert <BL image file> --aeskey <decryption key file> --outfile <BL image with decryption key>
```

4. The bootloader communication interfaces are placed in the NS area to support various communication components below for firmware upgrades.
   - BGAPI UART
   - EZSP-SPI
   - UART XMODEM
5. The NS communication functions call into the bootloader APIs placed in the bootloader NSC region. The Secure application validates all input arguments before processing the request.
6. Before transiting from bootloader to normal operation, it resets the SAU to default configuration to make all the flash for bootloader as Secure.
7. The Non-secure application software can call bootloader APIs through application NSC, and the corresponding Secure function releases the non-executable (XN) restriction on the bootloader during normal operation.

Secure Library

The goal of the Secure library is to keep the PSA Crypto key and attestation token protected from malicious code on the NSPE. The following figure overviews multiple components to support the Secure library.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. The NS interfaces in NSPE are responsible for packing and passing all input arguments over the NSC functions on wrappers in SPE.
2. The wrappers in SPE validate all input arguments before calling into the corresponding APIs in different drivers.
3. Because of the system memory layout limitation, the flash for NVM3 storage is located in the NSPE. Therefore the updated PSA Internal Trusted Storage (ITS) driver needs to encrypt all crypto keys before storing them in Non-secure NVM.
4. Data stored directly using the NVM3 APIs are not encrypted.

The following table describes the new and updated components of the Secure library.

Component	Description
SE Manager NS interface	This component contains SE Manager API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the SE Manager wrapper's NSC functions.
SE manager wrapper	This component contains the interface into SE Manager exposed to the NSPE. These NSC functions grant access to the SE Manager utility API and validate all input arguments before calling into SE Manager.
PSA Crypto & Attestation NS interface	This component contains PSA Crypto and attestation API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the PSA Crypto and attestation wrapper's NSC functions.
PSA Crypto & Attestation wrapper	This component contains the interface into PSA Crypto and attestation exposed to the NSPE. These NSC functions grant access to the entire PSA Crypto and attestation API and validate all input arguments before calling into PSA Crypto and attestation.
PSA attestation	This component in SPE provides the functionality required by the PSA attestation specification.
Encrypted PSA ITS	The PSA ITS layer builds on top of NVM3. This component is updated to support encrypted storage to secure stored keys. The encryption is based on the device's TrustZone Root Key.
NVM3 NS interface	This component contains NVM3 API callable from the NSPE. It packages the list of input arguments in the appropriate format before calling into the NVM3 wrapper's NSC functions.
NVM3 wrapper	This component contains the interface into NVM3 exposed to the NSPE. These NSC functions grant access to the NVM3 API and validate all input arguments before calling into NVM3.

Notes:

- The SE Manager NS interface, PSA Crypto NS interface, and NVM3 NS interface in the NSPE provide drop-in replacement on SE Manager utility, PSA Crypto, and NVM3 APIs for existing wireless stacks and user applications.
- The NSC calls can only take a limited number of arguments, so all NSC functions take a pointer to a list of parameters to support a long list of arguments. All arguments must be validated using the intrinsic functions from CMSIS.

TrustZone Secure Key Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

The TrustZone Secure Key Storage provides a solution to store a user key in Secure RAM or an encrypted form in Non-secure flash.

The TrustZone Root Key stored in the SE NVM for Secure Key Storage encryption is generated or renewed by following operations.

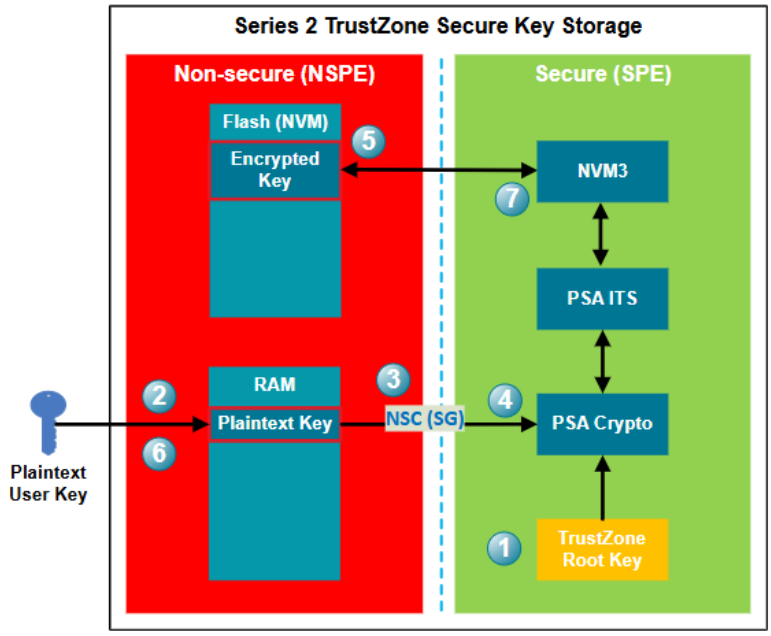
- The TrustZone Root Key had already existed if the shipped Series 2 device with SE firmware version supports this key.
- Generate a TrustZone Root Key when upgrading from a SE firmware version that did not support this key to the one that does.
- Renew a TrustZone Root Key after performing a Device Erase.

> Note: The TrustZone Root Key cannot be renewed if Device Erase is disabled.

The TrustZone Root Key is not exposable to the NSPE, and access to this key in SPE is different in HSE and VSE devices.

- HSE - The SPE can access the TrustZone Root Key as a built-in non-exportable key in HSE NVM.
- VSE - The SPE can access the TrustZone Root Key in Secure RAM, which is copied from VSE NVM during boot.

The TrustZone Root Key value after reset is identical to the value before reset. TrustZone Root Keys are unique on each device. The key allows a user to securely store a key in the Non-secure flash, limiting the number of keys that can be saved only by the amount of Non-secure storage. The following figure describes using the TrustZone Root Key to encrypt a plaintext key and store it in Non-secure NVM.



1. After power-on, the device's TrustZone Root Key is available for the SPE.
2. A user key is generated and imported into the device's Non-secure memory. In this example, the key is imported into Non-secure RAM for easy deletion, and the key is lost if device power is removed.
3. Call PSA Crypto API (`psa_import_key()` or `psa_generate_key()`) through SG in NSC to generate a key for crypto operations.
4. The plaintext key is passed to the PSA Crypto in SPE, where it is encrypted (AES-GCM) with the TrustZone Root Key.
5. The encrypted key is stored to the NVM in NSPE through the PSA ITS and NVM3 drivers.
6. The plaintext key can now be deleted from the Non-secure RAM.
7. Only the PSA Crypto in SPE can retrieve the encrypted key from NVM in NSPE and decrypt it for crypto operations in SPE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note: Ignore steps 2 and 6 if the plaintext key is randomly generated by the PSA Crypto.

The following tables describe the storage differences between SVM and SVH devices with and without TrustZone Secure Key Storage (SKS).

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (without TrustZone SKS)	RAM	RAM
Persistent Plaintext (without TrustZone SKS)	NVM	NVM
Volatile Wrapped (without TrustZone SKS)	Not supported	RAM (1)
Persistent Wrapped (without TrustZone SKS)	Not supported	NVM (1)

Key Type	Storage on SVM Device	Storage on SVH Device
Volatile Plaintext (with TrustZone SKS)	Secure RAM (2)	Secure RAM
Persistent Plaintext (with TrustZone SKS)	Encrypted plaintext key in NS NVM (2)	Encrypted plaintext key in NS NVM
Volatile Wrapped (with TrustZone SKS)	Not supported	Secure RAM
Persistent Wrapped (with TrustZone SKS)	Not supported	Encrypted wrapped key in NS NVM

Notes:

- The NVM or NS NVM is at the last part of the main flash.
- It is possible to replace the wrapped key solution on the SVH device (1) with TrustZone Secure Key Storage on the SVM device (2), but this is a less secure approach.

PSA Attestation

The device attestation service creates a token that contains a fixed set of device-specific data when requested by the caller. Each device must have a unique Initial Attestation Key (IAK) pair. The device uses the private IAK to sign the token, and the caller uses the public IAK to verify the token's authenticity.

The generation of the private IAK is different in SVM and SVH devices.

- SVM - If the private IAK does not exist in NVM3, it is randomly generated when requested from the PSA Attestation driver and saved to NVM3 through the TrustZone Secure Key Storage.
- SVH - The private IAK is generated and securely stored in the HSE during chip production.

An Entity Attestation Token (EAT) is a mini-report that is cryptographically signed. An EAT is encoded in either one of two standardized data formats: a Concise Binary Object Representation (CBOR) or in the text-based format JSON. A digital signature is then used to protect its content. The technical specification defining the content of the EAT, which are claims about the hardware and the software running on a device, is specified by the Internet Engineering Task Force (IETF).

The EAT is a crypto-signed report card with claims. A claim is a data item that is represented as a Key-Value pair. Claims can relate to the device's pedigree or anything one wants the device to attest. Collected data can originate from the Root of Trust (RoT), any protected area, or non-protected areas.

The EAT must be signed following the structure of the CBOR Object Signing and Encryption (COSE) specification. For asymmetric key algorithms, the signature structure must be COSE-Sign1. A COSE-Sign1 is a CBOR encoded, self-secured data blob that contains headers, a payload, and a signature.

The primary need for EAT verification is to check correct formation and signing as for any token. In addition, though, the verifier can operate a policy where values of some of the claims in this profile can be compared to reference values that are registered with the verifier for a given deployment, to confirm that the device is endorsed by the manufacturer supply chain.

The PSA attestation token (aka Initial Attestation Token - IAT) is a profiled EAT. The Series 2 device will generate this token by (Nonce claim below) unless the SE OTP is uninitialized or the `SECURE_BOOT_ENABLE` option in SE OTP is disabled.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The following tables describe claims used in the PSA attestation token of the Series 2 device.

Table: Claims of PSA Attestation Token

Key	Claim Name (Present)	Claim Description	Claim Value
265 (-75000)	Profile Definition (Must)	The Profile Definition claim encodes the unique identifier corresponds to the EAT profile.	http://arm.com/psa/2.0.0
2394 (-75001)	Client ID (Must)	The Client ID claim represents the security domain of the caller.	See note below (2 byes)
2395 (-75002)	Security Lifecycle (Must)	The Security Lifecycle claim represents the current lifecycle state of the PSA RoT.	Device dependent (2 bytes)
2396 (-75003)	Implementation ID (Must)	The Implementation ID claim uniquely identifies the implementation of the immutable PSA RoT.	Device dependent (32 bytes)
2397 (-75004)	Boot Seed (Optional)	The Boot Seed claim represents a value created at system boot time that will allow differentiation of reports from different boot sessions.	Device dependent (32 bytes)
2399 (-75006)	Software Components (Must)	The Software Components claim is a list of software components that includes all the software loaded by the PSA RoT.	See note below
10 (-75008)	Nonce (Must)	The Nonce claim is used to carry the challenge provided by the caller to demonstrate freshness of the generated token. The length must be either 32, 48, or 64 bytes.	Random nonce (32/48/64 bytes)
256 (-75009)	Instance ID (Must)	The Instance ID claim represents the unique identifier of the IAK. The length must be 33 bytes.	SHA-256 hash of public IAK (32 bytes) with header 0x01

Notes:

- Some claims MUST be present in a PSA attestation token.
- The keys `-7500x` were defined in a previous version of the PSA attestation token specification (`PSA_IOT_PROFILE_1` profile) that is still used in the HSE-SVH firmware.
- The actual claims returned from the tokens on the SVH device are HSE firmware version-dependent.
- Key 2394: In PSA, a security domain is represented by a signed integer where negative values represent callers from the NSPE and positive IDs represent callers from the SPE. The value 0 is not permitted.
- Key 2395 (For the definitions of these lifecycle states, refer to the ARM PSA Security Model):
  - UNKNOWN (`0x0000 - 0x00ff`)
  - ASSEMBLY_AND_TEST (`0x1000 - 0x10ff`)
  - PSA_ROT_PROVISIONING (`0x2000 - 0x20ff`)
  - SECURED (`0x3000 - 0x30ff`)
  - NON_PSA_ROT_DEBUG (`0x4000 - 0x40ff`)
  - RECOVERABLE_PSA_ROT_DEBUG (`0x5000 - 0x50ff`)
  - DECOMMISSIONED (`0x6000 - 0x60ff`)
- Key 2396:
  - Word[0]: Die revision
  - Word[1]: SE OTP version (return 0 for VSE SE firmware < v1.2.14)
  - Word[2]: Security capability (not applicable to HSE-SVH device, always returns 1 in this word)
    - 0: Unknown security capability
    - 1: Security capability not applicable
    - 2: Basic security capability
    - 3: Root of Trust security capability
    - 4: HSE-SVM security capability
    - 5: HSE-SVH security capability (run HSE-SVM binary on HSE-SVH device)
  - Word[3]: Production version
  - Word[4:7]: Reserved (zeros)
-

Copyright © 2025 Silicon Laboratories. All rights reserved.

Key 2399: Each software component uses the attributes described in the following table, and some MUST be present in a software component claim.

Key	Attribute (Present)	Description	Value
1	Measurement Type (Optional)	The Measurement Type attribute is a short string representing the role of this software component.	See note below
2	Measurement Value (Must)	The Measurement Value attribute represents a hash of the invariant software component in memory at startup time.	SHA-256 hash (32 bytes) of the firmware
4	Version (Optional)	The Version attribute is the issued software version in the form of a text string.	A string of 8 bytes

The following measurement types may be used for Key 1:

- "BL": a Bootloader
- "PRoT": a component of the PSA Root of Trust
- "ARoT": a component of the Application Root of Trust
- "App": a component of the NSPE application
- "TS": a component of a Trusted Subsystem

The PSA Attestation API allows access to the PSA attestation token, so an external entity can cryptographically verify the identity and trust status of the device.

Table: PSA Attestation API

PSA Attestation API	Usage
psa_initial_attest_get_token	Retrieve the PSA attestation Token.
psa_initial_attest_get_token_size	Calculate the size of a PSA attestation Token.
sl_tz_attestation_get_public_key	Get the public IAK key for PSA attestation token signature verification.

> Note: The `sl_tz_attestation_get_public_key` is a Silicon Labs custom API.

SE Manager

SE Manager is the foundation for the Secure library cryptographic operations on HSE devices. It means that SE Manager has to move into the SPE.

The following SE Manager core APIs are always available in the NSPE.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_init	Y	Y	Y
sl_se_deinit	Y	Y	Y
sl_se_init_command_context	Y	Y	Y
sl_se_deinit_command_context	Y	Y	Y
sl_se_set_yield	Y	Y	Y

The following SE Manager core APIs expose to the NSPE through the NSC interface for the VSE devices.

SE Manager Core API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_read_executed_command	Y	-	-
sl_se_ack_command	Y	-	-

Copyright © 2025 Silicon Laboratories. All rights reserved.

The following SE Manager utility APIs expose to the NSPE through the NSC interface for configuring the security features of HSE or VSE devices.

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_check_se_image	Y	Y	Y
sl_se_apply_se_image	Y	Y	Y
sl_se_get_upgrade_status_se_image	Y	Y	Y
sl_se_check_host_image	Y	Y	Y
sl_se_apply_host_image	Y	Y	Y
sl_se_get_upgrade_status_host_image	Y	Y	Y
sl_se_init_otp_key	Y	Y	Y
sl_se_read_pubkey	Y	Y	Y
sl_se_init_otp	Y	Y	Y
sl_se_read_otp	Y	Y	Y
sl_se_get_se_version	Y	Y	Y
sl_se_get_debug_lock_status	Y	Y	Y
sl_se_apply_debug_lock	Y	Y	Y
sl_se_get_otp_version	Y	Y	Y
sl_se_write_user_data	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_erase_user_data	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_reset_cause	-	Y (EFR32xG21 only)	Y (EFR32xG21 only)
sl_se_get_status	-	Y	Y
sl_se_get_serialnumber	-	Y	Y
sl_se_enable_secure_debug	-	Y	Y
sl_se_disable_secure_debug	-	Y	Y

SE Manager Utility API	VSE-SVM	HSE-SVM	HSE-SVH
sl_se_set_debug_options	-	Y	Y
sl_se_erase_device	-	Y	Y
sl_se_disable_device_erase	-	Y	Y
sl_se_get_challenge	-	Y	Y
sl_se_roll_challenge	-	Y	Y
sl_se_open_debug	-	Y	Y
sl_se_disable_tamper	-	-	Y
sl_se_read_cert_size	-	-	Y
sl_se_read_cert	-	-	Y

> Note: The NSPE cannot access the other SE Manager APIs for cryptographic and attestation operations.

Common Vulnerabilities and Exposures (CVE)

At this writing, the following known TrustZone CVE had been fixed in the current implementation.

- CVE-2020-16273: Stack sealing

Copyright © 2025 Silicon Laboratories. All rights reserved.

CVE-2021-36465: VLLDM instruction/floating-point vulnerability

Copyright © 2025 Silicon Laboratories. All rights reserved.

Upgrade Existing Application To TrustZone

Upgrade Existing Application to TrustZone

The main concerns when upgrading existing deployment to the TrustZone solution are:

- The Secure/Non-secure pair for the bootloader (24 kB) does not fit inside the current allotted bootloader space (16 kB).
- The Secure/Non-secure pair for the application does not fit inside the current allotted application space.
- The PSA ITS moves from a non-encrypted to an encrypted format, so the existing stored cryptographic keys in NVM3 cannot be reused after upgrading the current application to TrustZone.

The Secure Library is based on PSA Crypto, so the existing application cannot integrate with the TrustZone if one of the following conditions is valid.

- Use SE Manager APIs for cryptographic and attestation operations.
- Use classic Mbed TLS APIs for cryptographic operations (except for X.509 certificate) and Transport Layer Security (TLS) protocol.

System Requirements

The following table lists the tools and software required for TrustZone development on Series 2 devices.

Tool/Software	Required Version	Description
GCC	v10.3.1	Fix a bug (ID 99271) on cmse_nonsecure_call attribute.
IAR EWARM	v9.20.4	Fix a bug (EWARM-9484) on __cmse_nonsecure_call attribute.
Segger J-Link	≥ v7.6.2c	v7.6.2c is the first version to add basic TrustZone support on Series 2 devices.
Simplicity Studio	≥ v5.6.3.0	v5.6.3.0 is the first version to support TrustZone software development on Series 2 devices.
Simplicity Commander	≥ v1.13.3	v1.13.3 includes a TrustZone-aware flash loader and supports features required for TrustZone development.
GSDK	≥ v4.2.2	GSDK v4.2.2 is the first version to support TrustZone software development on Series 2 devices.
SE Firmware	≥ v1.2.14	v1.2.14 is the first version to fully support TrustZone on xG21 (HSE) and xG22 (VSE) devices.
SE Firmware	≥ v2.2.1	v2.2.1 is the first version to fully support TrustZone on other Series 2 HSE and VSE devices.

Notes:

- Required GCC and IAR EWARM versions are GSDK-dependent.
- Bug list of GCC v10.3
- IAR EWARM release note
- Segger J-Link release note
- Simplicity Studio user guide
- Latest version of Simplicity Commander
- GSDK release note
- Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required TrustZone features. The latest SE firmware image and release notes after installing the GSDK (Windows folder): C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public

Peripheral Addresses in Device Header Files

Copyright © 2025 Silicon Laboratories. All rights reserved.

The device header files (e.g., efr32mg21b020f1024im32.h) need to be configurable for different situations. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` definitions specify whether the compilation is for Secure or Non-secure applications. The `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` should be exclusive. If none of the definitions are true, the state should be similar to the Non-secure configuration, but the startup code (`SystemInit()` in system_*.c) will be responsible for reconfiguring the system.

Define (Software Component)	Default Peripheral Pointer	Startup Code
SL_TRUSTZONE_SECURE (TrustZone Secure)	Point to Secure peripherals (*_BASE = *_S_BASE)	No effect on SystemInit()
SL_TRUSTZONE_NONSECURE (TrustZone Non-Secure)	Point to Non-secure peripherals (*_BASE = *_NS_BASE)	No effect on SystemInit()
None of the above (-)	Point to Non-secure peripherals (*_BASE = *_NS_BASE)	SystemInit() moves peripherals to Non-secure

When building a Secure application (`SL_TRUSTZONE_SECURE` is true), all peripherals shall have their non-suffixed default address pointing to the Secure location of the peripheral (e.g., EMU). But the definitions in sl_trustzone_secure_config.h can force the addresses of specific peripherals pointing to the Non-secure location.

```
#ifndef SL_TRUSTZONE_SECURE_CONFIG_H
#define SL_TRUSTZONE_SECURE_CONFIG_H
// Specify security configuration of peripherals. Peripherals that are not
// included here will automatically have their _BASE addresses point to their
// secure address. This might not be true, since most peripherals are configured
// to be non-secure -- but it's also not a problem if the peripheral is not
// accessed from the S app.
// Used in multiple places.
#define SL_TRUSTZONE_PERIPHERAL_CMU_S (0)
// Used by SE Manager service.
#define SL_TRUSTZONE_PERIPHERAL_AHBRADIO_S (0)
// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMA_S (1)
// Used by MSC service.
#define SL_TRUSTZONE_PERIPHERAL_LDMAXBAR_S (1)
#endif // SL_TRUSTZONE_SECURE_CONFIG_H
```

```
#if defined(SL_CATALOG_TRUSTZONE_SECURE_CONFIG_PRESENT)
#include "sl_trustzone_secure_config.h"
#endif
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_EMU_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_EMU_S) && (SL_TRUSTZONE_PERIPHERAL_EMU_S != 0)))
#define EMU_BASE              (EMU_S_BASE)                  /* EMU base address */
#else
```

In other cases (`SL_TRUSTZONE_NONSECURE` is true or both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), all peripherals shall have their non-suffixed default address pointing to the Non-secure location of the peripheral (e.g., EMU).

```
#define EMU_BASE              (EMU_NS_BASE)                 /* EMU base address */
```

> Note: Do not install the TrustZone Secure or TrustZone Non-Secure software component to the TrustZone-unaware application.

Startup Code

Copyright © 2025 Silicon Laboratories. All rights reserved.

The startup code moves peripherals from Secure to Non-secure to support the default peripheral locations. In a TrustZone-aware application (either `SL_TRUSTZONE_SECURE` or `SL_TRUSTZONE_NONSECURE` is true), this is the application's responsibility (skip lines 168 to 194 in `SystemInit()`) and is done in the Secure firmware of the system.

For the TrustZone-unaware application (both `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` are false), the `SystemInit()` in system_*.c (e.g., system_efr32mg21.c) moves peripherals to the Non-secure location.

- The `SystemInit()` sets the accesses of all peripherals to Non-secure except for the SMU and HSE SEMAILBOX (lines 172 to 178).
- The `SystemInit()` sets the SAU in All Non-secure configuration (lines 180 to 187).
  - It ensures Non-secure access to Non-secure peripherals.
  - The device component files (e.g., `efr32mg21b020f1024im32.slcc`) enable the CMSE compiler option (`-mcmse` for GCC and `--cmse` for IAR) to pass the condition in line 181 to program the SAU.
  - To catch the missing CMSE compiler option, it will generate a preprocessor error (line 186) if the CMSE flag is not set when manually upgrading a project from GSDK v4.0.x to ≥v4.1.x for the TrustZone-unaware application.
- The `SystemInit()` does not program the ESAU (default Secure flash is 32 MB), so the whole program is run in the Secure state.
- The `SystemInit()` also enables the `BMPUSEC` and `PPUSEC` interrupts in the SMU (lines 189 to 193). It ensures the TrustZone-unaware application catches any violations of Bus Master and peripheral security access permissions.

```
144   void SystemInit(void)
145   {
146   #if defined (__VTOR_PRESENT) && (__VTOR_PRESENT == 1U)
149
150   #if defined(UNALIGNED_SUPPORT_DISABLE)
153
154   #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
158
159   /* Secure app takes care of moving between the security states.
160    * SL_TRUSTZONE_SECURE MACRO is for secure access.
161    * SL_TRUSTZONE_NONSECURE MACRO is for non-secure access.
162    * When both the MACROS are not defined, during start-up below code makes sure
163    * that all the peripherals are accessed from non-secure address except SMU,
164    * as SMU is used to configure the trustzone state of the system. */
165   #if !defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_NONSECURE) \
166     && defined(__TZ_PRESENT)
167
168   #if (_SILICON_LABS_32B_SERIES_2_CONFIG >= 2)
169     CMU->CLKEN1_SET = CMU_CLKEN1_SMU;
170   #endif
171
172     /* config SMU to Secure and other peripherals to Non-Secure. */
173     SMU->PPUSATD0_CLR = _SMU_PPUSATD0_MASK;
174   #if defined (SEMAILBOX_PRESENT)
175     SMU->PPUSATD1_CLR = (_SMU_PPUSATD1_MASK & (~SMU_PPUSATD1_SMU & ~SMU_PPUSATD1_SEMAILBOX));
176   #else
177     SMU->PPUSATD1_CLR = (_SMU_PPUSATD1_MASK & ~SMU_PPUSATD1_SMU);
178   #endif
179
180     /* SAU treats all accesses as non-secure */
181   #if defined(__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
182     SAU->CTRL = SAU_CTRL_ALLNS_Msk;
183     __DSB();
184     __ISB();
185   #else
186     #error "The startup code requires access to the CMSE toolchain extension to set proper SAU settings."
187   #endif /* __ARM_FEATURE_CMSE */
188
189   /* Clear and Enable the SMU PPUSEC and BMPUSEC interrupt. */
190     NVIC_ClearPendingIRQ(SMU_SECURE_IRQn);
191     SMU->IF_CLR = SMU_IF_PPUSEC | SMU_IF_BMPUSEC;
192     NVIC_EnableIRQ(SMU_SECURE_IRQn);
193     SMU->IEN = SMU_IEN_PPUSEC | SMU_IEN_BMPUSEC;
194   #endif /*SL_TRUSTZONE_SECURE */
195   }
```

The `SMU_BASE` and HSE `SEMAILBOX_HOST_BASE` in device header files must point to the Secure location regardless of the `SL_TRUSTZONE_SECURE` and `SL_TRUSTZONE_NONSECURE` settings to avoid security violations on peripherals in the TrustZone-unaware application (SMU and HSE SEMAILBOX are set to Secure peripherals).

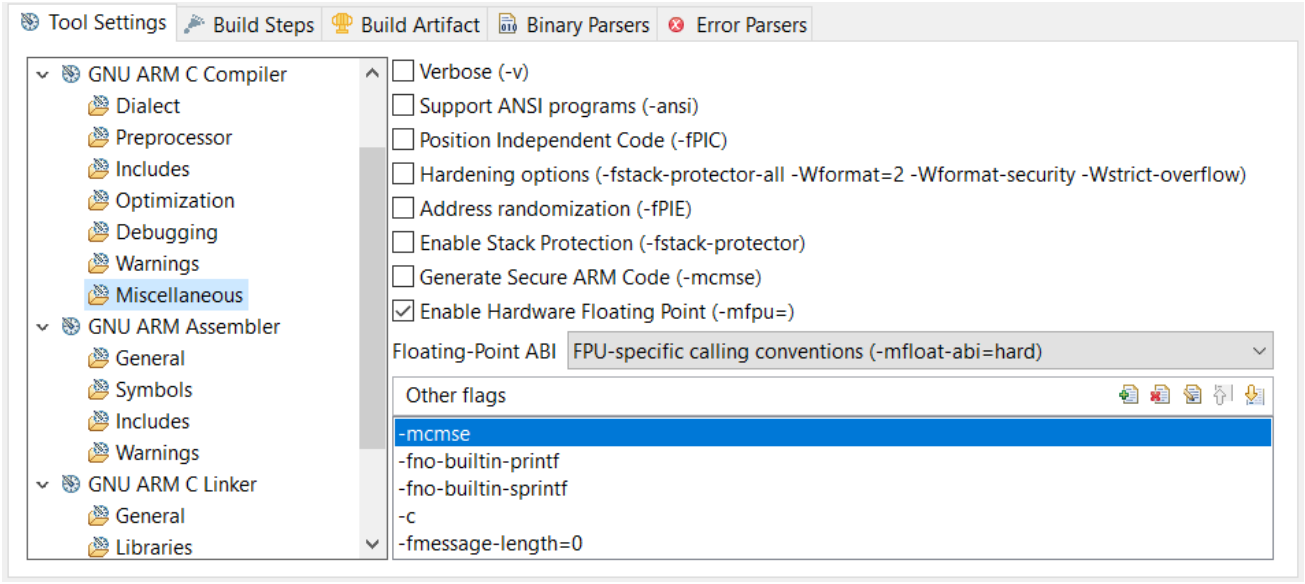
Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SMU_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_SMU_S) && (SL_TRUSTZONE_PERIPHERAL_SMU_S != 0)))
#define SMU_BASE                (SMU_S_BASE)                    /* SMU base address */
#else
#define SMU_BASE                (SMU_S_BASE)                    /* SMU base address */
```

```
#if ((defined(SL_TRUSTZONE_SECURE) && !defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S))
    || (defined(SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S) && (SL_TRUSTZONE_PERIPHERAL_SEMAILBOX_HOST_S != 0)))
#define SEMAILBOX_HOST_BASE     (SEMAILBOX_S_HOST_BASE)         /* SEMAILBOX_HOST base address */
#else
#define SEMAILBOX_HOST_BASE     (SEMAILBOX_S_HOST_BASE)         /* SEMAILBOX_HOST base address */
```

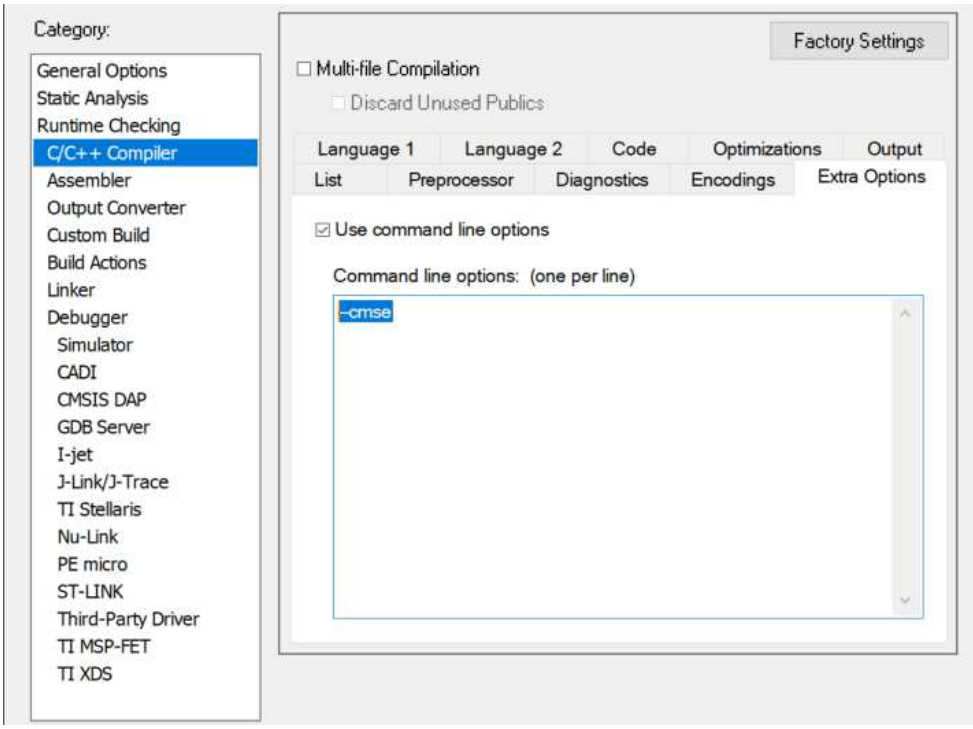
Notes:

- The CMSE compiler option of GCC is in the Other flagswindow under C/C++ Build → Settings → Tool Settings → GNU ARM C Compiler→ Miscellaneous.



- The CMSE compiler option of IAR is in the Command line options: (one per line) window under Options... → C/C++ Compiler → Extra Options.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Linker File

The `template_contribution` defined in the slcp files of Secure and Non-secure projects will override the default memory settings defined in the device component files (e.g., `efr32mg21b020f1024im32.slcc`) to generate the linker files for Secure and Non-secure applications.

Memory Region	Default Setting in Device Component File	Override Setting in template_contribution
Flash start address	device_flash_addr	memory_flash_start
Flash size	device_flash_size	memory_flash_size
RAM start address	device_ram_addr	memory_ram_start
RAM size	device_ram_size	memory_ram_size

Default setting (efr32mg21b020f1024im32.slcc)

```
- name: device_family
  value: efr32mg21
- name: device_flash_addr
  value: 0
- name: device_flash_size
  value: 1048576
- name: device_flash_page_size
  value: 8192
- name: device_ram_addr
  value: 536870912
- name: device_ram_size
  value: 98304
```

template_contribution example (Secure)

```
template_contribution:
  - name: memory_flash_start
    value: 0x0
  - name: memory_flash_size
    value: 0x2C000
  - name: memory_ram_start
    value: 0x20000000
  - name: memory_ram_size
    value: 0x3000
```

template_contribution example (Non-secure)

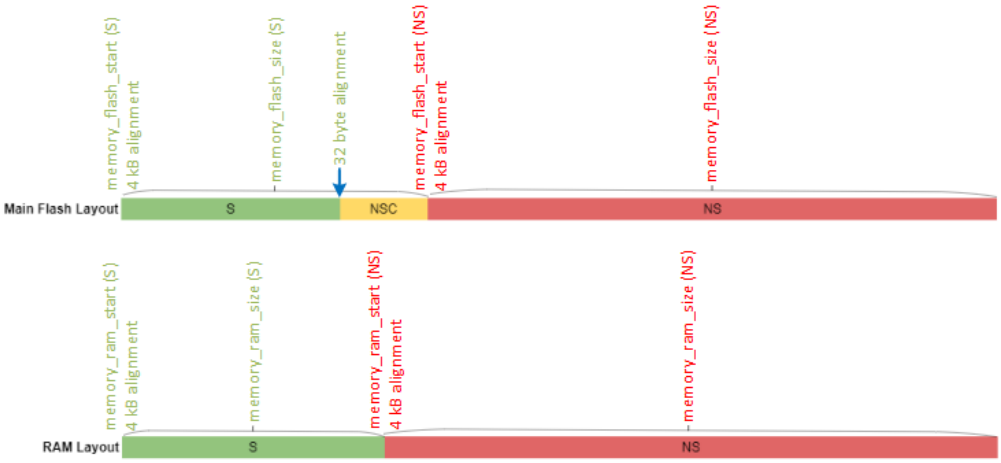
```
template_contribution:
  - name: memory_flash_start
    value: 0x2C000
  - name: memory_flash_size
    value: 0x54000
  - name: memory_ram_start
    value: 0x20003000
  - name: memory_ram_size
    value: 0x5000
```

The ESAU sets the flash and RAM start address, so these addresses should be alignment at 4 kB (0x1000). The Secure project linker file needs to have a section for NSC (Secure Gateway) at the end of the Secure flash section. The SAU sets the start address of the NSC section, so this section only needs to be 32 bytes aligned.

- GCC NSC: The `.gnu.sgstubs` region in the Secure application map file (.map)
- IAR NSC: The `Veneer$$CMSE` region in the Secure application map file (.map)

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Secure and Non-secure flash and RAM sizes are incremented or decremented in 4 kB. The memory configurations in Secure and Non-secure applications are correlated, so the flash and RAM settings are in pairs.



> Note: Users should not directly edit the `template_contribution` in the `slcp` file, but rather use the Memory Editor in Simplicity Studio to update the memory configuration.

Debugger

Simplicity Studio supports two debuggers:

- GNU Debugger (GDB) client and SEGGER's GDB server
- Simplicity Studio Debugger

The TrustZone-unaware and TrustZone-aware applications enable the `PPUSEC` interrupts in the SMU. The debugger will trigger the `SMU_SECURE_IRQHandler` if the [Registers] or [Peripherals] view feature violates peripheral security access permission.

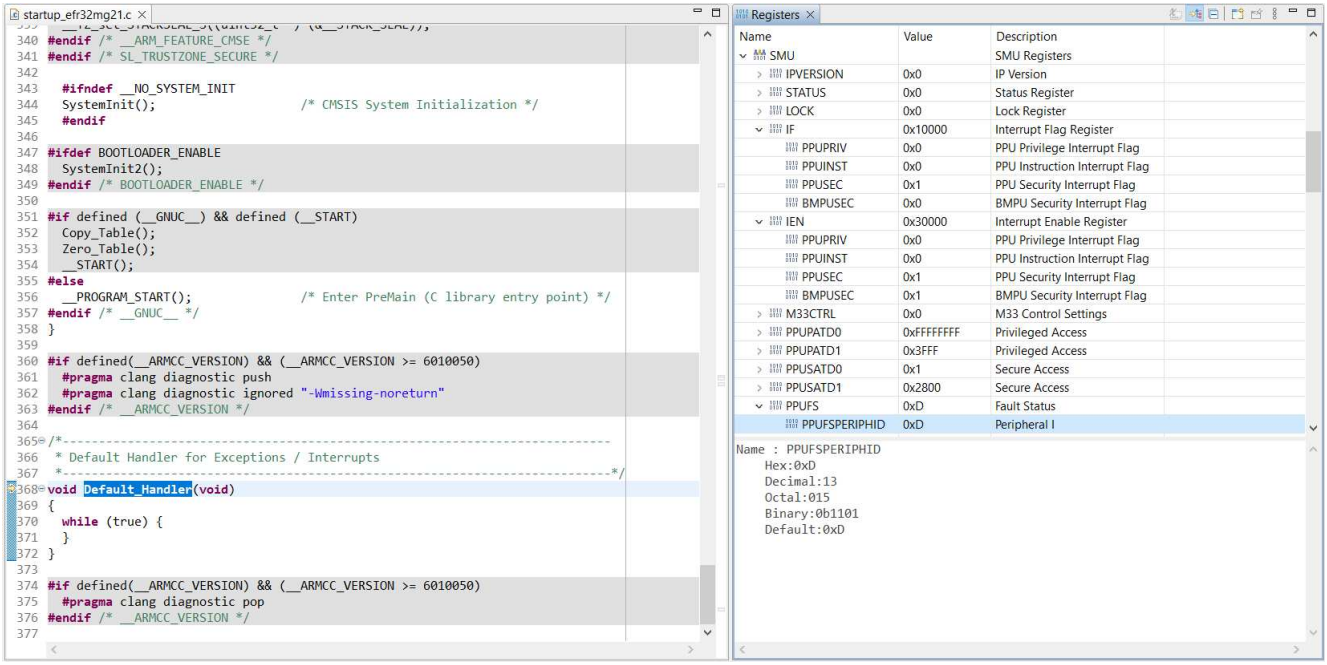
Simplicity Studio Debugger

The [Registers] view of Simplicity Studio Debugger can only access the Secure location of a peripheral. The following figure demonstrates the `Default_Handler` (`SMU_SECURE_IRQHandler` not defined) is triggered (`PPUSEC` in `SMU->IF` = 1) when viewing the registers of GPIO peripheral (`PPUFSPERIPHID` = 13) that is set to Non-secure access in the SMU.

The debugger can access the registers of the SMU since this peripheral is set to Secure access in the SMU.

This limitation does not apply to GSDK < v4.1.0 since no peripherals are configured for Non-secure access.

Copyright © 2025 Silicon Laboratories. All rights reserved.

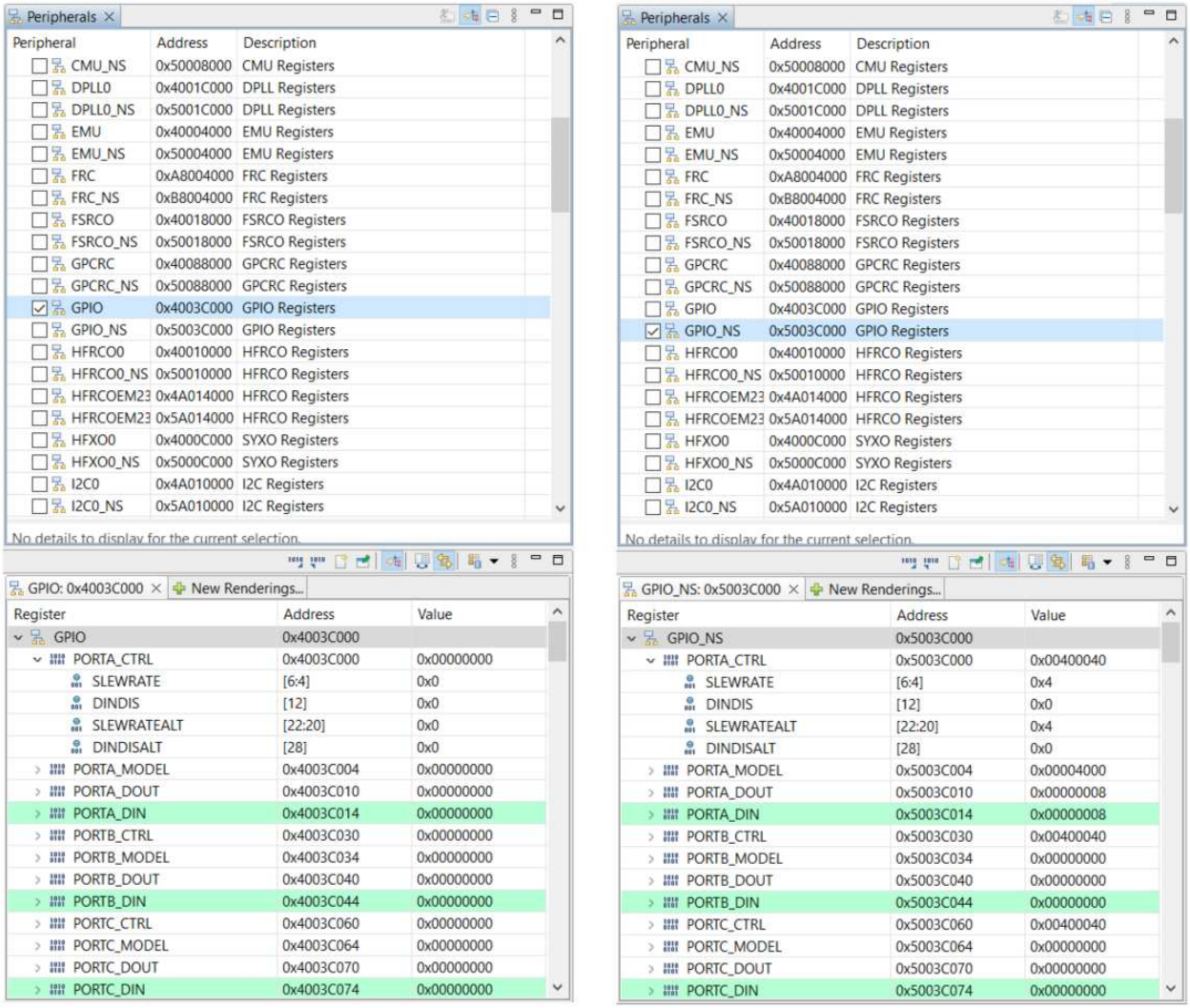


The Simplicity Studio Debugger is not the preferred choice for TrustZone debugging since it has limitations on viewing Non-secure access peripherals.

GNU Debugger (GDB)

The [Peripherals] view of GNU Debugger can access either the Secure or Non-secure location of the peripheral to avoid conflicts on security access permission. The following figure shows the registers of GPIO on Secure (`GPIO` at `0x4003C000`) and Non-secure (`GPIO_NS` at `0x5003C000`) addresses. The GPIO peripheral is set to Non-secure access in the SMU, so the registers in the Secure address are displayed as zero.

Copyright © 2025 Silicon Laboratories. All rights reserved.



The GNU Debugger is the preferred choice for TrustZone debugging and is the default debugger for Simplicity Studio ≥ v5.5.0.0.

Copyright © 2025 Silicon Laboratories. All rights reserved.

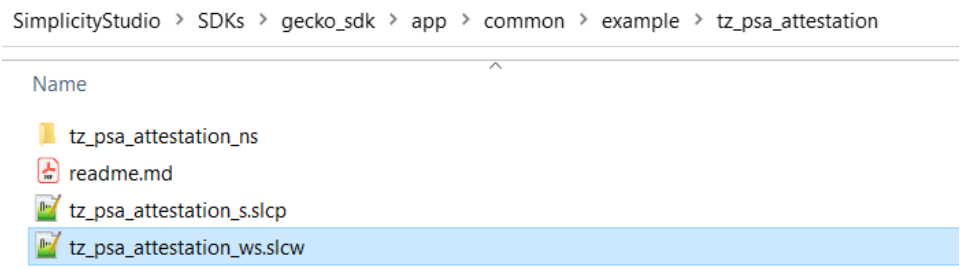
TrustZone Platform Examples

TrustZone Platform Examples

The following TrustZone platform examples located in the C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\app\common\example folder (Windows) demonstrate the TrustZone implementation on Series 2 devices. All TrustZone platform examples do not include Gecko Bootloader.

TrustZone PSA Attestation





Example Folder	Description
tz_psa_attestation	The workspace description file (tz_psa_attestation_ws.slcw) creates the TrustZone PSA Attestation example. The project description file (tz_psa_attestation_s.slcp) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_attestation_ns	The project description file (tz_psa_attestation_ns.slcp) configures a Non-secure application for the TrustZone PSA Attestation example.

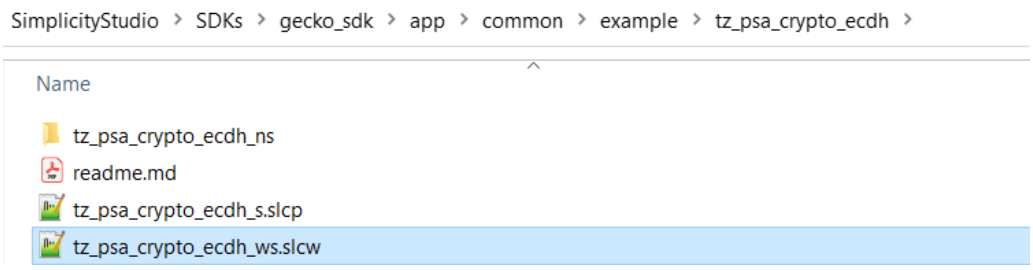
Notes:

- This example cannot run if the `SECURE_BOOT_ENABLE` (root of trust of the attestation) option in SE OTP is disabled.
- The combined image of Secure and Non-secure applications is signed by the `example_signing_key.pem` (private key) in C:\Users<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\commonfolder (Windows). The `example_signing_pubkey.pem` (public key) in the same folder is installed to the SE OTP to verify the image signature during Secure Boot.

TrustZone PSA Crypto ECDH

Copyright © 2025 Silicon Laboratories. All rights reserved.





Example Folder	Description
tz_psa_crypto_ecdh	The workspace description file (tz_psa_crypto_ecdh_ws.slcw) upgrades the existing Platform - PSA Crypto ECDH example to TrustZone-aware. The project description file (tz_psa_crypto_ecdh_s.slcp) configures a Secure application that provides the Secure Library functionality required by the Non-secure application.
tz_psa_crypto_ecdh_ns	The project description file (tz_psa_crypto_ecdh_ns.slcp) configures the existing Platform - PSA Crypto ECDH example as a Non-secure application. The source code can be reused without changes.

The following sections use Simplicity Studio v5.6.3.0 and GSDK v4.2.2. The procedures and pictures may be different if using higher versions of Simplicity Studio 5 and GSDK.

Project Description File

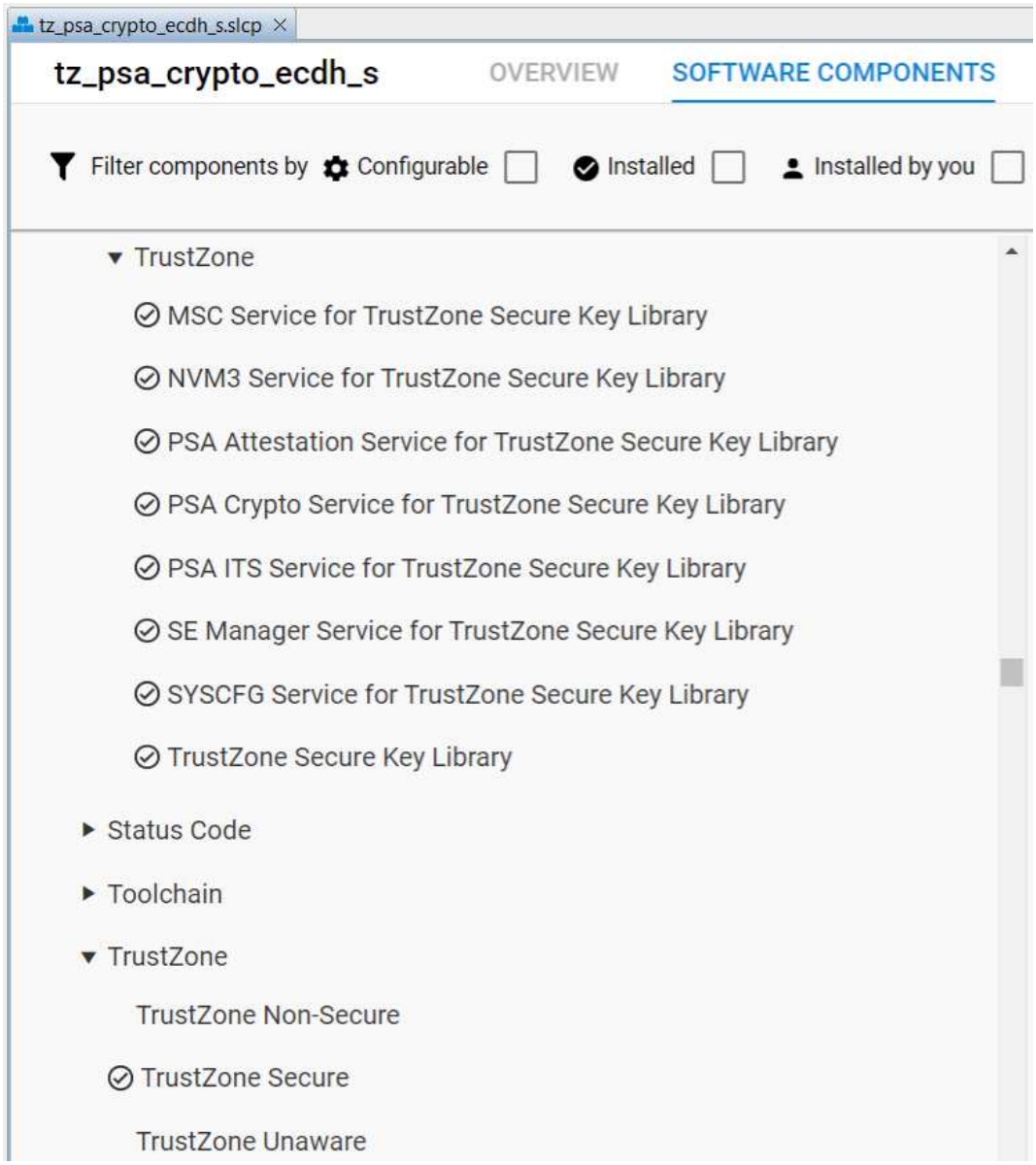
The project description file (`.slcp`) contains references to the GSDK used and a list of components to use from these. The TrustZone-aware application requires separate `slcp` files for the Secure and Non-secure applications.

Users should not directly edit the `slcp` files, but rather use the Memory Editor and Post Build Editor in Simplicity Studio to update the memory configuration and post-build actions.

Secure Application

The following figure describes which TrustZone software components are installed for the TrustZone Secure library of the TrustZone PSA Crypto ECDH example.

Copyright © 2025 Silicon Laboratories. All rights reserved.



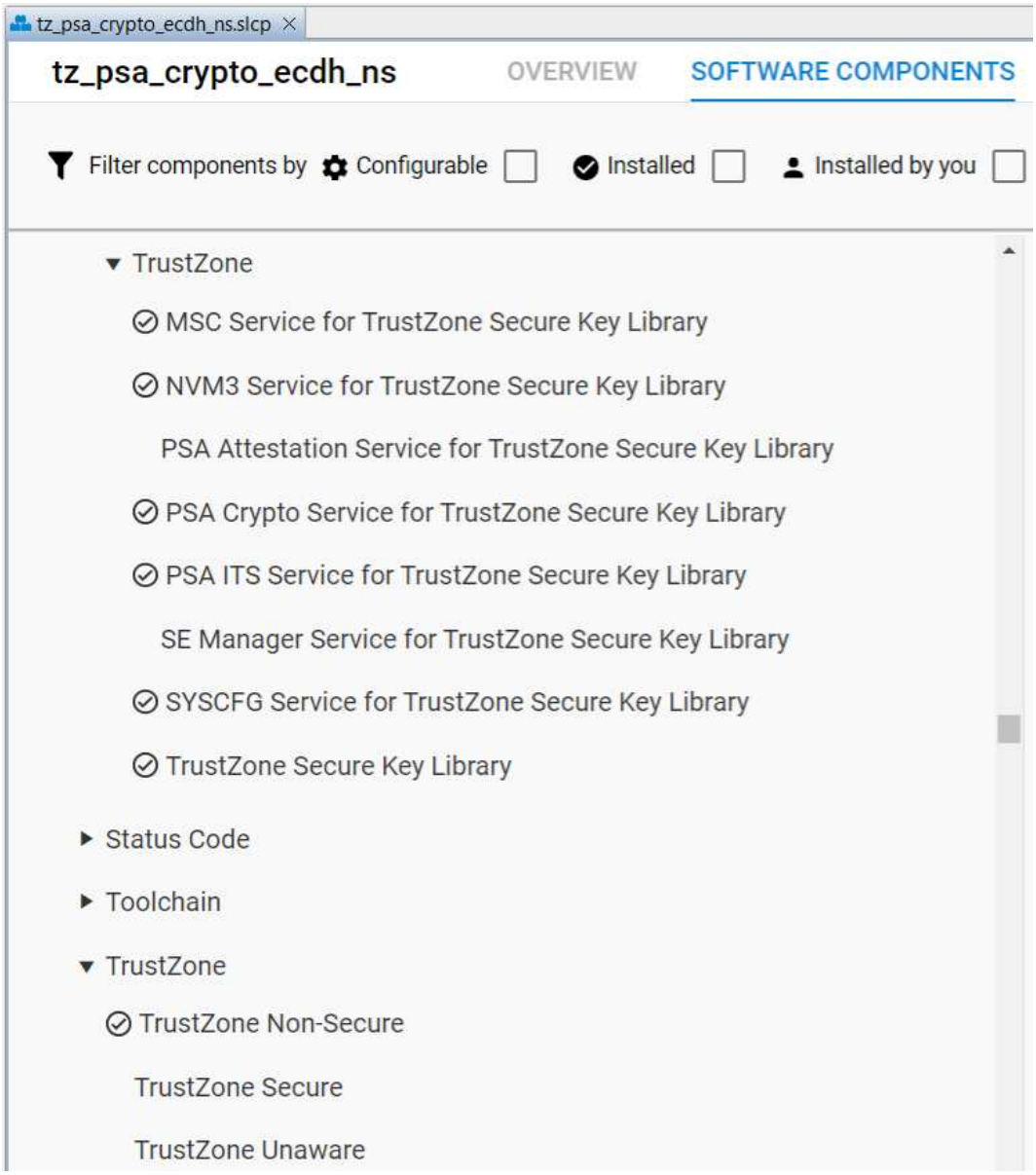
Notes:

- The services provided by the Secure library are standardized.
- The source files for the Secure library will be automatically added to the application when generating the Secure project from the `slcp` file. For the current TrustZone implementation, modifications of the source files of the Secure library are not recommended.

Non-secure Application

The following figure describes which TrustZone software components are installed for the Non-secure application of the TrustZone PSA Crypto ECDH example.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Notes:

- The following software components are automatically installed when PSA Crypto and ITS services are used on the Non-secure application.
  - MSC Service for TrustZone Secure Key Library
  - NVM3 Service for TrustZone Secure Key Library
  - PSA Crypto Service for TrustZone Secure Key Library
  - PSA ITS Service for TrustZone Secure Key Library
  - SYSCFG Service for TrustZone Secure Key Library
- The following software components can be installed to the Non-secure application when those services are required.
  - PSA Attestation Service for TrustZone Secure Key Library
  - SE Manager Service for TrustZone Secure Key Library

Workspace

A workspace is a structure that can contain multiple projects. 'Workspace' is a generic term for this construct. In the context of Simplicity Studio, where workspace has a different, Eclipse-based, meaning, workspaces are referred to as Solutions.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The workspace description file (`.slcw`) contains references to projects (`.slcp`) that make up the workspace. Users should not directly edit the `slcw` file, but rather use the Post Build Editor in Simplicity Studio to update the post-build actions.

Memory Configuration

The memory configurations in the TrustZone platform examples are based on the Series 2 radio board with minimum flash (512 kB) and RAM (32 kB), so these configurations can run on all Series 2 radio boards. Users can customize the settings when more flash and RAM are available on the selected device.

- Memory flash size (total) = `memory_flash_size` (S) + `memory_flash_size` (NS) = 512 kB
- Memory RAM size (total) = `memory_ram_size` (S) + `memory_ram_size` (NS) = 32 kB

Secure Application

The project description file of the Secure application (`*_s.slcp`) uses the default memory setting below to generate the Secure linker file (linkerfile.ld for GCC and linkerfile.icf for IAR in the project autogen folder).

The actual memory usage during software development is unknown, so it needs to reserve enough flash (`memory_flash_size` : 176 kB) and RAM (`memory_ram_size` : 12 kB) for the Secure part of all TrustZone platform examples. The bigger RAM size (including stack and heap) is mainly for the software fallback on cryptographic operations in PSA Crypto.

Default Memory Setting (Secure)	xG21 and xG22 Devices	Other Series 2 Devices
memory_flash_start	0x00000000	0x08000000
memory_flash_size	0x0002C000 (176 kB)	0x0002C000 (176 kB)
memory_ram_start	0x20000000	0x20000000
memory_ram_size	0x00003000 (12 kB)	0x00003000 (12 kB)

```
MEMORY
{
  FLASH   (rx)  : ORIGIN = 0x0, LENGTH = 0x2c000
  RAM     (rwx) : ORIGIN = 0x20000000, LENGTH = 0x3000
}
```

Non-secure Application

The project description files of the Non-secure application (*_ns.slcp) use the default memory setting below to generate the Non-secure linker file (linkerfile.ld for GCC and linkerfile.icf for IAR in the project autogen folder).

The actual memory usage during software development is unknown, so the remaining flash (`memory_flash_size` : 336 kB) and RAM (`memory_ram_size` : 20 kB) should be big enough for the Non-secure part of all TrustZone platform examples.

Default Memory Setting (Non-secure)	xG21 and xG22 Devices	Other Series 2 Devices
memory_flash_start	0x0002C000 (176 kB)	0x0802C000 (176 kB)
memory_flash_size	0x00054000 (336 kB)	0x00054000 (336 kB)
memory_ram_start	0x20003000 (12 kB)	0x20003000 (12 kB)
memory_ram_size	0x00005000 (20 kB)	0x00005000 (20 kB)

```
MEMORY
{
  FLASH   (rx)  : ORIGIN = 0x2c000, LENGTH = 0x54000
  RAM     (rwx) : ORIGIN = 0x20003000, LENGTH = 0x5000
}
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

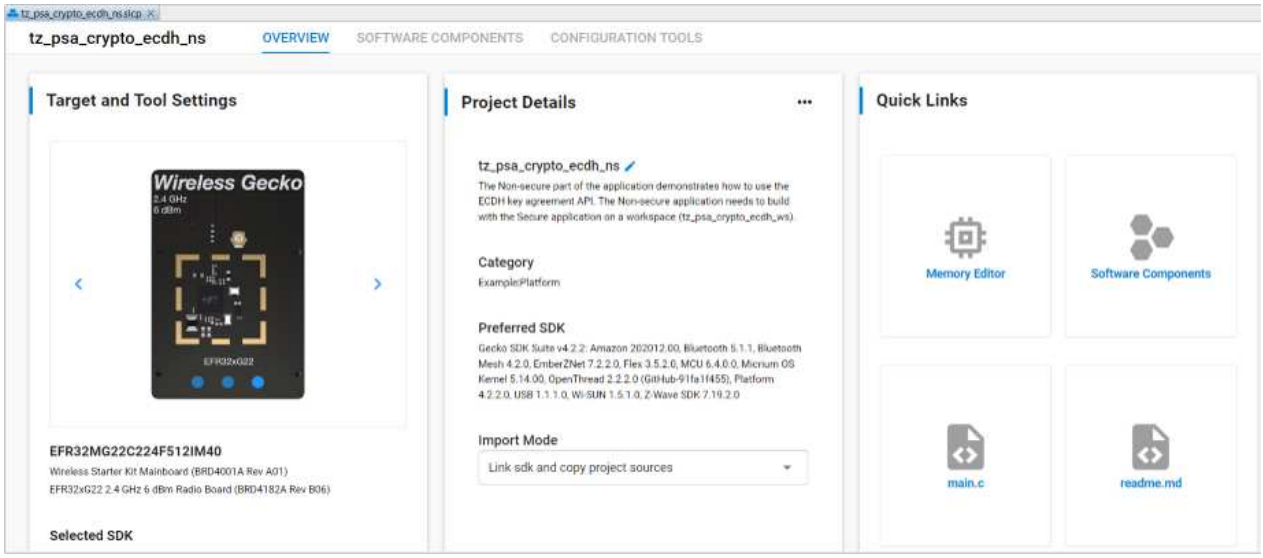
> Note: The usable flash for Non-secure code should be equal to `memory_flash_size - NVM size` (default is 40 kB) if NVM3 storage is required.

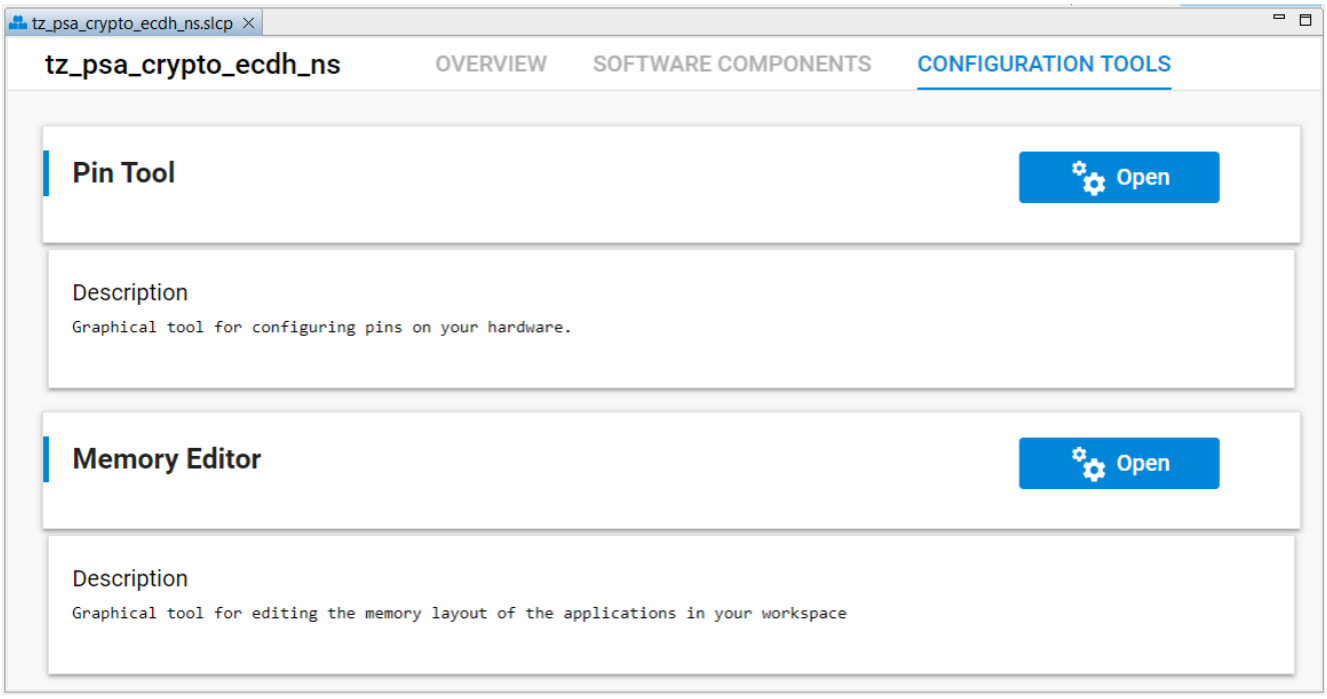
Memory Editor

The default memory setting of Secure and Non-secure applications are good enough for software development and debugging. The final memory layouts of Secure and Non-secure projects are deduced by inspecting the flash and RAM usage in the Secure application memory map file (.map).

The Memory Editor in Simplicity Studio 5 is a graphical tool for editing the memory layout (flash and RAM) of the applications in the workspace. The Memory Editor will update the linker file in the project `autogen` folder with the custom settings. Rebuild the projects to use the new memory configurations in the linker files.

The Memory Editor is located at the Quick Links and CONFIGURATION TOOLS of Secure or Non-secure `slcp` file.

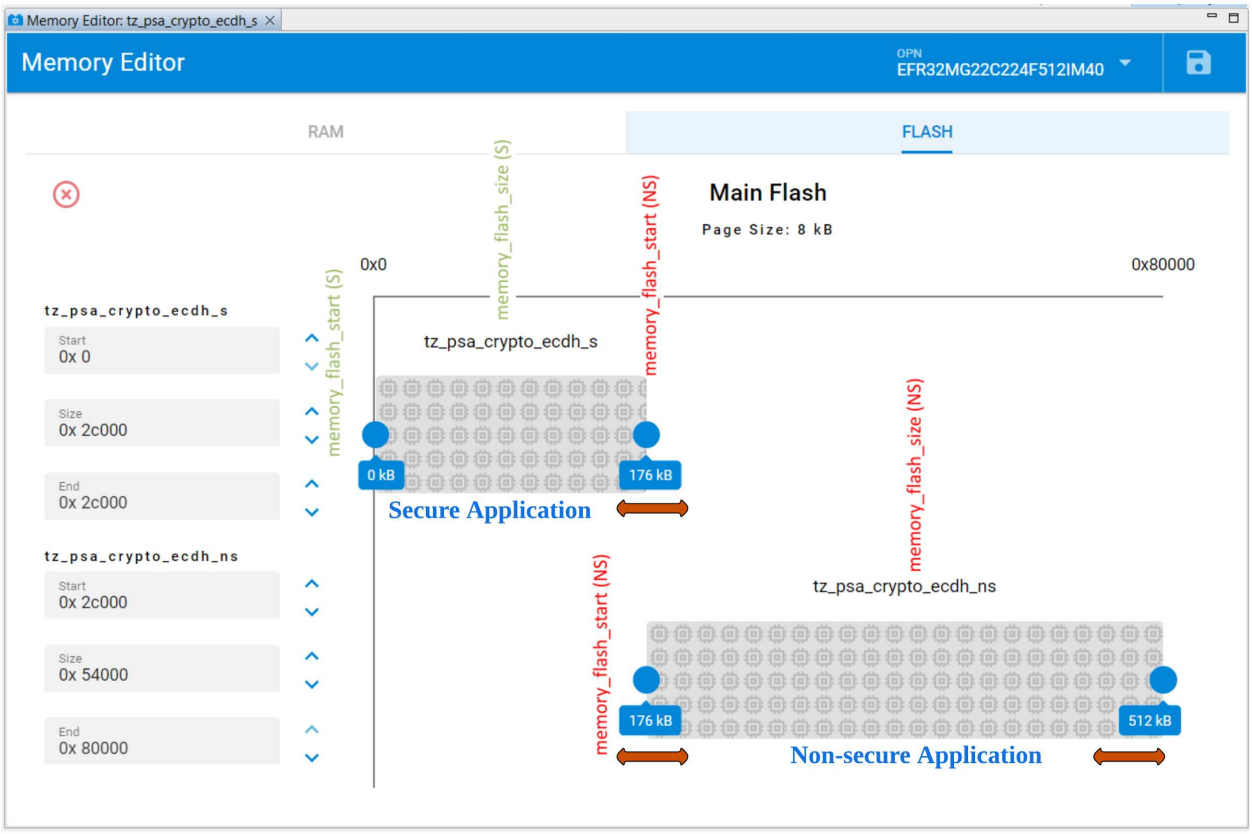




Copyright © 2025 Silicon Laboratories. All rights reserved.

The following items will be determined by the flash usage in the Secure application memory map file:

- memory_flash_size (S)
- memory_flash_start (NS)
- memory_flash_szie (NS)

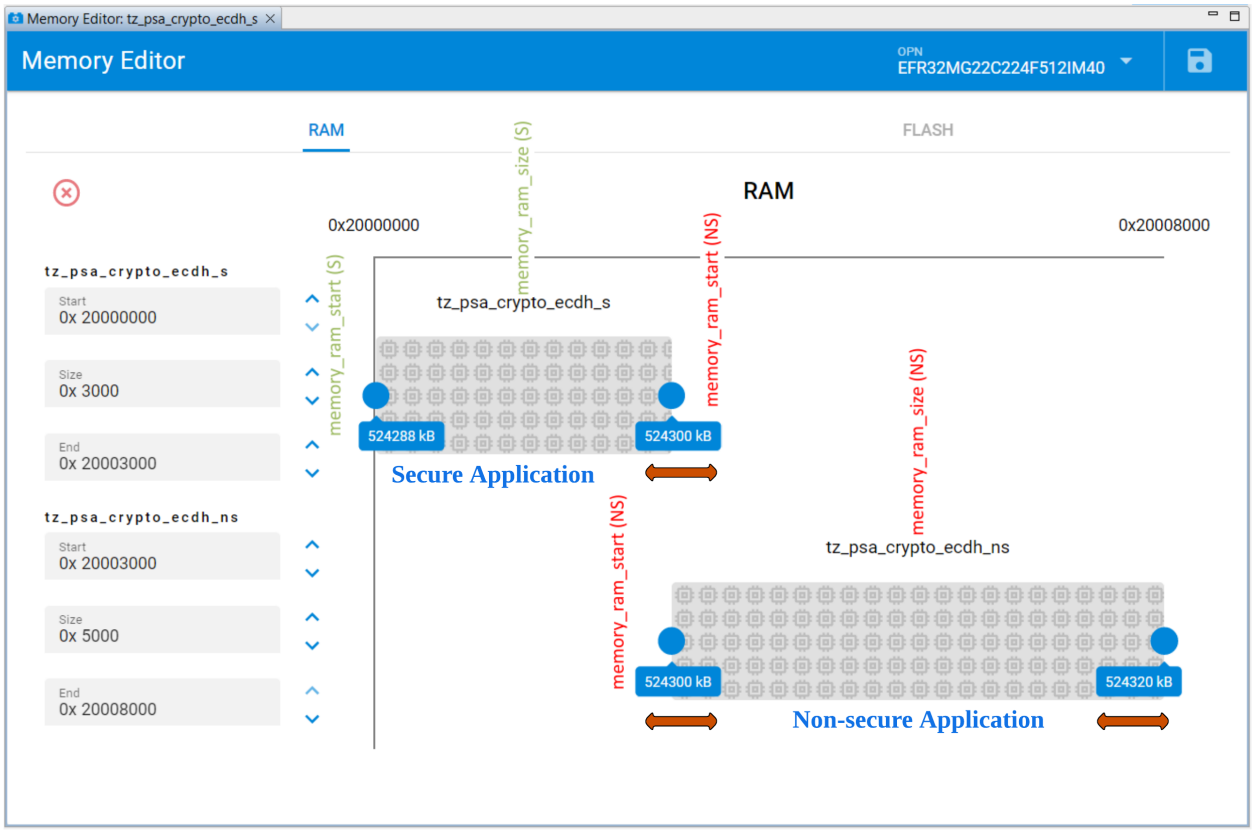


> Note: The Memory Editor in Simplicity Studio v5.6.3.0 can only adjust the flash size in 8 kB (page size) alignment, which may not fit the 4kB alignment between the Secure and Non-secure flash boundary.

The following items will be determined by the RAM usage in the Secure application memory map file:

- memory_ram_size (S)
- memory_ram_start (NS)
- memory_ram_szie (NS)

Copyright © 2025 Silicon Laboratories. All rights reserved.



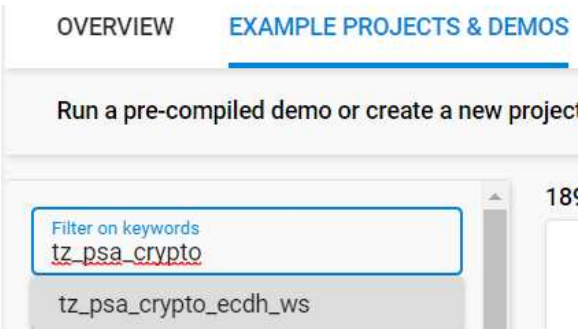
Build

The Secure project must be built first to create the Secure object library (trustzone_secure_library.o) with function entries for the Non-secure project. Both projects need to be rebuilt if any changes in the Secure project. Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to build the TrustZone platform examples.

Simplicity IDE

The following procedures are based on the TrustZone PSA Crypto ECDH example on BRD4182A Radio Board (EFR32MG22C224F512IM40).

1. Use the `tz_psa_crypto` keyword to search in EXAMPLE PROJECTS & DEMOS tab. Select the `tz_psa_crypto_ecdh_ws` example.

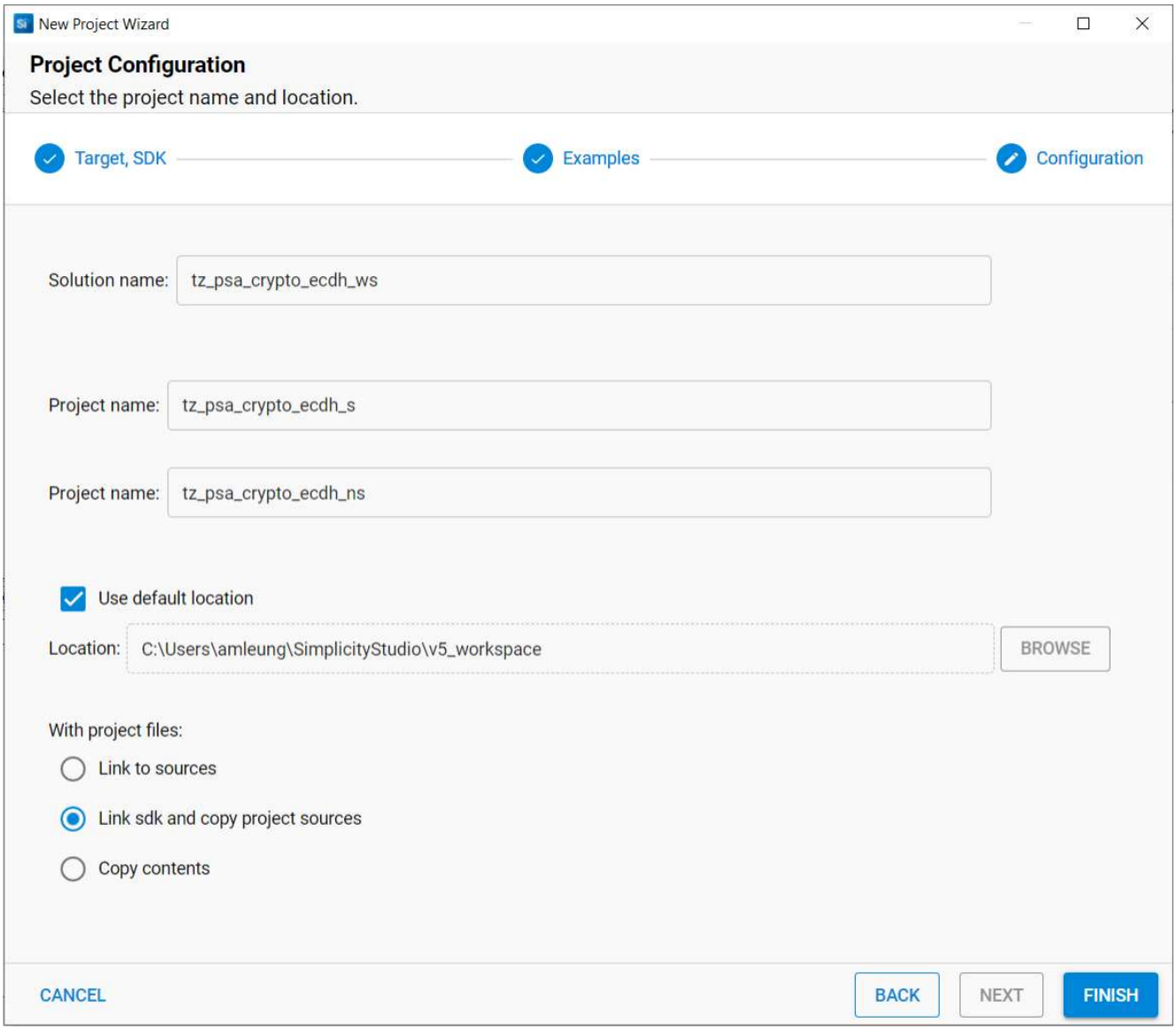


2. Click [CREATE] to generate the solution.

Copyright © 2025 Silicon Laboratories. All rights reserved.

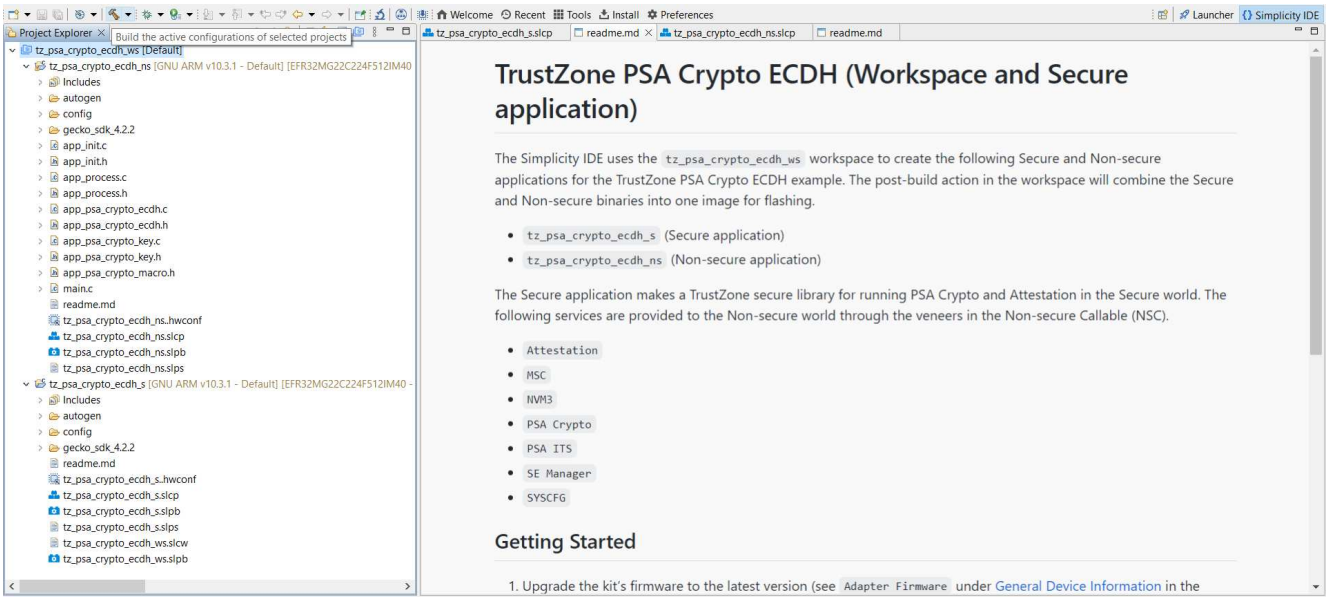


3. The Project Configuration dialog shows the Secure and Non-secure projects in the target solution. Click [FINISH] to start the creation process.

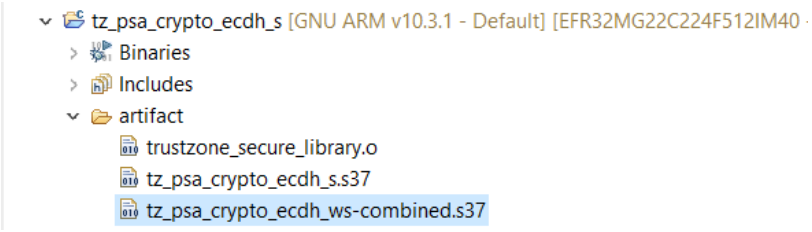


4. The Simplicity IDE perspective opens after finishing the solution creation. Click Build on the Simplicity IDE perspective toolbar to build the projects of a selected solution in order (Secure then Non-secure).

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. The post-build actions (`.slpb` files) of the Secure project, Non-secure project, and workspace will be processed in sequence if the solution is successfully built. The combined image (tz_psa_crypto_ecdh_ws-combined.s37) in the Secure project artifact folder can be used for programming the device or debugging.



6. Use Memory Editor to finalize the memory layouts of Secure and Non-secure applications and rebuild the solution to update the memory configurations.

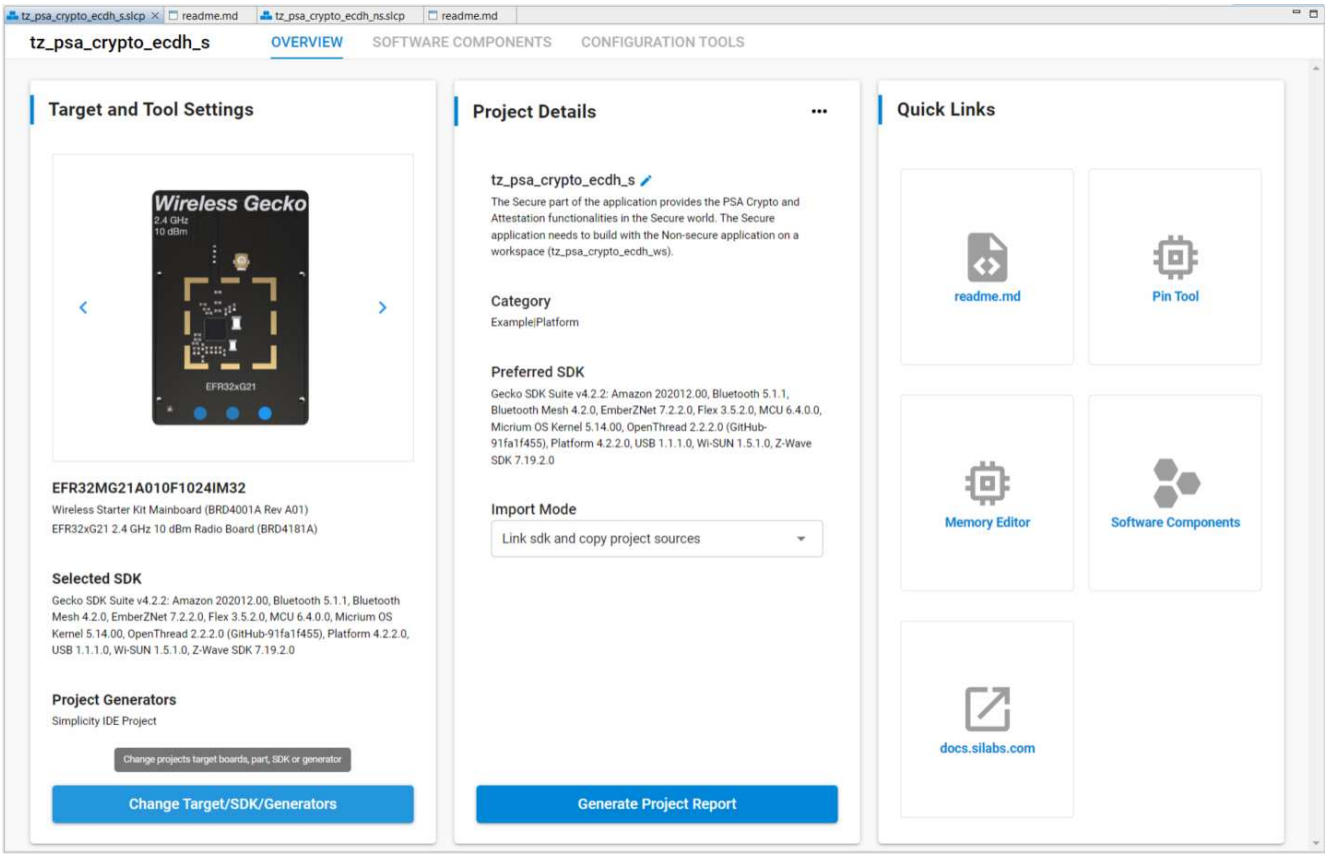
> Note: The Simplicity IDE can only apply the post-build action to a particular project if multiple Secure or Non-secure projects exist in the solution.

IAR EWARM

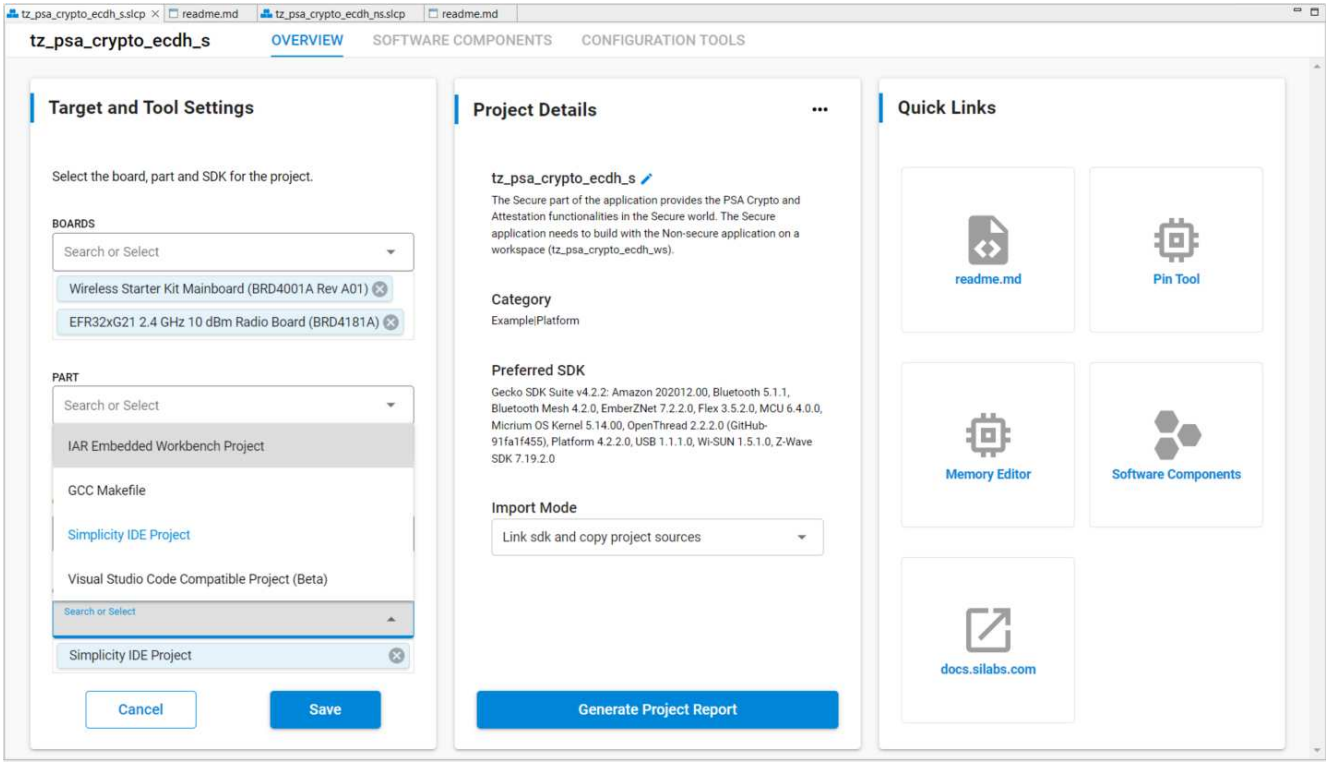
The following procedures are based on the TrustZone PSA Crypto ECDH example on BRD4181A Radio Board (EFR32MG21A010F1024IM32).

1. Follow steps 1 to 3 in TrustZone PSA Crypto ECDH to generate the solution for the `tz_psa_crypto_ws` . Select the `tz_psa_crypto_ecdh_s.slcp` file.
2. The Overview tab shows the Target and Tool Settings card on the left side. Scroll down if necessary and click [ChangeTarget/SDK/Generators].

Copyright © 2025 Silicon Laboratories. All rights reserved.

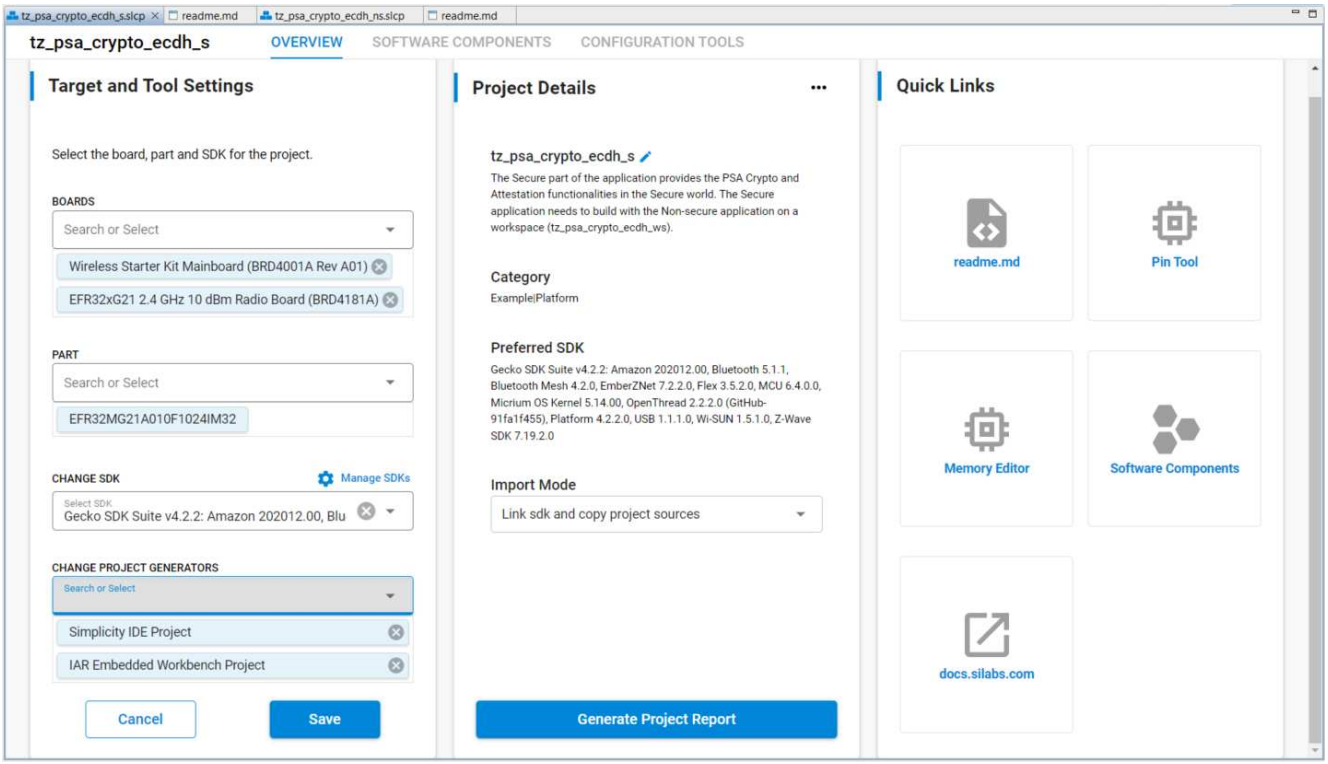


3. Drop down the CHANGE PROJECT GENERATORS list and select IAR Embedded Workbench Project.



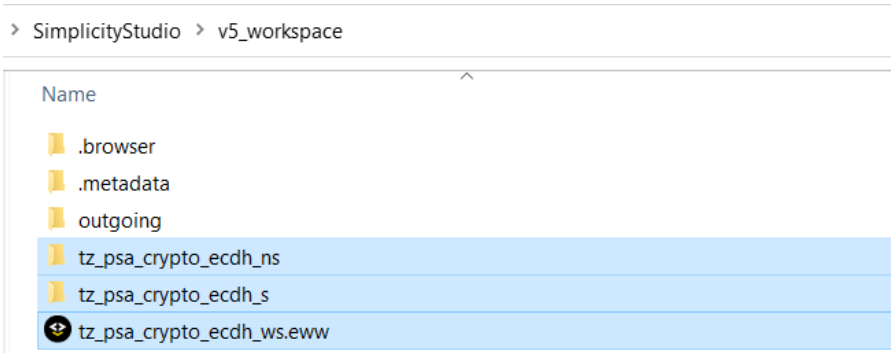
4. Click [Save] to generate an IAR Secure project (tz_psa_crypto_ecdh_s.ewp).

Copyright © 2025 Silicon Laboratories. All rights reserved.



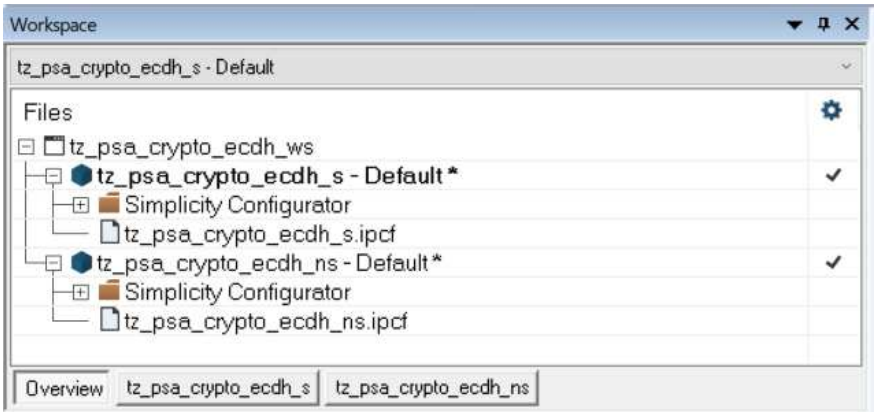
5. Select the `tz_psa_crypto_ecdh_ns.slcp` file. Repeat steps 2 to 4 to generate an IAR Non-secure* project (tz_psa_crypto_ecdh_ns.ewp). 6. Use a text editor to create an IAR tz_psa_crypto_ecdh_ws.ewwfile (shown below) to house the projects (tz_psa_crypto_ecdh_s.ewpand tz_psa_crypto_ecdh_ns.ewp) generated in steps 4 and 5. The location of the tz_psa_crypto_ecdh_ws.eww is the directory for WS_DIR.

```
<?xml version ="1.0" encoding="iso-8859-1"?>
<workspace>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_s\tz_psa_crypto_ecdh_s.ewp</path>
  </project>
  <project>
    <path>$WS_DIR$\tz_psa_crypto_ecdh_ns\tz_psa_crypto_ecdh_ns.ewp</path>
  </project>
  <batchBuild/>
</workspace>
```

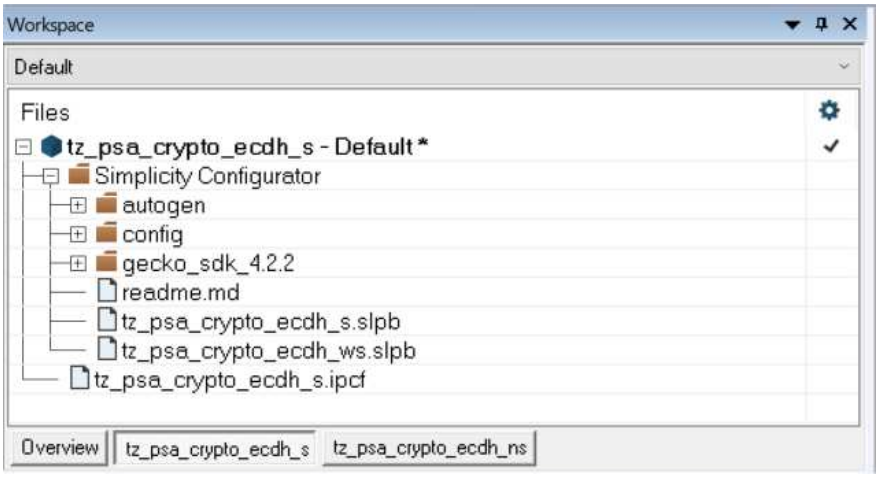


7. Double-click the tz_psa_crypto_ecdh_ws.ewwfile to open the workspace that includes Secure and Non-secure projects.

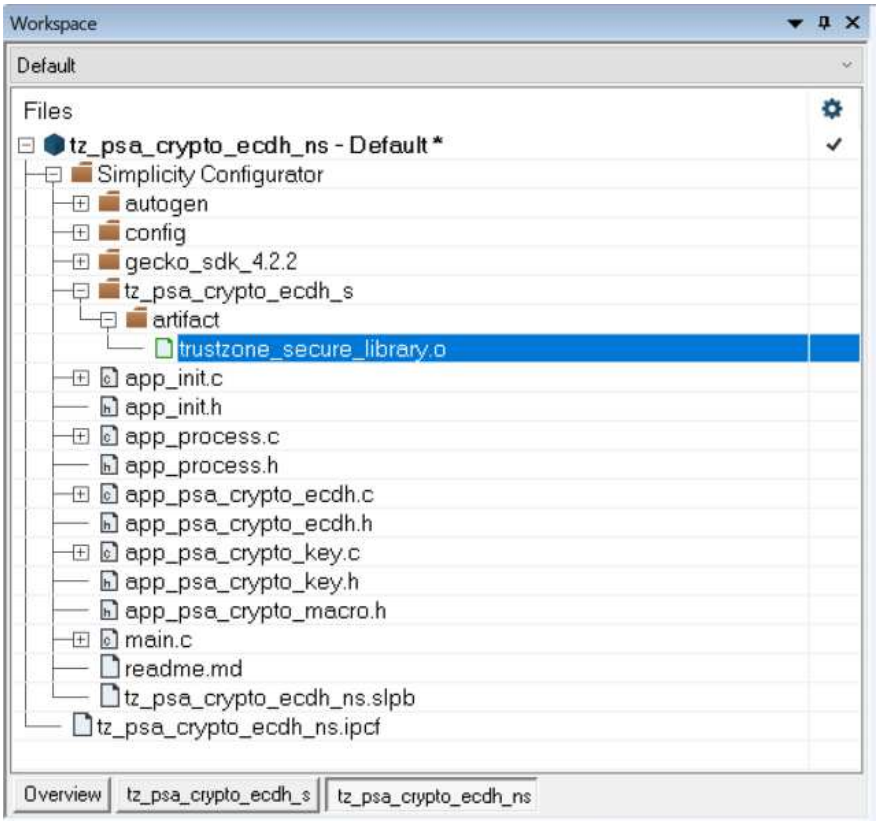
Copyright © 2025 Silicon Laboratories. All rights reserved.



8. Click the tz_psa_crypto_ecdh_s tab to open the Secure project. Click (Make) to build. It exports the Secure object library (trustzone_secure_library.o) for function entries that will be used by the Non-secure project.



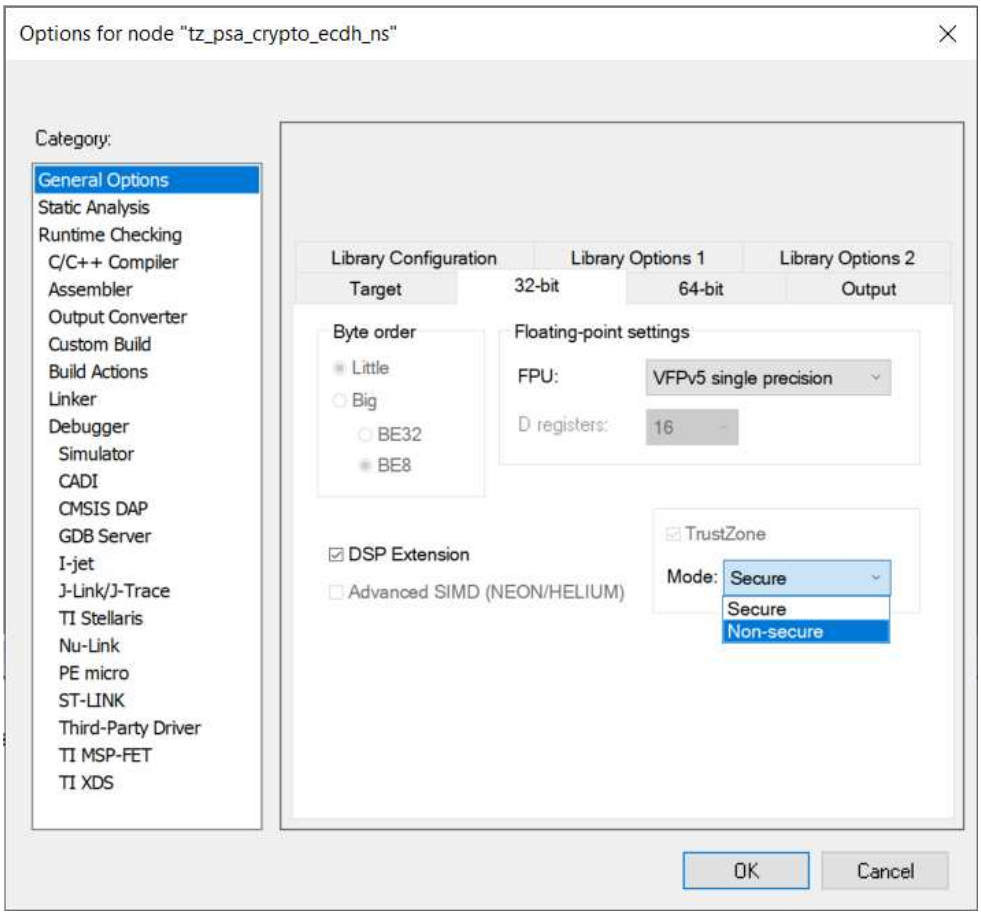
9. Click the tz_psa_crypto_ecdh_ns tab to open the Non-secure project.



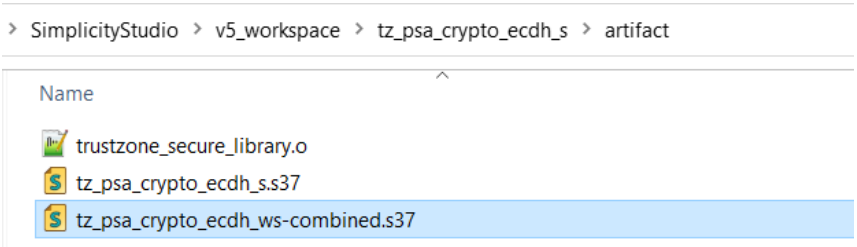
10. The `SL_TRUSTZONE_NONSECURE` defined in the Non-secure project disables the CMSE compiler option (`--cmse`) regardless of whether the Project → Options... → General

Copyright © 2025 Silicon Laboratories. All rights reserved.

Options → 32-bit → TrustZone → Mode: setting is Secure or Non-secure. So changing this configuration from Secure to Non-secure is optional. Click [OK] to exit.



11. Click (Make) to build the Non-secure project. The post-build actions of the workspace (tz_psa_crypto_ecdh_ws.slpb) will be triggered in IAR to combine the Secure and Non-secure images (tz_psa_crypto_ecdh_ws-combined.s37) to the artifact folder of `tz_psa_crypto_ecdh_s` for programming the device.



12. Use Memory Editor to finalize the memory layouts of Secure and Non-secure applications and rebuild the Secure and Non-secure projects to update the memory configurations.

> Note: The IAR EWARM can only apply the workspace post-build action to a particular project if multiple Secure or Non-secure projects exist in the workspace.

Debugging

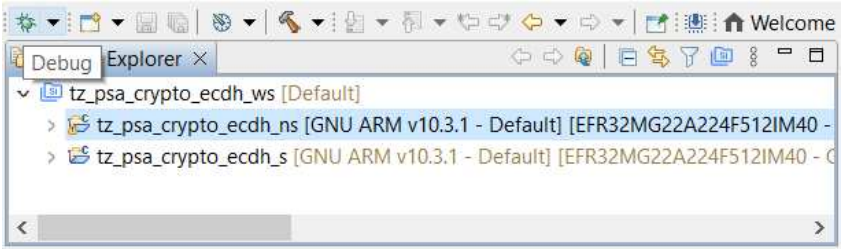
Users can use Simplicity IDE in Simplicity Studio 5 or IAR EWARM v9.20.4 to debug the TrustZone platform examples. Building the projects with Optimization Level None (-O0) is recommended for debugging.

Simplicity IDE

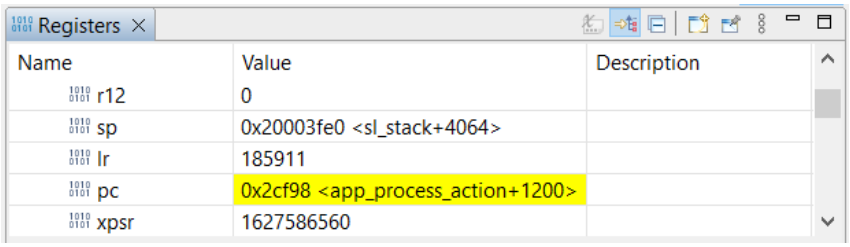
The TrustZone debugging process on Simplicity IDE is similar to the existing sample projects in Simplicity Studio.

Copyright © 2025 Silicon Laboratories. All rights reserved.

2. Flash the combined image (tz_psa_crypto_ecdh_ws-combined.s37) generated in Simplicity IDE to the device.
3. Select the Secure or Non-secure project and use the Debug icon to launch a debug session.



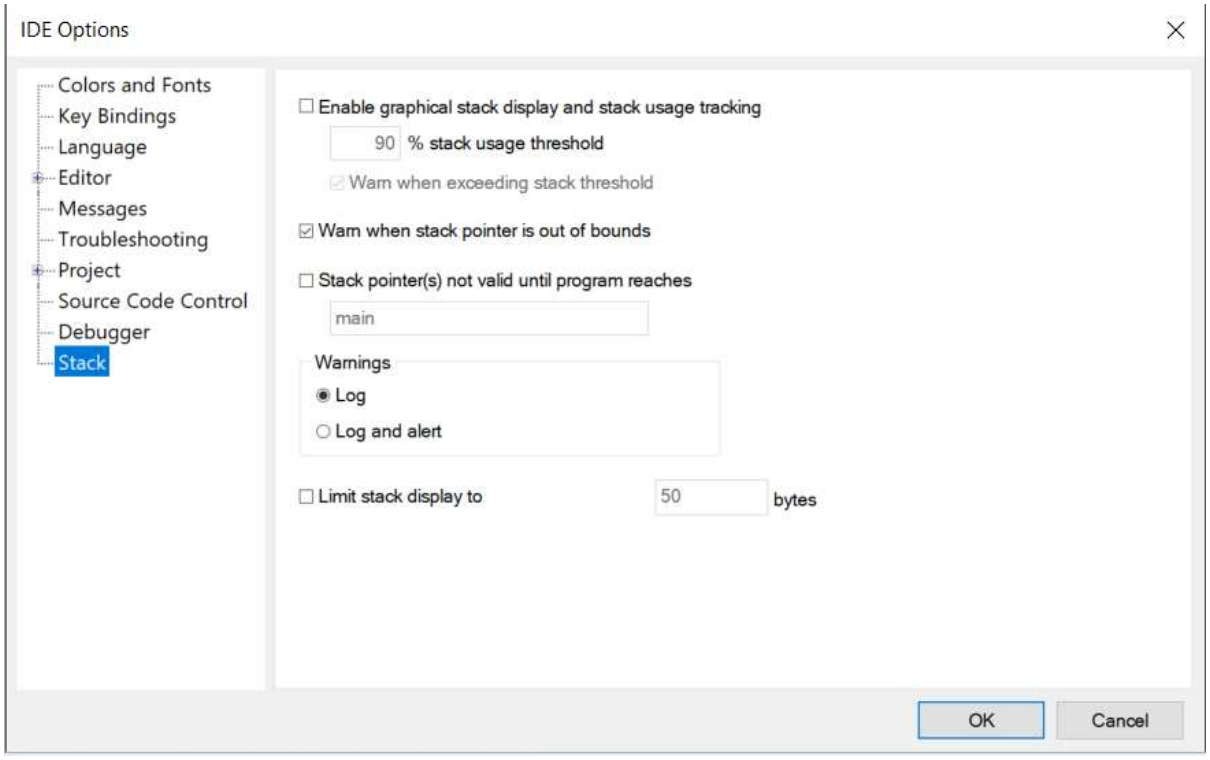
4. Follow the instructions in the Using the Debugger section in Simplicity Studio 5 User's Guide to debug the Secure or Non-secure application. 5. The debugger cannot step into the function in a Non-secure application when debugging the Secure application and vice versa. Use the Program Counter (PC in Secure or Non-secure address) in the Registers window to determine the program status.



IAR EWARM

Use the tz_psa_crypto_ecdh_ws.eww workspace created in IAR EWARM for the debugger settings. Except for a minor difference in step 3, the following steps are the same as those to set up the Secure (tz_psa_crypto_ecdh_s) and Non-secure (tz_psa_crypto_ecdh_ns) projects for debugging.

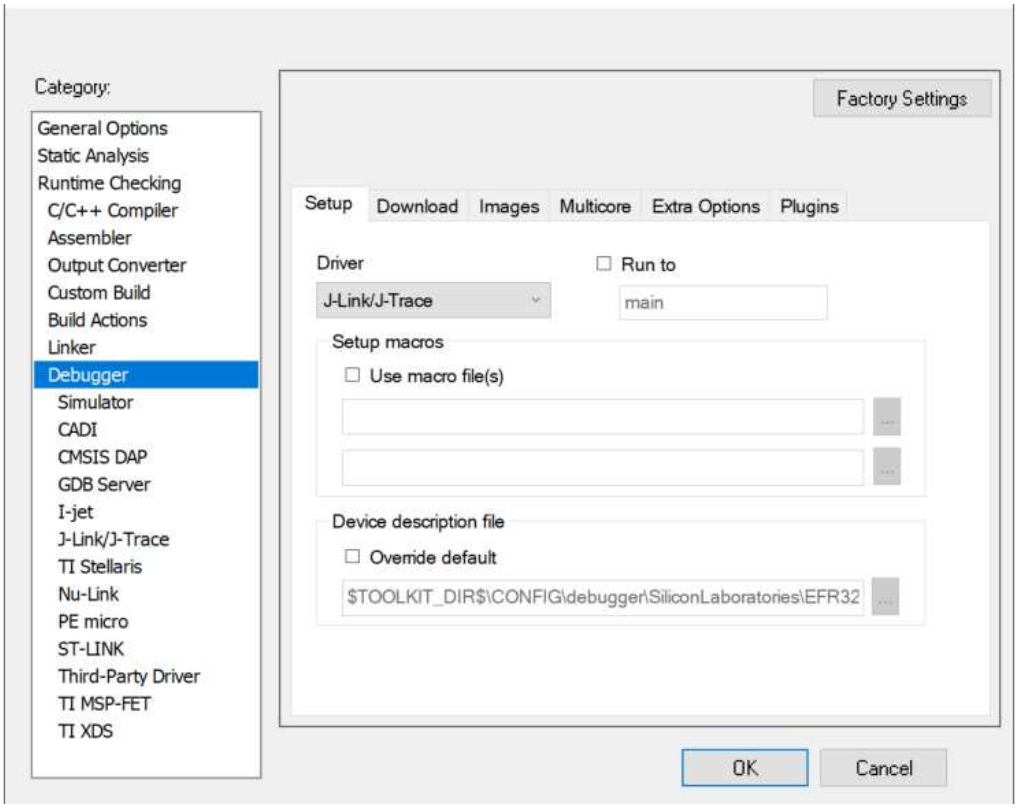
1. Select Options... in the J-Link Tools Window context menu of the Secure or Non-secure project and open the IDE Options → Stack dialog. Uncheck the Stack pointer(s) not valid until program reaches* checkbox. Click [OK] to exit.



2. Select Options... in the View Project J-Link context menu of the Secure or Non-secure project and open the window for Debugger

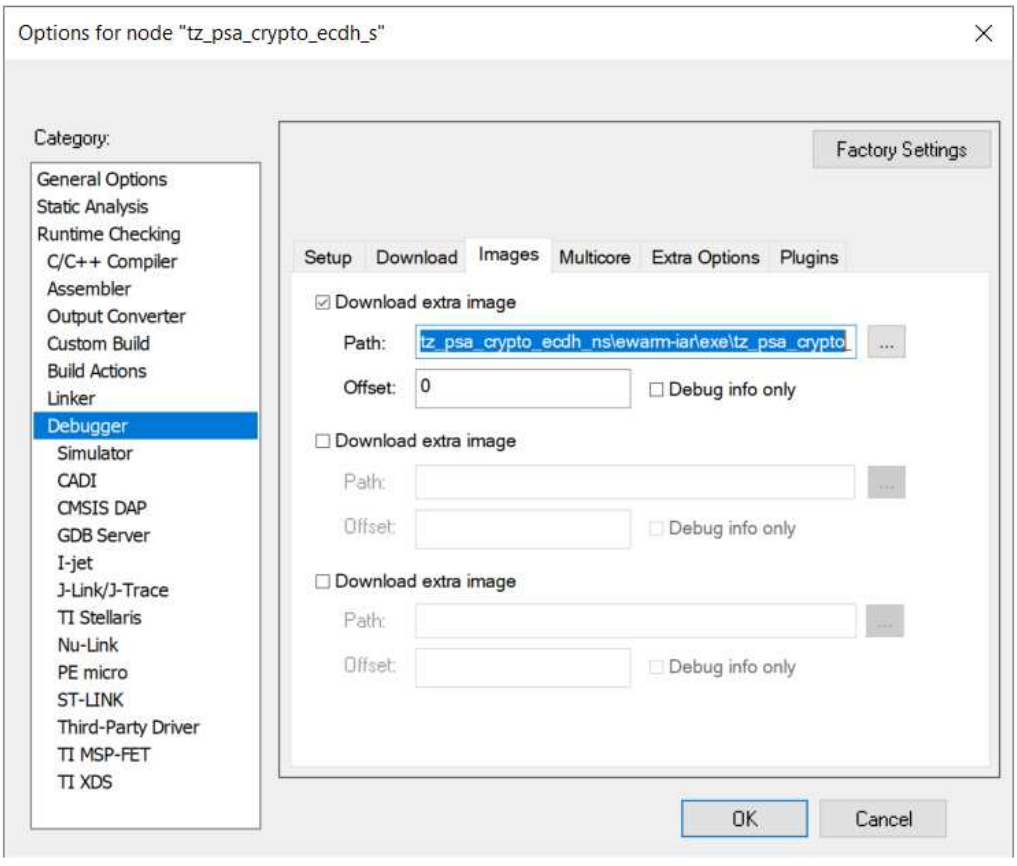
Copyright © 2025 Silicon Laboratories. All rights reserved.

options. Click the Setup tab to open a dialog, and uncheck the Run to → main checkbox. Click the Images tab to set up another option.



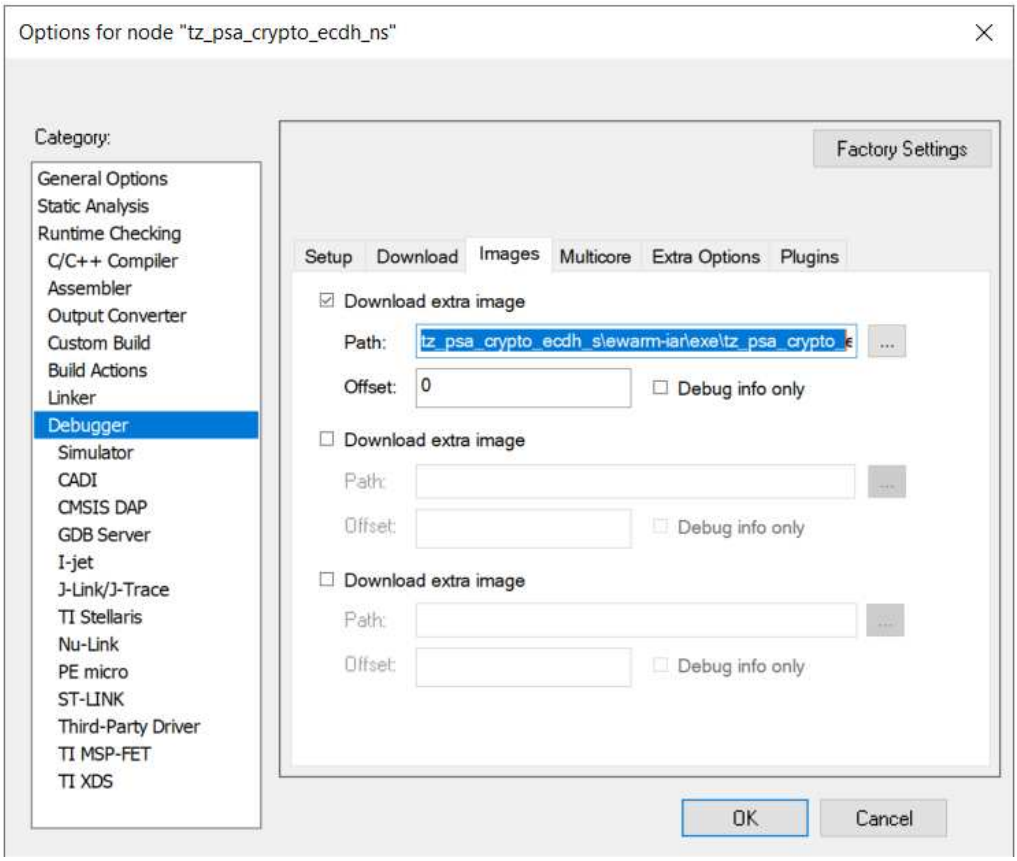
3. Check the ownload extra image option. Enter the location of the .out file to Path: with Offset: set to 0. All project relative paths are resolved from the directory location of the tz_psa_crypto_ecdh_ws.eww workspace file.

Location of Non-secure .out file for Secure project: tz_psa_crypto_ecdh_ns\ewarm-iar\exe\tz_psa_crypto_ecdh_ns.out

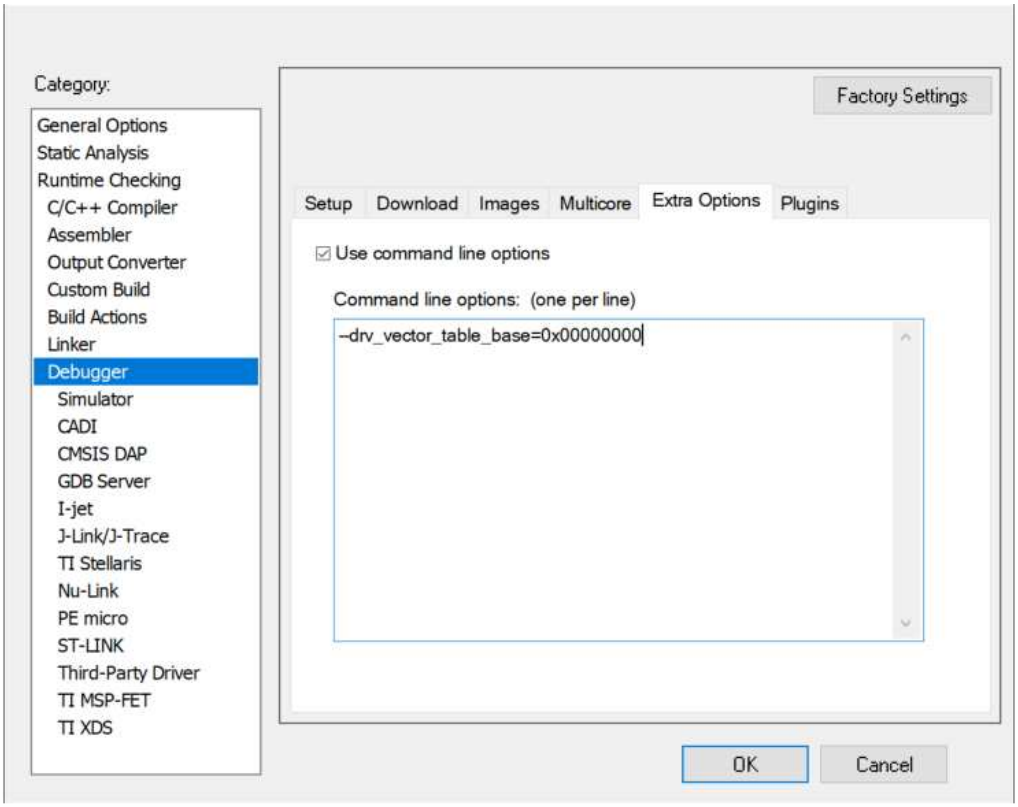


Copyright © 2025 Silicon Laboratories. All rights reserved.

Location of Secure .out file for Non-secure project: tz_psa_crypto_ecdh_s\ewarm-iar\exe\tz_psa_crypto_ecdh_s.out



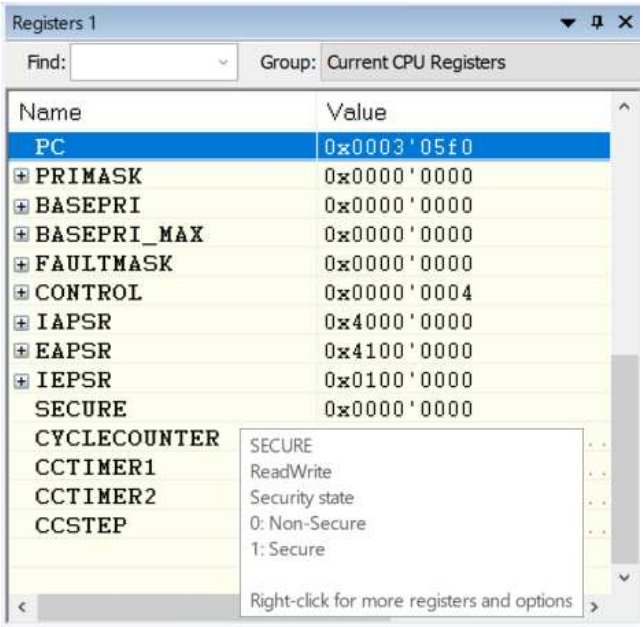
4. Click the Extra Options tab to set up another option. 5. Check the Use command line options. Enter --drv_vector_table_base=0x00000000 to Command line options: (one per line) window. Click [OK] to exit.



6. Finish the debug settings in Secure and Non-secure projects, and click (Download and Debug) in the Secure or Non-secure project to download the Secure and Non-secure images for debugging (assume both projects had successfully built before). Click (Go) to start running

Copyright © 2025 Silicon Laboratories. All rights reserved.

the code in a Secure or Non-secure project. 7. The debugger will automatically switch between Secure and Non-secure projects when stepping into a function or hitting a breakpoint in a Secure or Non-secure project. Use the Program Counter (PC in Secure or Non-secure address) or SECURE (0 or 1) in the Registers window to determine the program status.



8. Click (Stop Debugging) to end the debug session.

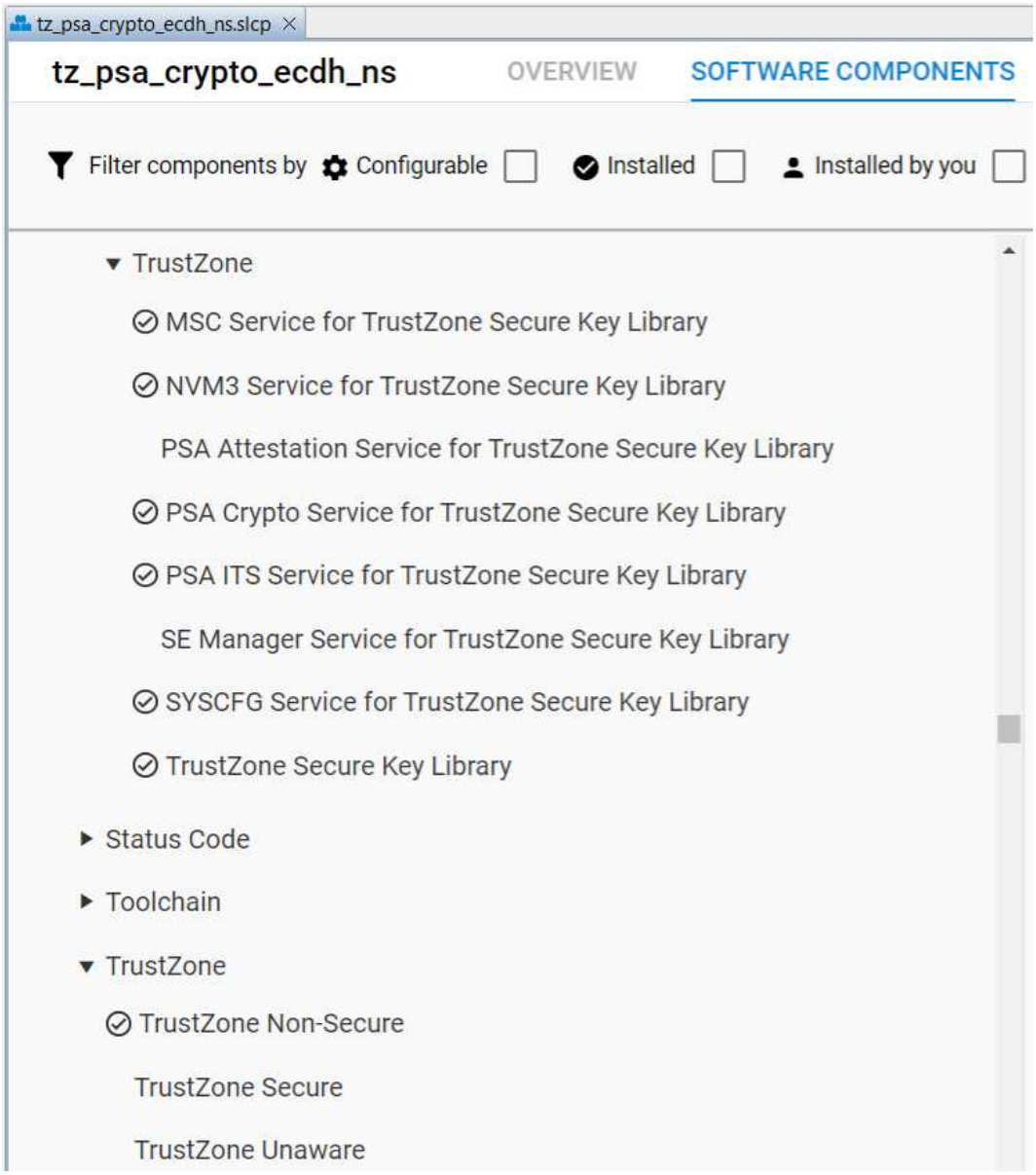
Benchmark

The TrustZone implementation will affect the memory footprint and performance of cryptographic operations. The following comparisons are based on the TrustZone PSA Crypto ECDH example on BRD4182A Radio Board (EFR32MG22C224F512IM40) with SE firmware v1.2.14.

Memory Footprint

The memory footprint of a TrustZone project depends on which services (software components in the figure below) provided by the Secure Library are used in the Non-secure application (`tz_psa_crypto_ecdh_ns` project).

Copyright © 2025 Silicon Laboratories. All rights reserved.



The following tables compare the memory footprint of the TrustZone-unaware (`Platform - PSA Crypto ECDH`) and TrustZone-aware projects (`tz_psa_crypto_ecdh_ws`) based on the following conditions.

- The `tz_psa_crypto_ecdh_ns` reuses the source code from the `Platform - PSA Crypto ECDH` example without any changes.
- The total size in `tz_psa_crypto_ecdh_ns` does not consider the 4 kB alignment on the Secure and Non-secure flash and RAM. The 4 kB alignment requirement will increase the actual usage of flash and RAM.
- All source code is compiled with Optimize for size (-Os) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.

Table: Flash Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	64688 B	-	-	64688 B
tz_psa_crypto_ecdh_ws	79172 B	288 B	29264 B	108724 B

> Note: The NSC is part of the Secure code, and the total size does not include the flash for NVM3 storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Table: RAM Size Comparison

Platform Example	Secure	NSC	Non-secure	Total
Platform - PSA Crypto ECDH	3784 B	-	-	3764 B
tz_psa_crypto_ecdh_ws	2156 B	-	1200 B	3356 B

> Note: The total size does not include the RAM for the stack and heap. The Secure and Non-secure applications have their independent stack and heap.

PSA Crypto Performance

The following sections compare the PSA Crypto performance of the TrustZone-unaware (`Platform - PSA Crypto ECDH`) and TrustZone-aware projects (`tz_psa_crypto_ecdh_ws`) based on the following conditions.

- The `tz_psa_crypto_ecdh_ns` reuses the source code from the `Platform - PSA Crypto ECDH` example without any changes.
- All source code is compiled with Optimize most (-O3) in Simplicity IDE (GNU ARM v10.3.1) of Simplicity Studio 5.
- Use ECC curve `SECP256R1` on volatile and persistent keys.
- The EFR32MG22C224 runs at 38 MHz HFRCODPLL.

Volatile key ECDH operation on `Platform - PSA Crypto ECDH`

```
  . ECDH Client
  + Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 2928 time: 77 us)
  + Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 2960 time: 77 us)
  + Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 332134 time:
8740 us)
  + Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 336860 time:
8864 us)
```

Volatile key ECDH operation on `tz_psa_crypto_ecdh_ws`

```
  . ECDH Client
  + Creating a SECP256R1 (256-bit) VOLATILE PLAIN client key... PSA_SUCCESS (cycles: 5047 time: 132 us)
  + Creating a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 5067 time: 133 us)
  + Exporting a public key of a SECP256R1 (256-bit) VOLATILE PLAIN server key... PSA_SUCCESS (cycles: 333956 time:
8788 us)
  + Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338470 time:
8907 us)
```

Persistent key ECDH operation on `Platform - PSA Crypto ECDH`

```
  . ECDH Client
  + Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 27489 time: 723 us)
  + Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 27587 time: 725 us)
  + Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 332949 time:
8761 us)
  + Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 337803 time:
8889 us)
```

Persistent key ECDH operation on `tz_psa_crypto_ecdh_ws`

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
  . ECDH Client
  + Creating a SECP256R1 (256-bit) PERSISTENT PLAIN client key... PSA_SUCCESS (cycles: 46998 time: 1236 us)
  + Creating a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 45962 time: 1209 us)
  + Exporting a public key of a SECP256R1 (256-bit) PERSISTENT PLAIN server key... PSA_SUCCESS (cycles: 334127 time: 8792 us)
  + Computing client shared secret with a SECP256R1 (256-bit) server public key... PSA_SUCCESS (cycles: 338321 time: 8903 us)
```

The overheads on the TrustZone-aware project (`tz_psa_crypto_ecdh_ws`) are due to the following operations of Secure Library implementation.

- Packages the list of input arguments in the appropriate format before calling into the NSC function.
- Switches from a Non-secure to a Secure state.
- Validates all input arguments before calling into the function in SPE.
- Encrypts PSA ITS if using a persistent key.
- Returns to a Non-secure state.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Protocol-Specific Information

These pages provide information about using a Gecko Bootloader in the context of a specific protocol. Additional protocol-specific content is available through the docs.silabs.com homepage.

- Silicon Labs Bluetooth: Includes detailed information on using the Gecko Bootloader with Silicon Labs Bluetooth applications.
- Silicon Labs Connect: Provides information on using the Silicon Labs Gecko Bootloader with Connect.
- Zigbee EmberZNet: Includes detailed information on using the Silicon Labs Gecko Bootloader with EmberZNet.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

NOTE: This section replaces *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*. Further updates to this application note will be provided here.

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with Silicon Labs Bluetooth applications for Gecko SDK (GSDK) 4.1.0 and higher or Simplicity SDK (SiSDK) 2024.6.0 and higher. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrading image files, refer to Bootloader Fundamentals.

Key Points

- Gecko Bootloader overview
- Using Gecko Bootloader for BGAPI UART DFU
- Using Gecko Bootloader for Bluetooth OTA upgrade
- Using Gecko Bootloader to update firmware from the user application
- Delta DFU
- Post-Build Editor

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of bootloader functions, from device initialization to firmware upgrades. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). Currently two formats are actively supported, GBL3 (or just GBL) for Series 2 or older and GBL4 for Series 3. These images are produced with the file extension ".gbl". Additional information on the GBL file format is provided in Bootloader Fundamentals.

The Gecko Bootloader has a two-stage design, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the software component configuration. Software components can be enabled and configured through the Simplicity Studio IDE.

This document describes how to configure and use the Gecko Bootloader for device firmware upgrades over UART, and for OTA (over-the-air) upgrades using Bluetooth.

The Gecko Bootloader does not come bundled into the application example projects (although some precompiled demos do contain it). Therefore, you must compile and load the bootloader separately from the application image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BGAPI UART Device Firmware Upgrade

BGAPI UART Device Firmware Upgrade (DFU)

This is the firmware upgrade used in Network Co-Processor (NCP) Bluetooth applications. For more information on NCP applications, see Using the Silicon Labs Bluetooth Stack v3.x and Higher in Network Co-Processor Mode.

In the BGAPI UART DFU implementation a GBL image containing the new firmware is written to target device using UART as the physical interface.

UART DFU Options

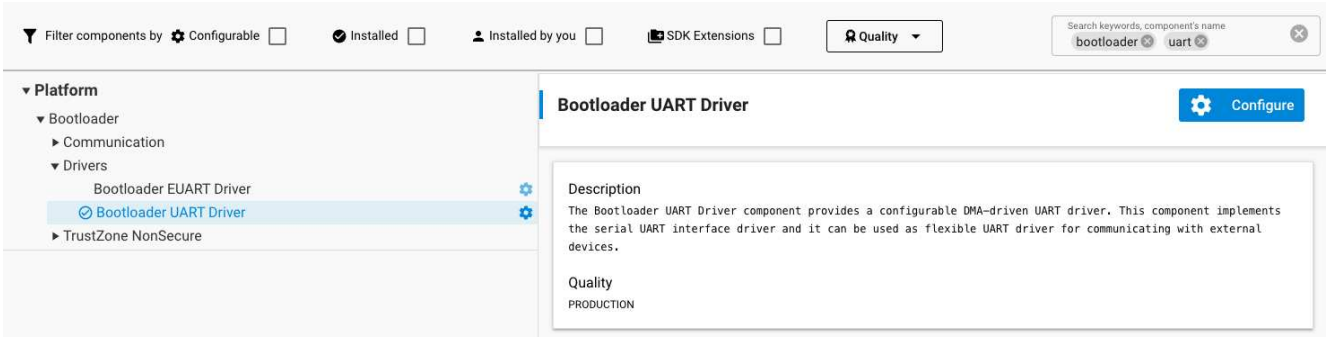
The target device must be programmed with the Gecko Bootloader sample project Bootloader - NCP BGAPI UART DFU. Gecko Bootloader is configured automatically for the selected radio board. The BGAPI UART DFU bootloader is a standalone bootloader, so no storage area needs to be configured. During UART DFU upgrade the bootloader writes the new firmware image directly on top of the old firmware image and therefore no temporary download area is needed.

GPIO Settings

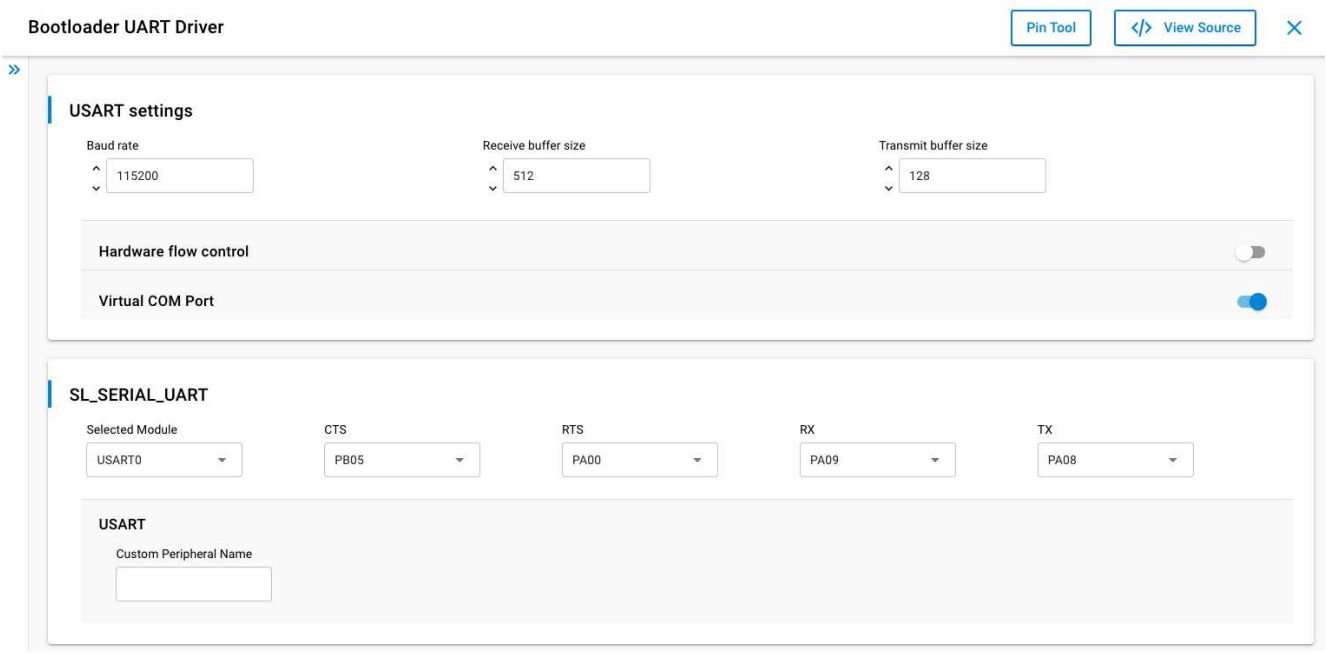
The default settings are suitable for testing with a WSTK (Wireless Starter Kit). These settings can be easily changed by through the Software Components tab. Select the Bootloader UART Driver component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

Flow control settings of the radio board and the WSTK must match. The WSTKs flow control can be configured through the admin console. More information on the admin console can be found in the user guide for the respective development kit. To configure the flow control through the admin console:

- In Simplicity Studios Debug Adapters view, right click on the connected device.
- Select Launch Console.
- In the Admin tab, type *serial vcom config handshake disable/enable*, depending on if you want to disable or enable flow control.



Copyright © 2025 Silicon Laboratories. All rights reserved.



UART DFU Process

The basic steps involved in the UART DFU, using NCP, are as follows:

1. Boot the target device into DFU mode by calling the bootloader API `bootloader_rebootAndInstall()`. In NCP-mode, this can be achieved by calling `sl_bt_user_reset_to_dfu()`. When using Series 1 devices, reset into DFU is achieved with `sl_bt_system_reset(1)`. Alternatively, if you have GPIO activation enabled in the bootloader, press the bootloader activation pin while the device is reset.
2. Wait for the `sl_bt_evt_dfu_boot` event.
3. Send the command `sl_bt_dfu_flash_set_address(address)` to start the firmware upgrade.
4. Send the entire contents of the GBL upgrade image with `sl_bt_dfu_flash_upload(data)`.
5. After sending all data, the host sends the command `sl_bt_dfu_flash_upload_finish()`.
6. To finalize the upgrade, the host resets the target device into normal mode with `sl_bt_system_reboot()`. If using Gecko SDK 4.4.x and older, the device is reset with `sl_bt_system_reset(0)`.

A detailed description of the DFU-related BGAPI commands is found in the Bluetooth Software API Reference Manual.

At the beginning of the upgrade, the NCP host uses the command `sl_bt_dfu_flash_set_address` to define the start address. The start address shall always be set as zero. During the data upload (step 4 above) the target device calculates the flash offset automatically. The host does not need to explicitly set any write offset.

The UART DFU procedure may fail if the update image is either corrupted or data upload is interrupted for some reason. Failure due to either of these conditions is detected by the CRC check performed by the UART DFU bootloader before jumping into the main program. In this case, a `dfu_boot_failure` event is sent by the stack. The returned reason codes align with the `sl_status` codes, as shown in the platform documentation.

Creating Upgrade Images for the Bluetooth NCP Application

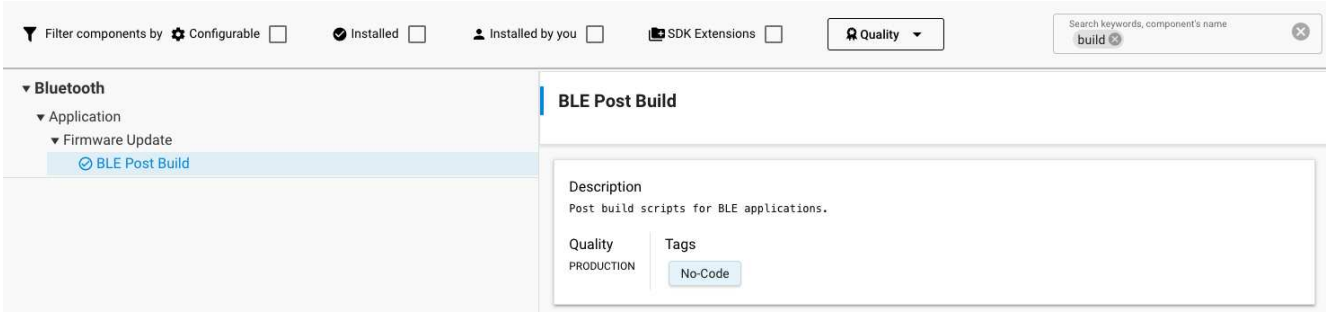
Building a C-based NCP project in Simplicity Studio does not generate the UART DFU upgrade images (GBL files) automatically. The GBL files need to be created separately by using the Post-Build Editor or by running a script located in the application projects root folder. Before running the script, the application must be compiled.

Three scripts are provided in the SDK examples:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- create_bl_files.bat (for Windows)
- create_bl_files.sh (for Linux / Mac)
- create_bl_files.py (multiplatform)

The GBL files can be generated by invoking the script from the project directory. As of now, all three are available in the latest releases. In the future releases, the Python version will enjoy exclusive support and the others are going to be deprecated. If they are not present in the project, the BLE Post Build component must be installed to make them available. On Unix platforms, the scripts might have to be re-saved with "LF" end of line sequences instead of "CRLF" to avoid interpretation issues.



You need to define two environment variables, `PATH_SCMD` and `PATH_GCCARM` before running the script, as shown in the following table.

Variable Name	Example Variable Values
PATH_SCMD	C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander /Applications/Simplicity Studio 5.app/Contents/Eclipse/developer/adapter_packs/commander ~/SimplicityStudio_v5/developer/adapter_packs/commander
PATH_GCCARM	C:\SiliconLabs\SimplicityStudio\v5\developer\toolchains\gnu_arm\12.2.rel1_2023.7 /Applications/Simplicity Studio 5.app/Contents/Eclipse/developer/toolchains/gnu_arm/12.2.rel1_2023.7 ~/SimplicityStudio_v5/developer/toolchains/gnu_arm/12.2.rel1_2023.7

Running the create_bl_files script creates multiple GBL files in a subfolder named output_gbl. The file named full.gbl is the upgrade image used for UART DFU. The other files are related to OTA upgrades, and they can be ignored.

If signing and/or encryption keys (named app-sign-key.pem, app-encrypt-key.txt) are present in the same directory, then the script also creates secure variants of the GBL files. More information on creating secure firmware for DFU can be found in our training document: Secure Firmware Upgrade using OTA.

UART DFU Host Example

The UART DFU host example is a C program that is located under the Bluetooth SDK examples in the following directory (the exact path depends on the installed SDK version):

```
~\SimplicityStudio\SDKs\<simplicity_sdk|gecko_sdk>\app\bluetooth\example_host\bt_host_uart_dfu
```

In Windows this program can be built using MinGW. (It is recommended that you use the MSYS2 development toolchain, available at https://www.msys2.org/. Follow the instructions found in that location on installing GCC.) In Linux or Mac, the program can be built using the GCC toolchain.

The project is built by running `make` (or `mingw32-make`) in the project root directory. After a successful build, an executable is created in the subfolder named exe. The executable filename is bt_host_uart_dfu(.exe).

Before running the example, you need to check the COM port number or TCP/IP address associated with your NCP target. For more details, see Using the Silicon Labs Bluetooth Stack v3.x and Higher in Network Co-Processor Mode.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The bt_host_uart_dfu program requires three command line arguments:

- Serial port (-u) or TCP/IP address (-t) – TCP port is not needed
- Baud rate (-b) – in case of Serial port
- Name of the (full) GBL file

Furthermore, the device must be flashed with the BGAPI UART DFU Bootloader and an NCP-application.

Example usage and expected output in v4.0 or higher:

```
./bt_host_uart_dfu -u /dev/tty.usbmodem0004402253201 -b 115200 ./full.gbl
[I] NCP host initialised.
[I] Reset NCP target in bootloader mode...
[I] DFU booted: v0x03000000
[I] Pressing Crtl+C aborts the update process.
[I] WARNING! If the update process is aborted, the device will stay in bootloader mode.

259648/259648 (100%)
[I] DFU finished successfully. Resetting the device.
```

The number of bytes uploaded in one DFU flash upload command is configurable. The UART DFU host example included in the SDK uses a 48-byte payload. The maximum usable payload length is 128 bytes. The maximum number of bytes sent in one command is specified using a C preprocessor directive named MAX_DFU_PACKET. The value of MAX_DFU_PACKET must be divisible by four.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bluetooth OTA Upgrade

Bluetooth OTA Upgrade

This is the firmware upgrade method used in SoC-mode Bluetooth applications. A GBL file containing new firmware is sent to a target device through a Bluetooth connection. The firmware upgrade image can be stored to an empty flash area and applied later by the user application, or immediately overwrite the original application using a component called AppLoader. The table below shows the differences between AppLoader-based and Application-based OTA.

Feature	AppLoader-based OTA	Application-based OTA
Implementation	Integrated into the bootloader	Implemented in the user application
Supported Devices	Series 1 & 2	Series 2 & 3
Storage slot required	No, the old firmware is directly overwritten.	Yes, the new firmware must be stored in flash before overwriting.
Supported PHY	1M	1M, 2M, Coded
Connection Encryption	Not supported	Supported

Currently, it is recommended to use the Application-based OTA as it enables improved security and customizability. (While GBL signing and encryption is supported for both, the AppLoader approach makes for a less secure connection.) Furthermore, AppLoader is not supported in Series 3 devices. The two available options are described in more detail in the sections, *Application-based OTA* below and AppLoader (Not recommended for new development). An example of creating the GBL files and using the Application-based OTA is shown in the OTA DFU Example section.

Application-based OTA

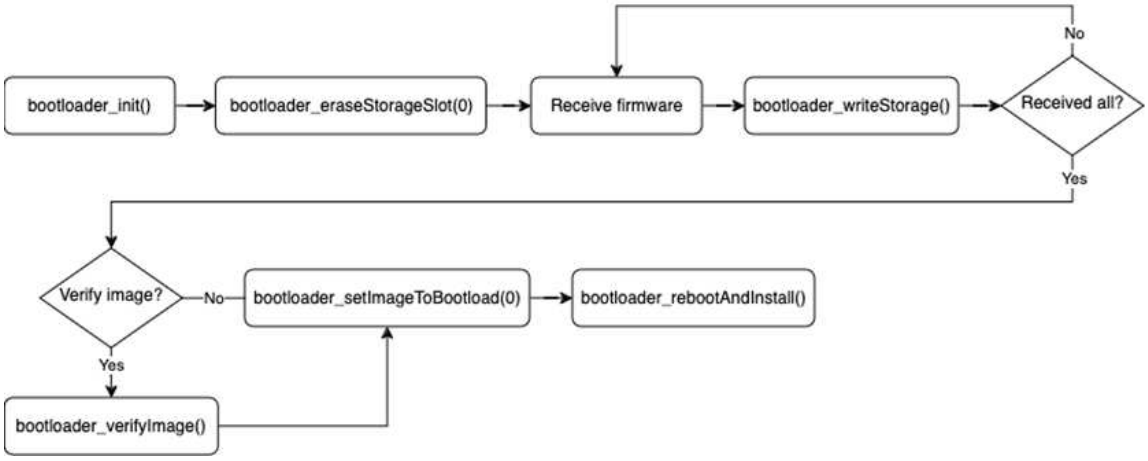
Application-based OTA is completely implemented in the user-application. This makes it possible to use a custom GATT service instead of the Silicon Labs OTA service used in the AppLoader. Furthermore, in case of UART DFU updates, the application can be designed to support other protocols and interfaces than BGAPI and UART. The user application can also be designed to support both OTA and UART DFU updates if needed.

To use this update mechanism, any application bootloader using internal or external storage may be used. At least one storage area must be defined, and the area must be large enough to fit the full GBL file, while not overlapping with the user application. These parameters can be controlled by configuring the Bootloader Storage Slot Setup (bootloader_storage_slots) component included in the bootloader project.

The basic steps for the update process are described below:

1. Application initializes the Gecko bootloader by calling `bootloader_init()`.
2. The download area for the new firmware is erased by calling `bootloader_eraseStorageSlot(0)`.
3. The update image (full GBL file) is received either over-the-air or through some physical interface like UART.
4. The application writes the received bytes to the download area by calling `bootloader_writeStorage()`.
5. (optional) Application can verify the integrity of the received GBL file by calling `bootloader_verifyImage()`.
6. Before rebooting, call `bootloader_setImageToBootload(0)` to specify the slot ID where new image is stored.
7. Reboot and instruct Gecko bootloader to perform the update by calling `bootloader_rebootAndInstall()`.

Copyright © 2025 Silicon Laboratories. All rights reserved.



It is assumed here that only one download area is configured and therefore the slot index in the above function calls is set to 0. The general firmware upgrade sequence is explained in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

Note that the erase procedure in step 2 above takes several seconds to complete. If the new image is downloaded over a Bluetooth connection, the supervision timeout must be set long enough to avoid connection drops. Alternatively, the download area can be erased in advance, before the Bluetooth connection is opened. A third alternative is to erase the download area one flash page at a time during the writing progresses. This can be done using `bootloader_eraseRawStorage()`.

Enabling the Gecko Bootloader API

To call the required Gecko Bootloader functions from the user application, the following interface sources must be included in the project:

btl_interface.c (common interface)

btl_interface_storage.c (interface to storage functionality)

These source files and their corresponding header files are copied to SDK sample projects by default and can be found in the SDK under:

```
<simplicity_sdk|gecko_sdk>\platform\bootloader\api\
```

When using the functions in the application, the correct header files can be included with:

```
#include "btl_interface.h"
#include "btl_interface_storage.h"
```

> Note: If UARTDRV (and DMADRV) is used inside the application (i.e. for logging via UART), that might cause LDMA channel conflict with the Bootloader storage access, thus preventing a successful OTA. To avoid this, both the bootloader and the application need to be configured appropriately by reducing the "Number of available channels" in the DMADRV Configuration UI of the application, allowing the Bootloader to utilize a DMA channel above it.

AppLoader (Not recommended for new development)

The AppLoader, introduced in Series 1, can be viewed as a small extension that manages everything related to OTA DFU. It contains a minimal version of the Bluetooth stack including only features that are necessary to perform OTA updates. Any Bluetooth features that are not necessary to support OTA updates are disabled in AppLoader to minimize the flash footprint.

Copyright © 2025 Silicon Laboratories. All rights reserved.

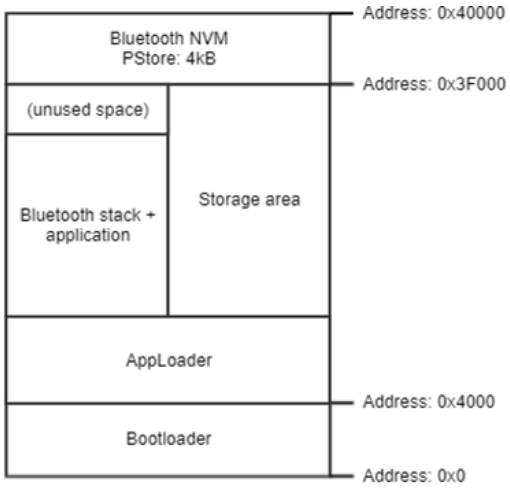
The AppLoader features and limitations are summarized below:

- Enables OTA updating of user application.
- Only one Bluetooth connection is supported, GATT server role only.
- Bluetooth parameters used during firmware upgrade, such as advertisement and connection interval, can't be changed.
- Encryption and other security features such as bonding are not supported.

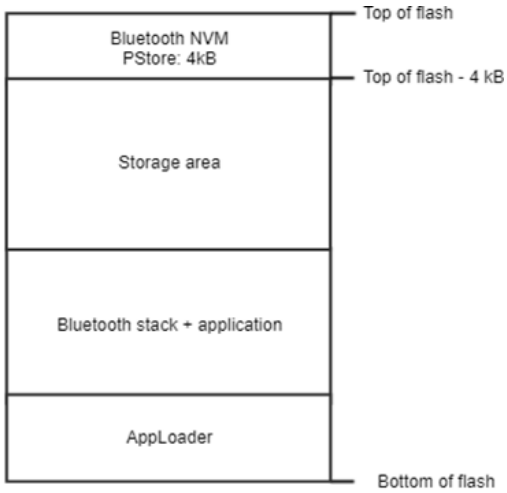
The AppLoader is placed before the user application in flash. The default linker script provided in the SDK places the user application so that it starts at the flash sector following the AppLoader. The user application contains a full-featured version of the Bluetooth stack, and it can run independently of the AppLoader. If in-place OTA update is not needed, the AppLoader can be removed completely to free up flash for other use (code space or data storage).

Series 1 vs Series 2

From GSDK v4.1.0 and higher, there is a clear distinction between the integration of Apploader for Series 1 and Series 2 devices. For Series 1 devices, the Apploader is a separate application from both the main and the bootloader.



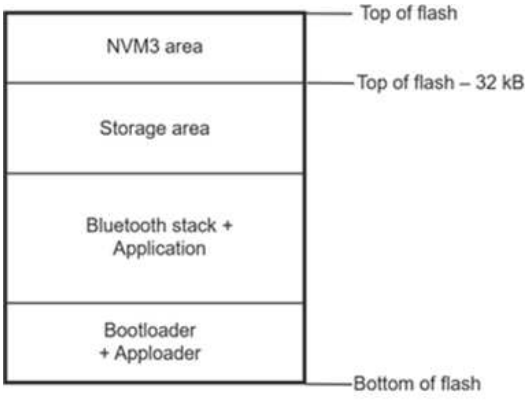
Main flash layout using Bluetooth in-place OTA DFU bootloader on EFR32xG1



Example main flash layout using bootloader with internal flash storage on EFR32xG12

For Series 2 devices, the AppLoader is added as a part of the bootloader. This results in the following for Series 2 devices:

- The AppLoader cannot be upgraded or flashed by itself.
- The AppLoader is upgraded by upgrading the bootloader.
- The OTA advertising data cannot be changed.
- The size of the combined bootloader and AppLoader is much larger than that of a regular bootloader, and it does not fit into the regular bootloader area.



Example main flash layout using bootloader with internal flash storage on Series 2 devices

Copyright © 2025 Silicon Laboratories. All rights reserved.

AppLoader OTA Process

Most of the OTA functionality is handled autonomously by the AppLoader. The minimum requirement for the user application is to trigger a reboot into DFU mode. After the upload is complete, AppLoader will reboot the device back into normal mode.

Reboot into DFU mode can be triggered in a variety of ways. It is up to the application developer to decide which is most applicable. Most of the example applications provided in the Bluetooth SDK already have OTA support built into the code. In these examples, the DFU mode is triggered through the Silicon Labs OTA service that is included as part of the application's GATT database. An example of a user defined trigger can be found in Triggering Reboot into DFU Mode from the User Application.

AppLoader supports two types of updates:

- Partial update: only the user application is updated
- Full update: both AppLoader and the user application are updated (1)

> Note: For Series 1 devices using older than v4.1.0 versions of GSDK, we recommend a combined upgrade of Bootloader and AppLoader to avoid 0x84 (NOT_SUPPORTED) parsing errors and a separate application upgrade step to avoid the 0x87 (APPLICATION_OVERLAP_APPLOADER) error. For detailed error explanations, see OTA Error Codes.

For Series 2 devices, the same steps can be followed, but the AppLoader is updated as part of the bootloader, after which the application must be updated. For this, the bootloader must be converted into .gbl format by using Simplicity Commander:

```
commander gbl create bootloader-apploader.gbl --bootloader bootloader-apploader.s37
```

The partial update process using AppLoader consists of the following steps:

1. OTA client connects to target device.
2. Client requests target device to reboot into DFU mode.
3. After reboot, client connects again.
4. During the 2nd connection, target device is running AppLoader (not the user application).
5. New firmware image is uploaded to the target.
6. AppLoader copies the new application on top of the existing application.
7. When upload is finished and connection closed, AppLoader reboots back to normal mode.
8. Update complete.

With partial update, it is possible to update the Bluetooth stack and user application. AppLoader is not modified during partial update.

Full update enables updating both the AppLoader and the user application. A full update is always recommended when moving from one SDK version to another. The size of AppLoader can vary depending on the SDK version, so this may prevent a partial OTA update if the new application image overlaps with the old AppLoader version.

Full update is done in two steps. Updating the AppLoader always erases the user application and therefore the AppLoader update must be followed by an application update.

The first phase of full update updates the AppLoader, and it consists of following steps for Series 1 devices:

1. OTA client connects to the target device.
2. The Client requests target device to reboot into DFU mode.
3. After reboot, the client reconnects.
4. During the 2nd connection, the target device is running AppLoader (not the user application).
5. New AppLoader image is uploaded to the target.
6. AppLoader copies the image into the download area (specified in Gecko bootloader configuration).
7. When the upload is finished and the connection closed, AppLoader reboots and requests Gecko Bootloader to install the downloaded image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

8. Gecko Bootloader updates AppLoader using the downloaded image and reboots.
9. After reboot, the new AppLoader is started.

At the end of the AppLoader update, the device does not contain a valid user application and therefore AppLoader will remain in DFU mode. To complete the update, a new user application is uploaded following the same sequence of operations that were described for the partial update.

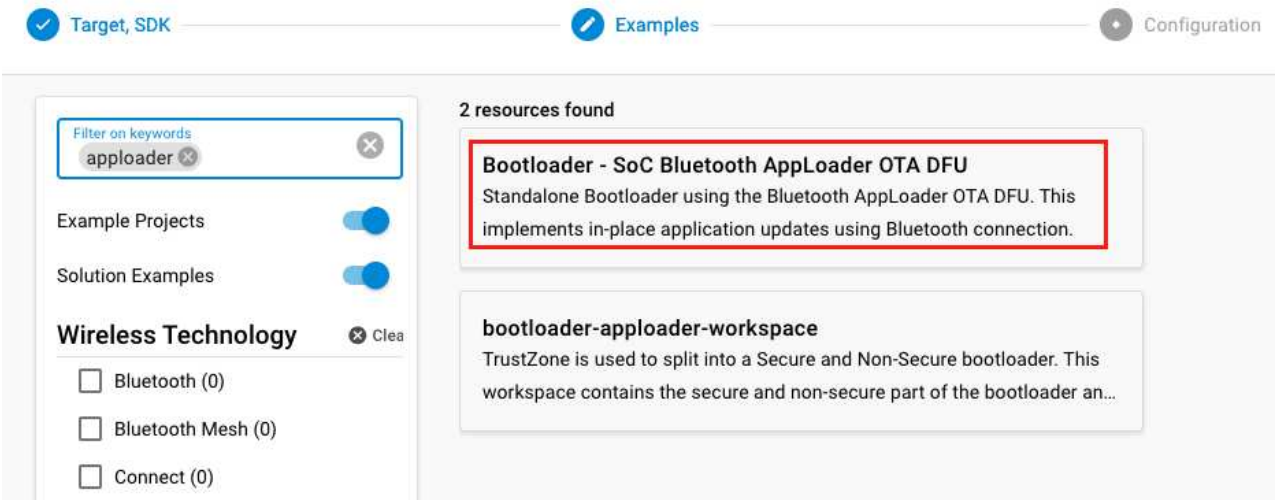
Configuring the AppLoader

In order to use OTA for firmware upgrades, the user must first configure the bootloader (only for Series 2) and the application to support the AppLoader. Once this is done, the user is ready to generate the .gbl file, as described in Creating the GBL Files.

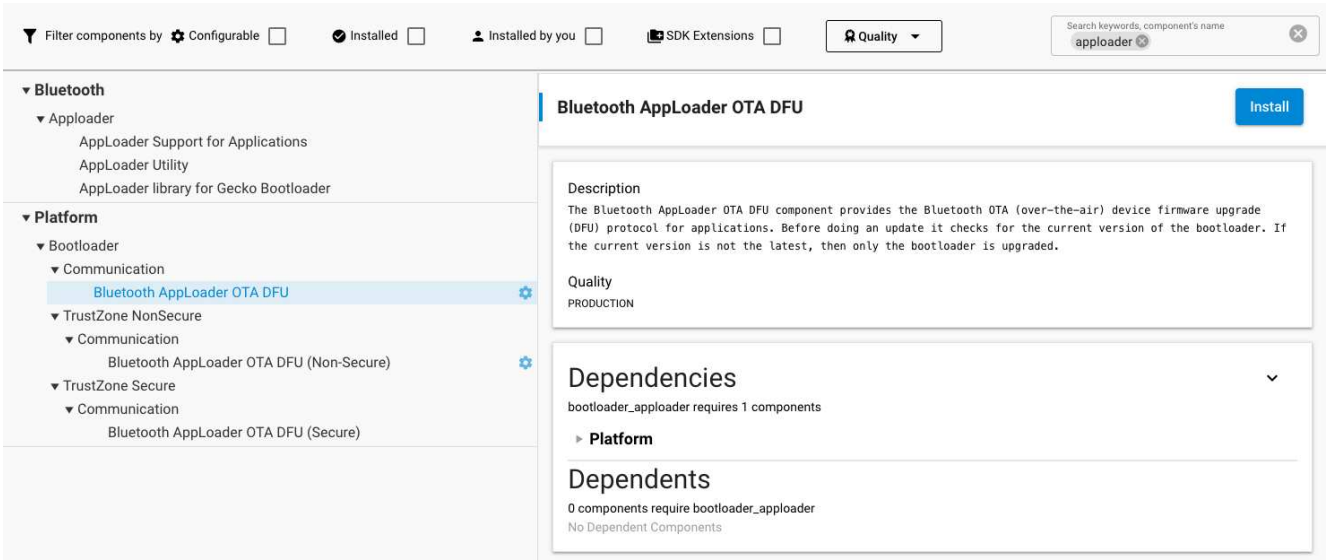
Bootloader

This step is only required for Series 2. If you are using a Series 1 device, you can move on to the application.

Configuring the bootloader for OTA can be done in two ways. The most straightforward method is to include the sample Bootloader project, SoC Bluetooth AppLoader OTA DFU, where all necessary configurations are already set.



Alternatively, you can add the Bluetooth AppLoader OTA DFU (bootloader_apploader) software component to your already existing bootloader. After this component is added, the Base address of bootloader upgrade image must be updated in the Bootloader Core component. This is done to prevent the new bootloader from overwriting itself during the OTA update. Any appropriate value can be used as long as the base address is larger than the size of the bootloader itself. The recommended value is 0x18000.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Application

In your application project, only the In-Place OTA DFU (in_place_ota_dfu) component needs to be added. If you are using a sample project, this component might be included by default.



The In-Place OTA DFU (in_place_ota_dfu) component requires the following components, which are automatically included:

- AppLoader Support for Applications (apploader) which on:
- Series 1 devices: includes the AppLoader binary as an additional prebuilt library.
- Series 2 devices: moves the application start address to give space for an AppLoader OTA DFU Bootloader. It also requires a Gecko Bootloader with an AppLoader OTA DFU plugin to be present on the device.
- AppLoader Utility (apploader_util), which provides utility functions related to OTA DFU, such as a unified API for resetting the device to DFU mode.



Triggering Reboot into DFU Mode from the User Application

The minimum functional requirement to enable OTA in the user application is to implement a ‘hook’ that allows the device to be rebooted into DFU mode.

1. The code checks if the OTA control characteristic was written, and if so, sets `boot_to_dfu` to true. After this, the connection is closed.
2. The device is rebooted into DFU mode with `sl_bt_system_reset(2)` or `sl_apploader_util_reset_to_ota_dfu()`. Parameter value 2 indicates that the device is to be rebooted into OTA DFU mode. The rest of the OTA upgrade is managed by AppLoader and no further actions are needed from the user application.
3. This code is now incorporated into the `sl_bt_in_place_ota_dfu.c` source file, part of the In-Place OTA DFU (in_place_ota_dfu) component.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
// -------------------------------
// This event indicates that a remote GATT client is attempting to write
// a value of a user type attribute in to the local GATT database.
case sl_bt_evt_gatt_server_user_write_request_id:
  // If user-type OTA Control Characteristic was written, boot the device
  // into Device Firmware Upgrade (DFU) mode. Written value is ignored.
  if (evt->data.evt_gatt_server_user_write_request.characteristic
      == gattdb_ota_control) {
    // Always check security status before the transfer.
    if (sl_bt_in_place_ota_dfu_security_status(addr, conn_hdl, bond_hdl)
        != SL_BT_IN_PLACE_OTA_DFU_SECURITY_ACCEPT) {
      // Security requirements not fulfilled. Reject the request.
      attr_status = SL_STATUS_BT_ATT_WRITE_REQUEST_REJECTED;
    } else {
      // Boot into DFU mode.
      boot_to_dfu = true;
    }

    // Send response to user write request.
    sc = sl_bt_gatt_server_send_user_write_response(
      evt->data.evt_gatt_server_user_write_request.connection,
      gattdb_ota_control,
      (uint8_t)attr_status);
    app_assert_status(sc);

    // Start delay timer before closing connection.
    // Forward connection ID to the timer callback.
    sc = app_timer_start(&connection_close_delay,
                         delay_additional_ms,
                         delay_timer_cb,
                         (void *)((uint32_t) evt->data.evt_gatt_server_user_write_request.connection),
                         false);
    app_assert_status(sc);
  }
  break;
```

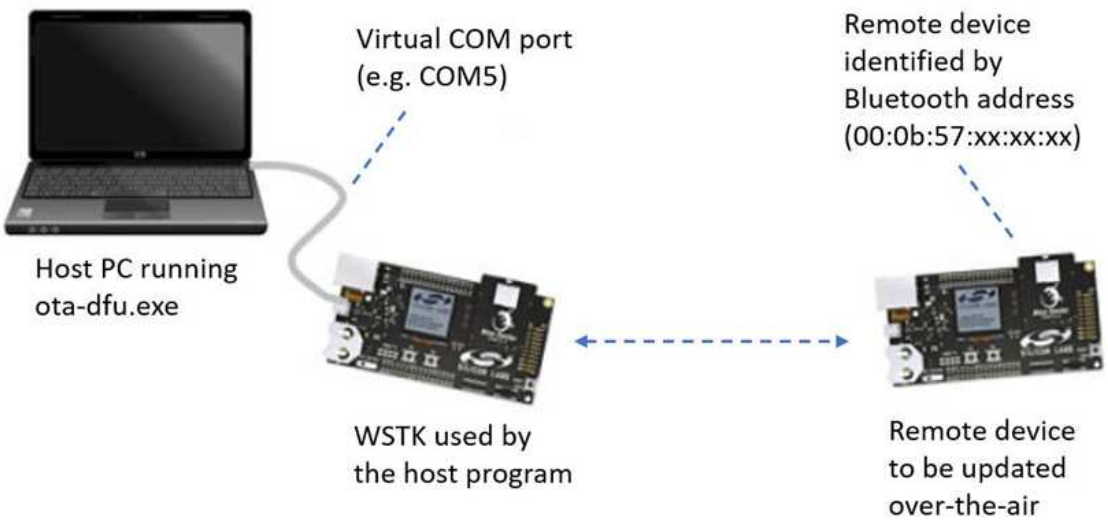
```
// -------------------------------
// This event indicates that a connection was closed.
case sl_bt_evt_connection_closed_id:
  // Check if need to boot to OTA DFU mode.
  if (boot_to_dfu) {
    sl_apploader_util_reset_to_ota_dfu();
  }
  break;
```

OTA DFU Example

The Bluetooth SDK includes an OTA host reference implementation that can be used for testing OTA. The example is written in C and uses a WSTK in Network Co-Processor (NCP) mode. The OTA host application itself runs on the host computer. For more information on the NCP mode of operation, see AN1259: Using the Silicon Labs Bluetooth Stack v3.x and Higher in Network Co-Processor Mode.

The target device to be upgraded over-the-air is identified by its Bluetooth address.

Copyright © 2025 Silicon Laboratories. All rights reserved.



In this example, we will use the application-based OTA, but AppLoader can also be used.

Preparing the SoC Application

In this example, we use the Bluetooth – SoC Empty sample application, together with one of the internal-storage bootloaders.



As the In-Place OTA DFU (in_place_ota_dfu) is not used, it can be removed from the project. The source of Application OTA DFU (app_ota_dfu) can be found at:

```
<simplicity_sdk|gecko_sdk>\app\bluetooth\common\app_ota_dfu
```

Creating the GBL files

Once the Application-based OTA has been added, the project can be compiled. After compiling, the GBL files must be created separately by using the Post-Build Editor or by running one of the following scripts from the project root folder:

Three scripts are provided in the SDK examples:

- create_bl_files.bat (for Windows)
- create_bl_files.sh (for Linux / Mac)
- create_bl_files.py (multiplatform)

The GBL files can be generated by invoking the script from the project directory. As of now, all three are available in the latest releases, but in future releases the Python version will enjoy exclusive support with the others being deprecated. If they are not present in the project, the BLE Post Build component must be installed to make them available. On Unix platforms, the scripts might have to be resaved with "LF" end of line sequences, instead of "CRLF" to avoid interpretation issues.

Before running the scripts, two environment variables must be defined, `PATH_SCMD` and `PATH_GCCARM`, as seen in the table below:

Copyright © 2025 Silicon Laboratories. All rights reserved.

Variable Name	Example Variable Values
PATH_SCMD	C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander /Applications/Simplicity Studio 5.app/Contents/Eclipse/developer/adapter_packs/commander ~/SimplicityStudio_v5/developer/adapter_packs/commander
PATH_GCCARM	C:\SiliconLabs\SimplicityStudio\v5\developer\toolchains\gnu_arm\12.2.rel1_2023.7 /Applications/Simplicity Studio 5.app/Contents/Eclipse/developer/toolchains/gnu_arm/12.2.rel1_2023.7 ~/SimplicityStudio_v5/developer/toolchains/gnu_arm/12.2.rel1_2023.7

Running the create_bl_files script creates GBL files in a subfolder named output_gbl. The application.gbl file is the relevant file for OTA.

If signing and/or encryption keys (named app-sign-key.pem, app-encrypt-key.txt) are present in the project directory then the script also creates secure variants of the GBL files. More information on creating secure firmware for DFU can be found in our training document: Secure Firmware Upgrade using OTA.

If properly named bootloader binaries (such as bootloader-second-stage.s37 or bootloader-apploader.s37) are present in the application project folder, the create_bl_files script will incorporate those in the final GBL file.

Preparing the NCP Application

The development kit that is used on the host side should be programmed with firmware that is suitable for NCP mode. The Bluetooth SDK includes an example project called “Bluetooth NCP” that can be used for this purpose.

The WSTK features an on-board USB-to-UART converter so that the board will be seen as a virtual Serial port by the host computer. The WSTK also has an Ethernet controller which can be used for the NCP connection.

Preparing the Host Application

The OTA host example is found in the following directory under the SDK installation tree (the exact path depends on the installed SDK version), but for example:

{@Manual intervention TextBox Need consider what to do}

The project folder contains a makefile that allows the program to be built using for example MinGW. (It is recommended that you use the MSYS2 development toolchain, available at https://www.msys2.org/. Follow the instructions there on how to install GCC.) Alternatively, use make in any Unix environment. An executable file is created in subfolder named exe. The executable filename is bt_host_ota_dfu(.exe).

Running OTA with the NCP Host Example

The OTA host program expects the following command-line arguments:

- Serial port (-u) or TCP/IP address (-t) – TCP port is not needed
- Baud rate (-b) – in case of Serial port
- Name of the GBL file to be uploaded into target device (-g)
- Bluetooth address of the target device (-a)
- (optional) Force write without response (possible values 0 / 1, default is 0) (-w)

A full OTA upgrade is done in two parts and it requires two separate GBL files, one for the bootloader and another for the user application. If the bootloader and application has been combined into one image, as described in Creating the GBL Files or Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher, the upgrade can be done in one part.

For upgrading the bootloader and application separately, the host application must be invoked twice:

```
./bt_host_ota_dfu -u /dev/tty.usbmodem0004402253201 -b 115200 -g ./ bootloader-second_stage.gbl -a 00:0B:57:0B:49:23
./bt_host_ota_dfu -u /dev/tty.usbmodem0004402253201 -b 115200 -g ./application.gbl -a 00:0B:57:0B:49:23
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

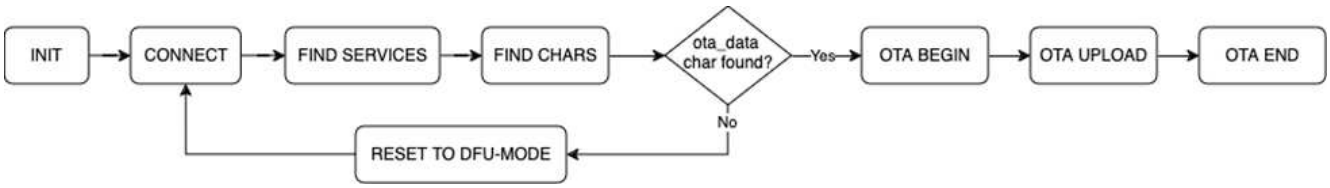
> Note: The GBL files can also be uploaded using the Simplicity Connect mobile application, as described in Using Simplicity Connect for OTA DFU.

If the application alone is going to be upgraded, then the host program is run once, with the application.gbl file passed as parameter.

OTA Host Example Internal Operation

The OTA host example is implemented as a state machine. The key steps in the OTA sequence are summarized below. Note that the program execution is independent of the type of upgrade image that is used. The program simply uploads one GBL file into the target device. It is up to the user to invoke the program either once or twice, depending on the upgrade type (partial OTA or full OTA).

The following diagram illustrates the state transitions in the OTA host example program in a slightly simplified form.



INIT: The program checks the total size of the GBL file that is passed as a command-line parameter. The GBL file content is not parsed. It is enough to know the file size so that the entire content can be uploaded to target device.

CONNECT: The program tries to open a connection to the target device whose Bluetooth address is given as a command line parameter. The host program does not scan for devices. If the target device is not advertising, then the connection open attempt causes the program to be blocked.

FIND SERVICES: After a connection has been established, the program performs a service discovery. In this case only the OTA service is of interest, and therefore the program performs discovery of services with that specific UUID using the API call `sl_bt_gatt_discover_primary_services_by_uuid`. More information on the OTA service can be found in Silicon Labs OTA GATT Service.

FIND CHARACTERISTICS: After the service has been found, the characteristics of the OTA service are queried using API call `sl_bt_gatt_discover_characteristics`. The handle value for the ota_control needs to be discovered in order to proceed with the OTA procedure.

RESET TO DFU: If the target is not already in the DFU-mode, the host program requests reboot into DFU mode by writing value 0x00 to the ota_control characteristic. The execution then jumps back to the CONNECT state.

OTA BEGIN: If both ota_data and ota_control characteristic handles have been detected, the host program initiates OTA by writing value `0x00` to the `ota_control` characteristic. This does not cause reboot or any other side effects because the target device is already in DFU mode.

OTA UPLOAD: This is where the GBL file is uploaded to the target device. The whole content of the GBL file is uploaded to the target device, by performing multiple write operations into the ota_data characteristic. The host program uses the write-without-response transfer type to optimize throughput. Note that even if the write-without-response operations are not acknowledged at the application level, error checking (and retransmission when needed) at the lower protocol layers ensures that all packets are delivered reliably to the target device.

OTA END: Once the OTA upload is completed, the host program ends the OTA procedure by writing value `0x03` to the `ota_control` characteristic. Finally, the program terminates.

Some error cases have been omitted from the state diagram for simplicity. For example, the program exits with an error code if the OTA service is not found when performing service discovery or if the `ota_control` characteristic is not discovered in FIND CHARACTERISTICS state.

Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note: When the target device reboots into DFU mode, the host program must perform full service and characteristic discovery again. It is not possible to store the `ota_control` and `ota_data` characteristic handles in memory and use those cached values during the second connection. This is because the target device has two GATT databases that are independent of each other: one that is used by the application in normal mode and the other that is used by AppLoader in OTA DFU mode. While both of these GATT databases might include the Silicon Labs OTA service, the characteristic handles are likely to have different values. Therefore any kind of GATT caching cannot be used.

OTA GATT Database and Generic Attribute Service

When booted into DFU mode, the AppLoader uses a GATT database that is different than the normal GATT used by the application. The OTA DFU GATT database used by AppLoader contains following services:

- Generic Attribute (UUID 0x1801)
- Generic Access (UUID 0x1800)
- Silicon Labs OTA service (UUID 0x1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0)

The Bluetooth specification requires that, if GATT-based services can change in the lifetime of the device, then the Generic Attribute Service (UUID 0x1801) and the Service Changed characteristic (UUID 0x2A05) shall exist in the GATT database. For details, please see Bluetooth Core specification, Version 6.0, Vol. 3, Part G, 7 DEFINED GENERIC ATTRIBUTE PROFILE SERVICE.

The Generic Attribute service is automatically included in the AppLoader GATT database used during OTA. To avoid any interoperability issues due to GATT caching, it is strongly recommended that the application GATT database used in normal mode also enables this service. Generic Attribute service is enabled by default in the SDK example applications.

> Note: AppLoader does not generate a service changed indication when rebooting to DFU mode or rebooting back to normal mode.

Automatic service changed indication requires that the client is bonded and has enabled the indication for this characteristic. AppLoader does not support bonding and therefore the service changed indication is not generated.

The Generic Attribute Service can also be explicitly defined in the application's GATT database using the same XML notation that is used for other services. The Generic Attribute service must be the first service in the list, to ensure it is aligned with the Generic Attribute Service that is used during OTA. The Bluetooth specification requires that the attribute handle of the Service Changed characteristic shall not change and therefore this service must be first on the list (the same as in the OTA GATT database).

More details on the Generic Attribute Service can be found on the Bluetooth SIG website:

https://www.bluetooth.com/specifications/specs/

Note also that AppLoader does not support the GATT caching enhancements that were introduced in the Bluetooth Core Specification 5.1 and Silicon Labs Bluetooth SDK 2.11.1.

Silicon Labs OTA GATT Service

The following XML representation defines the Silicon Labs OTA service. It is a custom service using 128-bit UUID values. The service content and the UUID values are fixed and must not be changed.

The OTA service characteristics are described in the following table. The UUID value of the service itself is 1d14d6eefd63-4fa1-bfa4-8f47b42119f0.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Characteristic	UUID	Type	Length	Support	Properties
OTA Control Attribute	F7BF3564-FB6D-4E53-88A4-5E37E0326063	Hex	1 byte	Mandatory	Write (4)
OTA Data Attribute (1)	984227F3-34FC-4045-A5D0-2C581F81A153	Hex	Variable; max 244 bytes	Mandatory	Write without response; Write
AppLoader version (2) (Bluetooth stack version 2,3)	4F4A2368-8CCA-451E-BFFF-CF0E2EE23E9F	Hex	8	Optional	Read
OTA version (2)	4CC07BCF-0868-4B32-9DAD-BA4CC41E5316	Hex	1	Optional	Read
Gecko Bootloader version (2)	25F05C0A-E917-46E9-B2A5-AA2BE1245AFE	Hex	4	Optional	Read
Application version	0D77CC11-4AC1-49F2-BFA9-CD96AC7A92F8	Hex	4	Optional	Read

Notes:

1. This characteristic is excluded from the user application GATT database.
2. Version information is automatically added by AppLoader when running in DFU mode. These are optional in the application GATT database.
3. This characteristic exposes the AppLoader version.
4. Silicon Labs highly recommends that the default property (i.e, write) be used only over bonded connections to prevent uploading of new firmware by untrusted/unknown devices.

The table below provides possible control words written to the OTA control characteristic.

Hex value	Description
0x00	OTA client initiates the upgrade procedure by writing value 0.
0x03	After the entire GBL file has been uploaded the client writes this value to indicate that upload is finished.
0x04	Request the target device to close connection. Typically the connection is closed by OTA client but using this control value it is possible to request that disconnection is initiated by the OTA target device.
Other values	Other values are reserved for future use and must not be used by application.

In DFU mode, AppLoader uses the full OTA service described above. The GATT database of the user application includes only a subset of the full OTA service. The minimum application requirement is to include the OTA Control characteristic. The application must not include the OTA Data characteristic in its GATT database (unless the OTA update is implemented fully in application code, as described in Application-Based OTA.)

From the user application viewpoint, only the OTA Control Attribute is relevant. In the OTA host example reference implementation included in the SDK, the OTA procedure is triggered when the client writes value 0 to the OTA Control Attribute. The user application does not handle data transfers related to OTA upgrades, so the OTA Data Attribute is excluded from the user applications GATT.

It is also possible to use an application-specific trigger to enter OTA mode, and therefore it is not necessary to include the OTA Control Attribute in the application's GATT database. If reboot into DFU mode is handled using some other mechanism, then it is possible to exclude the whole OTA service from the application GATT. However, it should be noted that to be compatible with the OTA host example from the SDK or the Si Connect smartphone app the OTA trigger must be implemented as described above.

The presence of the OTA Data Attribute in the GATT database is used by the OTA host example application to check whether the target device is running in normal mode (user application) or DFU mode (AppLoader). Therefore, the OTA Data Attribute must not be included in the user application's GATT. The OTA-enabled examples in the Bluetooth SDK only expose the OTA Control Attribute.

The four characteristics after the OTA Data Attribute are automatically added in the GATT database that is used by AppLoader. These include version information that can be read by the OTA client before starting the firmware update. For example, by checking the Ap- pLoader version, the OTA client may check if a full or partial update is needed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The AppLoader version is a 8-byte value that consists of four two-byte fields, indicating the AppLoader version in the form <major>.<mi- nor>.<patch>-<build>. For example, value 010000000000170b can be interpreted as version “1.0.0-2839”.

The OTA version is a 1-byte value that indicates the OTA protocol version for compatibility checking. This version number is incremented only when needed, if there are some changes in the OTA implementation that may cause backward compatibility issues.

The Gecko Bootloader version is a 4-byte value that is configured in a Gecko Bootloader project in config/btl_core_config.h. The two most significant bytes are the major and minor numbers. The other two bytes are customer-specific, and they can be set to indicate certain Gecko Bootloader configuration options (for example, whether secure boot is required or not). As an example, value 00000401 indicates that the Gecko Bootloader version is “1.4” and the customer-specific part is 0x0000 (this is the default if no customer-specific version info has been configured in the Gecko Bootloader project).

The application version is a 4-byte value, and it is initialized to the same value that is defined in config/app_properties_config.h. The encoding of this value is application specific. The application properties file is discussed in more detail in Application Properties in OTA Mode.

AppLoader does not include support for encryption or bonding and therefore there are no access restrictions on any of the characteristics listed in Silicon Labs OTA GATT Service. Because the user application has its own GATT database it is possible to include additional security requirements there as needed. For example, the user application can require that the OTA Control Attribute is writable only by a bonded client so that only bonded client can trigger reboot into DFU mode.

For additional security, it is recommended to configure the Gecko Bootloader to use secure boot and signed GBL images.

OTA Advertising Data

The default OTA advertising data includes the device name, TX power, advertising flags, and the Bluetooth device address. The following text snippet illustrates typical default raw OTA advertising data and how it is dissected into different advertising data elements.

```
0x02010604094F5441081B005B7728E20A68020A00: raw OTA advertising data
02:                             length = 2bytes
01:                             type = flags
06:                             value = 6 (General Discoverable Mode, BR/EDR Not Supported)
-----------------------------------------------------------------------------------------
04:                             length = 4 bytes
09:                             type = complete local name
4F5441:                         value = OTA
-----------------------------------------------------------------------------------------
08:                             length = 8bytes
1B:                             type = BL device address
005B7728E20A68:                 value = 00(public) 5B:77:28:E2:0A:68
-----------------------------------------------------------------------------------------
02:                             length = 2 bytes
0A:                             type = Tx Power
00:                             value = 0dBm
```

For Series 1 devices, `sl_bt_ota_set_advertising_data(packet_type, adv_data_len, adv_data)` can be used to set custom OTA advertising data (only available in GSDK releases). The packet type identifies whether data is intended for advertising packets or scan response packets.

2: OTA advertising packets

4: OTA scan response packets

You can set a maximum of 31 bytes of data.

> Note: The OTA configuration commands must be called after NVM3 (PS) initialization has been done; that is, after `sl_bt_init()`.

The Bluetooth device address that is used in OTA mode is determined as follows:

- Use a static random address if it has been enabled in the OTA configuration flags.

Copyright © 2025 Silicon Laboratories. All rights reserved.

- If the user application has overridden the default Bluetooth address (using command `sl_bt_system_set_identity_address(address, type)`), then use this address.
- The default Bluetooth address (programmed into the device in production) is used if neither a static random address nor custom address has been defined.

For Series 1 devices, in OTA mode, the TX power is hardcoded to 0 dBm. For Series 2 devices, the TX power level can be configured using the Bluetooth AppLoader OTA DFU (bootloader_apploader) component inside the bootloader-apploader project.

Application Properties in OTA Mode

The source file app_properties.c must be included in projects that use OTA and the Gecko Bootloader. This file is included in the SDK examples by default. Application properties are stored in a fixed location in code flash so that AppLoader can access the data when the device is running in OTA mode. The properties include a 32-bit version number that is application-specific. It is up to the application designer to decide how this value is encoded. This value is exposed in the GATT database used by AppLoader so that the OTA client can read it over the Bluetooth connection after the device has been rebooted into OTA mode.

The version information is set using following #define in config/app_properties_config.h:

```
// <o SL_APPLICATION_VERSION> Version number for this application
// <0-4294967295:1>
// <i> Default: 1 [0-4294967295]
#define SL_APPLICATION_VERSION                1
```

The default value is set to 1, but it is strongly recommended that meaningful version number information is added here so that the OTA client can check the exact version that is installed on the target device. This allows better management of OTA updates of units that are deployed in the field, especially in cases where units are running different versions of the application. If AppLoader does not detect any valid application at all, then the application version in the GATT database is initialized to value zero.

OTA Error Codes

When a new GBL file is being uploaded, the AppLoader performs various checks. AppLoader can signal possible errors to the OTA client in two ways:

1. Response to the OTA termination code (0x03) that is written to the OTA Control characteristic.
2. Response to writes to the OTA Data characteristic.

Option 2 is not available if the client uses unacknowledged writes. In that case, the possible error code is not available until the entire file has been uploaded and client finishes the upload by writing to the OTA Control characteristic.

The OTA client must always check the response value to the last write to the OTA Control characteristic. Any non-zero value indicates that the update was not successful. In that case, the device is not able to boot into the main program but rather stays in OTA mode. This makes it possible to try the update again.

The following table summarizes the possible result codes returned by AppLoader.

Result Code	Name	Description
0x0000	OK	Success / No errors found.
0x0480	CRC_ERROR	CRC check failed, or signature failure (if enabled).
0x0481	WRONG_STATE	This error is returned if the OTA has not been started (by writing value 0x0 to the control endpoint) and the client tries to send data or terminate the update.
0x0482	BUFFERS_FULL	AppLoader has run out of buffer space.
0x0483	IMAGE_TOO_BIG	New firmware image is too large to fit into flash, or it overlaps with AppLoader.
0x0484	NOT_SUPPORTED	GBL file parsing failed. Potential causes are for example: 1. Attempting a partial update from one SDK version to another (such as 2.3.0 to 2.4.0). 2. The file is not a valid GBL file (for example, client is sending an EBL file).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Result Code	Name	Description
0x0485	BOOTLOADER	The Gecko bootloader cannot erase or write flash as requested by AppLoader, for example if the download area is too small to fit the entire GBL image.
0x0486	INCORRECT_BOOTLOADER	Wrong type of bootloader. For example, target device has UART DFU bootloader instead of OTA bootloader installed.
0x0487	APPLICATION_OVERLAP_APPLOADER	New application image is rejected because it would overlap with the AppLoader.
0x0488	INCOMPATIBLE_BOOTLOADER_VERSION	AppLoader in requires Gecko Bootloader v1.11 or higher.
0x0489	ATT_ERROR_APPLICATION_VER-SION_CHECK_FAIL	AppLoader fails checking application version.

Note that the error codes listed above are applicable only when testing with the NCP host example. The upper half of the result code (0x04**) is generated by the BLE stack running on the NCP host device. The size of the ATT error code that is transmitted over the air is one octet. Values in the range 0x80-0x9F are reserved for application-specific errors in the Bluetooth specification.

Firmware Upgrade from PS Store to NVM3

If an application is already in the field using PS Store and should be upgraded to use NVM3, it can be upgraded using OTA DFU with new firmware that already uses NVM3.

However, in this case the data stored in the PS Store cannot be preserved. All bonding information and stored user data will be lost. Nevertheless, the new application can reinitialize the NVM area (at the end of the main flash) to use NVM3 instead of PS Store, and after the upgrade NVM3 will be functional.

Upgrading software from PS Store to NVM3 is challenging, mostly because the application provides information to the AppLoader through non-volatile memory (PS Store / NVM3), which gets upgraded as well. The following are the detailed steps to perform an OTA upgrade from PS Store to NVM3, where the device doing the upgrade is bonded with the device to be upgraded.

> Note: The procedure illustrates the situation where the upgrader and the device to be upgraded are bonded to highlight all the challenges. Bonding is not a condition for upgrading from PS Store to NVM3.

1. The device uses an application with PS Store.
2. The application sets `random address OTA flag` and `OTA device name` in PS Store 10.2_2020q4.
3. The Smartphone opens a connection to this device and gets bonded (if not bonded already).
4. The application stores bonding information in PS Store.
5. The Smartphone resets the device into OTA mode by writing 0x00 into the OTA control characteristic.
6. AppLoader (with PS Store support) starts.
7. AppLoader advertises with a random address and the OTA device name.
8. The Smartphone connects and uploads a new AppLoader (with NVM3 support).
9. The device resets and applies the new AppLoader image.
10. The new AppLoader (with NVM3 support) starts.
11. The AppLoader advertises with the public address and with the default name ("AppLoader"), because it cannot read the random address flag and the OTA device name from PS Store.
12. The Smartphone sees the device as bonded because bonding information is associated with the public address, but AppLoader does not support bonding.
13. The Smartphone removes bonding information for the device before re-connecting.
14. The Smartphone connects and uploads a new application (with NVM3 support).
15. The new application starts.
16. The application initializes NVM3 by reformatting the NVM area.
17. The application sets `random address OTA flag` and `OTA device name` in NVM3.
18. The Smartphone opens a connection and gets bonded (again).
19. The application stores bonding information in NVM3.

After this, NVM3 to NVM3 update will work normally:

1. The device uses an application with NVM3.
2.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The application sets `random address OTA flag` and `OTA device name` in NVM3.

3. The Smartphone opens a connection to this device and gets bonded (if not bonded already).
4. The application stores bonding information in NVM3.
5. The Smartphone resets the device into OTA mode by writing 0x00 into the OTA control characteristic.
6. AppLoader (with NVM3 support) starts.
7. AppLoader advertises with a random address and the OTA device name.
8. The Smartphone connects and uploads a new AppLoader (with NVM3 support).
9. The device resets and applies the new AppLoader image.
10. The new AppLoader (with NVM3 support) starts.
11. AppLoader advertises with a random address and the OTA device name.
12. The Smartphone connects and uploads a new application (with NVM3 support).
13. The new application starts.
14. The Smartphone opens a connection and encrypts the connection with existing bonding information.
15. Bonding information is still stored in NVM3.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Working with Apploader and Secure Boot

Working with AppLoader and Secure Boot

When employing secure boot, the AppLoader and application must be signed individually for the application to be allowed to run. This is because the AppLoader authenticates the application before allowing it to run, just as the bootloader authenticates the AppLoader before allowing it to run.



This section describes how to work with this capability in a production environment.

Creating a Single Signed Image with a Batch File

As described previously, a signed GBL file can be produced by executing `create_bl_files.bat/create_bl_files.sh/create_bl_files.py` with a private signing key file, `app-sign-key.pem` in the same folder or by using the Post-Build Editor. The resulting GBL file, `full.gbl`, can be flashed directly to the target device for a successful boot. This method is convenient for testing secure boot during the development process, but is not secure for production since it requires the private signing key to be available in plain PEM format, rather than isolating it in a Hardware Security Module (HSM) and does not support the use of bootloader certificates.

Signing Firmware Images for Production

For Series 1 Devices:

To sign a firmware image using an HSM, the image must first be separated into the AppLoader and application parts as follows.

1. Extract the AppLoader portion with the following command: `objcopy -O srec -j .text_apploader* apploader.s37`.
2. Sign the AppLoader for secure boot. For specific instructions on signing images with an HSM, see *Signing an Application for Secure Boot using a Hard Security Module* in Simplicity Commander Reference Guide. It is also possible to sign the AppLoader with a certificate although direct signing is sufficient for most use cases. For instructions on signing with a certificate with an HSM, see *Signing an Application for Secure Boot using an Intermediary Certificate* in the Simplicity Commander Reference Guide.
3. Extract the application with the following command: `objcopy -O srec -R .text_apploader* -R .text_signature* application.s37`.
4. Sign the application for secure boot. The instructions for signing the AppLoader in step 2 above also apply to the application.
5. Combine the signed AppLoader and application into a single image as follows: `commander convert <signed apploader>.s37 <signed application>.s37 –outfile signed_fw_image.s37`.
6. Optionally, see *Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module* and *Creating a Signed GBL File Using a Hardware Security Module* in the *Simplicity Commander Reference Guide*.

For Series 2 devices (EFR32xG2x), the AppLoader can be included in the bootloader project as a software component. This makes it possible to sign the application and bootloader binaries without any need to perform steps 1 – 6 above.

For more information on secure boot, see Series 2 and Series 3 Secure Boot with RTSL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

OTA Delta DFU

SiSDK 2024.6.1 introduced a new feature called Delta DFU. This feature can be used to speedup OTA DFU and is especially useful in large wireless networks when upgrading large amounts of devices. Delta DFU is accomplished by:

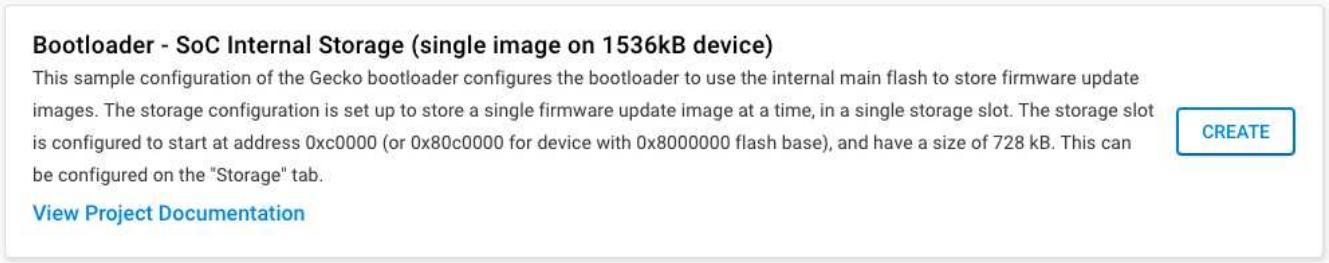
1. Comparing the current and new application firmwares
2. Creating a patch file based on the comparison
3. Only uploading the patch file to the target device

Delta DFU is not supported when using AppLoader. Instead, OTA must be implemented in the application as described in Section OTA DFU Example, with an internal-storage bootloader. The storage space must be large enough to hold the patch file and the reconstructed firmware. We recommend the implementation of an additional handshake process to determine the current firmware version before the appropriate previously-generated Delta DFU image is being selected for upload.

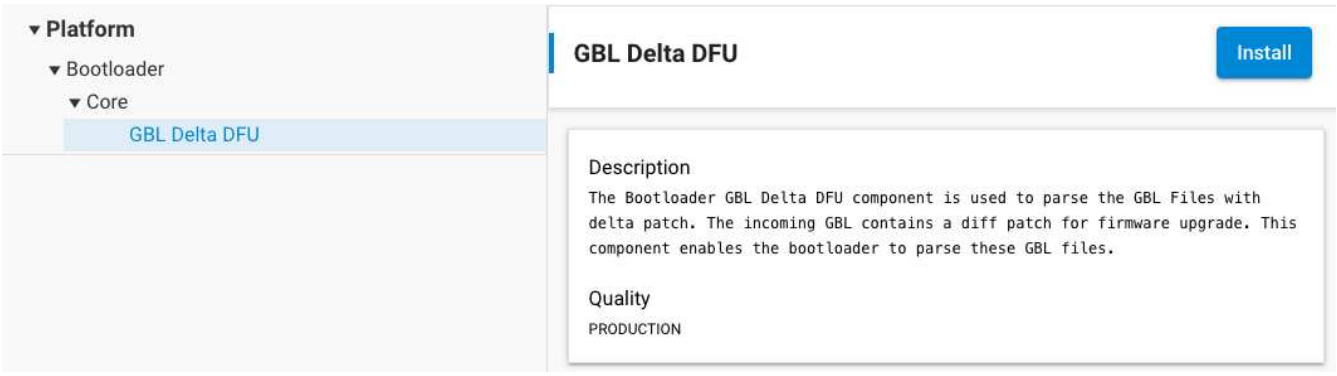
Preparing the Bootloader

In Delta DFU, the bootloader takes care of applying the patch file to the current application firmware. For this, we need to prepare the bootloader:

1. Create an internal-storage bootloader project



2. Add the GBL Delta DFU (bootloader_gbl_delta_dfu) software component

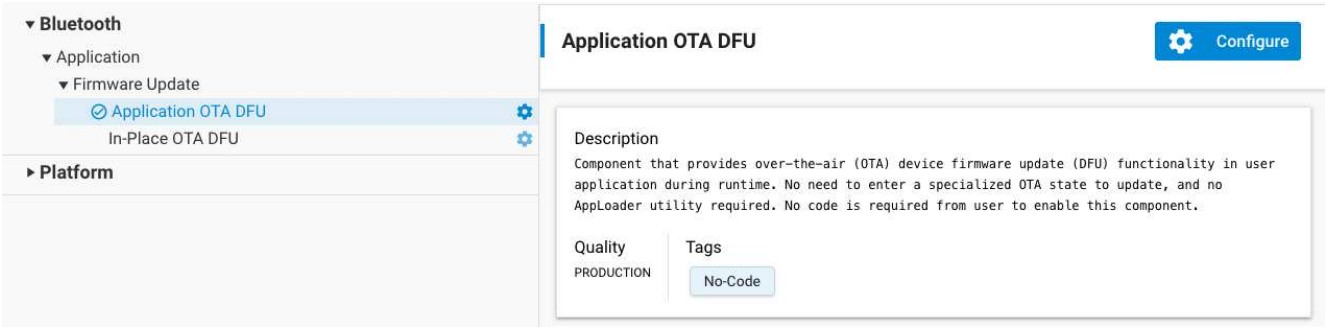


Preparing the Application

Both the current and the new application must support application-based OTA. Sample applications include the In-Place OTA DFU (in_place_ota_dfu) software component by default. However, OTA Delta DFU does not use the in-place method for DFU.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Therefore, In-Place OTA DFU (in_place_ota_dfu) must be replaced by the Application OTA DFU (app_ota_dfu) software component.



Creating a Patch File

The patch file can be created using Simplicity Commander. For this, we need both the current firmware file, and the new version.

```
commander gbl create patch.gbl --app app_v2.out --delta-app app_v1.out
```

For this to work, an ELF file with the same name must exist next to the application image. Once the patch file is created it can further be compressed using LZMA or LZ4. More information on creating a patch file can be found in Simplicity Commander Reference Guide.

> Note: Patch-file creation with GBL4 is not supported.

Delta DFU process

Once a patch file has been created, it can be uploaded to the target device via OTA just as before in a partial update. When the patch file has been uploaded, the bootloader will reconstruct it into a full firmware image. This firmware image will then be written over the old image.

Delta DFU and Secure Boot

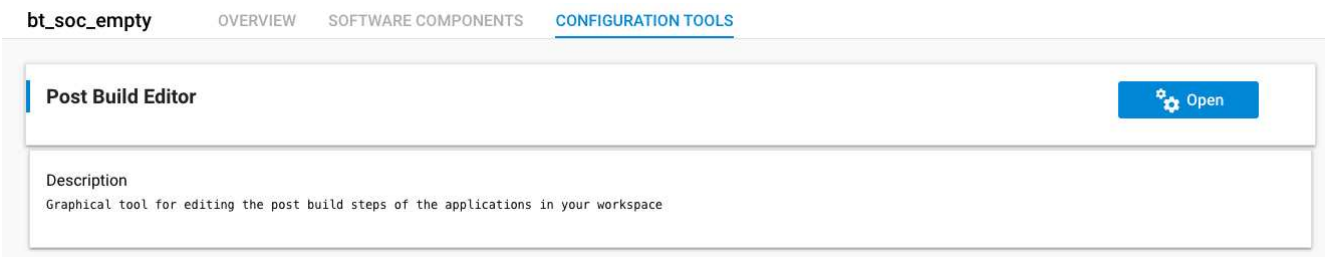
When using Delta DFU with Secure Boot, make sure both the current firmware and the new version (app_v1 and app_v2, respectively) are signed using the same private key before creating the patch file. When the patch file is created, as described in Creating a patch file, the signature of app_v2 will be included in the patch. Therefore, the reconstructed firmware will have the signature of app_v2, and the new application will boot as expected.

Copyright © 2025 Silicon Laboratories. All rights reserved.

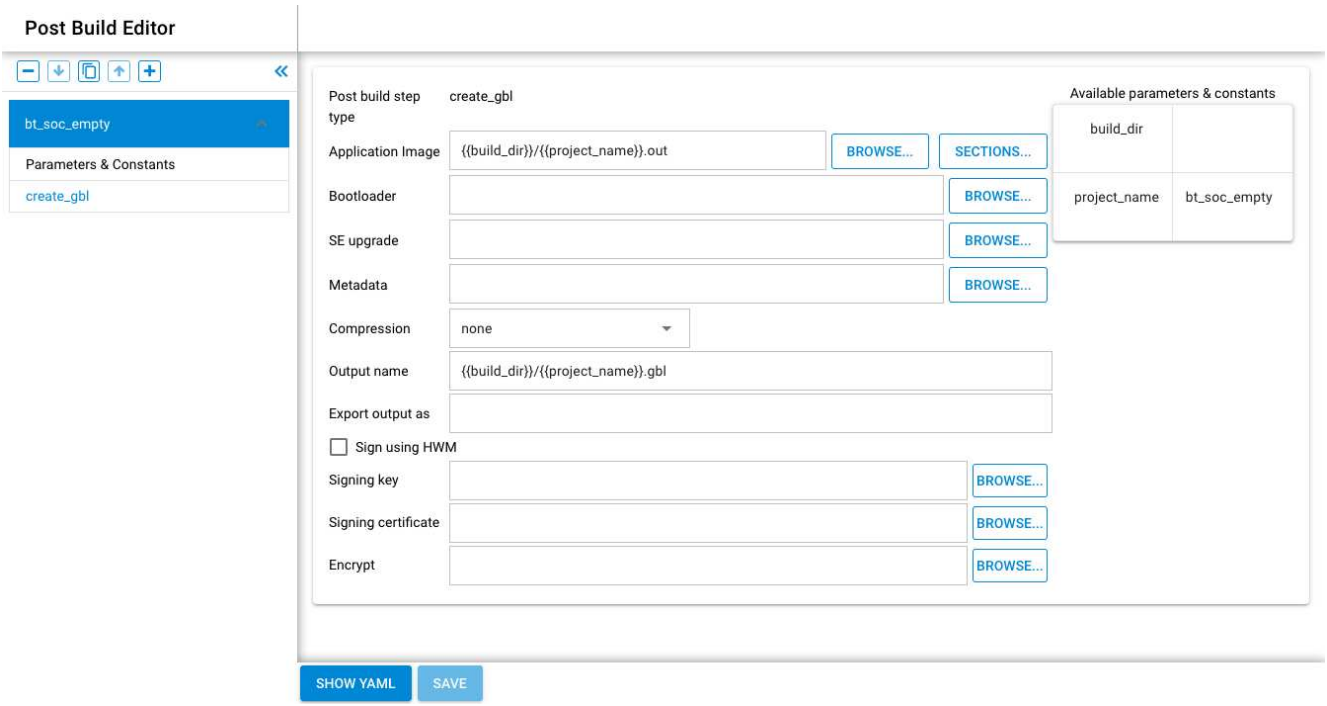
Post Build Editor

Post Build Editor

From SiSDK 2024.12.0 and after, there is a new way of generating the GBL files for OTA (GBL3 format only, as of now) via a convenient graphical user interface instead of by using the command line solutions introduced previously. The option is to be found among the CONFIGURATION TOOLS:



Every previously introduced configuration option is available here, including compression, signing, and encryption:



Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Bootloader with Silicon Labs Connect

Using the Gecko Bootloader with Silicon Labs Connect

NOTE: This section replaces *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*. Further updates to this application note will be provided here.

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with the Silicon Labs Connect stack, part of the Silicon Labs Flex SDK (Software Development Kit). It supplements the general Gecko Bootloader implementation information provided in *UG266: Silicon Labs Gecko Bootloader User's Guide*. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to *UG103.6: Bootloading Fundamentals*.

The *Connect User's Guide* is a series of documents that provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. Refer to Bootloading and OTA with Silicon Labs Connect to learn about the bootloader options (standalone, application, and Over the Air (OTA)) available for use within Connect-based applications.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of bootload functions, from device initialization to firmware upgrades. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). These images are produced with the file extension ".gbl". Additional information on the GBL file format is provided in *UG103.6: Bootloading Fundamentals*.

The Gecko Bootloader has a two-stage design, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be enabled and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image by a physical connection such as UART or SPI. To function as a standalone bootloader, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by reprogramming the device's flash with the firmware upgrade image stored in a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, and so on). The download space is either an external memory device such as an EEPROM or dataflash or a section of the chip's internal flash. The Gecko Bootloader can partition the download space into multiple storage slots, and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation has to be configured.

OTA (over the air) is often confused with application bootloader, since application bootloader can be used to flash an image that was distributed OTA. However, OTA is an application feature. For example, an NCP gateway device typically uses the standalone bootloader, so the host can update it, but the Host-NCP application can work as an OTA server.

This document describes how to use both standalone and application models with Silicon Labs Connect.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Standalone Bootloaders

Using the Gecko Standalone Bootloaders

A Gecko Bootloader-based standalone bootloader receives an application image onto a target device by serial transfer via SPI or UART. If using UART, you can establish a serial connection between a source device and a target device's serial interface and upload a new software image to it using the XModem protocol. If you need information on the XModem protocol, a good place to start is http://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up-to-date links to protocol documentation.

Performing a Serial Upload – UART XMODEM Bootloader

Serial upload can be performed with any source device that provides the expected serial interface method. This can be a Windows-based PC, a Linux or Mac OS-based device, or an embedded MCU with no operating system. UART transfer can be done with a third-party serial terminal program like TeraTerm on Windows or Linux minicom or with user-compiled host code. However, drivers for SPI Master or UART may vary with operating systems, and serial terminal programs may vary in timing and performance, so if you are unsure about what driver or program to use on your source code, please consult Silicon Labs technical support.

To open a serial connection over UART, the source device connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1), with no flow control by default. These options may be changed using the component options in the Bootloader project.

> Note: The UART-based serial bootloader configurations do not employ any flow control in the communication channel by default, because the XModem protocol used for image transfer already has built-in flow control mechanisms. However, Silicon Labs' normally-supplied NCP firmware does utilize either hardware-based (RTS/CTS) or software-based (XON/XOFF) flow control, so a host device must take care to temporarily disable the flow control when placing its NCP into serial bootloading mode. Alternatively, application designers can change the options in the provided bootloader project and customize the serial bootloader's handling of the UART to add hardware flow control at their discretion.

Once the connection with a UART-based serial bootloader is established:

1. The target device's bootloader sends output over its serial port after it receives a carriage return from the source device at the expected baud rate. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port. Note that serial bootloaders typically don't enforce any timeout when awaiting the initial serial handshake via carriage return, so the bootloader will wait indefinitely in this mode until guided by the source device or until the chip is reset.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following ASCII-based output, (where X.Y.A corresponds to the major, minor, and sub-minor fields of the bootloader version number, respectively):

```
Gecko Bootloader vX.Y.A
1. upload gbl
2. run
3. ebl info
BL >
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> Note:The third menu option has no effect. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new options might be added.

After listing the menu options, the bootloader's “BL >” prompt displays, and the ASCII character corresponding to the number of each option can then be entered by the source to select the described action, such as ‘2’ (ASCII code 0x32) to run the firmware presently loaded in the application area. Here again, no timeout is enforced by the bootloader, so it will wait indefinitely until a character is received or the chip is reset. Note that while the menu interface is designed for human interaction, the transfer can still be performed programmatically or through a scripted interface, provided the source device sends the expected ASCII characters to the target at appropriate times.

> Note: Scripts that interact with the bootloader should use only the “BL >” prompt to determine when the bootloader is ready for input.

Selecting menu option 1 initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of a GBL file over the expected serial interface, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. Once uploading begins (first XModem SOH data packet received), the bootloader expects each successive XModem SOH packet within 1 second, or else a timeout error will be generated and the session will abort.
4. After an image successfully uploads, the XModem transaction completes and the bootloader displays ‘Serial upload complete’ before redisplaying the menu.

Errors and Status Codes

If an error occurs during the upload, the UART serial bootloader displays the message ‘Serial upload aborted,’ followed by a more detailed message and a hex error code. Some of the more common errors are shown in the following table. The UART serial bootloader then redisplays the bootloader menu.

The following tables describe the normal status codes, error conditions, and special characters or enumerations used by the Gecko Bootloader. For additional status codes, see the Gecko Bootloader API documentation installed with your SDK in the platform/bootloader/documentation folder.

Serial Uploading Statuses and Error Messages

Hex code	Constant	Description
0x00	BL_SUCCESS	Default success status.
0x01	BL_ERR	General error processing packet.
0x1C	BLOCK_TIMEOUT	The bootloader timed out waiting for some part of the XModem frame.
0x21	BLOCKERR_SOH	The bootloader did not find the expected start of header (SOH) character at the beginning of the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected the sequence check byte of the XModem frame was not the inverse of the sequence byte.
0x23	BLOCKERR_CRCH	The bootloader encountered an error while comparing the high bytes of the received and calculated CRCx of the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while comparing the low bytes of the received and calculated CRCs of the XModem frame.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hex code	Constant	Description
0x25	BLOCKERR_SEQUENCE	The bootloader did not receive the expected sequence number in the current XModem frame.
0x26	BLOCKERR_PARTIAL	The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost).
0x27	BLOCKERR_DUPLICATE	The bootloader encountered a duplicate of the previous XModem frame.
0x40	BL_ERR_MASK	Bitmask for any bootloader error codes returned in CAN or NAK frame.
0x41	BL_ERR_HEADER_EXP	No GBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Failed to write header or CRC.
0x43	BL_ERR_CRC	File or written image failed CRC check.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in GBL image.
0x45	BL_ERR_SIG	Invalid GBL header contents.
0x46	BL_ERR_ODD_LEN	Trying to flash odd number of bytes.
0x47	BL_ERR_BLOCK_INDEX	Indexed past end of block buffer.
0x48	BL_ERR_OVWR_BL	Attempt to overwrite bootloader flash.
0x49	BL_ERR_OVWR_SIMEE	Attempt to overwrite SIMEE flash.
0x4A	BL_ERR_ERASE_FAIL	Flash erase failed.
0x4B	BL_ERR_WRITE_FAIL	Flash write failed.
0x4C	BL_ERR_CRC_LEN	End tag CRC wrong length.
0x4D	BL_ERR_NO_QUERY	Received data before query request/response.
0x4E	BL_ERR_BAD_LEN	An invalid length was detected in the upgrade image file.
0x4F	BL_ERR_TAGBUF	Insufficient tag buffer size or an invalid length was found in the GBL image.

Special Characters Used in Packet Types

Hex code	Constant	Description
0x01	SOH	Start of Header.
0x03	CTRL_C	Cancel (from sender).
0x04	EOT	End of Transmission.
0x06	ACK	Acknowledged.
0x15	NAK	Not acknowledged.
0x18	CAN	Cancel
0x43	C	ASCII ‘C’.
0x51	QUERY	ASCII ‘Q’.
0x52	QRESP	ASCII ‘R’.

Status Codes Returned in a Synchronous Response

Hex code	Constant	Description
0x16	TIMEOUT	Bootloader timed out expecting characters.
0x17	FILEDONE	EOT process successfully.
0x18	FILEABORT	Transfer aborted prematurely.
0x19	BLOCKOK	Data block processed OK.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hex code	Constant	Description
0x1A	QUERYFOUND	Successful query.

Running the Application Image

For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu. For SPI-based variants, which don't use a menu, the application starts up immediately upon ACKing the EOT frame from the source device.

Performing a Bootloader Upgrade

To perform a bootloader upgrade with the standalone bootloaders, simply transmit two GBL files in succession. The first GBL file should contain a main bootloader upgrade image. After upload is completed, the device resets to upgrade the main bootloader. For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the first stage bootloader to perform the upgrade, before returning to the upgraded main bootloader. The second GBL file, containing an application upgrade image, can then be uploaded.

Upload Recovery

If an image upload fails, the target node is left without a valid application image. The standalone bootloader will re-enter firmware upgrade mode if it determines that the application image isn't valid. However, the application image may have a valid structure, but contain a bug preventing normal operation. Regardless of the serial interface supported by your standalone bootloader, a GPIO-based trigger can be used to facilitate recovery via serial upload.

You can configure your standalone bootloader to use a software-based GPIO pin check or other schemes of recovery mode activation such as button recovery by configuring the GPIO Activation component.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Application Bootloader

Using the Gecko Application Bootloader

Acquiring a New Image

The application bootloader relies on application code to obtain new code images. The bootloader itself only knows how to read a GBL image stored in the download space and copy the relevant portions to the main flash block. This approach means that the application developer is free to acquire the new code image in any way that makes sense (serial, OTA, and so on).

Typically, application developers choose to acquire the new code image over-the-air (OTA) since this is readily available on all devices. The acquisition of a new image in Connect is covered extensively in the Connect Users Guide.

Performing an Application Upgrade

The application and the bootloader have limited indirect contact. Their only interaction is through passing non-volatile data across module reboots.

Once the application decides to install a new image saved in the download space it calls the Ember HAL API `halAppBootloaderInstallNewImage()`. This call indicates to the bootloader that it should attempt to perform a firmware upgrade from storage slot 0, and reboots the device. This API is backwards-compatible with the legacy Ember bootloader. If performing a firmware upgrade from a different storage slot than slot 0 is desired, the Gecko Bootloader API should be used instead, by calling `bootloader_setBootloadList()`. If the bootloader fails to install the new image, it sets the reset cause to `RESET_BOOTLOADER_BADIMAGE` and resets the module. Upon startup, the application should read the reset cause with `halGetExtendedResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_BADIMAGE`, the application knows the install process failed and can attempt to obtain a new image. A printable error string can be acquired from calling `halGetExtendedResetString()`. Under normal circumstances, the application bootloader does not print anything on the serial line.

External Storage Application Bootloader

See *UG266: Silicon Labs Gecko Bootloader User's Guide* for information about memory configuration for external storage bootloaders.

Application bootloaders typically use an external device to store the downloaded application image. This device can be accessed over either an I2C or SPI serial interface. Refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* for a list of supported Dataflash/EEPROM devices. It is important to select a device whose size is at least the size of your flash in order to fit the application being bootloaded.

The default recommendation for external SPI serial flash is the MX25R, as this is available in standard and smaller packages, is supported by standard drivers, and has very low software-enabled sleep current without the need for an external shutdown control circuit. Most radio boards from Silicon Labs are populated with the MX25R8035F for evaluation purposes. In general, customers should use parts that have low sleep current, don't require external shutdown control circuitry, and have software shutdown control. When using components with high idle/sleep current and no software shutdown control, an external shutdown control circuit is recommended to reduce sleep current.

For EFR32xG1, only the serial dataflash option is available; no internal storage application bootloader is presently offered.

> Note: Some of these chips have compatible pinouts with others, but there are several incompatible variations. Contact Silicon Labs for details on connecting I2C or other SPI dataflash chips to an EFR32.

Copyright © 2025 Silicon Laboratories. All rights reserved.

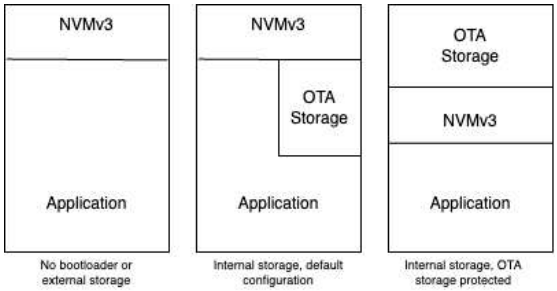
Read-Modify-Write pertains to a feature of certain dataflash chips that their corresponding driver exposes, and that is exploited by the bootloader library. Chips without this feature require a page erase to be performed before writing to that page, which precludes random-access writes by an application.

Internal Storage Application Bootloader

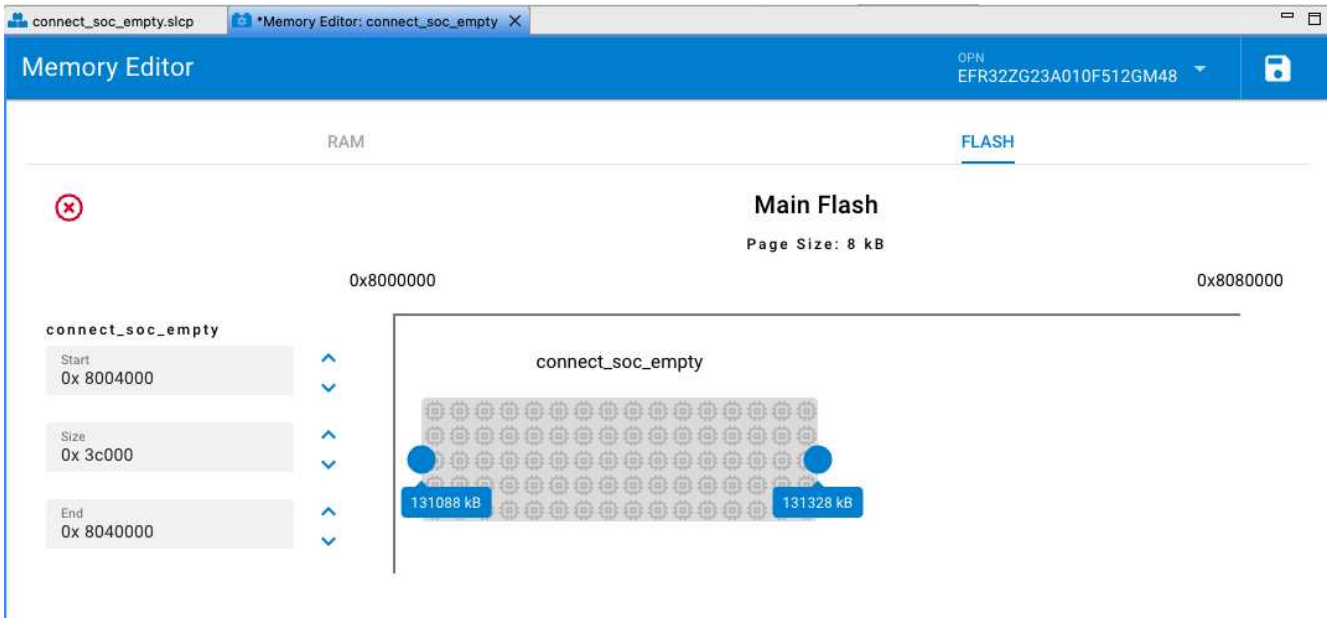
The internal storage bootloader is essentially an application bootloader with a data flash driver that uses a portion of the on-chip flash for image storage instead of an external storage chip. See *UG266: Gecko Bootloader User's Guide* for information about memory configuration for internal storage.

If you don't modify the flash layout of your application, the NVMv3 token storage is at the end of the flash memory. By default, our internal storage bootloader sets up a slot that protects at least the default 36k of token storage (48k on some parts). This configuration however has some drawbacks:

- You might want to optimize your storage space, and use a smaller NVMv3 token storage, in which case the bootloader protects more than what it needs to.
- Apart from the size outputs of the compiler, you won't get notified if you run out of flash space. If you program the device directly, not through the bootloader, you are allowed to overlap your application with OTA storage, which will cause failure the first time you erase the OTA storage (depicted in the middle below)

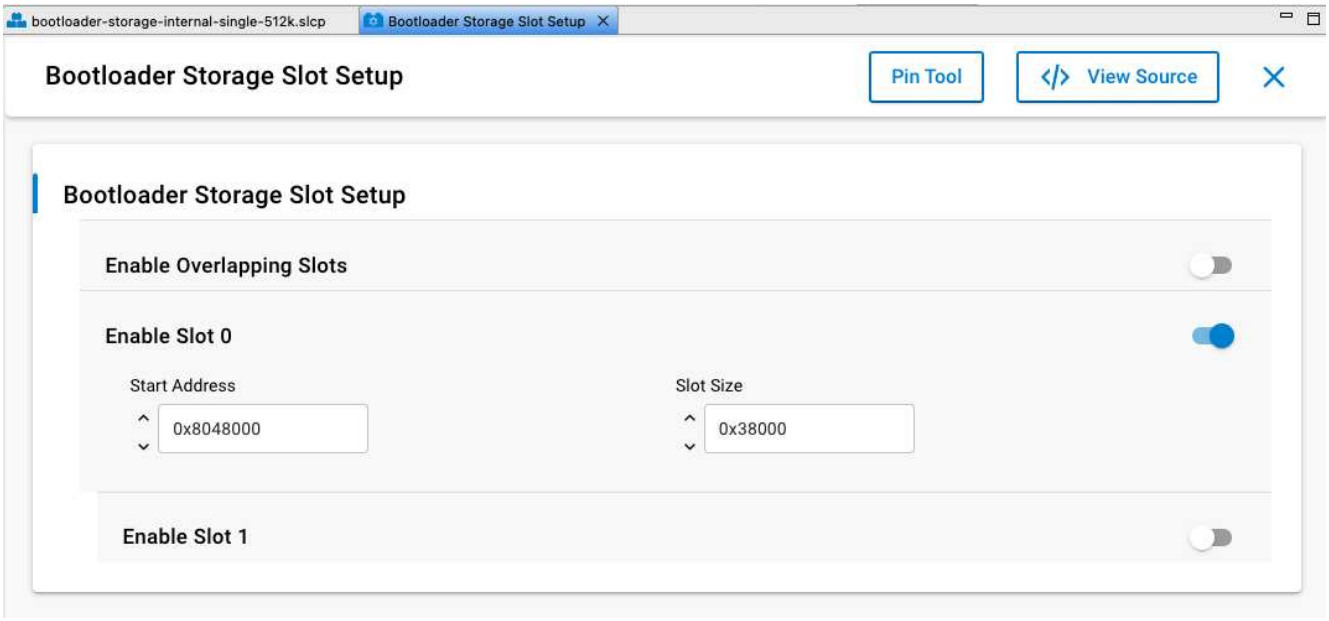


To avoid these problems, you can modify your main application's memory allocation in the memory editor. For example, a typical application will use half, or a bit more than half, of the flash for the application and NVMv3, while the remainder can be used as OTA storage. The OTA storage can be smaller, because it doesn't include the NVMv3 and because bootloader can be configured to use compressed images.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Finally, you should configure the bootloader project's Bootloader Storage Slot Setup component to utilize the space you protected from your application:



> Note: On EFR32xG1 and Series 2 devices, the application is offset by 16 kB to accommodate the bootloader at the bottom of main flash. On other EFR32 Series 1 devices, the Gecko Bootloader resides in the Bootloader flash region outside of main flash block.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using The Gecko Bootloader With Emberznet

Using the Gecko Bootloader with EmberZNet

Note: This section replaces *AN1084: Using the Gecko Bootloader with EmberZNet*. Further updates to this application note will be provided here.

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with EmberZNet. It supplements the general Gecko Bootloader implementation information provided in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher or *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrading image files, refer to *UG103.6: Bootloader Fundamentals*.

Key Points

- Gecko Bootloader overview.
- Using Gecko standalone bootloaders.
- Using Gecko application bootloaders.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of bootload functions, from device initialization to firmware upgrades. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). These images are produced with the file extension ".gbl". Additional information on the GBL file format is provided in *UG103.6: Bootloader Fundamentals*.

On EFR32 Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image by computing a CRC32 checksum, before copying the upgrade image to the main bootloader location.

On EFR32 Series 2 devices, the Gecko bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the hardware peripheral Secure Element. The Secure Element provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Element installs it. On some devices, the Secure Element is also capable of verifying the authenticity of the main bootloader upgrade image against a root of trust. The Secure Element itself is upgradable using the same mechanism. More details are provided in *UG266: Silicon Labs Gecko Bootloader User's Guide*.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be enabled and configured through the Simplicity Studio IDE.

> Note: Zigbee EmberZNet SDK 7.0 introduced a new component-based architecture, along with a Project Configurator and other tools to replace AppBuilder and plugin configuration. In general, the new software components are comparable to the old plugins. Unless otherwise specified, the term 'component' should be understood to refer to the corresponding plugin, if you are working in SDK 6.10.x or lower. For more information, see *AN1301: Transitioning from Zigbee EmberZNet SDK 6.x to SDK 7.x*.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image by a physical connection such as UART or SPI. To function as a standalone bootloader, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by reprogramming the device's flash with the firmware upgrade image stored in a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, and so on). The download space is either an external memory device such as an EEPROM or dataflash or a section of the chip's internal flash. The Gecko Bootloader can partition the download space into multiple storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

slots, and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation has to be configured.

This document describes how to use both models with Zigbee EmberZNet.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Standalone Bootloaders

Using the Gecko Standalone Bootloaders

A Gecko Bootloader-based standalone bootloader receives an application image onto a target device by serial transfer via SPI or UART. If using UART, you can establish a serial connection between a source device and a target device's serial interface and upload a new software image to it using the XModem protocol. If you need information on the XModem protocol, a good place to start is https://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up-to-date links to protocol documentation.

Performing a Serial Upload – UART XMODEM Bootloader

Serial upload can be performed with any source device that provides the expected serial interface method. This can be a Windows- based PC, a Linux or Mac OS-based device, or an embedded MCU with no operating system. UART transfer can be done with a third-party serial terminal program like Windows HyperTerminal or Linux "lrzsz" or with user-compiled host code. However, drivers for SPI Master or UART may vary with operating systems, and serial terminal programs may vary in timing and performance, so if you are unsure about what driver or program to use on your source code, please consult Silicon Labs technical support.

To open a serial connection over UART, the source device connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1), with no flow control by default. These options may be changed using the component options in the Bootloader project.

> Note: The UART-based serial bootloader configurations do not employ any flow control in the communication channel by default, because the XModem protocol used for image transfer already has built-in flow control mechanisms. However, Silicon Labs' normally-supplied NCP firmware does utilize either hardware-based (RTS/CTS) or software-based (XON/XOFF) flow control, so a host device must take care to temporarily disable the flow control when placing its NCP into serial bootloading mode. Alternatively, application designers can change the options in the provided bootloader project and customize the serial bootloader's handling of the UART to add hardware flow control at their discretion.

Once the connection with a UART-based serial bootloader is established:

1. The target device's bootloader sends output over its serial port after it receives a carriage return from the source device at the expected baud rate. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port. Note that serial bootloaders typically don't enforce any timeout when awaiting the initial serial handshake via carriage return, so the bootloader will wait indefinitely in this mode until guided by the source device or until the chip is reset.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following ASCII-based output, where X.Y.A corresponds to the major, minor, and sub-minor fields of the bootloader version number, respectively:

```
Gecko Bootloader vX.Y.A
1. upload gbl
2. run
3. ebl info
BL >
```

> Note: The third menu option has no effect. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
options might be added.
```

After listing the menu options, the bootloader's "BL >" prompt displays, and the ASCII character corresponding to the number of each option can then be entered by the source to select the described action, such as ‘2’ (ASCII code 0x32) to run the firmware presently loaded in the application area. Here again, no timeout is enforced by the bootloader, so it will wait indefinitely until a character is received or the chip is reset. Note that while the menu interface is designed for human interaction, the transfer can still be performed programmatically or through a scripted interface, provided the source device sends the expected ASCII characters to the target at appropriate times.

```
Note: Scripts that interact with the bootloader should use only the "BL >" prompt to
determine when the bootloader is ready for input.
```

Selecting menu option 1 initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of a GBL file over the expected serial interface, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. Once uploading begins (first XModem SOH data packet received), the bootloader expects each successive XModem SOH packet within 1 second, or else a timeout error will be generated and the session will abort.
4. After an image successfully uploads, the XModem transaction completes and the bootloader displays ‘Serial upload complete’ before redisplaying the menu.

Performing a Serial Upload - SPI

To open a serial connection over SPI, the source device must act as SPI Master using Mode 0 or Mode 2. It must also react to edge- triggered interrupts from the slave device using the same nHOST_INT logic and SPI framing as the EZSP-SPI protocol described in *AN711: SPI Host Interfacing Guide for Zigbee*.

Once the SPI slave enters bootloader mode, which includes a reset sequence with host interrupt and Reset response frame similar to the reset sequence of a normal EZSP-SPI NCP, the bootloader sits in a Waiting state looking for SPI input in the form of bootloader packets (SPI bootloader frames with 0xFD SPI byte). The source device then must perform a bootloader Query transaction, which involves the source sending a Query packet and expecting a Response packet. Note that the first Query transaction yields a Query- Found result (status byte 0x1A), while the subsequent Query will yield the expected Response result (status byte ‘R’ or 0x52). For de- tails, refer to the sample SPI bootloading process described in Sample EZSP-SPI Bootloader Transcript..

Once a query transaction has completed successfully, that is with expected Response frame, the transfer of data packets can begin. The transfer process follows standard XModem-CRC protocol, just like the UART-based serial bootloader uses, but with SPI framing similar to that used to encapsulate EZSP data frames. This SPI-based XModem adaptation adheres to the following rules, some of which may differ from the SPI protocol used by the normal EZSP NCP firmware:

- The 0xFD SPI byte and a length byte (for number of bytes to follow) prefix every command or response frame.
- The 0xA7 frame terminator byte concludes every SPI command or response frame. This byte is not included in the length count used in the length byte.
- The NCP operates as a SPI slave, so nSSEL must be asserted before each transaction.
- No EZSP frame control bytes are used in the SPI frame. Consequently, no sleep mode operation is supported by the target during the bootload.
- SPI timing (timeouts, signal transitions) is similar to EZSP. See *AN711: SPI Host Interfacing Guide for Zigbee* for details.
- The nHOST_INT signal is asserted by the target to indicate a pending response.
- In place of the EZSP "callbacks" command, the host should use the bootloader’s Query packet to prompt the target to push the asynchronous response such as XModem ACK back to the host.
- Each SPI frame typically generates an initial, synchronous response from the target, such as a BLOCKOK status, and a follow-up, asynchronous response, which must be queried for by the host. For example, the EOT packet generates a synchronous response with FILEDONE status, then Query transaction yields XModem ACK with block number of lastBlock+1 before rebooting into new firmware.
- The host must wait for nHOST_INT to assert (become low) before querying for status, as the SPI bootloader is edge-triggered rather than level-triggered. Thus, acting too fast on the host side can cause an edge transition to be missed and the bootloading

Copyright © 2025 Silicon Laboratories. All rights reserved.

state machines at the host and NCP to get out of synchronization, resulting in problems later on.

- SPI Status and SPI Version commands (SPI bytes 0x0A and 0x0B) are still supported.
- Prior to the first data block being processed, the SPI bus is polled at a rate of once per second.
- Once the data transmission begins (first block processed), the bootloader will wait up to 60 seconds for the next data packet, polling at 5-second intervals.
- If either of the timeouts above is exceeded, the bootloader signals a cancellation (CAN frame) and reboots, restarting the state machine.
- XModem data packets consist of:
  - The SOH byte (ASCII 0x01)
  - A 1-byte incrementing block number (beginning at 1 and wrapping back around from 255 to 0)
  - The block number's complement
  - 128 bytes of data read directly from the GBL file being uploaded
  - A 16-bit CRC of the data bytes from that packet
- Each packet is followed by an XModem ACK or NAK from the target device (the NCP running the bootloader), which confirms or refutes the current data packet.
- If the target receives a duplicate block, it simply sends the ACK for that block again. If the target receives a block that had an XModem frame error (such as bad CRC), the bootloader expects that data block to be retransmitted and then the bootload can continue. Other kinds of errors are considered unrecoverable and cause the bootload to abort.
- If the bootload process aborts for any reason (including receiving an XModem Cancel (CAN) frame from the source), an XModem Cancel frame is echoed on the SPI interface from the target and the target then reboots, restarting the bootloader state machine.
- When the source device reaches the last XModem data block, it should be padded to 128 bytes of data using SUB (ASCII 0x1A) characters.
- Once the last block is ACKed by the target, the transfer should be finalized by an EOT (ASCII 0x04) packet from the source. Once this packet is confirmed via XModem ACK from the target, the device reboots, causing the new firmware to be launched.

> Note: The ACK for the last XModem data packet may take much longer (1-3 seconds) to be received than prior data packets. This is due to the CRC32 checksum and optional GBL file signature verification being performed across the received GBL file data before sending the ACK. The source device must ensure that its SPI XModem state machine waits a sufficient amount of time to allow this checksum process to occur without timing out on the response just before the EOT is sent.

Sample EZSP-SPI Bootloader Transcript

The following is a record of the SPI frames transmitted and received by the EZSP-SPI host during the SPI bootload process, as captured from an EZSP Host application running the ota-bootload-ncp-spi.c state machine from the Silicon Labs Zigbee application framework's OTA Platform Bootloader component with an EZSP-SPI NCP device as target using the ezsp-spi-bootloader.

A "TX:" line means that the hexadecimal byte values contained in brackets ("[…]") are transmitted by the source via SPI to the target. An "RX:" line means that the byte values that follow in brackets are received by the source via SPI from the target NCP device.

> Note: Firmware data (in the form of an EBL file for an EZSP-SPI NCP) is being streamed to the host's UART (for relaying down to the target) during this process, but that serial stream is not shown here.

Comments about the process are indicated in italics and are not part of the data transmitted on the SPI bus.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Check to make sure NCP booted properly into bootloader...
TX: [0B A7]
RX: [00 09 A7]
TX: [0B A7]
RX: [C1 A7]
TX: [0A A7]
RX: [82 A7]
TX: [FD 01 51 A7] RX: [FD 01 1A A7]
TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF 00 02 02 02 20 0A A7]
TX: [FD 01 51 A7] RX: [FD 01 1A A7]
TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF 00 02 02 02 20 0A A7]
Starting SPI bootloading...
TX: [FD 01 51 A7]
RX: [FD 01 1A A7] TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF 00 02 02 02 20 0A A7]
TX: [FD 85 01 01 FE 00 00 00 3C 14 20 E2 60 48 7E AA 56 42 50 7C 53 0A
71 77 77 FF FF FF FF FF FF 78 61 70 32 62 2D 65 6D 32 36 30 2D 65 6D 32
35 30 2D 64 65 76 30 34 37 30 00 00 00 00 00 00 00 80 ED 3A AA FD 03 21
00 02 E0 00 04 57 40 00 00 3B E0 00 00 05 00 A1 E0 14 40 CA B7 E0 32
05 38 1E 05 92 6C 00 02 80 0B 54 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 01 00 A7]
TX: [FD 85 01 02 FD FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF C0 1B 48 00 00 2B 20 C8 0A F0 C0 1B 01 00 C0 C8 04 F0 00
04 01 00 91 E0 01 00 89 E0 C0 DB C0 2B FA 00 40 19 01 38 FA 00 40 29 FA
00 41 89 03 E4 FA 00 41 29 46 00 7A DB A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 02 00 A7]
TX: [FD 85 01 03 FC 18 99 75 F4 C4 27 C6 23 C8 02 CA 03 CC 2F 7A 00 3E
15 DC 27 46 00 1A 15 CE 27 00 14 D8 27 7A 00 3E 25 46 00 00 15 CE C7 D8
B7 56 F4 00 10 D6 27 02 C4 03 00 49 F0 02 B0 D6 17 80 00 00 C4 03 00 D3
F0 80 00 00 B0 D6 17 40 00 00 C4 02 00 68 F0 40 00 00 B0 D6 17 04 00 00
C4 77 F0 04 00 00 B0 D6 17 20 0C 20 0F 00 00 00 B0 D6 17 10 00 00 C4 02
00 96 F0 10 00 00 B0 D6 17 08 00 C4 1E A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 03 00 A7]
TX: [FD 85 01 04 FB 00 C4 03 00 B6 F0 08 00 00 B0 D6 17 02 00 00 C4 01
00 92 F0 02 00 00 B0 D6 17 01 00 00 C4 01 00 C3 F0 01 00 00 B0 D6 17 40
C4 01 00 38 F0 40 B0 D6 17 20 C4 01 00 F0 F0 20 B0 D6 17 10 C4 02 00 F7
F0 10 B0 D6 17 04 C4 03 00 D0 F0 D6 27 CE D7 CE 27 D6 17 19 E0 CE 17 46
00 1A 25 CC 1F DC 17 7A 00 3E 25 CA 07 C8 06 C6 13 C4 17 01 58 FA 00 40
29 C0 1B 05 EC C0 00 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 04 00 A7]
250 more data block transmissions follow; omitted here for brevity.
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
The frames following immediately below illustrate how the sequence number wraparound condition is handled...
TX: [FD 85 01 FE 01 22 17 24 57 2C C5 EA 25 20 17 24 37 EA 11 FC 3C FF
00 B1 9C 04 3C 28 3C FE E3 FE 2B F4 27 F6 23 C4 3C 1E 00 6D 9C 34 27 3E
17 FE 27 3C 17 FC 27 FE 2D FE 15 20 34 30 13 FC 3C FF 00 11 9C 40 37 2C
C5 04 3C 36 27 34 13 2C 81 03 F0 30 13 34 23 79 00 5E 19 34 3B 80 00 00
16 04 54 2C C5 38 27 00 14 FE 27 04 14 FC 27 38 17 FA 27 FE 2D FE 15 10
34 34 13 FA 3C 58 9D 3A 17 3E 13 06 3C 20 00 83 32 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 FE 00 A7]
TX: [FD 85 01 FF 00 C0 9C FE 00 C6 15 0F F4 FE 2D FE 15 20 34 FE 27 00
14 FC 27 FE 2D FE 15 10 34 04 10 FC 3C FE 00 6A 9C 04 3C FE 00 C6 15 0B
F4 FE 2D FE 15 20 34 FE 27 30 17 36 13 FE 3C FE 00 C4 9C 02 3C E2 2D E2
15 FE 27 FE 2D FE 15 20 34 FC 27 00 14 FA 27 F8 27 36 17 F6 27 32 17 F4
27 30 17 00 10 F4 3C FD 00 F9 9C 04 14 0A 27 FE 2D FE 11 0C 30 FE 2D FE
15 1C 34 0A 3C 44 9D 02 3C 00 84 03 F0 01 14 79 65 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 FF 00 A7]
TX: [FD 85 01 00 FF 02 E0 00 14 3C 3C FE E3 FE 2B 10 14 42 00 18 25 42
00 44 15 01 B4 42 00 44 25 46 00 1A 15 08 B5 46 00 1A 25 6F 00 80 14 42
00 02 25 42 00 00 25 42 00 02 15 1F 34 42 00 02 25 01 14 42 00 18 25 7E
00 D8 25 6F 00 80 5 42 00 04 25 42 00 06 15 1F 34 18 25 00 14 7E 00 DA
25 01 14 F00 DC 25 FE E3 FE 2B FC 27 00 14 94 25 FA 00 9D 93 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 00 00 A7]
TX: [FD 85 01 01 FE DC 25 41 00 2E 25 32 00 64 14 41 00 2C 25 15 14 42
00 48 25 60 00 664 41 00 40 25 58 14 41 00 42 25 03 14 41 00 44 25 00
26 25 28 14 41 00 38 25 55 14 41 00 34 25 ED 00 80 14 41 00 3A 25 09 14
41 00 3E 25 1F 14 41 00 30 25 0E 15 FA 27 0E 11 11 14 FA 3C 50 9D 7A 00
D8 15 02 3C 0C 00 4C 9C 40 14 42 00 44 25 03 14 42 00 42 25 41 00 4A 25 FA 00 0E DE A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 01 00 A7]
TX: [FD 85 01 02 FD D6 15 0D 00 B0 9C 00 84 06 F4 01 00 2D 10 03 00 24
14 46 9D FA 00 D7 11 08 A0 08 A4 E2 21 E2 15 0D 00 47 9C 01 14 41 00 46
25 18 00 00 14F 00 B2 9C FF 10 FF 14 0F 00 A8 01 A2 E_00 7C 9C 01 14
0F 00 90 9C 01 14 41 00 14 25 46 00 1A 15 06 B5 46 00 1A 25 04 00 F3 14
46 00 1E 0 1C 25 FF 00 4D 9C 00 14 04 3C F 0F 42 _B A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 02 00 A7]
826 more data block transmissions follow; omitted here for brevity. The frames following immediately below
illustrate how the last data block is padded to 128 bytes and how the transmission is concluded successfully
with the EOT and a final query transaction...
TX: [FD 85 01 3F C0 01 00 01 00 FF FF FF 00 FF 00 FF 00 AB CD C1 10 FF
00 FC 04 00 04 5A 0C BA B8 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 9F 61 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 3F 00 A7]
TX: [FD 01 04 A7]
RX: [FD 01 17 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 40 00 A7]
NCP resets back into EZSP at this point, so normal reset sequence occurs and frames begin to use 0xFE as SPI byte...
TX: [0B A7]
TX: [0B A7]
TX: [0A A7]
TX: [0B A7]
TX: [0B A7]
TX: [0A A7]
TX: [FE 04 07 00 00 02 A7]
RX: [FE 07 07 80 00 02 02 40 32 A7]
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
TX:[FD 8501 FE 01221724572C C5 EA 2520172437 EA 11 FC 3C FF
00 B1 9C 043C 283C FE E3 FE 2B F4 27 F6 23 C4 3C 1E 006D 9C 34273E
17 FE 273C 17 FC 27 FE 2D FE 1520343013 FC 3C FF 00119C 40372C
C5 043C 362734132C 8103 F0 3013342379005E 19343B 8000001604542C C5 38270014 FE 270414 FC 273817 FA 27 FE 2D FE
1510343413 FA 3C 589D 3A 173E 13063C 20008332 A7]
RX:[FD 0119 A7]
TX:[FD 0151 A7]
RX:[FD 0306 FE 00 A7]
TX:[FD 8501 FF 00 C0 9C FE 00 C6 150F F4 FE 2D FE 152034 FE 270014 FC 27 FE 2D FE 1510340410 FC 3C FE 006A 9C 043C FE
00 C6 150B
F4 FE 2D FE 152034 FE 2730173613 FE 3C FE 00 C4 9C 023C E2 2D E2
15 FE 27 FE 2D FE 152034 FC 270014 FA 27 F8 273617 F6 273217 F4
2730170010 F4 3C FD 00 F9 9C 04140A 27 FE 2D FE 110C 30 FE 2D FE
151C 340A 3C 449D 023C 008403 F0 01147965 A7]
RX:[FD 0119 A7]
TX:[FD 0151 A7]
RX:[FD 0306 FF 00 A7]
TX:[FD 850100 FF 02 E0 00143C 3C FE E3 FE 2B 1014420018254200441501 B4 4200442546001A 1508 B5 46001A 256F
00801442000225420000254200021 51F 3442000225011442001825 7E
00 D8 256F 008054200042542000 6151F 34182500147E 00 DA
250114 F00 DC 25 FE E3 FE 2B FC 2700149425 FA 009D 93 A7]
RX:[FD 0119 A7]
TX:[FD 0151 A7]
RX:[FD 03060000 A7]
TX:[FD 850101 FE DC 2541002E 253200641441002C
2515144200482560006644100402558144100422503144100442500262528144100382555144100 3425 ED 00801441003A 25091441003E 251F
14410030250E 15 FA 270E 111114 FA 3C 509D 7A 00
D8 15023C 0C 004C 9C 401442004425031442004225410 04A 25 FA 000E DE A7]
RX:[FD 0119 A7]
TX:[FD 0151 A7]
RX:[FD 03060100 A7]
TX:[FD 850102 FD D6 150D 00 B0 9C 008406 F4 01002D 1003002414469D FA 00 D7 1108 A0 08 A4 E2 21 E2 150D 00479C
01144100462518000014F 00 B2 9C FF 10 FF 140F 00 A8 01 A2 E_00 7C 9C 01140F 00909C 01144100142546001A 1506 B5 46001A
250400 F3 1446001E 01C 25 FF 004D 9C 0014043C F 0F 42 _B A7]
RX:[FD 0119 A7]
TX:[FD 0151 A7]
RX:[FD 03060200 A7]826 more data block transmissions follow; omitted here for brevity. The frames following
immediately below
illustrate how the last data block is padded to 128 bytes and how the transmission is concluded successfully
with the EOT and a final query transaction...
TX:[FD 85013F C0 01000100 FF FF FF 00 FF 00 FF 00 AB CD C1 10 FF
00 FC 0400045A 0C BA B8 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 
```

If an error occurs during the upload, the UART serial bootloader displays the message ‘Serial upload aborted,’ followed by a more detailed message and a hex error code. Some of the more common errors are shown in the following table. The UART serial bootloader then redisplays the bootloader menu. If an error occurs during the SPI serial bootload, the target produces an error response followed by an XModem Cancel frame and a reboot.

The following tables describe the normal status codes, error conditions, and special characters or enumerations used by some or all of the Ember standalone bootloader variants as well as the Gecko Bootloader. For additional status codes specific to the Gecko Bootloader, see the Gecko Bootloader API documentation installed with your SDK in theplatform/bootloader/documentation directory.

Table: Serial Uploading Statuses and Error Messages

Hex code	Constant	Description
0x00	BL_SUCCESS	Default success status.
0x01	BL_ERR	General error processing packet.
0x1C	BLOCK_TIMEOUT	The bootloader timed out waiting for some part of the XModem frame.
0x21	BLOCKERR_SOH	The bootloader did not find the expected start of header (SOH) character at the beginning of the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected the sequence check byte of the XModem frame was not the inverse of the sequence byte.
0x23	BLOCKERR_CRCH	The bootloader encountered an error while comparing the high bytes of the received and calculated CRCx of the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while comparing the low bytes of the received and calculated CRCs of the XModem frame.
0x25	BLOCKERR_SEQUENCE	The bootloader did not receive the expected sequence number in the current XModem frame.
0x26	BLOCKERR_PARTIAL	The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost).
0x27	BLOCKERR_DUPLICATE	The bootloader encountered a duplicate of the previous XModem frame.
0x40	BL_ERR_MASK	Bitmask for any bootloader error codes returned in CAN or NAK frame.
0x41	BL_ERR_HEADER_EXP	No GBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Failed to write header or CRC.
0x43	BL_ERR_CRC	File or written image failed CRC check.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in GBL image.
0x45	BL_ERR_SIG	Invalid GBL header contents.
0x46	BL_ERR_ODD_LEN	Trying to flash odd number of bytes.
0x47	BL_ERR_BLOCK_INDEX	Indexed past end of block buffer.
0x48	BL_ERR_OVWR_BL	Attempt to overwrite bootloader flash.
0x49	BL_ERR_OVWR_SIMEE	Attempt to overwrite SIMEE flash.
0x4A	BL_ERR_ERASE_FAIL	Flash erase failed.
0x4B	BL_ERR_WRITE_FAIL	Flash write failed.
0x4C	BL_ERR_CRC_LEN	End tag CRC wrong length.
0x4D	BL_ERR_NO_QUERY	Received data before query request/response.
0x4E	BL_ERR_BAD_LEN	An invalid length was detected in the upgrade image file.
0x4F	BL_ERR_TAGBUF	Insufficient tag buffer size or an invalid length was found in the GBL image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Table: Special Characters Used in Packet Types

Hex Code	Constant	Description
0x01	SOH	Start of Header.
0x03	CTRL_C	Cancel (from sender).
0x04	EOT	End of Transmission.
0x06	ACK	Acknowledged.
0x15	NAK	Not acknowledged.
0x18	CAN	Cancel
0x43	C	ASCII ‘C’.
0x51	QUERY	ASCII ‘Q’.
0x52	QRESP	ASCII ‘R’.

Table: Status Codes Returned in a Synchronous Response

Hex Code	Constant	Description
0x16	TIMEOUT	Bootloader timed out expecting characters.
0x17	FILEDONE	EOT process successfully.
0x18	FILEABORT	Transfer aborted prematurely.
0x19	BLOCKOK	Data block processed OK.
0x1A	QUERYFOUND	Successful query.

Running the Application Image

For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu. For SPI-based variants, which don’t use a menu, the application is run immediately upon ACKing the EOT frame from the source device.

Performing a Bootloader Upgrade

Bootloader upgrade functionality is provided by the first stage bootloader on Series 1 devices, or the Secure Element on Series 2 devices. On Series 2 devices, the Secure Element itself is also upgradable. On Series 1 devices, the first stage bootloader is not upgradable. For more information on how to upgrade the secure element, see *Silicon Labs Gecko Bootloader Users Guide* for your GSDK version.

To perform a bootloader upgrade with the standalone bootloaders, simply transmit two GBL files in succession. The first GBL file should contain a main bootloader upgrade image. After upload is completed, the device resets to upgrade the main bootloader. For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the first stage bootloader to perform the upgrade, before returning to the upgraded main bootloader. The second GBL file, containing an application upgrade image, can then be uploaded.

For SPI-based variants, which don’t use a menu, the upgraded bootloader is run immediately upon ACKing the EOT frame from the source device. Once the bootloader upgrade has been applied, the SPI Host loses connection to the device. An EmberZNet Host application will not be able to re-connect with the device with only a bootloader in place.

Upload Recovery

If an image upload fails, the target node is left without a valid application image. The standalone bootloader will re-enter firmware upgrade mode if it determines that the application image isn’t valid. However, the application image may have a valid structure, but contain a bug preventing normal operation. Regardless of the serial interface supported by your standalone bootloader, a GPIO-based trigger can be used to facilitate recovery via serial upload.

Copyright © 2025 Silicon Laboratories. All rights reserved.

You can configure your standalone bootloader to use a software-based GPIO pin check or other schemes of recovery mode activation such as button recovery by configuring the EZSP GPIO Activation component, or by adding custom code to the bootloader project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Application Bootloader

Using the Gecko Application Bootloader

Acquiring a New Image

The application bootloader relies on application code to obtain new code images. The bootloader itself only knows how to read a GBL image stored in the download space and copy the relevant portions to the main flash block. This approach means that the application developer is free to acquire the new code image in any way that makes sense (serial, OTA, and so on).

Typically, application developers choose to acquire the new code image over-the-air (OTA) since this is readily available on all devices. For OTA bootloading in Zigbee networks, Silicon Labs recommends using the standard OTA Upgrade cluster defined in the Zigbee Cluster Library (ZCL). Code for this cluster is available in the Silicon Labs Zigbee application framework as several different components. *AN728: Over-the-Air Bootload Server and Client Setup* walks through how this can be set up and run in SDK versions 6.10.0 and lower and *AN1384: Over-the-Air Bootload Server and Client Setup for Zigbee SDK 7.0 and Higher* provides the same information for more recent versions. For OTA bootloading in non-Zigbee networks where a ZCL-based application layer is not available, the application layer may define its own means of conveying firmware data over the networking protocol, or the application developer may define a proprietary means of accomplishing an OTA image transfer between a source device and a target.

For customers who want to design their own application to acquire an image rather than using the Silicon Labs Zigbee application framework's components, routines for interfacing with the download space are provided. These routines allow you to get information about the storage device and interact with it. Code and documentation for these routines are in the source files bootloader-interface-app.c and bootloader-interface-app.h in the platform/service/legacy_hal directory. If you do want to call these routines directly, it may be helpful to look at how the OTA Cluster Platform Bootloader component code works to ensure that these routines are used correctly. (See related files in the app/framework/plugin/ota-bootload directory of the EmberZNet PRO installation for more information.)

Performing an Application Upgrade

The application and the bootloader have limited indirect contact. Their only interaction is through passing non-volatile data across module reboots.

Once the application decides to install a new image saved in the download space it calls the Ember HAL API `halAppBootloaderInstallNewImage()`. This call indicates to the bootloader that it should attempt to perform a firmware upgrade from storage slot 0, and reboots the device. This API is backwards-compatible with the legacy Ember bootloader. If performing a firmware upgrade from a different storage slot than slot 0 is desired, the Gecko Bootloader API should be used instead, by calling `bootloader_setBootloadList()`. If the bootloader fails to install the new image, it sets the reset cause to `RESET_BOOTLOADER_BADIMAGE` and resets the module. Upon startup, the application should read the reset cause with `halGetExtendedResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_BADIMAGE,` the application knows the install process failed and can attempt to obtain a new image. A printable error string can be acquired by calling `halGetExtendedResetString()`. Under normal circumstances, the application bootloader does not print anything on the serial line.

Example Application Bootload

For details on how to set up and run an application bootload, see *AN1384: Over-the-Air Bootload Server and Client Setup for Zigbee SDK 7.0 and Higher*. If you are using an earlier version, see *AN728: Over-the-Air Bootload Server and Client Setup*.

External Storage Application Bootloader

See the *Silicon Labs Gecko Bootloader User's Guide* for your GSDK version for information about memory configuration for external storage bootloaders. Note that, at the time of this writing, the OTA Simple Storage Module component code in the EmberZNet stack only supports storage configurations with a single storage slot. The start address of the storage slot in the bootloader needs to be configured to the same value as the OTA Storage Start Offset of the OTA Simple Storage EEPROM Driver

Copyright © 2025 Silicon Laboratories. All rights reserved.

component in the Silicon Labs Zigbee application framework. The application developer is responsible for ensuring that this value is consistent across the stack and bootloader projects. Also, keep in mind that the start address of the first storage slot in the bootloader should be offset from the beginning of the storage area by two times the page size of the underlying storage medium if more than one storage slot is configured.

Application bootloaders typically use a remote device to store the downloaded application image. This device can be accessed over either an I2C or SPI serial interface. Refer to the *Silicon Labs Gecko Bootloader User's Guide* for your GSDK version for a list of supported dataflash/EEPROM devices. It is important to select a device whose size is at least the size of your flash in order to fit the application image being used by the bootloader.

The default recommendation for external SPI serial flash is the MX25R, as this is available in standard and smaller packages, is supported by standard drivers, and has very low software-enabled sleep current without the need for an external shutdown control circuit. Most radio boards from Silicon Labs are populated with the MX25R8035F for evaluation purposes. In general, customers should use parts that have low sleep current, don't require external shutdown control circuitry, and have software shutdown control. When using components with high idle/sleep current and no software shutdown control, an external shutdown control circuit is recommended to reduce sleep current.

For EFR32MG-based devices using the EFR32MG1, only the serial dataflash option is available; no local storage application bootloader is presently offered. However, some EFR32MG ICs contain an integrated serial flash that can be utilized just like off-chip serial dataflash but without any additional components. EFR32MG-based devices using the EFR32MG12 and higher platforms support local storage application bootloaders.

Note that some of these chips have compatible pinouts with others, but there are several incompatible variations. Contact Silicon Labs for details on connecting I2C or other SPI dataflash chips to an EFR32.

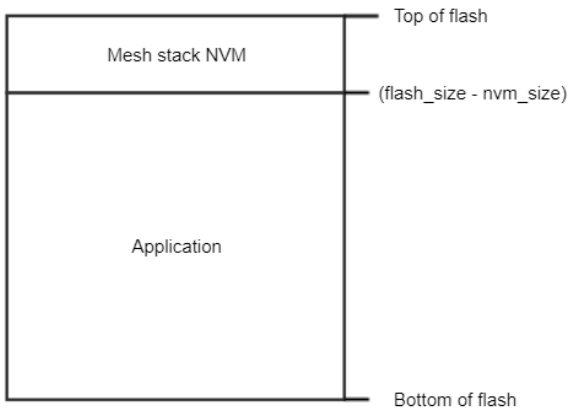
Read-Modify-Write pertains to a feature of certain dataflash chips that their corresponding driver exposes, and that is exploited by the bootloader library. Chips without this feature require a page erase to be performed before writing to that page, which precludes random-access writes by an application. When using the Silicon Labs Zigbee application framework, the OTA Simple Storage EEPROM Driver component needs to be configured to take this into consideration.

Local Storage Application Bootloader

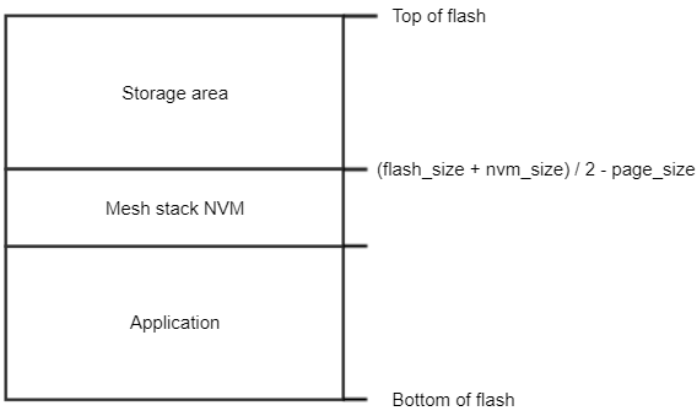
The local storage bootloader is essentially an application bootloader with a dataflash driver that uses a portion of the on-chip flash for image storage instead of an external storage chip. See the *Silicon Labs Gecko Bootloader User's Guide* for your GSDK version for information about memory configuration for internal storage. Note that the OTA Simple Storage Module component code in the EmberZNet stack only supports storage configurations with a single storage slot at the time of this writing. The start address of the storage slot in the bootloader needs to be configured to the same value as the OTA Storage Start Offset of the OTA Simple Storage EEPROM component in the Silicon Labs Zigbee application framework. The application developer is responsible for ensuring that this value is consistent across the stack and bootloader projects. Also, keep in mind that the start address of the first storage slot in the bootloader should be offset from the beginning of the storage area by two times the page size of the underlying storage medium if more than one storage slot is configured.

Since the local storage application bootloader changes the chip's flash memory layout you must build your application with knowledge of this. To accomplish this, the LOCAL_STORAGE_BTL global define must be in your project file. In Ember Zigbee SDK v6.10.x and lower, this is done for you when you select Local Storage from AppBuilder's bootloader dropdown. In SDK v7.0 and higher, local storage is set as the default.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Main flash layout using Application Bootloader (external SPI flash storage)



Main flash layout using Local Storage Bootloader (internal flash storage)

> Note: On EFR32 Series 2 devices, the application is offset by 16 kB to accommodate the bootloader at the bottom of main flash. On other Series 1 EFR32 devices, the Gecko Bootloader resides in the bootloader flash region outside of main flash block.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader API Reference.

Silicon Labs Gecko Bootloader

Introduction

The Silicon Labs Gecko Bootloader is a common bootloader for all newer Silicon Labs MCUs and wireless MCUs (WMCUs). Key features of the bootloader are:

- Universal bootloader across MCU families (MCU and WMCU)
- Supports image verification and encryption for
  - Integrity
  - Authenticity
  - Confidentiality
- In-field upgradeable
- Configurable

This document describes the requirements and the usage of the Gecko Bootloader.

Device Compatibility

The Silicon Labs Gecko Bootloader is a customizable bootloader compatible with the following devices:

- EFM32 and EFR32 Series 1 Devices
- EFM32 and EFR32 Series 2 Devices

The Gecko bootloader has a two-stage design. The first stage enables upgrading the main bootloader, which enables in-field bootloader upgrades.

Requirements

Flash Size

See UG103.6 Bootloader Fundamentals, section '3. Memory Space For Bootloading' for bootloader flash size requirements.

Capabilities

ECDSA signature verification of the application SHA-256 digest is supported for integrity and security reasons. As a result, the bootloader can only run on devices that support hardware acceleration of these operations. The bootloader is therefore only compatible with the EFM32 and EFR32 Series 1 devices and EFM32 and EFR32 Series 2 devices.

Application Interface

The bootloader includes an application interface, which is exposed through a function table in the bootloader. The application interface provides APIs to store and retrieve upgrade images, verify the integrity of the images, and so on.

To use the application interface, include the `api/btl_interface.h` header in your application and add the following files to the build:

- `api/btl_interface.c`
- `api/btl_interface_storage.c`

Copyright © 2025 Silicon Laboratories. All rights reserved.

See the Application Interface documentation for more information.

Bootloader Components

The bootloader itself consists of the following parts:

Core

The bootloader core contains the main function of both bootloader stages. It also contains functionality to write to the internal main Flash, to perform a bootloader upgrade, and to reset into the application flagging applicable reset reasons. See the Core Documentation for more information.

Driver

Different bootloading applications require different hardware drivers for use by the other components of the bootloader. See the Driver Documentation for more information.

Components

All parts of the bootloader that are either optional or swappable for different implementations are implemented as components. Each component has a generic header file and one or more implementations. The current release contains components for functionality, such as UART and SPI communication protocols, SPI Flash storage, internal Flash storage, and different cryptographic operations. For more information about different components, see the Components Documentation.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Interface

Application Interface

Application interface to the bootloader.

The application interface consists of functions that can be included in the customer application that and will communicate with the bootloader through the MainBootloaderTable_t. This table contains function pointers to the bootloader. The 10th word of the bootloader contains a pointer to this struct, allowing any application to easily locate it. To access the bootloader table, use wrapper functions. Avoid accessing the bootloader table directly.

Modules

Application Parser Interface

Application Properties

Application Storage Interface

Common Application Interface

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Parser Interface

Application Parser Interface

Application interface for the bootloader image parser.

The Parser Interface can be used to parse upgrade images from the context of the application.

Modules

BootloaderParserCallbacks_t

Typedefs

typedef void(* BootloaderParserCallback_t)(uint32_t address, uint8_t *data, size_t length, void *context)

Bootloader parser callback.

typedef void(* BootloaderParserCallback_t)(uint32_t address, uint8_t *data, size_t length, unsigned int region_idx, void *context)

Bootloader parser callback.

Functions

int32_t bootloader_initParser(BootloaderParserContext_t *context, size_t contextSize)

Initialize the image parser.

int32_t bootloader_parseBuffer(BootloaderParserContext_t *context, BootloaderParserCallbacks_t *callbacks, uint8_t data[], size_t numBytes)

Parse a buffer.

int32_t bootloader_parseImageInfo(BootloaderParserContext_t *context, uint8_t data[], size_t numBytes, ApplicationData_t *appInfo, uint32_t *bootloaderVersion)

Parse a buffer and get application and bootloader upgrade metadata from the buffer.

uint32_t bootloader_parserContextSize(void)

Find the size of the context struct BootloaderParserContext used by the bootloader image parser to store parser state.

Typedef Documentation

BootloaderParserCallback_t

```
typedef void(* BootloaderParserCallback_t) (uint32_t address, uint8_t *data, size_t length,
void *context) )(uint32_t address, uint8_t *data, size_t length, void *context)
```

Bootloader parser callback.

Parameters

Type	Direction	Argument Name	Description
	N/A	address	Address of the data
	N/A	data	Raw data
	N/A	length	Size in bytes of raw data.
	N/A	context	A context variable defined by the implementation that is implementing this callback.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderParserCallback_t

```
typedef void(* BootloaderParserCallback_t) (uint32_t address, uint8_t *data, size_t length,
unsigned int region_idx, void *context) )(uint32_t address, uint8_t *data, size_t length,
unsigned int region_idx, void *context)
```

Bootloader parser callback.

Parameters

Type	Direction	Argument Name	Description
	N/A	`address`	Address of the data
	N/A	`data`	Raw data
	N/A	`length`	Size in bytes of raw data.
	N/A	`region_idx`	ID of region
	N/A	`context`	A context variable defined by the implementation that is implementing this callback.

Function Documentation

bootloader_initParser

```
int32_t bootloader_initParser (BootloaderParserContext_t * context, size_t contextSize)
```

Initialize the image parser.

Parameters

Type	Direction	Argument Name	Description
BootloaderParserContext_t *	[in]	`context`	Pointer to the parser context struct.
size_t	[in]	`contextSize`	Size of the context struct.

Returns

- BOOTLOADER_OK if success, BOOTLOADER_ERROR_PARSE_CONTEXT if context struct is too small.

bootloader_parseBuffer

```
int32_t bootloader_parseBuffer (BootloaderParserContext_t * context,
BootloaderParserCallbacks_t * callbacks, uint8_t data, size_t numBytes)
```

Parse a buffer.

Parameters

Type	Direction	Argument Name	Description
BootloaderParserContext_t *	[in]	`context`	Pointer to the parser context struct.
BootloaderParserCallbacks_t *	[in]	`callbacks`	Callbacks to be called by the parser.
uint8_t	[in]	`data`	Data to be parsed.
size_t	[in]	`numBytes`	Size of the data buffer.

Returns

Copyright © 2025 Silicon Laboratories. All rights reserved.

BOOTLOADER_ERROR_PARSE_CONTINUE if the chunk was parsed correctly, and a new chunk is expected. BOOTLOADER_ERROR_PARSE_ERROR if something went wrong while parsing. BOOTLOADER_ERROR_PARSE_SUCCESS if the entire file was successfully parsed.

bootloader_parseImageInfo

```
int32_t bootloader_parseImageInfo (BootloaderParserContext_t * context, uint8_t data, size_t
numBytes, ApplicationData_t * appInfo, uint32_t * bootloaderVersion)
```

Parse a buffer and get application and bootloader upgrade metadata from the buffer.

Parameters

Type	Direction	Argument Name	Description
BootloaderParserContext_t *	[in]	`context`	Pointer to the parser context struct.
uint8_t	[in]	`data`	Data to be parsed.
size_t	[in]	`numBytes`	Size of the data buffer.
ApplicationData_t *	[out]	`appInfo`	Pointer to ApplicationData_t struct.
uint32_t *	[out]	`bootloaderVersion`	Pointer to an integer representing bootloader version.

Note

- `appInfo` and `bootloaderVersion` will default to zeros.

Returns

- BOOTLOADER_OK if metadata was filled successfully.

bootloader_parserContextSize

```
uint32_t bootloader_parserContextSize (void )
```

Find the size of the context struct BootloaderParserContext used by the bootloader image parser to store parser state.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- size of BootloaderParserContext, returns 0 if something went wrong.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderParserCallbacks_t

Function pointers to parser callbacks.

Public Attributes

void *	context Opaque pointer passed to the callback functions.
BootloaderParserCallback_t	applicationCallback Callback function pointer for application image data.
BootloaderParserCallback_t	metadataCallback Callback function pointer for image metadata.
BootloaderParserCallback_t	bootloaderCallback Callback function pointer for bootloader upgrade image data.

Public Attribute Documentation

context

```
void * BootloaderParserCallbacks_t::context
```

Opaque pointer passed to the callback functions.

applicationCallback

```
BootloaderParserCallback_t BootloaderParserCallbacks_t::applicationCallback
```

Callback function pointer for application image data.

metadataCallback

```
BootloaderParserCallback_t BootloaderParserCallbacks_t::metadataCallback
```

Callback function pointer for image metadata.

bootloaderCallback

```
BootloaderParserCallback_t BootloaderParserCallbacks_t::bootloaderCallback
```

Callback function pointer for bootloader upgrade image data.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Properties

Application Properties

Properties of the application that can be accessed by the bootloader.

Applications must contain an ApplicationProperties_t struct declaring the application version and capabilities, and so on. The metadata contained in this struct will be extracted from the application by the Simplicity Commander tool and placed in the GBL upgrade file. If this struct is not in the application image, it will be added to the GBL file by the Simplicity Commander.

The struct is also used to declare whether the application image is signed and what type of signature is used. If no ApplicationProperties_t struct is present, the bootloader will assume that the application image is signed using APPLICATION_SIGNATURE_ECDSA_P256.

To ensure that the bootloader can easily locate the ApplicationProperties_t struct, if not already done by the linker, Simplicity Commander will modify word 13 of the application to insert a pointer to the ApplicationProperties_t struct.

Modules

ApplicationData_t

ApplicationCertificate_t

ApplicationProperties_t

Macros

#define APPLICATION_PROPERTIES_MAGIC undefined
Magic value declaring the existence of an ApplicationProperties_t struct.

#define APPLICATION_PROPERTIES_REVERSED undefined
Byte-reversed version of APPLICATION_PROPERTIES_MAGIC.

#define APPLICATION_PROPERTIES_VERSION_MAJOR (1UL)
Major version number of the AppliationProperties_t struct.

#define APPLICATION_PROPERTIES_VERSION_MINOR (2UL)
Minor version number of the AppliationProperties_t struct.

#define APPLICATION_CERTIFICATE_VERSION (1UL)
Version number of the ApplicationCertificate_t struct.

#define APPLICATION_SIGNATURE_NONE (0UL)
The application is not signed.

#define APPLICATION_SIGNATURE_ECDSA_P256 (1UL << 0UL)
The SHA-256 digest of the application is signed using ECDSA with the NIST P-256 curve.

#define APPLICATION_SIGNATURE_CRC32 (1UL << 1UL)
The application is not signed, but has a CRC-32 checksum.

#define APPLICATION_TYPE_ZIGBEE (1UL << 0UL)
The application contains a Zigbee wireless stack.

#define APPLICATION_TYPE_THREAD (1UL << 1UL)
The application contains a Thread wireless stack.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define APPLICATION_TYPE_FLEX (1UL << 2UL)
The application contains a Flex wireless stack.

#define APPLICATION_TYPE_BLUETOOTH (1UL << 3UL)
The application contains a Bluetooth wireless stack.

#define APPLICATION_TYPE_MCU (1UL << 4UL)
The application is an MCU application.

#define APPLICATION_TYPE_BLUETOOTH_APP (1UL << 5UL)
The application contains a Bluetooth application.

#define APPLICATION_TYPE_BOOTLOADER (1UL << 6UL)
The application contains a bootloader.

#define APPLICATION_TYPE_ZWAVE (1UL << 7UL)
The application contains a Zwave wireless stack.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ApplicationData_t

Application Data.

Public Attributes

uint32_t	type Bitfield representing type of application, e.g., APPLICATION_TYPE_ZIGBEE.
uint32_t	version Version number for this application.
uint32_t	capabilities Capabilities of this application.
uint8_t	productId Unique ID (UUID or GUID) for the product this application is built for.

Public Attribute Documentation

type

```
uint32_t ApplicationData_t::type
```

Bitfield representing type of application, e.g., APPLICATION_TYPE_ZIGBEE.

version

```
uint32_t ApplicationData_t::version
```

Version number for this application.

capabilities

```
uint32_t ApplicationData_t::capabilities
```

Capabilities of this application.

productId

```
uint8_t ApplicationData_t::productId[16]
```

Unique ID (UUID or GUID) for the product this application is built for.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ApplicationCertificate_t

Application Certificate.

Public Attributes

uint8_t	structVersion Version of the certificate structure.
uint8_t	flags Reserved flags.
uint8_t	key Public key.
uint32_t	version The version number of this certificate.
uint8_t	signature Signature of the certificate.

Public Attribute Documentation

structVersion

```
uint8_t ApplicationCertificate_t::structVersion
```

Version of the certificate structure.

flags

```
uint8_t ApplicationCertificate_t::flags[3]
```

Reserved flags.

key

```
uint8_t ApplicationCertificate_t::key[64]
```

Public key.

version

```
uint32_t ApplicationCertificate_t::version
```

The version number of this certificate.

signature

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint8_t ApplicationCertificate_t::signature[64]
```

Signature of the certificate.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ApplicationProperties_t

Application Properties struct.

Public Attributes

uint8_t	magic Magic value indicating this is an ApplicationProperties_t struct.
uint32_t	structVersion Version number of this struct.
uint32_t	signatureType Type of signature this application is signed with.
uint32_t	signatureLocation Location of the signature. Typically points to the end of the application.
ApplicationData_t	app Information about the application.
ApplicationCertificate_t *	cert Pointer to information about the certificate.
uint8_t *	longTokenSectionAddress Pointer to Long Token Data Section.
const uint8_t	decryptKey Parser Decryption Key.

Public Attribute Documentation

magic

```
uint8_t ApplicationProperties_t::magic[16]
```

Magic value indicating this is an ApplicationProperties_t struct.

Must equal APPLICATION_PROPERTIES_MAGIC

structVersion

```
uint32_t ApplicationProperties_t::structVersion
```

Version number of this struct.

signatureType

```
uint32_t ApplicationProperties_t::signatureType
```

Type of signature this application is signed with.

Copyright © 2025 Silicon Laboratories. All rights reserved.

signatureLocation

```
uint32_t ApplicationProperties_t::signatureLocation
```

Location of the signature. Typically points to the end of the application.

app

```
ApplicationData_t ApplicationProperties_t::app
```

Information about the application.

cert

```
ApplicationCertificate_t* ApplicationProperties_t::cert
```

Pointer to information about the certificate.

longTokenSectionAddress

```
uint8_t* ApplicationProperties_t::longTokenSectionAddress
```

Pointer to Long Token Data Section.

decryptKey

```
const uint8_t ApplicationProperties_t::decryptKey[16]
```

Parser Decryption Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Storage Interface

Application Storage Interface

Application interface for interfacing with the bootloader storage.

Note

- These Bootloader APIs are not reentrant and should be wrapped in critical section where needed.

The Storage Interface is only available on bootloaders that declare they support BOOTLOADER_CAPABILITY_STORAGE.

- Example

Example

Snippet for the OTA use case:

```
* OTA Example
* Assuming the user has an upgrade image downloaded which will be used to upgrade the application
*
* Initialize the bootloader interface
* bootloader_init();
*
* Erase the storage slot in internal/SPI flash memory
* bootloader_eraseStorageSlot(0);
*
* Write the upgrade image (GBL file data) to the slot. blinkGbl   uint8 array holding the GBL data in memory
* bootloader_writeStorage(0, 0, blinkGbl, sizeof(blinkGbl));
*
* Reboot into the bootloader to install the new image
* bootloader_rebootAndInstall();
*
* The general flow for bootloader interface APIs from the application is:
*
* General flow
*
* Initialize the bootloader interface
* bootloader_init();
*
* Interface API accesses
* ..............
* ..............
* ..............
*
* De-initialize the bootloader interface
* bootloader_deinit();
*
```

Modules

BootloaderStorageSlot_t

BootloaderStorageImplementationInformation_t

BootloaderStorageInformation_t

BootloaderEraseStatus_t

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageFunctions_t

Enumerations

enum BootloaderStorageType_t {

SPIFLASH
INTERNAL_FLASH
CUSTOM_STORAGE
EXTERNAL_FLASH
INTERNAL_FLASH
CUSTOM_STORAGE

}

Possible storage types.

enum BootloaderStorageType_t {

SPIFLASH
INTERNAL_FLASH
CUSTOM_STORAGE
EXTERNAL_FLASH
INTERNAL_FLASH
CUSTOM_STORAGE

}

Possible storage types.

Functions

void bootloader_getStorageInfo(BootloaderStorageInformation_t *info)

Get information about the storage component.

int32_t bootloader_getStorageSlotInfo(uint32_t slotId, BootloaderStorageSlot_t *slot)

Get information about a storage slot.

int32_t bootloader_readStorage(uint32_t slotId, uint32_t offset, uint8_t *buffer, size_t length)

Read data from a storage slot.

int32_t bootloader_writeStorage(uint32_t slotId, uint32_t offset, uint8_t *buffer, size_t length)

Write data to a storage slot.

int32_t bootloader_eraseWriteStorage(uint32_t slotId, uint32_t offset, uint8_t *buffer, size_t length)

Erase and write data to a storage slot.

int32_t bootloader_eraseStorageSlot(uint32_t slotId)

Erase all contents of a storage slot.

int32_t bootloader_initChunkedEraseStorageSlot(uint32_t slotId, BootloaderEraseStatus_t *eraseStat)

Initialize chunked erase of a storage slot.

int32_t bootloader_chunkedEraseStorageSlot(BootloaderEraseStatus_t *eraseStat)

Erase one page from a storage slot according to the struct BootloaderEraseStatus_t.

int32_t bootloader_setImagesToBootload(int32_t *slotIds, size_t length)

Set a prioritized list of images to attempt to bootload.

int32_t bootloader_getImagesToBootload(int32_t *slotIds, size_t length)

Get the prioritized list of images the bootloader will attempt to bootload.

int32_t bootloader_appendImageToBootloadList(int32_t slotId)

Append a single image to the list of images to attempt to bootload.

Copyright © 2025 Silicon Laboratories. All rights reserved.

int32_t bootloader_setImageToBootload(int32_t slotId)
Set a single image to attempt to bootload.

int32_t bootloader_initVerifyImage(uint32_t slotId, void *context, size_t contextSize)
Initialize image verification.

int32_t bootloader_continueVerifyImage(void *context, BootloaderParserCallback_t metadataCallback)
Continue image verification.

int32_t bootloader_verifyImage(uint32_t slotId, BootloaderParserCallback_t metadataCallback)
Verify that the image in the given storage slot is valid.

int32_t bootloader_getImageInfo(uint32_t slotId, ApplicationData_t *appInfo, uint32_t *bootloaderVersion)
Get application and bootloader upgrade metadata from the storage slot.

bool bootloader_storageIsBusy(void)
Check whether the bootloader storage is busy.

int32_t bootloader_readRawStorage(uint32_t address, uint8_t *buffer, size_t length)
Read raw data from storage.

int32_t bootloader_writeRawStorage(uint32_t address, uint8_t *buffer, size_t length)
Write data to storage.

int32_t bootloader_eraseRawStorage(uint32_t address, size_t length)
Erase data from storage.

int32_t bootloader_getAllocatedDMAChannel(void)
Get allocated DMA channel for MSC write.

void bootloader_getAppMinVersion(uint32_t *version)
Get application minimum version.

Macros

#define BOOTLOADER_STORAGE_VERIFICATION_CONTEXT_SIZE (384)
Context size for bootloader verification context.

#define BOOTLOADER_STORAGE_INFO_VERSION (0x30000U)
Current version of the BootloaderStorageInformation_t struct.

#define BOOTLOADER_STORAGE_IMPL_INFO_VERSION (0x0210U)
Current version of the BootloaderStorageImplementationInformation_t struct.

#define BOOTLOADER_STORAGE_IMPL_INFO_VERSION_MAJOR (0x0200U)
Major version of the BootloaderStorageImplementationInformation_t struct.

#define BOOTLOADER_STORAGE_IMPL_INFO_VERSION_MAJOR_MASK (0xFF00U)
Major version mask for BOOTLOADER_STORAGE_IMPL_INFO_VERSION.

#define BOOTLOADER_STORAGE_IMPL_CAPABILITY_ERASE_SUPPORTED (1 << 0)
Spiflash capability indicating that it supports erase.

#define BOOTLOADER_STORAGE_IMPL_CAPABILITY_PAGE_ERASE_REQUIRED (1 << 1)
Spiflash capability indicating it requires full page erases before new data can be written.

#define BOOTLOADER_STORAGE_IMPL_CAPABILITY_BLOCKING_WRITE (1 << 2)
Spiflash capability indicating that the write function is blocking.

#define BOOTLOADER_STORAGE_IMPL_CAPABILITY_BLOCKING_ERASE (1 << 3)
Spiflash capability indicating that the erase function is blocking.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#define BOOTLOADER_STORAGE_ISSI_IS25LQ040B (1U << 0)
```
ISSI IS25LQ040B SPI Flash.

```
#define BOOTLOADER_STORAGE_ISSI_IS25LQ020B (1U << 1)
```
ISSI IS25LQ020B SPI Flash.

```
#define BOOTLOADER_STORAGE_ISSI_IS25LQ010B (1U << 2)
```
ISSI IS25LQ010B SPI Flash.

```
#define BOOTLOADER_STORAGE_ISSI_IS25LQ512B (1U << 3)
```
ISSI IS25LQ512B SPI Flash.

```
#define BOOTLOADER_STORAGE_ISSI_IS25LQ025B (1U << 4)
```
ISSI IS25LQ025B SPI Flash.

```
#define BOOTLOADER_STORAGE_NUMONYX_M25P16 (1U << 5)
```
Numonyx M25P16 SPI Flash.

```
#define BOOTLOADER_STORAGE_NUMONYX_M25P80 (1U << 6)
```
Numonyx M25P80 SPI Flash.

```
#define BOOTLOADER_STORAGE_NUMONYX_M25P40 (1U << 7)
```
Numonyx M25P40 SPI Flash.

```
#define BOOTLOADER_STORAGE_NUMONYX_M25P20 (1U << 8)
```
Numonyx M25P20 SPI Flash.

```
#define BOOTLOADER_STORAGE_ADESTO_AT25SF041 (1U << 9)
```
Adesto AT25SF041 SPI Flash.

```
#define BOOTLOADER_STORAGE_ATMEL_AT25DF081A (1U << 10)
```
Atmel AT25DF081A SPI Flash.

```
#define BOOTLOADER_STORAGE_ATMEL_AT25DF041A (1U << 11)
```
Atmel AT25DF041A SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25R6435F (1U << 12)
```
Macronix MX25R6435F SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25R3235F (1U << 13)
```
Macronix MX25R6435F SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25U1635E (1U << 14)
```
Macronix MX25U1635E SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25L1606E (1U << 15)
```
Macronix MX25L1606E SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25R8035F (1U << 16)
```
Macronix MX25R8035F SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25L8006E (1U << 17)
```
Macronix MX25L8006E SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25L4006E (1U << 18)
```
Macronix MX25L4006E SPI Flash.

```
#define BOOTLOADER_STORAGE_MACRONIX_MX25L2006E (1U << 19)
```
Macronix MX25L2006E SPI Flash.

```
#define BOOTLOADER_STORAGE_WINBOND_W25Q80BV (1U << 20)
```
Winbond W25Q80BV SPI Flash.

```
#define BOOTLOADER_STORAGE_WINBOND_W25X20BV (1U << 21)
```
Winbond W25X20BV SPI Flash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define BOOTLOADER_STORAGE_SPANSION_S25FL208K (1U << 22)
Spansion S25L208K SPI Flash.

#define BOOTLOADER_STORAGE_INTERNAL_STORAGE (1U << 30)
Internal storage.

#define BOOTLOADER_STORAGE_JEDEC (1U << 31)
JEDEC Supported SPI Flash.

#define BOOTLOADER_STORAGE_VERIFICATION_CONTEXT_SIZE (3000)
Context size for bootloader verification context.

#define BOOTLOADER_STORAGE_INFO_VERSION (0x30000U)
Current version of the BootloaderStorageInformation_t struct.

Enumeration Documentation

BootloaderStorageType_t

```
BootloaderStorageType_t
```

Possible storage types.

Enumerator	
SPIFLASH	Storage backend is a SPI flash.
INTERNAL_FLASH	Storage backend is internal flash.
CUSTOM_STORAGE	Storage backend is custom.
EXTERNAL_FLASH	Storage backend is a SPI flash.
INTERNAL_FLASH	Storage backend is internal flash.
CUSTOM_STORAGE	Storage backend is custom.

BootloaderStorageType_t

```
BootloaderStorageType_t
```

Possible storage types.

Enumerator	
SPIFLASH	Storage backend is a SPI flash.
INTERNAL_FLASH	Storage backend is internal flash.
CUSTOM_STORAGE	Storage backend is custom.
EXTERNAL_FLASH	Storage backend is a SPI flash.
INTERNAL_FLASH	Storage backend is internal flash.
CUSTOM_STORAGE	Storage backend is custom.

Function Documentation

bootloader_getStorageInfo

```
void bootloader_getStorageInfo (BootloaderStorageInformation_t * info)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Get information about the storage component.

Parameters

Type	Direction	Argument Name	Description
BootloaderStorageInformation_t *	[out]	info	Information about the storage component.

bootloader_getStorageSlotInfo

```
int32_t bootloader_getStorageSlotInfo (uint32_t slotId, BootloaderStorageSlot_t * slot)
```

Get information about a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	slotId	ID of the slot to get information about
BootloaderStorageSlot_t *	[out]	slot	Information about the storage slot

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_readStorage

```
int32_t bootloader_readStorage (uint32_t slotId, uint32_t offset, uint8_t * buffer, size_t
length)
```

Read data from a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	slotId	ID of the slot
uint32_t	[in]	offset	Offset into the slot to start reading from
uint8_t *	[out]	buffer	Buffer to store the data
size_t	[in]	length	Amount of data to read

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_writeStorage

```
int32_t bootloader_writeStorage (uint32_t slotId, uint32_t offset, uint8_t * buffer, size_t
length)
```

Write data to a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	slotId	ID of the slot

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint32_t	[in]	`offset`	Offset into the slot to start writing to
uint8_t *	[in]	`buffer`	Buffer to read data to write from
size_t	[in]	`length`	Amount of data to write

Note

- If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices. DMA-based MSC write is enabled, both offset and buffer should be word aligned. In case the buffer is not aligned, the normal write procedure is used instead of DMA.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

Note

- If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices. DMA-based MSC write is enabled, both offset and buffer should be word aligned. In case the buffer is not aligned, the normal write procedure is used instead of DMA.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_eraseWriteStorage

```
int32_t bootloader_eraseWriteStorage (uint32_t slotId, uint32_t offset, uint8_t * buffer,
size_t length)
```

Erase and write data to a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	ID of the slot
uint32_t	[in]	`offset`	Offset into the slot to start writing to
uint8_t *	[in]	`buffer`	Buffer to read data to write from
size_t	[in]	`length`	Amount of data to write

Note

- This function automatically erases the following Flash page whenever the written data crosses a page boundary. In other words, the function can't be used to perform multiple sequential writes to the same address range unless the range starts at a page boundary. For a sequential write, the first call to this function should have a start address at a page boundary. Otherwise, the corresponding page of the starting address needs to be erased explicitly. If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

Note

- This function automatically erases the following Flash page whenever the written data crosses a page boundary. In other words, the function can't be used to perform multiple sequential writes to the same address range unless the range starts at a page boundary. For a sequential write, the first call to this function should have a start address at a page boundary. Otherwise, the corresponding page of the starting address needs to be erased explicitly. If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_eraseStorageSlot

```
int32_t bootloader_eraseStorageSlot (uint32_t slotId)
```

Erase all contents of a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	slotId	ID of the slot

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_initChunkedEraseStorageSlot

```
int32_t bootloader_initChunkedEraseStorageSlot (uint32_t slotId, BootloaderEraseStatus_t *
eraseStat)
```

Initialize chunked erase of a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	slotId	ID of the slot
BootloaderEraseStatus_t *	[in]	eraseStat	Erase status struct

Note

- This function must be called before calling bootloader_chunkedEraseStorageSlot in a loop.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_chunkedEraseStorageSlot

```
int32_t bootloader_chunkedEraseStorageSlot (BootloaderEraseStatus_t * eraseStat)
```

Erase one page from a storage slot according to the struct BootloaderEraseStatus_t.

Parameters

Type	Direction	Argument Name	Description
BootloaderEraseStatus_t *	[in]	eraseStat	Erase status struct

Note

- bootloader_initChunkedEraseStorageSlot must be called before calling this function, in order to prepare BootloaderEraseStatus_t.
- This can be called sequentially to, for example, erase all contents of a storage slot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- BOOTLOADER_ERROR_STORAGE_CONTINUE if erasing a page was successful. Erase can be continued by calling this function again. BOOTLOADER_OK if the entire slot has been erased, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

Note

- bootloader_initChunkedEraseStorageSlot must be called before calling this function, in order to prepare BootloaderEraseStatus_t.
- This can be called sequentially to, for example, erase all contents of a storage slot.

Returns

- SL_STATUS_BL_STORAGE_CONTINUE if erasing a page was successful. Erase can be continued by calling this function again. BOOTLOADER_OK if the entire slot has been erased, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_setImagesToBootload

```
int32_t bootloader_setImagesToBootload (int32_t * slotIds, size_t length)
```

Set a prioritized list of images to attempt to bootload.

Parameters

Type	Direction	Argument Name	Description
int32_t *	[in]	`slotIds`	Prioritized list of slot IDs to attempt to bootload. The first image to pass verification will be bootloaded.
size_t	[in]	`length`	Length of the slotIds array

The last call to this function determines which slot will be installed when bootloader_rebootAndInstall is called.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_BOOTLOAD_BASE range

bootloader_getImagesToBootload

```
int32_t bootloader_getImagesToBootload (int32_t * slotIds, size_t length)
```

Get the prioritized list of images the bootloader will attempt to bootload.

Parameters

Type	Direction	Argument Name	Description
int32_t *	[out]	`slotIds`	Prioritized list of slot IDs to attempt to bootload. The first image to pass verification will be bootloaded.
size_t	[in]	`length`	Length of the slotIds array

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_BOOTLOAD_BASE range

bootloader_appendImageToBootloadList

```
int32_t bootloader_appendImageToBootloadList (int32_t slotId)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Append a single image to the list of images to attempt to bootload.

Parameters

Type	Direction	Argument Name	Description
int32_t	[in]	`slotId`	ID of the slot to append

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_BOOTLOAD_BASE range

bootloader_setImageToBootload

```
int32_t bootloader_setImageToBootload (int32_t slotId)
```

Set a single image to attempt to bootload.

Parameters

Type	Direction	Argument Name	Description
int32_t	[in]	`slotId`	ID of the slot

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_BOOTLOAD_BASE range

bootloader_initVerifyImage

```
int32_t bootloader_initVerifyImage (uint32_t slotId, void * context, size_t contextSize)
```

Initialize image verification.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	ID of the slot to check.
void *	N/A	`context`	Pointer to memory used to hold the parser context.
size_t	[in]	`contextSize`	Size of the context. An error is returned if the supplied context is too small.

Initialize verification of an upgrade image stored in a bootloader storage slot.

Note

- This function must be called before calling bootloader_continueVerifyImage in a loop.
- The context pointer must point to memory allocated by the caller. The caller must ensure that the context pointer is 32 bit aligned. The required size of this context may depend on the version of the bootloader. The required size for the bootloader associated with this version of the application interface is given by the define BOOTLOADER_STORAGE_VERIFICATION_CONTEXT_SIZE.
- Instead of calling bootloader_initVerifyImage followed by bootloader_continueVerifyImage, call bootloader_verifyImage if no time-critical tasks are needed and sufficient stack space is available for the automatically allocated context. The purpose of the init-and-continue functions is to allow the caller to service system needs during verification.

Returns

- BOOTLOADER_OK if the image parser was initialized, else error code.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader_continueVerifyImage

```
int32_t bootloader_continueVerifyImage (void * context, BootloaderParserCallback_t
metadataCallback)
```

Continue image verification.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	`context`	Pointer to a context structure that has been initialized by calling bootloader_initVerifyImage()
BootloaderParserCallback_t	[in]	`metadataCallback`	Function pointer, which is called when the binary metadata in the image is currently verified. Set to NULL if not required.

Continue verification of an upgrade image stored in a bootloader storage slot.

Note

- This function must be called in a loop until anything other than BOOTLOADER_ERROR_PARSE_CONTINUE is returned.
- bootloader_initVerifyImage must be called before calling this function to reset the parser.
- Instead of calling bootloader_initVerifyImage followed by bootloader_continueVerifyImage, call bootloader_verifyImage if no time-critical tasks are needed. The purpose of the init-and-continue functions is to allow the caller to service system needs during verification.

Returns

- BOOTLOADER_ERROR_PARSE_CONTINUE if parsing was successful, and the parser expects more data. BOOTLOADER_ERROR_PARSE_SUCCESS if the parser has successfully parsed the image and it passes verification. Else error code.

Continue verification of an upgrade image stored in a bootloader storage slot.

Note

- This function must be called in a loop until anything other than SL_STATUS_BL_PARSE_CONTINUE is returned.
- bootloader_initVerifyImage must be called before calling this function to reset the parser.
- Instead of calling bootloader_initVerifyImage followed by bootloader_continueVerifyImage, call bootloader_verifyImage if no time-critical tasks are needed. The purpose of the init-and-continue functions is to allow the caller to service system needs during verification.

Returns

- SL_STATUS_BL_PARSE_CONTINUE if parsing was successful, and the parser expects more data. SL_STATUS_BL_PARSE_SUCCESS if the parser has successfully parsed the image and it passes verification. Else error code.

bootloader_verifyImage

```
int32_t bootloader_verifyImage (uint32_t slotId, BootloaderParserCallback_t metadataCallback)
```

Verify that the image in the given storage slot is valid.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	ID of the slot to check
BootloaderParserCallback_t	[in]	`metadataCallback`	Function pointer, which is called when binary metadata is present in the storage slot. Set to NULL if not required.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Note

- This function allocates a context structure of the size given by BOOTLOADER_STORAGE_VERIFICATION_CONTEXT_SIZE on the caller's stack. To manage memory and allocate the context elsewhere (on the heap, as a global variable, and so on), use bootloader_initVerifyImage and bootloader_continueVerifyImage functions instead.

Returns

- BOOTLOADER_OK if the image is valid, else error code.

bootloader_getImageInfo

```
int32_t bootloader_getImageInfo (uint32_t slotId, ApplicationData_t * appInfo, uint32_t *
bootloaderVersion)
```

Get application and bootloader upgrade metadata from the storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	ID of the slot to check
ApplicationData_t *	[out]	`appInfo`	Pointer to ApplicationData_t struct
uint32_t *	[out]	`bootloaderVersion`	Pointer to an integer representing bootloader version

Returns

- BOOTLOADER_OK if metadata was filled successfully

bootloader_storageIsBusy

```
bool bootloader_storageIsBusy (void )
```

Check whether the bootloader storage is busy.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the storage is busy

bootloader_readRawStorage

```
int32_t bootloader_readRawStorage (uint32_t address, uint8_t * buffer, size_t length)
```

Read raw data from storage.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`address`	Address to start reading from
uint8_t *	[out]	`buffer`	Buffer to store the data

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
size_t	[in]	length	Amount of data to read

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_writeRawStorage

```
int32_t bootloader_writeRawStorage (uint32_t address, uint8_t * buffer, size_t length)
```

Write data to storage.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	address	Address to start writing to
uint8_t *	[in]	buffer	Buffer to read data to write from
size_t	[in]	length	Amount of data to write

Note

- If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

Note

- If DMA-based MSC write is enabled on the bootloader, writing data from flash to flash is not supported on Series-1 devices.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

bootloader_eraseRawStorage

```
int32_t bootloader_eraseRawStorage (uint32_t address, size_t length)
```

Erase data from storage.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	address	Address to start erasing from
size_t	[in]	length	Length of data to erase

Note

- Erasing storage must adhere to the limitations of the underlying storage medium, such as requiring full page erases. Use bootloader_getStorageInfo to learn about the limitations of the configured storage medium.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader_getAllocatedDMAChannel

```
int32_t bootloader_getAllocatedDMAChannel (void )
```

Get allocated DMA channel for MSC write.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A positive number channel. -1 if DMA-based MSC write is not enabled. Otherwise, the error code BOOTLOADER_ERROR_INIT_STORAGE.

bootloader_getAppMinVersion

```
void bootloader_getAppMinVersion (uint32_t * version)
```

Get application minimum version.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[in]	`version`	Application minimum version

Note

- Get the application minimum version to validate with application version when ROLLBACK PROTECTION is enabled in bootloader core configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageSlot_t

Information about a storage slot.

Public Attributes

uint32_t	address Address of the slot.
uint32_t	length Size of the slot.

Public Attribute Documentation

address

```
uint32_t BootloaderStorageSlot_t::address
```

Address of the slot.

length

```
uint32_t BootloaderStorageSlot_t::length
```

Size of the slot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageImplementationInformation_t

Information about the bootloader storage implementation Note: From Gecko Bootloader version >= 2.1, the pointer partType will only contain a zero value.

The partType variable can be used to find information about the attached storage.

Public Attributes

uint16_t	version The version of this data structure.
uint16_t	capabilitiesMask A bitmask describing the capabilities of this particular storage.
uint32_t	pageEraseMs Maximum time it takes to erase a page. (in milliseconds)
uint32_t	partEraseMs Maximum time it takes to erase the entire part. (in milliseconds)
uint32_t	pageSize The size of a single erasable page in bytes.
uint32_t	partSize The total size of the storage in bytes.
char *	partDescription Pointer to a string describing the attached storage.
uint8_t	wordSizeBytes The number of bytes in a word for the storage.
uint32_t	partType Value representing the attached storage.

Public Attribute Documentation

version

```
uint16_t BootloaderStorageImplementationInformation_t::version
```

The version of this data structure.

capabilitiesMask

```
uint16_t BootloaderStorageImplementationInformation_t::capabilitiesMask
```

A bitmask describing the capabilities of this particular storage.

pageEraseMs

```
uint32_t BootloaderStorageImplementationInformation_t::pageEraseMs
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Maximum time it takes to erase a page. (in milliseconds)

partEraseMs

```
uint32_t BootloaderStorageImplementationInformation_t::partEraseMs
```

Maximum time it takes to erase the entire part. (in milliseconds)

pageSize

```
uint32_t BootloaderStorageImplementationInformation_t::pageSize
```

The size of a single erasable page in bytes.

partSize

```
uint32_t BootloaderStorageImplementationInformation_t::partSize
```

The total size of the storage in bytes.

partDescription

```
char* BootloaderStorageImplementationInformation_t::partDescription
```

Pointer to a string describing the attached storage.

wordSizeBytes

```
uint8_t BootloaderStorageImplementationInformation_t::wordSizeBytes
```

The number of bytes in a word for the storage.

partType

```
uint32_t BootloaderStorageImplementationInformation_t::partType
```

Value representing the attached storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageInformation_t

Information about the bootloader storage
Note: The flashInfo variable is only usable with Gecko Bootloader version >= 2.0.

All previous versions of the Gecko Bootloader do not support the flashInfo data field.

Public Attributes

uint32_t	version The version of this data structure.
uint32_t	capabilities The capabilities of the storage component.
BootloaderStorageType_t	storageType Type of storage.
uint32_t	numStorageSlots Number of storage slots.
BootloaderStorageImplementationInformation_t *	info A pointer to detailed information about the attached storage.
BootloaderStorageImplementationInformation_t	flashInfo Detailed information about the attached storage(available for use only with Gecko Bootloader version >= 2.0)

Public Attribute Documentation

version

```
uint32_t BootloaderStorageInformation_t::version
```

The version of this data structure.

capabilities

```
uint32_t BootloaderStorageInformation_t::capabilities
```

The capabilities of the storage component.

storageType

```
BootloaderStorageType_t BootloaderStorageInformation_t::storageType
```

Type of storage.

numStorageSlots

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint32_t BootloaderStorageInformation_t::numStorageSlots
```

Number of storage slots.

info

```
BootloaderStorageImplementationInformation_t* BootloaderStorageInformation_t::info
```

A pointer to detailed information about the attached storage.

flashInfo

```
BootloaderStorageImplementationInformation_t BootloaderStorageInformation_t::flashInfo
```

Detailed information about the attached storage(available for use only with Gecko Bootloader version >= 2.0)

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderEraseStatus_t

Erase status struct.

Public Attributes

uint32_t	currentPageAddr Address of the current page to be erased.
uint32_t	pageSize The size of a single erasable page in bytes.
BootloaderStorageSlot_t	storageSlotInfo Information about a storage slot.

Public Attribute Documentation

currentPageAddr

```
uint32_t BootloaderEraseStatus_t::currentPageAddr
```

Address of the current page to be erased.

pageSize

```
uint32_t BootloaderEraseStatus_t::pageSize
```

The size of a single erasable page in bytes.

storageSlotInfo

```
BootloaderStorageSlot_t BootloaderEraseStatus_t::storageSlotInfo
```

Information about a storage slot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageFunctions_t

Storage API accessible from the application.

Public Attributes

uint32_t	version Version of this struct.
void(*	getInfo Get information about the storage – capabilities, layout, configuration.
int32_t(*	getSlotInfo Get information about storage slot – size, location.
int32_t(*	read Read bytes from slot into buffer.
int32_t(*	write Write bytes from buffer into slot.
int32_t(*	erase Erase an entire slot.
int32_t(*	setImagesToBootload Mark a list of slots for bootload.
int32_t(*	getImagesToBootload Mark a list of slots for bootload.
int32_t(*	appendImageToBootloadList Append a slot to bootload list.
int32_t(*	initParseImage Start image parsing.
int32_t(*	verifyImage Continue image verification.
int32_t(*	getImageInfo Get app and bootloader upgrade information from storage slot.
bool(*	isBusy Check whether the bootloader storage is busy.
int32_t(*	readRaw Read raw bytes from storage.
int32_t(*	writeRaw Write bytes to raw storage.
int32_t(*	eraseRaw Erase storage.
int32_t(*	getDMAchannel Get configured DMA channel.
void(*	getAppMinVersion Get application minimum version.

Public Attribute Documentation

Copyright © 2025 Silicon Laboratories. All rights reserved.

version

```
uint32_t BootloaderStorageFunctions_t::version
```

Version of this struct.

getInfo

```
void(* BootloaderStorageFunctions_t::getInfo)(BootloaderStorageInformation_t *info)
```

Get information about the storage – capabilities, layout, configuration.

getSlotInfo

```
int32_t(* BootloaderStorageFunctions_t::getSlotInfo)(uint32_t slotId, BootloaderStorageSlot_t
*slot)
```

Get information about storage slot – size, location.

read

```
int32_t(* BootloaderStorageFunctions_t::read)(uint32_t slotId, uint32_t offset, uint8_t
*buffer, size_t length)
```

Read bytes from slot into buffer.

write

```
int32_t(* BootloaderStorageFunctions_t::write)(uint32_t slotId, uint32_t offset, uint8_t
*buffer, size_t length)
```

Write bytes from buffer into slot.

erase

```
int32_t(* BootloaderStorageFunctions_t::erase) (uint32_t slotId)
```

Erase an entire slot.

setImagesToBootload

```
int32_t(* BootloaderStorageFunctions_t::setImagesToBootload)(int32_t *slotIds, size_t length)
```

Mark a list of slots for bootload.

Copyright © 2025 Silicon Laboratories. All rights reserved.

getImagesToBootload

```
int32_t(* BootloaderStorageFunctions_t::getImagesToBootload)(int32_t *slotIds, size_t length)
```

Mark a list of slots for bootload.

appendImageToBootloadList

```
int32_t(* BootloaderStorageFunctions_t::appendImageToBootloadList)(int32_t slotId)
```

Append a slot to bootload list.

initParseImage

```
int32_t(* BootloaderStorageFunctions_t::initParseImage)(uint32_t slotId,
BootloaderParserContext_t *context, size_t contextSize)
```

Start image parsing.

verifyImage

```
int32_t(* BootloaderStorageFunctions_t::verifyImage)(BootloaderParserContext_t *context,
BootloaderParserCallback_t metadataCallback)
```

Continue image verification.

getImageInfo

```
int32_t(* BootloaderStorageFunctions_t::getImageInfo)(BootloaderParserContext_t *context,
ApplicationData_t *appInfo, uint32_t *bootloaderVersion)
```

Get app and bootloader upgrade information from storage slot.

isBusy

```
bool(* BootloaderStorageFunctions_t::isBusy)(void)
```

Check whether the bootloader storage is busy.

readRaw

```
int32_t(* BootloaderStorageFunctions_t::readRaw)(uint32_t address, uint8_t *buffer, size_t
length)
```

Read raw bytes from storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

writeRaw

```
int32_t(* BootloaderStorageFunctions_t::writeRaw)(uint32_t address, uint8_t *buffer, size_t
length)
```

Write bytes to raw storage.

eraseRaw

```
int32_t(* BootloaderStorageFunctions_t::eraseRaw)(uint32_t address, size_t length)
```

Erase storage.

getDMAchannel

```
int32_t(* BootloaderStorageFunctions_t::getDMAchannel)(void)
```

Get configured DMA channel.

getAppMinVersion

```
void(* BootloaderStorageFunctions_t::getAppMinVersion) (uint32_t *version)
```

Get application minimum version.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Common Application Interface

Common Application Interface

Generic application interface available on all versions of the bootloader, regardless of the available components.

Note

- These Bootloader APIs are not reentrant and should be wrapped in critical section where needed.

Modules

BareBootTable_t

BootloaderInformation_t

BootloaderHeader_t

FirstBootloaderTable_t

MainBootloaderTable_t

Reset Information

Enumerations

```
enum    BootloaderType_t {
          NO_BOOTLOADER = 0
          SL_BOOTLOADER = 1
          NO_BOOTLOADER = 0
          SL_BOOTLOADER = 1
        }
```

Bootloader interface APIs are trust zone aware.

```
enum    BootloaderType_t {
          NO_BOOTLOADER = 0
          SL_BOOTLOADER = 1
          NO_BOOTLOADER = 0
          SL_BOOTLOADER = 1
        }
```

Type of bootloader.

Functions

```
   void    bootloader_getInfo(BootloaderInformation_t *info)
```

Get information about the bootloader on this device.

```
int32_t    bootloader_init(void)
```

Initialize components of the bootloader so the app can use the interface.

```
int32_t    bootloader_deinit(void)
```

De-initialize components of the bootloader that were previously initialized.

```
   void    bootloader_rebootAndInstall(void)
```

Reboot into the bootloader to perform an install.

```
   bool    bootloader_verifyApplication(uint32_t startAddress)
```

Verify the application image stored in the Flash memory starting at the address startAddress.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool	bootloader_secureBootEnforced(void) Check whether signature verification on the application image in internal flash is enforced before every boot.
bool	bootloader_getUpgradeLocation(uint32_t *location) Get base address of the bootloader upgrade image.
uint32_t	bootloader_remainingApplicationUpgrades(void) Count the total remaining number of application upgrades.
BootloaderResetCause_t	bootloader_getResetReason(void) Get reset cause of the bootloader.
bool	bootloader_pointerValid(const void *ptr) Check if a pointer is valid and if it points to the bootloader main stage.
bool	bootloader_getCertificateVersion(uint32_t *version) Get bootloader certificate version.
uint32_t	bootloader_GetAliasedAddr(uint32_t address) Get Aliased flash address.

Macros

#define	BOOTLOADER_VERSION_MAJOR_SHIFT (24U) Bootloader version major version shift value.
#define	BOOTLOADER_VERSION_MINOR_SHIFT (16U) Bootloader version minor version shift value.
#define	BOOTLOADER_VERSION_MAJOR_MASK (0xFF000000U) Bootloader version major version mask.
#define	BOOTLOADER_VERSION_MINOR_MASK (0x00FF0000U) Bootloader version minor version mask.
#define	BOOTLOADER_CAPABILITY_ENFORCE_UPGRADE_SIGNATURE (1 << 0) Bootloader enforces signed application upgrade images.
#define	BOOTLOADER_CAPABILITY_ENFORCE_UPGRADE_ENCRYPTION (1 << 1) Bootloader enforces encrypted application upgrade images.
#define	BOOTLOADER_CAPABILITY_ENFORCE_SECURE_BOOT (1 << 2) Bootloader enforces signature verification of the application image before every boot.
#define	BOOTLOADER_CAPABILITY_BOOTLOADER_UPGRADE (1 << 4) Bootloader has the capability of being upgraded.
#define	BOOTLOADER_CAPABILITY_GBL (1 << 5) Bootloader has the capability of parsing GBL files.
#define	BOOTLOADER_CAPABILITY_GBL_SIGNATURE (1 << 6) Bootloader has the capability of parsing signed GBL files.
#define	BOOTLOADER_CAPABILITY_GBL_ENCRYPTION (1 << 7) Bootloader has the capability of parsing encrypted GBL files.
#define	BOOTLOADER_CAPABILITY_ENFORCE_CERTIFICATE_SECURE_BOOT (1 << 8) Bootloader enforces signature verification of the application image before every boot using certificate.
#define	BOOTLOADER_CAPABILITY_ROLLBACK_PROTECTION (1 << 9) Bootloader has the capability of application rollback protection.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define BOOTLOADER_CAPABILITY_PERIPHERAL_LIST (1 << 10)
Bootloader has the capability to check the peripherals in use.

#define BOOTLOADER_CAPABILITY_STORAGE (1 << 16)
Bootloader has the capability of storing data in an internal or external storage medium.

#define BOOTLOADER_CAPABILITY_COMMUNICATION (1 << 20)
Bootloader has the capability of communicating with host processors using a communication interface.

#define BOOTLOADER_MAGIC_FIRST_STAGE (0xB00710ADUL)
Magic word indicating first stage bootloader table.

#define BOOTLOADER_MAGIC_MAIN (0x5ECDB007UL)
Magic word indicating main bootloader table.

#define mainBootloaderTable undefined
Pointer to main bootloader table.

#define BOOTLOADER_VERSION_MAJOR_SHIFT (24U)
Bootloader version major version shift value.

#define BOOTLOADER_VERSION_MINOR_SHIFT (16U)
Bootloader version minor version shift value.

#define BOOTLOADER_VERSION_MAJOR_MASK (0xFF000000U)
Bootloader version major version mask.

#define BOOTLOADER_VERSION_MINOR_MASK (0x00FF0000U)
Bootloader version minor version mask.

#define BOOTLOADER_CAPABILITY_ENFORCE_UPGRADE_SIGNATURE (1 << 0)
Bootloader enforces signed application upgrade images.

#define BOOTLOADER_CAPABILITY_ENFORCE_UPGRADE_ENCRYPTION (1 << 1)
Bootloader enforces encrypted application upgrade images.

#define BOOTLOADER_CAPABILITY_ENFORCE_SECURE_BOOT (1 << 2)
Bootloader enforces signature verification of the application image before every boot.

#define BOOTLOADER_CAPABILITY_BOOTLOADER_UPGRADE (1 << 4)
Bootloader has the capability of being upgraded.

#define BOOTLOADER_CAPABILITY_GBL (1 << 5)
Bootloader has the capability of parsing GBL files.

#define BOOTLOADER_CAPABILITY_GBL_SIGNATURE (1 << 6)
Bootloader has the capability of parsing signed GBL files.

#define BOOTLOADER_CAPABILITY_GBL_ENCRYPTION (1 << 7)
Bootloader has the capability of parsing encrypted GBL files.

#define BOOTLOADER_CAPABILITY_ENFORCE_CERTIFICATE_SECURE_BOOT (1 << 8)
Bootloader enforces signature verification of the application image before every boot using certificate.

#define BOOTLOADER_CAPABILITY_ROLLBACK_PROTECTION (1 << 9)
Bootloader has the capability of application rollback protection.

#define BOOTLOADER_CAPABILITY_PERIPHERAL_LIST (1 << 10)
Bootloader has the capability to check the peripherals in use.

#define BOOTLOADER_CAPABILITY_REJECT_DIRECT_SIGNED_IMG (1 << 11)
Bootloader rejects directly signed images without certificates.

#define BOOTLOADER_CAPABILITY_STORAGE (1 << 16)
Bootloader has the capability of storing data in an internal or external storage medium.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define BOOTLOADER_CAPABILITY_COMMUNICATION (1 << 20)
Bootloader has the capability of communicating with host processors using a communication interface.

#define BOOTLOADER_MAGIC_FIRST_STAGE (0xB00710ADUL)
Magic word indicating first stage bootloader table.

#define BOOTLOADER_MAGIC_MAIN (0x5ECDB007UL)
Magic word indicating main bootloader table.

#define mainBootloaderTable undefined
Pointer to main bootloader table.

Enumeration Documentation

BootloaderType_t

```
BootloaderType_t
```

Bootloader interface APIs are trust zone aware.

Type of bootloader

Enumerator	
NO_BOOTLOADER	No bootloader present.
SL_BOOTLOADER	Bootloader is a Silicon Labs bootloader.
NO_BOOTLOADER	No bootloader present.
SL_BOOTLOADER	Bootloader is a Silicon Labs bootloader.

BootloaderType_t

```
BootloaderType_t
```

Type of bootloader.

Enumerator	
NO_BOOTLOADER	No bootloader present.
SL_BOOTLOADER	Bootloader is a Silicon Labs bootloader.
NO_BOOTLOADER	No bootloader present.
SL_BOOTLOADER	Bootloader is a Silicon Labs bootloader.

Function Documentation

bootloader_getInfo

```
void bootloader_getInfo (BootloaderInformation_t * info)
```

Get information about the bootloader on this device.

Parameters

Type	Direction	Argument Name	Description
BootloaderInformation_t *	[out]	info	Pointer to the bootloader information struct.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The returned information is fetched from the main bootloader information table.

bootloader_init

```
int32_t bootloader_init (void )
```

Initialize components of the bootloader so the app can use the interface.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This typically includes initializing serial peripherals for communication with external SPI flashes, and so on.

Returns

- Error code. BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_INIT_BASE range.

bootloader_deinit

```
int32_t bootloader_deinit (void )
```

De-initialize components of the bootloader that were previously initialized.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This typically includes powering down external SPI flashes and de-initializing the serial peripheral used for communication with the external flash.

Returns

- Error code. BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_INIT_BASE range.

bootloader_rebootAndInstall

```
void bootloader_rebootAndInstall (void )
```

Reboot into the bootloader to perform an install.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

If there is a storage component and a slot is marked for bootload, install the image in that slot after verifying it.

If a communication component is present, open the communication channel and receive an image to be installed.

bootloader_verifyApplication

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
bool bootloader_verifyApplication (uint32_t startAddress)
```

Verify the application image stored in the Flash memory starting at the address startAddress.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`startAddress`	Starting address of the application.

If the secure boot is enforced, the function will only return true if the cryptographic signature of the application is valid. Otherwise, the application is verified according to the signature type defined in the ApplicationProperties_t structure embedded in the application. Silicon Labs wireless stacks include a declaration of this structure. However, applications not using a full wireless stack may need to instantiate the structure.

Examples of results when the secure boot is not enforced:

- App has no signature: Valid if initial stack pointer and program counter have reasonable values.
- App has CRC checksum: Valid if checksum is valid.
- App has ECDSA signature: Valid if ECDSA signature is valid.

When secure boot is enforced, only ECDSA signed applications with a valid signature are considered valid.

Returns

- True if the application is valid, else false.

bootloader_secureBootEnforced

```
bool bootloader_secureBootEnforced (void )
```

Check whether signature verification on the application image in internal flash is enforced before every boot.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if signature verification is enforced, else false.

bootloader_getUpgradeLocation

```
bool bootloader_getUpgradeLocation (uint32_t * location)
```

Get base address of the bootloader upgrade image.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[out]	`location`	the base address of bootloader upgrade image.

Returns

- Returns true if the location was found.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader_remainingApplicationUpgrades

```
uint32_t bootloader_remainingApplicationUpgrades (void )
```

Count the total remaining number of application upgrades.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- remaining number of application upgrades.

bootloader_getResetReason

```
BootloaderResetCause_t bootloader_getResetReason (void )
```

Get reset cause of the bootloader.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Reset cause of the bootloader.

bootloader_pointerValid

```
bool bootloader_pointerValid (const void * ptr)
```

Check if a pointer is valid and if it points to the bootloader main stage.

Parameters

Type	Direction	Argument Name	Description
const void *	[in]	`ptr`	The pointer to check

This function checks pointers to bootloader jump tables.

Returns

- True if the pointer points into the bootloader main stage, false if not.

bootloader_getCertificateVersion

```
bool bootloader_getCertificateVersion (uint32_t * version)
```

Get bootloader certificate version.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint32_t *	[out]	`version`	Bootloader certificate version

Returns

- True if a valid certificate version is found.

bootloader_GetAliasedAddr

```
uint32_t bootloader_GetAliasedAddr (uint32_t address)
```

Get Aliased flash address.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`address`	to be converted based on secure/nonsecure

This function gets aliased flash address

Returns

- Aliased address

Copyright © 2025 Silicon Laboratories. All rights reserved.

BareBootTable_t

Bare boot table. Can be mapped on top of vector table to access contents.

Public Attributes

uint32_t *	stackTop Pointer to top of stack.
void(*	resetVector Pointer to reset vector.
uint32_t	reserved0 Reserved pointers to fault handlers.
uint32_t	reserved1 Reserved pointers to RESERVED fields.
void *	table Pointer to bootloader table.
uint32_t	reserved2 Reserved pointers to SVC and DebugMon interrupts.
void *	signature Pointer to application signature.

Public Attribute Documentation

stackTop

```
uint32_t * BareBootTable_t::stackTop
```

Pointer to top of stack.

resetVector

```
void(* BareBootTable_t::resetVector)(void)
```

Pointer to reset vector.

reserved0

```
uint32_t BareBootTable_t::reserved0
```

Reserved pointers to fault handlers.

reserved1

```
uint32_t BareBootTable_t::reserved1
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Reserved pointers to RESERVED fields.

table

```
void * BareBootTable_t::table
```

Pointer to bootloader table.

reserved2

```
uint32_t BareBootTable_t::reserved2
```

Reserved pointers to SVC and DebugMon interrupts.

signature

```
void * BareBootTable_t::signature
```

Pointer to application signature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderInformation_t

Information about the current bootloader.

Public Attributes

BootloaderType_t	type The type of bootloader.
uint32_t	version Version number of the bootloader.
uint32_t	capabilities Capability mask for the bootloader.

Public Attribute Documentation

type

```
BootloaderType_t BootloaderInformation_t::type
```

The type of bootloader.

version

```
uint32_t BootloaderInformation_t::version
```

Version number of the bootloader.

capabilities

```
uint32_t BootloaderInformation_t::capabilities
```

Capability mask for the bootloader.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderHeader_t

Common header for bootloader tables.

Public Attributes

uint32_t	type Type of image.
uint32_t	layout Version number of the bootloader/application table.
uint32_t	version Version number of the image.

Public Attribute Documentation

type

```
uint32_t BootloaderHeader_t::type
```

Type of image.

layout

```
uint32_t BootloaderHeader_t::layout
```

Version number of the bootloader/application table.

version

```
uint32_t BootloaderHeader_t::version
```

Version number of the image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

FirstBootloaderTable_t

Address table for the First Stage Bootloader.

Public Attributes

BootloaderHeader_t	header Header of the First Stage Bootloader table.
BareBootTable_t *	mainBootloader Start address of the Main Bootloader.
BareBootTable_t *	upgradeLocation Location of the Main Bootloader upgrade image.

Public Attribute Documentation

header

```
BootloaderHeader_t FirstBootloaderTable_t::header
```

Header of the First Stage Bootloader table.

mainBootloader

```
BareBootTable_t * FirstBootloaderTable_t::mainBootloader
```

Start address of the Main Bootloader.

upgradeLocation

```
BareBootTable_t * FirstBootloaderTable_t::upgradeLocation
```

Location of the Main Bootloader upgrade image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

MainBootloaderTable_t

Address table for the Main Bootloader.

Public Attributes

BootloaderHeader_t	**header** Header of the Main Bootloader table.
uint32_t	**size** Size of the Main Bootloader.
BareBootTable_t *	**startOfAppSpace** Start address of the application.
uint32_t *	**endOfAppSpace** End address of the allocated application space.
uint32_t	**capabilities** Capabilities of the bootloader.
int32_t(*	**init** Initialize bootloader for use from application.
int32_t(*	**deinit** Deinitialize bootloader after use from application.
bool(*	**verifyApplication** Verify application.
int32_t(*	**initParser** Initialize parser.
int32_t(*	**parseBuffer** Parse a buffer.
const BootloaderStorageFunctions_t *	**storage** Function table for storage component.
int32_t(*	**parseImageInfo** Parse a buffer and get application and bootloader upgrade metadata from the buffer.
uint32_t(*	**parserContextSize** Size of context buffer used by bootloader image parser to store parser state.
uint32_t(*	**remainingApplicationUpgrades** Remaining number of application upgrades.
void(*	**getPeripheralList** Get the list of the peripheral that is used by the bootloader.
uint32_t(*	**getUpgradeLocation** Get base address of bootloader upgrade image.

Public Attribute Documentation

header

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
BootloaderHeader_t MainBootloaderTable_t::header
```

Header of the Main Bootloader table.

size

```
uint32_t MainBootloaderTable_t::size
```

Size of the Main Bootloader.

startOfAppSpace

```
BareBootTable_t * MainBootloaderTable_t::startOfAppSpace
```

Start address of the application.

endOfAppSpace

```
uint32_t * MainBootloaderTable_t::endOfAppSpace
```

End address of the allocated application space.

capabilities

```
uint32_t MainBootloaderTable_t::capabilities
```

Capabilities of the bootloader.

init

```
int32_t(* MainBootloaderTable_t::init)(void)
```

Initialize bootloader for use from application.

deinit

```
int32_t(* MainBootloaderTable_t::deinit)(void)
```

Deinitialize bootloader after use from application.

verifyApplication

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
bool(* MainBootloaderTable_t::verifyApplication)(uint32_t startAddress)
```

Verify application.

initParser

```
int32_t(* MainBootloaderTable_t::initParser)(BootloaderParserContext_t *context, size_t
contextSize)
```

Initialize parser.

parseBuffer

```
int32_t(* MainBootloaderTable_t::parseBuffer)(BootloaderParserContext_t *context, const
BootloaderParserCallbacks_t *callbacks, uint8_t data[], size_t numBytes)
```

Parse a buffer.

storage

```
const BootloaderStorageFunctions_t * MainBootloaderTable_t::storage
```

Function table for storage component.

parseImageInfo

```
int32_t(* MainBootloaderTable_t::parseImageInfo)(BootloaderParserContext_t *context, uint8_t
data[], size_t numBytes, ApplicationData_t *appInfo, uint32_t *bootloaderVersion)
```

Parse a buffer and get application and bootloader upgrade metadata from the buffer.

parserContextSize

```
uint32_t(* MainBootloaderTable_t::parserContextSize)(void)
```

Size of context buffer used by bootloader image parser to store parser state.

remainingApplicationUpgrades

```
uint32_t(* MainBootloaderTable_t::remainingApplicationUpgrades)(void)
```

Remaining number of application upgrades.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void(* MainBootloaderTable_t::getPeripheralList)(uint32_t *ppusatd0, uint32_t *ppusatd1)
```

Get the list of the peripheral that is used by the bootloader.

getUpgradeLocation

```
uint32_t(* MainBootloaderTable_t::getUpgradeLocation)(void)
```

Get base address of bootloader upgrade image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Reset Information

Reset Information

Passing information when resetting into and out of the bootloader.

To signal the bootloader to run, the application needs to write the BootloaderResetCause_t structure to the first address of RAM, setting BootloaderResetCause_t::reason to BOOTLOADER_RESET_REASON_BOOTLOAD.

The reset cause is only valid if BootloaderResetCause_t::signature is set to BOOTLOADER_RESET_SIGNATURE_VALID.

```
BootloaderResetCause_t resetCause = {
  .reason = BOOTLOADER_RESET_REASON_BOOTLOAD,
  .signature = BOOTLOADER_RESET_SIGNATURE_VALID
}
```

When the bootloader reboots back into the app, it sets the reset reason to BOOTLOADER_RESET_REASON_GO if the bootload succeeded, or BOOTLOADER_RESET_REASON_BADIMAGE if the bootload failed due to errors when parsing the upgrade image.

Note

- The reset information is automatically filled out before reset if the bootloader_rebootAndInstall function is called.

Modules

BootloaderResetCause_t

Macros

#define BOOTLOADER_RESET_REASON_UNKNOWN 0x0200u
Unknown bootloader cause (should never occur)

#define BOOTLOADER_RESET_REASON_GO 0x0201u
Bootloader caused reset telling app to run.

#define BOOTLOADER_RESET_REASON_BOOTLOAD 0x0202u
Application requested that bootloader runs.

#define BOOTLOADER_RESET_REASON_BADIMAGE 0x0203u
Bootloader detected bad external upgrade image.

#define BOOTLOADER_RESET_REASON_FATAL 0x0204u
Fatal Error or assert in bootloader.

#define BOOTLOADER_RESET_REASON_FORCE 0x0205u
Forced bootloader activation.

#define BOOTLOADER_RESET_REASON_OTAVALID 0x0206u
OTA Bootloader mode activation.

#define BOOTLOADER_RESET_REASON_DEEPSLEEP 0x0207u
Bootloader initiated deep sleep.

#define BOOTLOADER_RESET_REASON_BADAPP 0x0208u
Application verification failed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define BOOTLOADER_RESET_REASON_UPGRADE 0x0209u
Bootloader requested that first stage upgrades main bootloader.

#define BOOTLOADER_RESET_REASON_TIMEOUT 0x020Au
Bootloader timed out waiting for upgrade image.

#define BOOTLOADER_RESET_REASON_FAULT 0x020Bu
Soft-reset was forced to handle a fault.

#define BOOTLOADER_RESET_REASON_TZ_FAULT 0x020Cu
Soft-reset was forced to handle a security fault.

#define BOOTLOADER_RESET_REASON_NO_SLOT_SPACE 0x020Du
Insufficient slot space to re-create a new firmware.

#define BOOTLOADER_RESET_REASON_BADCRC 0x020Eu
CRC mismatch of the newly re-constructed firmware.

#define BOOTLOADER_RESET_REASON_DDFU_FAIL 0x020Fu
Re-creation of the new application using the DDFU library failed.

#define BOOTLOADER_RESET_SIGNATURE_VALID 0xF00Fu
Reset signature is valid.

#define BOOTLOADER_RESET_SIGNATURE_INVALID 0xC33Cu
Reset signature is invalid.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderResetCause_t

Reset cause of the bootloader.

Public Attributes

uint16_t	reason Reset reason as defined in the reset information.
uint16_t	signature Signature indicating whether the reset reason is valid.

Public Attribute Documentation

reason

```
uint16_t BootloaderResetCause_t::reason
```

Reset reason as defined in the reset information.

signature

```
uint16_t BootloaderResetCause_t::signature
```

Signature indicating whether the reset reason is valid.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader Core

Bootloader Core

Core bootloader functionality.

Core functionality for the Silicon Labs Bootloader.

Modules

Bootload

Flash

Reset

TrustZone

Upgrade

Functions

`int32_t` `btl_init(void)`
Initialize bootloader.

`int32_t` `btl_deinit(void)`
Deinitialize bootloader.

Function Documentation

btl_init

```
int32_t btl_init (void )
```

Initialize bootloader.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code. BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_INIT_BASE range.

btl_deinit

```
int32_t btl_deinit (void )
```

Deinitialize bootloader.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code. BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_INIT_BASE range.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootload

Bootload

Methods to verify and bootload application images.

Functions

uint32_t **bootload_getUpgradeLocation**(void)
Get base address of the bootloader upgrade image.

uint32_t **bootload_getBootloaderVersion**(void)
Get the version of the bootloader.

bool **bootload_getApplicationVersion**(uint32_t *version)
Get the version of the application.

bool **bootload_checkApplicationPropertiesMagic**(void *appProperties)
Check application properties magic.

bool **bootload_checkApplicationPropertiesVersion**(void *appProperties)
Check application properties struct version.

bool **bootload_verifyApplication**(uint32_t startAddress)
Verify the application image stored in the Flash memory starting at the address startAddress.

void **bootload_bootloaderCallback**(uint32_t offset, uint8_t data[], size_t length, void *context)
Bootloader upgrade callback implementation.

void **bootload_applicationCallback**(uint32_t address, uint8_t data[], size_t length, void *context)
Image data callback implementation.

bool **bootload_commitBootloaderUpgrade**(uint32_t upgradeAddress, uint32_t size)
Perform a bootloader upgrade using the upgrade image present at upgradeAddress with length size.

bool **bootload_verifyApplicationVersion**(uint32_t appVersion, bool checkRemainingAppUpgrades)
Verify the application version for rollback protection.

bool **bootload_storeApplicationVersion**(uint32_t startAddress)
Store the application version.

uint32_t **bootload_remainingApplicationUpgrades**(void)
Count the total remaining number of application upgrades.

void **bootload_storeApplicationVersionResetMagic**(void)
Store application version reset magic.

void **bootload_removeStoredApplicationVersions**(void)
Clean the application versions seen.

uint32_t **bootload_getApplicationVersionStorageCapacity**(void)
Get the application version storage capacity.

uint32_t * **bootload_getApplicationVersionStoragePtr**(uint32_t index)
Get the address of the application version storage buffer.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool bootload_gotCertificate(const void *appProp)
Check whether the application contains a certificate.

bool bootload_verifyCertificate(const void *cert)
Verify a certificate with a bootloader certificate.

bool bootload_verifyApplicationCertificate(const void *appProp, const void *gotCert)
Verify the application certificate.

Function Documentation

bootload_getUpgradeLocation

```
uint32_t bootload_getUpgradeLocation (void )
```

Get base address of the bootloader upgrade image.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Returns the base address of bootloader upgrade image.

bootload_getBootloaderVersion

```
uint32_t bootload_getBootloaderVersion (void )
```

Get the version of the bootloader.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Returns the version of the bootloader.

bootload_getApplicationVersion

```
bool bootload_getApplicationVersion (uint32_t * version)
```

Get the version of the application.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[out]	version	The retrieved application version

Returns

- Returns true if the version was retrieved successfully

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootload_checkApplicationPropertiesMagic

```
bool bootload_checkApplicationPropertiesMagic (void * appProperties)
```

Check application properties magic.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	appProperties	Pointer to ApplicationProperties_t

Returns

- True if the application properties magic is valid.

bootload_checkApplicationPropertiesVersion

```
bool bootload_checkApplicationPropertiesVersion (void * appProperties)
```

Check application properties struct version.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	appProperties	Pointer to ApplicationProperties_t

Returns

- True if the application properties struct version is compatible with the bootloader.

bootload_verifyApplication

```
bool bootload_verifyApplication (uint32_t startAddress)
```

Verify the application image stored in the Flash memory starting at the address startAddress.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	startAddress	Starting address of the application

If secure boot is enforced, the function will only return true if the cryptographic signature of the application is valid. Else, the application is verified according to the signature type defined in the ApplicationProperties_t structure embedded in the application. Silicon Labs wireless stacks declare this structure. Applications, which are not using a full wireless stack may need to instantiate the structure.

Examples of results when secure boot is not enforced:

- App has no signature: Valid if initial stack pointer and program counter have reasonable values
- App has CRC checksum: Valid if checksum is valid
- App has ECDSA signature: Valid if ECDSA signature is valid.

When secure boot is enforced, only ECDSA-signed applications with a valid signature are considered valid.

Returns

-

Copyright © 2025 Silicon Laboratories. All rights reserved.

True if the image is deemed valid

bootload_bootloaderCallback

```
void bootload_bootloaderCallback (uint32_t offset, uint8_t data, size_t length, void * context)
```

Bootloader upgrade callback implementation.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`offset`	Offset of bootloader data (byte counter incrementing from 0)
uint8_t	N/A	`data`	Raw bootloader data
size_t	N/A	`length`	Size in bytes of raw bootloader data.
void *	N/A	`context`	A context variable defined by the implementation that is implementing this callback.

bootload_applicationCallback

```
void bootload_applicationCallback (uint32_t address, uint8_t data, size_t length, void *
context)
```

Image data callback implementation.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`address`	Address (inside the raw image) the data starts at
uint8_t	N/A	`data`	Raw image data
size_t	N/A	`length`	Size in bytes of raw image data. Always constrained to a multiple of four.
void *	N/A	`context`	A context variable defined by the implementation that is implementing this callback.

bootload_commitBootloaderUpgrade

```
bool bootload_commitBootloaderUpgrade (uint32_t upgradeAddress, uint32_t size)
```

Perform a bootloader upgrade using the upgrade image present at upgradeAddress with length size.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`upgradeAddress`	The starting address of the upgrade image
uint32_t	[in]	`size`	The length of the upgrade image in bytes

If the bootloader upgrade process starts successfully, this function does not return and execution will resume from the reset handler of the upgraded bootloader.

Returns

- False if the bootloader upgrade process didn't start

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootload_verifyApplicationVersion

```
bool bootload_verifyApplicationVersion (uint32_t appVersion, bool checkRemainingAppUpgrades)
```

Verify the application version for rollback protection.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`appVersion`	Application version to be checked.
bool	[in]	`checkRemainingAppUpgrades`	Check remaining application upgrades.

Returns

- True if the application version is higher or equal than the application versions seen. False if the application version is lower than the application versions seen. False if no remaining application upgrades are left when `checkRemainingAppUpgrades` is true.

bootload_storeApplicationVersion

```
bool bootload_storeApplicationVersion (uint32_t startAddress)
```

Store the application version.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`startAddress`	Start address of application.

Note

- Only the version of the verified application should be stored.

Returns

- True if application version is successfully stored.

bootload_remainingApplicationUpgrades

```
uint32_t bootload_remainingApplicationUpgrades (void )
```

Count the total remaining number of application upgrades.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- remaining number of application upgrades.

bootload_storeApplicationVersionResetMagic

```
void bootload_storeApplicationVersionResetMagic (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Store application version reset magic.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Store application version reset magic to ensure that application versions are cleaned after a bootloader upgrade.

bootload_removeStoredApplicationVersions

```
void bootload_removeStoredApplicationVersions (void )
```

Clean the application versions seen.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- The application versions are cleaned only if this is requested with a magic and the application version storage is not already empty.

bootload_getApplicationVersionStorageCapacity

```
uint32_t bootload_getApplicationVersionStorageCapacity (void )
```

Get the application version storage capacity.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Application version storage capacity.

bootload_getApplicationVersionStoragePtr

```
uint32_t * bootload_getApplicationVersionStoragePtr (uint32_t index)
```

Get the address of the application version storage buffer.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`index`	Index of the application version storage buffer.

Returns

- Address of the application version storage buffer with the given index.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootload_gotCertificate

```
bool bootload_gotCertificate (const void * appProp)
```

Check whether the application contains a certificate.

Parameters

Type	Direction	Argument Name	Description
const void *	N/A	`appProp`	Pointer to ApplicationProperties_t of application.

Returns

- True if application contains a certificate.

bootload_verifyCertificate

```
bool bootload_verifyCertificate (const void * cert)
```

Verify a certificate with a bootloader certificate.

Parameters

Type	Direction	Argument Name	Description
const void *	N/A	`cert`	Pointer to ApplicationCertificate_t.

Returns

- True if certificate is verified.

bootload_verifyApplicationCertificate

```
bool bootload_verifyApplicationCertificate (const void * appProp, const void * gotCert)
```

Verify the application certificate.

Parameters

Type	Direction	Argument Name	Description
const void *	N/A	`appProp`	Pointer to ApplicationProperties_t of application.
const void *	N/A	`gotCert`	Boolean to store application certificate presence.

Note

- This function will always return true if certificate support is not enabled. Also true if `appProp` does not contain any certificate and direct signed applications can be accepted.

Returns

- True if application certificate is verified.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Flash

Flash

Interface to internal flash.

Used for writing application images to the main flash.

Functions

bool `flash_erasePage`(uint32_t address)
Erase a flash page.

bool `flash_writeBuffer_dma`(uint32_t address, const void *data, size_t length, int ch)
Write buffer to internal flash.

bool `flash_writeBuffer`(uint32_t address, const void *data, size_t length)
Write buffer to internal flash.

Macros

#define `SL_GBL_MSC_LDMA_CHANNEL 2`
DMA Channel for MSC write.

Function Documentation

flash_erasePage

```
bool flash_erasePage (uint32_t address)
```

Erase a flash page.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`address`	Start address of the flash page to erase.

Returns

- True if operation was successful

flash_writeBuffer_dma

```
bool flash_writeBuffer_dma (uint32_t address, const void * data, size_t length, int ch)
```

Write buffer to internal flash.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`address`	Starting address to write data to. Must be half-word aligned.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
const void *	N/A	`data`	Data buffer to write to internal flash
size_t	N/A	`length`	Amount of bytes in the data buffer to write
int	N/A	`ch`	DMA channel to use

Returns

- True if operation was successful

flash_writeBuffer

```
bool flash_writeBuffer (uint32_t address, const void * data, size_t length)
```

Write buffer to internal flash.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`address`	Starting address to write data to. Must be half-word aligned.
const void *	N/A	`data`	Data buffer to write to internal flash
size_t	N/A	`length`	Amount of bytes in the data buffer to write

Returns

- True if operation was successful

Copyright © 2025 Silicon Laboratories. All rights reserved.

Reset

Reset

Methods to reset from the bootloader to the app.

Functions

void	reset_resetWithReason(uint16_t resetReason) Reset from the bootloader with a reset cause.
void	reset_setResetReason(uint16_t resetReason) Set a reset reason.
void	reset_enableResetCounter(void) Use the lower 4 bits of the reset cause signature to store a reset counter able to count up to 15 before wrapping around.
void	reset_disableResetCounter(void) Clear and disable the reset counter.
bool	reset_resetCounterEnabled(void) Check whether the reset counter is enabled.
void	reset_incrementResetCounter(void) Increment the reset counter by one.
uint8_t	reset_getResetCounter(void) Get the reset counter value.
uint16_t	reset_getResetReason(void) Get the reset reason without verifying it.
void	reset_invalidateResetReason(void) Invalidate the reset reason.
uint16_t	reset_classifyReset(void) Classify reset and get the reset reason.

Function Documentation

reset_resetWithReason

```
void reset_resetWithReason (uint16_t resetReason)
```

Reset from the bootloader with a reset cause.

Parameters

Type	Direction	Argument Name	Description
uint16_t	N/A	resetReason	A reset reason as defined in the bootloader interface

Note

- This function does not return.

reset_setResetReason

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void reset_setResetReason (uint16_t resetReason)
```

Set a reset reason.

Parameters

Type	Direction	Argument Name	Description
uint16_t	N/A	`resetReason`	A reset reason as defined in the bootloader interface

reset_enableResetCounter

```
void reset_enableResetCounter (void )
```

Use the lower 4 bits of the reset cause signature to store a reset counter able to count up to 15 before wrapping around.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

The value of the counter will be preserved across resets.

reset_disableResetCounter

```
void reset_disableResetCounter (void )
```

Clear and disable the reset counter.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

reset_resetCounterEnabled

```
bool reset_resetCounterEnabled (void )
```

Check whether the reset counter is enabled.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if reset counter is enabled, else false.

reset_incrementResetCounter

```
void reset_incrementResetCounter (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Increment the reset counter by one.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Wraps around to 0 if the current counter value is equal to 15.

reset_getResetCounter

```
uint8_t reset_getResetCounter (void )
```

Get the reset counter value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- The reset counter has to be enabled for this value to be valid.

Returns

- The reset counter value

reset_getResetReason

```
uint16_t reset_getResetReason (void )
```

Get the reset reason without verifying it.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The reset reason

reset_invalidateResetReason

```
void reset_invalidateResetReason (void )
```

Invalidate the reset reason.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

Copyright © 2025 Silicon Laboratories. All rights reserved.

This will also disable the reset counter if it is in use.

reset_classifyReset

```
uint16_t reset_classifyReset (void )
```

Classify reset and get the reset reason.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Reset cause or BOOTLOADER_RESET_REASON_UNKNOWN

Copyright © 2025 Silicon Laboratories. All rights reserved.

TrustZone

TrustZone

TrustZone utilities.

Functions

`void` [bl_fatal_assert_action](#)`(void)`
TrustZone fatal error handler.

`bool` [bl_verify_ns_memory_access](#)`(const void *p, size_t s)`
Validate the non-secure pointers.

Function Documentation

bl_fatal_assert_action

```
void bl_fatal_assert_action (void )
```

TrustZone fatal error handler.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function triggers a soft-reset with the reset reason set to BOOTLOADER_RESET_REASON_TZ_FAULT.

bl_verify_ns_memory_access

```
bool bl_verify_ns_memory_access (const void * p, size_t s)
```

Validate the non-secure pointers.

Parameters

Type	Direction	Argument Name	Description
const void *	N/A	p	Pointer to the memory.
size_t	N/A	s	Size of the memory to be validated.

Validate if the address of the memory is actually non-secure as expected.

Returns

- True if the memory being accessed is from NS

Copyright © 2025 Silicon Laboratories. All rights reserved.

Upgrade

Upgrade

Methods to verify and upgrade the main bootloader.

Functions

`bool` `btl_checkForUpgrade(void)`
Check whether a bootloader upgrade is available.

`bool` `btl_applyUpgrade(void)`
Apply a bootloader upgrade.

Function Documentation

btl_checkForUpgrade

```
bool btl_checkForUpgrade (void )
```

Check whether a bootloader upgrade is available.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if an upgrade image is in the upgrade location

btl_applyUpgrade

```
bool btl_applyUpgrade (void )
```

Apply a bootloader upgrade.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the bootloader upgrade was applied successfully

Copyright © 2025 Silicon Laboratories. All rights reserved.

Components

Components

Modules

- Communication
- Debug
- Decompressor
- GPIO Activation
- Image Parser
- Security
- Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

Communication

Communication

Host communication interface.

The Communication component provides an interface for implementing communication with a host device, such as a computer or a microcontroller.

Communication Protocol Implementations

Several components implement the communication interface, using different transports and protocols.

Modules

EZSP-SPI

Bluetooth Apploader OTA DFU

UART XMODEM

BGAPI UART DFU

Utils

Functions

`void` `bootloader_ezsp_communication_init(void)`
Initialize hardware for the EZSP SPI Bootloader communication.

`int32_t` `bootloader_ezsp_communication_start(void)`
Initialize communication between the EZSP SPI bootloader and external host.

`int32_t` `bootloader_ezsp_communication_main(ImageProperties_t *imageProps, ParserContext_t *parserContext, const BootloaderParserCallbacks_t *parseCb)`
Communication main for the EZSP SPI bootloader.

`void` `bootloader_ezsp_communication_shutdown(void)`
Stop communication between the bootloader and external host.

`void` `bootloader_apploader_communication_init(void)`
Initialize hardware for the BLE Apploader OTA DFU Bootloader communication.

`int32_t` `bootloader_apploader_communication_start(void)`
Initialize communication between the BLE Apploader OTA DFU bootloader and external host.

`int32_t` `bootloader_apploader_communication_main(ImageProperties_t *imageProps, void *parserContext, void *decryptContext, void *authContext, const BootloaderParserCallbacks_t *parseCb)`
Communication main for the BLE Apploader OTA DFU bootloader.

`void` `bootloader_apploader_get_custom_device_address(sl_apploader_address_t *btAddress)`
Get custom device address.

`void` `bootloader_xmodem_communication_init(void)`
Initialize hardware for the UART XMODEM Bootloader communication.

Copyright © 2025 Silicon Laboratories. All rights reserved.

int32_t bootloader_xmodem_communication_start(void)
Initialize communication between the UART XMODEM bootloader and external host.

int32_t bootloader_xmodem_communication_main(ImageProperties_t *imageProps, const BootloaderParserCallbacks_t *parseCb)
Communication main for the UART XMODEM bootloader.

void bootloader_bgapi_communication_init(void)
Initialize hardware for the BGAPI UART DFU Bootloader communication.

int32_t bootloader_bgapi_communication_start(void)
Initialize communication between the BGAPI UART DFU bootloader and external host.

int32_t bootloader_bgapi_communication_main(ImageProperties_t *imageProps, ParserContext_t *parserContext, const BootloaderParserCallbacks_t *parseCb)
Communication main for the BGAPI UART DFU bootloader.

void communication_init(void)
Initialize hardware for the communication component.

int32_t communication_start(void)
Initialize communication between the bootloader and external host.

int32_t communication_main(void)
Not supposed to return until either the host signals the end of the current session or a new image has been flashed and verified.

void communication_shutdown(void)
Stop communication between the bootloader and external host.

Function Documentation

bootloader_ezsp_communication_init

```
void bootloader_ezsp_communication_init (void )
```

Initialize hardware for the EZSP SPI Bootloader communication.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

bootloader_ezsp_communication_start

```
int32_t bootloader_ezsp_communication_start (void )
```

Initialize communication between the EZSP SPI bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

bootloader_ezsp_communication_main

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
int32_t bootloader_ezsp_communication_main (ImageProperties_t * imageProps, ParserContext_t *
parserContext, const BootloaderParserCallbacks_t * parseCb)
```

Communication main for the EZSP SPI bootloader.

Parameters

Type	Direction	Argument Name	Description
ImageProperties_t *	N/A	imageProps	The image file processed
ParserContext_t *	N/A	parserContext	Image parser context
const BootloaderParserCallbacks_t *	N/A	parseCb	Bootloader parser callbacks

Returns

- Error code indicating success or failure.

bootloader_ezsp_communication_shutdown

```
void bootloader_ezsp_communication_shutdown (void )
```

Stop communication between the bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

bootloader_apploader_communication_init

```
void bootloader_apploader_communication_init (void )
```

Initialize hardware for the BLE Apploader OTA DFU Bootloader communication.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

bootloader_apploader_communication_start

```
int32_t bootloader_apploader_communication_start (void )
```

Initialize communication between the BLE Apploader OTA DFU bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader_apploader_communication_main

```
int32_t bootloader_apploader_communication_main (ImageProperties_t * imageProps, void *
parserContext, void * decryptContext, void * authContext, const BootloaderParserCallbacks_t *
parseCb)
```

Communication main for the BLE Apploader OTA DFU bootloader.

Parameters

Type	Direction	Argument Name	Description
ImageProperties_t *	N/A	imageProps	The image file processed
void *	N/A	parserContext	Image parser context
void *	N/A	decryptContext	Image decryption context
void *	N/A	authContext	Image authentication context
const BootloaderParserCallbacks_t *	N/A	parseCb	Bootloader parser callbacks

Returns

- Error code indicating success or failure.

bootloader_apploader_get_custom_device_address

```
void bootloader_apploader_get_custom_device_address (sl_apploader_address_t * btAddress)
```

Get custom device address.

Parameters

Type	Direction	Argument Name	Description
sl_apploader_address_t *	N/A	btAddress	Device address

The address must be set in little endian format Default implementation of this function reads the address from MFG_CUSTOM_EUI_64 manufacturing token stored in user data page. The function can be overridden if custom implementation is wanted.

bootloader_xmodem_communication_init

```
void bootloader_xmodem_communication_init (void )
```

Initialize hardware for the UART XMODEM Bootloader communication.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

bootloader_xmodem_communication_start

```
int32_t bootloader_xmodem_communication_start (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initialize communication between the UART XMODEM bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

bootloader_xmodem_communication_main

```
int32_t bootloader_xmodem_communication_main (ImageProperties_t * imageProps, const
BootloaderParserCallbacks_t * parseCb)
```

Communication main for the UART XMODEM bootloader.

Parameters

Type	Direction	Argument Name	Description
ImageProperties_t *	N/A	`imageProps`	The image file processed
const BootloaderParserCallbacks_t *	N/A	`parseCb`	Bootloader parser callbacks

Returns

- Error code indicating success or failure.

bootloader_bgapi_communication_init

```
void bootloader_bgapi_communication_init (void )
```

Initialize hardware for the BGAPI UART DFU Bootloader communication.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

bootloader_bgapi_communication_start

```
int32_t bootloader_bgapi_communication_start (void )
```

Initialize communication between the BGAPI UART DFU bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader_bgapi_communication_main

```
int32_t bootloader_bgapi_communication_main (ImageProperties_t * imageProps, ParserContext_t *
parserContext, const BootloaderParserCallbacks_t * parseCb)
```

Communication main for the BGAPI UART DFU bootloader.

Parameters

Type	Direction	Argument Name	Description
ImageProperties_t *	N/A	imageProps	The image file processed
ParserContext_t *	N/A	parserContext	Image parser context
const BootloaderParserCallbacks_t *	N/A	parseCb	Bootloader parser callbacks

Returns

- Error code indicating success or failure.

communication_init

```
void communication_init (void )
```

Initialize hardware for the communication component.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

communication_start

```
int32_t communication_start (void )
```

Initialize communication between the bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

For example, indicate that all is well to the external host.

Returns

- Error code indicating success or failure

Return values

- BOOTLOADER_OK: on success
- BOOTLOADER_ERROR_COMMUNICATION_START: on failure

communication_main

```
int32_t communication_main (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Not supposed to return until either the host signals the end of the current session or a new image has been flashed and verified.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure

Return values

- BOOTLOADER_OK: when a new image was flashed
- BOOTLOADER_ERROR_COMMUNICATION_ERROR: on communication error
- BOOTLOADER_ERROR_COMMUNICATION_DONE: when no image was received
- BOOTLOADER_ERROR_COMMUNICATION_IMAGE_ERROR: when received image is invalid

communication_shutdown

```
void communication_shutdown (void )
```

Stop communication between the bootloader and external host.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

EZSP-SPI

EZSP-SPI

By enabling the EZSP-SPI communication component, the bootloader communication interface implements the EZSP protocol over SPI.

This component makes the bootloader compatible with the legacy `ezsp-spi-bootloader` that was previously released with the EmberZnet and SL-Thread wireless stacks.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bluetooth Apploader OTA DFU

Bluetooth Apploader OTA DFU

By enabling the Apploader communication component, the bootloader communication interface implements Bluetooth Apploader over-the-air (OTA) device firmware upgrade (DFU) protocol.

Copyright © 2025 Silicon Laboratories. All rights reserved.

UART XMODEM

UART XMODEM

By enabling the UART XMODEM communication component, the bootloader communication interface implements the XMODEM-CRC protocol over UART.

This component makes the bootloader compatible with the legacy `serial-uart-bootloader` that was previously released with the EmberZnet and SL-Thread wireless stacks.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BGAPI UART DFU

BGAPI UART DFU

By enabling the BGAPI communication component, the bootloader communication interface implements the UART DFU protocol using BGAPI commands.

This component makes the bootloader compatible with the legacy UART bootloader that was previously released with the Silicon Labs Bluetooth stack.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Utils

Modules

XMODEM Parser

Copyright © 2025 Silicon Laboratories. All rights reserved.

XMODEM Parser

XMODEM Parser

Parser for XMODEM packets.

XMODEM packet parser supporting XMODEM-CRC.

Modules

XmodemPacket_t

Commands

Functions

```
void    xmodem_reset(void)
```
Reset the XMODEM parser to start a new transfer.

```
int32_t    xmodem_parsePacket(const XmodemPacket_t *packet, uint8_t *response)
```
Parse an XMODEM packet.

```
uint8_t    xmodem_getLastPacketNumber(void)
```
Return the packet number of the last packet that was successfully parsed.

Macros

```
#define    XMODEM_DATA_SIZE 128
```
Size of an XMODEM packet.

Function Documentation

xmodem_reset

```
void xmodem_reset (void )
```

Reset the XMODEM parser to start a new transfer.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

xmodem_parsePacket

```
int32_t xmodem_parsePacket (const XmodemPacket_t * packet, uint8_t * response)
```

Parse an XMODEM packet.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
const XmodemPacket_t *	[in]	`packet`	The XMODEM packet to parse.
uint8_t *	[out]	`response`	The XMODEM response to the parsed frame

Returns

- BOOTLOADER_OK on success, else error code

xmodem_getLastPacketNumber

```
uint8_t xmodem_getLastPacketNumber (void )
```

Return the packet number of the last packet that was successfully parsed.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Last packet number received. Defaults to 0 if nothing has been received yet.

Copyright © 2025 Silicon Laboratories. All rights reserved.

XmodemPacket_t

XMODEM packet.

Public Attributes

uint8_t	header	Packet header (XMODEM_CMD_SOH)
uint8_t	packetNumber	Packet sequence number.
uint8_t	packetNumberC	Complement of packet sequence number.
uint8_t	data	Payload.
uint8_t	crcH	CRC high byte.
uint8_t	crcL	CRC low byte.

Public Attribute Documentation

header

```
uint8_t XmodemPacket_t::header
```

Packet header (XMODEM_CMD_SOH)

packetNumber

```
uint8_t XmodemPacket_t::packetNumber
```

Packet sequence number.

packetNumberC

```
uint8_t XmodemPacket_t::packetNumberC
```

Complement of packet sequence number.

data

```
uint8_t XmodemPacket_t::data[XMODEM_DATA_SIZE]
```

Payload.

Copyright © 2025 Silicon Laboratories. All rights reserved.

crcH

```
uint8_t XmodemPacket_t::crcH
```

CRC high byte.

crcL

```
uint8_t XmodemPacket_t::crcL
```

CRC low byte.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Commands

Commands

Macros

`#define` `XMODEM_CMD_SOH (0x01)`
Start of Header.

`#define` `XMODEM_CMD_EOT (0x04)`
End of Transmission.

`#define` `XMODEM_CMD_ACK (0x06)`
Acknowledge.

`#define` `XMODEM_CMD_NAK (0x15)`
Not Acknowledge.

`#define` `XMODEM_CMD_CAN (0x18)`
Cancel.

`#define` `XMODEM_CMD_CTRL_C (0x03)`
Ctrl+C.

`#define` `XMODEM_CMD_C (0x43)`
ASCII 'C'.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debug

Debug

Debug Component.

This Component provides the bootloader with support for debugging functions. The Component implements two types of debugging functionality:

- Defining SL_DEBUG_ASSERT enables assertions on compile-time configurable parameters in the bootloader
- Defining SL_DEBUG_PRINT enables debug prints at strategic points in the code.

Macros

`#define` `BTL_ASSERT (exp)`
Assertion in bootloader.

`#define` `BTL_DEBUG_INIT ()`
Initialize debug output.

`#define` `BTL_DEBUG_PRINT (str)`
Print a string to debug out.

`#define` `BTL_DEBUG_PRINTLN (str)`
Print a string followed by a newline to debug out.

`#define` `BTL_DEBUG_PRINTC (chr)`
Print a character to debut out.

`#define` `BTL_DEBUG_PRINT_CHAR_HEX (number)`
Print a single hex byte.

`#define` `BTL_DEBUG_PRINT_SHORT_HEX (number)`
Print two hex bytes.

`#define` `BTL_DEBUG_PRINT_WORD_HEX (number)`
Print a hex word.

`#define` `BTL_DEBUG_PRINT_LF ()`
Print a newline.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Decompressor

Decompressor

Decompressors for Gecko Bootloader.

The decompressor module adds support for decompressing OTA upgrade files that have been compressed using the LZ4 or the LZMA compression scheme.

Modules

LZ4 Decompressor

LZMA Decompressor

Copyright © 2025 Silicon Laboratories. All rights reserved.

LZ4 Decompressor

LZ4 Decompressor

LZ4 decompressor LZ4 is a lossless data compression algorithm that is focused on compression and decompression speed.

It belongs to the LZ77 family of byte-oriented compression schemes.

Modules

Lz4Context_t

Typedefs

typedef int32_t(* Lz4DataWrite_t)(uint8_t *data, size_t length)
Function to output data from LZ4 decompressor.

typedef int32_t(* Lz4DataRead_t)(size_t backtrackOffset, uint8_t *data, size_t length)
Function to read data into LZ4 decompressor.

Functions

int32_t lz4_init(Lz4Context_t *ctx, Lz4DataRead_t readFunction)
Initialize the LZ4 decompressor.

int32_t lz4_decompress(Lz4Context_t *ctx, void *inputData, size_t inputLength, Lz4DataWrite_t writeFunction)
Decompress a chunk of data.

int32_t lz4_finish(Lz4Context_t *ctx)
Finish decompressing data.

Macros

#define LZ4_STATE_TOKEN 00U
LZ4 state machine: Token byte.

#define LZ4_STATE_LITERAL_LENGTH 10U
LZ4 state machine: Literal length byte.

#define LZ4_STATE_LITERAL_VALUE 20U
LZ4 state machine: Literal value byte.

#define LZ4_STATE_OFFSET_LSB 30U
LZ4 state machine: LSB of match offset.

#define LZ4_STATE_OFFSET_MSB 35U
LZ4 state machine: MSB of match offset.

#define LZ4_STATE_MATCH_LENGTH 40U
LZ4 state machine: Match length.

#define LZ4_STATE_BACKTRACKING 50U
LZ4 state machine: Backtracking to get matched data.

#define OUTPUT_BUFFER_SIZE 4

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#define    ALIGNMENT_SIZE 4
```

Typedef Documentation

Lz4DataWrite_t

```
typedef int32_t(* Lz4DataWrite_t) (uint8_t *data, size_t length) )(uint8_t *data, size_t
length)
```

Function to output data from LZ4 decompressor.

Lz4DataRead_t

```
typedef int32_t(* Lz4DataRead_t) (size_t backtrackOffset, uint8_t *data, size_t length) )
(size_t backtrackOffset, uint8_t *data, size_t length)
```

Function to read data into LZ4 decompressor.

Function Documentation

lz4_init

```
int32_t lz4_init (Lz4Context_t * ctx, Lz4DataRead_t readFunction)
```

Initialize the LZ4 decompressor.

Parameters

Type	Direction	Argument Name	Description
Lz4Context_t *	N/A	ctx	Decompressor context
Lz4DataRead_t	N/A	readFunction	Function pointer to read back previously written data

Returns

- Error code

lz4_decompress

```
int32_t lz4_decompress (Lz4Context_t * ctx, void * inputData, size_t inputLength,
Lz4DataWrite_t writeFunction)
```

Decompress a chunk of data.

Parameters

Type	Direction	Argument Name	Description
Lz4Context_t *	N/A	ctx	Decompressor context
void *	N/A	inputData	Compressed input data
size_t	N/A	inputLength	Length of inputData in bytes
Lz4DataWrite_t	N/A	writeFunction	Function pointer that is called with decompressed data

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- Error code

lz4_finish

```
int32_t lz4_finish (Lz4Context_t * ctx)
```

Finish decompressing data.

Parameters

Type	Direction	Argument Name	Description
Lz4Context_t *	N/A	ctx	Decompressor context

Returns

- Error code indicating success or failure

Return values

- BOOTLOADER_OK: on success
- BOOTLOADER_ERROR_COMPRESSION_STATE: if the last block wasn't completed

Copyright © 2025 Silicon Laboratories. All rights reserved.

Lz4Context_t

LZ4 decompressor context.

Public Attributes

uint32_t	literalLength Length of literals.
uint32_t	matchLength Length of match.
uint16_t	backtrackOffset Offset from current write for match.
uint8_t	state Current decompressor state.
Lz4DataRead_t	readFunction Function to read previously written data into decompressor during backtracking.

Public Attribute Documentation

literalLength

```
uint32_t Lz4Context_t::literalLength
```

Length of literals.

matchLength

```
uint32_t Lz4Context_t::matchLength
```

Length of match.

backtrackOffset

```
uint16_t Lz4Context_t::backtrackOffset
```

Offset from current write for match.

state

```
uint8_t Lz4Context_t::state
```

Current decompressor state.

readFunction

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Lz4DataRead_t Lz4Context_t::readFunction
```

Function to read previously written data into decompressor during backtracking.

Copyright © 2025 Silicon Laboratories. All rights reserved.

LZMA Decompressor

LZMA Decompressor

LZMA is a lossless data compression algorithm.

LZMA uses an enhanced LZ77 algorithm with large dictionaries, generating a stream of literals and references. This stream is then optimally encoded bit by bit using a range encoder with complex probability models. Custom GBL Tags for more information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GPIO Activation

GPIO Activation

Enter bootloader based on GPIO state.

This component provides functionality to enter firmware upgrade mode automatically after reset if a GPIO pin is active during boot.

The GPIO pin location and polarity are configurable.

Modules

Button GPIO

EZSP GPIO

Copyright © 2025 Silicon Laboratories. All rights reserved.

Button GPIO

Button GPIO

Enter bootloader based on Button GPIO state.

By enabling the GPIO activation component, the firmware upgrade mode can be activated by the GPIO configuration.

Functions

`bool` `gpio_enterBootloader(void)`
Enter the bootlader if the GPIO pin is active.

Function Documentation

gpio_enterBootloader

```
bool gpio_enterBootloader (void )
```

Enter the bootlader if the GPIO pin is active.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the bootloader should be entered

Copyright © 2025 Silicon Laboratories. All rights reserved.

EZSP GPIO

EZSP GPIO

Enter bootloader based on EZSP GPIO state.

The EZSP communication protocol over SPI can be used together with this component.

By enabling the EZSP GPIO component, the firmware upgrade mode can be entered by activating the nWake pin.

Functions

`bool` `ezsp_gpio_enterBootloader(void)`
Enter the bootlader if the GPIO pin is active.

Function Documentation

ezsp_gpio_enterBootloader

```
bool ezsp_gpio_enterBootloader (void )
```

Enter the bootlader if the GPIO pin is active.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the bootloader should be entered

Copyright © 2025 Silicon Laboratories. All rights reserved.

Image Parser

Image Parser

The image parser parses the data and returns bootloader upgrade data in a callback.

Modules

GBL Parser

Copyright © 2025 Silicon Laboratories. All rights reserved.

GBL Parser

GBL Parser

GBL parser implementation.

Image parser for GBL files. Parses GBL files based on the GBL file format specification. Callbacks are used to present data and metadata contents of the GBL file to the bootloader.

Modules

GblTagParsingInfo_t

ImageProperties_t

ParserContext_t

GblInputBuffer_t

Custom GBL Tags

GBL Format

Enumerations

```
enum    GblParserState_t {
          GblParserStateInit
          GblParserStateIdle
          GblParserStateHeader
          GblParserStateBootloader
          GblParserStateBootloaderData
          GblParserStateApplication
          GblParserStateMetadata
          GblParserStateMetadataData
          GblParserStateProg
          GblParserStateProgData
          GblParserStateEraseProg
          GblParserStateFinalize
          GblParserStateDone
          GblParserStateEncryptionInit
          GblParserStateEncryptionContainer
          GblParserStateSignature
          GblParserStateError
        }
```

State in the GBL parser state machine.

Functions

```
const GblTagParsingInfo_t *    gbl_getTagParsingInfoFromTagId(uint32_t tagId)
```

Function for looking up and retrieving the parsing information struct associated with a particular GBL tag type / tag ID.

```
int32_t    parser_init(void *context, void *decryptContext, void *authContext, uint8_t flags)
```

Initialize the parser's context.

```
int32_t    parser_parse(void *context, ImageProperties_t *imageProperties, uint8_t buffer[],
           size_t length, const BootloaderParserCallbacks_t *callbacks)
```

Parse an image file to extract the binary and some metadata.

```
int32_t    parser_verifyCertificate(void *context, void *input, void *blProperties, void
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

*shaState)
Verify the GBL certificate.

int32_t gbl_writeProgData(ParserContext_t *context, uint8_t buffer[], size_t length, const BootloaderParserCallbacks_t *callbacks)
Write application data.

Macros

#define PARSER_FLAG_ENCRYPTED (1U << 0U)
GBL file is encrypted.

#define PARSER_FLAG_PARSE_CUSTOM_TAGS (1U << 5U)
Parse custom tags rather than silently traversing them.

#define PARSER_FLAGS_PUBLIC_MASK PARSER_FLAG_PARSE_CUSTOM_TAGS
Some flags are public, some are internal to the parser.

#define GBL_PARSER_BUFFER_SIZE 64UL
GBL parser buffer size.

#define PARSER_REQUIRE_AUTHENTICITY (false)
Bootloader/parser configurations.

#define PARSER_REQUIRE_CONFIDENTIALITY (false)
Parser requires upgrade images to be encrypted, providing confidentiality, if true.

#define PARSER_REQUIRE_CERTIFICATE_AUTHENTICITY (false)
Parser requires upgrade images to be authenticated by the bootloader certificate, if true.

#define PARSER_REQUIRE_ANTI_ROLLBACK_PROTECTION (false)
Parser requires rollback protection of applications, if true.

#define PARSER_APPLICATION_MINIMUM_VERSION_VALID (0UL)
Defines the minimum application version that can be accepted.

#define BTL_IMAGE_CONTENT_APPLICATION 0x01U
Upgrade image contains application upgrade.

#define BTL_IMAGE_CONTENT_BOOTLOADER 0x02U
Upgrade image contains bootloader upgrade.

#define BTL_IMAGE_CONTENT_SE 0x04U
Upgrade image contains SE upgrade.

#define BTL_IMAGE_CONTENT_DELTA 0x08U
Upgrade image contains Delta upgrade.

#define BTL_IMAGE_INSTRUCTION_APPLICATION 0x01U
Application upgrade should be applied from upgrade image.

#define BTL_IMAGE_INSTRUCTION_BOOTLOADER 0x02U
Bootloader upgrade should be applied from upgrade image.

#define BTL_IMAGE_INSTRUCTION_SE 0x04U
SE upgrade should be applied from upgrade image.

Enumeration Documentation

GblParserState_t

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
GblParserState_t
```

State in the GBL parser state machine.

Enumerator	
GblParserStateInit	Initial state.
GblParserStateIdle	Idle state.
GblParserStateHeader	Parsing header tag.
GblParserStateBootloader	Parsing bootloader tag.
GblParserStateBootloaderData	Parsing bootloader tag data.
GblParserStateApplication	Parsing application tag.
GblParserStateMetadata	Parsing metadata tag.
GblParserStateMetadataData	Parsing metadata tag data.
GblParserStateProg	Parsing flash program tag.
GblParserStateProgData	Parsing flash program tag data.
GblParserStateEraseProg	Parsing flash erase&program tag.
GblParserStateFinalize	Finalizing file.
GblParserStateDone	Parsing complete.
GblParserStateEncryptionInit	Parsing encryption init tag.
GblParserStateEncryptionContainer	Parsing encryption data tag.
GblParserStateSignature	Parsing signature tag.
GblParserStateError	Error state.

Function Documentation

gbl_getTagParsingInfoFromTagId

```
const GblTagParsingInfo_t * gbl_getTagParsingInfoFromTagId (uint32_t tagId)
```

Function for looking up and retrieving the parsing information struct associated with a particular GBL tag type / tag ID.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`tagId`	The tag ID to be looked up.

Returns

- A pointer to the GblTagParsingInfo_t struct associated with tagId, or a NULL pointer if the provided tagId cannot be found.

parser_init

```
int32_t parser_init (void * context, void * decryptContext, void * authContext, uint8_t flags)
```

Initialize the parser's context.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void *	N/A	context	Pointer to context for the parser implementation
void *	N/A	decryptContext	Pointer to context for decryption of parsed file
void *	N/A	authContext	Pointer to context for authentication of parsed file
uint8_t	N/A	flags	Flags for parser support

Returns

- BOOTLOADER_OK if OK, error code otherwise.

parser_parse

```
int32_t parser_parse (void * context, ImageProperties_t * imageProperties, uint8_t buffer,
size_t length, const BootloaderParserCallbacks_t * callbacks)
```

Parse an image file to extract the binary and some metadata.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	context	Pointer to the specific parser's context variable
ImageProperties_t *	N/A	imageProperties	Pointer to the image file state variable
uint8_t	N/A	buffer	Pointer to byte array containing data to parse
size_t	N/A	length	Size in bytes of the data in buffer
const BootloaderParserCallbacks_t *	N/A	callbacks	Struct containing function pointers to be called by the parser to pass the extracted binary data back to BTL.

Pushes data into the image file parser to be parsed.

Returns

- BOOTLOADER_OK if OK, error code otherwise.

parser_verifyCertificate

```
int32_t parser_verifyCertificate (void * context, void * input, void * blProperties, void *
shaState)
```

Verify the GBL certificate.

Parameters

Type	Direction	Argument Name	Description
void *	[inout]	context	GBL parser context
void *	[in]	input	Input data
void *	[in]	blProperties	Pointer to ApplicationProperties_t of bootloader
void *	[inout]	shaState	Pointer to Sha256Context_t used to store SHA256 of GBL certificate

Note

- The behavior of this function depends on the context state.

Returns

Copyright © 2025 Silicon Laboratories. All rights reserved.

BOOTLOADER_ERROR_PARSER_PARSED if done parsing the current input buffer. BOOTLOADER_OK if input data is stored in the internal buffer. BOOTLOADER_OK if the certificate in GBL is accepted. BOOTLOADER_ERROR_PARSER_SIGNATURE if the certificate in GBL is rejected.

gbl_writeProgData

```
int32_t gbl_writeProgData (ParserContext_t * context, uint8_t buffer, size_t length, const
BootloaderParserCallbacks_t * callbacks)
```

Write application data.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	context	GBL parser context
uint8_t	N/A	buffer	Input buffer containing data to be written
size_t	N/A	length	Size of input buffer
const BootloaderParserCallbacks_t *	N/A	callbacks	GBL Parser callbacks for writing data

This function is called when parsing any tag with GblProg_t structured content.

Returns

- Error code

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblTagParsingInfo_t

GBL tag parsing info struct. Contains information about the GBL tag order, etc.

Public Attributes

uint32_t	tagId Tag ID.
GblParserState_t	parserState The parser state associated with the tag.
uint8_t	tagOrder Encodes correct order of occurrence in GBL.
uint8_t	reserved Reserved.
uint16_t	flags Flags defining parser behavior.

Public Attribute Documentation

tagId

```
uint32_t GblTagParsingInfo_t::tagId
```

Tag ID.

parserState

```
GblParserState_t GblTagParsingInfo_t::parserState
```

The parser state associated with the tag.

tagOrder

```
uint8_t GblTagParsingInfo_t::tagOrder
```

Encodes correct order of occurrence in GBL.

reserved

```
uint8_t GblTagParsingInfo_t::reserved
```

Reserved.

flags

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint16_t GblTagParsingInfo_t::flags
```

Flags defining parser behavior.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ImageProperties_t

Structure containing state of the image file processed.

Public Attributes

uint8_t	contents Image contents.
bool	imageCompleted Flag to indicate parsing has completed.
bool	imageVerified Flag to indicate the image file has been validated.
uint32_t	bootloaderVersion Version number of main bootloader extracted from image file.
uint32_t	bootloaderUpgradeSize Size of the bootloader upgrade contained in the image file.
uint8_t	instructions Parser instructions.
ApplicationData_t	application Information about the application.

Public Attribute Documentation

contents

```
uint8_t ImageProperties_t::contents
```

Image contents.

imageCompleted

```
bool ImageProperties_t::imageCompleted
```

Flag to indicate parsing has completed.

imageVerified

```
bool ImageProperties_t::imageVerified
```

Flag to indicate the image file has been validated.

bootloaderVersion

```
uint32_t ImageProperties_t::bootloaderVersion
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Version number of main bootloader extracted from image file.

bootloaderUpgradeSize

```
uint32_t ImageProperties_t::bootloaderUpgradeSize
```

Size of the bootloader upgrade contained in the image file.

instructions

```
uint8_t ImageProperties_t::instructions
```

Parser instructions.

application

```
ApplicationData_t ImageProperties_t::application
```

Information about the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ParserContext_t

Image parser context definition.

Public Attributes

uint8_t	internalBuffer Buffer contents.
uint8_t	bytesInInternalBuffer Amount of bytes present in buffer.
uint8_t	internalBufferOffset Current reading offset into the buffer (circular)
uint8_t	flags Parser flags.
bool	inEncryptedContainer Parser is currently inside an encrypted tag.
bool	gotSignature Parser has received and verified signature.
uint8_t	receivedFlags Parser has received bootloader upgrade tag.
GblParserState_t	internalState State of the GBL parser state machine.
void *	aesContext AES-CCM decryption (= AES-CTR) context.
void *	shaContext SHA256 hashing context.
size_t	lengthOfTag Total length of the tag currently being parsed.
size_t	offsetInTag Current offset into tag being parsed.
size_t	lengthOfEncryptedTag Total length of current encrypted data block.
size_t	offsetInEncryptedTag Offset into current encrypted data block.
uint32_t	programmingAddress Current address the image needs to be written to.
uint32_t	tagAddress Current offset of metadata/bootloader being handled (starts at 0)
uint8_t	withheldApplicationVectors Withheld application data.
uint8_t	withheldUpgradeVectors Withheld bootloader upgrade data during app parsing.
uint8_t	withheldBootloaderVectors Withheld bootloader upgrade data during bootloader parsing.

Copyright © 2025 Silicon Laboratories. All rights reserved.

uint32_t	fileCrc	Running CRC-32 over the incoming GBL file.
uint32_t	customTagId	Context for custom tag.
uint8_t	currentTagOrder	Current tag order.
uint8_t	reservedFlags	Reserved flags.

Public Attribute Documentation

internalBuffer

```
uint8_t ParserContext_t::internalBuffer[64]
```

Buffer contents.

bytesInInternalBuffer

```
uint8_t ParserContext_t::bytesInInternalBuffer
```

Amount of bytes present in buffer.

internalBufferOffset

```
uint8_t ParserContext_t::internalBufferOffset
```

Current reading offset into the buffer (circular)

flags

```
uint8_t ParserContext_t::flags
```

Parser flags.

inEncryptedContainer

```
bool ParserContext_t::inEncryptedContainer
```

Parser is currently inside an encrypted tag.

gotSignature

```
bool ParserContext_t::gotSignature
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parser has received and verified signature.

receivedFlags

```
uint8_t ParserContext_t::receivedFlags
```

Parser has received bootloader upgrade tag.

internalState

```
GblParserState_t ParserContext_t::internalState
```

State of the GBL parser state machine.

aesContext

```
void* ParserContext_t::aesContext
```

AES-CCM decryption (= AES-CTR) context.

shaContext

```
void* ParserContext_t::shaContext
```

SHA256 hashing context.

lengthOfTag

```
size_t ParserContext_t::lengthOfTag
```

Total length of the tag currently being parsed.

offsetInTag

```
size_t ParserContext_t::offsetInTag
```

Current offset into tag being parsed.

lengthOfEncryptedTag

```
size_t ParserContext_t::lengthOfEncryptedTag
```

Total length of current encrypted data block.

Copyright © 2025 Silicon Laboratories. All rights reserved.

offsetInEncryptedTag

```
size_t ParserContext_t::offsetInEncryptedTag
```

Offset into current encrypted data block.

programmingAddress

```
uint32_t ParserContext_t::programmingAddress
```

Current address the image needs to be written to.

tagAddress

```
uint32_t ParserContext_t::tagAddress
```

Current offset of metadata/bootloader being handled (starts at 0)

withheldApplicationVectors

```
uint8_t ParserContext_t::withheldApplicationVectors[24]
```

Withheld application data.

withheldUpgradeVectors

```
uint8_t ParserContext_t::withheldUpgradeVectors[4]
```

Withheld bootloader upgrade data during app parsing.

withheldBootloaderVectors

```
uint8_t ParserContext_t::withheldBootloaderVectors[4]
```

Withheld bootloader upgrade data during bootloader parsing.

fileCrc

```
uint32_t ParserContext_t::fileCrc
```

Running CRC-32 over the incoming GBL file.

customTagId

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint32_t ParserContext_t::customTagId
```

Context for custom tag.

currentTagOrder

```
uint8_t ParserContext_t::currentTagOrder
```

Current tag order.

reservedFlags

```
uint8_t ParserContext_t::reservedFlags[3]
```

Reserved flags.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblInputBuffer_t

GBL parser input buffer.

Public Attributes

const uint8_t *	buffer Pointer to a buffer.
const size_t	length Length of the buffer.
size_t	offset Offset of the buffer.

Public Attribute Documentation

buffer

```
const uint8_t* GblInputBuffer_t::buffer
```

Pointer to a buffer.

length

```
const size_t GblInputBuffer_t::length
```

Length of the buffer.

offset

```
size_t GblInputBuffer_t::offset
```

Offset of the buffer.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Custom GBL Tags

Custom GBL Tags

Handle custom GBL tags for added functionality in the GBL parser.

Modules

GblCustomTag_t

LZ4 Programming Tag

LZMA Programming Tag

Functions

bool `gbl_isCustomTag(const GblTagHeader_t *tagHeader)`
Indicate whether the GBL tag described by tagHeader is a custom tag.

const GblCustomTag_t * `gbl_getCustomTagProperties(uint32_t tagId)`
Get properties for a custom GBL tag.

Function Documentation

gbl_isCustomTag

```
bool gbl_isCustomTag (const GblTagHeader_t * tagHeader)
```

Indicate whether the GBL tag described by tagHeader is a custom tag.

Parameters

Type	Direction	Argument Name	Description
const GblTagHeader_t *	N/A	tagHeader	Pointer to the GBL tag header

Returns

- True if the tag is a custom tag, else false

gbl_getCustomTagProperties

```
const GblCustomTag_t * gbl_getCustomTagProperties (uint32_t tagId)
```

Get properties for a custom GBL tag.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	tagId	GBL Tag ID of the custom tag

Returns

-

Copyright © 2025 Silicon Laboratories. All rights reserved.

Pointer to the custom tag descriptor

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblCustomTag_t

Custom tag descriptor.

Public Attributes

uint32_t	tagId GBL Tag ID of the custom tag.
int32_t(*	enterTag Function to call upon entering the tag.
int32_t(*	parseTag Function to call while parsing the tag.
int32_t(*	exitTag Function to call upon exiting the tag.
size_t(*	numBytesRequired Function returning how many bytes should be collected before calling parseTag the next time.

Public Attribute Documentation

tagId

```
uint32_t GblCustomTag_t::tagId
```

GBL Tag ID of the custom tag.

enterTag

```
int32_t(* GblCustomTag_t::enterTag) (ParserContext_t *ctx)
```

Function to call upon entering the tag.

parseTag

```
int32_t(* GblCustomTag_t::parseTag) (ParserContext_t *ctx, void *data, size_t length, const
BootloaderParserCallbacks_t *callbacks)
```

Function to call while parsing the tag.

exitTag

```
int32_t(* GblCustomTag_t::exitTag) (ParserContext_t *ctx, const BootloaderParserCallbacks_t
*callbacks)
```

Function to call upon exiting the tag.

Copyright © 2025 Silicon Laboratories. All rights reserved.

numBytesRequired

```
size_t(* GblCustomTag_t::numBytesRequired) (ParserContext_t *ctx)
```

Function returning how many bytes should be collected before calling parseTag the next time.

Copyright © 2025 Silicon Laboratories. All rights reserved.

LZ4 Programming Tag

LZ4 Programming Tag

Tag to handle LZ4 compressed programming data.

Modules

Lz4ParserContext_t

Functions

int32_t gbl_lz4EnterProgTag(ParserContext_t *ctx)
Enter an LZ4 compressed programming tag.

int32_t gbl_lz4ParseProgTag(ParserContext_t *ctx, void *data, size_t length, const BootloaderParserCallbacks_t *callbacks)
Parse a chunk of data from an LZ4 compressed programming tag.

int32_t gbl_lz4ExitProgTag(ParserContext_t *ctx, const BootloaderParserCallbacks_t *callbacks)
Exit an LZ4 compressed programming tag.

size_t gbl_lz4NumBytesRequired(ParserContext_t *ctx)
Number of bytes needed for the next stage of parsing.

Function Documentation

gbl_lz4EnterProgTag

```
int32_t gbl_lz4EnterProgTag (ParserContext_t * ctx)
```

Enter an LZ4 compressed programming tag.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context

Returns

- Error code

gbl_lz4ParseProgTag

```
int32_t gbl_lz4ParseProgTag (ParserContext_t * ctx, void * data, size_t length, const BootloaderParserCallbacks_t * callbacks)
```

Parse a chunk of data from an LZ4 compressed programming tag.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context
void *	N/A	data	Input data to parse
size_t	N/A	length	Length of data
const BootloaderParserCallbacks_t *	N/A	callbacks	Callbacks to call with parsed data

Returns

- Error code

gbl_lz4ExitProgTag

```
int32_t gbl_lz4ExitProgTag (ParserContext_t * ctx, const BootloaderParserCallbacks_t *
callbacks)
```

Exit an LZ4 compressed programming tag.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context
const BootloaderParserCallbacks_t *	N/A	callbacks	Callbacks to call with parsed data

Returns

- Error code

gbl_lz4NumBytesRequired

```
size_t gbl_lz4NumBytesRequired (ParserContext_t * ctx)
```

Number of bytes needed for the next stage of parsing.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context

Returns

- Number of bytes required

Copyright © 2025 Silicon Laboratories. All rights reserved.

Lz4ParserContext_t

LZ4 Compressed Programming GBL Tag Parser Context.

Public Attributes

uint8_t	outputBuffer Buffer to store unaligned decompressed data.
uint32_t	outputOffset Offset into outputBuffer.
bool	firstCall Whether this is the first call to the parser for this tag.
ParserContext_t *	parserContext Stored pointer to the GBL parser context.
const BootloaderParserCallbacks_t *	parserCallbacks Stored pointer to the GBL parser callbacks.
Lz4Context_t	lz4Context Context of the LZ4 decompressor.

Public Attribute Documentation

outputBuffer

```
uint8_t Lz4ParserContext_t::outputBuffer[OUTPUT_BUFFER_SIZE]
```

Buffer to store unaligned decompressed data.

outputOffset

```
uint32_t Lz4ParserContext_t::outputOffset
```

Offset into outputBuffer.

firstCall

```
bool Lz4ParserContext_t::firstCall
```

Whether this is the first call to the parser for this tag.

parserContext

```
ParserContext_t* Lz4ParserContext_t::parserContext
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Stored pointer to the GBL parser context.

parserCallbacks

```
const BootloaderParserCallbacks_t* Lz4ParserContext_t::parserCallbacks
```

Stored pointer to the GBL parser callbacks.

lz4Context

```
Lz4Context_t Lz4ParserContext_t::lz4Context
```

Context of the LZ4 decompressor.

Copyright © 2025 Silicon Laboratories. All rights reserved.

LZMA Programming Tag

LZMA Programming Tag

Tag to handle LZMA compressed programming data.

Functions

`int32_t` `gbl_lzmaEnterProgTag(ParserContext_t *ctx)`
Enter an LZMA compressed programming tag.

`int32_t` `gbl_lzmaParseProgTag(ParserContext_t *ctx, void *data, size_t length, const BootloaderParserCallbacks_t *callbacks)`
Parse a chunk of data from an LZMA compressed programming tag.

`int32_t` `gbl_lzmaExitProgTag(ParserContext_t *ctx, const BootloaderParserCallbacks_t *callbacks)`
Exit an LZMA compressed programming tag.

`size_t` `gbl_lzmaNumBytesRequired(ParserContext_t *ctx)`
Number of bytes needed for next stage of parsing.

Macros

`#define` `LZMA_COUNTER_SIZE_KB (10UL)`
The maximum size of the array holding probability model counters.

`#define` `LZMA_DICT_SIZE_KB (8UL)`
The maximum size of the dictionary.

Function Documentation

gbl_lzmaEnterProgTag

```
int32_t gbl_lzmaEnterProgTag (ParserContext_t * ctx)
```

Enter an LZMA compressed programming tag.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context

Returns

- Error code

gbl_lzmaParseProgTag

```
int32_t gbl_lzmaParseProgTag (ParserContext_t * ctx, void * data, size_t length, const BootloaderParserCallbacks_t * callbacks)
```

Parse a chunk of data from an LZMA compressed programming tag.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context
void *	N/A	data	Input data to parse
size_t	N/A	length	Length of data
const BootloaderParserCallbacks_t *	N/A	callbacks	Callbacks to call with parsed data

Returns

- Error code

gbl_lzmaExitProgTag

```
int32_t gbl_lzmaExitProgTag (ParserContext_t * ctx, const BootloaderParserCallbacks_t *
callbacks)
```

Exit an LZMA compressed programming tag.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context
const BootloaderParserCallbacks_t *	N/A	callbacks	Callbacks to call with parsed data

Returns

- Error code

gbl_lzmaNumBytesRequired

```
size_t gbl_lzmaNumBytesRequired (ParserContext_t * ctx)
```

Number of bytes needed for next stage of parsing.

Parameters

Type	Direction	Argument Name	Description
ParserContext_t *	N/A	ctx	Parser context

Returns

- Number of bytes required

Copyright © 2025 Silicon Laboratories. All rights reserved.

GBL Format

GBL Format

Modules

- GblTagHeader_t
- GblHeader_t
- VersionDependency_t
- GblApplication_t
- GblBootloader_t
- GblTagDelta_t
- GblSeUpgrade_t
- GblMetadata_t
- GblProg_t
- GblEnd_t
- GblEncryptionInitAesCcm_t
- GblEncryptionData_t
- GblCertificateEcdsaP256_t
- GblSignatureEcdsaP256_t

Macros

```
#define  GBL_IMAGE_MAGIC_WORD 0xE35050E3UL
```
Magic word indicating GBL image.

```
#define  GBL_COMPATIBILITY_MAJOR_VERSION 0x03000000UL
```
Major version of the GBL spec.

```
#define  GBL_TAG_ID_HEADER_V3 0x03A617EBUL
```
Tag ID for the GBL header tag.

```
#define  GBL_TAG_ID_BOOTLOADER 0xF50909F5UL
```
Tag ID for the GBL bootloader tag.

```
#define  GBL_TAG_ID_APPLICATION 0xF40A0AF4UL
```
Tag ID for the GBL application info tag.

```
#define  GBL_TAG_ID_METADATA 0xF60808F6UL
```
Tag ID for the GBL metadata tag.

```
#define  GBL_TAG_ID_PROG 0xFE0101FEUL
```
Tag ID for the GBL flash program tag.

```
#define  GBL_TAG_ID_PROG_LZ4 0xFD0505FDUL
```
Tag ID for the GBL flash program tag (LZ4)

```
#define  GBL_TAG_ID_PROG_LZMA 0xFD0707FDUL
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tag ID for the GBL flash program tag (LZMA)

#define GBL_TAG_ID_ERASEPROG 0xFD0303FDUL
Tag ID for the GBL flash erase&program tag.

#define GBL_TAG_ID_DELTA 0xF80A0AF8UL
Tag ID for the GBL delta dfu tag.

#define GBL_TAG_ID_DELTA_LZ4 0xF80B0BF8UL
Tag ID for the GBL delta dfu tag (LZ4)

#define GBL_TAG_ID_DELTA_LZMA 0xF80C0CF8UL
Tag ID for the GBL delta dfu tag (LZMA)

#define GBL_TAG_ID_END 0xFC0404FCUL
Tag ID for the GBL end tag.

#define GBL_TAG_ID_SE_UPGRADE 0x5EA617EBUL
Tag ID for the SE upgrade tag.

#define GBL_TAG_ID_VERSION_DEPENDENCY 0x76A617EBUL
Tag ID for the version dependency tag.

#define GBL_TAG_ID_ENC_HEADER 0xFB0505FBUL
Tag ID for the GBL encryption header tag.

#define GBL_TAG_ID_ENC_INIT 0xFA0606FAUL
Tag ID for the GBL encryption init tag.

#define GBL_TAG_ID_ENC_GBL_DATA 0xF90707F9UL
Tag ID for the GBL encryption data tag.

#define GBL_TAG_ID_SIGNATURE_ECDSA_P256 0xF70A0AF7UL
Tag ID for the GBL ECDSA secp256r1 signature tag.

#define GBL_TAG_ID_CERTIFICATE_ECDSA_P256 0xF30B0BF3UL
Tag ID for the GBL ECDSA certfificate tag.

#define GBL_TYPE_NONE 0x00000000UL
GBL type: Standard GBL.

#define GBL_TYPE_ENCRYPTION_AESCCM 0x00000001UL
GBL type: AES-CCM encrypted GBL.

#define GBL_TYPE_SIGNATURE_ECDSA 0x00000100UL
GBL type: ECDSA P256-signed GBL.

#define GBL_VERSION_DEPENDENCY_TYPE_APPLICATION 0x01U
Image Type : Application Image.

#define GBL_VERSION_DEPENDENCY_TYPE_BOOTLOADER 0x02U
Image Type : Bootloader Image.

#define GBL_VERSION_DEPENDENCY_TYPE_SE 0x03U
Image Type : Secure Engine Image.

#define GBL_VERSION_DEPENDENCY_OPERATOR_MASK 0x0FU
Operator encoding : Operator Mask.

#define GBL_VERSION_DEPENDENCY_OPERATOR_SHIFT 0x00U
Operator encoding : Operator Shift.

#define GBL_VERSION_DEPENDENCY_OPERATOR_TYPE_MASK 0x0EU
Operator encoding : Operator Type Mask.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define GBL_VERSION_DEPENDENCY_OPERATOR_NEGATOR_BIT_MASK 0x01U
Operator encoding : Negator Bit mask.

#define GBL_VERSION_DEPENDENCY_OPERATOR_LT 0x00U
GBL Version Dependency Operator LT.

#define GBL_VERSION_DEPENDENCY_OPERATOR_LEQ 0x02U
GBL Version Dependency Operator LEQ.

#define GBL_VERSION_DEPENDENCY_OPERATOR_EQ 0x04U
GBL Version Dependency Operator EQ.

#define GBL_VERSION_DEPENDENCY_OPERATOR_GEQ 0x06U
GBL Version Dependency Operator GEQ.

#define GBL_VERSION_DEPENDENCY_OPERATOR_GT 0x08U
GBL Version Dependency Operator GT.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_MASK 0xF0U
Connective encoding : Connective Encoding Mask.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_SHIFT 0x04U
Connective encoding : Encoding Shift.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_TYPE_MASK 0x0EU
Connective encoding : Encoding Type Mask.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_NEGATOR_BIT_MASK 0x01U
Connective encoding : Negator Bit Mask.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_AND 0x00U
Connective AND Mask.

#define GBL_VERSION_DEPENDENCY_CONNECTIVE_OR 0x02U
Connective OR Mask.

#define GBL_VERSION_DEPENDENCY_SE_VERSION_MASK 0x00FFFFFFUL
SE version mask for ignoring the compatibility byte when comparing versions.

#define GBL_TAG_ORDER_INIT 0U
Tag order.

#define GBL_TAG_ORDER_HEADER_V3 1U
GBL Tag Order Header.

#define GBL_TAG_ORDER_VERSION_DEPENDENCY 2U
GBL Tag Order Version Dependency.

#define GBL_TAG_ORDER_ENC_INIT 4U
GBL Tag Order encoding Init.

#define GBL_TAG_ORDER_APPLICATION 5U
GBL Tag Order Application.

#define GBL_TAG_ORDER_SE_UPGRADE 6U
GBL Tag Order SE Upgarde.

#define GBL_TAG_ORDER_BOOTLOADER 7U
GBL Tag Order Bootloader.

#define GBL_TAG_ORDER_PROG_AND_METADATA 8U
The Programming and Metadata tags share the same order.

#define GBL_TAG_ORDER_ENC_GBL_DATA 9U
GBL Tag Order Enc GBL Data.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define GBL_TAG_ORDER_CERTIFICATE 10U

GBL Tag Order Certificate.

#define GBL_TAG_ORDER_SIGNATURE 11U

GBL Tag Order Signature.

#define GBL_TAG_ORDER_END 12U

GBL_TAG_ORDER_END must always be strictly greater than any other tag order.

#define GBL_TAG_FLAG_SINGLE_OCCURRENCE_ONLY (1UL << 0)

Flags.

#define GBL_TAG_FLAG_ALWAYS_UNENCRYPTED (1UL << 1)

Flags.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblTagHeader_t

GBL tag header. Must be the first element in all GBL tags.

Public Attributes

uint32_t	tagId Tag ID.
uint32_t	length Length (in bytes) of the rest of the tag.

Public Attribute Documentation

tagId

```
uint32_t GblTagHeader_t::tagId
```

Tag ID.

length

```
uint32_t GblTagHeader_t::length
```

Length (in bytes) of the rest of the tag.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblHeader_t

GBL header tag type.

Public Attributes

GblTagHeader_t	header	Tag ID and length.
uint32_t	version	Version of the GBL spec used in this file.
uint32_t	type	Type of GBL.

Public Attribute Documentation

header

```
GblTagHeader_t GblHeader_t::header
```

Tag ID and length.

version

```
uint32_t GblHeader_t::version
```

Version of the GBL spec used in this file.

type

```
uint32_t GblHeader_t::type
```

Type of GBL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

VersionDependency_t

GBL version dependency tag type.

Public Attributes

uint8_t	imageType Type of image (application, bootloader, SE)
uint8_t	statement Encoded dependency statement (ex. appVersion > (0).1.2.3)
uint16_t	reserved Reserved.
uint32_t	version The version number used in the statement (ex. (0).1.2.3)

Public Attribute Documentation

imageType

```
uint8_t VersionDependency_t::imageType
```

Type of image (application, bootloader, SE)

statement

```
uint8_t VersionDependency_t::statement
```

Encoded dependency statement (ex. appVersion > (0).1.2.3)

reserved

```
uint16_t VersionDependency_t::reserved
```

Reserved.

version

```
uint32_t VersionDependency_t::version
```

The version number used in the statement (ex. (0).1.2.3)

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblApplication_t

GBL application tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
ApplicationData_t	appInfo Information about the application.

Public Attribute Documentation

header

```
GblTagHeader_t GblApplication_t::header
```

Tag ID and length.

appInfo

```
ApplicationData_t GblApplication_t::appInfo
```

Information about the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblBootloader_t

GBL bootloader tag type.

Public Attributes

GblTagHeader_t	**header** Tag ID and length.
uint32_t	**bootloaderVersion** Version number of the bootloader.
uint32_t	**address** Address of the bootloader.
uint8_t *	**data** Array of data for bootloader upgrade.

Public Attribute Documentation

header

```
GblTagHeader_t GblBootloader_t::header
```

Tag ID and length.

bootloaderVersion

```
uint32_t GblBootloader_t::bootloaderVersion
```

Version number of the bootloader.

address

```
uint32_t GblBootloader_t::address
```

Address of the bootloader.

data

```
uint8_t* GblBootloader_t::data
```

Array of data for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblTagDelta_t

GBL delta tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint32_t	newCrc CRC for new app.
uint32_t	newSize Size of the new app after re-creation.
uint32_t	flashAddr Final flashing address of app.
uint8_t *	data Array of data to flash.

Public Attribute Documentation

header

```
GblTagHeader_t GblTagDelta_t::header
```

Tag ID and length.

newCrc

```
uint32_t GblTagDelta_t::newCrc
```

CRC for new app.

newSize

```
uint32_t GblTagDelta_t::newSize
```

Size of the new app after re-creation.

flashAddr

```
uint32_t GblTagDelta_t::flashAddr
```

Final flashing address of app.

data

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint8_t* GblTagDelta_t::data
```

Array of data to flash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblSeUpgrade_t

GBL SE upgrade tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint32_t	blobSize Size of the SE upgrade blob.
uint32_t	version Version of the SE image.
uint8_t *	data Array of data for SE upgrade.

Public Attribute Documentation

header

```
GblTagHeader_t GblSeUpgrade_t::header
```

Tag ID and length.

blobSize

```
uint32_t GblSeUpgrade_t::blobSize
```

Size of the SE upgrade blob.

version

```
uint32_t GblSeUpgrade_t::version
```

Version of the SE image.

data

```
uint8_t* GblSeUpgrade_t::data
```

Array of data for SE upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblMetadata_t

GBL metadata tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint8_t *	metaData Array of metadata.

Public Attribute Documentation

header

```
GblTagHeader_t GblMetadata_t::header
```

Tag ID and length.

metaData

```
uint8_t* GblMetadata_t::metaData
```

Array of metadata.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblProg_t

GBL flash program tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint32_t	flashStartAddress Address to start flashing.
uint8_t *	data Array of data to flash.

Public Attribute Documentation

header

```
GblTagHeader_t GblProg_t::header
```

Tag ID and length.

flashStartAddress

```
uint32_t GblProg_t::flashStartAddress
```

Address to start flashing.

data

```
uint8_t* GblProg_t::data
```

Array of data to flash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblEnd_t

GBL end tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint32_t	gblCrc CRC32 of the entire GBL file.

Public Attribute Documentation

header

```
GblTagHeader_t GblEnd_t::header
```

Tag ID and length.

gblCrc

```
uint32_t GblEnd_t::gblCrc
```

CRC32 of the entire GBL file.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblEncryptionInitAesCcm_t

GBL encryption init tag type. Used with AES-CCM encryption.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint32_t	msgLen Length of the cipher text in bytes.
uint8_t	nonce Random nonce used for AES-CTR in this message.

Public Attribute Documentation

header

```
GblTagHeader_t GblEncryptionInitAesCcm_t::header
```

Tag ID and length.

msgLen

```
uint32_t GblEncryptionInitAesCcm_t::msgLen
```

Length of the cipher text in bytes.

nonce

```
uint8_t GblEncryptionInitAesCcm_t::nonce[12]
```

Random nonce used for AES-CTR in this message.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblEncryptionData_t

GBL encryption data tag type.

Public Attributes

GblTagHeader_t	header Tag ID and length.
uint8_t *	encryptedGblData Encrypted data.

Public Attribute Documentation

header

```
GblTagHeader_t GblEncryptionData_t::header
```

Tag ID and length.

encryptedGblData

```
uint8_t* GblEncryptionData_t::encryptedGblData
```

Encrypted data.

After decryption, this data must represent one or more complete unencrypted GBL tags.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblCertificateEcdsaP256_t

GBL certificate chain for signing.

Public Attributes

GblTagHeader_t	header Tag ID and length.
ApplicationCertificate_t	certificate Certificate used to verify GBL.

Public Attribute Documentation

header

```
GblTagHeader_t GblCertificateEcdsaP256_t::header
```

Tag ID and length.

certificate

```
ApplicationCertificate_t GblCertificateEcdsaP256_t::certificate
```

Certificate used to verify GBL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

GblSignatureEcdsaP256_t

GBL ECDSA secp256r1 signature tag type.

Public Attributes

GblTagHeader_t	header	Tag ID and length.
uint8_t	r	R-point of ECDSA secp256r1 signature.
uint8_t	s	S-point of ECDSA secp256r1 signature.

Public Attribute Documentation

header

```
GblTagHeader_t GblSignatureEcdsaP256_t::header
```

Tag ID and length.

r

```
uint8_t GblSignatureEcdsaP256_t::r[32]
```

R-point of ECDSA secp256r1 signature.

s

```
uint8_t GblSignatureEcdsaP256_t::s[32]
```

S-point of ECDSA secp256r1 signature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Security

Security

Collection of security components.

Security components provide implementations of cryptographic operations as well as functionality to compute checksums and to read cryptographic keys from manufacturing tokens.

Modules

AES

CRC16

CRC32

Decryption

ECDSA

SHA256

SHA_256

Tokens

Copyright © 2025 Silicon Laboratories. All rights reserved.

AES

AES

AES decryption functionality for bootloader.

Modules

AesContext_t

AesCtrContext_t

Functions

void btl_initAesContext(void *ctx)
Initialize AES context.

void btl_setAesKey(void *ctx, const uint8_t *key, unsigned int keySize, bool encryptNotDecrypt)
Set AES key to use for encryption/decryption.

void btl_processAesBlock(void *ctx, uint8_t *inputBlock, uint8_t *outputBlock, bool encryptNotDecrypt)
Process one block of data using AES-ECB.

void btl_initAesCcm(void *ctx, uint8_t flags, uint8_t *nonce, uint32_t counter, const uint8_t *key, unsigned int keySize)
Set up the AES-CTR context structure in CCM mode.

void btl_processAesCtrData(void *ctx, const uint8_t *input, uint8_t *output, size_t length)
Process data using AES-CTR.

Function Documentation

btl_initAesContext

```
void btl_initAesContext (void * ctx)
```

Initialize AES context.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Context variable of type AesContext_t

Wipes the AES context struct before use.

btl_setAesKey

```
void btl_setAesKey (void * ctx, const uint8_t * key, unsigned int keySize, bool encryptNotDecrypt)
```

Set AES key to use for encryption/decryption.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Context variable of type AesContext_t
const uint8_t *	N/A	key	Pointer to the AES key
unsigned int	N/A	keySize	Size of the key in bits. Can be 128 or 256.
bool	N/A	encryptNotDecrypt	True if using this context for encryption, false if using for decryption.

Initializes the AES context struct with the key to use.

btl_processAesBlock

```
void btl_processAesBlock (void * ctx, uint8_t * inputBlock, uint8_t * outputBlock, bool
encryptNotDecrypt)
```

Process one block of data using AES-ECB.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Context variable of type AesContext_t
uint8_t *	N/A	inputBlock	128-bit (16 byte) buffer/block of data to be en/decrypted
uint8_t *	N/A	outputBlock	128-bit (16 byte) buffer/block of data to put the result of the en/decryption in.
bool	N/A	encryptNotDecrypt	True for encryption, false for decryption

Runs one block of data through the AES algorithm. In-place encryption/ decryption is supported.

btl_initAesCcm

```
void btl_initAesCcm (void * ctx, uint8_t flags, uint8_t * nonce, uint32_t counter, const
uint8_t * key, unsigned int keySize)
```

Set up the AES-CTR context structure in CCM mode.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Context variable of type AesCtrContext_t
uint8_t	N/A	flags	CCM flags
uint8_t *	N/A	nonce	12-byte nonce specific to this transmission
uint32_t	N/A	counter	3-byte running block counter
const uint8_t *	N/A	key	Pointer to the AES key
unsigned int	N/A	keySize	Size of the key in bits. Can be 128 or 256.

Initializes an AES-CTR context struct with parameters used in AES-CCM mode. Data can then be en/decrypted using btl_processAesCtrData.

btl_processAesCtrData

```
void btl_processAesCtrData (void * ctx, const uint8_t * input, uint8_t * output, size_t length)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Process data using AES-CTR.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	`ctx`	Context variable of type AesCtrContext_t
const uint8_t *	N/A	`input`	Raw data to en/decrypt
uint8_t *	N/A	`output`	Output buffer to put en/decrypted data
size_t	N/A	`length`	Size (in bytes) of the input/output buffers

Runs data for encryption or decryption (which uses the same function) through the AES-CTR algorithm. In-place encryption/decryption is supported.

Copyright © 2025 Silicon Laboratories. All rights reserved.

AesContext_t

Context variable type for AES-ECB.

Public Attributes

mbedtls_aes_context	aesContext mbedTLS AES context

Public Attribute Documentation

aesContext

```
mbedtls_aes_context AesContext_t::aesContext
```

mbedTLS AES context

Copyright © 2025 Silicon Laboratories. All rights reserved.

AesCtrContext_t

Context variable type for AES-CTR (and AES-CCM)

Public Attributes

mbedtls_aes_context	aesContext mbedTLS AES context
size_t	offsetInBlock Position in block of last byte en/decrypted.
uint8_t	streamBlock Current CTR encrypted block.
uint8_t	counter Current counter/CCM value.

Public Attribute Documentation

aesContext

```
mbedtls_aes_context AesCtrContext_t::aesContext
```

mbedTLS AES context

offsetInBlock

```
size_t AesCtrContext_t::offsetInBlock
```

Position in block of last byte en/decrypted.

streamBlock

```
uint8_t AesCtrContext_t::streamBlock[16]
```

Current CTR encrypted block.

counter

```
uint8_t AesCtrContext_t::counter[16]
```

Current counter/CCM value.

Copyright © 2025 Silicon Laboratories. All rights reserved.

CRC16

CRC16

CRC16 functionality for the bootloader.

Functions

`uint16_t` `btl_crc16(const uint8_t newByte, uint16_t prevResult)`
Calculate CRC16 on input.

`uint16_t` `btl_crc16Stream(const uint8_t *buffer, size_t length, uint16_t prevResult)`
Calculate CRC16 on input stream.

Macros

`#define` `BTL_CRC16_START 0xFFFFU`
CRC16 start value.

Function Documentation

btl_crc16

```
uint16_t btl_crc16 (const uint8_t newByte, uint16_t prevResult)
```

Calculate CRC16 on input.

Parameters

Type	Direction	Argument Name	Description
const uint8_t	N/A	`newByte`	Byte to append to CRC16 calculation
uint16_t	N/A	`prevResult`	Previous output from CRC algorithm. BTL_CRC16_START when starting a new calculation.

Returns

- Result of the CRC16 operation

btl_crc16Stream

```
uint16_t btl_crc16Stream (const uint8_t * buffer, size_t length, uint16_t prevResult)
```

Calculate CRC16 on input stream.

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	N/A	`buffer`	Buffer containing bytes to append to CRC16 calculation

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
size_t	N/A	`length`	Size of the buffer in bytes
uint16_t	N/A	`prevResult`	Previous output from CRC algorithm. BTL_CRC16_START when starting a new calculation.

Returns

- Result of the CRC16 operation

Copyright © 2025 Silicon Laboratories. All rights reserved.

CRC32

CRC32

CRC32 functionality for the bootloader.

Functions

`uint32_t` `btl_crc32Stream(const uint8_t *buffer, size_t length, uint32_t prevResult)`
Calculate CRC32 on input buffer.

Macros

`#define` `BTL_CRC32_START 0xFFFFFFFFUL`
CRC32 start value.

`#define` `BTL_CRC32_END 0xDEBB20E3UL`
CRC32 end value.

Function Documentation

btl_crc32Stream

```
uint32_t btl_crc32Stream (const uint8_t * buffer, size_t length, uint32_t prevResult)
```

Calculate CRC32 on input buffer.

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	N/A	`buffer`	Buffer containing bytes to append to CRC32 calculation
size_t	N/A	`length`	Size of the buffer in bytes
uint32_t	N/A	`prevResult`	Previous output from the CRC algorithm. Polynomial if starting a new calculation

Returns

- Result of the CRC32 operation

Copyright © 2025 Silicon Laboratories. All rights reserved.

Decryption

Decryption

Generic decryption functionality for bootloader.

Modules

DecryptContext_t

AuthContext_t

Copyright © 2025 Silicon Laboratories. All rights reserved.

DecryptContext_t

Generic decryption context.

Public Attributes

AesCtrContext_t	aesCtr Context for AES-CTR-128 decryption.

Public Attribute Documentation

aesCtr

```
AesCtrContext_t DecryptContext_t::aesCtr
```

Context for AES-CTR-128 decryption.

Copyright © 2025 Silicon Laboratories. All rights reserved.

AuthContext_t

Generic authentication context.

Public Attributes

Sha256Context_t	sha256 Context for SHA-256 digest.

Public Attribute Documentation

sha256

```
Sha256Context_t AuthContext_t::sha256
```

Context for SHA-256 digest.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ECDSA

ECDSA

ECDSA signing functionality for the bootloader.

Modules

ECC Library

Functions

int32_t btl_verifyEcdsaP256r1(const uint8_t *sha256, const uint8_t *signatureR, const uint8_t *signatureS, const uint8_t *keyX, const uint8_t *keyY)

Verify an ECDSA signature of a SHA256-hash using secp256r1.

Macros

#define BTL_SECURITY_ECDSA_SHA256_LENGTH 32

Number of bytes of data to verify the signature against.

#define BTL_SECURITY_ECDSA_POINT_LENGTH 32

Number of bytes in the EC points that the signature consists of.

Function Documentation

btl_verifyEcdsaP256r1

```
int32_t btl_verifyEcdsaP256r1 (const uint8_t * sha256, const uint8_t * signatureR, const
uint8_t * signatureS, const uint8_t * keyX, const uint8_t * keyY)
```

Verify an ECDSA signature of a SHA256-hash using secp256r1.

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	N/A	sha256	The hash of the data which is authenticated
const uint8_t *	N/A	signatureR	Byte array (MSB first) of R-point of the ECDSA signature
const uint8_t *	N/A	signatureS	Byte array (MSB first) of S-point of the ECDSA signature
const uint8_t *	N/A	keyX	Pointer to the X coordinate of the ECDSA public key
const uint8_t *	N/A	keyY	Pointer to the Y coordinate of the ECDSA public key

Verifies the authenticity of data by checking the ECDSA signature of the data's SHA256-hash. This function is only for use with the secp256r1 curve. The public key which the signature is validated against will be retrieved from the respective tokens in the lockbits-page.

Returns

- BOOTLOADER_OK if signature is valid, else error code in BOOTLOADER_ERROR_SECURITY_BASE range.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ECC Library

ECC Library

Elliptic Curve Cryptography Library.

The Elliptic Curve Cryptography (ECC) API includes ECDSA verification on one of the elliptic curves recommended by NIST in csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf :

- secp256r1: NIST/SECG X9.62 curve over a 256 bit prime field

Modules

ECC_Point_t

ECC_EcdsaSignature_t

Typedefs

typedef uint32_t **ECC_BigInt_t**[ECC_BIGINT_SIZE_IN_32BIT_WORDS]
ECC big integer type.

Functions

int32_t **ECC_ECDSA_VerifySignatureP256**(CRYPTO_TypeDef *crypto, const uint8_t *msgDigest, int msgDigestLen, const ECC_Point_t *publicKey, ECC_EcdsaSignature_t *signature)
Functions **********.

void **ECC_HexToBigInt**(ECC_BigInt_t bigint, const char *hex)
Convert a large integer from a hexadecimal string to the ECC_BigInt_t format.

void **ECC_BigIntToHex**(char *hex, ECC_BigInt_t bigint)
Convert a large integer from the ECC_BigInt_t to a hexadecimal string format.

void **ECC_ByteArrayToBigInt**(ECC_BigInt_t bigint, const uint8_t *bytearray)
Convert a big integer from a byte array to the ECC_BigInt_t format.

void **ECC_BigIntToByteArray**(uint8_t *bytearray, ECC_BigInt_t bigint)
Convert a large integer from the ECC_BigInt_t to the byte array format.

void **ECC_UnsignedIntToBigInt**(ECC_BigInt_t bigint, const uint32_t value)
Convert an integer from uint32_t to ECC_BigInt_t format.

Macros

#define **ECC_BIGINT_SIZE_IN_BITS** (256)
TYPEDEFS **********.

#define **ECC_BIGINT_SIZE_IN_BYTES** (ECC_BIGINT_SIZE_IN_BITS / 8)
ECC big integer size in bytes.

#define **ECC_BIGINT_SIZE_IN_32BIT_WORDS** undefined
ECC big integer size in words.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Typedef Documentation

ECC_BigInt_t

```
typedef uint32_t ECC_BigInt_t[ECC_BIGINT_SIZE_IN_32BIT_WORDS] [ECC_BIGINT_SIZE_IN_32BIT_WORDS]
```

ECC big integer type.

Function Documentation

ECC_ECDSA_VerifySignatureP256

```
int32_t ECC_ECDSA_VerifySignatureP256 (CRYPTO_TypeDef * crypto, const uint8_t * msgDigest, int
msgDigestLen, const ECC_Point_t * publicKey, ECC_EcdsaSignature_t * signature)
```

Functions **********.

Parameters

Type	Direction	Argument Name	Description
CRYPTO_TypeDef *	[in]	`crypto`	Pointer to CRYPTO peripheral instance
const uint8_t *	[in]	`msgDigest`	The message digest associated with the signature.
int	[in]	`msgDigestLen`	The length of the message digest.
const ECC_Point_t *	[in]	`publicKey`	Public key of entity that generated signature.
ECC_EcdsaSignature_t *	[out]	`signature`	The signature to verify.

Verify an ECDSA signature.

TBW

Returns

- Error code.

ECC_HexToBigInt

```
void ECC_HexToBigInt (ECC_BigInt_t bigint, const char * hex)
```

Convert a large integer from a hexadecimal string to the ECC_BigInt_t format.

Parameters

Type	Direction	Argument Name	Description
ECC_BigInt_t	[out]	`bigint`	Pointer to the location where to store the result.
const char *	[in]	`hex`	The hex representation of the large integer to convert.

Convert a large integer from a hexadecimal string representation to a ECC_BigInt_t representation.

ECC_BigIntToHex

```
void ECC_BigIntToHex (char * hex, ECC_BigInt_t bigint)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Convert a large integer from the ECC_BigInt_t to a hexadecimal string format.

Parameters

Type	Direction	Argument Name	Description
char *	[out]	hex	Buffer where to store the hexadecimal result.
ECC_BigInt_t	[in]	bigint	The ECC_BigInt_t representation of the large integer to convert.

Convert a large integer from a ECC_BigInt_t representation to a hexadecimal string representation.

ECC_ByteArrayToBigInt

```
void ECC_ByteArrayToBigInt (ECC_BigInt_t bigint, const uint8_t * bytearray)
```

Convert a big integer from a byte array to the ECC_BigInt_t format.

Parameters

Type	Direction	Argument Name	Description
ECC_BigInt_t	[out]	bigint	Pointer to the location where to store the result.
const uint8_t *	[in]	bytearray	The byte array representation of the large integer to convert.

Convert a large integer from a byte array representation to a ECC_BigInt_t representation.

ECC_BigIntToByteArray

```
void ECC_BigIntToByteArray (uint8_t * bytearray, ECC_BigInt_t bigint)
```

Convert a large integer from the ECC_BigInt_t to the byte array format.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	bytearray	Buffer to store the resulting byte array.
ECC_BigInt_t	[in]	bigint	The ECC_BigInt_t representation of the large integer to convert.

Convert a large integer from a ECC_BigInt_t representation to a byte array representation. Caution: byte array must be big enough to contain the result.

ECC_UnsignedIntToBigInt

```
void ECC_UnsignedIntToBigInt (ECC_BigInt_t bigint, const uint32_t value)
```

Convert an integer from uint32_t to ECC_BigInt_t format.

Parameters

Type	Direction	Argument Name	Description
ECC_BigInt_t	[out]	bigint	Pointer to the location to store the result.
const uint32_t	[in]	value	The value to convert.

Convert a integer from an uint32_t representation to a ECC_BigInt_t representation.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ECC_Point_t

Elliptic curve point structure.

Public Attributes

ECC_BigInt_t	X x coordinate of point.
ECC_BigInt_t	Y y coordinate of point.

Public Attribute Documentation

X

```
ECC_BigInt_t ECC_Point_t::X
```

x coordinate of point.

Y

```
ECC_BigInt_t ECC_Point_t::Y
```

y coordinate of point.

Copyright © 2025 Silicon Laboratories. All rights reserved.

ECC_EcdsaSignature_t

ECDSA signature as defined in FIPS PUB 186-3, Digital Signature Standard (DSS).

Public Attributes

ECC_BigInt_t	r The r component of the signature.
ECC_BigInt_t	s The s component of the signature.

Public Attribute Documentation

r

```
ECC_BigInt_t ECC_EcdsaSignature_t::r
```

The r component of the signature.

s

```
ECC_BigInt_t ECC_EcdsaSignature_t::s
```

The s component of the signature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SHA256

SHA256

SHA-256 Cryptography Library.

This file includes an alternative implementation of the standard mbed TLS SHA using hardware accelerator incorporated in MCU devices from Silicon Labs.

Modules

btl_sha256_context

Enumerations

enum SHA_Type_t {
  SHA256
}
Type variable for SHA-256 Cryptography.

Functions

void btl_sha256_init(btl_sha256_context *ctx)
Initialize SHA-256 context.

int btl_sha256_starts_ret(btl_sha256_context *ctx, int is224)
Set up SHA-256 context.

int btl_sha256_update_ret(btl_sha256_context *ctx, const unsigned char *input, size_t ilen)
SHA-256 process buffer.

int btl_sha256_finish_ret(btl_sha256_context *ctx, unsigned char output[32])
SHA-256 final digest.

int sha_x_process(SHA_Type_t algo, uint8_t *state_in, const unsigned char *blockdata, uint8_t *state_out, uint32_t num_blocks)
Process (a) block(s) of data to be hashed.

int sha_x_update(SHA_Type_t algo, const unsigned char *data, size_t data_len, uint8_t *state, unsigned char *buffer, uint32_t *counter)
Process an arbitrary number of bytes to be hashed.

int sha_x_finish(SHA_Type_t algo, uint8_t *state, uint8_t *buffer, uint32_t *counter, uint8_t *output)
Process an arbitrary number of bytes to be hashed.

Enumeration Documentation

SHA_Type_t

```
SHA_Type_t
```

Type variable for SHA-256 Cryptography.

Enumerator

Copyright © 2025 Silicon Laboratories. All rights reserved.

SHA256	SHA-256 type.

Function Documentation

btl_sha256_init

```
void btl_sha256_init (btl_sha256_context * ctx)
```

Initialize SHA-256 context.

Parameters

Type	Direction	Argument Name	Description
btl_sha256_context *	N/A	ctx	SHA-256 context to be initialized

btl_sha256_starts_ret

```
int btl_sha256_starts_ret (btl_sha256_context * ctx, int is224)
```

Set up SHA-256 context.

Parameters

Type	Direction	Argument Name	Description
btl_sha256_context *	N/A	ctx	context to be initialized
int	N/A	is224	0 = use SHA256, 1 = use SHA224

Returns

- `0` if successful

btl_sha256_update_ret

```
int btl_sha256_update_ret (btl_sha256_context * ctx, const unsigned char * input, size_t ilen)
```

SHA-256 process buffer.

Parameters

Type	Direction	Argument Name	Description
btl_sha256_context *	N/A	ctx	SHA-256 context
const unsigned char *	N/A	input	buffer holding the data
size_t	N/A	ilen	length of the input data

Returns

- `0` if successful

btl_sha256_finish_ret

```
int btl_sha256_finish_ret (btl_sha256_context * ctx, unsigned char output)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

SHA-256 final digest.

Parameters

Type	Direction	Argument Name	Description
btl_sha256_context *	N/A	`ctx`	SHA-256 context
unsigned char	N/A	`output`	SHA-224/256 checksum result

Returns

- `0` if successful

sha_x_process

```
int sha_x_process (SHA_Type_t algo, uint8_t * state_in, const unsigned char * blockdata,
uint8_t * state_out, uint32_t num_blocks)
```

Process (a) block(s) of data to be hashed.

Parameters

Type	Direction	Argument Name	Description
SHA_Type_t	N/A	`algo`	Which hashing algorithm to use
uint8_t *	[in]	`state_in`	Previous state of the hashing algorithm
const unsigned char *	[in]	`blockdata`	Pointer to the block(s) of data
uint8_t *	[out]	`state_out`	Pointer to block of memory to store state
uint32_t	N/A	`num_blocks`	Number of SHA blocks in data block

Note

- Watch the blocksize! Blocks are 64 bytes for SHA-1 through SHA-256, and 128 bytes for SHA-384 through SHA-512.
- Watch the state size! State size is half the block size.

Returns

- Zero on success. Negative error code on failure.

sha_x_update

```
int sha_x_update (SHA_Type_t algo, const unsigned char * data, size_t data_len, uint8_t *
state, unsigned char * buffer, uint32_t * counter)
```

Process an arbitrary number of bytes to be hashed.

Parameters

Type	Direction	Argument Name	Description
SHA_Type_t	N/A	`algo`	Which hashing algorithm to use
const unsigned char *	[inout]	`data`	Pointer to the hashing algorithm's state buffer
size_t	[in]	`data_len`	Pointer to the data to add to the hash
uint8_t *	[inout]	`state`	Pointer to a block buffer to retrieve/store a partial block in between calls to this function. Needs to have a size equal to the block size.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
unsigned char *	[inout]	`buffer`	Counter variable keeping track of the amount of bytes hashed, to later be used for hash finalization. For first use, initialize with zeroes.
uint32_t *	[in]	`counter`	Length to data to add to hash

Note

- Watch the blocksize! Blocks are 64 bytes for SHA-1 through SHA-256, and 128 bytes for SHA-384 through SHA-512.
- Watch the state size! State size is half the block size.
- Watch the counter size! Counter is 64 bytes for SHA-1 through SHA-256, and 128 bytes for SHA-384 through SHA-512.

Returns

- Zero on success. Negative error code on failure.

sha_x_finish

```
int sha_x_finish (SHA_Type_t algo, uint8_t * state, uint8_t * buffer, uint32_t * counter,
uint8_t * output)
```

Process an arbitrary number of bytes to be hashed.

Parameters

Type	Direction	Argument Name	Description
SHA_Type_t	N/A	`algo`	Which hashing algorithm to use
uint8_t *	[in]	`state`	Pointer to the hashing algorithm's state buffer
uint8_t *	[in]	`buffer`	Pointer to a block buffer to retrieve/store a partial block in between calls to this function. Needs to have a size equal to the block size.
uint32_t *	[inout]	`counter`	Counter variable keeping track of the amount of bytes hashed, to later be used for hash finalization. For first use, initialize with zeroes.
uint8_t *	[out]	`output`	Pointer to the destination of the hash.

Note

- Watch the blocksize! Blocks are 64 bytes for SHA-1 through SHA-256, and 128 bytes for SHA-384 through SHA-512.
- Watch the state size! State size is half the block size.
- Watch the counter size! Counter is 64 bytes for SHA-1 through SHA-256, and 128 bytes for SHA-384 through SHA-512.

Returns

- Zero on success. Negative error code on failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

btl_sha256_context

Context Variable type for SHA-256 Cryptography.

Public Attributes

uint32_t	total number of bytes processed
uint32_t	state intermediate digest state
unsigned char	buffer data block being processed

Public Attribute Documentation

total

```
uint32_t btl_sha256_context::total[2]
```

number of bytes processed

state

```
uint32_t btl_sha256_context::state[8]
```

intermediate digest state

buffer

```
unsigned char btl_sha256_context::buffer[64]
```

data block being processed

Copyright © 2025 Silicon Laboratories. All rights reserved.

SHA_256

SHA_256

SHA-256 digest functionality for the bootloader.

Modules

Sha256Context_t

Functions

void btl_initSha256(void *ctx)
Initialize SHA256 context variable.

void btl_updateSha256(void *ctx, const void *data, size_t length)
Run data through the SHA256 hashing function.

void btl_finalizeSha256(void *ctx)
Finalize the SHA256 calculation.

int32_t btl_verifySha256(void *ctx, const void *sha256)
Compare the SHA256 from the context variable to a known value.

Macros

#define BTL_SECURITY_SHA256_DIGEST_LENGTH 32
Number of bytes in a SHA-256 digest.

Function Documentation

btl_initSha256

```
void btl_initSha256 (void * ctx)
```

Initialize SHA256 context variable.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Pointer to the SHA256 context variable to be initialized

Wipes out the SHA256 context variable and sets it up for re-use.

btl_updateSha256

```
void btl_updateSha256 (void * ctx, const void * data, size_t length)
```

Run data through the SHA256 hashing function.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void *	N/A	ctx	Pointer to the SHA256 context variable
const void *	N/A	data	Pointer to an array of binary data to add to the SHA256 calculation in progress
size_t	N/A	length	Length of the byte array with data.

btl_finalizeSha256

```
void btl_finalizeSha256 (void * ctx)
```

Finalize the SHA256 calculation.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Pointer to the SHA256 context variable to be initialized

Finalizes the running SHA256 calculation. After finalization, the SHA value in the context variable will be valid and no more data can be added.

btl_verifySha256

```
int32_t btl_verifySha256 (void * ctx, const void * sha256)
```

Compare the SHA256 from the context variable to a known value.

Parameters

Type	Direction	Argument Name	Description
void *	N/A	ctx	Pointer to the SHA256 context variable to be initialized
const void *	N/A	sha256	Byte array containing SHA256 value to compare to

Returns

- BOOTLOADER_OK if both hash values are equal, else error code from BOOTLOADER_ERROR_SECURITY_BASE range.

Convenience function to compare a known SHA256 hash against the result of a calculation.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Sha256Context_t

Context type for SHA algorithm.

Public Attributes

btl_sha256_context	shaContext mbedTLS SHA256 context struct
uint8_t	sha resulting SHA hash

Public Attribute Documentation

shaContext

```
btl_sha256_context Sha256Context_t::shaContext
```

mbedTLS SHA256 context struct

sha

```
uint8_t Sha256Context_t::sha[32]
```

resulting SHA hash

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tokens

Tokens

Manufacturing token handling for the bootloader.

Functions

`const uint8_t *` **btl_getSignedBootloaderKeyXPtr**`(void)`
Get the X component of the ECDSA secp256r1 public key.

`const uint8_t *` **btl_getSignedBootloaderKeyYPtr**`(void)`
Get the Y component of the ECDSA secp256r1 public key.

`const uint8_t *` **btl_getImageFileEncryptionKeyPtr**`(void)`
Get the AES-CCM encryption key.

Function Documentation

btl_getSignedBootloaderKeyXPtr

```
const uint8_t * btl_getSignedBootloaderKeyXPtr (void )
```

Get the X component of the ECDSA secp256r1 public key.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Pointer to X component of the public key

btl_getSignedBootloaderKeyYPtr

```
const uint8_t * btl_getSignedBootloaderKeyYPtr (void )
```

Get the Y component of the ECDSA secp256r1 public key.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Pointer to Y component of the public key

btl_getImageFileEncryptionKeyPtr

```
const uint8_t * btl_getImageFileEncryptionKeyPtr (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Get the AES-CCM encryption key.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Pointer to the AES-CCM key

Copyright © 2025 Silicon Laboratories. All rights reserved.

Storage

Storage

Storage component.

This component provides the bootloader with multiple storage options. All storage implementations have to provide a slot-based API to access image files to be bootloaded.

Some storage implementations also support a raw storage API to access the underlying storage medium. This can be used by applications to store other data in parts of the storage medium that are not used for bootloading.

Storage Implementations

Modules

BootloaderStorageLayout_t

Flash Using JEDEC SFDP Standard

SPI Flash

Internal Flash

Bootload Info

SPI Flash Configurations

SPI Flash Configurations using SFDP

Functions

```
int32_t    storage_init(void)
```
Initialize the storage component.

```
int32_t    storage_main(void)
```
Main function for storage component.

```
int32_t    storage_shutdown(void)
```
Shutdown storage component.

```
   void    storage_getInfo(BootloaderStorageInformation_t *info)
```
Get information about the storage component running on the device.

```
int32_t    storage_getSlotInfo(uint32_t slotId, BootloaderStorageSlot_t *slot)
```
Get information about the layout of a storage slot.

```
int32_t    storage_getSlotMetadata(BootloaderParserContext_t *context, ApplicationData_t
           *appInfo, uint32_t *bootloaderVersion)
```
Get information about the contents of a storage slot.

```
int32_t    storage_initParseSlot(uint32_t slotId, BootloaderParserContext_t *context, size_t
           contextSize)
```
Initialize the context variable for checking a slot and trying to parse the image contained in it.

```
int32_t    storage_verifySlot(BootloaderParserContext_t *context, BootloaderParserCallback_t
           metadataCallback)
```
Check the given slot for a valid image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool storage_upgradeSeFromSlot(uint32_t slotId)
Upgrade SE using image contained in a slot.

bool storage_bootloadBootloaderFromSlot(uint32_t slotId, uint32_t version)
Bootload a bootloader image contained in a slot.

bool storage_bootloadApplicationFromSlot(uint32_t slotId, uint32_t version, uint32_t deltaGBLLen)
Bootload an image contained in a slot.

int32_t storage_eraseSlot(uint32_t slotId)
Erase the contents of a storage slot including all data and metadata.

int32_t storage_readSlot(uint32_t slotId, uint32_t offset, uint8_t *buffer, size_t numBytes)
Read a number of words from a storage slot.

int32_t storage_writeSlot(uint32_t slotId, uint32_t offset, uint8_t *data, size_t numBytes)
Write a number of words to a storage slot.

int32_t storage_readRaw(uint32_t address, uint8_t *buffer, size_t numBytes)
Read number of words from raw storage.

int32_t storage_writeRaw(uint32_t address, uint8_t *data, size_t numBytes)
Write a number of words to raw storage.

int32_t storage_getDMAchannel(void)
Get allocated DMA channel for MSC write.

int32_t storage_eraseRaw(uint32_t address, size_t length)
Erase the raw storage.

bool storage_isBusy(void)
Poll the storage implementation and check whether it is busy.

uint32_t storage_getSpiUsartPPUSATD(uint32_t *ppusatdNr)
Get PPUSATD word of the (E)USART in use.

Macros

#define BOOTLOADER_STORAGE_FUNCTIONS_VERSION 0x00000100
Version number for bootloader storage functions struct.

Function Documentation

storage_init

```
int32_t storage_init (void )
```

Initialize the storage component.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

Return values

- BOOTLOADER_OK: on success
- BOOTLOADER_ERROR_INIT_STORAGE: on failure

Copyright © 2025 Silicon Laboratories. All rights reserved.

storage_main

```
int32_t storage_main (void )
```

Main function for storage component.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- BOOTLOADER_OK on success, else error code

storage_shutdown

```
int32_t storage_shutdown (void )
```

Shutdown storage component.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Error code indicating success or failure.

Return values

- BOOTLOADER_OK: on success
- BOOTLOADER_ERROR_INIT_STORAGE: on failure

storage_getInfo

```
void storage_getInfo (BootloaderStorageInformation_t * info)
```

Get information about the storage component running on the device.

Parameters

Type	Direction	Argument Name	Description
BootloaderStorageInformation_t *	[out]	info	Pointer to BootloaderStorageInformation_t struct

storage_getSlotInfo

```
int32_t storage_getSlotInfo (uint32_t slotId, BootloaderStorageSlot_t * slot)
```

Get information about the layout of a storage slot.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	Slot ID to get info about
BootloaderStorageSlot_t *	[out]	`slot`	Pointer to BootloaderStorageSlot_t struct

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_getSlotMetadata

```
int32_t storage_getSlotMetadata (BootloaderParserContext_t * context, ApplicationData_t *
appInfo, uint32_t * bootloaderVersion)
```

Get information about the contents of a storage slot.

Parameters

Type	Direction	Argument Name	Description
BootloaderParserContext_t *	[in]	`context`	Parsing context. Should be allocated by the application and initialized by calling storage_initParseSlot before calling this function.
ApplicationData_t *	[out]	`appInfo`	Pointer to ApplicationData_t struct
uint32_t *	[out]	`bootloaderVersion`	Pointer to an unsigned integer representing the bootloader version number.

Note

- storage_initParseSlotmust be called before calling this function to initialize the context.
- If the slot does not contain an application or a bootloader, the the corresponding values are set to zero.

Returns

- BOOTLOADER_OK on success

storage_initParseSlot

```
int32_t storage_initParseSlot (uint32_t slotId, BootloaderParserContext_t * context, size_t
contextSize)
```

Initialize the context variable for checking a slot and trying to parse the image contained in it.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`slotId`	Slot to check for valid image
BootloaderParserContext_t *	[in]	`context`	Pointer to BootloaderParserContext_t struct
size_t	[in]	`contextSize`	Length of the context struct

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_PARSE_BASE range

storage_verifySlot

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
int32_t storage_verifySlot (BootloaderParserContext_t * context, BootloaderParserCallback_t
metadataCallback)
```

Check the given slot for a valid image.

Parameters

Type	Direction	Argument Name	Description
BootloaderParserContext_t *	[in]	`context`	Pointer to BootloaderImageParsingContext_t struct
BootloaderParserCallback_t	[in]	`metadataCallback`	Functionpointer which will be called with any binary metadata that might be contained within the image.

Call this function continuously until it stops returning. BOOTLOADER_ERROR_PARSE_CONTINUE.

The function returns BOOTLOADER_ERROR_PARSE_SUCCESS if the image in the slot was successfully verified. For detailed information on the parsed image, see imageProperties in the context variable.

Returns

- BOOTLOADER_ERROR_PARSE_CONTINUE if the parsing is not complete, BOOTLOADER_ERROR_PARSE_SUCCESS on success.

storage_upgradeSeFromSlot

```
bool storage_upgradeSeFromSlot (uint32_t slotId)
```

Upgrade SE using image contained in a slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`slotId`	Slot ID to bootload from

Note

- This function assumes the image located in the slotId has been verified first.

Returns

- True if the operation succeeded

storage_bootloadBootloaderFromSlot

```
bool storage_bootloadBootloaderFromSlot (uint32_t slotId, uint32_t version)
```

Bootload a bootloader image contained in a slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`slotId`	Slot ID to bootload from
uint32_t	N/A	`version`	Version number of new bootloader

Note

- This function assumes the image located in the slotId has been verified first.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- True if operation succeeded

storage_bootloadApplicationFromSlot

```
bool storage_bootloadApplicationFromSlot (uint32_t slotId, uint32_t version, uint32_t deltaGBLLen)
```

Bootload an image contained in a slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`slotId`	Slot ID to bootload from
uint32_t	N/A	`version`	Cached version number of the image contained in the slot (used for downgrade prevention)
uint32_t	N/A	`deltaGBLLen`	Length of the Delta GBL file.

Note

- This function assumes the image located in slotId has been verified first.

Returns

- True if the operation succeeded

storage_eraseSlot

```
int32_t storage_eraseSlot (uint32_t slotId)
```

Erase the contents of a storage slot including all data and metadata.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`slotId`	ID of the slot.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_readSlot

```
int32_t storage_readSlot (uint32_t slotId, uint32_t offset, uint8_t * buffer, size_t numBytes)
```

Read a number of words from a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`slotId`	ID of the slot.
uint32_t	N/A	`offset`	The offset into the slot in bytes.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t *	N/A	buffer	Pointer to buffer to store read data in.
size_t	N/A	numBytes	Number of bytes to read.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_writeSlot

```
int32_t storage_writeSlot (uint32_t slotId, uint32_t offset, uint8_t * data, size_t numBytes)
```

Write a number of words to a storage slot.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	slotId	ID of the slot.
uint32_t	N/A	offset	The offset into the slot in bytes.
uint8_t *	N/A	data	Pointer to data to write.
size_t	N/A	numBytes	Length of data to write. Must be a multiple of 4.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_readRaw

```
int32_t storage_readRaw (uint32_t address, uint8_t * buffer, size_t numBytes)
```

Read number of words from raw storage.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	address	The raw address of the storage.
uint8_t *	N/A	buffer	Pointer to the buffer to store read data in.
size_t	N/A	numBytes	Number of bytes to read.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_writeRaw

```
int32_t storage_writeRaw (uint32_t address, uint8_t * data, size_t numBytes)
```

Write a number of words to raw storage.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint32_t	N/A	address	The raw address of the storage.
uint8_t *	N/A	data	Pointer to data to write.
size_t	N/A	numBytes	Length of data to write. Must be a multiple of 4.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_getDMAchannel

```
int32_t storage_getDMAchannel (void )
```

Get allocated DMA channel for MSC write.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A positive number channel. -1 if DMA-based MSC write is not enabled.

storage_eraseRaw

```
int32_t storage_eraseRaw (uint32_t address, size_t length)
```

Erase the raw storage.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	address	Start address of the region to erase
size_t	N/A	length	Number of bytes to erase

Note

- Some devices, such as Flash-based storages, have restrictions on the alignment and size of erased regions. The details of the limitations of a particular storage can be found by reading the BootloaderStorageInformation_t struct using storage_getInfo.

Returns

- BOOTLOADER_OK on success, else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_isBusy

```
bool storage_isBusy (void )
```

Poll the storage implementation and check whether it is busy.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the storage is busy

storage_getSpiUsartPPUSATD

```
uint32_t storage_getSpiUsartPPUSATD (uint32_t * ppusatdNr)
```

Get PPUSATD word of the (E)USART in use.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	N/A	ppusatdNr	PPUSATD register number

Returns

- Word representing the PPUSATD bit field of the (E)USART in use. 0 if not found.

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloaderStorageLayout_t

Information about the storage backend.

Public Attributes

BootloaderStorageType_t	storageType Type of storage.
uint32_t	numSlots Number of slots.
BootloaderStorageSlot_t	slot Array of slot information.

Public Attribute Documentation

storageType

```
BootloaderStorageType_t BootloaderStorageLayout_t::storageType
```

Type of storage.

numSlots

```
uint32_t BootloaderStorageLayout_t::numSlots
```

Number of slots.

slot

```
BootloaderStorageSlot_t BootloaderStorageLayout_t::slot[]
```

Array of slot information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Flash Using JEDEC SFDP Standard

Flash Using JEDEC SFDP Standard

The SPI Flash storage implementation using JEDEC supports all SPI Flash parts that support JEDEC SFDP.

The supported devices will be detected and configured at runtime using the SFDP standard. The driver queries the parameter table present within a reserved section of the flash memory to get the device properties. The driver initializes the flash device in accordance with the values present in the parameter table. Including support for SFDP standard in the bootloader will lead to slightly larger bootloader code sizes

The SPI Flash storage implementation does not support any write protection functionality.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI Flash

SPI Flash

The SPI Flash storage implementation supports a variety of SPI Flash parts including the following:

- Spansion S25FL208K (8Mbit)
- Winbond W25X20BV (2Mbit), W25Q80BV (8Mbit)
- Macronix MX25L2006E (2Mbit), MX25L4006E (4Mbit), MX25L8006E (8Mbit), MX25R8035F (8Mbit low power), MX25L1606E (16Mbit), MX25U1635E (16Mbit 2Volt), MX25R3235F (32Mbit ultra low power) MX25R6435F (64Mbit low power)
- Atmel/Adesto AT25DF041A (4Mbit), AT25DF081A (8Mbit)
- Numonyx/Micron M25P20 (2Mbit), M25P40 (4Mbit), M25P80 (8Mbit), M25P16 (16Mbit)
- ISSI IS25LQ025B (256Kbit), IS25LQ512B (512Kbit), IS25LQ010B (1Mbit), IS25LQ020B (2Mbit), IS25LQ040B (4Mbit)

The subset of supported devices can be configured at compile time using the configuration defines given in SPI Flash Configurations. Including support for multiple devices requires more Flash space in the bootloader.

The SPI Flash storage implementation does not support any write protection functionality.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Internal Flash

Internal Flash

The Internal Flash storage implementation utilizes the internal Flash of the device for upgrade image storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootload Info

Bootload Info

Indicates which firmware update image should be bootloaded next.

This component provides the bootloader with support for storing multiple images and attempting to bootload a prioritized list. The Bootload Information struct is placed at a known location in storage, and points to a list of images to attempt to bootload.

While the Bootload Information list of images to attempt to bootload has a compile-time configurable size, the bootloader is capable of handling lists of images with different sizes, e.g., if a bootloader upgrade changes the slot layout or if a storage device that was initialized on a different device is used.

If only one storage slot is available, the functions available in this API will do nothing and return applicable error codes (BOOTLOADER_OK for storage_getBootloadList and storage_setBootloadList, and BOOTLOADER_ERROR_BOOTLOAD_LIST_FULL for storage_appendBootloadList).

Modules

BootloadInfo_t

Functions

`int32_t` `storage_getBootloadList(int32_t slotIds[], size_t length)`
Get list of firmware update images to attempt to bootload.

`int32_t` `storage_setBootloadList(int32_t slotIds[], size_t length)`
Set list of firmware update images to attempt to bootload.

`int32_t` `storage_appendBootloadList(int32_t slotId)`
Append a storage slot to the list of storage slots to try bootloading from.

Macros

`#define` `BTL_STORAGE_BOOTLOADINFO_MAGIC 0xE18F5239UL`
Magic word indicating BootloadInfo_t struct.

`#define` `BTL_STORAGE_BOOTLOADINFO_VERSION 0x00000001UL`
Version number for the BootloadInfo_t struct.

`#define` `BTL_STORAGE_BOOTLOAD_LIST_MAX_LENGTH 16UL`
Maximum number of items in the BootloadInfo_t::bootloadList.

Function Documentation

storage_getBootloadList

```
int32_t storage_getBootloadList (int32_t slotIds, size_t length)
```

Get list of firmware update images to attempt to bootload.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
int32_t	[out]	`slotIds`	Pointer to array of integers to fill with slot IDs, or -1 if list position does not contain a valid slot ID
size_t	[in]	`length`	Number of slot IDs to get

Returns

- Error code: BOOTLOADER_OK if list of slots was successfully filled else error code in BOOTLOADER_ERROR_STORAGE_BASE range

storage_setBootloadList

```
int32_t storage_setBootloadList (int32_t slotIds, size_t length)
```

Set list of firmware update images to attempt to bootload.

Parameters

Type	Direction	Argument Name	Description
int32_t	[in]	`slotIds`	Pointer to array of slot IDs to set
size_t	[in]	`length`	Number of slot IDs to set

Returns

- Error code: BOOTLOADER_OK if list of slots was successfully filled

storage_appendBootloadList

```
int32_t storage_appendBootloadList (int32_t slotId)
```

Append a storage slot to the list of storage slots to try bootloading from.

Parameters

Type	Direction	Argument Name	Description
int32_t	[in]	`slotId`	ID of the slot

Returns

- Error code: BOOTLOADER_OK if slot was successfully appended

Copyright © 2025 Silicon Laboratories. All rights reserved.

BootloadInfo_t

Definition of the Bootload Info struct.

The Bootload Info struct contains a prioritized list of firmware update images that the bootloader should attempt to bootload. The first image to pass verification will be bootloaded.

Public Attributes

uint32_t	magic Magic word indicating that this is a Bootload Info struct (BTL_STORAGE_BOOTLOADINFO_MAGIC)
uint32_t	structVersion Struct version number.
uint32_t	length Size of the BootloadInfo_t struct.
int32_t	bootloadList List of addresses of slots to bootload from.
uint32_t	crc32 Checksum of the struct.

Public Attribute Documentation

magic

```
uint32_t BootloadInfo_t::magic
```

Magic word indicating that this is a Bootload Info struct (BTL_STORAGE_BOOTLOADINFO_MAGIC)

structVersion

```
uint32_t BootloadInfo_t::structVersion
```

Struct version number.

length

```
uint32_t BootloadInfo_t::length
```

Size of the BootloadInfo_t struct.

bootloadList

```
int32_t BootloadInfo_t::bootloadList[BTL_STORAGE_BOOTLOAD_LIST_MAX_LENGTH]
```

List of addresses of slots to bootload from.

Copyright © 2025 Silicon Laboratories. All rights reserved.

crc32

```
uint32_t BootloadInfo_t::crc32
```

Checksum of the struct.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI Flash Configurations

SPI Flash Configurations

Configuration parameters for SPI flashes.

Variables

const BootloaderStorageImplementationInformation_t **spansion8MInfo**
Information for Spansion S25L208K.

const BootloaderStorageImplementationInformation_t **windbond2MInfo**
Information for Winbond W25X20BV.

const BootloaderStorageImplementationInformation_t **windbond8MInfo**
Information for Winbond W25Q80BV.

const BootloaderStorageImplementationInformation_t **macronix2MInfo**
Information for Macronix MX25L2006E.

const BootloaderStorageImplementationInformation_t **macronix4MInfo**
Information for Macronix MX25L4006E.

const BootloaderStorageImplementationInformation_t **macronix8MInfo**
Information for Macronix MX25L8006E.

const BootloaderStorageImplementationInformation_t **macronix8MLPInfo**
Information for Macronix MX25R8035F.

const BootloaderStorageImplementationInformation_t **macronix16MInfo**
Information for Macronix MX25L1606E.

const BootloaderStorageImplementationInformation_t **macronix16M2VInfo**
Information for Macronix MX25U1635E.

const BootloaderStorageImplementation-

Copyright © 2025 Silicon Laboratories. All rights reserved.

ionInformation_t	macronix32MLPInfo Information for Macronix MX25R3235F.
const BootloaderStorageImplementationInformation_t	macronix64MLPInfo Information for Macronix MX25R6435F.
const BootloaderStorageImplementationInformation_t	atmel4MInfo Information for Atmel AT25DF041A.
const BootloaderStorageImplementationInformation_t	atmel8MInfo Information for Atmel AT25DF081A.
const BootloaderStorageImplementationInformation_t	adesto4MInfo Information for Adesto AT25SF041.
const BootloaderStorageImplementationInformation_t	numonyx2MInfo Information for Numonyx M25P20.
const BootloaderStorageImplementationInformation_t	numonyx4MInfo Information for Numonyx M25P40.
const BootloaderStorageImplementationInformation_t	numonyx8MInfo Information for Numonyx M25P80.
const BootloaderStorageImplementationInformation_t	numonyx16MInfo Information for Numonyx M25P16.
const BootloaderStorageImplementationInformation_t	issi256KInfo Information for ISSI IS25LQ025B.
const BootloaderStorageImplementationInformation_t	issi512KInfo Information for ISSI IS25LQ512B.
const BootloaderStorageImplementationInformation_t	issi1MInfo Information for ISSI IS25LQ010B.

Copyright © 2025 Silicon Laboratories. All rights reserved.

const BootloaderStorageImplementationInformation_t	issi2MInfo Information for ISSI IS25LQ020B.
const BootloaderStorageImplementationInformation_t	issi4MInfo Information for ISSI IS25LQ040B.

Macros

#define BTL_STORAGE_SPIFLASH_ALL_DEVICES
Support all devices.

#define BTL_STORAGE_SPIFLASH_SPANSION_DEVICES
Support all Spansion devices.

#define BTL_STORAGE_SPIFLASH_SPANSION_S25FL208K 1
Support Spansion S25FL208K.

#define BTL_STORAGE_SPIFLASH_WINBOND_DEVICES
Support all Winbond devices.

#define BTL_STORAGE_SPIFLASH_WINBOND_W25X20BV 1
Support Winbond W25X20BV.

#define BTL_STORAGE_SPIFLASH_WINBOND_W25Q80BV 1
Support Winbond W25Q80BV.

#define BTL_STORAGE_SPIFLASH_MACRONIX_DEVICES
Support all Macronix devices.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25L2006E 1
Support Macronix MX25L2006E.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25L4006E 1
Support Macronix MX25L4006E.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25L8006E 1
Support Macronix MX25L8006E.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25R8035F 1
Support Macronix MX25R8035F.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25L1606E 1
Support Macronix MX25L1606E.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25U1635E 1
Support Macronix MX25U1635E.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25R3235F 1
Support Macronix MX25R3235F.

#define BTL_STORAGE_SPIFLASH_MACRONIX_MX25R6435F 1
Support Macronix MX25R6435F.

#define BTL_STORAGE_SPIFLASH_ATMEL_DEVICES
Support all Atmel devices.

#define BTL_STORAGE_SPIFLASH_ATMEL_AT25DF041A 1
Support Atmel AT25DF041A.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define BTL_STORAGE_SPIFLASH_ATMEL_AT25DF081A 1
Support Atmel AT25DF081A.

#define BTL_STORAGE_SPIFLASH_ADESTO_AT25SF041 1
Support Adesto AT25SF041.

#define BTL_STORAGE_SPIFLASH_NUMONYX_DEVICES
Support all Numonyx devices.

#define BTL_STORAGE_SPIFLASH_NUMONYX_M25P20 1
Support Numonyx M26P20.

#define BTL_STORAGE_SPIFLASH_NUMONYX_M25P40 1
Support Numonyx M26P40.

#define BTL_STORAGE_SPIFLASH_NUMONYX_M25P80 1
Support Numonyx M26P80.

#define BTL_STORAGE_SPIFLASH_NUMONYX_M25P16 1
Support Numonyx M26P16.

#define BTL_STORAGE_SPIFLASH_ISSI_DEVICES
Support all ISSI devices.

#define BTL_STORAGE_SPIFLASH_ISSI_IS25LQ025B 1
Support ISSI IS25LQ025B.

#define BTL_STORAGE_SPIFLASH_ISSI_IS25LQ512B 1
Support ISSI IS25LQ512B.

#define BTL_STORAGE_SPIFLASH_ISSI_IS25LQ010B 1
Support ISSI IS25LQ010B.

#define BTL_STORAGE_SPIFLASH_ISSI_IS25LQ020B 1
Support ISSI IS25LQ020B.

#define BTL_STORAGE_SPIFLASH_ISSI_IS25LQ040B 1
Support ISSI IS25LQ040B.

Variable Documentation

spansion8MInfo

```
const BootloaderStorageImplementationInformation_t spansion8MInfo
```

Information for Spansion S25L208K.

windbond2MInfo

```
const BootloaderStorageImplementationInformation_t windbond2MInfo
```

Information for Winbond W25X20BV.

windbond8MInfo

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
const BootloaderStorageImplementationInformation_t windbond8MInfo
```

Information for Winbond W25Q80BV.

macronix2MInfo

```
const BootloaderStorageImplementationInformation_t macronix2MInfo
```

Information for Macronix MX25L2006E.

macronix4MInfo

```
const BootloaderStorageImplementationInformation_t macronix4MInfo
```

Information for Macronix MX25L4006E.

macronix8MInfo

```
const BootloaderStorageImplementationInformation_t macronix8MInfo
```

Information for Macronix MX25L8006E.

macronix8MLPInfo

```
const BootloaderStorageImplementationInformation_t macronix8MLPInfo
```

Information for Macronix MX25R8035F.

macronix16MInfo

```
const BootloaderStorageImplementationInformation_t macronix16MInfo
```

Information for Macronix MX25L1606E.

macronix16M2VInfo

```
const BootloaderStorageImplementationInformation_t macronix16M2VInfo
```

Information for Macronix MX25U1635E.

macronix32MLPInfo

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
const BootloaderStorageImplementationInformation_t macronix32MLPInfo
```

Information for Macronix MX25R3235F.

macronix64MLPInfo

```
const BootloaderStorageImplementationInformation_t macronix64MLPInfo
```

Information for Macronix MX25R6435F.

atmel4MInfo

```
const BootloaderStorageImplementationInformation_t atmel4MInfo
```

Information for Atmel AT25DF041A.

atmel8MInfo

```
const BootloaderStorageImplementationInformation_t atmel8MInfo
```

Information for Atmel AT25DF081A.

adesto4MInfo

```
const BootloaderStorageImplementationInformation_t adesto4MInfo
```

Information for Adesto AT25SF041.

numonyx2MInfo

```
const BootloaderStorageImplementationInformation_t numonyx2MInfo
```

Information for Numonyx M25P20.

numonyx4MInfo

```
const BootloaderStorageImplementationInformation_t numonyx4MInfo
```

Information for Numonyx M25P40.

numonyx8MInfo

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
const BootloaderStorageImplementationInformation_t numonyx8MInfo
```

Information for Numonyx M25P80.

numonyx16MInfo

```
const BootloaderStorageImplementationInformation_t numonyx16MInfo
```

Information for Numonyx M25P16.

issi256KInfo

```
const BootloaderStorageImplementationInformation_t issi256KInfo
```

Information for ISSI IS25LQ025B.

issi512KInfo

```
const BootloaderStorageImplementationInformation_t issi512KInfo
```

Information for ISSI IS25LQ512B.

issi1MInfo

```
const BootloaderStorageImplementationInformation_t issi1MInfo
```

Information for ISSI IS25LQ010B.

issi2MInfo

```
const BootloaderStorageImplementationInformation_t issi2MInfo
```

Information for ISSI IS25LQ020B.

issi4MInfo

```
const BootloaderStorageImplementationInformation_t issi4MInfo
```

Information for ISSI IS25LQ040B.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI Flash Configurations using SFDP

SPI Flash Configurations using SFDP

Configuration parameters for SPI flashes Using JEDEC SFDP Standard.

Macros

`#define` `BTL_STORAGE_SPIFLASH_ATMEL_DEVICES`

Support all Atmel devices.

`#define` `BTL_STORAGE_SPIFLASH_ATMEL_AT25DF041A`

Support Atmel AT25DF041A.

`#define` `BTL_STORAGE_SPIFLASH_ATMEL_AT25DF081A`

Support Atmel AT25DF081A.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Driver

Driver

Hardware drivers for bootloader.

Modules

Delay

SPI

SPI Peripheral

UART

Copyright © 2025 Silicon Laboratories. All rights reserved.

Delay

Delay

Basic delay functionality.

Simple delay routines for use with components that require small delays.

Functions

`void` `delay_microseconds(uint32_t usecs)`
Delay for a number of microseconds.

`void` `delay_init(void)`
Initialize the delay driver's millisecond counter.

`void` `delay_milliseconds(uint32_t msecs, bool blocking)`
Delay for a number of milliseconds.

`bool` `delay_expired(void)`
Check whether the current delay has expired.

Function Documentation

delay_microseconds

```
void delay_microseconds (uint32_t usecs)
```

Delay for a number of microseconds.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`usecs`	Number of microseconds to delay

Note

- This function can be used without calling delay_init() first. This is not an accurate microsecond delay and can have a delay greater than expected, of about +50%. This error becomes significant for lower values delay.

delay_init

```
void delay_init (void )
```

Initialize the delay driver's millisecond counter.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

delay_milliseconds

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void delay_milliseconds (uint32_t msecs, bool blocking)
```

Delay for a number of milliseconds.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	`msecs`	Number of milliseconds to delay. The number should stay within a single TIMER0 overflow (approx. 3300 ms).
bool	N/A	`blocking`	Whether to block until the delay has expired. If false, the delay_expired() function can be called to check whether the delay has expired.

delay_expired

```
bool delay_expired (void )
```

Check whether the current delay has expired.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the delay has expired.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI

SPI

Basic Serial Peripheral Interface Driver.

Simple, blocking SPI controller implementation for communication with external devices.

Functions

void	spi_init(void) Initialize a USART peripheral for SPI.
void	spi_deinit(void) De-initialize a USART peripheral for SPI.
void	spi_writeByte(uint8_t data) Write a single byte discarding the received byte.
void	spi_writeHalfword(uint16_t data) Write two bytes discarding the received bytes.
void	spi_write3Byte(uint32_t data) Write three bytes discarding the received bytes.
uint8_t	spi_readByte(void) Read a byte by sending a 0xFF byte.
uint16_t	spi_readHalfword(void) Read two bytes by sending two 0xFF bytes.
void	spi_setCsActive(void) Assert the peripheral select line.
void	spi_setCsInactive(void) Dessert the peripheral select line.
uint32_t	spi_getUsartPPUSATD(uint32_t *ppusatdNr) Get PPUSATD word of the (E)USART in use.

Function Documentation

spi_init

```
void spi_init (void )
```

Initialize a USART peripheral for SPI.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

spi_deinit

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void spi_deinit (void )
```

De-initialize a USART peripheral for SPI.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

spi_writeByte

```
void spi_writeByte (uint8_t data)
```

Write a single byte discarding the received byte.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	`data`	The byte to send

spi_writeHalfword

```
void spi_writeHalfword (uint16_t data)
```

Write two bytes discarding the received bytes.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	`data`	The bytes to send, most significant byte first

spi_write3Byte

```
void spi_write3Byte (uint32_t data)
```

Write three bytes discarding the received bytes.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	`data`	The bytes to send, most significant byte first

spi_readByte

```
uint8_t spi_readByte (void )
```

Read a byte by sending a 0xFF byte.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The received byte

spi_readHalfword

```
uint16_t spi_readHalfword (void )
```

Read two bytes by sending two 0xFF bytes.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The received bytes, most significant byte first

spi_setCsActive

```
void spi_setCsActive (void )
```

Assert the peripheral select line.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Polarity is configured by spi_init.

spi_setCsInactive

```
void spi_setCsInactive (void )
```

Dessert the peripheral select line.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Polarity is configured by spi_init.

spi_getUsartPPUSATD

```
uint32_t spi_getUsartPPUSATD (uint32_t * ppusatdNr)
```

Get PPUSATD word of the (E)USART in use.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[out]	`ppusatdNr`	PPUSATD register number

Returns

- The PPUSATD word of the (E)USART in use

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI Peripheral

SPI Peripheral

SPI Peripheral Interface driver.

Flexible SPI Peripheral driver implementation for use in an NCP scenario.

This driver will support both blocking and non-blocking operation, with LDMA backing the background transfers to support nonblocking.

Functions

void	spi_peripheral_init(void) Initialize the configured USART peripheral for the SPI peripheral operation.
void	spi_peripheral_deinit(void) Disable the configured USART peripheral for the SPI operation.
int32_t	spi_peripheral_sendBuffer(uint8_t *buffer, size_t length) Write a data buffer to the controller next time the controller starts clocking SCLK.
int32_t	spi_peripheral_sendByte(uint8_t byte) Write one byte to the controller in a blocking fashion.
size_t	spi_peripheral_getTxBytesLeft(void) Get the amount of bytes left in the TX data buffer.
void	spi_peripheral_enableTransmitter(bool enable) Enable/disable MISO output.
void	spi_peripheral_enableReceiver(bool enable) Enable/disable receiving bytes from the controller into the internal buffer.
size_t	spi_peripheral_getRxAvailableBytes(void) Get the amount of bytes ready for reading.
int32_t	spi_peripheral_receiveBuffer(uint8_t *buffer, size_t requestedLength, size_t *receivedLength, bool blocking, uint32_t timeout) Read from the RX buffer into a local buffer.
int32_t	spi_peripheral_receiveByte(uint8_t *byte) Get one byte from the SPI peripheral in a blocking fashion.
void	spi_peripheral_flush(bool flushTx, bool flushRx) Flush one or both buffers.

Function Documentation

spi_peripheral_init

```
void spi_peripheral_init (void )
```

Initialize the configured USART peripheral for the SPI peripheral operation.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Also sets up GPIO settings for MOSI, MISO, SCLK and SS. After initialization, the SPI peripheral will have RX enabled.

spi_peripheral_deinit

```
void spi_peripheral_deinit (void )
```

Disable the configured USART peripheral for the SPI operation.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

spi_peripheral_sendBuffer

```
int32_t spi_peripheral_sendBuffer (uint8_t * buffer, size_t length)
```

Write a data buffer to the controller next time the controller starts clocking SCLK.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[in]	`buffer`	The data buffer to send
size_t	[in]	`length`	Number of bytes in the buffer to send

This transfer will be non-blocking and its progress can be tracked through spi_peripheral_getTxBytesLeft.

Returns

- BOOTLOADER_OK if successful, error code otherwise

spi_peripheral_sendByte

```
int32_t spi_peripheral_sendByte (uint8_t byte)
```

Write one byte to the controller in a blocking fashion.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	`byte`	The byte to send

Warnings

- If the controller goes down, this will block forever.

Returns

- BOOTLOADER_OK if successful, error code otherwise

Copyright © 2025 Silicon Laboratories. All rights reserved.

spi_peripheral_getTxBytesLeft

```
size_t spi_peripheral_getTxBytesLeft (void )
```

Get the amount of bytes left in the TX data buffer.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Number of bytes in the transmit buffer still needing to go out

spi_peripheral_enableTransmitter

```
void spi_peripheral_enableTransmitter (bool enable)
```

Enable/disable MISO output.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	`enable`	True to enable the transmitter, false to disable

spi_peripheral_enableReceiver

```
void spi_peripheral_enableReceiver (bool enable)
```

Enable/disable receiving bytes from the controller into the internal buffer.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	`enable`	True to enable the receiver, false to disable

This function makes it possible to avoid filling up the buffer with 0xFF while a controller is polling for new data.

spi_peripheral_getRxAvailableBytes

```
size_t spi_peripheral_getRxAvailableBytes (void )
```

Get the amount of bytes ready for reading.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Number of bytes in the receive buffer available for reading with spi_peripheral_receiveBuffer

Copyright © 2025 Silicon Laboratories. All rights reserved.

spi_peripheral_receiveBuffer

```
int32_t spi_peripheral_receiveBuffer (uint8_t * buffer, size_t requestedLength, size_t *
receivedLength, bool blocking, uint32_t timeout)
```

Read from the RX buffer into a local buffer.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	buffer	The data buffer to receive into
size_t	[in]	requestedLength	Number of bytes we'd like to read
size_t *	[out]	receivedLength	Number of bytes read
bool	[in]	blocking	Indicate whether we should wait for requestedLength bytes to be available and read before returning, or we can read out whatever is currently in the buffer and return.
uint32_t	[in]	timeout	Number of milliseconds to wait for data in blocking mode

Returns

- BOOTLOADER_OK if successful, error code otherwise

spi_peripheral_receiveByte

```
int32_t spi_peripheral_receiveByte (uint8_t * byte)
```

Get one byte from the SPI peripheral in a blocking fashion.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	byte	Pointer to where to put the received byte

Warnings

- If the controller never clocks in a byte, this function will block forever.

Returns

- BOOTLOADER_OK if successful, error code otherwise

spi_peripheral_flush

```
void spi_peripheral_flush (bool flushTx, bool flushRx)
```

Flush one or both buffers.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	flushTx	Flush the transmit buffer when true
bool	[in]	flushRx	Flush the receive buffer when true

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

UART

UART

Serial UART Interface Driver.

Flexible UART driver implementation for communication with external devices.

Functions

void	`uart_init(void)` Initialize the configured USART peripheral for UART operation.
void	`uart_deinit(void)` Disable the configured USART peripheral for UART operation.
int32_t	`uart_sendBuffer(const uint8_t *buffer, size_t length, bool blocking)` Write a data buffer to the UART.
int32_t	`uart_sendByte(uint8_t byte)` Write one byte to the UART in a blocking fashion.
bool	`uart_isTxIdle(void)` Find out whether the UART can accept more data to send.
size_t	`uart_getRxAvailableBytes(void)` Get the number of bytes ready for reading.
int32_t	`uart_receiveBuffer(uint8_t *buffer, size_t requestedLength, size_t *receivedLength, bool blocking, uint32_t timeout)` Read from the UART into a data buffer.
int32_t	`uart_receiveByte(uint8_t *byte)` Get one byte from UART in a blocking fashion.
int32_t	`uart_receiveByteTimeout(uint8_t *byte, uint32_t timeout)` Get one byte from UART in a blocking fashion.
int32_t	`uart_flush(bool flushTx, bool flushRx)` Flush one or both UART buffers.

Function Documentation

uart_init

```
void uart_init (void )
```

Initialize the configured USART peripheral for UART operation.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Also sets up GPIO settings for TX, RX, and, if configured, flow control.

Copyright © 2025 Silicon Laboratories. All rights reserved.

uart_deinit

```
void uart_deinit (void )
```

Disable the configured USART peripheral for UART operation.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

uart_sendBuffer

```
int32_t uart_sendBuffer (const uint8_t * buffer, size_t length, bool blocking)
```

Write a data buffer to the UART.

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	[in]	`buffer`	The data buffer to send
size_t	[in]	`length`	Number of bytes in the buffer to send
bool	[in]	`blocking`	Indicates whether this transfer can be offloaded to LDMA and return, or whether to wait on completion before returning.

Returns

- BOOTLOADER_OK if successful, error code otherwise

uart_sendByte

```
int32_t uart_sendByte (uint8_t byte)
```

Write one byte to the UART in a blocking fashion.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	`byte`	The byte to send

Returns

- BOOTLOADER_OK if successful, error code otherwise

uart_isTxIdle

```
bool uart_isTxIdle (void )
```

Find out whether the UART can accept more data to send.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
void	N/A		

Returns

- true if the UART is not currently transmitting

uart_getRxAvailableBytes

```
size_t uart_getRxAvailableBytes (void )
```

Get the number of bytes ready for reading.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Number of bytes in the receive buffer available for reading with uart_receiveBuffer

uart_receiveBuffer

```
int32_t uart_receiveBuffer (uint8_t * buffer, size_t requestedLength, size_t * receivedLength,
bool blocking, uint32_t timeout)
```

Read from the UART into a data buffer.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	`buffer`	The data buffer to receive into
size_t	[in]	`requestedLength`	Number of bytes to read
size_t *	[out]	`receivedLength`	Number of bytes read
bool	[in]	`blocking`	Indicates whether to wait for requestedLength bytes to be available and read before returning, or whether to read out data currently in the buffer and return.
uint32_t	[in]	`timeout`	Number of milliseconds to wait for data in blocking mode

Returns

- BOOTLOADER_OK if successful, error code otherwise

uart_receiveByte

```
int32_t uart_receiveByte (uint8_t * byte)
```

Get one byte from UART in a blocking fashion.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t *	[out]	byte	The byte to send

Returns

- BOOTLOADER_OK if successful, error code otherwise

uart_receiveByteTimeout

```
int32_t uart_receiveByteTimeout (uint8_t * byte, uint32_t timeout)
```

Get one byte from UART in a blocking fashion.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	byte	The byte to send
uint32_t	[in]	timeout	Maximum timeout before aborting transfer

Returns

- BOOTLOADER_OK if successful, error code otherwise

uart_flush

```
int32_t uart_flush (bool flushTx, bool flushRx)
```

Flush one or both UART buffers.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	flushTx	Flush the transmit buffer when true
bool	[in]	flushRx	Flush the receive buffer when true

Returns

- BOOTLOADER_OK

Copyright © 2025 Silicon Laboratories. All rights reserved.

Error Codes

Error Codes

Bootloader error codes.

Modules

Bootloading Error Codes

Communication Component Error Codes

Compression Error Codes

Error Code Base Values

Image Parser Error Codes

Initialization Error Codes

Parse Error Codes

RTOS Error Codes

SPI Peripheral Driver Error Codes

Security Error Codes

Storage Driver Error Codes

UART Driver Error Codes

XMODEM Error Codes

Macros

```
#define    BOOTLOADER_OK 0L
```

No error, operation OK.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloading Error Codes

Bootloading Error Codes

Bootloader error codes returned by the bootloading process.

Offset from BOOTLOADER_ERROR_BOOTLOAD_BASE

Macros

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_EMPTY (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x01L)
```
No images marked for bootload.

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_FULL (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x02L)
```
List of images marked for bootload is full.

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_ENTRY_EXISTS (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x03L)
```
Image already marked for bootload.

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_OVERFLOW (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x04L)
```
Bootload list overflowed, requested length too large.

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_NO_LIST (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x05L)
```
No bootload list found at the base of storage.

```
#define BOOTLOADER_ERROR_BOOTLOAD_LIST_INVALID (BOOTLOADER_ERROR_BOOTLOAD_BASE | 0x06L)
```
Bootload list found but with invalid CRC.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Communication Component Error Codes

Communication Component Error Codes

Bootloader error codes returned by communication components.

Offset from BOOTLOADER_ERROR_COMMUNICATION_BASE

Macros

```
#define BOOTLOADER_ERROR_COMMUNICATION_INIT (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x01L)
```
Invalid input parameter to security algorithm Could not initialize hardware resources for communication protocol.

```
#define BOOTLOADER_ERROR_COMMUNICATION_START (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x02L)
```
Could not start communication with host (timeout, sync error, version mismatch, ...)

```
#define BOOTLOADER_ERROR_COMMUNICATION_DONE (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x03L)
```
Host closed communication, no image received.

```
#define BOOTLOADER_ERROR_COMMUNICATION_ERROR (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x04L)
```
Unrecoverable error in host-bootloader communication.

```
#define BOOTLOADER_ERROR_COMMUNICATION_IMAGE_ERROR (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x05L)
```
Host closed communication, no valid image received.

```
#define BOOTLOADER_ERROR_COMMUNICATION_TIMEOUT (BOOTLOADER_ERROR_COMMUNICATION_BASE | 0x06L)
```
Communication aborted, no response from host.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Compression Error Codes

Compression Error Codes

Bootloader error codes returned by the decompressor.

Offset from BOOTLOADER_ERROR_COMPRESSION_BASE

Macros

```
#define BOOTLOADER_ERROR_COMPRESSION_INIT (BOOTLOADER_ERROR_COMPRESSION_BASE | 0x01)
```
Could not initialize decompressor.

```
#define BOOTLOADER_ERROR_COMPRESSION_STATE (BOOTLOADER_ERROR_COMPRESSION_BASE | 0x02)
```
Invalid decompressor state – possible invalid input.

```
#define BOOTLOADER_ERROR_COMPRESSION_DATA (BOOTLOADER_ERROR_COMPRESSION_BASE | 0x03)
```
Data error.

```
#define BOOTLOADER_ERROR_COMPRESSION_DATALEN (BOOTLOADER_ERROR_COMPRESSION_BASE | 0x04)
```
Data length error.

```
#define BOOTLOADER_ERROR_COMPRESSION_MEM (BOOTLOADER_ERROR_COMPRESSION_BASE | 0x05)
```
Memory error.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Error Code Base Values

Error Code Base Values

Bootloader error code base values, per logical function.

Macros

`#define` `BOOTLOADER_ERROR_INIT_BASE 0x0100L`
Initialization errors.

`#define` `BOOTLOADER_ERROR_PARSE_BASE 0x0200L`
Image verification errors.

`#define` `BOOTLOADER_ERROR_STORAGE_BASE 0x0400L`
Storage errors.

`#define` `BOOTLOADER_ERROR_BOOTLOAD_BASE 0x0500L`
Bootload errors.

`#define` `BOOTLOADER_ERROR_SECURITY_BASE 0x0600L`
Security errors.

`#define` `BOOTLOADER_ERROR_COMMUNICATION_BASE 0x0700L`
Communication component errors.

`#define` `BOOTLOADER_ERROR_XMODEM_BASE 0x0900L`
XMODEM parser errors.

`#define` `BOOTLOADER_ERROR_PARSER_BASE 0x1000L`
Image file parser errors.

`#define` `BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE 0x1100L`
SPI Peripheral driver errors.

`#define` `BOOTLOADER_ERROR_UART_BASE 0x1200L`
UART driver errors.

`#define` `BOOTLOADER_ERROR_COMPRESSION_BASE 0x1300L`
Compression errors.

`#define` `BOOTLOADER_ERROR_RTOS_BASE 0x1400L`
RTOS erros.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Image Parser Error Codes

Image Parser Error Codes

Bootloader error codes returned by the image file parser.

Offset from BOOTLOADER_ERROR_PARSER_BASE

Macros

#define BOOTLOADER_ERROR_PARSER_UNEXPECTED (BOOTLOADER_ERROR_PARSER_BASE | 0x01L)
Encountered unexpected data/option.

#define BOOTLOADER_ERROR_PARSER_BUFFER (BOOTLOADER_ERROR_PARSER_BASE | 0x02L)
Ran out of internal buffer space.

#define BOOTLOADER_ERROR_PARSER_PARSED (BOOTLOADER_ERROR_PARSER_BASE | 0x03L)
Internal state: done parsing the current input buffer.

#define BOOTLOADER_ERROR_PARSER_KEYERROR (BOOTLOADER_ERROR_PARSER_BASE | 0x04L)
Invalid encryption key or no key not present.

#define BOOTLOADER_ERROR_PARSER_CRC (BOOTLOADER_ERROR_PARSER_BASE | 0x05L)
Invalid checksum.

#define BOOTLOADER_ERROR_PARSER_SIGNATURE (BOOTLOADER_ERROR_PARSER_BASE | 0x06L)
Invalid signature.

#define BOOTLOADER_ERROR_PARSER_EOF (BOOTLOADER_ERROR_PARSER_BASE | 0x07L)
Image parsing is already done (or has previously errored out)

#define BOOTLOADER_ERROR_PARSER_UNKNOWN_TAG (BOOTLOADER_ERROR_PARSER_BASE | 0x08L)
Unknown data type in image file.

#define BOOTLOADER_ERROR_PARSER_VERSION (BOOTLOADER_ERROR_PARSER_BASE | 0x09L)
Image file version doesn't match with parser.

#define BOOTLOADER_ERROR_PARSER_FILETYPE (BOOTLOADER_ERROR_PARSER_BASE | 0x0AL)
Image file type doesn't match with parser.

#define BOOTLOADER_ERROR_PARSER_INIT (BOOTLOADER_ERROR_PARSER_BASE | 0x0BL)
Initialization failed.

#define BOOTLOADER_ERROR_PARSER_REJECTED (BOOTLOADER_ERROR_PARSER_BASE | 0x0CL)
Upgrade file was rejected.

#define BOOTLOADER_ERROR_PARSER_OVERLAP (BOOTLOADER_ERROR_PARSER_BASE | 0x0DL)
Upgrade file overlaps with the upgrade location.

#define BOOTLOADER_ERROR_PARSER_INVALID_TAG_ORDER (BOOTLOADER_ERROR_PARSER_BASE | 0x0EL)
A GBL tag occurred in an order forbidden by the GBL format spec.

#define BOOTLOADER_ERROR_PARSER_OOB_WRITE (BOOTLOADER_ERROR_PARSER_BASE | 0x0FL)
OOB write in the storage slot while parsing the GBL file.

#define BOOTLOADER_ERROR_PARSER_HASH (BOOTLOADER_ERROR_PARSER_BASE | 0x10L)
Invalid hash.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initialization Error Codes

Bootloader error codes returned by initialization code.

Offset from BOOTLOADER_ERROR_INIT_BASE

Macros

```
#define BOOTLOADER_ERROR_INIT_STORAGE (BOOTLOADER_ERROR_INIT_BASE | 0x01L)
```
Storage initialization error.

```
#define BOOTLOADER_ERROR_INIT_TABLE (BOOTLOADER_ERROR_INIT_BASE | 0x02L)
```
Bootloader table invalid.

```
#define BOOTLOADER_ERROR_INIT_SFDP (BOOTLOADER_ERROR_INIT_BASE | 0x03L)
```
Bootloader SFDP not supported.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parse Error Codes

Parse Error Codes

Bootloader error codes returned by image parsing.

Offset from BOOTLOADER_ERROR_PARSE_BASE

Macros

```
#define BOOTLOADER_ERROR_PARSE_CONTINUE (BOOTLOADER_ERROR_PARSE_BASE | 0x01L)
```
Parse not complete, continue calling the parsing function.

```
#define BOOTLOADER_ERROR_PARSE_FAILED (BOOTLOADER_ERROR_PARSE_BASE | 0x02L)
```
Verification failed.

```
#define BOOTLOADER_ERROR_PARSE_SUCCESS (BOOTLOADER_ERROR_PARSE_BASE | 0x03L)
```
Verification successfully completed. Image is valid.

```
#define BOOTLOADER_ERROR_PARSE_STORAGE (BOOTLOADER_ERROR_PARSE_BASE | 0x04L)
```
Bootloader has no storage, and cannot parse images.

```
#define BOOTLOADER_ERROR_PARSE_CONTEXT (BOOTLOADER_ERROR_PARSE_BASE | 0x05L)
```
Parse context incompatible with parse function.

Copyright © 2025 Silicon Laboratories. All rights reserved.

RTOS Error Codes

Bootloader error codes returned by the application interface.

Offset from BOOTLOADER_ERROR_RTOS_BASE

Macros

```
#define BOOTLOADER_ERROR_RTOS_MUTEX_SUCCESS (BOOTLOADER_ERROR_RTOS_BASE | 0x01)
```

Mutex operations executed successfully.

```
#define BOOTLOADER_ERROR_RTOS_MUTEX_FAILED (BOOTLOADER_ERROR_RTOS_BASE | 0x02)
```

Mutex operations failed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

SPI Peripheral Driver Error Codes

SPI Peripheral Driver Error Codes

Bootloader error codes returned by the SPI Peripheral driver.

Offset from BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE

Macros

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_UNINIT (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x01)
```
Operation not allowed because hardware has not been initialized.

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_INIT (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x02)
```
Hardware fail during initialization.

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_ARGUMENT (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x03)
```
Invalid argument.

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_TIMEOUT (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x04)
```
Timeout.

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_OVERFLOW (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x05)
```
Buffer overflow condition.

```
#define BOOTLOADER_ERROR_SPI_PERIPHERAL_BUSY (BOOTLOADER_ERROR_SPI_PERIPHERAL_BASE | 0x06)
```
Busy condition.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Security Error Codes

Security Error Codes

Bootloader error codes returned by security algorithms.

Offset from BOOTLOADER_ERROR_SECURITY_BASE

Macros

```
#define BOOTLOADER_ERROR_SECURITY_INVALID_PARAM (BOOTLOADER_ERROR_SECURITY_BASE | 0x01L)
```
Invalid input parameter to security algorithm.

```
#define BOOTLOADER_ERROR_SECURITY_PARAM_OUT_RANGE (BOOTLOADER_ERROR_SECURITY_BASE | 0x02L)
```
Input parameter to security algorithm is out of range.

```
#define BOOTLOADER_ERROR_SECURITY_INVALID_OPTION (BOOTLOADER_ERROR_SECURITY_BASE | 0x03L)
```
Invalid option for security algorithm.

```
#define BOOTLOADER_ERROR_SECURITY_REJECTED (BOOTLOADER_ERROR_SECURITY_BASE | 0x04L)
```
Authentication did not check out.

```
#define BOOTLOADER_ERROR_SECURITY_INIT_FAILED (BOOTLOADER_ERROR_SECURITY_BASE | 0x05L)
```
Security Init failed.

```
#define BOOTLOADER_ERROR_SECURITY_DE_INIT_FAILED (BOOTLOADER_ERROR_SECURITY_BASE | 0x06L)
```
Security De Init failed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Storage Driver Error Codes

Storage Driver Error Codes

Bootloader error codes returned by a storage driver.

Offset from BOOTLOADER_ERROR_STORAGE_BASE

Macros

```
#define BOOTLOADER_ERROR_STORAGE_INVALID_SLOT (BOOTLOADER_ERROR_STORAGE_BASE | 0x01L)
```
Invalid slot.

```
#define BOOTLOADER_ERROR_STORAGE_INVALID_ADDRESS (BOOTLOADER_ERROR_STORAGE_BASE | 0x02L)
```
Invalid address. Address not aligned/out of range.

```
#define BOOTLOADER_ERROR_STORAGE_NEEDS_ERASE (BOOTLOADER_ERROR_STORAGE_BASE | 0x03L)
```
The storage area needs to be erased before it can be used.

```
#define BOOTLOADER_ERROR_STORAGE_NEEDS_ALIGN (BOOTLOADER_ERROR_STORAGE_BASE | 0x04L)
```
The address or length needs to be aligned.

```
#define BOOTLOADER_ERROR_STORAGE_BOOTLOAD (BOOTLOADER_ERROR_STORAGE_BASE | 0x05L)
```
An error occured during bootload from storage.

```
#define BOOTLOADER_ERROR_STORAGE_NO_IMAGE (BOOTLOADER_ERROR_STORAGE_BASE | 0x06L)
```
There is no image in this storage slot.

```
#define BOOTLOADER_ERROR_STORAGE_CONTINUE (BOOTLOADER_ERROR_STORAGE_BASE | 0x07L)
```
Continue calling function.

```
#define BOOTLOADER_ERROR_STORAGE_GENERIC (BOOTLOADER_ERROR_STORAGE_BASE | 0x08L)
```
Generic storage error.

Copyright © 2025 Silicon Laboratories. All rights reserved.

UART Driver Error Codes

UART Driver Error Codes

Bootloader error codes returned by the UART driver.

Offset from BOOTLOADER_ERROR_UART_BASE

Macros

```
#define BOOTLOADER_ERROR_UART_UNINIT (BOOTLOADER_ERROR_UART_BASE | 0x01)
```
Operation not allowed because hardware has not been initialized.

```
#define BOOTLOADER_ERROR_UART_INIT (BOOTLOADER_ERROR_UART_BASE | 0x02)
```
Hardware fail during initialization.

```
#define BOOTLOADER_ERROR_UART_ARGUMENT (BOOTLOADER_ERROR_UART_BASE | 0x03)
```
Invalid argument.

```
#define BOOTLOADER_ERROR_UART_TIMEOUT (BOOTLOADER_ERROR_UART_BASE | 0x04)
```
Operation timed out.

```
#define BOOTLOADER_ERROR_UART_OVERFLOW (BOOTLOADER_ERROR_UART_BASE | 0x05)
```
Buffer overflow condition.

```
#define BOOTLOADER_ERROR_UART_BUSY (BOOTLOADER_ERROR_UART_BASE | 0x06)
```
Busy condition.

Copyright © 2025 Silicon Laboratories. All rights reserved.

XMODEM Error Codes

XMODEM Error Codes

Bootloader error codes returned by the XMODEM parser.

Offset from BOOTLOADER_ERROR_XMODEM_BASE

Macros

```
#define BOOTLOADER_ERROR_XMODEM_CRCL (BOOTLOADER_ERROR_XMODEM_BASE | 0x01L)
```
Could not verify lower CRC byte.

```
#define BOOTLOADER_ERROR_XMODEM_CRCH (BOOTLOADER_ERROR_XMODEM_BASE | 0x02L)
```
Could not verify upper CRC byte.

```
#define BOOTLOADER_ERROR_XMODEM_NO_SOH (BOOTLOADER_ERROR_XMODEM_BASE | 0x03L)
```
No start of header found.

```
#define BOOTLOADER_ERROR_XMODEM_PKTNUM (BOOTLOADER_ERROR_XMODEM_BASE | 0x04L)
```
Packet number doesn't match its inverse.

```
#define BOOTLOADER_ERROR_XMODEM_PKTSEQ (BOOTLOADER_ERROR_XMODEM_BASE | 0x05L)
```
Packet number error (unexpected sequence)

```
#define BOOTLOADER_ERROR_XMODEM_PKTDUP (BOOTLOADER_ERROR_XMODEM_BASE | 0x06L)
```
Packet number error (duplicate)

```
#define BOOTLOADER_ERROR_XMODEM_DONE (BOOTLOADER_ERROR_XMODEM_BASE | 0x07L)
```
Transfer is done (Technically not an error)

```
#define BOOTLOADER_ERROR_XMODEM_CANCEL (BOOTLOADER_ERROR_XMODEM_BASE | 0x08L)
```
Transfer is canceled.

Copyright © 2025 Silicon Laboratories. All rights reserved.