UG103.6

FUNDAMENTALS: BOOTLOADING

This document introduces bootloading for Ember ZigBee networking devices. It looks at the concepts of standalone and application bootloading and discusses their relative strengths and weaknesses. In addition, it looks at design and implementation details for each method.

New in This Revision

Initial release (contents previously published).

Contents

1 Introduction

The bootloader is a program stored in reserved flash memory that allows a node to update its application image on demand, either by serial communication or over the air. Production-level programming is typically done during the product manufacturing process yet it is desirable to be able to reprogram the system after production is complete. More importantly, it is valuable to be able to update the device's firmware with new features and bug fixes after deployment. The bootloading capability makes that possible.

Bootloading can be accomplished through a hardwired link to the device or over the air (that is through the wireless network) as shown in Figure 1.

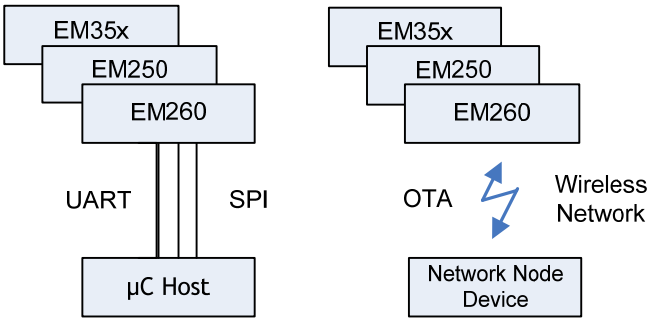

Figure 1. Bootloading Links

Ember ZigBee networking devices offer two main types of bootloaders: standalone and application. These two bootloaders can differ in the amount of flash required and location of the stored image, as discussed in the next two sections.

Silicon Labs supports devices that do not use a bootloader, but this requires external hardware such as the Debug Adapter or EM2xx USBLink to upgrade the firmware. Devices without a bootloader have no supported way of upgrading the firmware over the air once they are deployed, which is why Silicon Labs strongly advocates implementing a bootloader.

The bootloading situations described in this document assume that the source node (the device sending the firmware image to the target through a serial or OTA link) acquires the new firmware through some other means. For example, if a device on the local ZigBee network has an Ethernet gateway attached, this device could get or receive these firmware updates over the Internet. This necessary part of the bootloading process is system-dependent and beyond the scope of this document.

1.1 Memory Space for Bootloading

Figure 2 shows the memory maps for each of the Ember ZigBee networking devices.

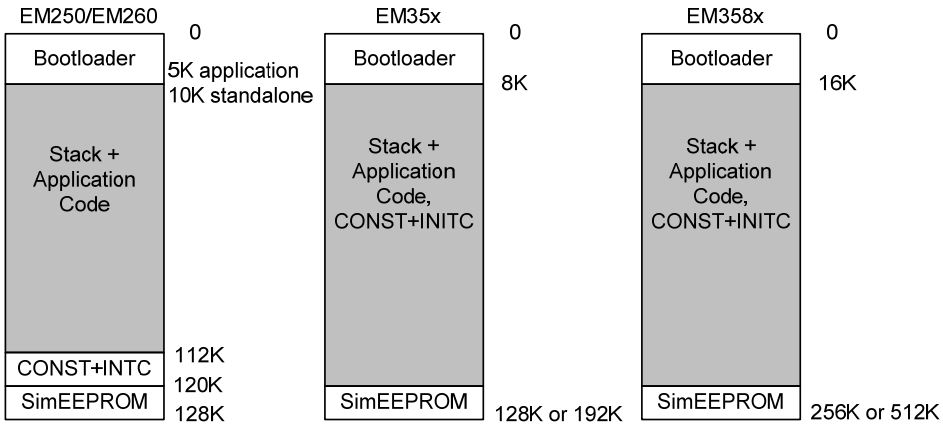

Figure 2. Ember ZigBee Networking Devices' Memory Maps

For the EM250 and EM260, a block of 5 kB or 10 kB of low flash memory is reserved to hold the bootloader and a block of 16 kB of high memory is reserved for a simulated EEPROM (8 kB) and CONST+INITC (8 kB). For the EM35x, a block of 8 kB of low flash memory is reserved to hold the bootloader, as well as either 4 kB or 8 kB of high flash memory for the simulated EEPROM. The EM358x chips reserve a block of 16kB of low flash memory for the bootloader and either 4 kB or 8 kB of high flash memory for the simulated EEPROM. In all cases, the balance of the memory space is unreserved and available to hold EmberZNet PRO and application code.

1.2 Standalone Bootloading

Standalone bootloading is 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. Very little interaction occurs between the standalone bootloader and the application running in flash. In general, the only time that the application interacts with the bootloader is when it calls `halLaunchStandaloneBootloader()`to run the bootloader. Once the bootloader is running, it receives bootload packets containing the (new) firmware image either by physical connections such as UART or SPI, or by the radio (over-the-air). Figure 3 illustrates what happens to the device's flash memory during bootloading.

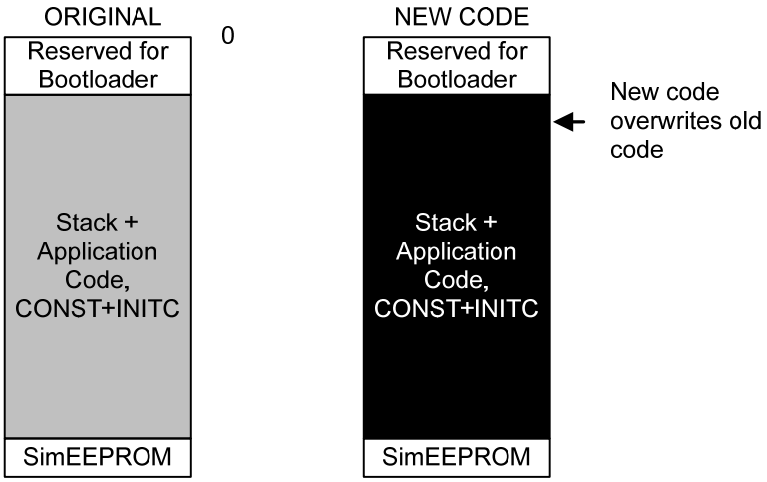

Figure 3. Standalone Bootloading Code Space (Typical)

When bootloading is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the code cannot be recovered and bootloading must start over at byte zero.

1.3 Application Bootloading

The application bootloader relies on the application to acquire the new firmware image. This image can be downloaded by the application in any way that is convenient (UART, over-the-air, etc.) but it must be stored into a region referred to as the download space. The download space is typically an external memory device such as an EEPROM or dataflash, but it can also be a section of the chip's internal flash when using the EM358x's local storage bootloader..Once the new image has been stored, the application bootloader is then called to validate the new image and copy it from the download space to flash. Since the application bootloader does not participate in acquiring the image it does not need code to operate the radio and is much smaller than the standalone bootloader. Figure 4 shows a typical memory map for the application bootloader.

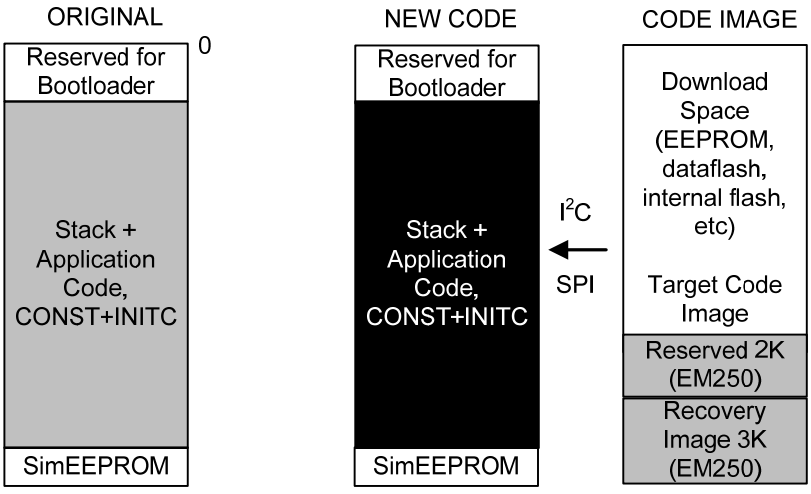

Figure 4. Application Bootloading Code Space (Typical)

Download errors do not adversely impact the current application image while storing the new image to the download space. The download process can be restarted or paused to acquire the image over time.

2 Design Decisions

Table 1 shows the bootloaders, the different types and what features are supported:

Table 1. Bootloader Types and Features

Features	Application-bootloader	Secure-application-bootloader	Local-storage-bootloader	Secure-local-storage-bootloader	Standalone-bootloader	Standalone-OTA-bootloader
Serial Link Download	Yes	Yes	Yes	Yes	Yes	Yes
Over-the-air Image Transfer *without* Application Running						Yes
Application runs while downloading new image	Yes	Yes	Yes	Yes		
Can be used in a multi-hop deployment	Yes	Yes	Yes	Yes		
Supports Encrypted Ember Bootloader Files (EBL)		Yes		Yes		
Bootload Failures can be recovered by loading stored image	Yes	Yes	Yes	Yes		
Requires External Storage	Yes	Yes				
On-chip Flash Requirements	em250: 4k em35x: 8k em358x: 16k	em35x: 8k em358x: 16k	em358x: 16k + 246k*	em358x: 16k + 246k*	em250: 4k em35x: 8k em358x: 16k	em250: 4k em35x: 8k em358x: 16k
em250	Yes				Yes	Yes
em351	Yes	Yes			Yes	Yes
em357, em3581, em3582 (192k & 256k parts)	Yes	Yes			Yes	Yes
em3585, em3586, em3588 (512k parts)	Yes	Yes	Yes	Yes	Yes	Yes

* The local storage can be configured to use more or less on-chip space for storage. 246k is a recommended amount based on a single, average-sized image kept. Individual application needs may vary. The actual bootloader is 16k.

The decision of what bootloader to deploy will depend on many factors. Some questions related to this will be:

1. Where does the device get the new upgrade image? Is this over-the-air via ZigBee? Using a separate interface connected to the Internet?
2. Will the device have an external memory chip to store a new application image?
3. If the device receives the new image over-the-air, will it be multiple hops away from the server holding the download image?
4. What kind of security of the image is necessary?

3 Ember Bootload (EBL) File

All bootloaders require the image they are processing to be in the Ember Bootload File format. The EBL format is described in this section, and is automatically created during the build process by the command-line programs **em2xx_convert** or **em3xx_convert**.

3.1 Basic File Format

The EBL file format is composed of a number of **tags** that indicate a format of the subsequent data and the length of the entire tag. The format of a tag is as follows:

Tag ID	Tag Length	Tag Payload
2-bytes	2-bytes	Variable (according to tag length)

The details of the tag formats can be found in these header files:

em2xx Series	hal/micro/xap2b/em250/em250-ebl.h
em3xx Series	hal/micro/cortexm3/bootloader/ebl.h

3.1.1 Unencrypted Tag Descriptions

Table 2 lists the tags for an unencrypted EBL image.

Table 2. Tags for Unencrypted EBL Image

Tag Name	ID	Description
EBL Header Tag	0x0000	This Contains information about the chip the image is intended for, the AAT (application address table), Ember Stack Version, Customer Application Version, build date, and build timestamp. This must be the first tag.
EBL Program Data	0xFE01	This contains information about what data to program at a specific address into the main flash memory.
EBL Program Manufacture Data	0x02FE	This contains information about what data to program at a specific address within the Customer Information Block (CIB) section of the chip.
EBL End	0xFC04	This tag indicates the end of the EBL file. It contains a 32-bit CRC for the entire file as an integrity check. The CRC is a non-cryptographic check. This must be the last tag.

A full EBL image looks is shown in Figure 5:

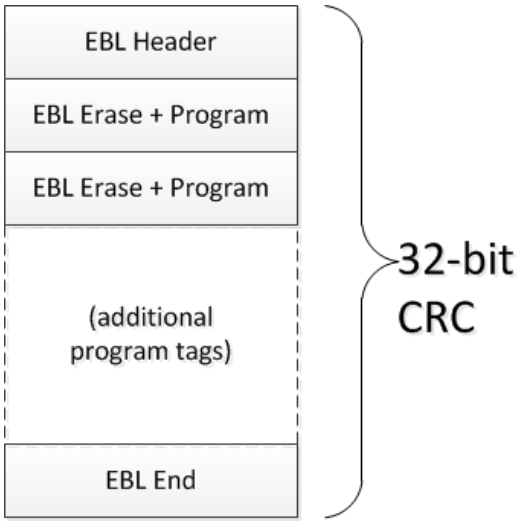

Figure 5. EBL Image

3.1.1 Data Verification

The EBL file format includes three 32bit CRC values to verify the integrity of the file. These values are computed using the halCommonCrc32() function which can be found in hal/micro/generic/crc.c. The initial value of the CRC used in the computation is 0xFFFFFFFF.

Table 3**Error! Reference source not found.** describes the data integrity checks built into the .EBL download format.

Table 3. EBL Data Integrity Checks

Integrity Check	Description
Header CRC	The header data contains the headerCrc field (aatCrc on the EM35x), a 4 byte, one's complement, LSB-first CRC of the header bytes only. This is used to verify the integrity of the header. On the EM35x this CRC assumes that the value of the type field in the AAT is set to 0xFFFF.
EBLTAG_END CRC	The end tag value is the one's complement, LSB-first CRC of the data download stream, including the header the end tag and the CRC value itself. This is used as a running CRC of the download stream, and it verifies that the download file was received properly. The CRC in the tag is the one's complement of the running CRC and when that value is add to the running calculation of the CRC algorithm it results in predefined remainder of 0xDEBB20E3.
Image CRC	The header's imageCrc field is the one's complement, MSB-first CRC of all the flash pages to be written including any unused space in the page (initialized to 0xFF). It does not include the EBL tag data and assumes that the first 128 bytes of the AAT on the EM35x and the whole EBL Header on the EM2xx are set to 0xFF. This is used after the image download is complete and everything but the header has been written to flash to verify the download. The download program does this by reading each flash page written as it is defined in the header's pageRanges[] array and calculating a running CRC.

3.2 Encrypted Ember Bootload File Format

The Ember encrypted bootloader file format is similar to the unencrypted version. It introduces a number of new tags. A bootloader is said to be a 'secure' bootloader if it accepts only encrypted EBL images.

3.2.1 Encrypted Tag Descriptions

Table 4 lists the encryption tags and their descriptions.

Table 4. Encrypted Tag Descriptions

Tag Name	ID	Description

EBL Encryption Header	0xFB05	This contains basic information about the image. The header is not authenticated or encrypted.
EBL Encryption Init Header	0xFA06	This contains information about the image encryption such as the Nonce, the amount of encrypted data, and an optional block of authenticated but non-encrypted data. The tag is authenticated.
EBL Encrypted Program Data	0xF907	This contains data about what to program into the flash memory. The contents are encrypted. The data is encrypted using AES-CCM.
EBL Encryption MAC	0xF709	This contains the message authentication code used to validate the contents of the authenticated and encrypted portions of the image.

An encrypted image will wrap normal, unsecured, EBL tags inside EBL Encrypted Program Data tags. The contents of each tag are encrypted but the encrypted data tag ID and tag length fields are not. For each tag that exists in the unencrypted EBL a corresponding Encrypted Program Data tag will be created.

The encrypted file format is shown in Figure 6:

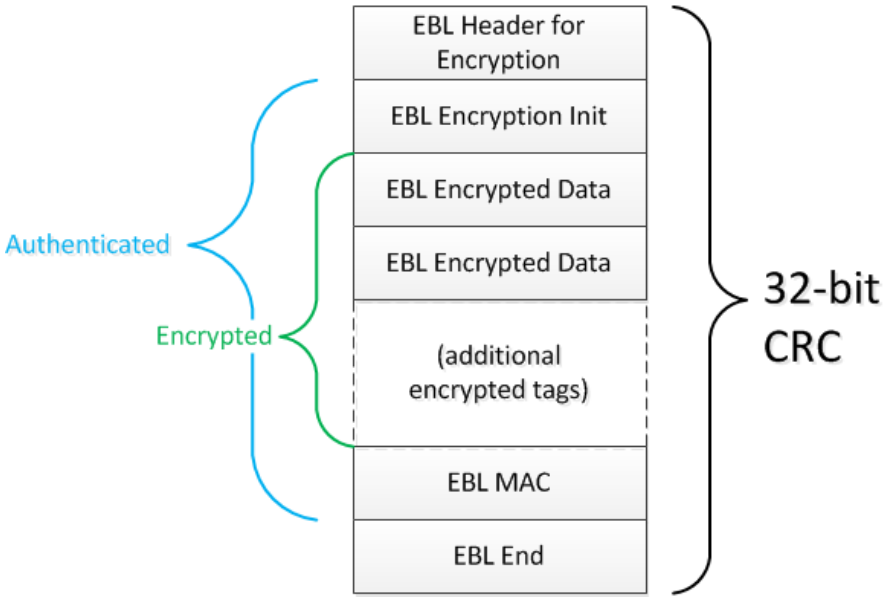

Figure 6. Encrypted File Format

3.2.2 Nonce Generation

The nonce for the encrypted image is a 12-byte value contained within the EBL Encryption Init tag. The **em3xx_convert** tool will generate a random nonce value during encryption and store this in the EBL Encryption Init tag.

It is important that a nonce value is not used twice to encrypt two different images with the same encryption key. This is because CCM relies on using XOR with a block of pseudo-random noise to encrypt the contents of a file. However with a 12-byte random nonce the chances of this are roughly 1 in 2^{96}.

3.2.3 Image Validation

The encrypted EBL image is protected by a message authentication code (MAC) that is calculated over the unencrypted contents of each tag, The MAC is stored in the EBL MAC tag and the secure bootloader will calculate and validate the stored MAC prior to loading any of the data from the image.

4 Standalone Bootloading

4.1 Overview

A standalone bootloader is a program that uses either the serial port or the radio in a single-hop, 802.15.4 MAC-only mode to get an application image. Standalone bootloading allows the new application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. Figure 7 shows a sample memory map and how the bootloading process overwrites the old code image. It should be clear from this illustration that the bootloading process is destructive and must proceed to completion if a functional application is to reside in this code space. A failure during bootloading means that the process must begin again. Bootloading for the EM35x and EM250 work similarly and are discussed together. The operating model of the EM260 is a little more complicated, but the basic idea behind bootloading an EM260 application is the same. Contact customer support if you have specific questions about bootloading an EM260.

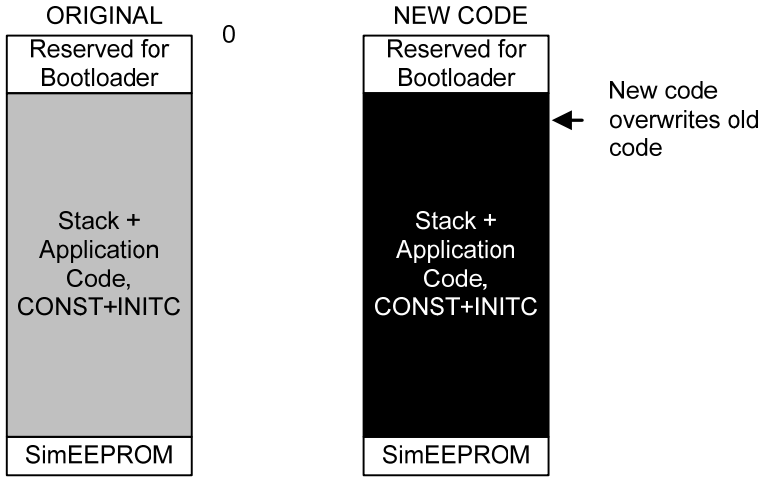

Figure 7. Standalone Bootloading Code Space (Typical)

Note: When bootloading is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the code cannot be recovered and bootloading must start over at byte zero.

For more details on Standalone Bootloading please see the app note AN760-.

5 Application Bootloading

5.1 Overview

An application bootloader is run after the running application has completely downloaded the upgrade image file. The application bootloader expects that the image either lives in external memory accessible by the bootloader, or it will be transferred to the bootloader via serial connection. The upgrade image file consumed by the application bootloader is an EBL file. The image may be wrapped in another format, such as the ZigBee Over-the-air Bootload Cluster format, but the EBL portion of the file must live at the top of the external memory, or stripped off prior to serial transfer to the bootloader.

Application bootloading requires another mechanism to transfer the file to the running application, such as an over-the-air protocol. The ZigBee Over-the-air Bootload cluster is an example of such a protocol.

Please see application note *AN772 – Using Application Bootloader* for more information.

5.2 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*.

CONTACT INFORMATION

Silicon Laboratories Inc.

400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page for ZigBee products:
www.silabs.com/zigbee-support and register to submit a technical support request