



AN1084: Using the Gecko Bootloader with EmberZNet

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with EmberZNet. It supplements the general Gecko Bootloader implementation information provided in *UG266: Silicon Labs Gecko Bootloader User's Guide*. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to *UG103.6: Bootloading Fundamentals*.

KEY POINTS

- Gecko Bootloader overview.
- Using Gecko standalone bootloaders.
- Using Gecko application bootloaders.

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. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). These images are produced with the file extension “.gbl”. Additional information on the GBL file format is provided in *UG103.6: Bootloading Fundamentals*.

On EFR32 Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image by computing a CRC32 checksum, before copying the upgrade image to the main bootloader location.

On EFR32 Series 2 devices, the Gecko bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the hardware peripheral Secure Element. The Secure Element provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Element installs it. On some devices, the Secure Element is also capable of verifying the authenticity of the main bootloader upgrade image against a root of trust. The Secure Element itself is upgradable using the same mechanism. More details are provided in *UG266: Silicon Labs Gecko Bootloader User's Guide*.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the 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.

This document describes how to use both models with EmberZNet.

2. Using the Gecko Standalone Bootloaders

A Gecko Bootloader-based standalone bootloader receives an application image onto a target device by serial transfer via SPI or UART. If using UART, you can establish a serial connection between a source device and a target device's serial interface and upload a new software image to it using the XModem protocol. If you need information on the XModem protocol, a good place to start is <http://en.wikipedia.org/wiki/XMODEM>, which should have a brief description and up-to-date links to protocol documentation.

2.1 Performing a Serial Upload – UART XMODEM Bootloader

Serial upload can be performed with any source device that provides the expected serial interface method. This can be a Windows-based PC, a Linux or Mac OS-based device, or an embedded MCU with no operating system. UART transfer can be done with a third-party serial terminal program like Windows HyperTerminal or Linux "lrzsz" or with user-compiled host code. However, drivers for SPI Master or UART may vary with operating systems, and serial terminal programs may vary in timing and performance, so if you are unsure about what driver or program to use on your source code, please consult Silicon Labs technical support.

To open a serial connection over UART, the source device connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1), with no flow control by default. These options may be changed using the plugin options in the Bootloader AppBuilder project.

Note: The UART-based serial bootloader configurations do not employ any flow control in the communication channel by default, because the XModem protocol used for image transfer already has built-in flow control mechanisms. However, Silicon Labs' normally-supplied NCP firmware does utilize either hardware-based (RTS/CTS) or software-based (XON/XOFF) flow control, so a host device must take care to temporarily disable the flow control when placing its NCP into serial bootloading mode. Alternatively, application designers can change the options in the provided bootloader project and customize the serial bootloader's handling of the UART to add hardware flow control at their discretion.

Once the connection with a UART-based serial bootloader is established:

1. The target device's bootloader sends output over its serial port after it receives a carriage return from the source device at the expected baud rate. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port. Note that serial bootloaders typically don't enforce any timeout when awaiting the initial serial handshake via carriage return, so the bootloader will wait indefinitely in this mode until guided by the source device or until the chip is reset.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following ASCII-based output (where X.Y.A corresponds to the major, minor, and sub-minor fields of the bootloader version number, respectively):

```
Gecko Bootloader vX.Y.A
1. upload gbl
2. run
3. ebl info
BL >
```

Note: The third menu option has no effect. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new options might be added.

After listing the menu options, the bootloader's "BL >" prompt displays, and the ASCII character corresponding to the number of each option can then be entered by the source to select the described action, such as '2' (ASCII code 0x32) to run the firmware presently loaded in the application area. Here again, no timeout is enforced by the bootloader, so it will wait indefinitely until a character is received or the chip is reset. Note that while the menu interface is designed for human interaction, the transfer can still be performed programmatically or through a scripted interface, provided the source device sends the expected ASCII characters to the target at appropriate times.

Note: Scripts that interact with the bootloader should use only the "BL >" prompt to determine when the bootloader is ready for input.

Selecting menu option 1 initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of a GBL file over the expected serial interface, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. Once uploading begins (first XModem SOH data packet received), the bootloader expects each successive XModem SOH packet within 1 second, or else a timeout error will be generated and the session will abort.
4. After an image successfully uploads, the XModem transaction completes and the bootloader displays 'Serial upload complete' before redisplaying the menu.

2.2 Performing a Serial Upload - SPI

To open a serial connection over SPI, the source device must act as SPI Master using Mode 0 or Mode 2. It must also react to edge-triggered interrupts from the slave device using the same nHOST_INT logic and SPI framing as the EZSP-SPI protocol described in *AN711: EZSP-SPI Host Interfacing Guide*.

Once the SPI slave enters bootloader mode, which includes a reset sequence with host interrupt and Reset response frame similar to the reset sequence of a normal EZSP-SPI NCP, the bootloader sits in a Waiting state looking for SPI input in the form of bootloader packets (SPI bootloader frames with 0xFD SPI byte). The source device then must perform a bootloader Query transaction, which involves the source sending a Query packet and expecting a Response packet. Note that the first Query transaction yields a Query-Found result (status byte 0x1A), while the subsequent Query will yield the expected Response result (status byte 'R' or 0x52). For details, refer to the sample SPI bootloading process described in section [2.3 Sample EZSP-SPI Bootloader Transcript](#).

Once a query transaction has completed successfully, that is with expected Response frame, the transfer of data packets can begin. The transfer process follows standard XModem-CRC protocol, just like the UART-based serial bootloader uses, but with SPI framing similar to that used to encapsulate EZSP data frames. This SPI-based XModem adaptation adheres to the following rules, some of which may differ from the SPI protocol used by the normal EZSP NCP firmware:

- The 0xFD SPI byte and a length byte (for number of bytes to follow) prefix every command or response frame.
- The 0xA7 frame terminator byte concludes every SPI command or response frame. This byte is not included in the length count used in the length byte.
- The NCP operates as a SPI slave, so nSSEL must be asserted before each transaction.
- No EZSP frame control bytes are used in the SPI frame. Consequently, no sleep mode operation is supported by the target during the bootload.
- SPI timing (timeouts, signal transitions) are similar to EZSP. See *AN711: EZSP-SPI Host Interfacing Guide* for details.
- The nHOST_INT signal is asserted by the target to indicate a pending response.
- In place of the EZSP “callbacks” command, the host should use the bootloader’s Query packet to prompt the target to push the asynchronous response such as XModem ACK back to the host.
- Each SPI frame typically generates an initial, synchronous response from the target, such as a BLOCKOK status, and a follow-up, asynchronous response, which must be queried for by the host. For example, the EOT packet generates a synchronous response with FILEDONE status, then Query transaction yields XModem ACK with block number of lastBlock+1 before rebooting into new firmware.
- The host must wait for nHOST_INT to assert (become low) before querying for status, as the SPI bootloader is edge-triggered rather than level triggered. Thus, acting too fast at the host side can cause an edge transition to be missed and the bootloading state machines at the host and NCP to get out of synchronization, resulting in problems later on.
- SPI Status and SPI Version commands (SPI bytes 0x0A and 0x0B) are still supported.
- Prior to the first data block being processed, the SPI bus is polled at a rate of once per second.
- Once the data transmission begins (first block processed), the bootloader will wait up to 60 seconds for the next data packet, polling at 5-second intervals.
- If either of the timeouts above is exceeded, the bootloader signals a cancellation (CAN frame) and reboots, restarting the state machine.
- XModem data packets consist of:
 - The SOH byte (ASCII 0x01)
 - A 1-byte incrementing block number (beginning at 1 and wrapping back around from 255 to 0)
 - The block number’s complement
 - 128 bytes of data read directly from the GBL file being uploaded
 - A 16-bit CRC of the data bytes from that packet
- Each packet is followed by an XModem ACK or NAK from the target device (the NCP running the bootloader), which confirms or refutes the current data packet.
- If the target receives a duplicate block, it simply sends the ACK for that block again. If the target receives a block that had an XModem frame error (such as bad CRC), the bootloader expects that data block to be retransmitted and then the bootload can continue. Other kinds of errors are considered unrecoverable and cause the bootload to abort.
- If the bootload process aborts for any reason (including receiving an XModem Cancel (CAN) frame from the source), an XModem Cancel frame is echoed on the SPI interface from the target and the target then reboots, restarting the bootloader state machine.
- When the source device reaches the last XModem data block, it should be padded to 128 bytes of data using SUB (ASCII 0x1A) characters.

- Once the last block is ACKed by the target, the transfer should be finalized by an EOT (ASCII 0x04) packet from the source. Once this packet is confirmed via XModem ACK from the target, the device reboots, causing the new firmware to be launched.

Note: The ACK for the last XModem data packet may take much longer (1-3 seconds) to be received than prior data packets. This is due to the CRC32 checksum and optional GBL file signature verification being performed across the received GBL file data before sending the ACK. The source device must ensure that its SPI XModem state machine waits a sufficient amount of time to allow this checksum process to occur without timing out on the response just before the EOT is sent.

2.3 Sample EZSP-SPI Bootloader Transcript

The following is a record of the SPI frames transmitted and received by the EZSP-SPI host during the SPI bootload process, as captured from an EZSP Host application running the `ota-bootload-ncp-spi.c` state machine from the Silicon Labs Zigbee application framework's OTA Platform Bootloader plugin with an EZSP-SPI NCP device as target using the `ezsp-spi-bootloader`.

A "TX:" line means that the hexadecimal byte values contained in brackets ("[...]") are transmitted by the source via SPI to the target. An "RX:" line means that the byte values that follow in brackets are received by the source via SPI from the target NCP device.

Note: Firmware data (in the form of an EBL file for an EZSP-SPI NCP) is being streamed to the host's UART (for relaying down to the target) during this process, but that serial stream is not shown here.

Comments about the process are indicated in italics and are not part of the data transmitted on the SPI bus.

2.4 Errors and Status Codes

If an error occurs during the upload, the UART serial bootloader displays the message 'Serial upload aborted,' followed by a more detailed message and a hex error code. Some of the more common errors are shown in the following table. The UART serial bootloader then redisplay the bootloader menu. If an error occurs during the SPI serial bootload, the target produces an error response followed by an XModem Cancel frame and a reboot.

The following table describes the normal status codes, error conditions, and special characters or enumerations used by some or all of the Ember standalone bootloader variants as well as the Gecko Bootloader. For additional status codes specific to the Gecko Bootloader see the Gecko Bootloader API documentation installed with your SDK in the platform/bootloader/documentation folder.

Table 2.1. Serial Uploading Statuses, Error Messages, and Special Characters

| Hex code | Constant | Description |
|----------|-------------------------|---|
| 0x00 | BL_SUCCESS | Default success status. |
| 0x01 | BL_ERR | General error processing packet. |
| 0x1C | BLOCK_TIMEOUT | The bootloader timed out waiting for some part of the XModem frame. |
| 0x21 | BLOCKERR_SOH | The bootloader did not find the expected start of header (SOH) character at the beginning of the XModem frame. |
| 0x22 | BLOCKERR_CHK | The bootloader detected the sequence check byte of the XModem frame was not the inverse of the sequence byte. |
| 0x23 | BLOCKERR_CRCH | The bootloader encountered an error while comparing the high bytes of the received and calculated CRCx of the XModem frame. |
| 0x24 | BLOCKERR_CRCL | The bootloader encountered an error while comparing the low bytes of the received and calculated CRCs of the XModem frame. |
| 0x25 | BLOCKERR_SEQUENCE | The bootloader did not receive the expected sequence number in the current XModem frame. |
| 0x26 | BLOCKERR_PARTIAL | The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost). |
| 0x27 | BLOCKERR_DUPLICATE | The bootloader encountered a duplicate of the previous XModem frame. |
| 0x40 | BL_ERR_MASK | Bitmask for any bootloader error codes returned in CAN or NAK frame. |
| 0x41 | BL_ERR_HEADER_EXP | No GBL header was received when expected. |
| 0x42 | BL_ERR_HEADER_WRITE_CRC | Failed to write header or CRC. |
| 0x43 | BL_ERR_CRC | File or written image failed CRC check. |
| 0x44 | BL_ERR_UNKNOWN_TAG | Unknown tag detected in GBL image. |
| 0x45 | BL_ERR_SIG | Invalid GBL header contents. |
| 0x46 | BL_ERR_ODD_LEN | Trying to flash odd number of bytes. |
| 0x47 | BL_ERR_BLOCK_INDEX | Indexed past end of block buffer. |
| 0x48 | BL_ERR_OVWR_BL | Attempt to overwrite bootloader flash. |
| 0x49 | BL_ERR_OVWR_SIMEE | Attempt to overwrite SIMEE flash. |
| 0x4A | BL_ERR_ERASE_FAIL | Flash erase failed. |
| 0x4B | BL_ERR_WRITE_FAIL | Flash write failed. |

| Hex code | Constant | Description |
|--|-----------------|---|
| 0x4C | BL_ERR_CRC_LEN | End tag CRC wrong length. |
| 0x4D | BL_ERR_NO_QUERY | Received data before query request/response. |
| 0x4E | BL_ERR_BAD_LEN | An invalid length was detected in the upgrade image file. |
| 0x4F | BL_ERR_TAGBUF | Insufficient tag buffer size or an invalid length was found in the GBL image. |
| Special Characters Used in Packet Types | | |
| 0x01 | SOH | Start of Header. |
| 0x03 | CTRL_C | Cancel (from sender). |
| 0x04 | EOT | End of Transmission. |
| 0x06 | ACK | Acknowledged. |
| 0x15 | NAK | Not acknowledged. |
| 0x18 | CAN | Cancel |
| 0x43 | C | ASCII 'C'. |
| 0x51 | QUERY | ASCII 'Q'. |
| 0x52 | QRESP | ASCII 'R'. |
| Status Codes Returned in a Synchronous Response | | |
| 0x16 | TIMEOUT | Bootloader timed out expecting characters. |
| 0x17 | FILEDONE | EOT process successfully. |
| 0x18 | FILEABORT | Transfer aborted prematurely. |
| 0x19 | BLOCKOK | Data block processed OK. |
| 0x1A | QUERYFOUND | Successful query. |

2.5 Running the Application Image

For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu. For SPI-based variants, which don't use a menu, the application is run immediately upon ACKing the EOT frame from the source device.

2.6 Performing a Bootloader Upgrade

Bootloader upgrade functionality is provided by the first stage bootloader on Series 1 devices, or the Secure Element on Series 2 devices. On Series 2 devices, the Secure Element itself is also upgradable. On Series 1 devices, the first stage bootloader is not upgradable. For more information on how to upgrade the secure element, see *UG266: Silicon Labs Gecko Bootloader User's Guide*.

To perform a bootloader upgrade with the standalone bootloaders, simply transmit two GBL files in succession. The first GBL file should contain a main bootloader upgrade image. After upload is completed, the device resets to upgrade the main bootloader. For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the first stage bootloader to perform the upgrade, before returning to the upgraded main bootloader. The second GBL file, containing an application upgrade image, can then be uploaded.

For SPI-based variants, which don't use a menu, the upgraded bootloader is run immediately upon ACKing the EOT frame from the source device. Once the bootloader upgrade has been applied, the SPI Host loses connection to the device. An EmberZNet Host application will not be able to re-connect with the device with only a bootloader in place.

2.7 Upload Recovery

If an image upload fails, the target node is left without a valid application image. The standalone bootloader will re-enter firmware upgrade mode if it determines that the application image isn't valid. However, the application image may have a valid structure, but contain a bug preventing normal operation. Regardless of the serial interface supported by your standalone bootloader, a GPIO-based trigger can be used to facilitate recovery via serial upload.

You can configure your standalone bootloader to use a software-based GPIO pin check or other schemes of recovery mode activation such as button recovery by configuring the EZSP GPIO Activation plugin in the Application Builder framework, or by adding custom code to the bootloader project.

3. Using the Gecko Application Bootloader

3.1 Acquiring a New Image

The application bootloader relies on application code to obtain new code images. The bootloader itself only knows how to read a GBL image stored in the download space and copy the relevant portions to the main flash block. This approach means that the application developer is free to acquire the new code image in any way that makes sense (serial, OTA, and so on).

Typically application developers choose to acquire the new code image over-the-air (OTA) since this is readily available on all devices. For OTA bootloading in zigbee networks, Silicon Labs recommends using the standard OTA Upgrade cluster defined in the Zigbee Cluster Library (ZCL). Code for this cluster is available in the Silicon Labs Zigbee application framework as several different plugins. *AN728: Zigbee Over-the-Air Bootload Cluster Server and Client Setup* walks through how this can be set up and run. For OTA bootloading in non-zigbee networks where a ZCL-based application layer is not available, the application layer may define its own means of conveying firmware data over the networking protocol, or the application developer may define a proprietary means of accomplishing an OTA image transfer between a source device and a target.

For customers who want to design their own application to acquire an image rather than using the the Silicon Labs Zigbee application framework's plugins, we provide some routines for interfacing with the download space. These routines allow you to get information about the storage device and interact with it. You can find the code and documentation for these routines in the source files **bootloader-interface-app.c** and **bootloader-interface-app.h** in the `platform/base/hal/micro/cortexm3/efm32` directory. If you do want to call these routines directly, it may be helpful to look at how the **OTA Cluster Platform Bootloader** plugin code works to ensure that these routines are used correctly. (See related files in the `app/framework/plugin/ota-bootload` directory of the EmberZNet PRO installation for more information.)

3.2 Performing an Application Upgrade

The application and the bootloader have limited indirect contact. Their only interaction is through passing non-volatile data across module reboots.

Once the application decides to install a new image saved in the download space it calls the Ember HAL API `halAppBootloaderInstallNewImage()`. This call indicates to the bootloader that it should attempt to perform a firmware upgrade from storage slot 0, and reboots the device. This API is backwards-compatible with the legacy Ember bootloader. If performing a firmware upgrade from a different storage slot than slot 0 is desired, the Gecko Bootloader API should be used instead, by calling `bootloader_setBootloadList()`. If the bootloader fails to install the new image, it sets the reset cause to `RESET_BOOTLOADER_BADIMAGE` and resets the module. Upon startup, the application should read the reset cause with `halGetExtendedResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_BADIMAGE`, the application knows the install process failed and can attempt to obtain a new image. A printable error string can be acquired from calling `halGetExtendedResetString()`. Under normal circumstances, the application bootloader does not print anything on the serial line.

3.3 Example Application Bootload

For details on how to set up and run an application bootload, see *AN728: Over-the-Air Bootload Server and Client Setup*.

3.4 External Storage Application Bootloader

See *UG266: Silicon Labs Gecko Bootloader User's Guide* for information about memory configuration for external storage bootloaders. Note that, at the time of this writing, the OTA Simple Storage Module plugin code in the EmberZNet stack only supports storage configurations with a single storage slot. The start address of the storage slot in the bootloader needs to be configured to the same value as the OTA Storage Start Offset of the OTA Simple Storage EEPROM Driver plugin in the Silicon Labs Zigbee application framework. The application developer is responsible for ensuring that this value is consistent across the stack and bootloader projects. Also keep in mind that the start address of the first storage slot in the bootloader should be offset from the beginning of the storage area by two times the page size of the underlying storage medium if more than one storage slot is configured.

Application bootloaders typically use a remote device to store the downloaded application image. This device can be accessed over either an I2C or SPI serial interface. Refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* for a list of supported Dataflash/EEPROM devices. It is important to select a device whose size is at least the size of your flash in order to fit the application being bootloaded.

The default recommendation for external SPI serial flash is the MX25R, as this is available in standard and smaller packages, is supported by standard drivers, and has very low software-enabled sleep current without the need for an external shutdown control circuit. Most radio boards from Silicon Labs are populated with the MX25R8035F for evaluation purposes. In general, customers should use parts that have low sleep current, don't require external shutdown control circuitry, and have software shutdown control. When using components with high idle/sleep current and no software shutdown control, an external shutdown control circuit is recommended to reduce sleep current.

For EFR32MG-based devices using the EFR32MG1, only the serial dataflash option is available; no local storage application bootloader is presently offered. However, some EFR32MG ICs contain an integrated serial flash that can be utilized just like off-chip serial dataflash but without any additional components. EFR32MG-based devices using the EFR32MG12 and higher platforms support local storage application bootloaders.

Note that some of these chips have compatible pinouts with others, but there are several incompatible variations. Contact Silicon Labs for details on connecting I2C or other SPI dataflash chips to an EFR32.

Read-Modify-Write pertains to a feature of certain dataflash chips that their corresponding driver exposes, and that is exploited by the bootloader library. Chips without this feature require a page erase to be performed before writing to that page, which precludes random-access writes by an application. When using the Silicon Labs Zigbee application framework, the **OTA Simple Storage EEPROM Driver** plugin needs to be configured to take this into consideration.

3.5 Local Storage Application Bootloader

The local storage bootloader is essentially an application bootloader with a data flash driver that uses a portion of the on-chip flash for image storage instead of an external storage chip. See *UG266: Gecko Bootloader User's Guide* for information about memory configuration for internal storage. Note that the OTA Simple Storage Module plugin code in the EmberZNet stack only supports storage configurations with a single storage slot at the time of this writing. The start address of the storage slot in the bootloader needs to be configured to the same value as the OTA Storage Start Offset of the OTA Simple Storage EEPROM plugin in the Silicon Labs Zigbee application framework. The application developer is responsible for ensuring that this value is consistent across the stack and bootloader projects. Also keep in mind that the start address of the first storage slot in the bootloader should be offset from the beginning of the storage area by two times the page size of the underlying storage medium if more than one storage slot is configured.

Since the local storage application bootloader changes the chip's flash memory layout you must build your application with knowledge of this. To accomplish this, you must add the LOCAL_STORAGE_BTL global define to your IAR project file. If you're creating your project file through AppBuilder in Simplicity Studio, then this will be done for you as long as you select Local Storage from the bootloader dropdown.

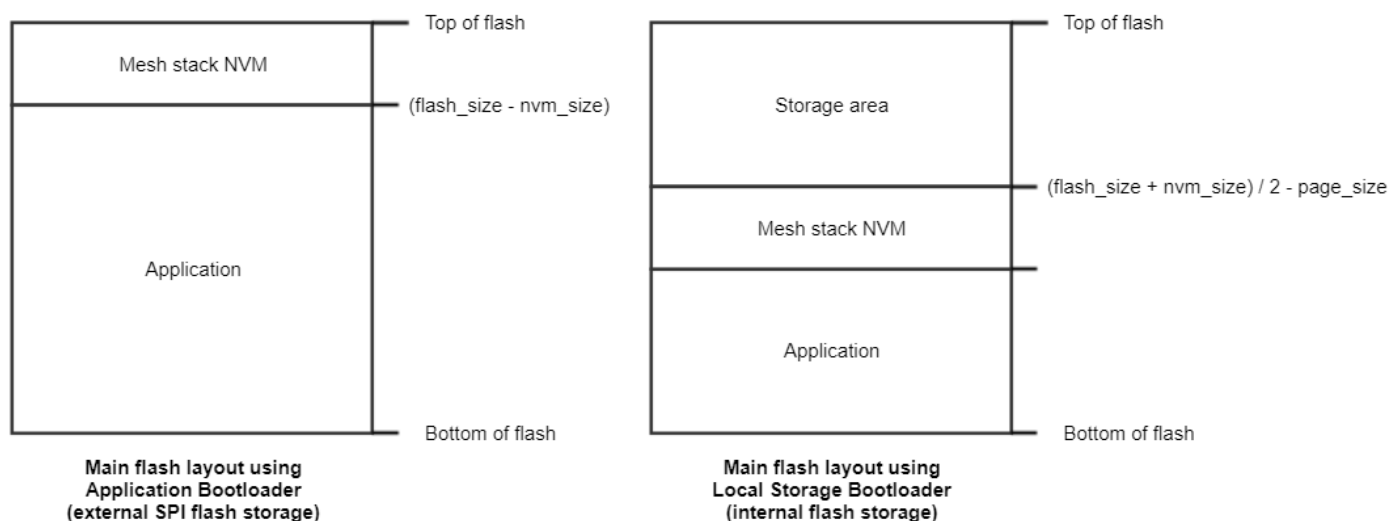
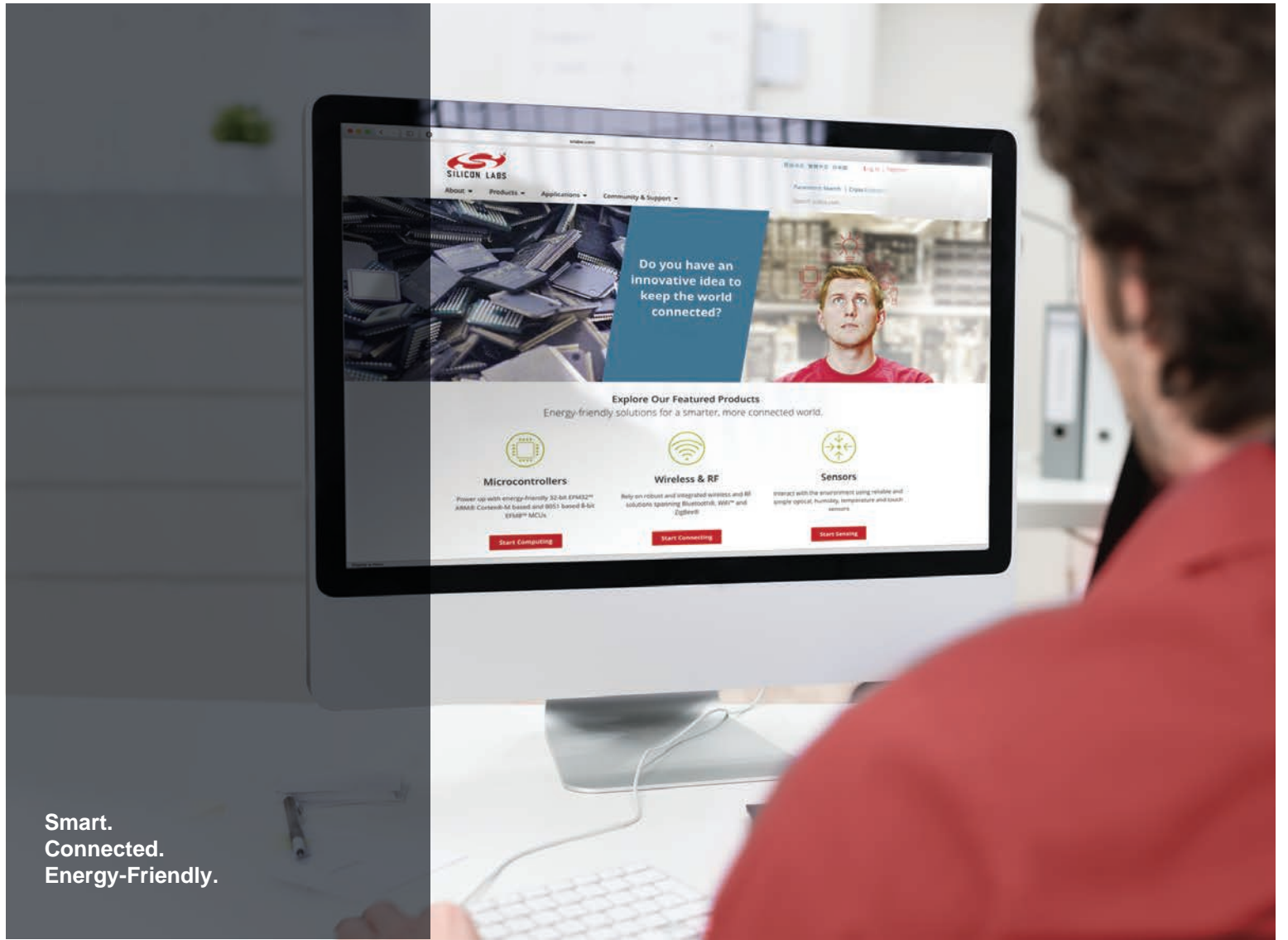


Figure 3.1. Main Flash Layout for Application Bootloader versus Local Storage Application Bootloader when Using Gecko Bootloader

Note: On EFR32MG1 and EFR32xG21 (Series 2) devices, the application is offset by 16 kB to accommodate the bootloader at the bottom of main flash. On other Series 1 EFR32 devices, the Gecko Bootloader resides in the Bootloader flash region outside of main flash block.



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 to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems 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. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

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®, Gecko OS, Gecko OS Studio, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, the Zentri logo and Zentri DMS, 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. Wi-Fi is a registered trademark of the Wi-Fi Alliance. 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>