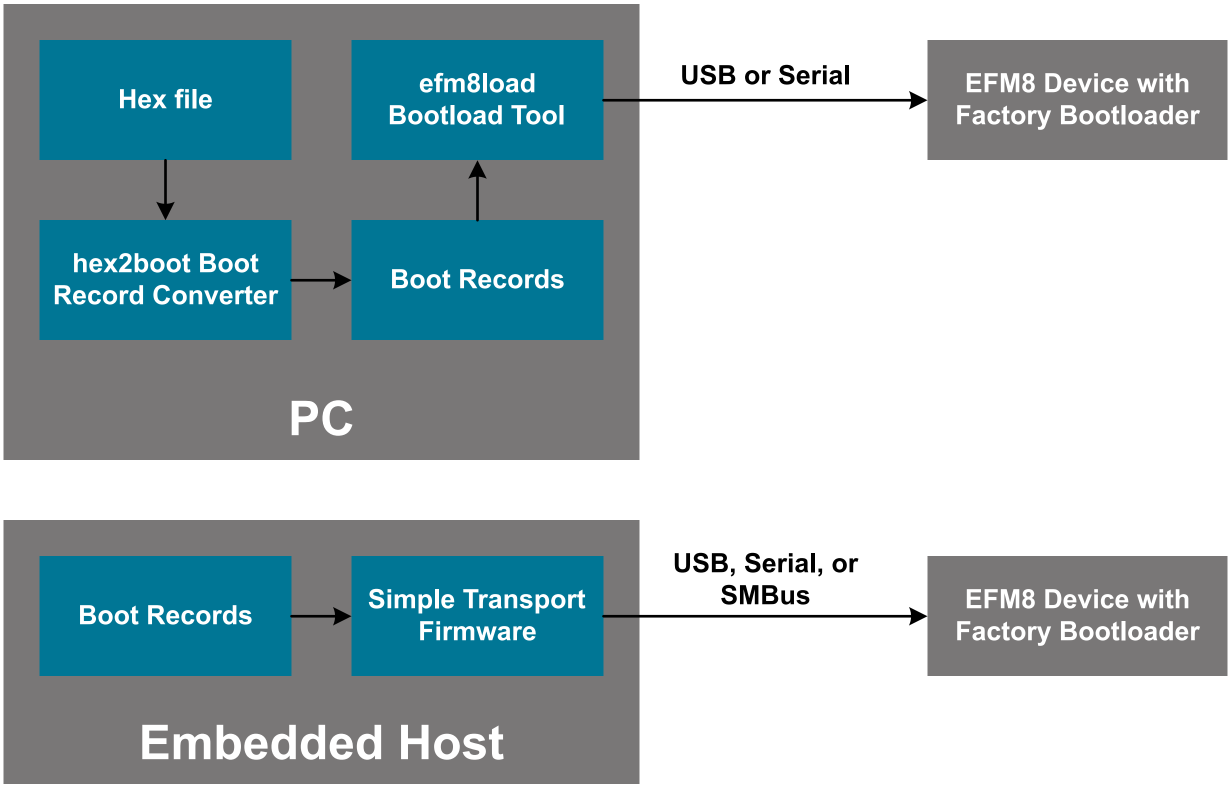
AN945: EFM8 Factory Bootloader User Guide

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.
- 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. This 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.
2. Field Updates — Updates can be issued to devices in the field without the need for end users to access the debug pins or use the debug adapter hardware.

The bootloader is designed mainly for production programming with a minimal feature set, but can also serve for field updates.

Because several of the EFM8 variants can have 2 KB of flash, the bootloaders are designed to be as small as possible. For example, the UART and SMBus versions consume a single 512 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 placement on each device can be found in the device data sheet or reference manual.

2. Getting Started with the USB or UART Bootloader

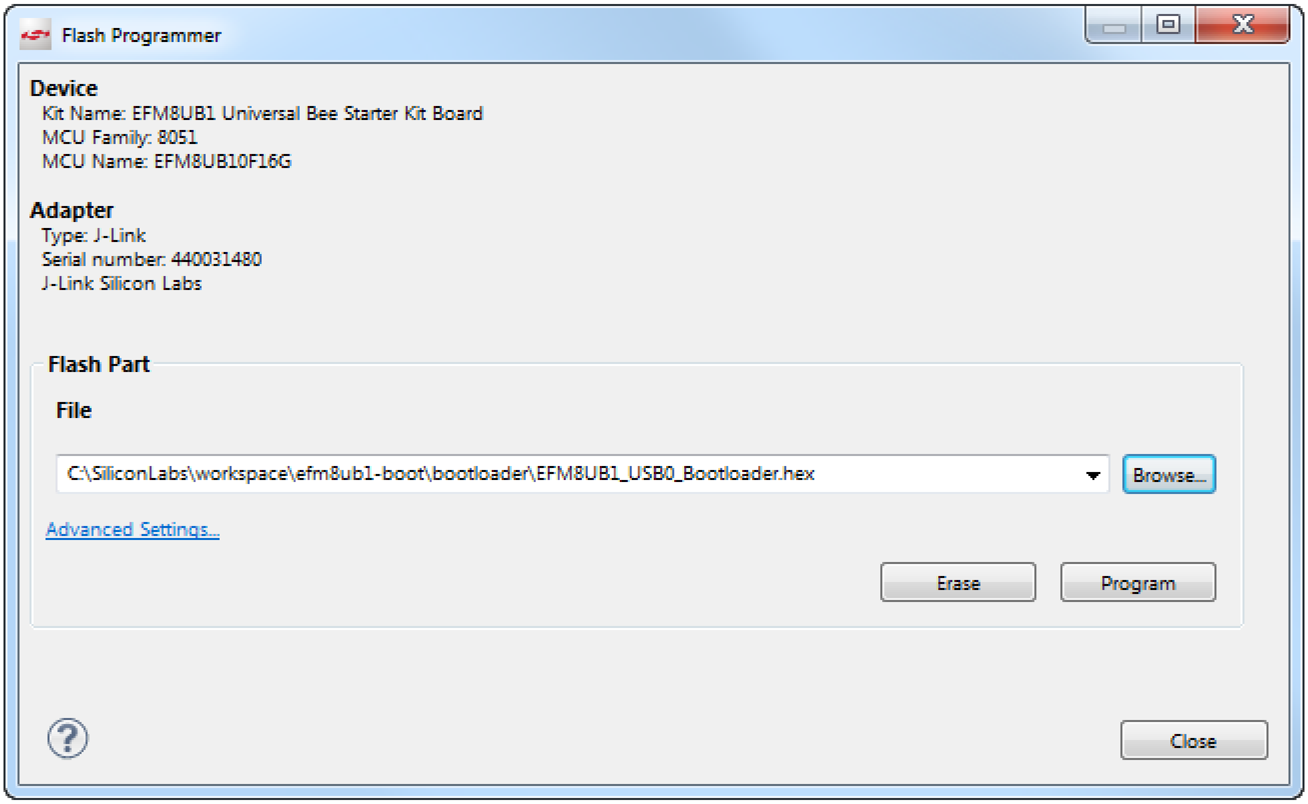
These steps assume the use of a Starter Kit. The steps are the same when using custom hardware.

These steps also assume that the application note zip file has been downloaded to the PC or that the files are accessed using Simplicity Studio. The application zip file can be found on the Silicon Labs website (www.silabs.com/8bit-appnotes).

- Download the Bootloader to the Device

  If the bootloader isn't already on the device, download the bootloader to the device using Simplicity Studio with the steps below. Devices with a date code in the top marking later than the date listed in the device errata can support the bootloader and may have the bootloader pre-installed. Devices with a date code 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**].



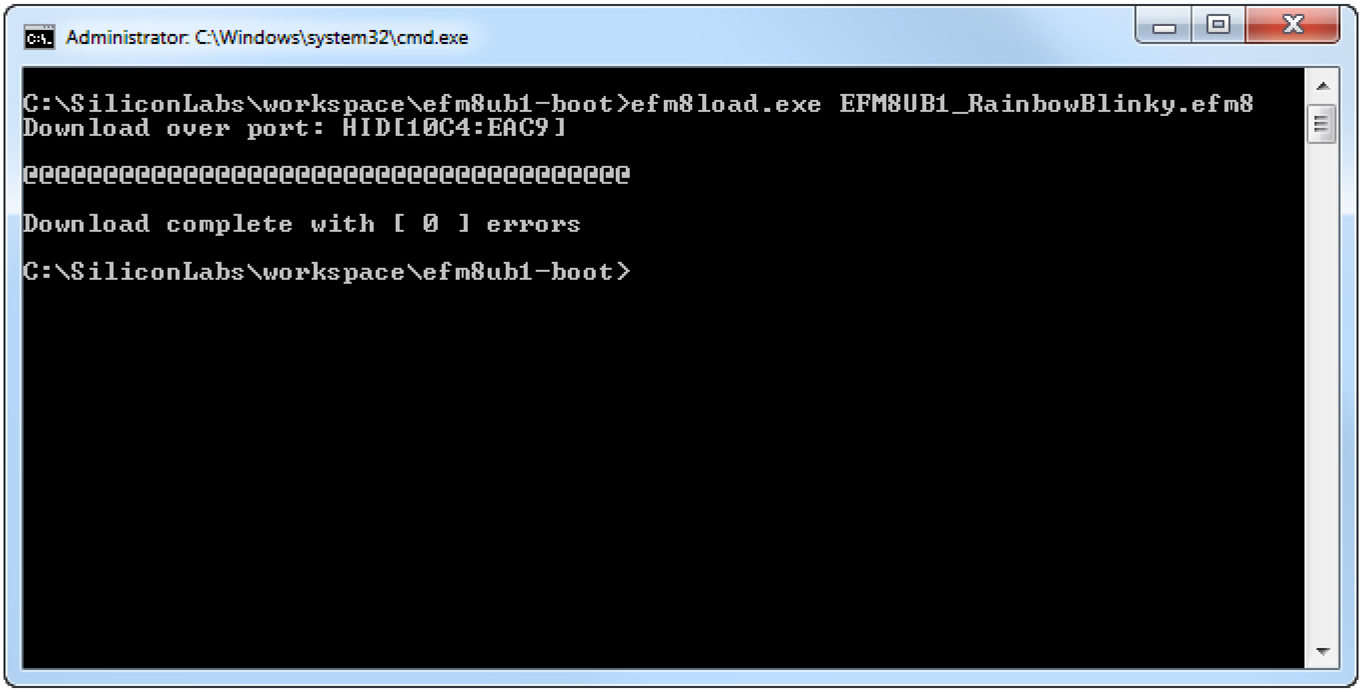
- Download an Application to the Device using the Bootloader

  - 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 (i.e. EFM8UB1), connect a second USB cable to the device USB connector on the bottom edge of the Starter Kit.
  - 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 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 5. Using the Host-Side Tools.


Administrator: C:\Windows\system32\cmd.exe
C:\SiliconLabs\workspace\efm8ub1-boot>efm8load.exe EFM8UB1_RainbowBlinky.efm8
Download over port: HID[10C4:EAC9]
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Download complete with [0] errors
C:\SiliconLabs\workspace\efm8ub1-boot>

3. Getting Started with the SMBus Bootloader

These steps assume the use of a Starter Kit. The steps are the same when using custom hardware.

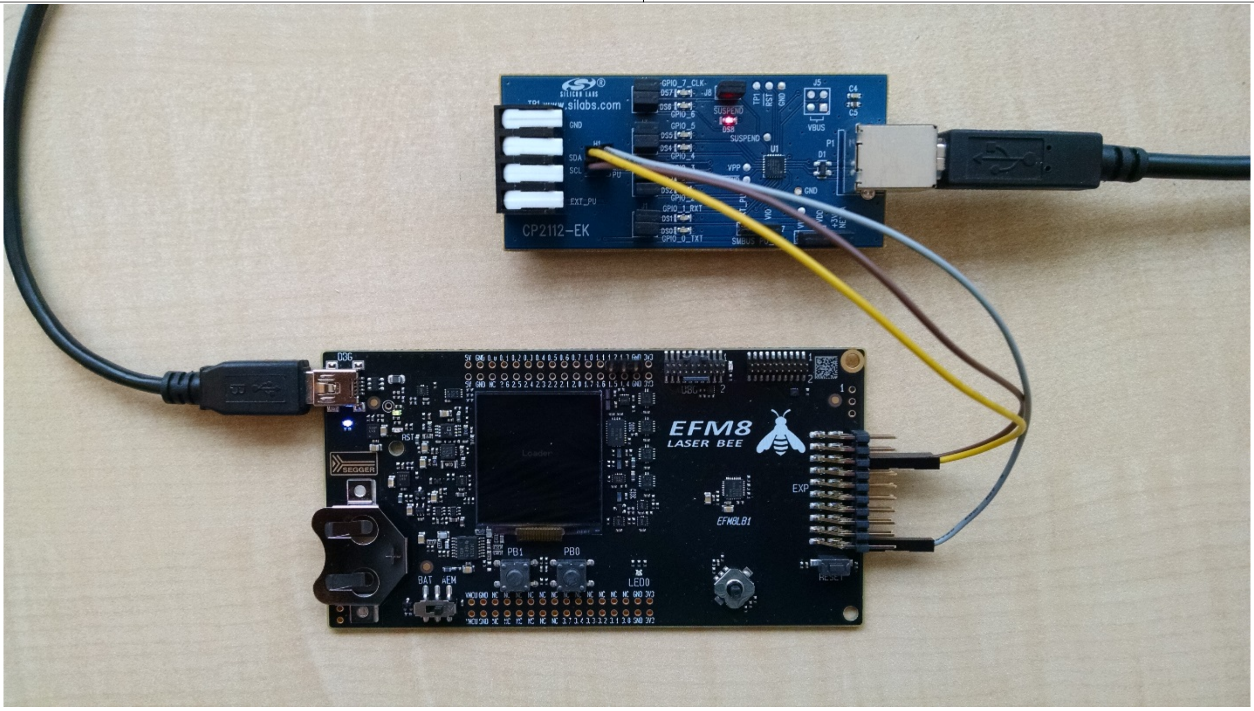
These steps also assume that the application note zip file has been downloaded to the PC or that the files are accessed using Simplicity Studio. The application zip file can be found on the Silicon Labs website (www.silabs.com/8bit-appnotes).

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. Note that the SMBus bootloader for the EFM8LB1 STK uses a non-standard pinout for SDA and SCK to accommodate the STK design. 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 3.1. 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



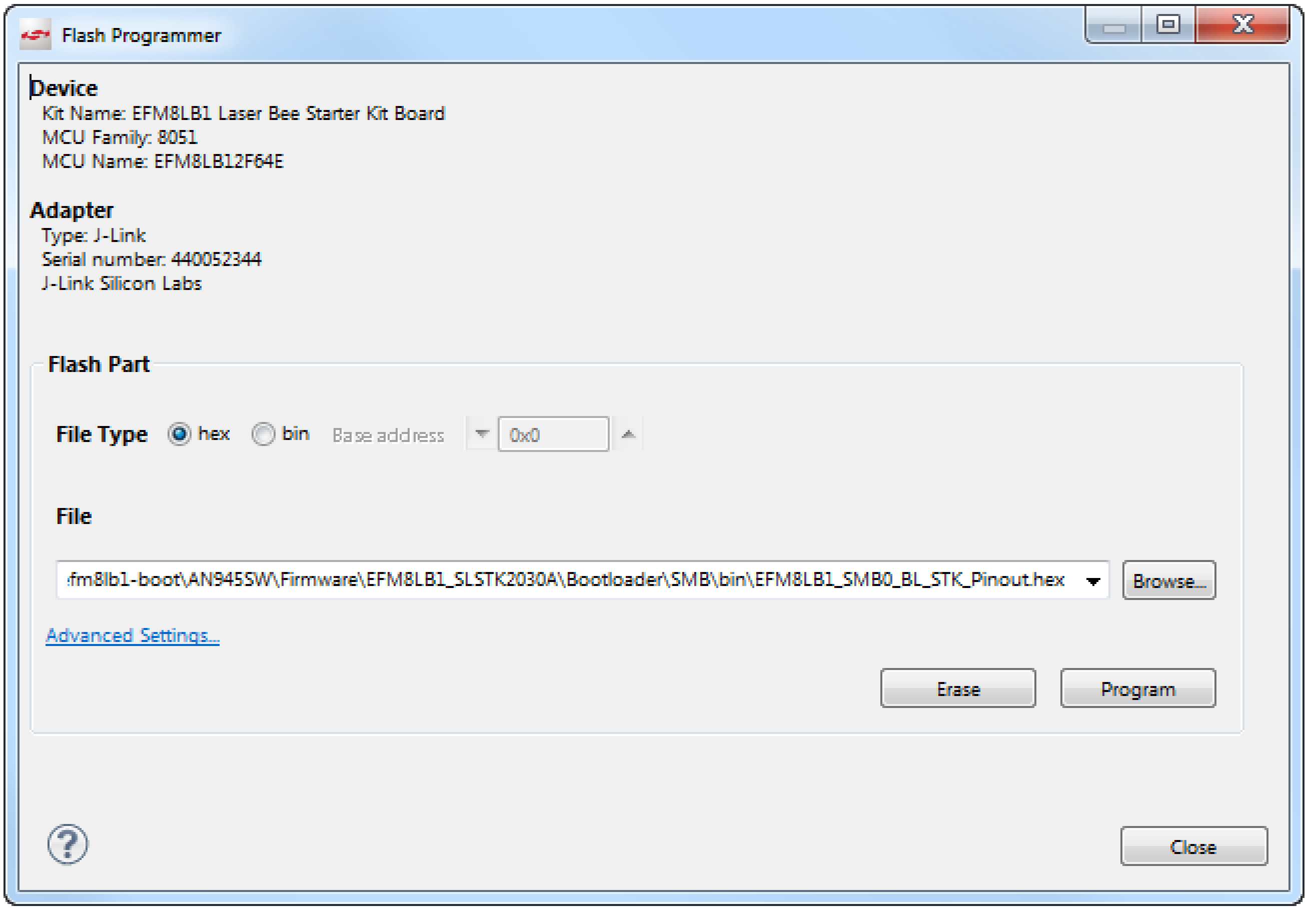
- Download the Bootloader to the Device

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

  Note: Note that the pinout for the SMBus bootloader for the STK is different than the standard SMBus bootloader pre-loaded into EFM8LB1 devices.

  - 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**].

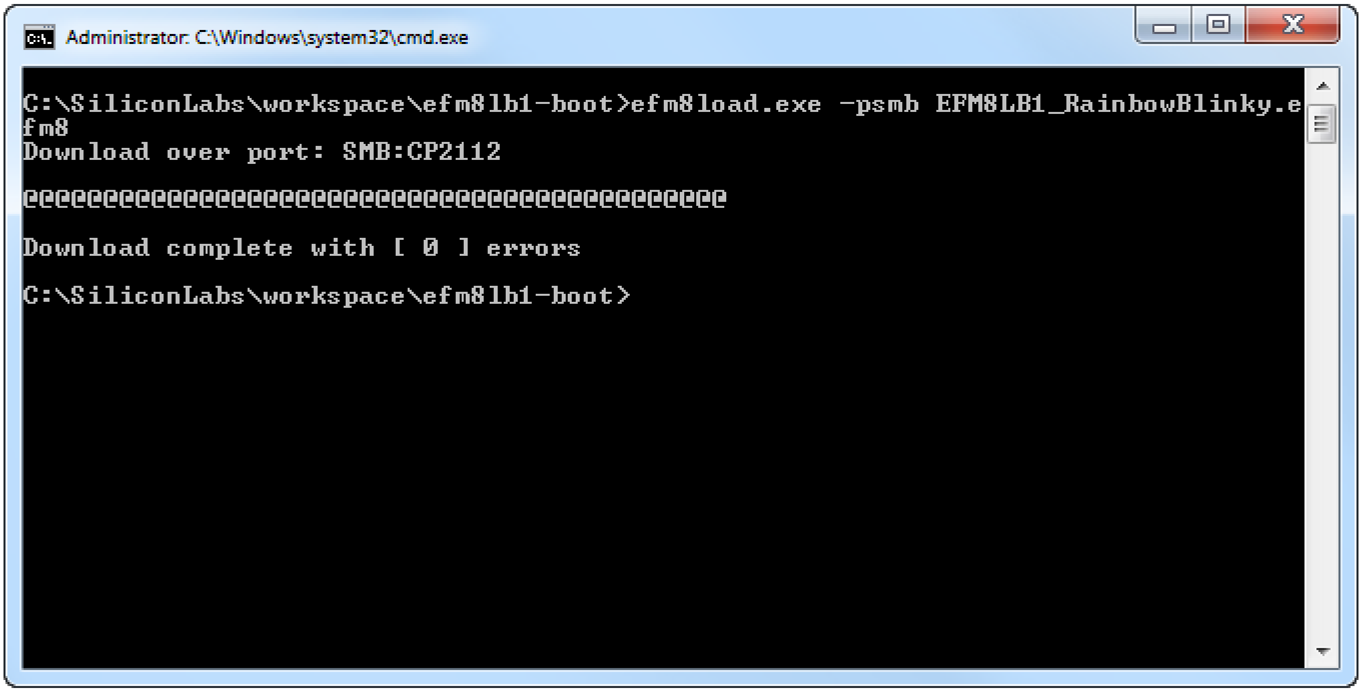


- Download an Application to the Device using the Bootloader
  - Since flash is initially empty (from the erase command in the previous step), the device will automatically start in bootload mode.
  - Open a command-line window in the same directory as the efm8load Python host tool ([**Start**]>[**Run**]>[**command**] in Windows).
  - Type `efm8load.exe`, followed by the SMBus bootloader switch 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**] switch to adjust the clock rate. With the RainbowBlinky application running on the STK, press the [**PB0**] button to start the bootloader. The RainbowBlinky pre-built boot image is located here: `.\AN945SW_SMB\Firmware\EFM8LB1_SLSTK2030A\RainbowBlinky\bin\EFM8LB1_RainbowBlinky.efm8`.

Note: Note that the bootload record is created by the `hex2boot.exe` host-side tool. More information on this tool can be found in 5. Using the Host-Side Tools.


Administrator: C:\Windows\system32\cmd.exe
C:\SiliconLabs\workspace\efm8lb1-boot>efm8load.exe -psmb EFM8LB1_RainbowBlinky.efm8
Download over port: SMB:CP2112
@@
Download complete with [0] errors
C:\SiliconLabs\workspace\efm8lb1-boot>

4. Feature Overview

4.1 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.

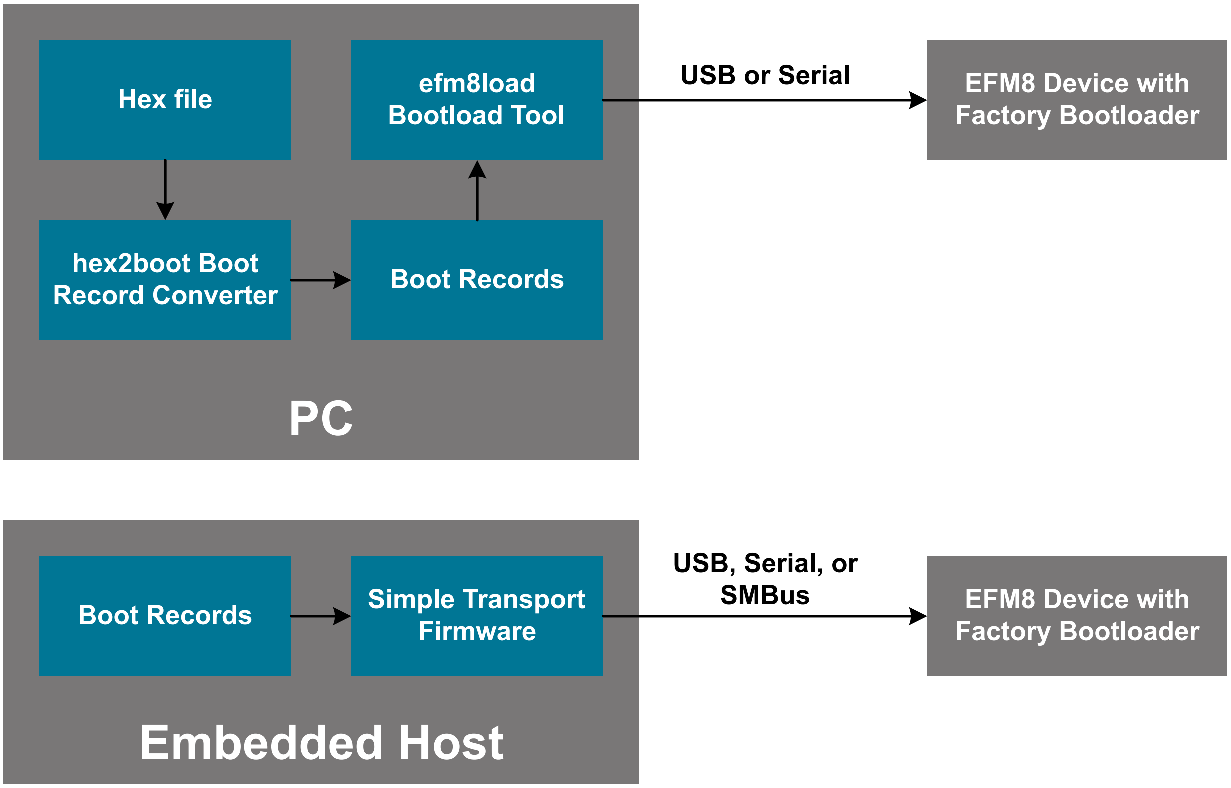


Figure 4.1. System Architecture

4.2 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.

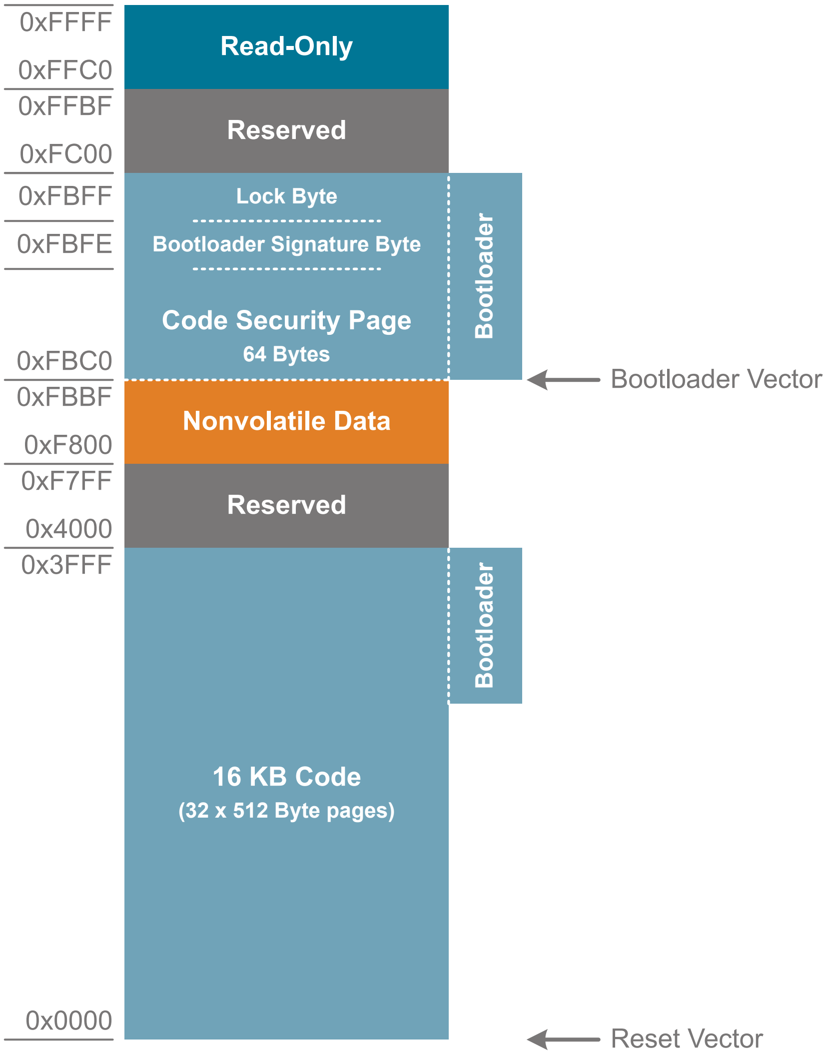


Figure 4.2. Example Flash Memory Map with Bootloader — EFM8UB1 16 KB Devices

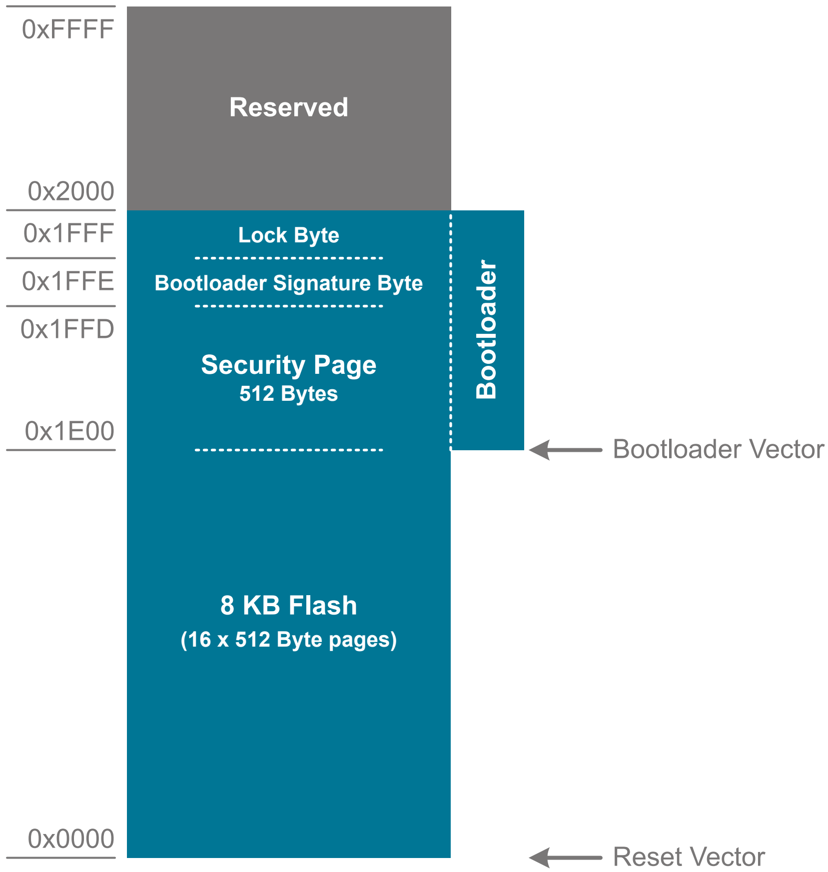


Figure 4.3. Example Flash Memory Map with Bootloader — EFM8SB1 8 KB Devices

4.3 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.

4.3.1 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 any interrupted bootload operations will restart when the interrupting event clears.
- 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 µs. A full list of entry pins for each device and package is available in Table 4.1 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 4.1. Summary of Pins for Bootload Mode Entry

Family	Device Package	Pin for Bootload Mode Entry
EFM8UB1	QFN28	P3.0 / C2D
	QSOP24	P2.0 / C2D
	QFN20	P2.0 / C2D
EFM8UB2	QFN48	P3.7
	QFP32	P3.0 / C2D
	QFN32	P3.0 / C2D
EFM8BB1	QSOP24	P2.0 / C2D
	QFN20	P2.0 / C2D
	SOIC16	P2.0 / C2D
EFM8BB2	QFN28	P3.0 / C2D
	QSOP20	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
EFM8LB1	QFN32	P3.7 / C2D
	QFP32	P3.7 / C2D
	QFN24	P3.0 / C2D
	QSOP24	P3.0 / C2D

4.3.1.1 Rainbow Blinky Example

The [**Rainbow Blinky**] (or [**PWM Blinky**] 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;
}
```

4.3.2 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 6. 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 Table 4.2 Summary of USB Bootloader PIDs on page 12. The PID is not customizable.

Table 4.2. Summary of USB Bootloader PIDs

Family	PID
EFM8UB1	0xEAC9
EFM8UB2	0xEACA

SMBus

- The SMBus bootloader is a slave interface only.
- The byte aligned slave address is 0xF0.
- The SMBus bootloader uses the SMB module (not the I2CSLAVE 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 5.1.2 Commands 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 SMBus bootloader assigns SDA to P0.2 and SCK to P0.3 for all EFM8LB1 packages. The STK uses these pins for another purpose, so there is a special SMBus bootloader build for the STK only.

4.3.3 Memory

- The bootloader always resides in the security page. This allows it to erase and write all application flash pages regardless of the lock byte status. This also prevents the bootloader from being accidentally erased by application code. If the bootloader is larger than the security page, it uses the top-most application-space pages.
- 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.
- 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.
- The bootloader does not support reading for security purposes. Verification of the flash contents can be done using a CRC.

4.3.4 Security and Encryption

- Encryption or authentication is not built-in as part of the bootloader.
- Application validation at startup (e.g. CRC over app) is not built-in as part of the bootloader.

5. 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.

5.1 Hex2Boot — Hex to Bootload Record Converter

5.1.1 Introduction

The `hex2boot` tool converts a standard Intel hex file that is output upon a successful build by 8051 build tools like Keil to a bootload record. The bootload record is a pure binary file composed of the bootloader commands described in 6. Bootloader Protocol.

Note: See the respective licenses in the zip file accompanying this document for information on restrictions for the intelhex module, which is used by `hex2boot`.

5.1.2 Commands

A list of commands 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 6. 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).

-m {bb2,sb2,ub1}, --map {bb2,sb2,ub1}

This parameter indicates a device with a special memory map (default = None). For example, the EFM8SB2 devices have a flash page size of 1024 bytes, rather than the 512 bytes for all other devices. The EFM8BB2 and EFM8UB1 devices also have special flash pages compared to the rest of the EFM8 family.

-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 this in cases where 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.

5.1.3 Usage Examples

To create a standard bootload record for an EFM8 device:

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

To create a standard bootload record for an EFM8SB2 device:

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

To create a bootload record that locks all pages in a device:

```
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
```

5.2 EFM8Load — Bootload Record Downloader

5.2.1 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 7. 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: See the respective licenses in the zip file accompanying this document for information on restrictions for the pyserial module, which is used by `efm8load`.

5.2.2 Commands

A list of commands 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.

5.2.3 Usage Examples

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

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

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

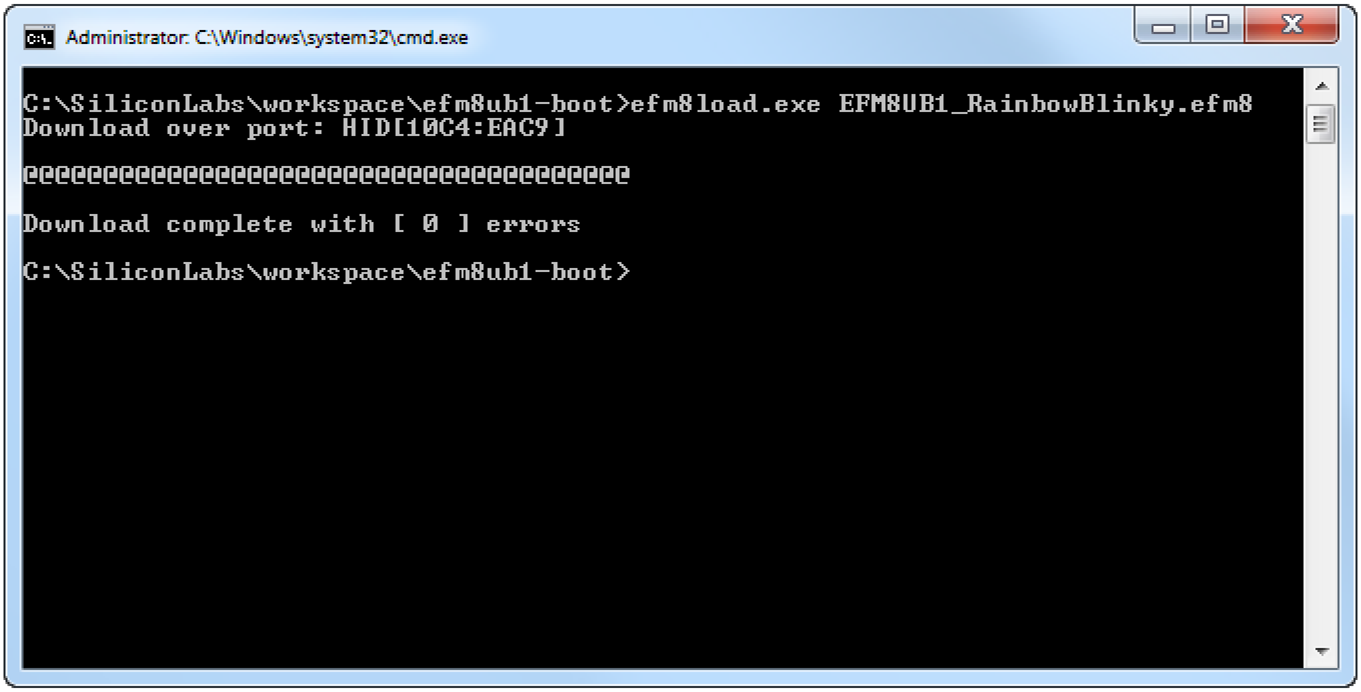


Figure 5.1. Basic `efm8load` Download Output

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

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

The output for this operation is shown below:

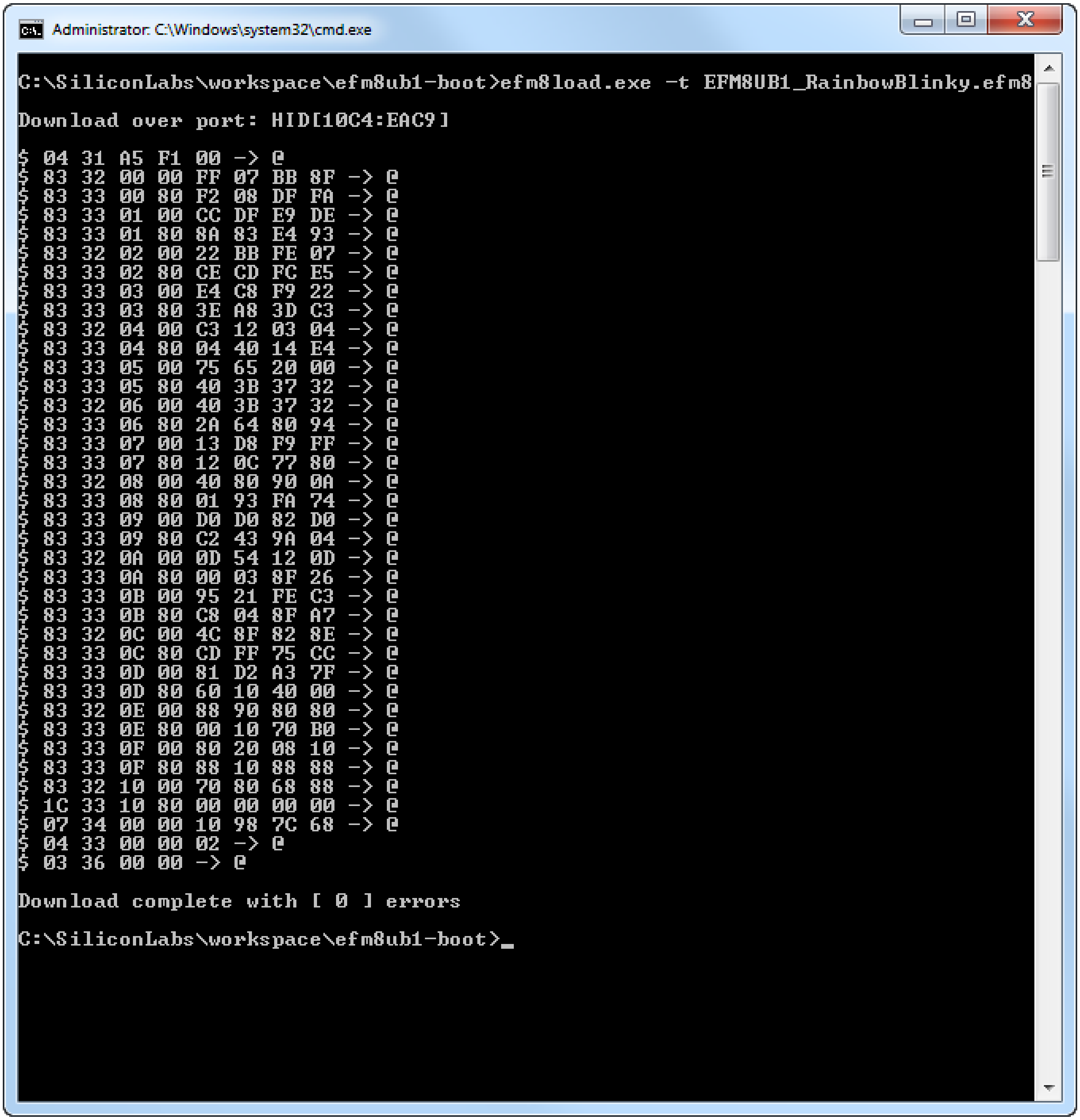


Figure 5.2. **`efm8load` Download Output with Trace**

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
```

6. 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:

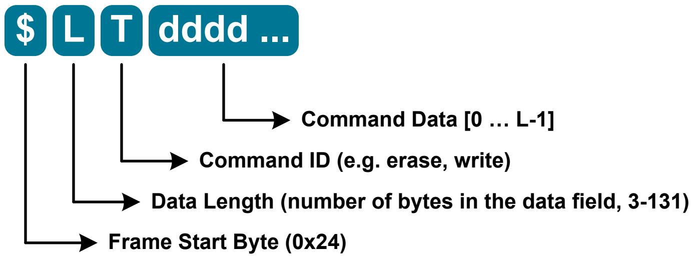


Figure 6.1. Binary Boot Record Format

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 EFM8LB1 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 4.3.2 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.

6.1 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. For added security, 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.

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).

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.

6.2 Command Summary

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

Table 6.1. Bootloader Command Summary

Command	Command Code	Length (bytes)	Return Code
Identify	0x30	3	ACK (0x40) if sent id matches device id 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 RANGE error (0x41) if the targeted address range cannot be written by the bootloader
Write	0x33	4 min, 131 max	ACK (0x40) normally 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 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)

7. Modifying and Rebuilding the Bootloader

7.1 Python Host-Side Tool (`efm8load`)

The Python host-side `efm8load` tool is designed to operate with:

- Python version 2.7 or 3.x
- Requires pyserial module, version 2.7 or later

See the respective licenses in the zip file accompanying this document for information on restrictions for the pyserial module. Any standard Python development environment like PyCharm can edit and run these files. To open the project:

1. Install the module versions above using the Python development environment.
2. Open the main `efm8load` source folder in the Python development environment.
3. Run the script.
4. Edit and debug using the development environment.

To rebuild the script into an independent executable, use the pyinstaller tool. This tool bundles the module dependencies into the executable so they do not have to be separately installed on the PC.

7.2 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.

If using the application note zip file downloaded from the Silicon Labs website, unzip the file in a directory on the PC and open Simplicity Studio. Click the [**Simplicity IDE**] tile and go to [**File**]>[**Import**]. From here, click on [**Simplicity Studio**]>[**MCU Project**], click [**Next**], [**Browse...**] to the unzipped `*.slsproj` bootloader project file, and complete the wizard steps to import the project.

To build the SMBus bootloader from source code, first import the project into Simplicity Studio following the instructions above. Once the project has been imported, select the [**STK_PINOUT**] configuration as shown below to build a bootloader for the STK.

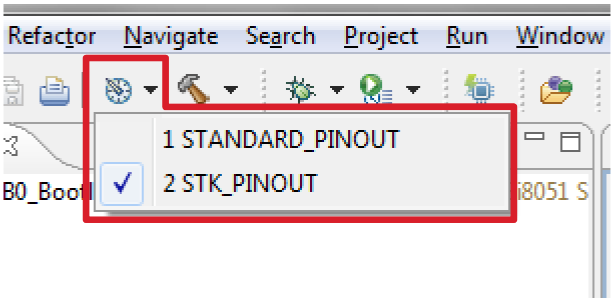


Figure 7.1. Building the SMBus Bootloader Firmware — STK_PINOUT Option

Otherwise, use the [**STANDARD_PINOUT**] configuration to build using the standard pinout described in 4.3.2 Communication Interface.

By default, the imported project builds the SMBus bootloader for the EFM8LB12F64E_QFN32 device. To build for a different device, you must change the project properties to add the [**EFM8LB1_DEVICE**] symbol. 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 [**EFM8LB1_DEVICE**] symbol settings the value to the desired device. The complete list of supported devices is in the `SI_EFM8LB1_Devices.h` header file. The following dialog shows how to configure to build for the [**EFM8LB12F64E_QSOP24**] device.

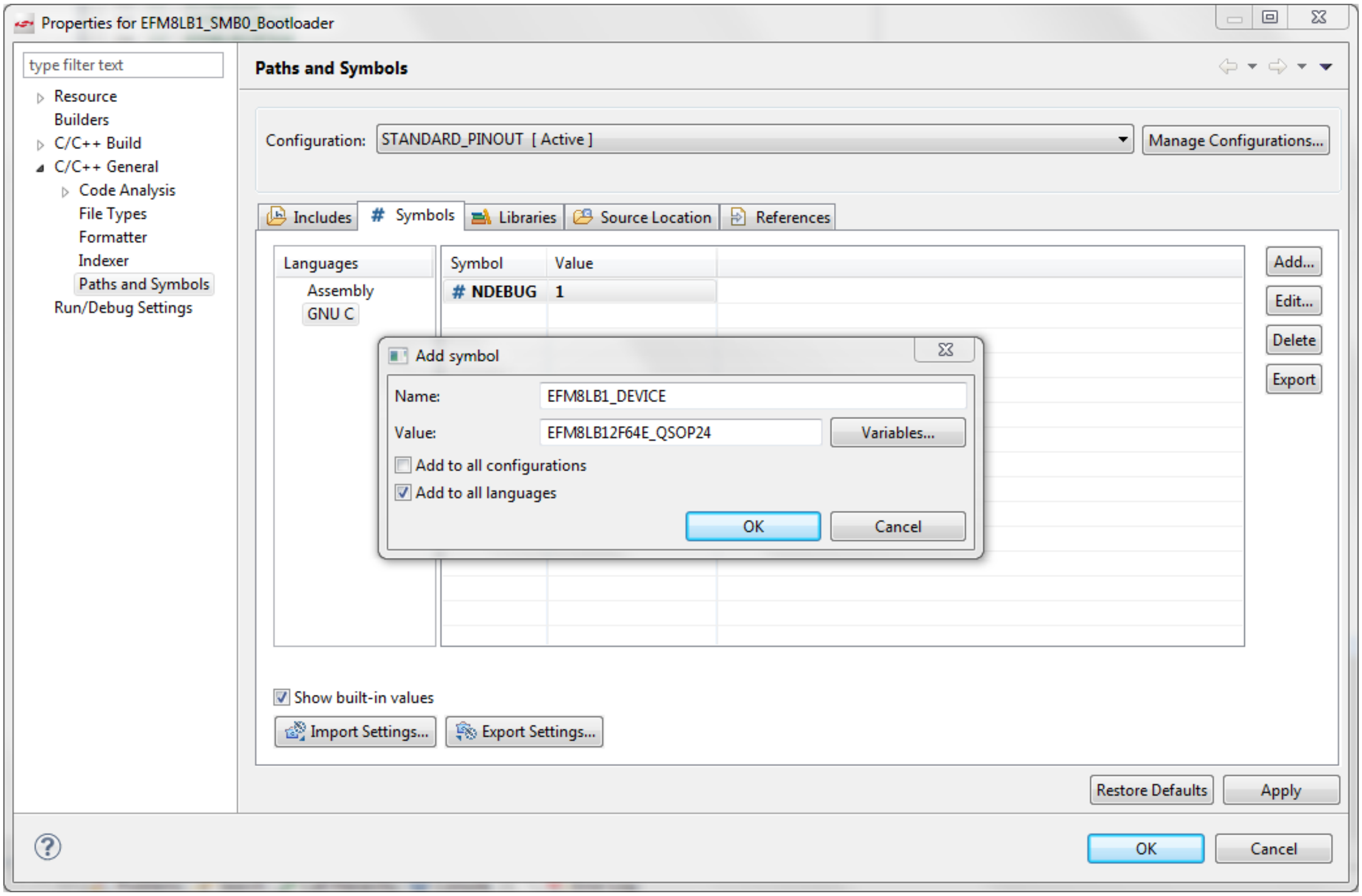


Figure 7.2. Reconfiguring the SMBus Bootloader for an EFM8LB12F64E-QSOP24 Device

8. Revision History

8.1 Revision 0.1

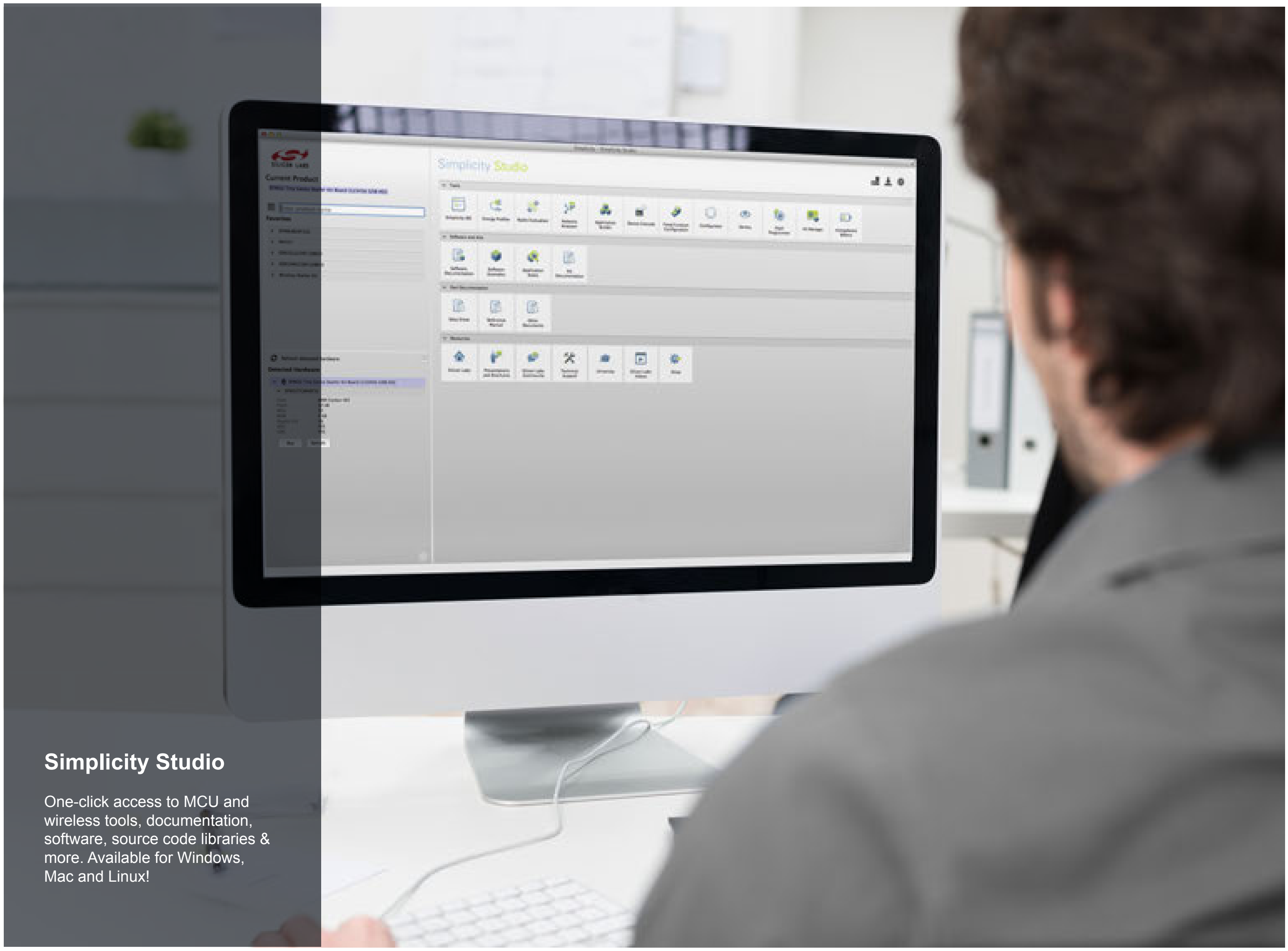
Initial release.

8.2 Revision 0.2

February 25, 2016

Added SMBus bootloader for EFM8LB1 devices.

Added EFM8LB1 devices.



IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

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