UG103.6: Bootloading Fundamentals

This document introduces bootloading for Silicon Labs networking devices. It summarizes the differences between the Silicon Labs Gecko Bootloader and the legacy Ember bootloaders and discusses their applicability by platform. It describes 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. Finally, it describes the bootloader file formats.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. The documents can be used a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

KEY POINTS

- Introduces the Gecko Bootloader and the legacy Ember bootloaders.
- Summarizes the key features the bootloaders support and the design decisions associated with selecting a bootloader.
- Describes bootloader file formats.

1. Introduction

The bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and possibly perform some integrity checks. Firmware image update occurs 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 firmware image update capability makes that possible.

Silicon Labs supports devices that do not use a bootloader, but this requires external hardware such as a Debug Adapter (Silicon Labs ISA3 or Wireless Starter Kit (WSTK)) or third-party SerialWire/JTAG programming device to update the firmware. Devices without a bootloader have no supported way of updating the firmware over the air once they are deployed, which is why Silicon Labs strongly advocates implementing a bootloader.

In March of 2017, Silicon Labs introduced the Gecko Bootloader, a code library configurable through Simplicity Studio's IDE to generate bootloaders that can be used with a variety of Silicon Labs protocol stacks. The Gecko Bootloader can be used with EFR32MG1/ EFR32BG1 (EFR32xG1) and EFR32xG1+Flash, however, beginning with the EFR32MG12/ EFR32BG12/ EFR32FG12 (EFR32xG12) platform, it and all future Mighty Gecko, Flex Gecko, and Blue Gecko releases will use the Gecko Bootloader only. Legacy Ember bootloader applications for use with specific protocols such as EmberZNet PRO and platforms including the EM3x will continue to be provided for use with those platforms. Support for legacy Bluetooth bootloaders was removed from version 2.7.0 of the Bluetooth SDK (Software Development Kit) in December 2017.

The Gecko Bootloader and the legacy Ember bootloaders use customized update image file formats, described further in section 5. Bootload File Formats. The update image file consumed by a Gecko Bootloader-generated application bootloader is a GBL (Gecko BootLoader) file, and that consumed by the legacy Ember bootloaders is an EBL (Ember BootLoader) file

Bootloading a firmware update image can be accomplished in two ways. The first is Over-The-Air (OTA), that is, through the wireless network, as shown in the following figure.

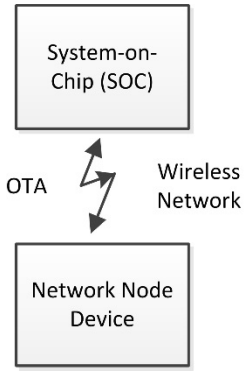


Figure 1.1. OTA Bootloading Use Case

The second is through a hardwired link to the device. The following figure represents the serial bootloader use cases for SoCs (System on Chips) using either a UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Protocol Interface), or USB (Universal Serial Bus) interface, and for NCPs (Network Coprocessors) using either UART or SPI.

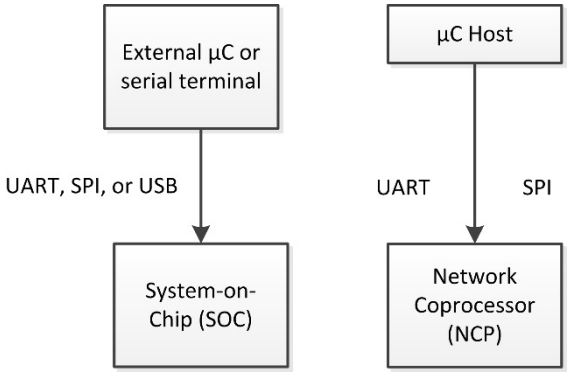


Figure 1.2. Serial Bootloaders Use Cases

Silicon Labs networking devices use bootloaders that perform firmware updates in two different modes: standalone (also called standalone bootloaders) and application (also called application bootloaders). Application bootloaders are further divided into those that use external storage for the download update image, and those that use local storage. These bootloader types are discussed in the next two sections.

The firmware update 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 firmware update process is system-dependent and beyond the scope of this document.

1.1 Standalone Bootloading

A standalone bootloader is a program that that uses an external communication interface, such as UART or SPI, to get an application image. Standalone firmware update 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 requests a reboot into 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).

When the firmware update process 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 the process must start over. For more information about legacy standalone bootloaders, see *AN760: Using the Ember Standalone Bootloader*. For information about configuring the Gecko Bootloader as a standalone bootloader, see *UG266: Silicon Labs Gecko Bootloader User Guide*.

1.2 Application Bootloading

An application bootloader begins the firmware update process after the running application has completely downloaded the update image file. The application bootloader expects that the image either lives in external memory accessible by the bootloader or in a portion of main flash memory (if the chip has sufficient memory to support this local storage model).

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 a local storage variant of the application 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, and the entire image is downloaded before the firmware update process is started, download errors do not adversely affect the running image. The download process can be restarted or paused to acquire the image over time. The integrity of the downloaded update image can be verified before initiating the firmware update process, to prevent a corrupt or non-functional image from being applied.

The legacy Ember application bootloader provides UART standalone bootloading capability as a recovery mechanism in case both the running application image and the upgrade image are corrupt. The Gecko Bootloader can be configured to accept a list of multiple upgrade images to attempt to verify and apply. This allows the Gecko Bootloader to store what is in effect a backup copy of the update image, which it can access if the first image is corrupt.

Note that EmberZNet and Silicon Labs Thread NCP platforms do not utilize an application bootloader because the application code resides on the host rather than on the NCP directly. Instead a device acting as a serial coprocessor would utilize a standalone bootloader designed to accept code over the same serial interface as the expected NCP firmware uses. However, the host application (residing on a separate MCU from the NCP) can utilize whatever bootloading scheme is appropriate. Silicon Labs Bluetooth NCPs can use the legacy OTA DFU bootloader.

For more information on application bootloaders, see *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN772: Using the Ember Application Bootloader*.

2. About the Gecko Bootloader

The Silicon Labs Gecko Bootloader is a configurable code library that can be used with all the newer Silicon Labs Gecko MCUs and wireless MCUs. It uses a specially-formatted update image file called a GBL file. The Gecko Bootloader has a two-stage design, where a minimal first stage bootloader is used to update the main bootloader. This allows for in-field updates of the main bootloader, including adding new capabilities, changing communication protocols, adding new security features and fixes, and so on. The Gecko Bootloader consists of three component parts:

Core: The bootloader core contains the main function of both bootloader stages. It also contains functionality to write to the internal main flash, to perform a bootloader update, and to reset into the application flagging applicable reset reasons.

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

Plugin: All parts of the main bootloader that are either optional or selectable for different configurations are implemented as plugins. Each plugin has a generic header file, and one or more implementations. The current release contains plugins for functionality like UART and SPI communication protocols, SPI flash storage, internal flash storage, and different cryptographic operations.

2.1 Features

Gecko Bootloader features include:

- Field-updateable
- Secure boot
- Signed GBL firmware update image file
- Encrypted GBL firmware update image file

These features are summarized in the following sections and described in more detail in *UG266: Silicon Labs Gecko Bootloader User Guide*. Protocol-specific information about using the Gecko Bootloader may be found in the following documents:

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

2.1.1 Field-Updateable

Bootloader firmware field update capability is provided by a two-stage design, first stage and main stage. The minimal first stage of the bootloader, which is not field-updateable, can only update the main bootloader by reading to and writing from fixed addresses in internal flash. To perform a main bootloader update, the running main bootloader verifies the integrity and authenticity of the bootloader update image, writes the bootloader update 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 update image before copying the update image to the main bootloader location.

2.1.2 Secure Boot

Secure boot is designed to prevent an untrusted image from running on the device. When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot using asymmetric cryptography. The signature algorithm used is ECDSA-P256-SHA256. The public key is written to the device during manufacturing, while the private key is kept secret. This ensures that the application was created and signed by a trusted party.

2.1.3 Signed GBL Update Image File

The Gecko Bootloader supports enforcing cryptographic signature verification of the update image file in addition to Secure Boot. This allows the bootloader and application to verify that the application or bootloader update comes from a trusted source before starting the update process. The signature algorithm used is ECDSA-P256-SHA256. The public key is the same key as for secure boot, written to the device during manufacturing, while the private key is never distributed. This ensures that the GBL file was created and signed by a trusted party.

2.1.4 Encrypted GBL Update File

The GBL update file can also be encrypted, to prevent eavesdroppers from getting hold of the plaintext firmware image. The encryption algorithm used is AES-CTR-128, and the encryption key is written to the device during manufacturing.

2.2 Applicability

The following table indicates which bootloaders can be used with different platforms.

Table 2.1. Bootloaders by Platform

Platform	Gecko Bootloader	Legacy Ember Bootloaders	Legacy Bluetooth Bootloaders
EM3x	No	Yes	*
EFR32MG1, EFR32MG1+Flash	Yes	No**	*
EFR32BG1	Yes	No	*
EFR32BG1+Flash	Yes	No	*
EFR32FG1	Yes	No**	*
EFR32MG12, EFR32BG12, EFR32FG12	Yes	No	*
Future products	Yes	No	*

* Support for the legacy Bluetooth Bootloaders was removed in the Bluetooth SDK version 2.7.0 release.

** Support for these platforms was deprecated in EmberZNet SDK version 6.1.0 and Silicon Labs Thread version 2.5.0.

3. Memory Space For Bootloading

3.1 Gecko Bootloader

The first stage of the bootloader takes up a single flash page. On devices with 2 kB flash pages, like EFR32MG1, this means that the first stage takes 2 kB.

The size of the main bootloader is dependent on the functionality required. With a typical bootloader configuration, the main bootloader takes up 14 kB of flash, bringing the total bootloader size to 16 kB.

Silicon Labs recommends reserving at least 16 kB for the bootloader.

On EFR32xG1 devices (Mighty Gecko, Flex Gecko, and Blue Gecko families), the bootloader resides in main flash.

- First stage bootloader @ 0x0
- Main bootloader @ 0x800
- Application @ 0x4000

On newer devices (EFR32xG12 and later), the bootloader resides in the bootloader area in the Information Block

- Application @ 0x0
- First stage bootloader @ 0x0FE10000
- Main bootloader @ 0x0FE10800

3.2 Legacy Ember Bootloaders

The following figure shows the memory maps for a typical Silicon Labs mesh networking SOC or NCP.

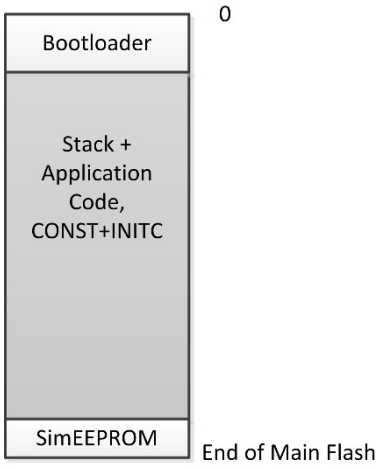


Figure 3.1. Typical Silicon Labs Mesh Networking Devices' Memory Map

For each Silicon Labs mesh networking platform (in either the SOC or NCP use case), a block of flash memory (typically 8 kB or 16 kB, depending on the IC variant used) is reserved at the start of main flash memory to hold the bootloader, and a block of flash memory (between 4 kB and 36 kB depending on the implementation) is reserved at the end of flash memory for the simulated EEPROM. In all cases except for the Local Storage Application Bootloader, the balance of the memory space is unreserved and available to hold networking stack and application code.

4. Design Decisions

The decision of what bootloader type to deploy depends on many factors. Note that the platform type and available flash memory may limit bootloader choices.

Some questions related to this are:

- Where does the device get the new update image? Is this over-the-air via the networking protocol? Using a separate interface connected to the Internet?
- Will the device have an external memory chip to store a new update image? If not, is there enough internal flash memory to store both a current and a newly downloaded copy of the largest expected application image?
- If the device receives the new image over-the-air, will it be multiple hops away from the server holding the download image?
- What kind of image security is needed?
- What communications driver will be used (in the single protocol case)?
- Does the use case require more than one protocol?

4.1 Gecko Bootloader

The configurable design of the Gecko Bootloader platform means that developers can create bootloaders to fit almost any design choice. See *UG266: Silicon Labs Gecko Bootloader User's Guide* for more details.

4.2 Legacy Ember Bootloaders

The following table shows the legacy Ember bootloaders, the different types, and what features the bootloaders support.

Table 4.1. Ember 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	EM34x/5x: 8 kB EM358x/9x: 16 kB EFR32: not supported[2]	EM34x/5x: 8 kB EM358x/9x: 16 kB EFR32: not supported[2]	EM34x/5x: not supported EM358x/9x: 16 kB + 246 kB[1] EFR32: not supported[2]	EM34x/5x: not supported EM358x/9x: 16 kB + 246 kB[1] EFR32: not supported[2]	EM34x/5x: 8 kB EM358x/9x: 16 kB EFR32: not supported[2]	EM34x/5x: 8 kB EM358x/9x: 16 kB EFR32: not supported[2]
EM34x, EM351	Yes	Yes			Yes	Yes
EM357, EM3581, EM3582 (192 kB & 256 kB parts)	Yes	Yes			Yes	Yes
EM3585, EM3586, EM3587, EM3588 (512 kB parts)	Yes	Yes	Yes	Yes	Yes	Yes
EFR32 (128 kB and 256 kB parts)	Not supported[2]	Not supported[2]			Not supported[2]	

[1]The local storage can be configured to use more or less on-chip space for storage. 246 kB is a recommended amount based on a single, average-sized image kept on a 512 kB part. Individual application needs may vary. The actual bootloader is 16 kB.

[2]Use the Gecko Bootloader to create a similar configuration for these platforms.

5. Bootload File Formats

The bootload file formats described in this section are generated by Simplicity Commander commands. For more information, see *UG162: Simplicity Commander Reference Guide*.

5.1 Gecko Bootload (GBL) File

The GBL file format is used by the Gecko Bootloader.

5.1.1 File Format

5.1.1.1 File Structure

The GBL 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
4 bytes	4 bytes	Variable (according to tag length)

5.1.1.2 Plaintext Tag Descriptions

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	This must be the first tag in the file. The header tag contains the version number of the GBL file specification, and flags indicating the type of GBL file – whether it is signed or encrypted.
GBL Application Info Tag	0xF40A0AF4	This tag contains information about the application update image that is contained in this GBL file
GBL Bootloader Tag	0xF50909F5	This tag contains a complete bootloader update image.
GBL Program Data Tag	0xFE0101FE or 0xFD0303FD	This tag contains information about what application data to program at a specific address into the main flash memory.
GBL Metadata Tag	0xF60808F6	This tag contains metadata that the bootloader does not parse, but can be returned to the application through a callback.
GBL Signature Tag	0xF70A0AF7	This tag contains the ECDSA-P256 signature of all preceding data in the file.
GBL End Tag	0xFC0404FC	This tag indicates the end of the GBL 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.

The allowed sequence of GBL tags in a GBL file is shown in the following figure.

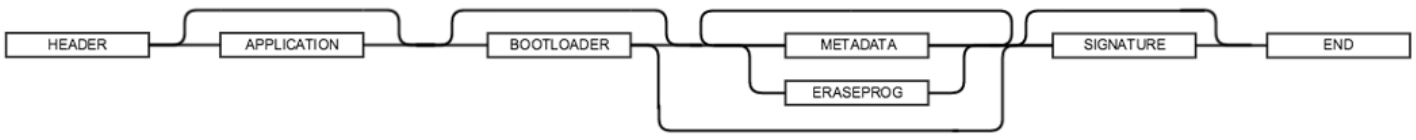


Figure 5.1. GBL Tag Sequence

5.1.1.3 Encrypted Tag Descriptions

The encrypted GBL file format is similar to the unencrypted version. It introduces a number of new tags.

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	The GBL header is the same as for a plain-text GBL file, but the flag indicating that the GBL file is encrypted must be set.
GBL Encryption Init Header	0xFA0606FA	This contains information about the image encryption such as the Nonce and the amount of encrypted data.
GBL Encrypted Program Data	0xF90707F9	This contains an encrypted payload containing a plaintext GBL tag, one of Application Info, Bootoader, Metadata or Program Data. The data is encrypted using AES-CTR-128.

The allowed sequence of GBL tags in an encrypted GBL file is shown in the following figure.

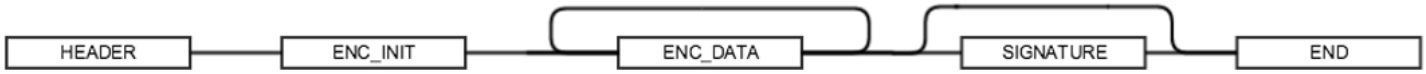


Figure 5.2. Encrypted GBL Tag Sequence

5.1.2 Image Verification

The optional GBL signature tag can be used to ensure the authenticity of the GBL file. Silicon Labs strongly recommends that the bootloader is configured to only accept signed GBL files.

5.2 Ember Bootload (EBL) File

All Ember bootloaders require the image they are processing to be in the EBL (Ember Bootload) File format.

5.2.1 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:

Platform	Header Filename
EM3x Series	`<hal>/micro/cortexm3/bootloader/ebl.h`
EFR32 Series	Not supported.

5.2.1.1 Unencrypted Tag Descriptions

The following table lists the tags for an unencrypted EBL image.

Table 5.1. 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), 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 (for EM35x devices) or UserData section (for EFR32™ devices) 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 the following figure.

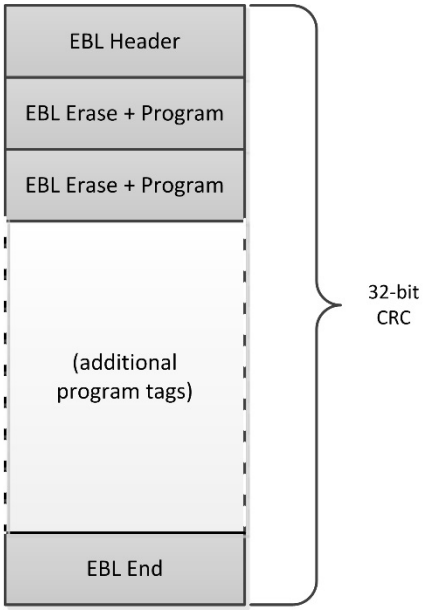


Figure 5.3. EBL Image

5.2.1.2 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.

The following table describes the data integrity checks built into the .EBL download format.

Table 5.2. EBL Data Integrity Checks

Integrity Check	Description
Header CRC	The header data contains the headerCrc field (also referred to as aatCrc in other areas of code), a 4-byte, one's complement, LSB-first CRC of the header bytes only. This is used to verify the integrity of the header. 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. 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.

5.2.2 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.

5.2.2.1 Encrypted Tag Descriptions

The following table lists the encryption tags and their descriptions.

Table 5.3. 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 the following figure:

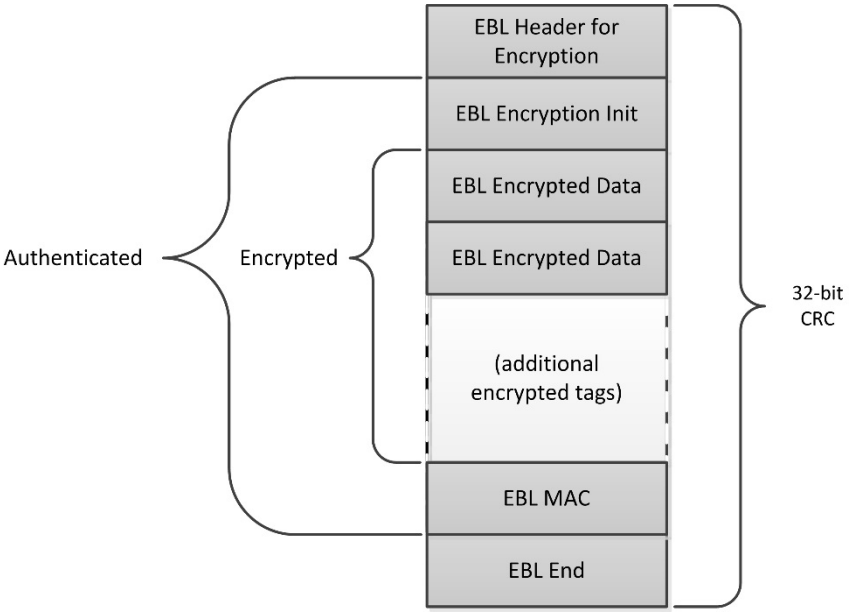


Figure 5.4. Encrypted File Format

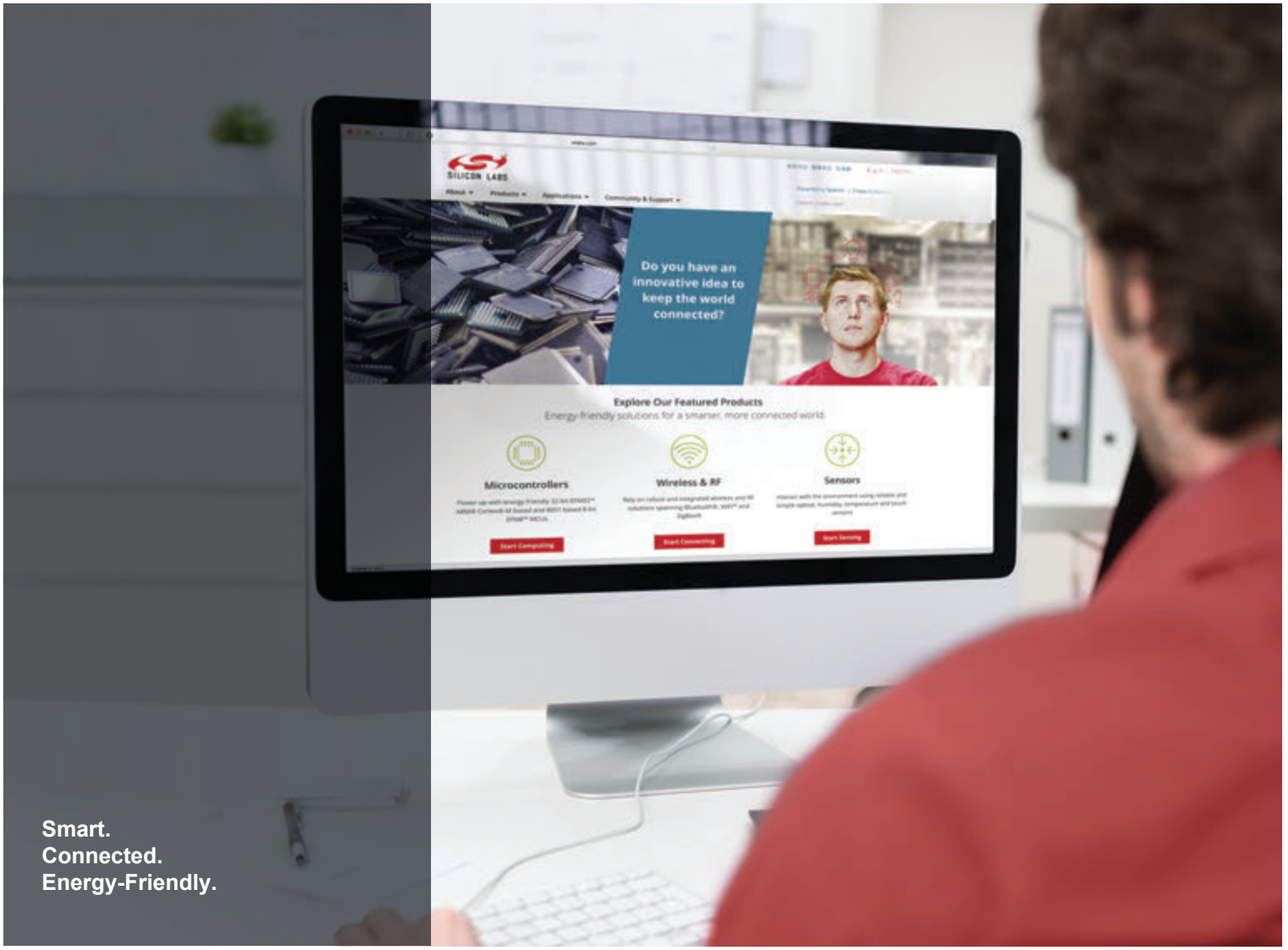
5.2.2.2 Nonce Generation

The nonce for the encrypted image is a 12-byte value contained within the EBL Encryption Init tag. The em3xx_convert or Simplicity Commander 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}.

5.2.2.3 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.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

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

http://www.silabs.com