AN772

USING THE APPLICATION BOOTLOADER

This application note 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.*

New in This Revision
Initial release of this document.

Contents

1 Application Bootloading

1.1 Introduction

The application bootloader 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 the larger EM358x series of chips you can store the image on the internal device flash. The image must be in the EBL format and stored starting at the 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.

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, there is also a simple serial recovery mode that 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.

1.2 Types of Application Bootloaders

All application bootloaders are very closely related to each other, but we’ve organized them along two dimensions: external vs. local storage, and unencrypted vs. 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 EBL files. Starting with EmberZNet 5.0 Silicon Labs has added support for encrypted EBLs on the EM351, EM357, and EM358x parts using a secure bootloader.

For the EM358x, which have more flash space available, Silicon Labs has added the unencrypted local storage and secure local storagetypes eliminating the need for an external storage device.

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, Silicon Labs recommends using the standard ZigBee OTA bootload cluster. Code for this cluster is available in the Ember application framework as several different plugins. These plugins are documented in Chapter 15 of document UG102, *Application Framework Developer Guide*. Additionally, document AN728, *ZigBee Over-the-air Bootload Cluster 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.

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.

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 Recover Mode

There are two ways to manually enter recovery mode. The first method is to ground the PA5 GPIO, reset the chip, and then send a carriage return at 115200 baud over the UART serial port (SC1). This will work on any EM35x chip. The second option requires you to rebuild your application bootloader but allows you to choose any GPIO or even a more complicated scheme for determining when to enter the recovery mode. To use this, you must modify the `bootloadForceActivation()` routine in `bootloader-gpio.c` and rebuild the bootloader. An example of utilizing PC6, which is connected to a button on the EM35x Breakout Board, is provided in `bootloader-gpio.c` and can be enabled by building the bootloader with `USE_BUTTON_RECOVERY` defined.

1.4.2 Functionality in 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.

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.

1.5 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_IMG_BAD` and resets the module. Upon startup, the application should read the reset cause with `halGetResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_IMG_BAD`, the application knows the install process failed and can attempt to obtain a new image. A printable error string can be acquired from calling `halGetResetString()`. The application bootloader does not print anything on the serial line. (It doesn't even configure SC1, except in recovery mode).

1.5.2 Recovery Mode Errors

If recovery mode encounters an error while uploading an image, it prints "Err" or "Stat" on the serial line followed by the error or status number in hex. It then restarts the upload process, emitting 'C' characters while waiting for a new upload to start. Table 6 lists 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 ZigBee Over-the-air Bootload Cluster Server and Client Setup using EM35x Development Kits*.

2 External Storage Application Bootloader

2.1 Memory Map

The EM35x uses the same memory map for the application bootloader as it does for the standalone bootloader, but with an external storage device that is addressed separately. A diagram of the memory layout is shown in Figure 10.

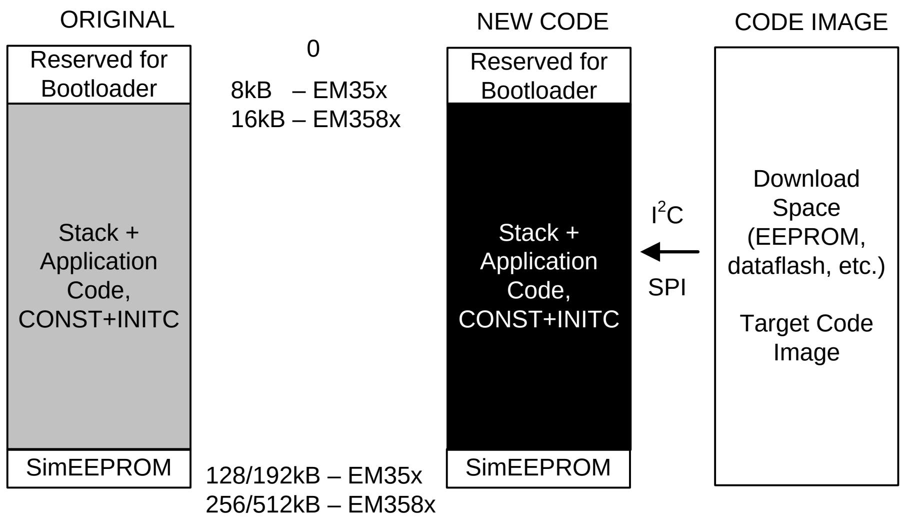

Figure 1. Application Bootloading Code Space (Typical)

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 5 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 Remote Memory Parts

Manufacturer Part Number	Ifc	Support on	Size	Temp Range (°C)	Read-Modify-Write?	Pin-compatible with	Driver
Microchip MC24AA1025	I2C	EM250 EM351	128 kB	-40 to +85	Yes	Microchip	mc24aa1025.c
ST Microelectronics M24M02-DR	I2C	EM35x	256 kB	-40 to +85	Yes	ST Micro	stm24m02.c
Atmel AT45DB021D/E	SPI	EM250 EM35x	256 kB	-40 to +85	Yes	Atmel	at45db021d.c
Micron (Numonyx) M45PE20-VMN6TP	SPI	EM35x	256 kB	-40 to +85	Yes	Atmel	m45pe20.c
Macronix MX25L2006EM1I-12G	SPI	EM35x	256 kB	-40 to +85	No	Winbond	spiflash-class1.c

Manufacturer Part Number	Ifc	Support on	Size	Temp Range (°C)	Read-Modify-Write?	Pin-compatible with	Driver
Macronix MX25L2006EM1R-12G	SPI	EM35x	256 kB	-40 to +105	No	Winbond	spiflash-class1.c
Macronix MX25L8006EM1I-12G	SPI	EM35x	1024 kB	-40 to +85	No	Winbond	spiflash-class1.c
Macronix MX25L8006EM1R-12G	SPI	EM35x	1024 kB	-40 to +105	No	Winbond	spiflash-class1.c
Winbond W25X20BVSNIG	SPI	EM35x	256 kB	-40 to +85	No	Winbond	spiflash-class1.c
Winbond W25X20CVSNJG	SPI	EM35x	256 kB	-40 to +105	No	Winbond	spiflash-class1.c
Winbond W25Q80BVSNIG	SPI	EM35x	1024 kB	-40 to +85	No	Winbond	spiflash-class1.c
Winbond W25Q80BVSNJG	SPI	EM35x	1024 kB	-40 to +105	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 the EM35x.

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 Ember application framework, 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 the EM35x, 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 `em35x/tool/bootloader-{`*`part number`*`}/app-bootloader/app-bootloader.eww`). 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 will need to rebuild the bootloader image when removing the Serial Flash Hardware Shutdown Control circuit from Ember reference designs. If your hardware does not implement this power-saving feature, then the EEPROM_USES_SHUTDOWN_CONTROL define symbol must be removed when building the application bootloader, regardless of which dataflash chip is being used. Comment out the definition of this symbol in your board header file. 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.

Ember I2C drivers configure the I2C bus to use the SC2 Serial Controller at 400 kbps in master mode.

Ember SPI drivers configure the SPI bus to use the SC2 Serial Controller with the following settings: 12 MHz, master mode, MSB transmitted first, sample on leading edge, rising leading edge.

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 the EM358x 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 10.

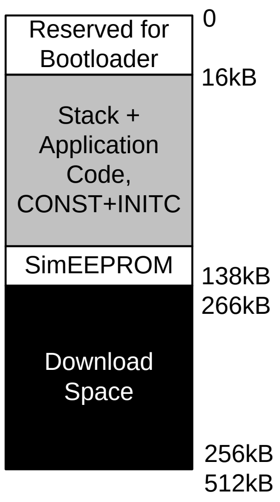

Figure 2. Local Storage Bootloader Memory Map

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. 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 App Builder, 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

Any application bootloader can have a secure variant designed to only accept encrypted EBL files. Encrypted EBL files are secured using a symmetric key that is stored in the device's CIB 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.

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 CIB 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 CIB 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 CIB token to change the security key without erasing the entire device.

4.2 Creating Encrypted EBL Images

An IAR workbench project file generated by AppBuilder will create an S37 file as the default output. If a bootloader is selected from AppBuilder then em3xx_convert will run as part of the IAR build post-processing and generate an EBL file.

In order to encrypt the image it is necessary to do the following.

1. Create a secure bootloader key file (one-time event)
2. Use **em3xx_convert** with the unencrypted EBL to generate an encrypted EBL.

4.2.1 Generating a Key File

The secure bootloader key file is used for two main purposes.

1. To program the MFG tokens of the chips running the secure bootloader
2. For em3xx_convert to encrypt EBL files

The file is a simple text file formatted as follows:

```
# Comment lines start with '#'
TOKEN_MFG_SECURE_BOOTLOADER_KEY: <ascii-hex-data>
```

Here is a sample file:

```
# Test Key file for encrypting/decrypting EBL
TOKEN_MFG_SECURE_BOOTLOADER_KEY: 0123456789ABCDEF0123456789ABCDEF
```

The **em3xx_convert** tool has the ability to generate a cryptographic quality random number for the secure bootloader key. It does this by using the Windows Cryptographic APIs, or `/dev/random` on Linux. This can be done with the following command:

```
em3xx_convert --generate <output-key-filename>
```

It is only necessary to generate a key file once. Subsequent encryption operations of additional images should use the same key file. Be sure to keep this key file secret and in a safe location to prevent attackers from being able to decrypt your images.

4.2.2 Encrypting the EBL image

To encrypt the EBL we run em3xx_convert utility specifying the key file and unencrypted EBL as inputs.

```
em3xx_convert --encrypt <key-filename> <unencrypted-ebl>
```

It is expected that the filename of the EBL have an ".ebl" extension. The resulting file will be an ".ebl.encyrpted" file with the same base name as the input. You can view the resulting file with the command:

```
em3xx_convert --print <encrypted-ebl>
```

You will see something that looks like this:

```
em3xx_convert (Version 3.1b14.1361288286)

Found EBL Tag = 0xFB05, length    6, [EBL Header for Encryption]
  Version:         0x201
  Encryption Type: 0x01 (CCM)
  Signature:       0xE350 (Correct)
Found EBL Tag = 0xFA06, length   16, [EBL Init for Encryption]
  Cipher Length: 149056
  Nonce:         A5CD0C1A1A32ECD3 50CBBC9C
  Associated Data Length: 0
Found EBL Tag = 0xF907, length  144, [Encrypted EBL Data]
Found EBL Tag = 0xF907, length 1928, [Encrypted EBL Data]
Found EBL Tag = 0xF907, length 2056, [Encrypted EBL Data]

(70 additional tags of the same type and length.)

Found EBL Tag = 0xF907, length 1008, [Encrypted EBL Data]
Found EBL Tag = 0xF709, length    8, [MAC Data for Encryption]
  MAC: FDA3172F281AD40C
Found EBL Tag = 0xFC04, length    4, [EBL End Tag]
  CRC: 0x922F3129

The CRC of this EBL file is valid (0xdebb20e3)

File has 0 bytes of end padding.
```

4.3 Decrypting an EBL file

Normally it is not necessary to use em3xx_convert to decrypt an encrypted EBL. The secure bootloader is the only consumer of encrypted EBL files that will be decrypted. However for the purposes of debugging it may be useful to decrypt the files to see the contents. To do this one must execute the following command:

```
em3xx_convert --decrypt <key-filename> <encrypted-ebl>
```

The encrypted EBL file must be named ".ebl.encrypted". The output file will have the same basename and without the ".encrypted" extension.

4.4 Loading the Secure Bootloader key onto the chip

In order for the secure bootloader to function properly, it must have a copy of the secure bootloader key in the manufacturing token. The em3xx_load tool can view, erase and modify the manufacturing tokens.

To load the secure bootloader key you will need the secure bootloader key file previously generated. The following command-line will load the value into the manufacturing tokens, assuming the device is connected via USB:

```
em3xx_load --cibtokenspatch <key-file>
```

You can verify that the key is loaded by executing the command:

```
em3xx_load --cibtokensprint
```

Examine the output and look for the line with TOKEN_MFG_SECURE_BOOTLOADER_KEY.

```
em3xx_load version 3.1b14.1361288286
Connecting to ISA via IP address rob-350-2
DLL version 4.56.4, compiled Nov 16 2012 16:48:41
SerialWire interface selected
SWJCLK speed is 500kHz
Targeting EM357


'General' token group
TOKEN_MFG_CIB_OBS               [16 byte array ] : A55AFFFFFFFFFFFF
FFFFFFFFFFFFFFFF
TOKEN_MFG_CUSTOM_VERSION        [16-bit integer] : 0xFFFF
TOKEN_MFG_CUSTOM_EUI_64         [ 8 byte array ] : FFFFFFFFFFFFFFFF
TOKEN_MFG_STRING                [16 byte string] : "rob-350-2" (9 of 16 chars)
                                                   726F622D3335302D
32FFFFFFFFFFFFFF
TOKEN_MFG_BOARD_NAME            [16 byte string] : "rob-350-2" (9 of 16 chars)
                                                   726F622D3335302D
32FFFFFFFFFFFFFF
TOKEN_MFG_MANUF_ID              [16-bit integer] : 0xFFFF
TOKEN_MFG_PHY_CONFIG            [16-bit integer] : 0xFFFF
TOKEN_MFG_BOOTLOAD_AES_KEY      [16 byte array ] : FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF
TOKEN_MFG_EZSP_STORAGE          [ 8 byte array ] : FFFFFFFFFFFFFFFF
TOKEN_MFG_OSC24M_BIAS_TRIM      [16-bit integer] : 0xFFFF
TOKEN_MFG_SYNTH_FREQ_OFFSET     [16-bit integer] : 0xFFFF
TOKEN_MFG_OSC24M_SETTLE_DELAY   [16-bit integer] : 0xFFFF
TOKEN_MFG_SECURITY_CONFIG       [16-bit integer] : 0xFFFF
TOKEN_MFG_CCA_THRESHOLD         [16-bit integer] : 0xFFFF
TOKEN_MFG_SECURE_BOOTLOADER_KEY [16 byte array ] : 0123456789ABCDEF
0123456789ABCDEF

'Smart Energy CBKE (TOKEN_MFG_CBKE_DATA)' token group
Device Implicit Cert [48 byte array ] : 0200E1F9D251F452 D0C753F7931093F1
                                        73252989153A000D 6F000092E1125445
                                        5354534543410000 0000000000000000
```

```
CA Public Key        [22 byte array ] : 0200FDE8A7F3D108 4224962A4E7C54E6
                                        9AC3F04DA6B8
Device Private Key   [21 byte array ] : 002C43A0550FA955 65735F9A7D5BE243
                                        245282789C
CBKE Flags           [ 1 byte array ] : 00

'Smart Energy Install Code (TOKEN_MFG_INSTALLATION_CODE)' token group
Install Code Flags [ 2 byte array ] : 0000
Install Code       [16 byte array ] : 1234567890ABFFFF FFFFFFFFFFFFFFFF
CRC                [16-bit integer] : 0x9EBA
```

4.5 Example Application Bootload

For details on how to setup and run an application bootload using em35x development kits (including secure application bootload) please refer to the application note *AN728 ZigBee Over-the-air Bootload Cluster Server and Client Setup using EM35x Development Kits*.