

# AN945: EFM8 Factory Bootloader User's Guide

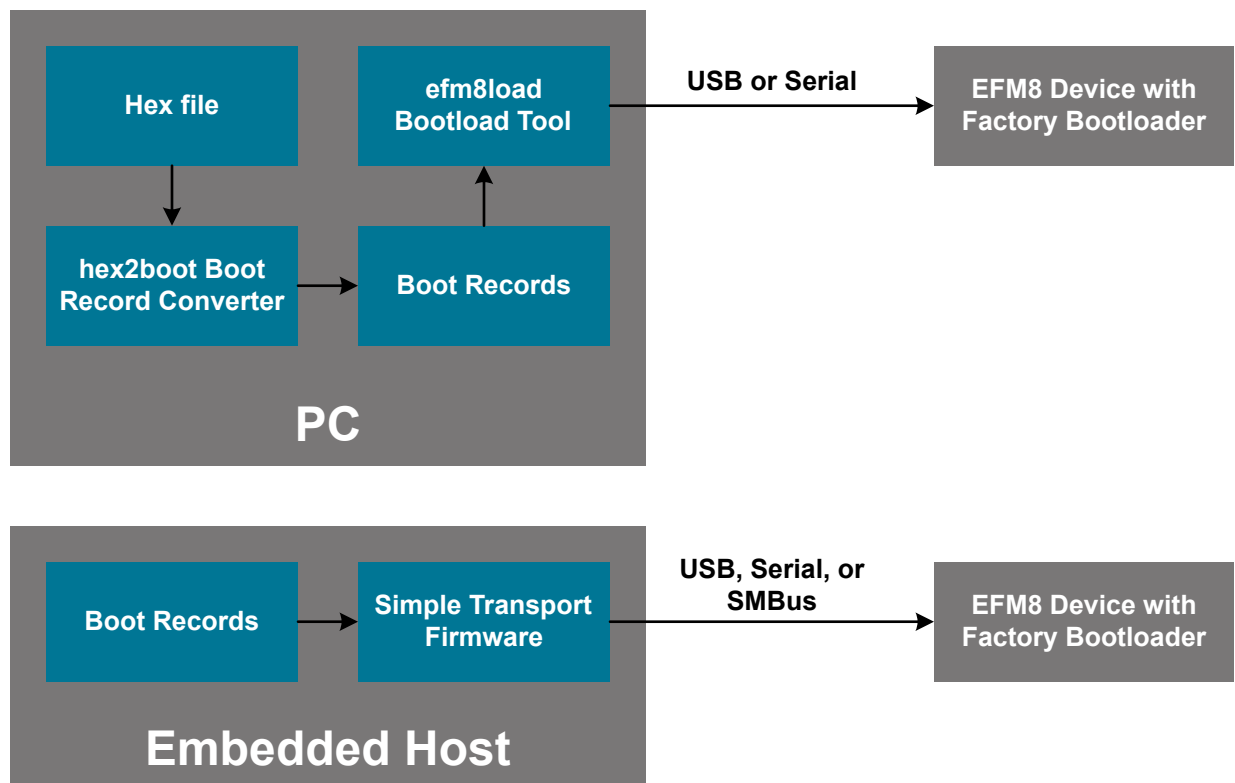


This document describes the factory-programmed bootloaders available in EFM8 devices.

In addition to describing the bootloader features, this document details how to use the bootloader and make updates to the bootloader firmware source code or Python host software if customizations are needed.

#### KEY POINTS

- The EFM8 factory-programmed bootloader provides basic production programming or field update support.
- It does not provide for secure code delivery across the host interface or secure storage inside the device.
- Source code is provided for both the host-side python tools and for the bootloader firmware to enable customizations.



## 1. Introduction

The EFM8 devices are factory programmed with a bootloader. [Table 1.1 EFM8 Device Bootloader Support on page 2](#) indicates which devices are orderable with the specified factory-programmed bootloader. The term 'Example' indicates bootloaders that are provided as example source code, not precompiled hex files.

**Table 1.1. EFM8 Device Bootloader Support**

| Device Family | UART Boot-loader | USB Boot-loader | SMBus Boot-loader | Notes   |
|---------------|------------------|-----------------|-------------------|---|
| EFM8BB1x      | Yes              |                 |                   | Bootloader support on Rev A devices with date code $\geq$ 1601. |
| EFM8BB2x      | Yes              |                 |                   | Bootloader support on Rev C devices with date code $\geq$ 1601. |
| EFM8BB3x      | Yes              |                 | Example           |   |
| EFM8LB1x      | Yes              |                 | Yes               | SMBus bootloader available in two pinout variations.            |
| EFM8SB1x      | Yes              |                 |                   | Bootloader support on Rev A devices with date code $\geq$ 1544. |
| EFM8SB2x      | Yes              |                 |                   | Bootloader support on Rev B devices.                            |
| EFM8UB1x      | Example          | Yes             |                   | Bootloader support on Rev C devices with date code $\geq$ 1601. |
| EFM8UB2x      | Example          | Yes             |                   | Bootloader support on Rev B devices.                            |
| EFM8UB3x      | Example          | Yes             |                   |   |

**Note:** Device revisions and date codes prior to those specified in the table do not have a factory-installed bootloader, nor are they compatible with the provided bootloader hexfiles or example code. For information on using earlier-revision devices with bootloaders, see this Knowledge Base entry: [https://www.silabs.com/community/mcu/8-bit/knowledge-base.entry.html/2017/06/12/an945\\_bootloaderon-Li6k](https://www.silabs.com/community/mcu/8-bit/knowledge-base.entry.html/2017/06/12/an945_bootloaderon-Li6k)

The EFM8 bootloader enables:

1. Production Programming — Devices can be programmed in a production environment without using the debug interface, which requires access points on the PCB and a debug adapter that supports the proprietary C2 flash programming protocol. Using the factory bootloader allows the flash to be programmed via a standard serial interface such as UART, SMBus, or USB.
2. Field Updates — Updates can also be issued to devices in the field without the need for end users to access the debug pins or use the debug adapter hardware. See the [Security](#) section of this note for additional recommendations regarding field updates.

The bootloaders are designed to be as small as possible. For example, the UART and SMBus versions consume a single 512-byte flash page, and the USB version consumes 1.5 KB of flash. In addition, the bootloader is generally located in the code security page to enable the bootloader to write and erase locked application space. More information on the bootloader memory placement on each device can be found in the device data sheet or reference manual.

### 1.1 AN945SW Software Package

**Note:** The descriptions of the AN945SW package in this document apply to the AN945SW Revision 0.3 and newer releases, not necessarily previous releases. The software release number is specified in the `README.txt` file in the AN945SW package.

The AN945SW software package contains precompiled bootloaders and applications, and utility programs for working with EFM8 bootloaders. The AN945SW software package contains three subdirectories:

- `ProductionDeviceHexFiles` (Contains precompiled bootloader hex files for all production EFM8 devices)
- `StarterKitHexFiles` (Contains precompiled bootloader and application hex files for use with EFM8 Starter Kits)
- `Tools` (Contains utilities and Python scripts used for working with EFM8 bootloaders)

The precompiled bootloader and application files are provided so that the user can quickly download and evaluate the bootloader and app for a particular device, and so that the original bootloader can be restored to the device whenever desired. The project and source files for bootloaders and example applications are installed with Simplicity Studio.

## 2. Security

- This bootloader is designed to be small and easy to use. It doesn't have any security properties or features.
- The bootloader and the C2 debug interface should both be considered as vulnerable from a security standpoint. For example, the flash read or verification functionality of either interface could be compromised, or the flash write functionality could be used to inject a "tiny flash dump" routine to extract the code contents. Both interfaces should be explicitly disabled prior to exposing the device to an untrusted environment if code security is a concern.
- Devices are shipped from the factory with the bootloader enabled and the flash unlocked.
- The bootloader supports two security controls: one to program the "Lock Byte" in flash and the other to program the "Bootloader Signature" byte. The Lock Byte can be used to protect flash contents from being read across the C2 debug interface. **The Lock Byte value has no effect on the bootloader's behavior or capabilities.** The C2 debug interface treats the bootloader like "normal" user code.
- Writing a non-0xA5 value to the "Bootloader Signature" byte will disable the bootloader. After this occurs, none of the bootloader commands are accessible and it is no longer possible for the device to enter bootload mode from a device reset or from an application command.
- Both the "Lock Byte" and the "Bootloader Signature" byte can be reset to 0xFF only by performing a Device Erase via the C2 debug interface.
- The bootloader resides in the "Lock Byte" page of flash which has some unique properties. Code running on the device can write to the "Lock Byte" page but cannot erase that page. This means that the bootloader cannot update itself and cannot be updated by any code running on the device. However, the bootloader can modify itself and can be modified by other code. Updating the bootloader or repairing a damaged bootloader can only be accomplished via the C2 debug interface.
- Bootloaders that are larger than one flash page (such as USB) will occupy the top-most pages of user flash. The non-Lock Byte pages can be erased by the bootloader or other code running on the device.

### 3. Getting Started with the USB or UART Bootloader

These steps assume the use of a Starter Kit with a Windows PC. The Python tools can be used under Linux with some modifications. These are described in a Knowledge Base article on the Silicon Labs website called, "[How to use EFM8 UART Bootloader on Linux.](#)"

These steps also assume that the AN945SW zip file has been downloaded to the PC and extracted, or that the files are accessed using Simplicity Studio. This zip file can be found on the Silicon Labs website (<http://www.silabs.com/documents/public/example-code/AN945SW.zip>).

- Download the Bootloader to the Device

If the bootloader isn't already on the device, download the bootloader to the device using Simplicity Studio using the steps below. Devices with a date code in the top marking later than the date listed in [Table 1.1 EFM8 Device Bootloader Support on page 2](#) can support the bootloader and should have the bootloader pre-installed. Devices with revisions or date codes prior to this will not function with the bootloader.

- Open Simplicity Studio.
- Connect the Starter Kit to the PC.
- Move the kit switch to the **[AEM]** position.
- Click the **[Refresh detected hardware]** button in the left pane of Simplicity Studio. The kit should appear in the **[Detected Hardware]** area.
- Click on the kit and click the **[Flash Programmer]** tile in the **[Tools]** area of Simplicity Studio.
- Click the **[Erase]** button.
- Click the **[Browse]** button, navigate to the pre-compiled bootloader hex file for the kit device, click **[Open]**, and click **[Program]**.
- Download an Application to the Device using the Bootloader
  - Since flash is initially empty (from the erase command in the previous step), the device will automatically start in bootload mode.
  - For USB bootloader devices (e.g. EFM8UB1), connect a second USB cable to the device micro-USB connector on the bottom edge of the Starter Kit.
  - Open a command-line window in the same directory as the `efm8load.exe` Python-based executable host tool.
  - Type `efm8load.exe`, followed by the name of the bootload record. For example:

```
efm8load.exe EFM8UB1_RainbowBlinky.efm8
```

This will download the file to the Starter Kit and automatically run the application.

**Note:** Note that the bootload record is created by the `hex2boot.exe` host-side tool. More information on this tool can be found in [Section 6. Using the Host-Side Tools.](#)

## 4. Getting Started with the SMBus Bootloader

These steps assume the use of an EFM8LB1 Starter Kit (EFM8LB1-SLSTK2030A) with a Windows PC.

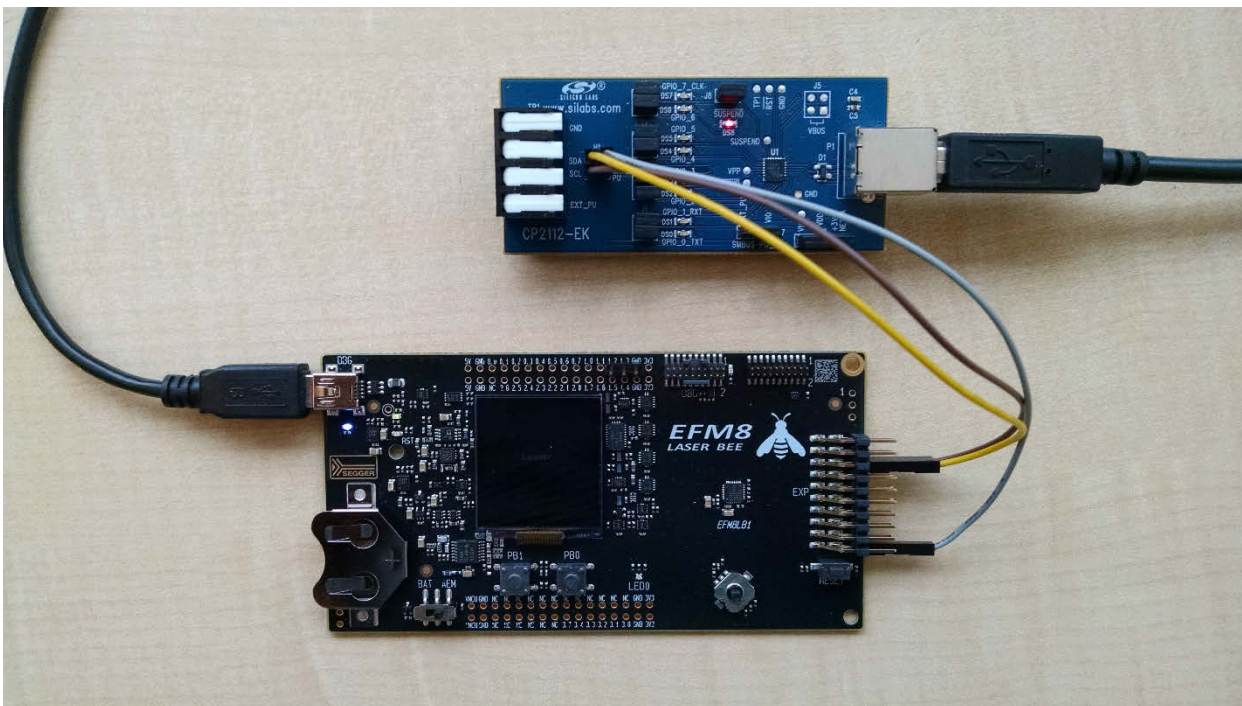
These steps also assume that the AN945SW zip file has been downloaded to the PC or that the files are accessed using Simplicity Studio. This zip file can be found on the Silicon Labs website (<http://www.silabs.com/documents/public/example-code/AN945SW.zip>).

Since SMBus is not a bus type readily available as a port on a PC, these steps use a CP2112 USB-to-SMBus evaluation board for demonstration purposes.

- Connect the EFM8LB1 STK to a CP2112 EK.
- Install three jumper wires between the EFM8LB1 STK and the CP2112 EK to connect the SDA, SCK and GND signals. Use the following table as a guide. Also, make sure to install a jumper on J7 pins 1:2 of the CP2112 EK to enable the SMBus pull-up resistors.
- Connect both boards to a PC with a USB cable.

**Table 4.1. EFM8LB1 STK and CP2112 EK Connections**

| EFM8LB1 STK EXP Header        | CP2112 EK H1 Header |
|-------------------------------|---------------------|
| Pin 16 – SMBus SDA (MCU P1.2) | Pin 3 – SDA         |
| Pin 15 – SMBus SCL (MCU P1.3) | Pin 4 – SCL         |
| Pin 1 – Ground                | Pin 1 – GND         |



- Download the Bootloader to the Device

If the SMBus bootloader isn't already on the device, download the bootloader to the device using Simplicity Studio with the steps below.

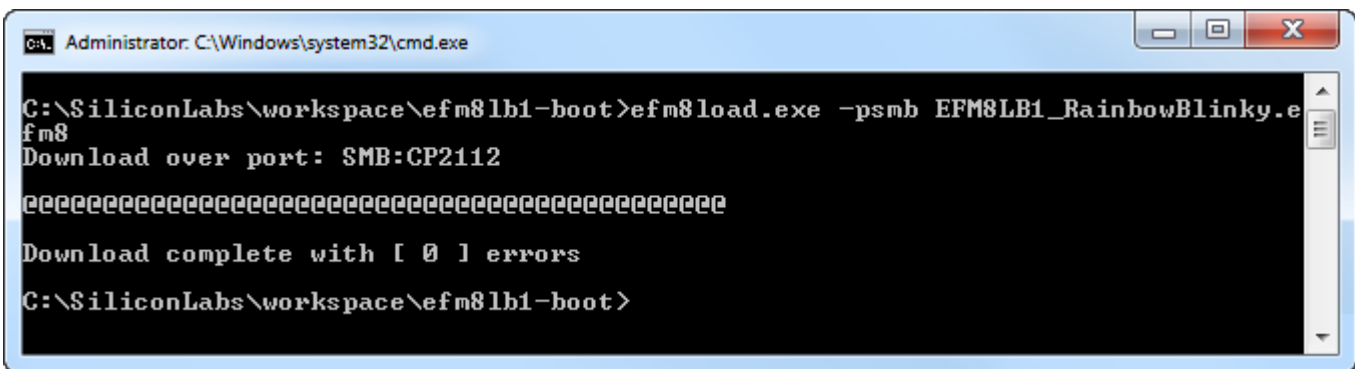
**Note:** The pinout for the SMBus bootloader for the EFM8LB1-SLSTK2030A board is different than the two SMBus bootloaders pre-loaded into EFM8LB1 devices. The prebuilt bootloader hex image for this board is [EFM8LB1\_SMB0\_BL\_STK\_Pinout.hex], and is located in the [StarterKitHexfiles] subdirectory of the AN945SW software package. Do not use the prebuilt bootloader hex images from the [ProductionDeviceHexfiles] subdirectory.

- Open Simplicity Studio.
- Connect the Starter Kit to the PC.
- Move the kit switch to the [AEM] position.
- Click the [Refresh detected hardware] button in the left pane of Simplicity Studio. The kit should appear in the [Detected Hardware] area.
- Click on the kit and click the [Flash Programmer] tile in the [Tools] area of Simplicity Studio.
- Click the [Erase] button.
- Click the [Browse] button, navigate to the pre-compiled bootloader hex file for the kit device (EFM8LB12F64E\_QFN32), click [Open], and click [Program].
- Download an Application to the Device using the Bootloader
  - Since flash is initially empty (from the erase command in the previous step), the device will automatically start in bootload mode.
  - Open a command-line window in the same directory as the efm8load Python host tool ([Start]>[Run]>[command] in Windows).
  - Type [efm8load.exe], followed by the [-psmb] argument (indicates SMBus bootloader) and the name of the bootload record. For example:

```
efm8load.exe -psmb EFM8LB1_RainbowBlinky.efm8
```

This will download the file to the Starter Kit through the CP2112 EK. The efm8load utility uses a default 100 kHz clock rate. Use the [-baud] argument to adjust the clock rate. With the RainbowBlinky application running on the STK, press the [PB0] button to start the bootloader.

**Note:** The bootload record is created by the hex2boot.exe host-side tool. More information on this tool can be found in Section 6. Using the Host-Side Tools.



## 5. Feature Overview

### 5.1 System Architecture

As shown in the diagram below, the EFM8 bootloader tool will read an image file (hex or binary) and translate it into a series of bootloader commands. These commands are formatted into boot records, which are stored in a file or transmitted directly to the EFM8 device. The same binary record format is used whether saving to a file or sending over the bootloader transport. The bootloader commands are designed to not return any data, and they contain all the information needed to execute the command and determine success or failure. To implement an embedded host, read each record from the file, send it to the EFM8 device, and wait for acknowledgment. Because the boot record file is binary, it adds little overhead to the image and should be easy to embed.

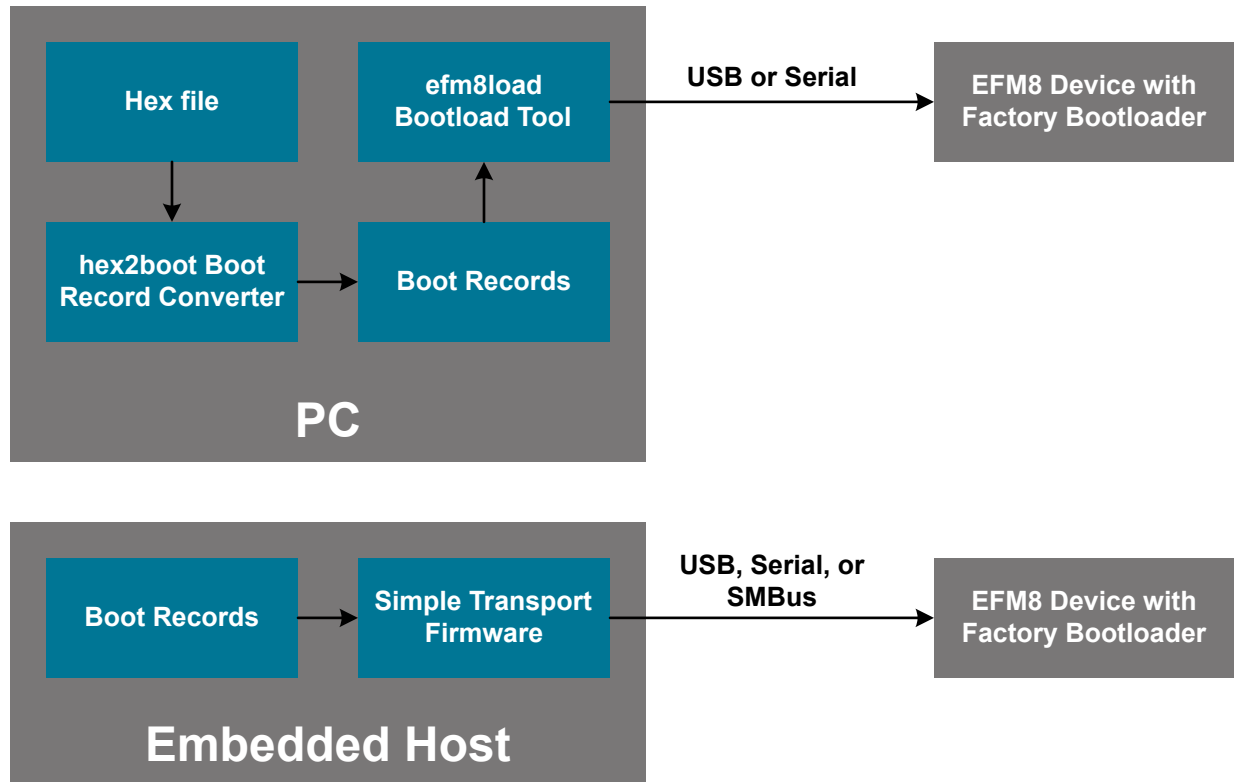


Figure 5.1. System Architecture

## 5.2 Bootloader Overview

All devices come pre-programmed with a UART (~500 bytes), SMBus (~500 bytes), or USB HID (~1.5 kB) bootloader. This bootloader resides in the code security page. If it's too large to fit in the code security page, it will also occupy the last pages of code flash. It can be erased if it is not needed.

The byte before the Lock Byte is the Bootloader Signature Byte. Setting this byte to a value of 0xA5 indicates the presence of the bootloader in the system. Any other value in this location indicates that the bootloader is not present in flash.

When a bootloader is present, the device will jump to the bootloader vector after any reset, allowing the bootloader to run. The bootloader then determines if the device should stay in bootload mode or jump to the reset vector located at 0x0000. When the bootloader is not present, the device will jump to the reset vector of 0x0000 after any reset.

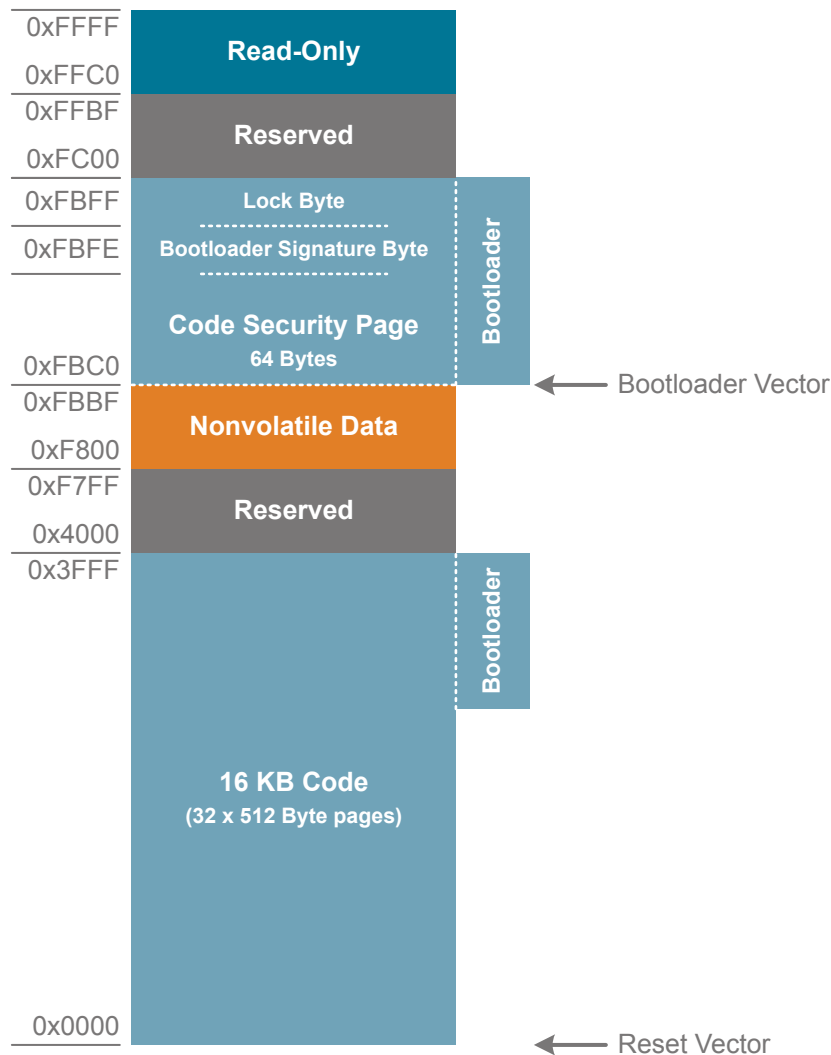


Figure 5.2. Example Flash Memory Map with Bootloader — EFM8UB1 16 KB Devices



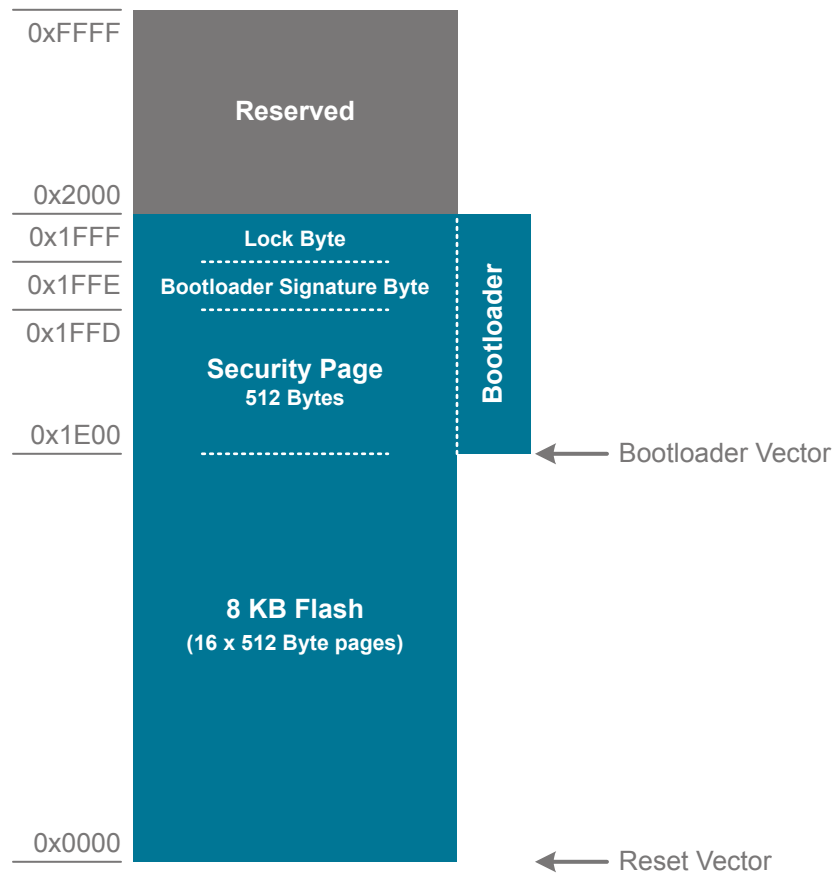


Figure 5.3. Example Flash Memory Map with Bootloader — EFM8SB1 8 KB Devices

### 5.3 Bootloader Feature Specifics

This section discusses the specific features of the bootloader firmware.

Note that bootloader source code and pre-built hex files are available in Simplicity Studio using the [Software Examples] tile or available on the Silicon Labs website (<https://www.silabs.com/8bit-appnotes>). This enables the bootloader to be reprogrammed on any device if it's accidentally erased, and the bootloader can be customized for any application as needed.

### 5.3.1 Entering Bootload Mode

- On any reset, the bootloader will start if flash address 0x0000 is 0xFF (i.e. first byte of the reset vector is not programmed). This ensures new or erased parts start the bootloader for production programming. For robustness, the bootloader erases flash page 0 first and writes flash address 0x0000 last. This ensures that a device will restart in bootloader mode if the previous bootload operation was not completed successfully.
- To start the bootloader on demand, the application firmware can set the signature value 0xA5 in R0 in Bank 0 (data address 0x00) and then initiate a software reset of the device. If the bootloader sees a software reset with the signature value in R0, it will start bootloader execution instead of jumping to the application.
- To provide fail-safe operation in case the application is corrupted, the bootloader starts on either power-on reset (POR) or pin resets if a pin is held low for longer than 50  $\mu$ s. A full list of entry pins for each device and package is available in [Table 5.1 Summary of Pins for Bootload Mode Entry on page 10](#). The pin for this bootloader entry method can also be found by looking at the `efm8_device.h` file in the bootloader source code. There is no option to disable this entry method.

**Table 5.1. Summary of Pins for Bootload Mode Entry**

| Family  | Package | Pin for Bootload Mode Entry |
|---------|---------|-----------------------------|
| EFM8UB1 | QFN28   | P3.0 / C2D                  |
|         | QSOP24  | P2.0 / C2D                  |
|         | QFN20   | P2.0 / C2D                  |
| EFM8UB2 | QFP48   | P3.7                        |
|         | QFP32   | P3.0 / C2D                  |
|         | QFN32   | P3.0 / C2D                  |
| EFM8UB3 | QFN24   | P2.0 / C2D                  |
|         | QSOP24  | P2.0 / C2D                  |
|         | QFN20   | P2.0 / C2D                  |
| EFM8BB1 | QSOP24  | P2.0 / C2D                  |
|         | QFN20   | P2.0 / C2D                  |
|         | SOIC16  | P2.0 / C2D                  |
| EFM8BB2 | QFN28   | P3.0 / C2D                  |
|         | QSOP24  | P3.0 / C2D                  |
|         | QFN20   | P2.0 / C2D                  |
| EFM8BB3 | QFN32   | P3.7 / C2D                  |
|         | QFP32   | P3.7 / C2D                  |
|         | QFN24   | P3.0 / C2D                  |
|         | QSOP24  | P3.0 / C2D                  |
| EFM8SB1 | QFN20   | P2.7 / C2D                  |
|         | QFN24   | P2.7 / C2D                  |
|         | QSOP24  | P2.7 / C2D                  |
|         | CSP16   | P2.7 / C2D                  |
| EFM8SB2 | QFN32   | P2.7 / C2D                  |
|         | QFN24   | P2.7 / C2D                  |
|         | QFP32   | P2.7 / C2D                  |

| Family  | Package | Pin for Bootload Mode Entry |
|---------|---------|-----------------------------|
| EFM8LB1 | QFN32   | P3.7 / C2D                  |
|         | QFP32   | P3.7 / C2D                  |
|         | QFN24   | P3.0 / C2D                  |
|         | QSOP24  | P3.0 / C2D                  |

### 5.3.1.1 Rainbow Blinky Example

The **[Rainbow Blinky]** (or **[PwmBlinky]** for EFM8SB1) example in the **[Demos]** folder available using the **[Software Examples]** tile in Simplicity Studio demonstrates how to create an application that can enter bootload mode using the software reset entry method.

In addition to the standard Rainbow Blinky operation, this application looks to see if PB0 on the Starter Kit is pressed. If it is, then the example displays **[Loader]** on the LCD screen, writes the signature value of 0xA5 to the R0 address (data address 0x00), and initiates a software reset. When this occurs, the bootloader sees the signature value at the data address and will remain in bootload mode. The code to do this is very simple and is shown below:

```
// Start the bootloader if PB0 is pressed
if (BSP_PB0 == BSP_PB_PRESSED)
{
    // Print "Loader" to the screen
    colorIndex = 8;
    DrawColorName();

    // Write R0 and issue a software reset
    *((uint8_t SI_SEG_DATA *)0x00) = 0xA5;
    RSTSRC = RSTSRC_SWRSF__SET | RSTSRC_PORSF__SET;
}
```

### 5.3.2 Communication Interface

- The bootloader communication interface is polled, since the bootloader does not have access to the interrupt vectors on page 0.
- Communication uses a custom binary protocol. The protocol is common across all communication interfaces and is described in more detail in Section 7. [Bootloader Protocol](#).

#### UART

- The UART bootloader supports an autobaud mechanism that uses 0xFF as the autobaud training byte. To save space, the mechanism will only support the highest T1 prescaler range, which will limit the lowest baudrate to 115200 on UBx devices and 57600 on all other devices. The maximum baud rate for all devices is 460800.
- To provide support for a consistent range of baud rates, the bootloader will select HFOSC0 for the system clock, which is 24.5 or 20 MHz for most parts.

#### USB

- The USB bootloader uses HID class. This class does not require a driver installation and enables a smaller implementation. To save additional space, the bootloader does not support USB string descriptors (e.g. serial number). This means that only one USB bootloader device can be connected to a PC at a time, since the PC cannot distinguish between multiple devices.
- The USB bootloader passes the chapter 9 compliance test, but does not support suspend and resume.
- The USB bootloader uses the Silicon Labs VID (0x10C4), and each device family will have a unique PID, described in Section [Table 5.2 Summary of USB Bootloader PIDs on page 12](#). The PID is not customizable.

**Table 5.2. Summary of USB Bootloader PIDs**

| Device Family | USB Product ID (PID) |
|---------------|----------------------|
| EFM8UB1       | 0xEAC9               |
| EFM8UB2       | 0xEACA               |
| EFM8UB3       | 0xEACB               |

#### SMBus

- The SMBus bootloader is a slave interface only.
- The byte aligned slave address is 0xF0.
- The SMBus bootloader uses the SMB peripheral interface module (not the I2CSLAVE peripheral interface module).
- At SCK clock rates faster than 100 kHz, the SMBus slave may employ clock stretching by holding the SCK signal low after the acknowledge bit. This clock stretching is necessary to give the bootloader sufficient time to complete the flash write cycle. To prevent a long clock stretch during flash erase, it is necessary to use the [-e1] option when using the hex2boot utility to create the boot image. See [6.1.2 Hex2Boot Command-line Arguments](#) for more information about this new hex2boot option.
- The bootloader erase and verify commands can take tens of milliseconds to complete. While these commands are processing, the bootloader is unable to service the SMBus slave. As part of the bootloader protocol, the SMBus master must query the slave to receive the result of the last command. To let the master know that it is busy processing the last command, the bootloader uses the ACK polling mechanism. This is the same mechanism I2C EEPROM devices use during flash write cycles. The primary feature of ACK polling is that the SMBus bootloader will NAK its slave address while it is processing the current boot record. The master polls the bootloader by attempting a master read transfer. If the slave address is NAK'd, the master knows the bootloader is still busy with the last command. The master should continue to retry the transfer until the slave address is ACK'd and the read transfer is completed. At that point, the master can proceed by writing the next boot record.
- The EFM8LB1 devices with factory-loaded SMBus bootloaders are available in two SMBus bootloader pin assignments:
  - EFM8LB1xFxxES0 devices: SDA:P0.2, SCL:P0.3
  - EFM8LB1xFxxES1 devices: SDA:P0.2, SCL:P0.4

However, the EFM8LB1-SLSTK2030A board uses a nonstandard pinout (SDA: P1.2 , SCL:P1.3). For this reason, the SMBus bootloader has three separate project files that build the bootloader for the three pinout variants.

### 5.3.3 Memory

- All bootloader memory can be recovered for use by the application by erasing the device. The only exception to this is the Bootloader Signature Byte, which should not be programmed to 0xA5 if the bootloader is not present.
- Devices are shipped from the factory with the flash unlocked.
- The bootloader provides a means to write the Lock Byte and Bootloader Signature bytes, which can be used to lock the flash and disable the bootloader respectively.
- The bootloader can be disabled by writing a 0x00 to the Bootloader Signature byte. By doing this, the MCU jumps directly to the application and never calls the bootloader. As a result, all of the bootloader entry methods are also disabled.
- For additional information, see the [Security](#) chapter of this note.

## 6. Using the Host-Side Tools

The host-side bootloader tools consist of two parts: `hex2boot`, which creates a bootload record from a hex file, and `efm8load`, which downloads the bootload record to the EFM8 device.

### 6.1 Hex2Boot — Hex to Bootload Record Converter

#### 6.1.1 Introduction

The **[Hex2Boot]** tool converts a standard Intel hex file that is output upon a successful build by 8051 build tools to a bootload record. The bootload record is a pure binary file composed of the bootloader commands described in Section 7. [Bootloader Protocol](#).

**Note:** The **[Hex2Boot]** utility uses the Python module **[IntelHex]**, and so the usage and distribution of **[Hex2Boot]** is also governed by the **[IntelHex]** license. See the `intelhex-LICENSE.txt` file (included in the AN945SW package) for more information.

## 6.1.2 Hex2Boot Command-line Arguments

A list of [Hex2Boot] command-line arguments can be seen at any time using `hex2boot.exe -h` on the command line. More information on the bootloader commands referenced here can be found in Section 7. [Bootloader Protocol](#). The available commands for `hex2boot` are:

### **-h, --help**

This command shows the help message and exits.

### **-v, --version**

This command displays the `hex2boot` program version and exits.

### **-o OUT, --out OUT**

The OUT parameter is the boot record output file and is required.

### **-b {0,1}, --bank {0,1}**

This parameter specifies the bank for the download as a parameter for the bootloader `Setup` command with a default equal to 0. This parameter is only currently useful on EFM8SB2 devices that have the standard flash and scratchpad areas.

### **-i [ID [ID ...]], --id [ID [ID ...]]**

This parameter specifies the id for the `Identify` bootloader command, with a default of None. A bootload record can specify one id, which means the device must exactly match the id for it to be programmed. A bootload record can also specify several id's in case the record should match multiple devices, or no id in the case where the bootload record should download to any device.

### **-l LOCK, --lock LOCK**

This parameter specifies the lock value (default = None) for the `Lock` bootloader command. The LOCK parameter is 16-bits with the MSB holding the signature byte and the LSB holding the flash lock byte. Use a value of 0xFF for either byte to not change its value. The input can be decimal (65535) or hexadecimal (0xFFFF). Silicon Labs recommends that the host lock the device flash and disable the bootloader after production programming by using a LOCK setting of 0x0000. See the [Security](#) chapter of this note for additional details.

### **-m {bb2,sb2,ub1,ub3}, --map {bb2,sb2,ub1,ub3}**

This parameter indicates a device with a special memory map (default = None). The EFM8BB2, EFM8SB2, EFM8UB1, and EFM8UB3 devices have special flash pages compared to the rest of the EFM8 family. The corresponding `[-m]` parameter should be set when programming these device families. If programming a device family other than {bb2,sb2,ub1, or ub3}, the `[-m]` or `[-map]` parameter should be omitted.

### **-s ADDR, --start ADDR**

This parameter indicates the starting address for the bootload record (default = 0). In the cases where there are empty areas of flash, the hex file may still contain blocks of 0xFF values to program. This parameter coupled with the `-t ADDR` parameter enables these blocks of 0xFF values to be removed. The input can be decimal (5200) or hexadecimal (0x0400).

### **-t ADDR, --top ADDR**

This parameter indicates the top or ending address for the bootload record (default = 65535). In the cases where there are empty areas of flash, the hex file may still contain blocks of 0xFF values to program. This parameter coupled with the `-s ADDR` parameter enables these blocks of 0xFF values to be removed. The input can be decimal (5200) or hexadecimal (0x0400).

### **-w, --wait**

This parameter specifies that the device should remain in bootload mode after the application download completes. This enables multiple bootload records to be downloaded in the same bootload session.

### **-e {0,1,2}, --erase {0,1,2}**

Version 1.10 of `hex2boot` utility adds a new option for controlling how the erase command is generated. The erase command can include data to write to flash. This is the default behavior as it produces the fewest number of boot records. However, when using the SMBus bootloader, erase commands that carry data produce a several millisecond long clock stretch while the erase occurs. To avoid this long clock stretch, `hex2boot` can generate erase records that do not include any write data. The `[-e0]` option tells `hex2boot` to write data without erasing the flash first. Only use the `[-e0]` option when the target flash is guaranteed to be blank. The `[-e1]` option instructs `hex2boot` to create erase commands without any data. Use this option when generating an image to use with the SMBus bootloader. The `[-e2]` option causes `hex2boot` to create erase commands that include write data. This is the default behavior as it generates the fewest number of boot records.

If a hex file is not provided as an input to the `hex2boot` program, the utility will create a record that erases the specified address range.

### 6.1.3 Evaluation and Development Usage Examples

Except where noted, these examples leave the bootloader enabled and leave the device flash unlocked. These settings are recommended only for development, evaluation, and testing. See the [Security](#) chapter of this note for additional details.

To create a standard bootload record for an EFM8 device leaving the debug interface and the bootloader unlocked (not recommended):

```
hex2boot.exe input_file.hex -o Filename.efm8
```

To create a standard bootload record for an EFM8SB2 device leaving the debug interface and the bootloader unlocked (not recommended):

```
hex2boot.exe input_file.hex -o Filename.efm8 -m sb2
```

To create a bootload record that locks all pages in a device but leaves the bootloader enabled (not recommended):

```
hex2boot.exe input_file.hex -o Filename.efm8 -l 0xFF00
```

To create a bootload record that programs part of the application space:

```
hex2boot.exe input_file.hex -o Filename.efm8 -s 0 -t 4000
```

### 6.1.4 Production Usage Examples

These examples disable the bootloader after factory programming and lock the device flash. These settings are recommended for production devices. See the [Security](#) chapter of this note for additional details.

To create a standard bootload record for an EFM8 device and disable the bootloader and lock the device flash:

```
hex2boot.exe input_file.hex -o Filename.efm8 -l 0x0000
```

To create a standard bootload record for an EFM8SB2 device and disable the bootloader and lock the device flash:

```
hex2boot.exe input_file.hex -o Filename.efm8 -m sb2 -l 0x0000
```

To create a bootload record that programs part of the application space and disables the bootloader and locks the device flash:

```
hex2boot.exe input_file.hex -o Filename.efm8 -s 0 -t 4000 -l 0x0000
```

## 6.2 EFM8Load — Bootload Record Downloader

### 6.2.1 Introduction

The `efm8load` Python script downloads the bootload record created by `hex2boot` to the target device. This downloader is intended to be simple enough that it could be recreated on an embedded host. Source files are available for this tool, and [Section 8. Modifying and Rebuilding the Bootloader](#) discusses how to rebuild the Python scripts.

The `efm8load` utility will select the USB HID and J-Link (from the Starter Kit Virtual COM Port) communication options automatically, and a single USB HID or J-Link board should be connected. If multiple are connected, the specific port can be chosen using the command-line parameters.

**Note:** The `efm8load` utility uses the Python module `pySerial`, and so the usage and distribution of `efm8load` is also governed by the `pySerial` license. See the `pyserial-LICENSE.txt` file (included in the AN945SW package) for more information.



## 6.2.2 EFM8Load Command-line Arguments

A list of `efm8load` command-line arguments can be seen at any time using `efm8load.exe -h` on the command line. The available commands for `efm8load` are:

### **-h, --help**

This command shows the help message and exits.

### **--version**

This command displays the `efm8load` program version and exits.

### **-b BAUD, --baud BAUD**

Specify the baud rate that should be used for UART communication (default = 115200). Note that the UART bootloader uses auto-baud, so any baud rate between 115200 and 460800 can be used.

### **-p PORT, --port PORT**

If multiple UART bootloader devices are connected, this parameter can select the COM port to specify a specific device. If the device is connected using the Starter Kit J-Link Virtual COM Port (VCP), then this parameter is not required. Note that the USB bootloader only supports one device connected to the PC at a time.

### **-t, --trace**

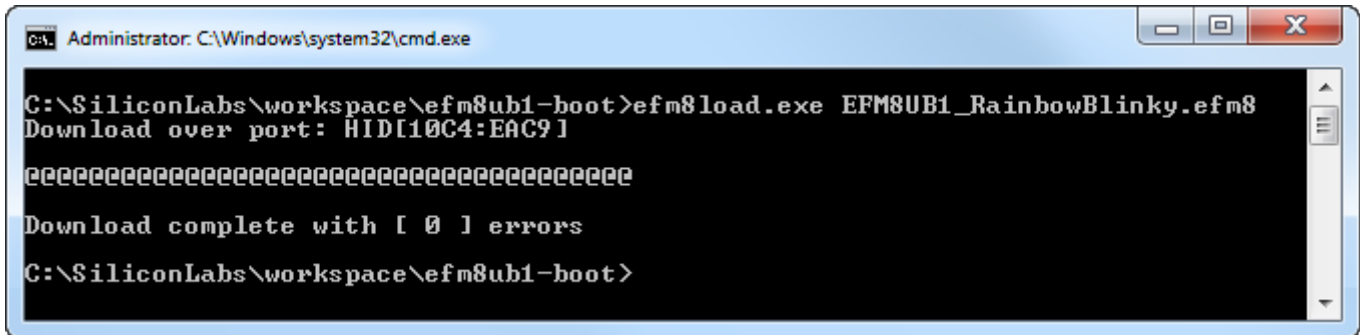
The trace parameter enables verbose output for the download process. This output will indicate a successful set of data is programmed with a `[@]` symbol. Errors in the download will also be highlighted for each data set.

### 6.2.3 Usage Examples

To do a simple bootloader record download, use `efm8load` as follows:

```
efm8load.exe Filename.efm8
```

The successful output for this command is shown in the figure below:

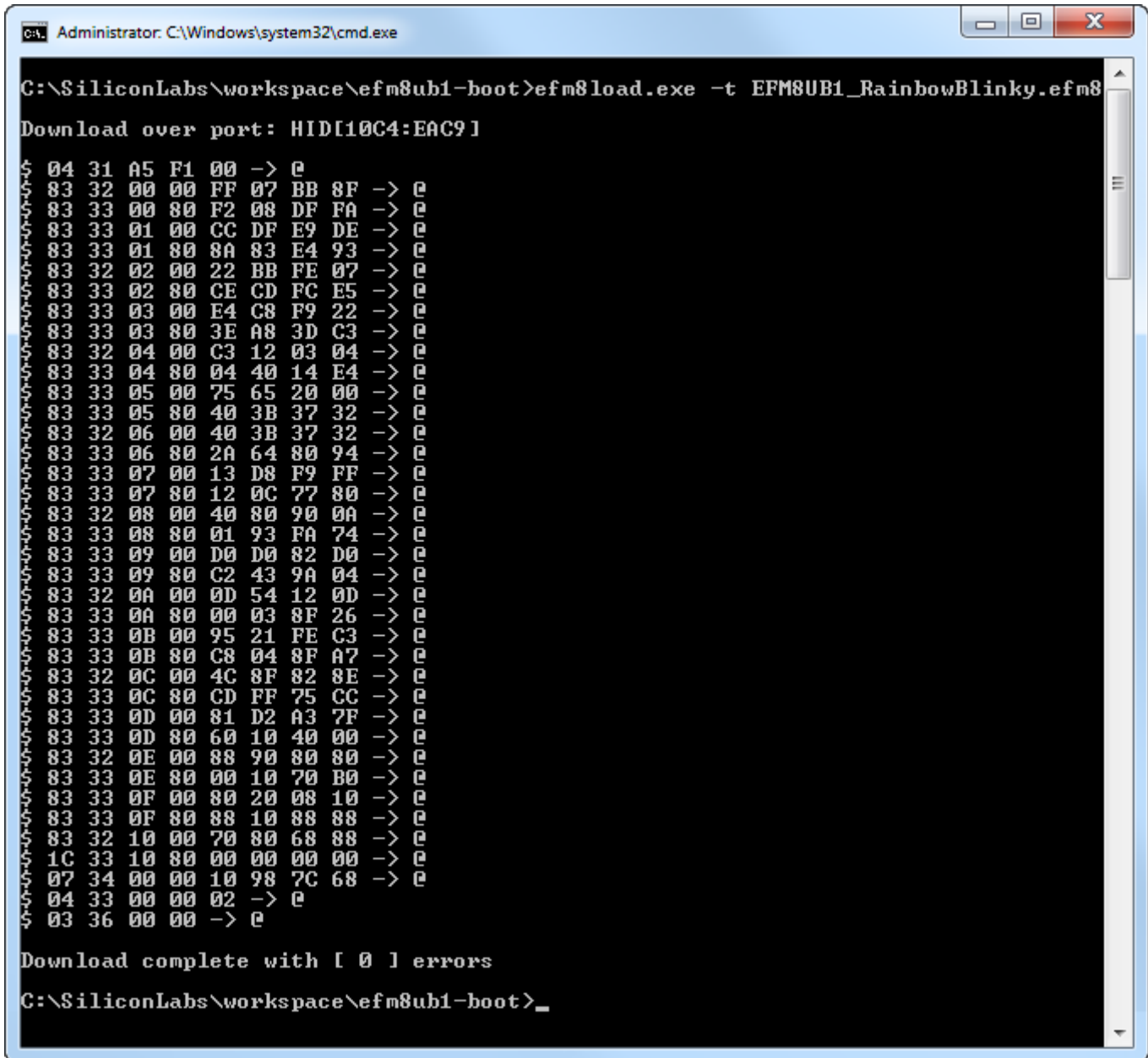


**Figure 6.1. Basic `efm8load` Download Output**

To output detailed information on the programming session, use the `[-t]` parameter:

```
efm8load.exe -t Filename.efm8
```

The successful output for this operation is shown below:



```
Administrator: C:\Windows\system32\cmd.exe
C:\SiliconLabs\workspace\efm8ub1-boot>efm8load.exe -t EFM8UB1_RainbowBlinky.efm8
Download over port: HID[10C4:EAC9]
04 31 A5 F1 00 -> 0
83 32 00 00 FF 07 BB 8F -> 0
83 33 00 80 F2 08 DF FA -> 0
83 33 01 00 CC DF E9 DE -> 0
83 33 01 80 8A 83 E4 93 -> 0
83 32 02 00 22 BB FE 07 -> 0
83 33 02 80 CE CD FC E5 -> 0
83 33 03 00 E4 C8 F9 22 -> 0
83 33 03 80 3E A8 3D C3 -> 0
83 32 04 00 C3 12 03 04 -> 0
83 33 04 80 04 40 14 E4 -> 0
83 33 05 00 75 65 20 00 -> 0
83 33 05 80 40 3B 37 32 -> 0
83 32 06 00 40 3B 37 32 -> 0
83 33 06 80 2A 64 80 94 -> 0
83 33 07 00 13 D8 F9 FF -> 0
83 33 07 80 12 0C 77 80 -> 0
83 32 08 00 40 80 90 0A -> 0
83 33 08 80 01 93 FA 74 -> 0
83 33 09 00 D0 D0 82 D0 -> 0
83 33 09 80 C2 43 9A 04 -> 0
83 32 0A 00 0D 54 12 0D -> 0
83 33 0A 80 00 03 8F 26 -> 0
83 33 0B 00 95 21 FE C3 -> 0
83 33 0B 80 C8 04 8F A7 -> 0
83 32 0C 00 4C 8F 82 8E -> 0
83 33 0C 80 CD FF 75 CC -> 0
83 33 0D 00 81 D2 A3 7F -> 0
83 33 0D 80 60 10 40 00 -> 0
83 32 0E 00 88 90 80 80 -> 0
83 33 0E 80 00 10 70 B0 -> 0
83 33 0F 00 80 20 08 10 -> 0
83 33 0F 80 88 10 88 88 -> 0
83 32 10 00 70 80 68 88 -> 0
1C 33 10 80 00 00 00 00 -> 0
07 34 00 00 10 98 7C 68 -> 0
04 33 00 00 02 -> 0
03 36 00 00 -> 0
Download complete with [ 0 ] errors
C:\SiliconLabs\workspace\efm8ub1-boot>_
```

Figure 6.2. efm8load Download Output with Trace

If multiple UART bootloader devices are connected to the PC, select a specific COM port using the [-p] parameter:

```
efm8load.exe -p COM3 Filename.efm8
```

## 7. Bootloader Protocol

Bootloader commands are formatted into a binary record format. This simple and consistent format, which is inspired by hex records, is designed to make parsing easy. By including a frame start byte and an explicit length field, a file parser or communication transport code can be written without knowledge of the underlying commands. Also, the format allows command parameters to be added at a future date without impacting backward compatibility. The following diagram shows the format for the binary boot record:

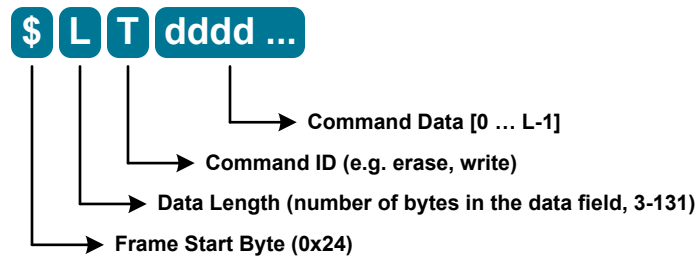


Figure 7.1. Binary Boot Record Format

The communication protocol operates as follows:

1. The host sends one complete record at a time and then waits for acknowledgment from the bootloader.
2. The bootloader processes the record and sends a one-byte response that indicates if the command was successful or failed.

This simple command/response handshake is simple to implement and provides flow control for transports that don't already have it (i.e. UART).

The SMBus bootloader uses this same protocol. However, there are several behaviors specific to the SMBus bootloader.

- Each boot record may be sent as one master write transfer or split into multiple smaller transfers. The size of each master write transfer does not matter to the bootloader. However, the master must perform a one byte read after each boot record is written to receive the return code. The bootloader will not process another boot record until the previous return code is read.
- To receive the return code, the master must use the ACK polling mechanism described in Section 5.3.2 [Communication Interface](#). For most commands, the response is returned on the first poll. For erase and verify commands, it may be tens of milliseconds before the response is returned.

## 7.1 Command Details

Each boot record carries one bootloader command. None of the commands return any data. Instead, the commands are designed to be acknowledged with a simple one-byte response. This makes it possible to implement an embedded host without any knowledge of the individual commands.

The command set is minimalist to keep the implementation small. There is no read command, and verification occurs through a CRC16 mechanism. In the following descriptions, data payload fields are shown in brackets, and the size of the field is given after the colon. Fields that are 2 bytes (16 bits) wide are sent in big-endian (i.e. MSB first) order.

**Note:** If the bootloader receives an unknown command, it will respond with the version of the bootloader. The initial version is 0x90, corresponding to version 1.0.

### Identify 0x30 — [id:2]

This optional command is normally the first command sent. It is used to confirm that the boot image is compatible with the target. The id is the device and derivative ID's concatenated together [device\_id:derivative\_id], and these ID's can be found in the device Reference Manual. A BADID error (0x42) is returned if the id field does not match the target id. If a boot image is compatible with multiple targets, this command can be resent with different id's until an ACK (0x40) is received.

### Setup 0x31 — [keys:2, bank:1]

This command must be sent once before any command that modifies or verifies flash. It passes the flash keys to the bootloader and selects the active flash bank. The keys parameter for all devices is 0xA5F1. The bank parameter should be set to 0x00 for all parts except EFM8SB2, where it can be used to select scratchpad flash. For the SB2 devices, a bank value of 0x00 selects user flash, and 0x01 selects scratchpad flash. This command always returns ACK (0x40).

### Erase 0x32 — [addr:2, data:0-128]

The erase command behaves the same as the write command except that it erases the flash page at the desired address before writing any data. To perform a page erase without writing data, simply do not include data with the command. The data range of an erase command must not cross a flash page boundary, as the bootloader is not aware of page boundaries and only erases the flash page of the starting address of the command. A RANGE error (0x41) is returned if the targeted address range cannot be written by the bootloader.

### Write 0x33 — [addr:2, data:1-128]

Writes the payload data to flash starting at the indicated address. Does not erase the flash before writing. A RANGE error (0x41) is returned if the targeted address range cannot be written by the bootloader.

### Verify 0x34 — [addr1:2, addr2:2, CRC16:2]

This command computes a CRC16 (CCITT-16, XModem) over the flash contents starting at addr1 up to and including addr2 and compares the result to CRC16. Returns a CRC error (0x43) if the CRC's do not match.

**Note:** Flash inspection operations, such as the Verify command, present a potential security risk even if they do not return the contents of flash memory directly. These commands should be removed, disabled, or replaced if flash security is a concern. See the [Security](#) chapter of this note for further details.

### Lock 0x35 — [sig:1, lock:1]

This command overwrites the bootloader signature and flash lock bytes with the payload values. Setting the signature to 0xA5 will enable the bootloader, and setting it to 0x00 will permanently disable the bootloader. The signature or lock values are not changed if their corresponding parameter is set to 0xFF, which enables writing the lock byte without changing the signature and vice versa. This command always returns ACK (0x40). For production devices, Silicon Labs recommends disabling the bootloader after production programming and locking the flash memory by setting the signature and lock values both to 0x00. See the [Security](#) chapter of this note for further details.

### RunApp 0x36 — [option:2]

Resets the device in order to start the application. Currently the option field is unused. The command always returns ACK (0x40) and the USB bootloader will delay 100 ms before resetting to give the host time to close the connection.

## 7.2 Command Summary

Here is a summary of all the commands, their length, and the return codes:

**Table 7.1. Bootloader Command Summary**

| Command  | Command Code | Length (bytes) | Return Code   |
|----------|--------------|----------------|---|
| Identify | 0x30         | 3              | ACK (0x40) if sent id matches device id<br>BADID error (0x42) if the id field does not match the target id  |
| Setup    | 0x31         | 4              | always ACK (0x40)   |
| Erase    | 0x32         | 3 min, 131 max | ACK (0x40) normally<br>RANGE error (0x41) if the targeted address range cannot be written by the bootloader |
| Write    | 0x33         | 4 min, 131 max | ACK (0x40) normally<br>RANGE error (0x41) if the targeted address range cannot be written by the bootloader |
| Verify   | 0x34         | 7              | ACK (0x40) if the CRC's do match<br>CRC error (0x43) if the CRC's do not match                              |
| Lock     | 0x35         | 3              | always ACK (0x40)   |
| RunApp   | 0x36         | 3              | always ACK (0x40)   |
| Unknown  | Any          | 1              | bootloader version (initial value is 0x90 corresponding to version 1.0)                                     |

## 8. Modifying and Rebuilding the Bootloader

### 8.1 Bootloader Firmware

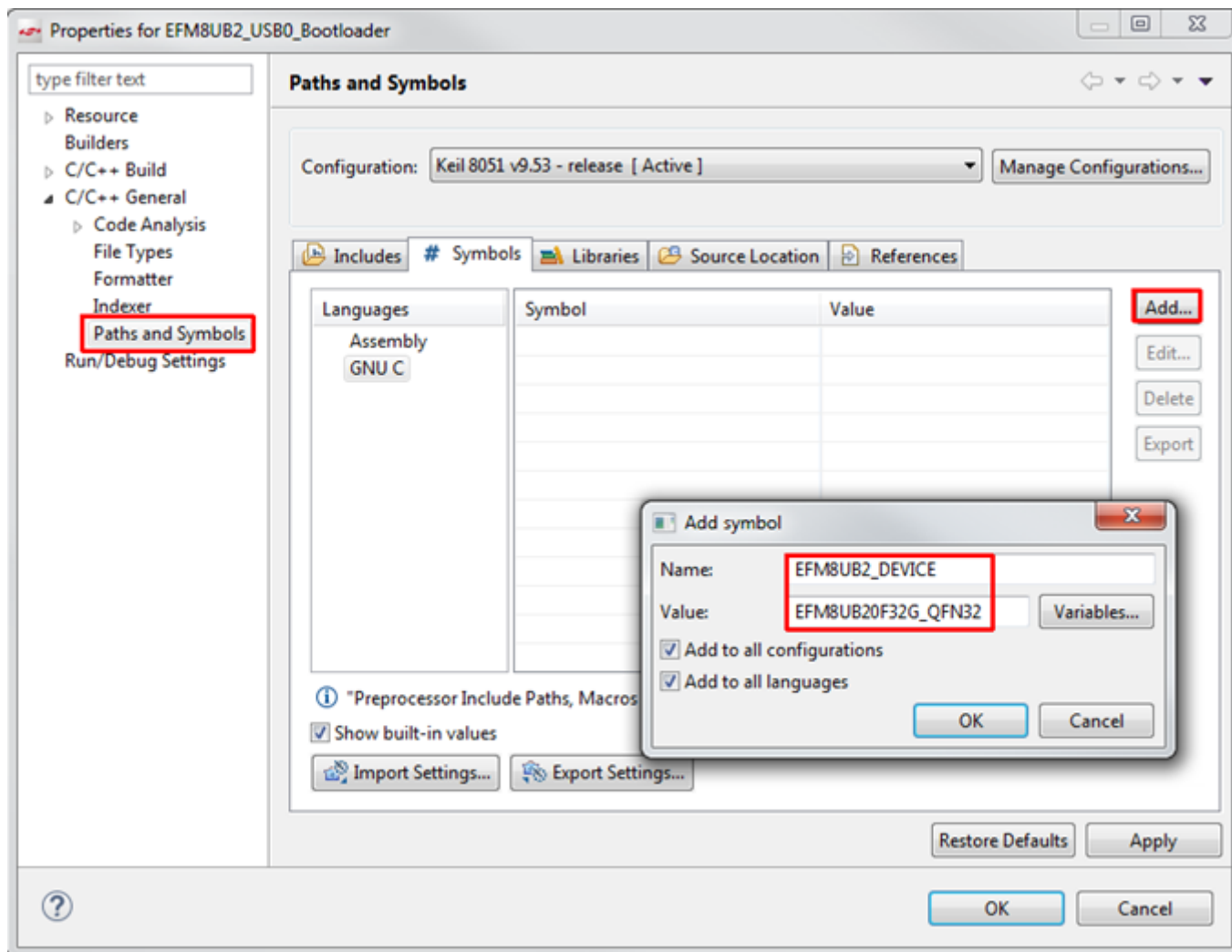
Simplicity Studio includes a project for each device family and supported bootloader type that implements the specific linker command-line flags to properly place the bootloader at the required addresses. The output of these projects is a hex file with the bootloader placed at the proper addresses encoded in the file.

To open one of these projects, open Simplicity Studio, select the desired EFM8 device in the **[Product]** text box or connect the appropriate hardware and select the kit, click the **[Software Examples]** tile, select the bootloader example, and navigate through the wizard to create the project. Choosing the **[Link libraries and copy sources]** or **[Copy contents]** options will ensure that the original bootloader example files remain untouched.

By default, the device defined in a given bootloader's `efm8_device.h` file is the device that is used on the corresponding starter kit (STK) board for that device family. For example, this is an excerpt from the `efm8_device.h` file in the EFM8UB2 USB bootloader example project:

```
// Select the STK device if one has not been specified by the project
#ifndef EFM8UB2_DEVICE
#define EFM8UB2_DEVICE EFM8UB20F64G_QFP48
#endif
```

To build the chosen bootloader example for a different device (i.e. not the device that is used on the corresponding STK board and defined in the `efm8_device.h` file), the user must define a new symbol in their project. This can be done by right-clicking the project in Simplicity Studio and selecting **[Properties]** from the pop-up menu. From there, navigate to **[C/C++ General]>[Paths and Symbols]** and select the **[Symbols]** tab. Then, add the appropriate symbol name and value for the desired device. The complete list of supported devices for a given device family is in the `SI_EFM8xxxx_Devices.h` header file in the EFM8 SDK. For example, the following screenshot shows how to configure to build for the **[EFM8UB20F32G\_QFN32]** device.



**Figure 8.1. Reconfiguring the EFM8UB2 Bootloader for an EFM8UB20F32G\_QFN32 Device**

The EFM8LB1 SMBus bootloader requires a non-standard pinout for use on the SLSTK2030A board. For this reason, the EFM8LB1 bootloader source code has three different project files as shown in [Table 8.1 EFM8LB1 Bootloader Source Code Project Files on page 24](#).

**Table 8.1. EFM8LB1 Bootloader Source Code Project Files**

| Project File                        | Symbol     | Notes  |
|-------------------------------------|------------|--|
| EFM8LB1_SMB0_S0_Bootloader.slsproj  | S0_PINOUT  | Production 'S0' device pinout, e.g. EFM8LB10F16ES0_QFN24 |
| EFM8LB1_SMB0_S1_Bootloader.slsproj  | S1_PINOUT  | Production 'S1' device pinout, e.g. EFM8LB10F16ES1_QFN24 |
| EFM8LB1_SMB0_STK_Bootloader.slsproj | STK_PINOUT | For use with nonstandard STK pinout.                     |

**Note:** The default device variant for each of the project files is the EFM8LB1\_DEVICE EFM8LB12F64E\_QFN32. Building for a different device requires the user to define a new device-dependent symbol for the project, as described previously.



## 9. Revision History

### Revision 0.4

September, 2018

- Added the [Security](#) chapter and updated usage examples and text throughout to reflect recommended best practices for code security.

### Revision 0.3

April, 2018

- Added EFM8UB3 devices.
- Added additional information on EFM8LB1 devices.
- Added [Table 1.1 EFM8 Device Bootloader Support on page 2](#).
- Updated [1. Introduction](#).
- Added [1.1 AN945SW Software Package](#).
- Updated [3. Getting Started with the USB or UART Bootloader](#).
- Updated [4. Getting Started with the SMBus Bootloader](#).
- Updated [5.3.2 Communication Interface](#).
- Updated [6.1 Hex2Boot — Hex to Bootload Record Converter](#).
- Updated [6.2 EFM8Load — Bootload Record Downloader](#).
- Updated [8.1 Bootloader Firmware](#).

### Revision 0.2

February, 2016

- Added SMBus bootloader for EFM8LB1 devices.
- Added EFM8LB1 devices.

### Revision 0.1

November, 2015

- Initial release.

Silicon Labs

# Simplicity Studio™4



## Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!



IoT Portfolio  
[www.silabs.com/IoT](http://www.silabs.com/IoT)



SW/HW  
[www.silabs.com/simplicity](http://www.silabs.com/simplicity)



Quality  
[www.silabs.com/quality](http://www.silabs.com/quality)



Support and Community  
[community.silabs.com](http://community.silabs.com)

### Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

### Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Micrium, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, Z-Wave, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



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

<http://www.silabs.com>