AN772: Using the Application Bootloader

This document describes some of the specific types of Application Bootloaders released by Silicon Labs. For general information about the Application Bootloader and how it compares with the Standalone Bootloaders, see *UG103.6, Fundamentals: Bootloading.*

KEY FEATURES

- Describes the principles of the Application Bootloader model compared to the Standalone Bootloader model.
- Compares and contrasts the available Application Bootloader variants.
- Lists supported drivers and model numbers for external storage options.

1 Application Bootloading

1.1 Introduction

The application bootloader is supported on all EM3x and Mighty Gecko (EFR32MG) System on Chip (SoC) devices and has the single purpose of reprogramming the flash with an application image stored in some download space. Typically this download space is an external memory device like a dataflash or EEPROM, but, on SoC devices with larger memory capacity, you can store the image in a portion of the SoC's internal flash memory. The image must be in the EBL format and stored starting at the logical beginning of the download space. Silicon Labs does not currently support bootloading from multiple images stored in the download space though it is possible for the application to store as many images as will fit.

Since acquiring the image is the responsibility of the application, this bootloader is simpler and more flexible than the standalone bootloader. The application is free to upload the new image to the download space any way it wants. The image could come over-the-air from multiple hops away, over a physical connection like the serial port or USB, or anything else the application developer can think of. The application is also free to take as much time as necessary to upload the new image in the background and only program it once ready. The stored image could even be in an encrypted form if the secure application bootloader is used.

Since the new image is stored externally, the current application isn't overwritten until the new image has been successfully saved and verified. If something goes wrong during the update process, the bootloader will automatically try to bootload the image again when restarted. In the unlikely event that both the application image and the image in the download space become corrupted, a failsafe recovery mode can be entered by all of our application bootloaders.

The main downside of an application bootloader is that it requires you to allocate some storage space for the download region. In the case of the local storage bootloader, this uses up some quantity of flash that could otherwise be used for your application. In all other application bootloaders, you're required to add an external data storage part to your design, which increases the cost.

The application bootloader can only be used in conjunction with SoC devices. Network Coprocessors (NCPs) only support standalone bootloaders.

1.2 Types of Application Bootloaders

All application bootloaders are very closely related to each other, but they are generally organized along two dimensions: external versus local storage, and unencrypted versus secure (encrypted).

The original application bootloader developed by Silicon Labs was an unencrypted external storage bootloader. It expected the download space to be an external memory device allowing only unencrypted Ember bootloader (EBL) files. Silicon Labs has since added support for encrypted EBLs via a secure bootloader variant. For devices containing at least 512 kB of internal flash, Silicon Labs added the unencrypted local storage and secure local storage types, eliminating the need for an external storage device.

Note: Secure bootloader support and local storage bootloader support are not yet available for the EFR32MG family but are planned for support in a future release.

Bootloaders may also differ in how the recovery mechanism is presented. For most application bootloaders, recovery mode involves a simple XModem serial transfer protocol. However, chips with USB capability can run an application bootloader where recovery mode causes the chip to appear as a mass storage device to PCs. Mass Storage Device (MSD) application bootloaders are available in both external storage and local storage variants.

1.3 Acquiring a New Image

As mentioned above, the application bootloader relies on application code or the recovery mode to obtain new code images. The bootloader itself only knows how to read an EBL 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 for them (serial, OTA, etc.).

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 Application Framework V2 as several different plugins. AN728, *Over-the-air Bootload Server and Client Setup Using EM35x Development Kits*, walks through how this can be set up and run using Ember EM35x development kits. These documents can be used as a reference point for implementing application bootloading with different hardware. (Additional documentation for demonstrating this with the Mighty Gecko family and the Wireless Starter Kit (WSTK) is expected to appear in a future release.) 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 our application framework 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`. 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 installation for more information.)

1.4 Recovery Mode

Recovery mode is used as a failsafe mechanism to recover a module that has no valid application image. It is invoked by the application bootloader when both the application image in the main flash block and the EBL image in the download space are invalid. It can also be entered manually. Recovery mode uses a serial connection to download a new EBL file into the download space.

1.4.1 Manually Entering Recovery Mode

There are two ways to manually enter recovery mode. The first method uses the FIB Monitor Mode, which is only available on EM3x-based devices. To perform this method, first start the EM3x device in FIB Monitor mode by grounding GPIO pin PA5 (nBOOTMODE) while resetting the chip. Then send a carriage return at 115200 baud over the UART serial port (SC1). This will cause the FIB Monitor code to pass control to the bootloader installed in the main flash block and will ensure that the bootloader is started in its recovery mode rather than allowing the application to boot normally.

The second option requires you to rebuild your application bootloader from the provided IAR Embedded Workbench project file, but it allows you to choose any GPIO or specify your own scheme for determining when to enter the recovery mode. To use this, you must modify the `bootloadForceActivation()` routine in `bootloader-gpio.c` (found in the `hal/micro/cortexm3/{mcuFamily}/bootloader` subdirectory of your stack installation) and rebuild the bootloader. An example utilizing PC6 is provided in `bootloader-gpio.c` and can be enabled by building the bootloader with USE_BUTTON_RECOVERY defined.

1.4.2 XModem Recovery Mode

Recovery mode uses the XModem protocol with CRC to upload a new image over the serial line. Once activated it immediately starts the upload sequence by sending 'C' characters out the serial line. The SC1 serial controller is used as a UART at 115200 baud, 8 bits, no parity, 1 stop bit. The 'C' characters are sent every 1 second until an XModem upload sequence is detected.

Tip: If you need to modify the UART settings used by the XModem Recovery Mode or move the serial functions to a different serial controller (if supported by the SoC), you can edit the `bootloader-uart.c` file found in the `hal/micro/cortexm3/{mcuFamily}/bootloader` subdirectory of your stack installation.

Use a terminal emulator on a PC to send the application EBL file to the node connected by serial cable.

Once the image has been saved to the download space, recovery mode resets the module and attempts to bootload the image just stored there.

1.4.3 Mass Storage Device (MSD) Recovery Mode

The MSD application bootloader available for EM358x and EM359x chips with USB replaces the aforementioned recovery mode with an MSD recovery mode. When the device enters MSD recovery mode, the download space will be formatted as a blank FAT12 volume that will enumerate as a USB mass storage device. The size of the volume is dictated by the size of the storage device (external or local), and any stored data will be erased during the initialization of recovery mode. An upgrade EBL file may be uploaded to the device by copying the file to the empty drive. To trigger a reset/bootload, disconnect USB and the module will attempt to locate and bootload the transferred image. The MSD recovery mode is currently only available for the unencrypted application bootloader.

1.5 Errors during Application Bootloading

1.5.1 Application Bootloader Errors

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 `halAppBootloaderInstallNewImage()`. This call sets the bootload mode and reboots the module. 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.

1.5.2 Recovery Mode Errors

If recovery mode encounters an error while uploading an image, the chip will cancel the upload and reset. In XModem recovery mode, it also prints "Err" or "Stat" on the serial line followed by the error or status number in hex. After reset, recovery mode may be entered again if no valid images can be found or `bootloadForceActivation()` returns `true`. Table 1 lists some possible recovery mode bootload errors.

Table 1. Bootload Errors

Error/Status	Description
16	Timeout: Exceeded 60 seconds serial download timeout
18	File abort: Control-C on console
83	Write check error: Data read from external storage does not match data written
84	Image Size error: Download image size is greater than external storage space available

1.6 Example Application Bootload

For details on how to setup and run an application bootload using EM35x development kits, please refer to the application note *AN728, Over-the-air Bootload Server and Client Setup using EM35x Development Kits*. (Additional documentation for demonstrating this with the Mighty Gecko family and the Wireless Starter Kit (WSTK) is expected to appear in a future release.)

2 External Storage Application Bootloader

2.1 Memory Map

The application bootloaders use the same memory map as the standalone bootloaders, but with an external storage device that is addressed separately. A diagram of the memory layout is shown in Figure 1.

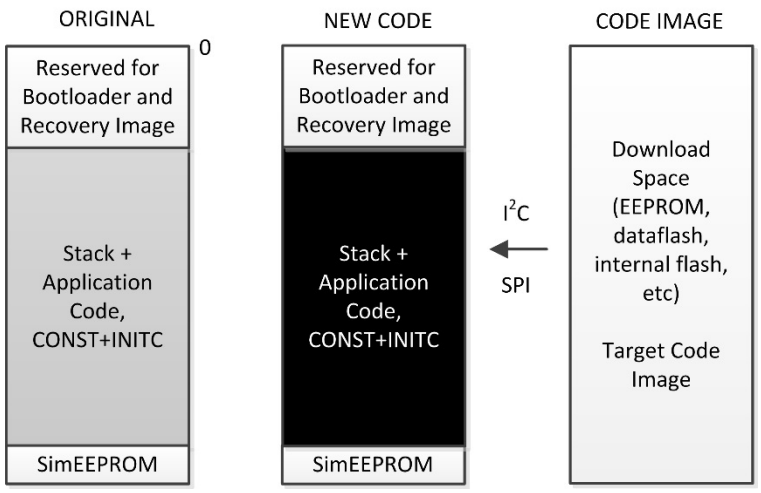

Figure 1. Application Bootloading Code Space (Typical)

2.2 Remote Memory Connection

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 Table 2 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.

Table 2. Supported Serial Dataflash/EEPROM External Memory Parts

Manufacturer Part Number	Ifc	Support on	Size	Read-Modify-Write?	Pin-compatible with	Driver
Microchip MC24AA1025	I²C	EM3x	128 kB	Yes	Microchip	mc24aa1025.c
ST Microelectronics M24M02-DR	I²C	EM3x	256 kB	Yes	ST Micro	stm24m02.c
Atmel AT45DB021D/E	SPI	EM3x	256 kB	Yes	Atmel	at45db021d.c
Micron (Numonyx) M45PE20-VMN6TP	SPI	EM3x	256 kB	Yes	Atmel	m45pe20.c
Macronix MX25L2006EM1I-12G (MX25L2006EM1R-12G for high-temperature support)	SPI	EM3x/EFR32	256 kB	No	Winbond	spiflash-class1.c
Macronix MX25L4006E	SPI	EM3x/EFR32	512 kB	No	Winbond	spiflash-class1.c
Macronix MX25L8006EM1I-12G (MX25L8006EM1R-12G for high-temperature support)	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Winbond W25X20BVSNIG (W25X20CVSNJG for high-temperature support)	SPI	EM3x/EFR32	256 kB	No	Winbond	spiflash-class1.c

Manufacturer Part Number	Ifc	Support on	Size	Read-Modify-Write?	Pin-compatible with	Driver
Winbond W25Q80BVSNIG (W25Q80BVSNJG for high-temperature support)	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Spansion S25FL208K	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Macronix MX25R8035F	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Macronix MX25L1606E	SPI	EM3x/EFR32	2048 kB	No	Winbond	spiflash-class1.c
Macornix MX25U1635E	SPI	EM3x/EFR32	2048 kB	No	Winbond	spiflash-class1.c
Macronix MX25R6435F	SPI	EM3x/EFR32	8192 B	No	Winbond	spiflash-class1.c
Atmel AT25DF041A	SPI	EM3x/EFR32	512 kB	No	Winbond	spiflash-class1.c
Atmel AT25DF081A	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Micron (Numonyx) M25P20	SPI	EM3x/EFR32	256 kB	No	Winbond	spiflash-class1.c
Micron (Numonyx) M25P40	SPI	EM3x/EFR32	512 kB	No	Winbond	spiflash-class1.c
Micron (Numonyx) M25P80	SPI	EM3x/EFR32	1024 kB	No	Winbond	spiflash-class1.c
Micron (Numonyx) M25P16	SPI	EM3x/EFR32	2048 kB	No	Winbond	spiflash-class1.c
ISSI IS25LQ025B	SPI	EM3x/EFR32	32 kB	No	Winbond	spiflash-class1.c
ISSI IS25LQ512B	SPI	EM3x/EFR32	64 kB	No	Winbond	spiflash-class1.c
ISSI IS25LQ010B	SPI	EM3x/EFR32	126 kB	No	Winbond	spiflash-class1.c
ISSI IS25LQ020B	SPI	EM3x/EFR32	256 kB	No	Winbond	spiflash-class1.c
ISSI IS25LQ040B	SPI	EM3x/EFR32	512 kB	No	Winbond	spiflash-class1.c

Note that some of these chips have compatible pinouts with others, but there are several incompatible variations. A schematic for connecting the Atmel AT45DB021D SPI-based dataflash chip to an EM35x can be found in the document TS6, the *EM35x Breakout Board Technical Specification*. Contact Silicon Labs for details on connecting I2C or other SPI dataflash chips to an EM3x or 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 Application Framework V2, the **OTA Simple Storage EEPROM Driver** plugin needs to be configured to take this into consideration.

Application bootloader images are supplied in S-record format (*.s37) for both I2C and SPI versions of the drivers noted in the table. In addition to these prebuilt images, you can build the bootloader image yourself using the supplied IAR Embedded Workbench project file found under `tool/bootloader-{part number}/{bootloader variant}/{bootloader variant}.eww`, such as `tool/bootloader-efr32mg1p132f256gm48/app-bootloader-spiflash/app-bootloader-spiflash.eww`, which uses the

spiflash-class1.c driver by default. By modifying the project file and rebuilding the bootloader, you can do things like use a custom remote storage driver or set up your own default GPIO configuration.

Users may want to rebuild the bootloader image to turn on Serial Flash Hardware Shutdown Control, for instance. If your hardware implements this power-saving feature, then the EEPROM_USES_SHUTDOWN_CONTROL define symbol must be added or uncommented in the Board Header file when building the application bootloader, regardless of which dataflash chip is being used. Also note that, because the power-on state of these dataflash chips is "standby," an application that always wants to minimize its current draw should initialize and then immediately shutdown the dataflash as part of its bootup procedure.

On EM3x devices with multiple available serial controllers, the serial controller for the external memory device may be selected using the EXTERNAL_FLASH_SERIAL_CONTROLLER define symbol. If undefined, the driver may default to SC2 unless SPI_FLASH_SC1 is defined. Furthermore, the serial rate may be defined with the EXTERNAL_FLASH_RATE_LINEAR (LIN) and EXTERNAL_FLASH_RATE_EXPONENTIAL (EXP) define symbols according to the following equation:

$$rate = \frac{PCLK}{(LIN + 1) * 2\text{^}EXP}$$

Where PCLK is the SYSCLK divided by two (generally 12 or 6 MHz). If left unspecified, the I2C driver will default to 400 kbps in master mode, and the SPI driver will default to 12 MHz, master mode, MSB transmitted first, sample on leading edge, rising leading edge.

On EFR32, the USART is fixed to be USART1, but the pins on which the signals come out can be customized using macros such as `EXTERNAL_FLASH_USART0_TXLOC`, `EXTERNAL_FLASH_MOSI_PORT`, and `EXTERNAL_FLASH_MOSI_PIN`. See `hal/micro/cortexm3/efm32/bootloader/spiflash-low-level.c` for more details.

3 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. This simplifies a customer's design, but also means that there is less storage space for the application and the SimEEPROM needed for application token storage during operation. Since a new application image has to fit in this storage region it needs to be roughly half of the chip's flash which means that this type of bootloader is only feasible on chips with enough flash like some EM358x and EM359x variants.

3.1 Memory Map

Since the storage region for the local storage application bootloader is in the internal flash memory map, this bootloader type shrinks the application area and adjusts the location of the SimEEPROM. A diagram of this memory layout is shown in Figure 2.

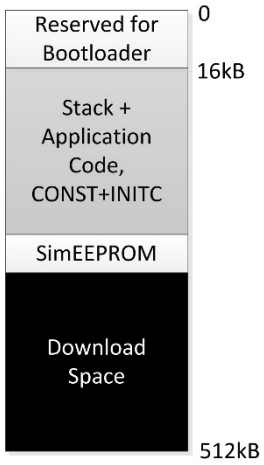

Figure 2. Local Storage Bootloader Memory Map

The numbers in the above diagram assume the standard 8kB SimEEPROM. If you select a larger SimEEPROM size, there are some changes in the size of the download space since the maximum size for your application is decreased. (See AN703: *Simulated EEPROM* for further information about SimEEPROM and the memory requirements of different SimEEPROM models.) The download space size is chosen automatically to be half of the memory available to your application with some additional space added for EBL format overhead. Specifically, we use the following formula with overhead set equal to 2kB.

```
DownloadSpace_size = (Flash_size – SimEEPROM_size – 16kB)/2 + Overhead
```

If you require more fine grained control over the size of the download region, please contact Silicon Labs support for more information.

3.2 Building your Application

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 Bootloader from the bootloader dropdown.

If you do not do this, your application will be built with a storage region size of 0 byes by default. An image built like this will not be deemed valid by the local storage application bootloader because it would result in a dangerous situation where your device could never be updated again.

4 Secure Bootloaders

Most application bootloaders can have secure variants designed to only accept encrypted EBL files. Encrypted EBL files are secured using a symmetric key that is stored among the device's manufacturing tokens in an area of the chip that can be secured (via read protection) and kept secret. This allows the device to verify that the image comes from someone who knows the key and not just anyone who knows how to create an EBL file. This also ensures that your application image cannot be read by anyone who does not have the symmetric key.

Note: Secure bootloader support and local storage bootloader support are not yet available for the EFR32MG family of parts but are planned for support in a future release.

4.1 Security Considerations

The secure bootloader uses the encrypted EBL file format to protect the data both in transit and while stored on the device. This format uses AES-128 operating in CCM* mode to encrypt the contents of the file. For more information about this file format, see section 3.2 of document UG103.6, *Fundamentals: Bootloading.*

Since we're using symmetric key encryption, the key must be stored on both the device and the machine that you will use to generate encrypted EBLs. Creating the key(s) and loading them on the device must be done in order for the secure bootloader to work. Developers may choose to have one key per device, one key for all devices, or any other partitioning. If you choose to have different keys for some devices, you will have to generate different encrypted EBL files for these devices and ensure that the correct EBL is sent to the correct device. Whatever you do, the key files must be kept safe on the machine generating the encrypted EBLs. Once a key file is released an attacker can encrypt and sign images for any device that uses that key file.

The key must also be kept safe on the devices in the field. The security key for the device is stored in the protected manufacturing token area (CIB region for EM3x-based devices and Lockbits region for EFR32-based devices) at a known location. To prevent an attacker from accessing this, Silicon Labs recommends turning on the Flash Readout Protection feature of the chip. This will prevent a debugger from being able to connect to the chip and read out the protected manufacturing token data. It's also important to make sure that your application is robust against software attacks that could attempt to read out this key. Keep in mind that once this protection is enabled there is no easy way to modify the manufacturing token to change the security key without erasing the entire device.

4.2 Using Simplicity Commander

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

- Creating encrypted EBL images
- Generating a keyfile
- Encrypting an EBL image
- Decrypting an EBL file

For more information on executing the commands to complete these tasks, refer to UG162, *Simplicity Commander Reference Guide*.