UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher

This version of UG489 has been deprecated with the release of Simplicity SDK Suite 2025.6.1 For the latest version, see docs.silabs.com.

****

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:

- *AN1084: Using the Gecko Bootloader with EmberZNet*
- *UG435.06: Bootloading and OTA with Silicon Labs Connect v3.x*
- *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*

For more information on Series 2 device security, see:

- *AN1218: Series 2 Secure Boot with RTSL*
- *AN1190: EFR32xG21 Secure Debug*
- *AN1222: Production Programming of Series 2 Devices*

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

- *AN1247: Anti-Tamper Protection Configuration and Use*
- *AN1268: Authenticating Silicon Labs Devices using Device Certificates*
- *AN1271: Secure Key Storage*

KEY POINTS

- Describes the Gecko Bootloader components.
- Summarizes how the Gecko Bootloader performs application upgrades and bootloader upgrades.
- Reviews how to create customized bootloaders in Simplicity Studio.
- Discusses the key configuration changes for various bootloader types.
- Describes Gecko Bootloader security features and discusses how to use them.

Table of Contents

1 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 2 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 5 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.

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 7 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 1.1. 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 8 Simplicity Commander and the Gecko Bootloader.

1.1 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.

1.1.1 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.

1.2 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.

1.3 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

1.3.1 Communication

The Communication components provide an interface for implementing communication with a host device, such as a computer or a microcontroller. 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 *AN1086: 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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.3.5 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

1.3.6 Storage

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 1.2. 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
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.

1.4 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

2 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 *UG162: Simplicity Commander Reference Guide*.

2.1 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)

2.2 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.

Tag Name	ID	Description
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.
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.

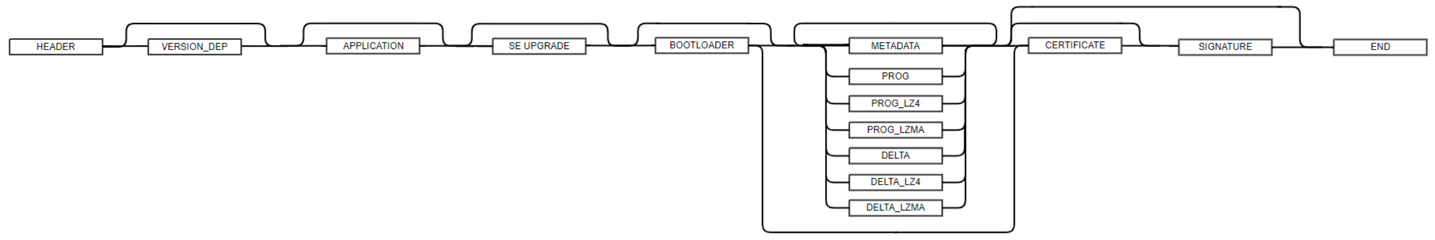


Figure 2.1. GBL Tag Sequence

2.3 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.

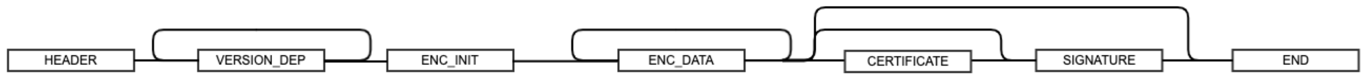


Figure 2.2. Encrypted GBL Tag Sequence

2.4 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

```
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

```
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;



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

```
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

```
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;
```

3 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 4 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*.

3.1 Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:

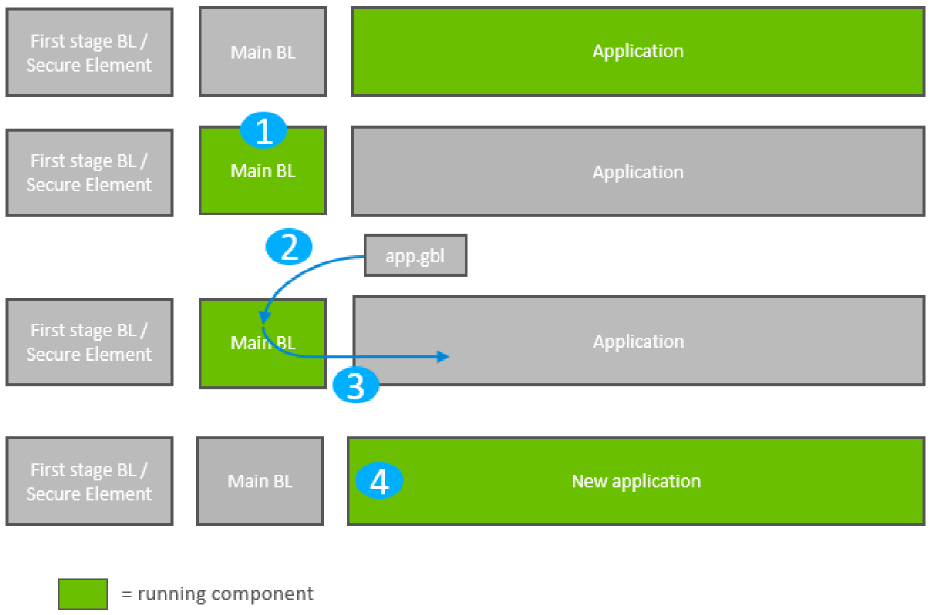


Figure 3.1. Standalone Bootloader Operation

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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.2 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.

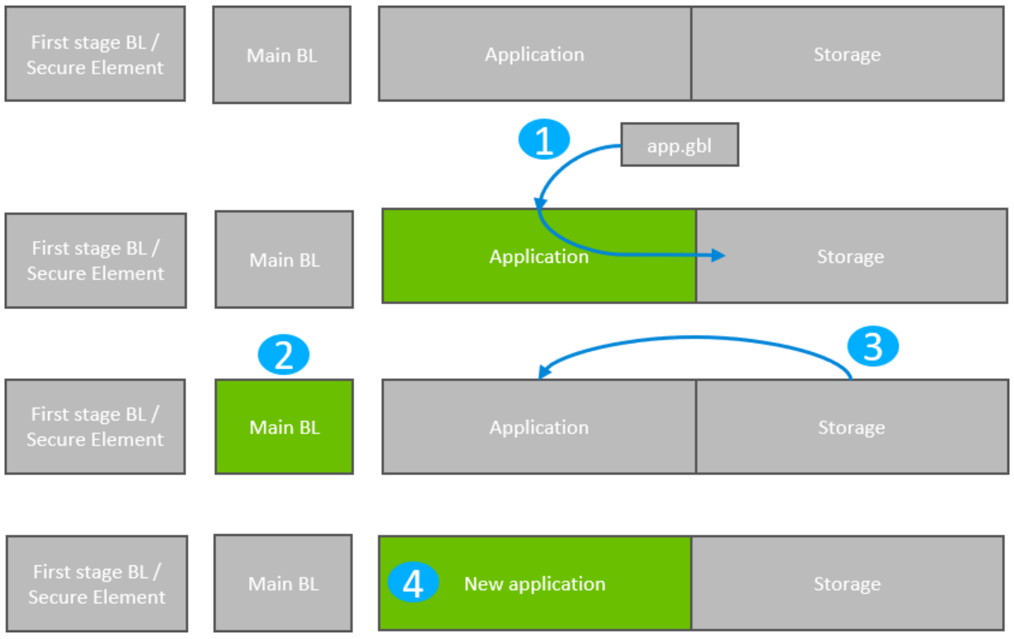

First stage BL / Secure Element
Main BL
Application
Storage
1
app.gbl
First stage BL / Secure Element
Main BL
Application
Storage
2
3
First stage BL / Secure Element
Main BL
Application
Storage
First stage BL / Secure Element
Main BL
4
New application
Storage

Figure 3.2. Application Bootloader Operation: Single Storage Slot

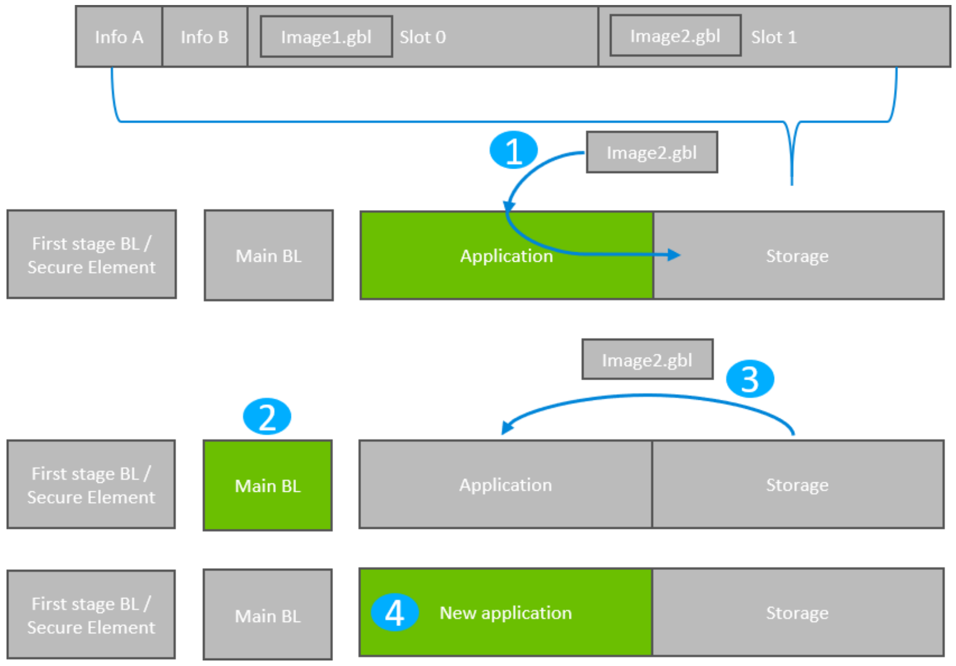


Figure 3.3. Application Bootloader Operation: Multiple Storage Slots

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.

3.2.1 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 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 3.3. 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.

3.2.2 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.

3.2.3 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.

4 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 5 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 9.3.3 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.

4.1 Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:

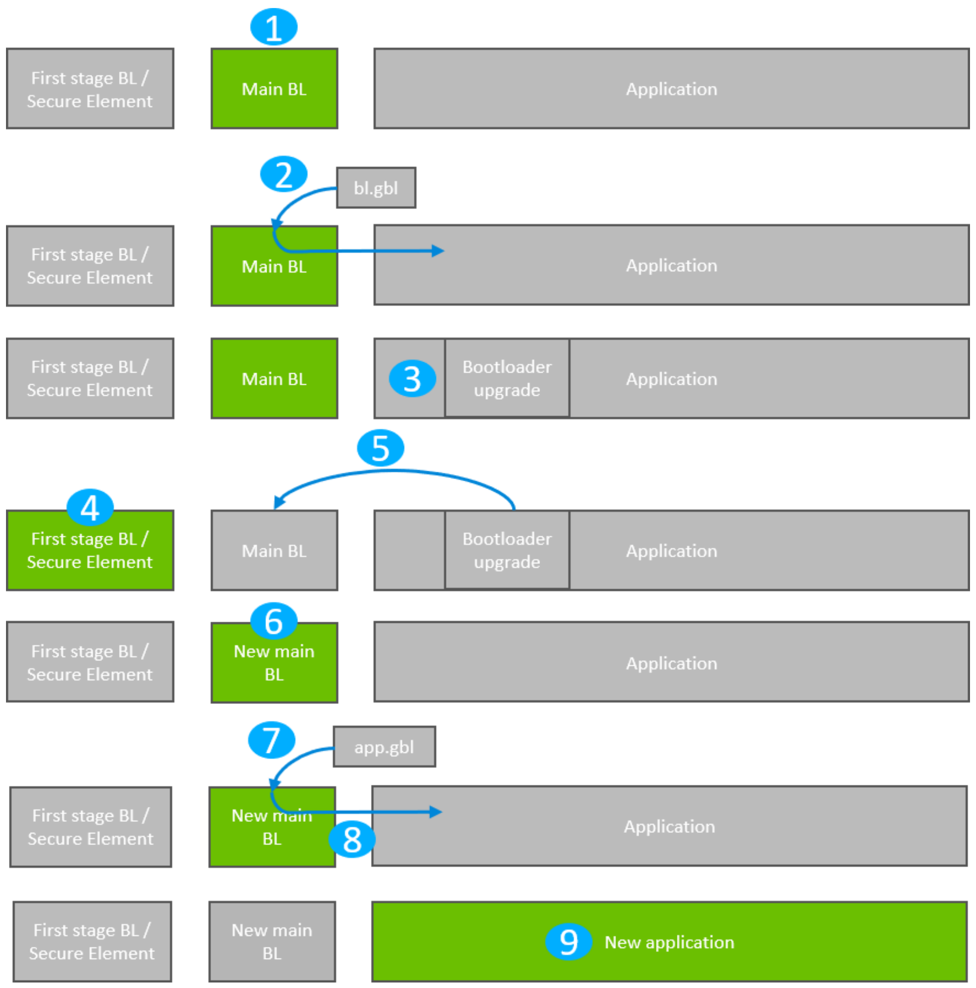


Figure 4.1. Standalone Bootloader: Bootloader Upgrade

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.

4.1.1 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 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 7.5 Setting a Version Number for more information about versioning bootloader images.

4.1.2 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 9.3.1 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 9.3.3 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 4.1 Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders).

4.1.3 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.
2. Generate a public/private Signing Key pair. See section 9.3.1 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 9.3.3 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 9.3.2 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 *AN1218: Series 2 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.

4.1.4 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 3.1 Standalone Bootloader Operation.

4.2 Bootloader Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.

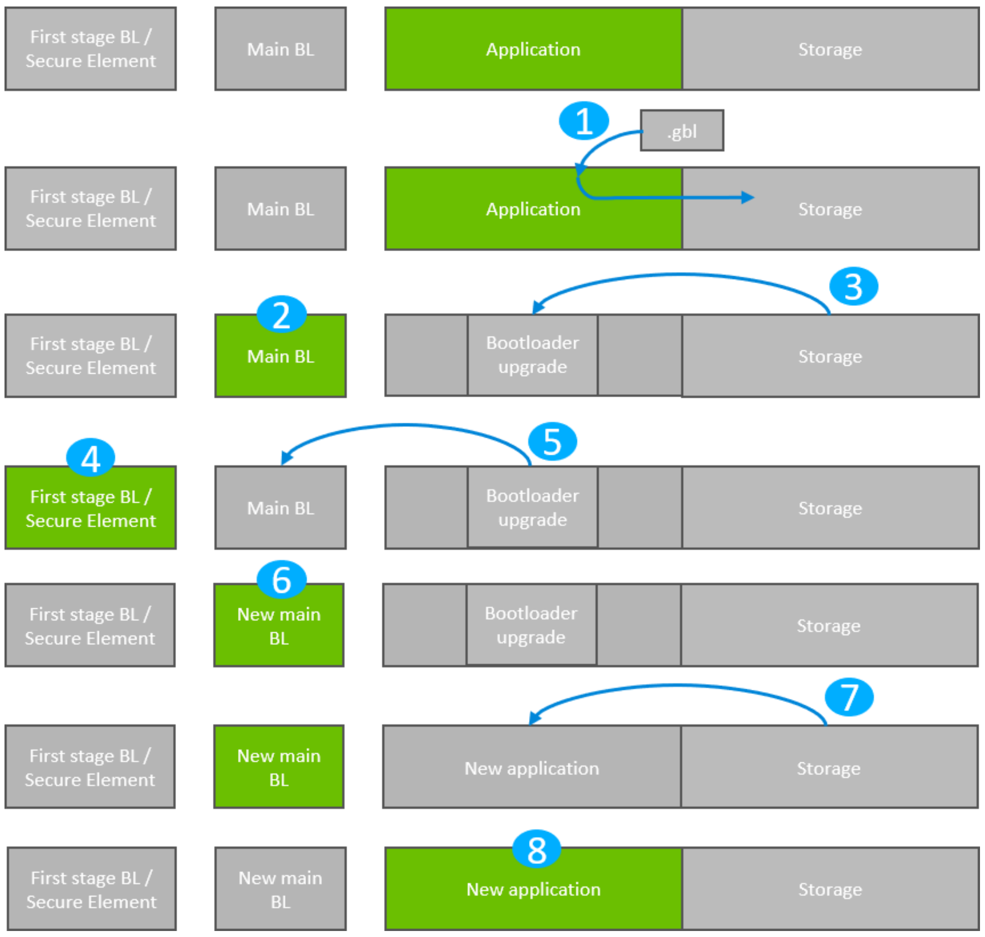


Figure 4.2. Application Bootloader: Bootloader Upgrade

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 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 7.5 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 3.2 Application Bootloader Operation.

4.2.1 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 7.8 Size Requirements for Different Bootloader Configurations for Series 1 Devices.

4.2.2 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 9.3.1 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 9.3.2 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 9.3.3 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 4.2 Bootloader Upgrade on Application Bootloaders with Storage.

4.2.3 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.
    - (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 9.3.1 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 unsigned GBL file from this application and upload it. For more information on installing public keys, see *AN1218: Series 2 Secure Boot*

with RTSL. For more details on creating a GBL file, see section 9.3.3 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 9.3.2 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 *AN1218: Series 2 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 9.3.3 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.

5 Gecko Bootloader Operation - Secure Engine Upgrade

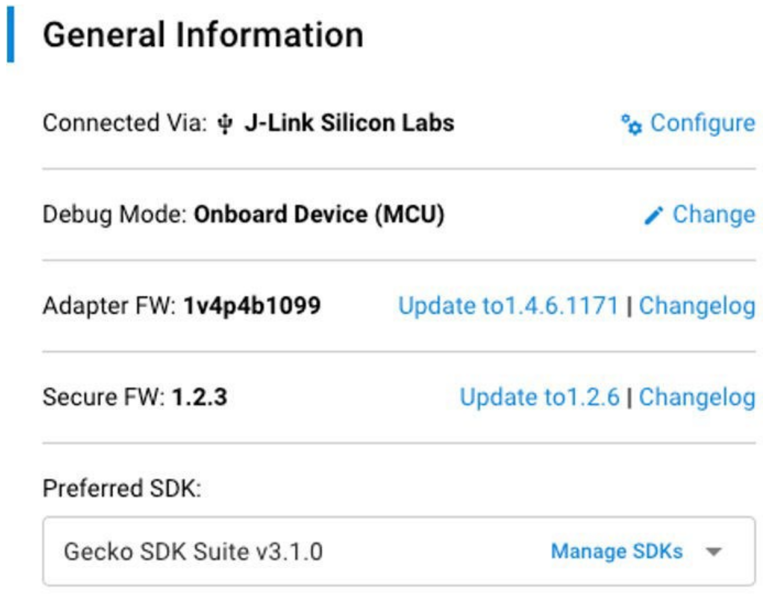
The Secure Engine is upgradable and requirements for upgrading the Secure Engine vary depending on the bootloader configuration:

- **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.


General Information

Connected Via: J-Link Silicon Labs

Debug Mode: Onboard Device (MCU)

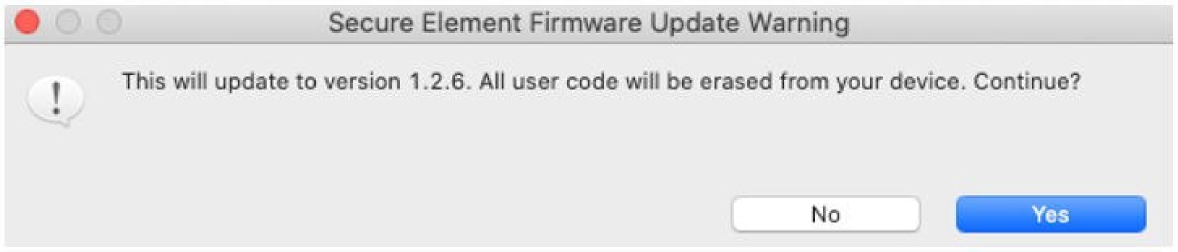
Adapter FW: 1v4p4b1099

Secure FW: 1.2.3

Preferred SDK:

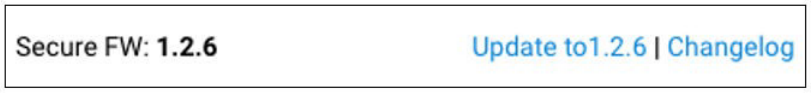
Gecko SDK Suite v3.1.0

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


Secure Element Firmware Update Warning

This will update to version 1.2.6. All user code will be erased from your device. Continue?

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


Secure FW: 1.2.6

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.

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

The process is illustrated in the following figure.

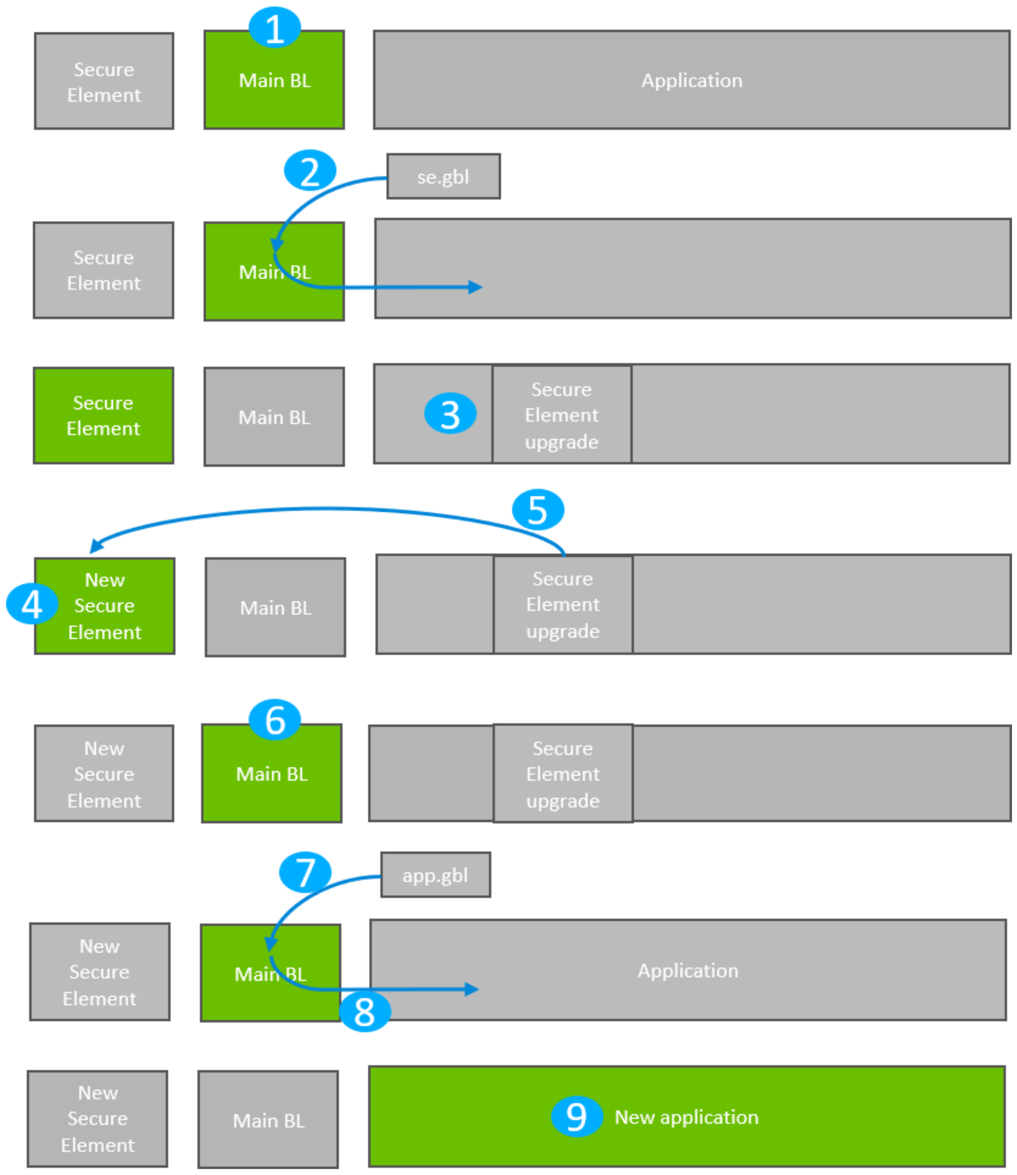


Figure 5.1. Standalone Bootloader: Secure Engine Bootloader Upgrade

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.

5.1.1 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 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.

5.1.2 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 3.1 Standalone Bootloader Operation.

5.2 Secure Engine Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.

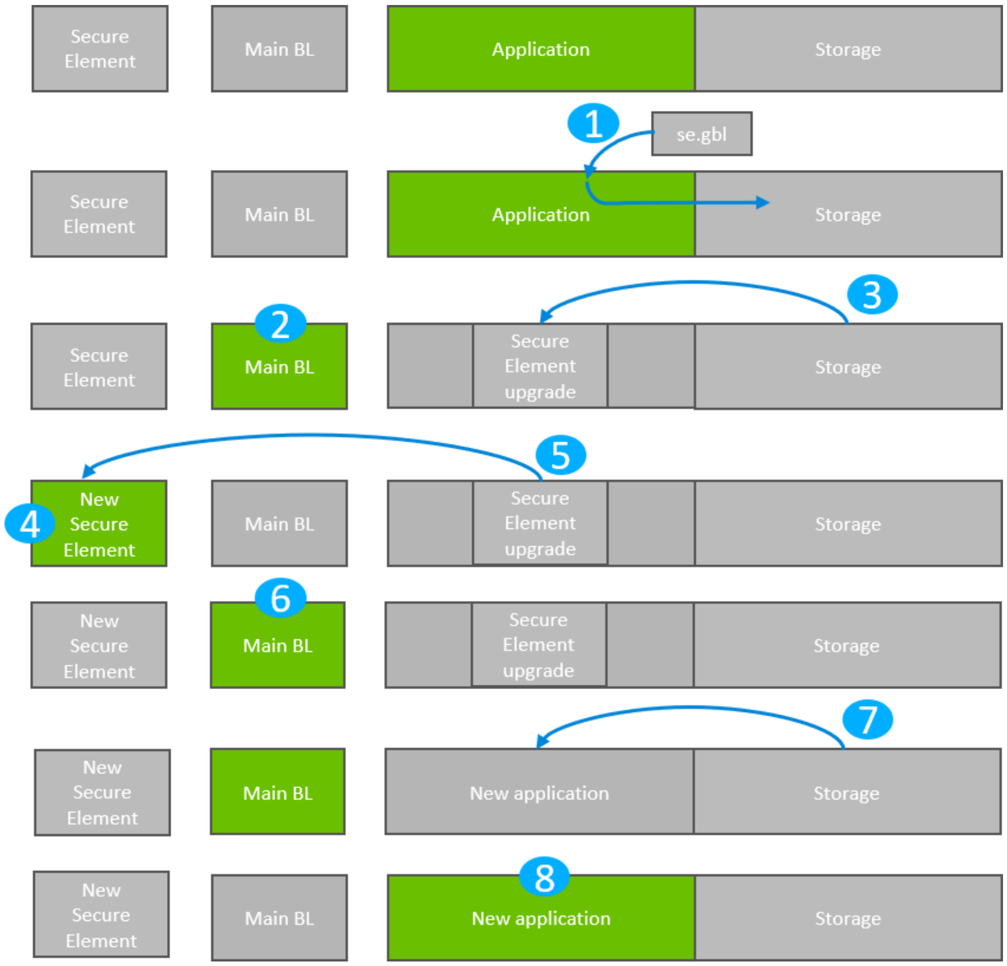


Figure 5.2. Application Bootloader: Secure Engine Upgrade

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.
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.

5.2.1 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

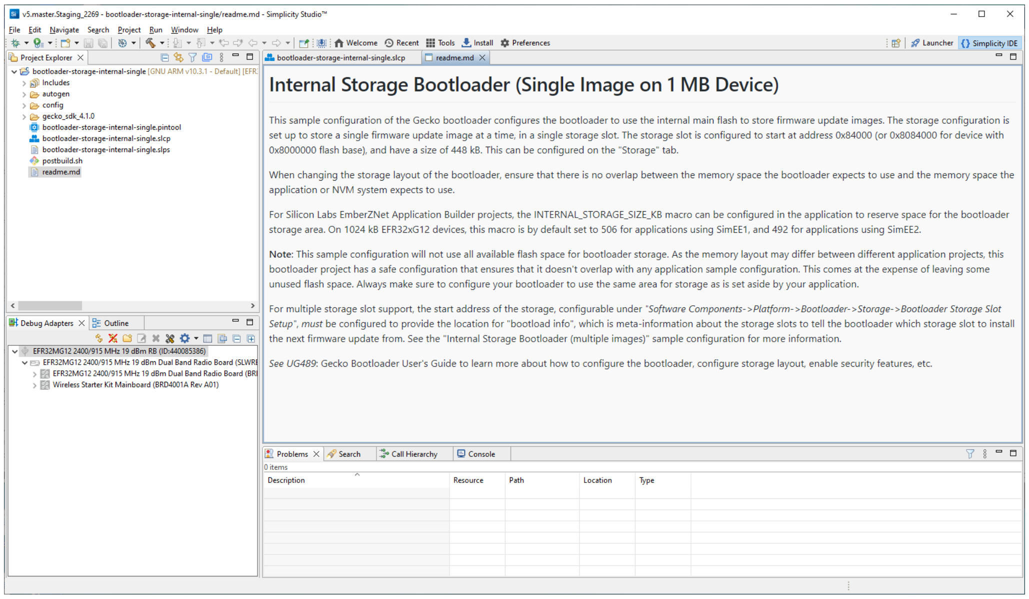
Depending on the configuration, the bootloader size can vary. For size requirements of the bootloader, see section 7.8 Size Requirements for Different Bootloader Configurations for Series 1 Devices. The bootloader size for EFR32xG21 devices can be up to 16 kB and for EFR32xG22, EFR32xG23, and EFR32xG24 devices the bootloader size can be up to 24 kB. For more details, see *UG103.6: Bootloader Fundamentals*.

6 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*
- *QSG168: 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.



Internal Storage Bootloader (Single Image on 1 MB Device)

This sample configuration of the Gecko bootloader configures the bootloader to use the internal main flash to store firmware update images. The storage configuration is set up to store a single firmware update image at a time, in a single storage slot. The storage slot is configured to start at address 0x84000 (or 0x8084000 for device with 0x8000000 flash base), and have a size of 448 kB. This can be configured on the "Storage" tab.

When changing the storage layout of the bootloader, ensure that there is no overlap between the memory space the bootloader expects to use and the memory space the application or NVM system expects to use.

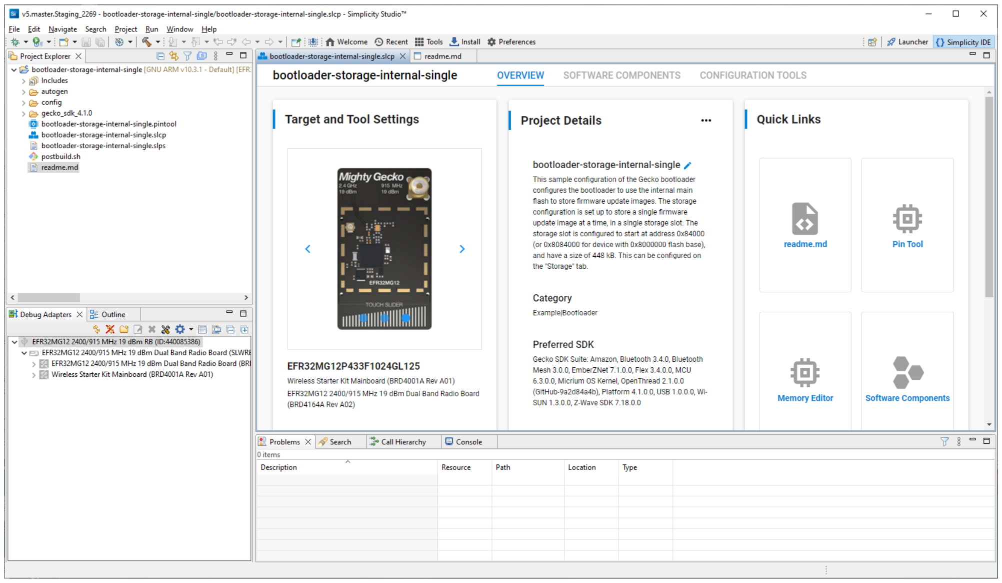
For Silicon Labs EmberZNet Application Builder projects, the INTERNAL_STORAGE_SIZE_KB macro can be configured in the application to reserve space for the bootloader storage area. On 1024 kB EFR32xG12 devices, this macro is by default set to 506 for applications using SimEE1, and 492 for applications using SimEE2.

Note: This sample configuration will not use all available flash space for bootloader storage. As the memory layout may differ between different application projects, this bootloader project has a safe configuration that ensures that it doesn't overlap with any application sample configuration. This comes at the expense of leaving some unused flash space. Always make sure to configure your bootloader to use the same area for storage as is set aside by your application.

For multiple storage slot support, the start address of the storage, configurable under "*Software Components->Platform->Bootloader->Storage->Bootloader Storage Slot Setup*", *must* be configured to provide the location for "bootload info", which is meta-information about the storage slots to tell the bootloader which storage slot to install the next firmware update from. See the "Internal Storage Bootloader (multiple images)" sample configuration for more information.

See UG489: Gecko Bootloader User's Guide to learn more about how to configure the bootloader, configure storage layout, enable security features, etc.

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



bootloader-storage-internal-single

Project Details

bootloader-storage-internal-single

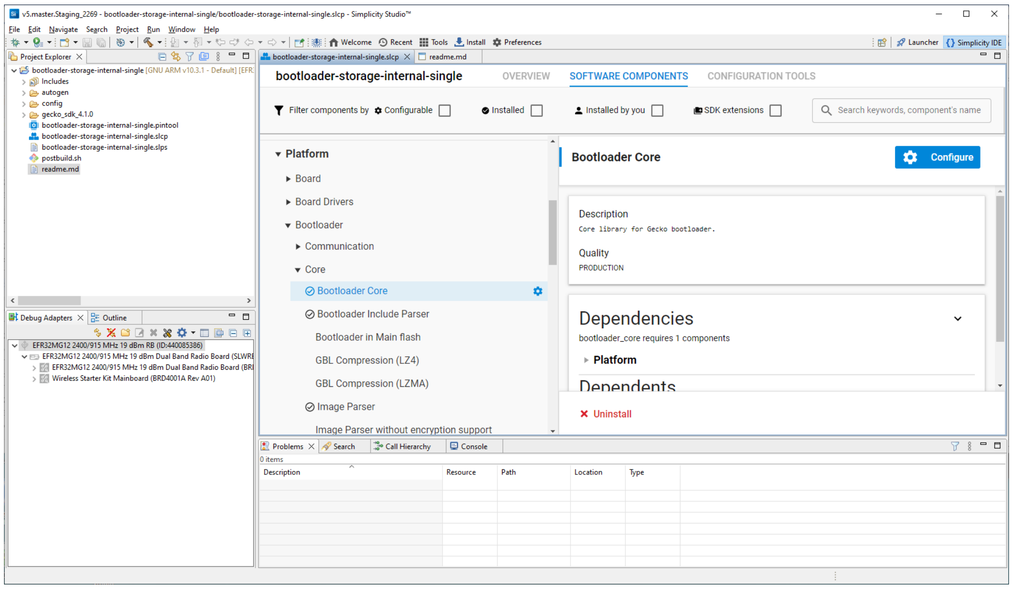
This sample configuration of the Gecko bootloader configures the bootloader to use the internal main flash to store firmware update images. The storage configuration is set up to store a single firmware update image at a time, in a single storage slot. The storage slot is configured to start at address 0x84000 (or 0x8084000 for device with 0x8000000 flash base), and have a size of 448 kB. This can be configured on the "Storage" tab.

Category: Example|Bootloader

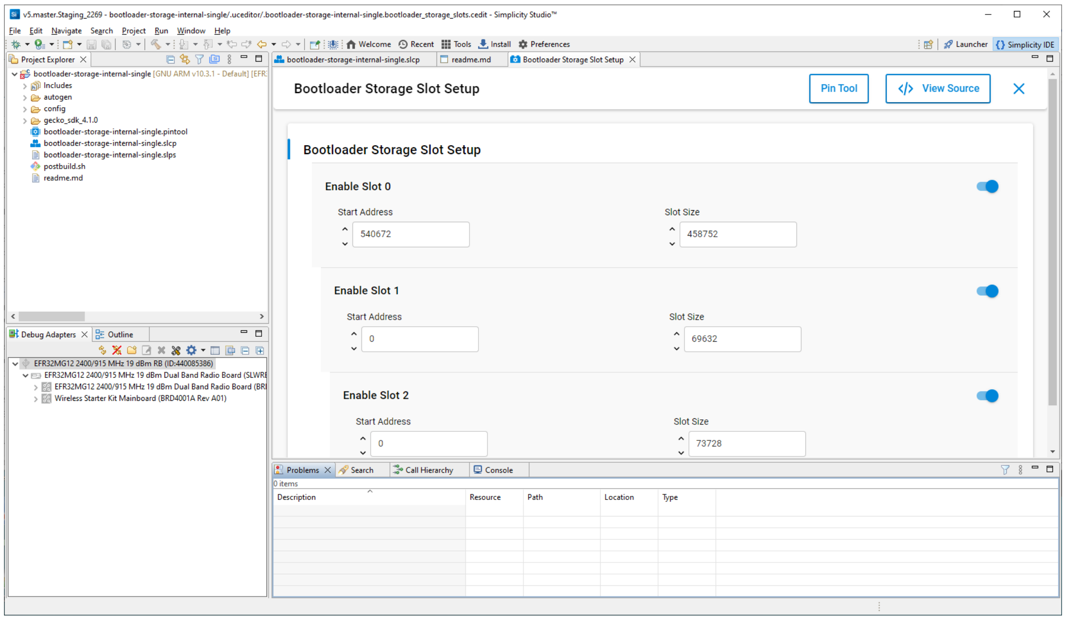
Preferred SDK: Gecko SDK Suite: Amazon, Bluetooth 3.4.0, Bluetooth Mesh 3.0.0, EmberZNet 7.1.0.0, Flex 3.4.0.0, MCU 6.3.0.0, Micrium OS Kernel, OpenThread 2.1.0.0 (GitHub-9a2d84a4b), Platform 4.1.0.0, USB 1.0.0.0, Wi-SUN 1.3.0.0, Z-Wave SDK 7.18.0.0

Target and Tool Settings: EFR32MG12P433F1024GL125
Wireless Starter Kit Mainboard (BRD4001A Rev A01)
EFR32MG12 2400/915 MHz 19 dBm Dual Band Radio Board (BRD4164A Rev A02)

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


v5.master.Staging_2269 - bootloader-storage-internal-single/bootloader-storage-internal-single.slcp - Simplicity Studio™
File Edit Navigate Search Project Run Window Help
Welcome Recent Tools Install Preferences
Launcher Simplicity IDE
Project Explorer
bootloader-storage-internal-single [GNU ARM v10.3.1 - Default] [EFR
Includes
autogen
config
gecko_sdk_4.1.0
bootloader-storage-internal-single.pintool
bootloader-storage-internal-single.slcp
bootloader-storage-internal-single.slps
postbuild.sh
readme.md
Debug Adapters
Outline
EFR32MG12 2400/915 MHz 19 dBm RB (ID:440085386)
EFR32MG12 2400/915 MHz 19 dBm Dual Band Radio Board (SLWRB
EFR32MG12 2400/915 MHz 19 dBm Dual Band Radio Board (BR
Wireless Starter Kit Mainboard (BRD4001A Rev A01)
bootloader-storage-internal-single.slcp
readme.md
bootloader-storage-internal-single
OVERVIEW
SOFTWARE COMPONENTS
CONFIGURATION TOOLS
Filter components by
Configurable
Installed
Installed by you
SDK extensions
Search keywords, component's name
Platform
Board
Board Drivers
Bootloader
Communication
Core
Bootloader Core
Bootloader Include Parser
Bootloader in Main flash
GBL Compression (LZ4)
GBL Compression (LZMA)
Image Parser
Image Parser without encryption support
Bootloader Core
Configure
Description
Core library for Gecko bootloader.
Quality
PRODUCTION
Dependencies
bootloader_core requires 1 components
Platform
Dependents
Uninstall
Problems
Search
Call Hierarchy
Console
0 items
Description
Resource
Path
Location
Type

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.


v5.master.Staging_2269 - bootloader-storage-internal-single/.uceditor/.bootloader-storage-internal-single.bootloader_storage_slots.cedit - Simplicity Studio™
File Edit Navigate Search Project Run Window Help
Welcome Recent Tools Install Preferences
Launcher Simplicity IDE
Project Explorer
bootloader-storage-internal-single [GNU ARM v10.3.1 - Default] [EFR
Includes
autogen
config
gecko_sdk_4.1.0
bootloader-storage-internal-single.pintool
bootloader-storage-internal-single.slcp
bootloader-storage-internal-single.slps
postbuild.sh
readme.md
Debug Adapters
Outline
EFR32MG12 2400/915 MHz 19 dBm RB (ID:440085386)
EFR32MG12 2400/915 MHz 19 dBm Dual Band Radio Board (SLWRB
EFR32MG12 2400/915 MHz 19 dBm Dual Band Radio Board (BR
Wireless Starter Kit Mainboard (BRD4001A Rev A01)
bootloader-storage-internal-single.slcp
readme.md
Bootloader Storage Slot Setup
Pin Tool
View Source
Problems
Search
Call Hierarchy
Console
0 items
Description
Resource
Path
Location
Type

Bootloader Storage Slot Setup

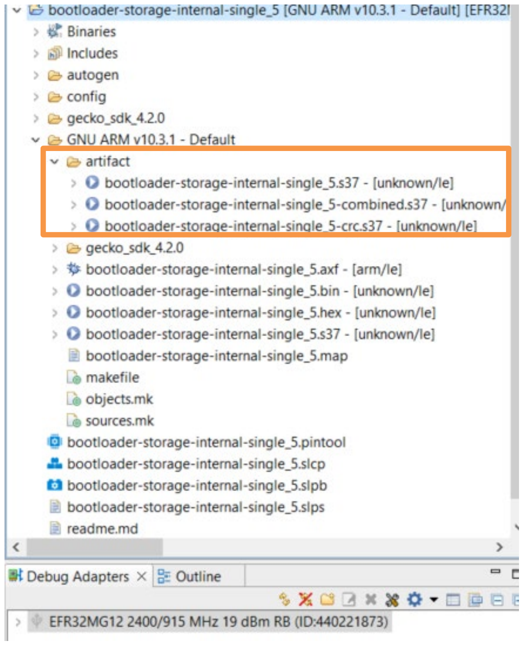
Enable Slot 0
Start Address: 540672
Slot Size: 458752

Enable Slot 1
Start Address: 0
Slot Size: 69632

Enable Slot 2
Start Address: 0
Slot Size: 73728

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.



```
bootloader-storage-internal-single_5 [GNU ARM v10.3.1 - Default] [EFR32I
  Binaries
  Includes
  autogen
  config
  gecko_sdk_4.2.0
  GNU ARM v10.3.1 - Default
    artifact
      bootloader-storage-internal-single_5.s37 - [unknown/le]
      bootloader-storage-internal-single_5-combined.s37 - [unknown/
      bootloader-storage-internal-single_5-crc.s37 - [unknown/le]
    gecko_sdk_4.2.0
    bootloader-storage-internal-single_5.axf - [arm/le]
    bootloader-storage-internal-single_5.bin - [unknown/le]
    bootloader-storage-internal-single_5.hex - [unknown/le]
    bootloader-storage-internal-single_5.s37 - [unknown/le]
    bootloader-storage-internal-single_5.map
    makefile
    objects.mk
    sources.mk
  bootloader-storage-internal-single_5.pintool
  bootloader-storage-internal-single_5.slcp
  bootloader-storage-internal-single_5.slpb
  bootloader-storage-internal-single_5.slps
  readme.md
```

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.

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.

7 Configuring the Gecko Bootloader

7.1 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.

7.1.1 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.

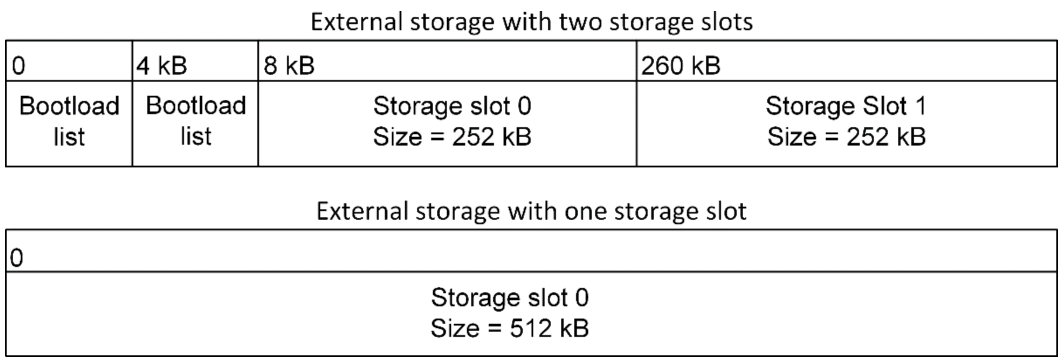

External storage with two storage slots
0
4 kB
8 kB
260 kB
Bootload list
Bootload list
Storage slot 0 Size = 252 kB
Storage Slot 1 Size = 252 kB
External storage with one storage slot
0
Storage slot 0 Size = 512 kB

Figure 7.1. SPI Flash Storage Area Configuration

7.1.2 Internal Storage Configuration

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.

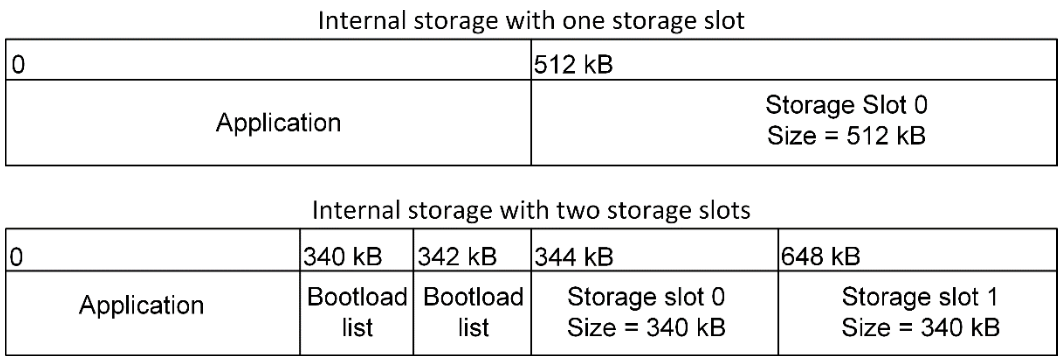


Figure 7.2. Internal Storage Area Configurations

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.

7.2 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%
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.

7.2.1 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.

7.3 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 9.3 Using Application Image Security Features to learn how to configure the security features of the Gecko Bootloader.

7.3.1 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.

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.

7.3.2 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.

7.3.3 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.

7.3.4 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.

7.3.5 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**.

7.3.6 SPI Flash Storage Bootloader SFDP

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*.

7.3.7 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 7.1. 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**
"	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)

7.4 Image Acquisition Application Example Configuration

These examples illustrate applications that acquire and store a GBL upload image for an application bootloader. For the running boot-loader 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.

7.5 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 4.1.1 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.

7.6 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.

7.7 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.

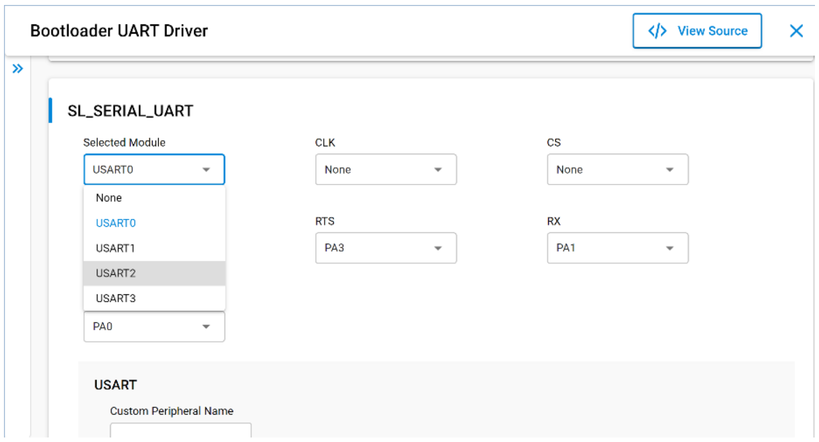


Figure 7.3. Example of USART Configuration for UART Driver

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.

7.8 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 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 7.2. 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

8 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.

8.1 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 5 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.`

9 Gecko Bootloader Security Features

9.1 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 *AN1218: Series 2 Secure Boot with RTSL*.

9.2 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.

9.2.1 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 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 9.4.1 Key Storage.

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

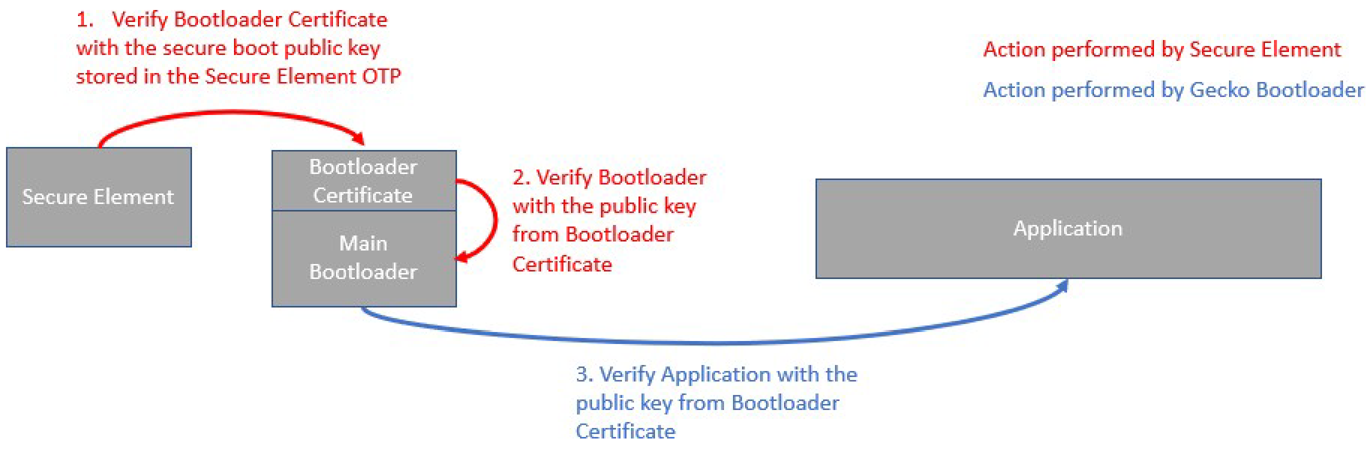

1. Verify Bootloader Certificate with the secure boot public key stored in the Secure Element OTP
Action performed by Secure Element
Action performed by Gecko Bootloader
Secure Element
Bootloader Certificate
Main Bootloader
2. Verify Bootloader with the public key from Bootloader Certificate
Application
3. Verify Application with the public key from Bootloader Certificate

Figure 9.1. Certificate-Based Secure Boot Procedure

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.

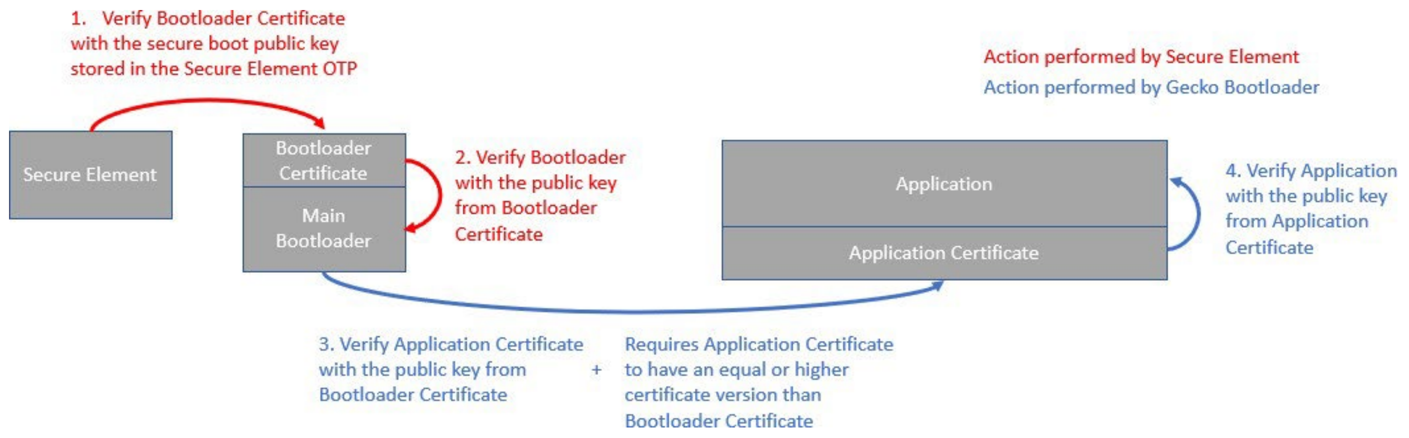

1. Verify Bootloader Certificate with the secure boot public key stored in the Secure Element OTP
Action performed by Secure Element
Action performed by Gecko Bootloader
Secure Element
Bootloader Certificate
Main Bootloader
2. Verify Bootloader with the public key from Bootloader Certificate
Application
Application Certificate
4. Verify Application with the public key from Application Certificate
3. Verify Application Certificate with the public key from Bootloader Certificate
+
Requires Application Certificate to have an equal or higher certificate version than Bootloader Certificate

Figure 9.2. Advanced Certificate-Based Secure Boot Procedure

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>
```

9.2.2 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. 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 *AN1222: Production Programming of Series 2 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.

9.3 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.

9.3.1 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 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
```

9.3.2 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 12.1 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
```

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
```

9.3.3 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.

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 12.1 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`.

9.4 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.

9.4.1 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 *AN1218: Series 2 Secure Boot with RTSL* and *UG162: 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.

9.4.2 Write-Protecting the Bootloader

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.

9.4.3 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.

9.4.4 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 Secure Engine. Thereafter, the debug lock command can be issued to the Secure Engine. For more details, refer to *AN1190: EFR32xG21 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.

10 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.

10.1 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 *AN1374: Series 2 TrustZone* and *UG520: 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 Section 6 of *AN1374: 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.

10.2 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.

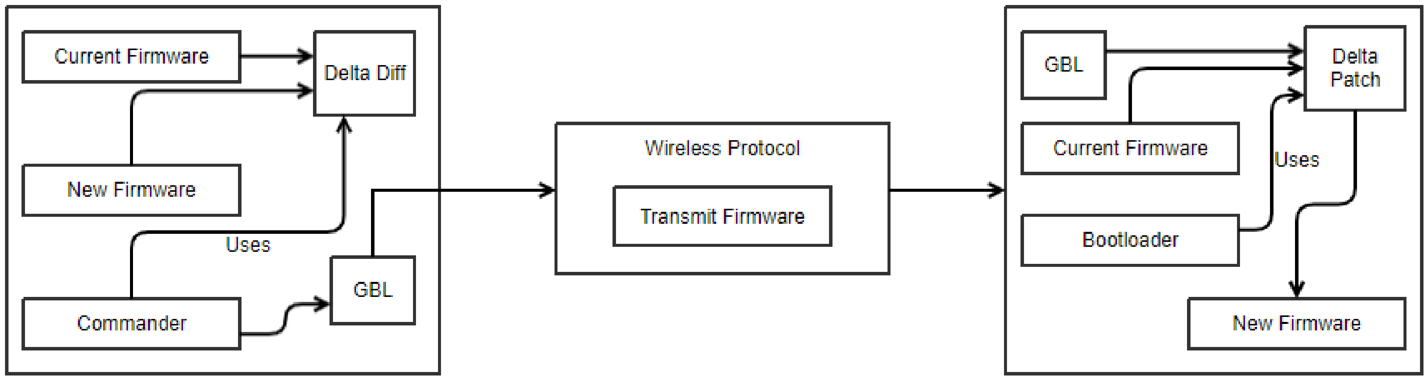
11 Gecko Bootloader and Delta DFU Upgrades

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.

11.1 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.



11.2 Gecko Bootloader Operation

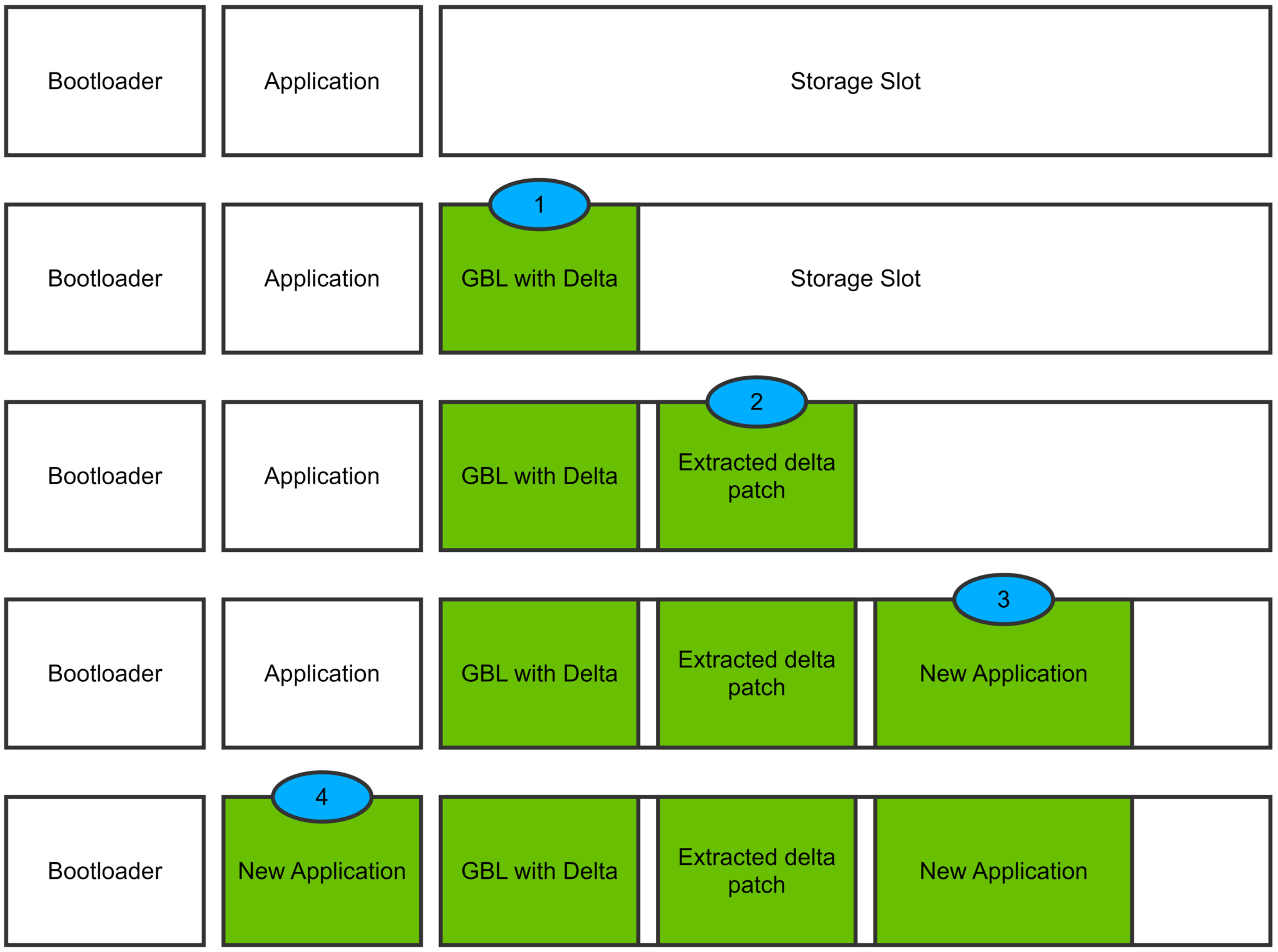
The Delta DFU upgrades can be enabled by including the Delta DFU component in the Gecko Bootloader. Simplicity Commander supports the creation of a Delta DFU GBL file.

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:



12 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)

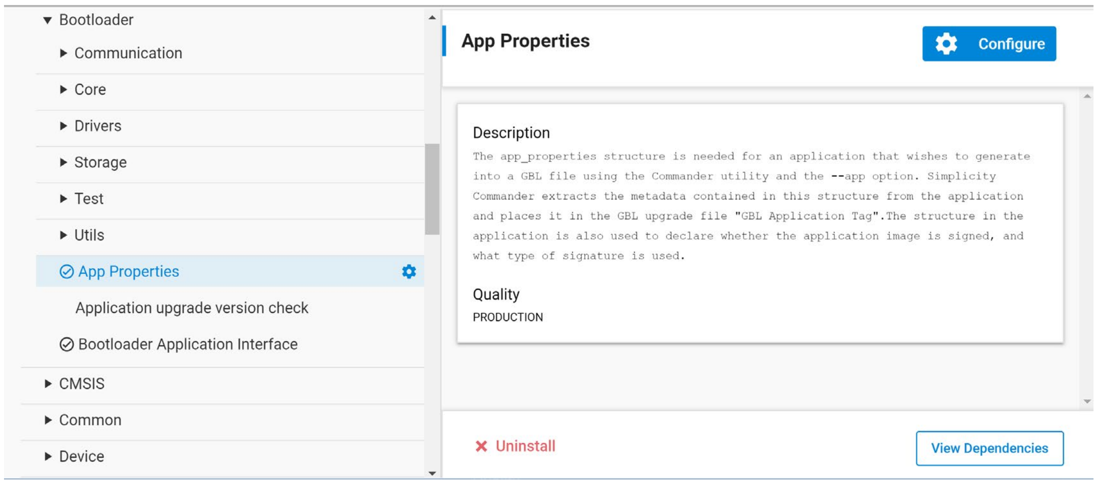
api/btl_interface_storage.c (interface to storage functionality)

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.

12.1 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.



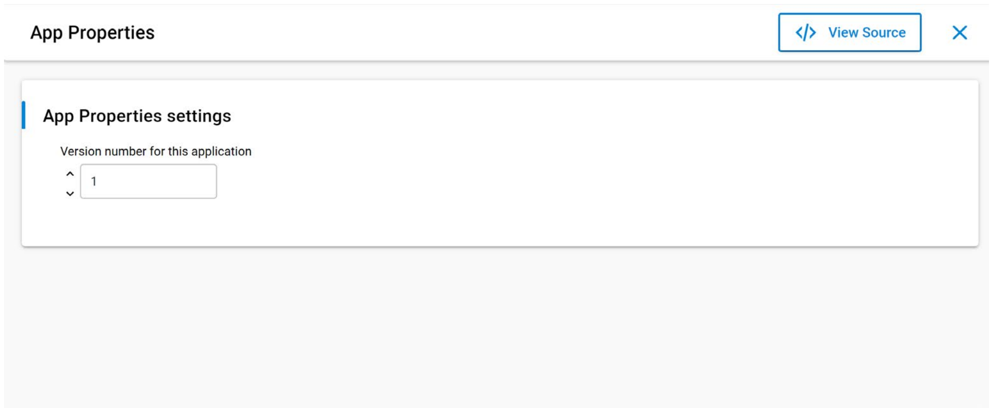
Description

The app_properties structure is needed for an application that wishes to generate into a GBL file using the Commander utility and the --app option. Simplicity Commander extracts the metadata contained in this structure from the application and places it in the GBL upgrade file "GBL Application Tag".The structure in the application is also used to declare whether the application image is signed, and what type of signature is used.

Quality

PRODUCTION

Click **Configure** to open the Component Editor.



App Properties settings

Version number for this application

1

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.

12.2 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 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