



UG266: Silicon Labs Gecko Bootloader User's Guide

This document describes the high-level implementation of the Silicon Labs Gecko Bootloader for EFR32 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: Bootloading Fundamentals*. For more information on using the Gecko Bootloader with different stacks, see the following:

- *AN1084: Using the Gecko Bootloader with EmberZNet and Silicon Labs Thread*
- *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*
- *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*

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	4
1.1 Core	5
1.2 Drivers	5
1.3 Plugins	6
1.3.1 Communication	6
1.3.2 Compression	6
1.3.3 Debug	6
1.3.4 GPIO Activation	6
1.3.5 Security	7
1.3.6 Storage	8
2. Gecko Bootloader Operation - Application Upgrade	10
2.1 Standalone Bootloader Operation	10
2.1.1 Rebooting Into the Bootloader	10
2.1.2 Downloading and Applying a GBL Upgrade File	10
2.1.3 Booting Into the Application	11
2.1.4 Error handling	11
2.2 Application Bootloader Operation	12
2.2.1 Downloading and Storing a GBL Image Upgrade File	13
2.2.2 Rebooting and Applying a GBL Upgrade File	13
2.2.3 Booting Into the Application	13
3. Gecko Bootloader Operation - Bootloader Upgrade	14
3.1 Bootloader Upgrade on Bootloaders With Communication Interface (Standalone Bootloaders).	14
3.1.1 Downloading and Applying a Bootloader GBL Upgrade File	15
3.1.2 Downloading and Applying an Application GBL Upgrade File	15
3.2 Bootloader Upgrade on Bootloaders With Storage (such as SoCs)	16
4. Getting Started with the Gecko Bootloader	18
5. Configuring the Gecko Bootloader	22
5.1 Configuring Storage	22
5.1.1 SPI Flash Storage Configuration	22
5.1.2 Internal Storage Configuration	23
5.2 Compressed Upgrade Images	24
5.2.1 LZMA Compression Settings	24
5.3 Bootloader Example Configurations	24
5.3.1 UART XMODEM Bootloader	25
5.3.2 BGAPI UART DFU Bootloader	25
5.3.3 EZSP SPI Bootloader	25
5.3.4 SPI Flash Storage Bootloader	25
5.3.5 Internal Storage Bootloader	25
5.3.6 Bluetooth In-Place OTA DFU Bootloader	26
5.4 Setting a Version Number For The Bootloader	26

6. Simplicity Commander and the Gecko Bootloader 27

6.1 Creating GBL Files Using Simplicity Commander 27

7. Gecko Bootloader Security Features 28

7.1 About Security Features 28

7.1.1 Secure Boot Procedure 28

7.1.2 Secure Firmware Upgrade 29

7.2 Using Gecko Bootloader Security Features 29

7.2.1 Generating Keys 30

7.2.2 Signing an Application Image for Secure Boot 30

7.2.3 Creating a Signed and Encrypted GBL Upgrade Image File From an Application 31

7.3 System Security Considerations. 32

8. Application Interface 33

8.1 Application Properties 33

8.2 Error Codes 34

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 bootload 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). These images are produced with the file extension “.gbl”. Additional information on the GBL file format is provided in *UG103.6: Application Development Fundamentals: Bootloading*.

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

The main bootloader consists of a common core, drivers, and a set of plugins that give the bootloader specific capabilities. The common bootloader core is delivered as a precompiled library, while the plugins are 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 plugin configuration. Plugins can be enabled and configured through the Simplicity Studio IDE.

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

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

Silicon Labs provides example bootloaders that come with a preconfigured set of plugins for configuration in either standalone or application mode, as described in section [5. Configuring the Gecko Bootloader](#). The Silicon Labs wireless protocol SDKs also include pre-compiled bootloader images for EFR32xG12 parts. As of this writing the images shown in the following table are provided.

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

Table 1.1. Prebuilt Bootloader Images

Use	Stack	Image Name	Mode	Interface
SoC	EmberZNet PRO / Silicon Labs Thread	SPI Flash Storage Bootloader	Application	SPI Serial Flash
SoC	Bluetooth	Bluetooth In-Place OTA DFU Bootloader	Application	OTA/internal flash
NCP	EmberZNet PRO / Silicon Labs Thread	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), 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 has to 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] has to 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.

The following sections provide an overview of the Gecko Bootloader common core, drivers, and plugins. 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. Read more about how to use Simplicity Commander with Gecko bootloader in section 6. [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 first word of SRAM is used as shared memory between the bootloader and the application to flag the reason for a reset. The different possible reset reasons are defined in the Reset Information part of the Application Interface, in the file `btl_reset_info.h`.

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.

1.2 Drivers

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

Driver modules include:

- Delay: Simple delay routines for use with plugins 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 plugins 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 plugins 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 Plugins

All parts of the bootloader that are either optional or that may be exchanged for different configurations are implemented as plugins. Each plugin has a generic header file, and one or more implementations. Plugins include:

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

1.3.1 Communication

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

- BGAPI UART DFU: By enabling the BGAPI communication plugin, the bootloader communication interface implements the UART DFU protocol using BGAPI commands. This plugin makes the bootloader compatible with the legacy UART bootloader that was previously released with the Silicon Labs Bluetooth SDK versions 2.0.0-2.1.1. See *AN1053: Bluetooth® Device Firmware Update over UART for EFR32xG1 and BGM11x Series Products* for more information about this legacy bootloader.
- EZSP-SPI: By enabling the EZSP-SPI communication plugin, the bootloader communication interface implements the EZSP protocol over SPI. This plugin makes the bootloader compatible with the legacy ezsp-spi-bootloader that was previously released with the EmberZNet and Silicon Labs Thread wireless stacks. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.
- UART XMODEM: By enabling the UART XMODEM communication plugin, the bootloader communication interface implements the XMODEM-CRC protocol over UART. This plugin makes the bootloader compatible with the legacy serial-uart-bootloader that was previously released with the EmberZNet and Silicon Labs Thread wireless stacks. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.

1.3.2 Compression

The Compression plugins provide capability for the bootloader GBL file parser to handle compressed GBL upgrade images. Each compression plugin 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)* plugins.

1.3.3 Debug

This plugin provides the bootloader with support for debugging output. If the plugin 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.

1.3.4 GPIO Activation

This plugin 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.

1.3.5 Security

Security plugins 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 plugins 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 bootloading. 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:** The SPI flash storage implementation supports a variety of SPI flash parts. The subset of devices supported can be configured at compile time using the checkboxes found in the plugin options area for the SPI Flash Storage plugin in AppBuilder's Bootloader framework. (The default configuration if no checkboxes 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 the following table.

Note: The 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 (μ A Typical)*
Macronix MX25R8035F (low power)	1024	0.007
Macronix MX25R6435SF (low power)	8192	0.007
Spansion S25FL208K	1024	15
Winbond W25X20BVSNI (W25X20CVSNJG for high- temperature support)	256	1
Winbond W25Q80BVSNI (W25Q80BVSNIJG 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

Manufacturer Part Number	Size (kB)	Quiescent Current (μ A Typical)*
ISSI IS25LQ020B	256	8
ISSI IS25LQ040B	512	8

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

2. 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 3. [Gecko Bootloader Operation - Bootloader Upgrade](#) provides the same information for updating the bootloader firmware.

2.1 Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:

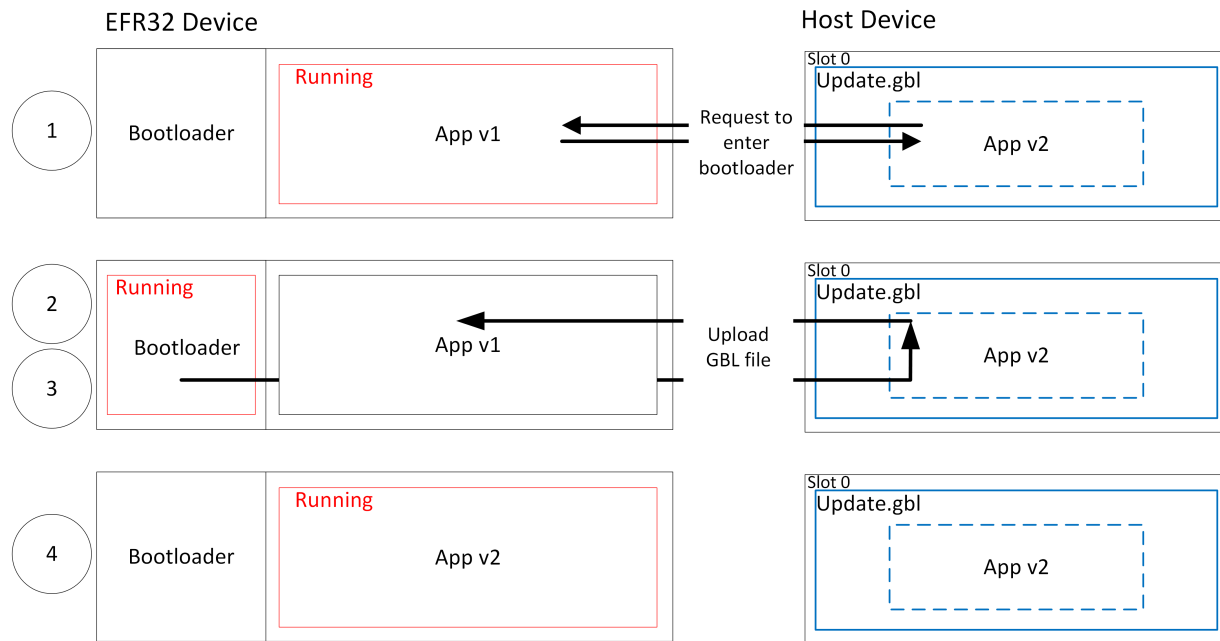


Figure 2.1. Standalone Bootloader Operation

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.

2.1.1 Rebooting Into the Bootloader

The Gecko Bootloader supports multiple mechanisms for triggering the bootloader. If the GPIO Activation plugin is enabled, 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.

2.1.2 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.

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.

2.1.3 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 that the Program Counter of the application is valid, and, optionally if Secure Boot is enabled, that the application passes signature verification.

2.1.4 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.

2.2 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.

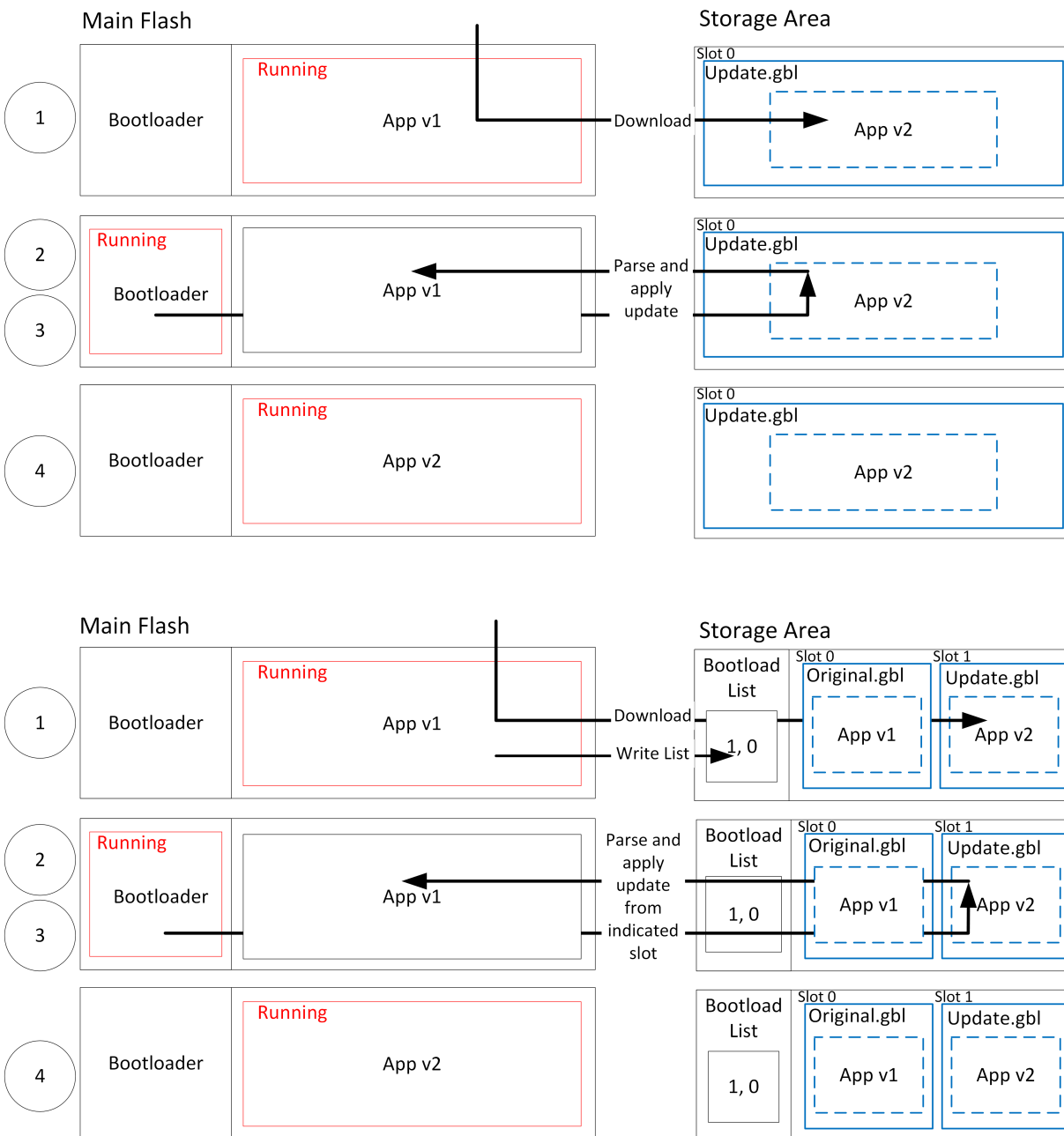


Figure 2.2. Application Bootloader Operation

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.

2.2.1 Downloading and Storing a GBL Image Upgrade File

To prepare for receiving an upgrade image, the application finds an available storage slot, or erases an existing one using `bootloader_eraseStorage`. 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, Thread, 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_setBootloadList`. 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 uses this slot implicitly.

2.2.2 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.

2.2.3 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 that the Program Counter of the application is valid and optionally, if Secure Boot is enabled, that the application passes signature verification.

3. Gecko Bootloader Operation - Bootloader Upgrade

The first stage bootloader is very simple and only knows how to upgrade the main bootloader. The first stage bootloader itself 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, as described in section [7.2.3 Creating a Signed and Encrypted GBL Upgrade Image File From an Application](#).

3.1 Bootloader Upgrade on Bootloaders With Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:

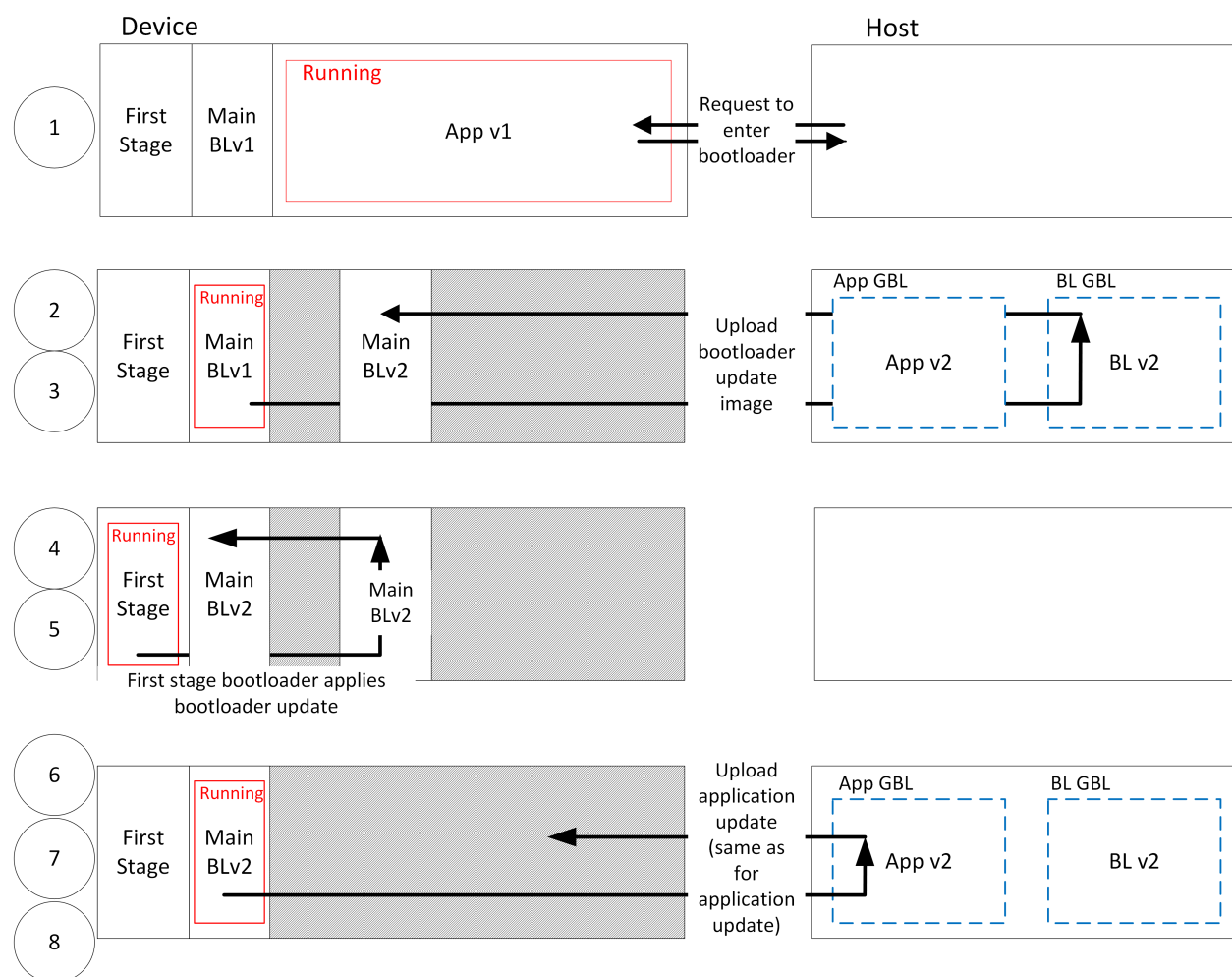


Figure 3.1. Standalone Bootloader: Bootloader Upgrade

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 fixed bootloader upgrade location in internal flash, overwriting the existing application.
4. The device reboots into the first stage bootloader.
5. The first stage bootloader replaces the main bootloader with the new version found in the fixed 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.

3.1.1 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 fixed bootloader upgrade address as given by the first stage bootloader.

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 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.

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. See section [5.4 Setting a Version Number For The Bootloader](#) for more information.

3.1.2 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 [2.1 Standalone Bootloader Operation](#).

3.2 Bootloader Upgrade on Bootloaders With Storage (such as SoCs)

The process is illustrated in the following figure.

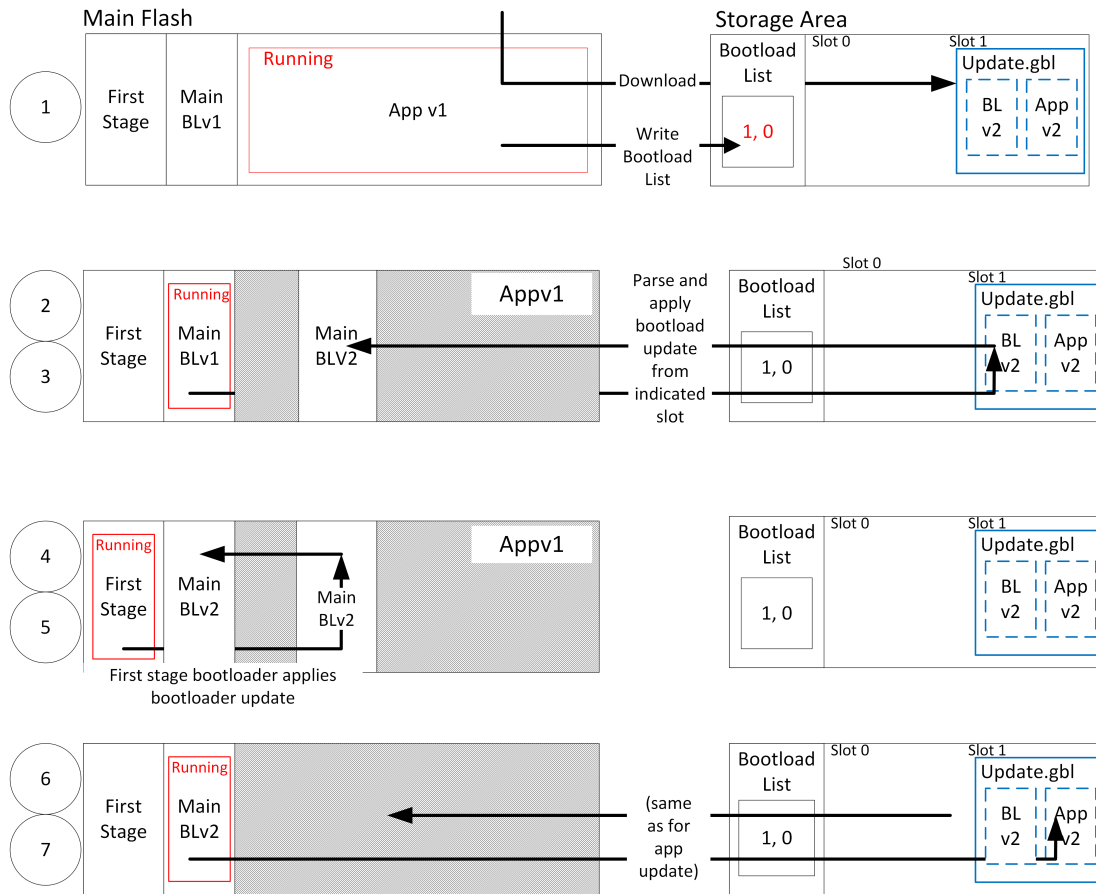


Figure 3.2. Application Bootloader: Bootloader Upgrade

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 bootloader.
3. The main bootloader copies its upgrade image into internal flash at the fixed bootloader upgrade location, overwriting the existing application.
4. The device reboots into the first stage bootloader.
5. The first stage bootloader replaces the main bootloader with the new version.
6. The device boots into the new main bootloader.
7. The bootloader applies the application image from the GBL upgrade file.
8. 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 given in the First Stage Bootloader Table.

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.

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.

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. See section [5.4 Setting a Version Number For The Bootloader](#) for more information.

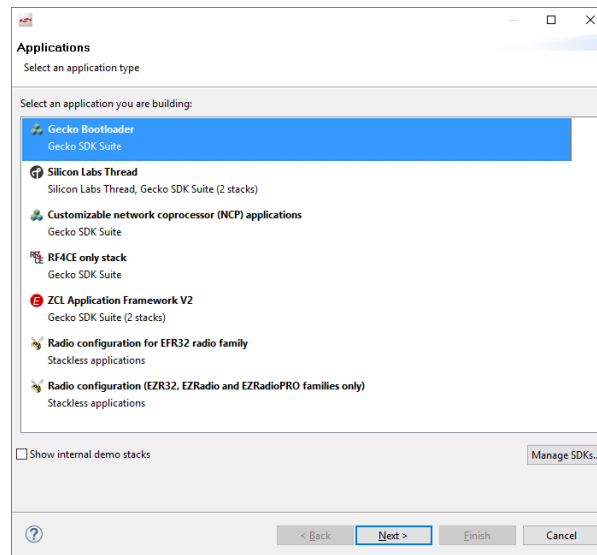
The new main bootloader is entered, and the images in the list of storage slots marked for bootload are verified. When the image parser parses the slot containing the GBL file with the bootloader + application upgrade, the version number of the bootloader upgrade is equal to the running main bootloader 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 [2.2 Application Bootloader Operation](#).

4. Getting Started with the Gecko Bootloader

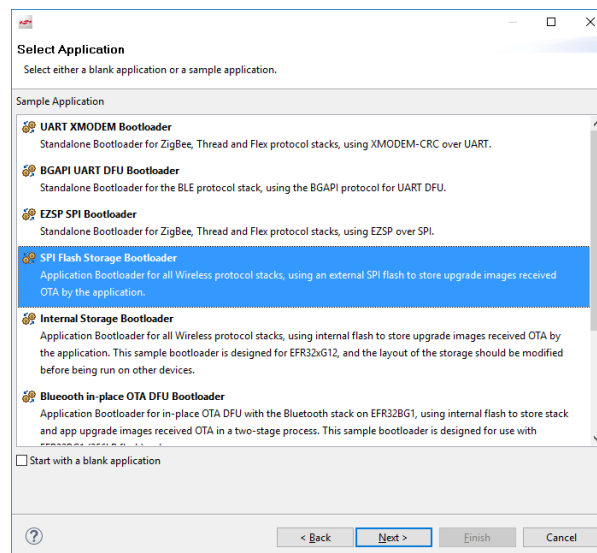
This section describes how to build a Gecko Bootloader from one of the provided examples. The instructions assume that you have installed the protocol SDK and associated utilities as described in the SDK's quick start guide, and that you are familiar with generating, compiling, and flashing an example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

- QSG106: *Getting Started with EmberZNet PRO*
- QSG113: *Getting Started with Silicon Labs Thread*
- QSG139: *Bluetooth Development with Simplicity Studio*
- QSG138: *Getting Started with the Silicon Labs Flex Software Development Kit for the Wireless Gecko (EFR32™) Portfolio*

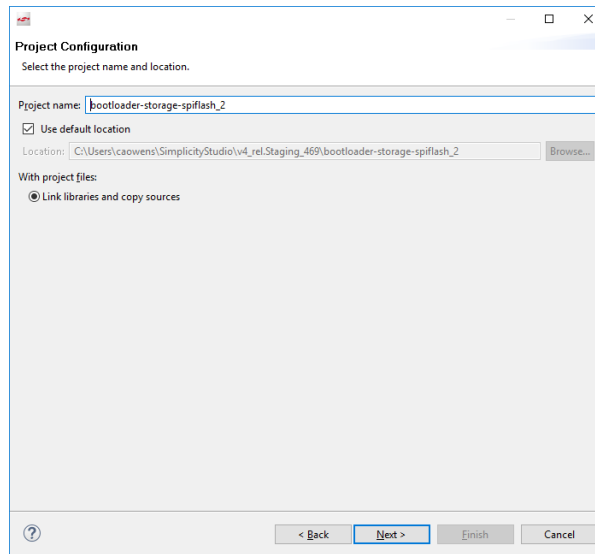
1. From the Launcher Perspective, click **[New Project]**.
2. In the Applications dialog, select **Gecko Bootloader** and click **[Next]**.



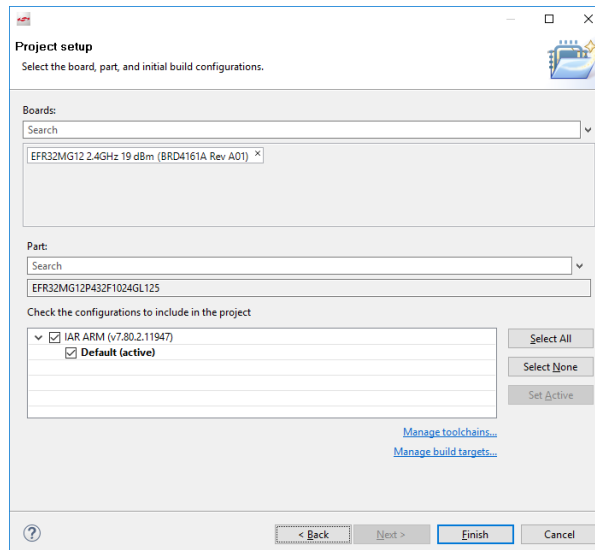
3. In the Select Application dialog, select your bootloader configuration example, and click **[Next]**.



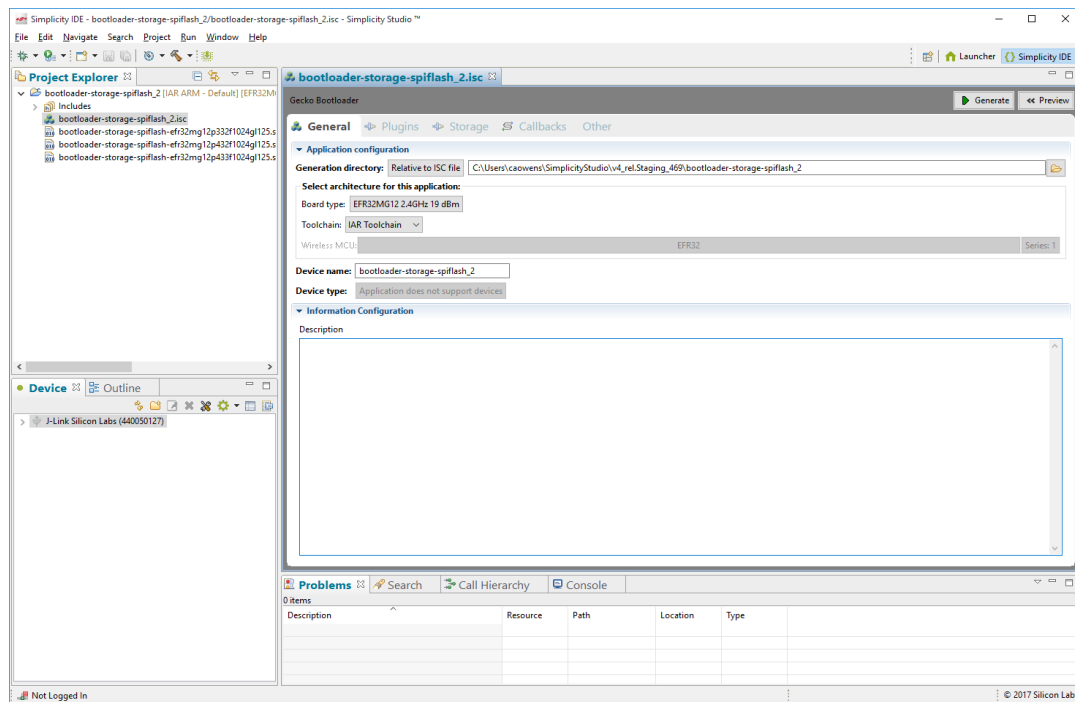
4. In the Project Configuration dialog, name your project and optionally select a different project location. Click **[Next]**.



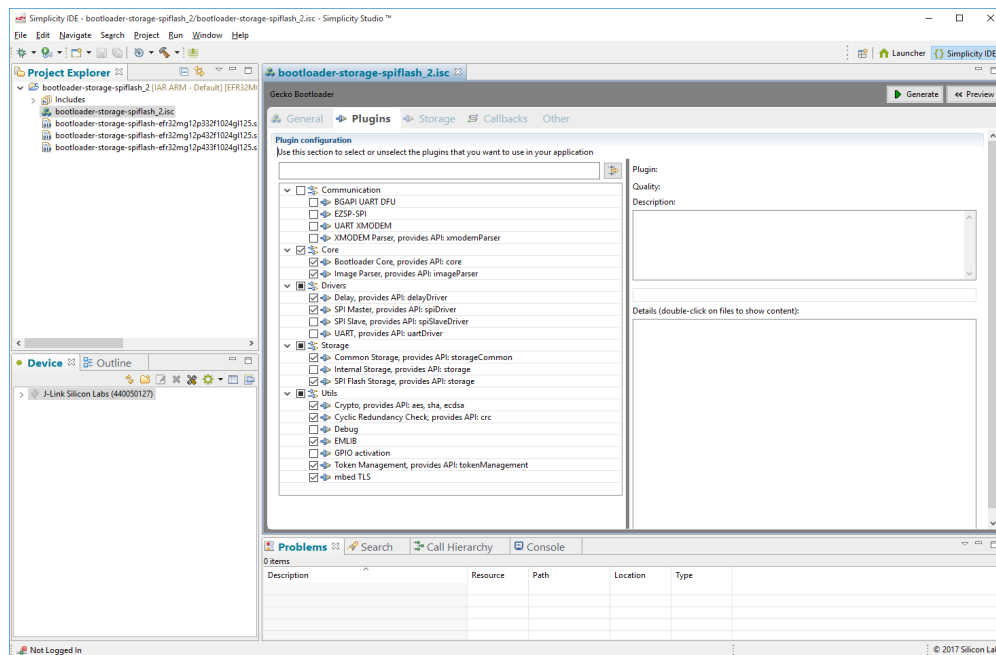
5. In the Project Setup dialog, if your part is not displayed, search for and select it. Select your compiler (in general, the same compiler you will use for the application). Click **[Finish]**. The Simplicity Studio IDE/AppBuilder perspective is displayed.



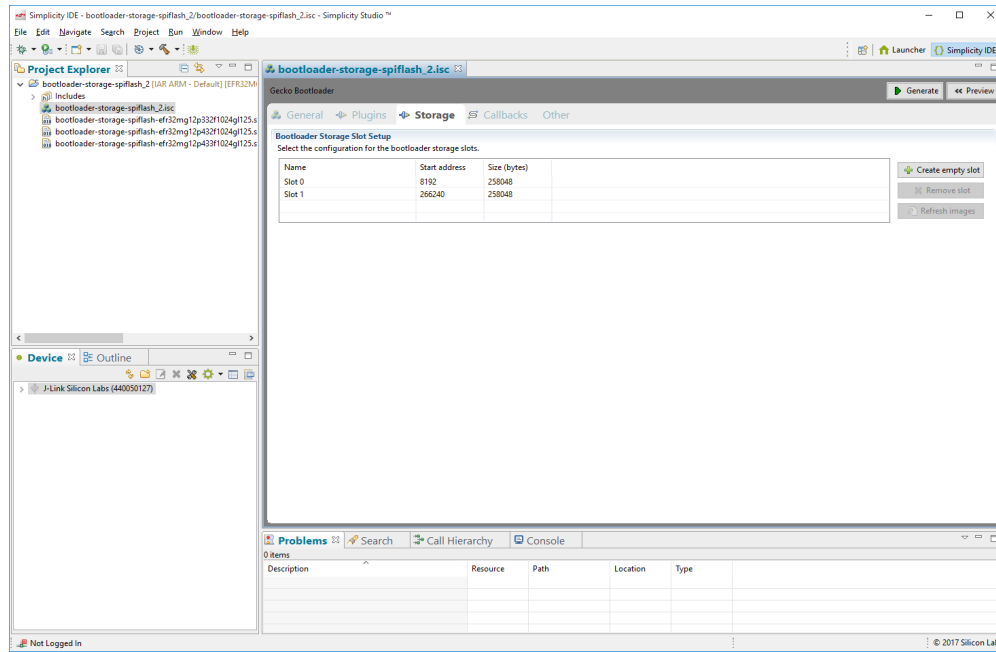
6. On the General tab, optionally enter a description.



The Plugins tab shows the configurations selected for the relevant example.




The Storage tab allows you to configure storage slots to be used if a storage plugin is enabled. The default configuration matches the target part and bootloader type.



7. Click **[Generate]**.

8. In the Generation Successful dialog, click **[OK]**.

9. Click the Build icon (). Two bootloader images are generated into the build directory: a main bootloader and a combined first stage and main bootloader. The main bootloader image is called **<projectname>.s37**, while the combined first stage + main bootloader image 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, the image containing only a main bootloader may be used. The image containing only a main bootloader is also the image that must be used to create a GBL file for bootloader upgrade.

5. Configuring the Gecko Bootloader

5.1 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. This is done by configuring the Storage plugins in the Bootloader application framework in Simplicity Studio.

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_setBootloadList` 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.

5.1.1 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 plugin. 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 on the Storage tab in AppBuilder. 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 plugin and the first storage slot configured on the Storage tab. 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.

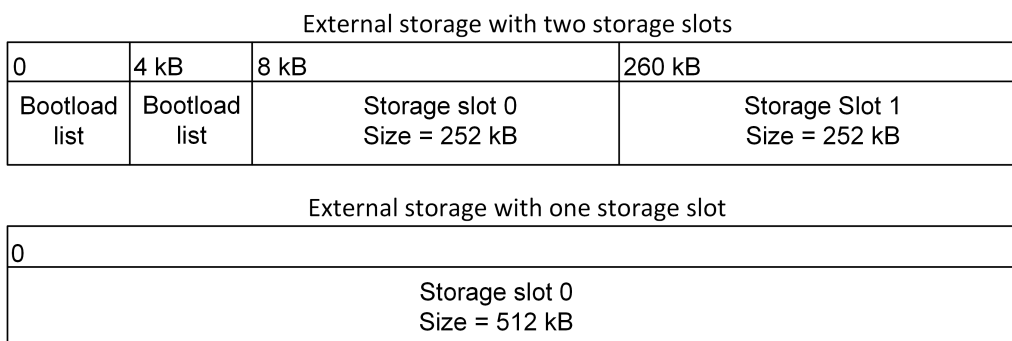


Figure 5.1. SPI Flash Storage Area Configuration

5.1.2 Internal 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 plugin. 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 on the Storage tab in AppBuilder. 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 plugin and the first storage slot configured on the Storage tab. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. The following figure illustrates how the storage area can be partitioned.

Internal storage with one storage slot				
0	512 kB			
Application	Storage Slot 0 Size = 512 kB			

Internal storage with two storage slots				
0	340 kB	342 kB	344 kB	648 kB
Application	Bootload list	Bootload list	Storage slot 0 Size = 340 kB	Storage slot 1 Size = 340 kB

Figure 5.2. 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.

5.2 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: Bootloading 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 AppBuilder plugin 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.

Compression Algorithm	Plugin	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZ4	GBL Compression (LZ4)	< 1 kB	~ 10%
LZMA	GBL 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 can not.

5.2.1 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: $lp + lc \leq 2$. 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.

5.3 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 the section [7.2 Using Gecko Bootloader Security Features](#) to learn how to configure the security features of the Gecko Bootloader.

5.3.1 UART XMODEM Bootloader

Standalone bootloader for EmberZNet PRO, Silicon Labs Thread, and Silicon Labs Flex protocol stacks, using XMODEM-CRC over UART.

In this configuration, the XMODEM UART communication plugin, XMODEM parser plugin, and UART driver plugin are enabled. In order for the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured. This is done by going to the Plugins tab of the AppBuilder project, and selecting the UART driver plugin. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation plugin is also enabled by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. This plugin can be disabled if this functionality is not desired, or the GPIO pin used for this can be configured under the GPIO Activation plugin on the Plugins tab.

5.3.2 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 plugin and UART driver plugin are enabled. In order for the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured. This is done by going to the Plugins tab of the AppBuilder project, and selecting the UART driver plugin. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation plugin is also enabled by default, allowing bootloader entry by activating a GPIO through reset. This plugin can be disabled if this functionality is not desired, or the GPIO pin used for this can be configured under the GPIO Activation plugin on the Plugins tab.

5.3.3 EZSP SPI Bootloader

Standalone bootloader for EmberZNet PRO, Silicon Labs Thread, and Silicon Labs Flex protocol stacks using EZSP for SPI.

In this configuration, the EZSP SPI communication plugin, XMODEM parser plugin, and SPI slave driver plugin are enabled. In order 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. This is done by going to the Plugins tab of the AppBuilder project, and selecting the SPI slave and EZSP SPI plugins respectively.

5.3.4 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 and common storage plugins, as well as the SPI driver plugin, are enabled. In order 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 SPI plugin, and the type of SPI flash needs to be configured in the SPI flash plugin. The base address of the storage area can be configured in the Common Storage plugin. The location and size of the storage slots themselves can be configured on the Storage tab in AppBuilder.

5.3.5 Internal Storage Bootloader

Application bootloader for all wireless protocol stacks, using internal flash to store upgrade images received over the air by the application. This example is designed for EFR32xG12. The layout of the storage should be modified before running the bootloader on any other devices. In this configuration, the internal flash and common storage plugins are enabled. The base address of the storage area is configured in the Common Storage plugin. The location and size of the storage slots can be configured on the Storage tab in AppBuilder. In the default example application, a single storage slot is configured.

5.3.6 Bluetooth In-Place OTA DFU Bootloader

Application bootloader for in-place over-the-air device firmware upgrade with the Bluetooth protocol stack, using internal flash to store stack and application upgrade images received over the air in a two-stage process. This example is designed for use with EFR32BG1 (256 kB flash) only.

In this configuration, the 256 kB internal flash is configured as follows.

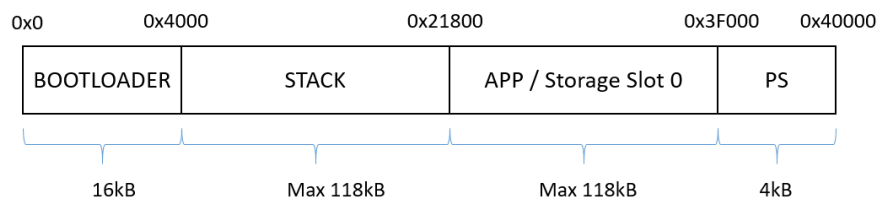


Figure 5.3. Flash Configuration in Bluetooth In-Place OTA DFU Bootloader

The first 16 kB are reserved for bootloader and the last 4 kB are used by Persistent Storage (PS). The size of these two are fixed. One continuous area between the bootloader and PS is used by the Bluetooth stack and user application. The size of this area is $(256 - 16 - 4) = 236$ kB. This area is split equally between the Bluetooth stack and the user application, meaning that the maximum size for both the stack and application is $236/2 = 118$ kB.

In a Bluetooth in-place OTA DFU Bootloader project one storage slot is defined as follows:

- Start at offset 0x21800 (137216)
- Size: 0x1D800 (120832 bytes)

5.4 Setting a Version Number For The Bootloader

In order to distinguish between different builds of the Gecko Bootloader, it is useful to set a version number. In order to perform a bootloader upgrade, the bootloader upgrade image must have a higher version number than the running bootloader image. A version number can be set in the Application Builder by defining the macro "BOOTLOADER_VERSION_MAIN_CUSTOMER" in the Additional Macros field on the Other tab, and checking the "-D?" checkbox to add the definition to the compiler command line. 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.

6. 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 keyfiles 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, refer to *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.

6.1 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.

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`.

7. Gecko Bootloader Security Features

7.1 About Security Features

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 the authenticity of an upgrade image before performing a bootloader, and optionally enforcing that upgrade images are encrypted.

7.1.1 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 enabled by default, and Silicon Labs recommends using it 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 in the device. Simplicity Commander can be used to generate a key pair and write the public key to the device. See *AN961: Bringing up Custom Devices for the Mighty Gecko and Flex Gecko Families* for more information.

7.1.2 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 in order to ensure its integrity and 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.

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.

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.

7.2 Using Gecko Bootloader Security Features

In this example, we assume that a bootloader called **bootloader-uart-xmodem** has been built using the Application Builder in Simplicity Studio. In the output directory, two files of interest have been generated:

- **bootloader-uart-xmodem.s37** – This file contains the main bootloader. Can be used for bootloader upgrade.
- **bootloader-uart-xmodem-combined.s37** – This file contains both the first stage and main bootloader 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.

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 a HSM to safeguard private keys.

7.2.1 Generating Keys

In order 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 gbl keygen --type ecc-p256 --outfile signing-key
```

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 gbl keygen --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
```

7.2.2 Signing an Application Image for Secure Boot

If the bootloader enforces Secure Boot, the application needs to be signed in order to pass verification. On every boot, an 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.

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
```

7.2.3 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 AppBuilder.

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**, and supply the resulting DER-formatted signature file **signature.der** back to Simplicity Commander:

```
commander gbl sign myapp-for-signing.gbl --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`.

7.3 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.

The encryption and signing 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.

By default, the region in flash used by the Gecko bootloader is readable and writeable. The region needs to stay readable in order for the Application Interface to be able to interact with the bootloader. Immediately after reset, the region also needs to be writable in order for the first stage bootloader to be able to perform bootloader upgrades. However, as soon as the main bootloader has started, 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. This is done by the Gecko bootloader main stage if the “Prevent bootloader write/erase” plugin option is enabled in the Core plugin in AppBuilder.

On EFR32xG1, where the bootloader resides in main flash rather than the information block, the BLWDIS option does not exist. On these devices, 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.

To prevent debugger access to the device, 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.

8. 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 plugins are present.

8.1 Application Properties

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. When using a protocol stack from Silicon Labs, this structure is already present in the application. 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 error can be suppressed using `--force`, in which case Simplicity Commander will add an all-zero **GBL Application Tag** to the GBL file. The contents of this 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`. In order 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.

8.2 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



Smart.
Connected.
Energy-Friendly.



Products

www.silabs.com/products



Quality

www.silabs.com/quality



Support and Community

community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Micrium, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, Z-Wave and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

<http://www.silabs.com>