AN760: Using the Ember Standalone Bootloader

NOTE: As of the end of 2017, the Ember Standalone Bootloader was no longer supported for any EFR32 products, only for the EM35x. For all EFR32 products, configure a standalone bootloader from the Silicon Labs Gecko Bootloader, as described in *UG266: Silicon Labs Gecko Bootloader User Guide*.

This application note describes the implementation of the Silicon Labs proprietary Ember standalone bootloader. The Ember standalone bootloader is a special firmware image intended to reside on a chip separately from the application/stack firmware. It is designed as a simple, dedicated program to facilitate input of new application/stack firmware. Variants exist allowing upload by Xmodem to a serial interface (SPI, UART, or USB) and/or by a proprietary, IEEE 802.15.4-based, single-hop MAC layer RF protocol (the OTA bootloader, which is not compatible with EFR32 devices).

KEY POINTS

- Describes bootloader modes and upload recovery.
- Documents the bootloader API.
- Reviews OTA manufacturing token requirements.
- Provides OTA Standalone Bootloader packet formats.

1. Introduction

A bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and possibly perform some integrity checks. 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). An application bootloader performs a firmware image update by reprogramming the flash with an update image stored in internal or external memory. Application bootloaders can only be used on SoC (System-on-Chip) devices. A standalone bootloader is a program that that uses an external communication interface, such as UART or SPI, to get a firmware update image. NCPs (Network Coprocessors) only support standalone bootloaders.

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 should be used with all EFR32 platforms. 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. This document applies to the legacy Ember standalone bootloaders.

2. Normal Operating Methods

The Ember standalone bootloader and its utility libraries support one or more basic modes for receiving an application image onto a target device from a source device, including:

- Serial transfer via SPI, UART, or USB (for use with EmberZNet PRO-based applications)
- Over-the-air transfer via single-hop, proprietary, 802.15.4-based protocol

All Ember standalone bootloaders use the proprietary Ember Bootload (EBL) file format for the received image, although some bootloaders use a pre-encrypted “secure” EBL file format. See *UG103.6: Bootloader Fundamentals* for details about this file format and about bootloading concepts in general.

A variety of different bootloader variants for different platforms can be found in platform-specific subdirectories entitled “bootloader-{platformName}” beneath the “tool” or “tools” folder of your wireless stack installation. Refer to the table entitled “Bootloader Types and Features” in *UG103.6: Bootloader Fundamentals* for a more comprehensive list of available legacy Ember bootloaders and how they compare with one another. In addition to pre-built binaries for each of these bootloader variants on supported platforms, these subdirectories also contain IAR Embedded Workbench-based workspaces and projects, which can be used to modify certain aspects of the bootloader behavior, such as initialization sequence and GPIO configuration or serial configuration and behavior.

The subset of bootloaders classified as “Standalone” are the following:

- **ezsp-spi-bootloader** - A serial-only bootloader that uses SPI slave mode to receive the target image, where the SPI protocol has the same low-level SPI framing as the SPI variant of EmberZNet Serial Protocol (EZSP). Note that the target device is generally assumed to be a network coprocessor (NCP) using a SPI-based communication protocol, and the source device is assumed to be the same one where the NCP’s host application normally resides.
- **secure-ezsp-spi-bootloader** - A variant of ezsp-spi-bootloader that expects pre-encrypted EBL files using a preconfigured AES-128 security key.
- **serial-uart-bootloader** - A serial-only bootloader that uses UART mode, typically without any flow control, to receive the target image. This is typically used with a UART-based NCP target device to allow the host processor to update firmware on its NCP. However, it may be used with an SOC target as well, in which case the source device is often a serial terminal program controlled by human interaction.
- **serial-uart-ota-bootloader** - A variant of the serial-uart-bootloader that also includes support for over-the-air (OTA) reception of an EBL file via Silicon Labs’ proprietary MAC layer protocol.
- **secure-serial-uart-bootloader** - A variant of serial-uart-bootloader that expects pre-encrypted EBL files using a preconfigured AES-128 security key.
- **serial-usb-bootloader** - A serial-only bootloader that uses USB-CDC mode to receive the target image. Like the UART-based bootloaders, this can be used with an NCP target or an SOC.
- **secure-serial-USB-bootloader** - A variant of serial-usb-bootloader that expects pre-encrypted EBL files using a preconfigured AES-128 security key.

2.1 Serial Upload

You can establish a serial connection between a source device and a target device’s serial interface and upload a new software image to it using the XModem protocol. If you need information on the XModem protocol, a good place to start is http://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up-to-date links to protocol documentation.

2.1.1 Performing a Serial Upload - UART or USB

Serial upload can be performed with any source device that provides the expected serial interface method. This can be a Windows-based PC, a Linux or Mac OS-based device, or an embedded MCU with no operating system. UART and USB transfer can be done with a third-party serial terminal program like Windows HyperTerminal or Linux “lrzsz” or with user-compiled host code, such as the “OTA Platform Bootloader” plugin supplied with the EmberZNet PRO SDK. However, drivers for USB-CDC, SPI Master, or UART may vary with operating systems, and serial terminal programs may vary in timing and performance, so if you are unsure about what driver or program to use on your source code, consult Silicon Labs technical support.

To open a serial connection over UART, the source device connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1), with no flow control.

Note: The UART-based serial bootloaders do not employ any flow control in the communication channel, because the XModem protocol used for image transfer already has built-in flow control mechanisms. However, Silicon Labs’ normally-supplied NCP firmware *does* utilize either hardware-based (RTS/CTS) or software-based (XON/XOFF) flow control, so a host device must take care to temporarily disable the flow control when placing its NCP into serial bootloading mode. Alternatively, application designers can edit the UART driver code in the provided bootloader project for their bootloader and customize the serial bootloader’s handling of the UART to add hardware flow control at their discretion.

To open a serial connection over USB (only available on devices that support USB), the source device connects to the virtual COM port created by the source device’s USB-CDC driver. When connecting to this virtual COM port, baud, data bits, parity bits, and stop bits are irrelevant. The USB-based serial bootloaders use an ASCII-based menu for interaction similar that of the UART-based serial bootloaders described above.

Note: Any reset of the target device connected over USB will cause USB to disconnect and re-enumerate. This includes running an application image after a successful upload.

Once the connection with a UART- or USB-based serial bootloader is established:

1. The target device’s bootloader sends output over its serial port after it receives a carriage return from the source device at the expected baud rate. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port. Note that serial bootloaders typically don’t enforce any timeout when awaiting the initial serial handshake via carriage return, so the bootloader will wait indefinitely in this mode until guided by the source device or until the chip is reset.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following ASCII-based output:

```
1. upload ebl
2. run
3. ebl info
BL >
```

After listing the menu options, the bootloader's “BL >” prompt displays, and the ASCII character corresponding to the number of each option can then be entered by the source to select the described action, such as ‘2’ (ASCII code 0x32) to run the firmware presently loaded in the application area. Here again, no timeout is enforced by the bootloader, so it will wait indefinitely until a character is received or the chip is reset. Note that while the menu interface is designed for human interaction, the transfer can still be performed programmatically or through a scripted interface, provided the source device sends the expected ASCII characters to the target at appropriate times.

Note: Scripts that interact with the bootloader should use only the “BL >” prompt to determine when the bootloader is ready for input. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new options might be added.

Selection of the menu option 1 (upload ebl) initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of an EBL file over the expected serial interface, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. Once uploading begins (first XModem SOH data packet received), the bootloader expects each successive XModem SOH packet within 1 second, or else a timeout error will be generated, and the session will abort.
4. After an image successfully uploads, the XModem transaction completes and the bootloader displays ‘Serial upload complete’ before redisplaying the menu.

2.1.2 Performing a Serial Upload - SPI

To open a serial connection over SPI, the source device must act as SPI Master using Mode 0 or Mode 2. It must also react to edge-triggered interrupts from the slave device using the same nHOST_INT logic and SPI framing as the EZSP-SPI protocol described in *AN711: EZSP-SPI Host Interfacing Guide*.

Once the SPI slave enters bootloader mode, which includes a reset sequence (with host interrupt and Reset response frame similar to the reset sequence of a normal EZSP-SPI NCP), the bootloader sits in a Waiting state looking for SPI input in the form of bootloader packets (SPI bootloader frames with 0xFD SPI byte). The source device then needs to perform a bootloader Query transaction, which involves the source sending a Query packet and expecting a Response packet. Note that the first Query transaction will yield a Query-Found result (status byte 0x1A), while the subsequent Query will yield the expected Response result (status byte 'R' or 0x52). For details, refer to the sample SPI bootloading process described in section 2.1.3 Sample EZSP-SPI Bootloader Transcript.

Once a query transaction has completed successfully (with expected Response frame), the transfer of data packets can begin. The transfer process follows standard XModem-CRC protocol, just like the UART- and USB-based serial bootloaders use, but with SPI framing similar to that used to encapsulate EZSP data frames. This SPI-based XModem adaptation adheres to the following rules, some of which may differ from the SPI protocol used by the normal EZSP NCP firmware:

- The 0xFD SPI byte and a length byte (for number of bytes to follow) prefix every command or response frame.
- The 0xA7 frame terminator byte concludes every SPI command or response frame. This byte is **not** included in the length count used in the length byte.
- The NCP operates as a SPI slave, so nSSEL must be asserted before each transaction.
- No EZSP frame control bytes are used in the SPI frame. Consequently, no sleep mode operation is supported by the target during the bootload.
- SPI timing (timeouts, signal transitions) are similar to EZSP. (See *AN711: EZSP-SPI Host Interfacing Guide,* for details.)
- The nHOST_INT signal is asserted by the target to indicate a pending response.
- In place of the EZSP “callbacks” command, the host should use the bootloader’s Query packet to prompt the target to push the asynchronous response (such as XModem ACK) back to the host.
- Each SPI frame will typically generate an initial, synchronous response from the target, such as a BLOCKOK status, and a follow-up, asynchronous response, which must be queried for by the host. For example, the EOT packet generates a synchronous response with FILEDONE status, then Query transaction yields XModem ACK with block number of lastBlock+1 before rebooting into new firmware.
- The host must wait for nHOST_INT to assert (become low) before querying for status, as the SPI bootloader is edge-triggered rather than level triggered. Thus, acting too fast at the host side can cause an edge transition to be missed and the bootloading state machines at the host and NCP to get out of synchronization, resulting in problems later on.
- SPI Status and SPI Version commands (SPI bytes 0x0A and 0x0B) are still supported.
- Prior to the first data block being processed, the SPI bus is polled at a rate of once per second.
- Once the data transmission begins (first block processed), the bootloader will wait up to 60 seconds for the next data packet, polling at 5 second intervals.
- If either of the timeouts above is exceeded, the bootloader will signal a cancellation (CAN frame) and will reboot, restarting the state machine.
- XModem data packets consist of:
  - The SOH byte (ASCII 0x01)
  - A 1-byte incrementing block number (beginning at 1 and wrapping back around from 255 to 0)
  - The block number’s complement
  - 28 bytes of data read directly from the EBL file being uploaded
  - A 16-bit CRC of the data bytes from that packet
- Each packet is followed by an XModem ACK or NAK from the target device (the NCP running the bootloader), which confirms or refutes the current data packet.
- If the target receives a duplicate block, it simply sends the ACK for that block again. If the target receives a block that had an XModem frame error (such as bad CRC), the bootloader expects that data block to be retransmitted and then the bootload can continue. Other kinds of errors are considered unrecoverable and will cause the bootload to abort.
- If the bootload process aborts for any reason (including receiving an XModem Cancel (CAN) frame from the source), an XModem Cancel frame is echoed on the SPI interface from the target, and the target will then reboot, restarting the bootloader state machine.
- When the source device reaches the last XModem data block, it should be padded to 128 bytes of data using SUB (ASCII 0x1A) characters.
- Once the last block is ACKed by the target, the transfer should be finalized by an EOT (ASCII 0x04) packet from the source. Once this packet is confirmed via XModem ACK from the target, the device will reboot, causing the new firmware to be launched.

  Note: The ACK for the last XModem data packet may take much longer (1-3 seconds) than prior data packets to be received. This is due to the CRC32 checksum being performed across the received EBL file data prior to sending the ACK. The source device must ensure that its SPI XModem state machine waits a sufficient amount of time to allow this checksum process to occur without timing out on the response just before the EOT is sent.

2.1.3 Sample EZSP-SPI Bootloader Transcript

The following is a record of the SPI frames transmitted and received by the EZSP-SPI host during the SPI bootload process, as captured from an EZSP Host application running the ota-bootload-ncp-spi.c state machine from the Silicon Labs Zigbee application framework's OTA Platform Bootloader plugin with an EZSP-SPI NCP device as target using the ezsp-spi-bootloader.

A "TX:" line means that the hexadecimal byte values contained in brackets ("[…]") are transmitted by the source via SPI to the target.

An "RX:" line means that the byte values that follow in brackets are received by the source via SPI from the target NCP device.

Note: Firmware data (in the form of an EBL file for an EZSP-SPI NCP) is being streamed to the host's UART (for relaying down to the target) during this process, but that serial stream is not shown here.

Comments about the process are indicated in italics and are not part of the data transmitted on the SPI bus.

```
Check to make sure NCP booted properly into bootloader...
TX: [0B A7]
RX: [00 09 A7]
TX: [0B A7]
RX: [C1 A7]
TX: [0A A7]
RX: [82 A7]
TX: [FD 01 51 A7]
RX: [FD 01 1A A7]
TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF
00 02 02 02 20 0A A7]
TX: [FD 01 51 A7]
RX: [FD 01 1A A7]
TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF
00 02 02 02 20 0A A7]
Starting SPI bootloading...
TX: [FD 01 51 A7]
RX: [FD 01 1A A7]
TX: [FD 01 51 A7]
RX: [FD 1A 52 01 FF FF 64 65 76 30 34 37 31 00 FF FF FF FF FF FF FF FF
00 02 02 02 20 0A A7]
TX: [FD 85 01 01 FE 00 00 00 3C 14 20 E2 60 48 7E AA 56 42 50 7C 53 0A
71 77 77 FF FF FF FF FF FF 78 61 70 32 62 2D 65 6D 32 36 30 2D 65 6D 32
35 30 2D 64 65 76 30 34 37 30 00 00 00 00 00 00 00 80 ED 3A AA FD 03 21
00 02 E0 00 04 57 40 00 00 3B E0 00 00 05 00 A1 E0 14 40 CA B7 E0 32
05 38 1E 05 92 6C 00 02 80 0B 54 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 01 00 A7]
TX: [FD 85 01 02 FD FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF C0 1B 48 00 00 2B 20 C8 0A F0 C0 1B 01 00 C0 C8 04 F0 00
04 01 00 91 E0 01 00 89 E0 C0 DB C0 2B FA 00 40 19 01 38 FA 00 40 29 FA
00 41 89 03 E4 FA 00 41 29 46 00 7A DB A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 02 00 A7]
TX: [FD 85 01 03 FC 18 99 75 F4 C4 27 C6 23 C8 02 CA 03 CC 2F 7A 00 3E
15 DC 27 46 00 1A 15 CE 27 00 14 D8 27 7A 00 3E 25 46 00 00 15 CE C7 D8
B7 56 F4 00 10 D6 27 02 C4 03 00 49 F0 02 B0 D6 17 80 00 00 C4 03 00 D3
F0 80 00 00 B0 D6 17 40 00 00 C4 02 00 68 F0 40 00 00 B0 D6 17 04 00 00
C4 77 F0 04 00 00 B0 D6 17 20 0C 20 0F 00 00 00 B0 D6 17 10 00 00 C4 02
00 96 F0 10 00 00 B0 D6 17 08 00 C4 1E A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 03 00 A7]
TX: [FD 85 01 04 FB 00 C4 03 00 B6 F0 08 00 00 B0 D6 17 02 00 00 C4 01
00 92 F0 02 00 00 B0 D6 17 01 00 00 C4 01 00 C3 F0 01 00 00 B0 D6 17 40
C4 01 00 38 F0 40 B0 D6 17 20 C4 01 00 F0 F0 20 B0 D6 17 10 C4 02 00 F7
F0 10 B0 D6 17 04 C4 03 00 D0 F0 D6 27 CE D7 CE 27 D6 17 19 E0 CE 17 46
00 1A 25 CC 1F DC 17 7A 00 3E 25 CA 07 C8 06 C6 13 C4 17 01 58 FA 00 40
29 C0 1B 05 EC C0 00 A7]
RX: [FD 01 19 A7]
```

```
TX: [FD 01 51 A7]
RX: [FD 03 06 04 00 A7]
250 more data block transmissions follow; omitted here for brevity.
The frames following immediately below illustrate how the sequence
number wraparound condition is handled...
TX: [FD 85 01 FE 01 22 17 24 57 2C C5 EA 25 20 17 24 37 EA 11 FC 3C FF
00 B1 9C 04 3C 28 3C FE E3 FE 2B F4 27 F6 23 C4 3C 1E 00 6D 9C 34 27 3E
17 FE 27 3C 17 FC 27 FE 2D FE 15 20 34 30 13 FC 3C FF 00 11 9C 40 37 2C
C5 04 3C 36 27 34 13 2C 81 03 F0 30 13 34 23 79 00 5E 19 34 3B 80 00 00
16 04 54 2C C5 38 27 00 14 FE 27 04 14 FC 27 38 17 FA 27 FE 2D FE 15 10
34 34 13 FA 3C 58 9D 3A 17 3E 13 06 3C 20 00 83 32 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 FE 00 A7]
TX: [FD 85 01 FF 00 C0 9C FE 00 C6 15 0F F4 FE 2D FE 15 20 34 FE 27 00
14 FC 27 FE 2D FE 15 10 34 04 10 FC 3C FE 00 6A 9C 04 3C FE 00 C6 15 0B
F4 FE 2D FE 15 20 34 FE 27 30 17 36 13 FE 3C FE 00 C4 9C 02 3C E2 2D E2
15 FE 27 FE 2D FE 15 20 34 FC 27 00 14 FA 27 F8 27 36 17 F6 27 32 17 F4
27 30 17 00 10 F4 3C FD 00 F9 9C 04 14 0A 27 FE 2D FE 11 0C 30 FE 2D FE
15 1C 34 0A 3C 44 9D 02 3C 00 84 03 F0 01 14 79 65 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 FF 00 A7]
TX: [FD 85 01 00 FF 02 E0 00 14 3C 3C FE E3 FE 2B 10 14 42 00 18 25 42
00 44 15 01 B4 42 00 44 25 46 00 1A 15 08 B5 46 00 1A 25 6F 00 80 14 42
00 02 25 42 00 00 25 42 00 02 15 1F 34 42 00 02 25 01 14 42 00 18 25 7E
00 D8 25 6F 00 80 5 42 00 04 25 42 00 06 15 1F 34 18 25 00 14 7E 00 DA
25 01 14 F00 DC 25 FE E3 FE 2B FC 27 00 14 94 25 FA 00 9D 93 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 00 00 A7]
TX: [FD 85 01 01 FE DC 25 41 00 2E 25 32 00 64 14 41 00 2C 25 15 14 42
00 48 25 60 00 664 41 00 40 25 58 14 41 00 42 25 03 14 41 00 44 25 00
26 25 28 14 41 00 38 25 55 14 41 00 34 25 ED 00 80 14 41 00 3A 25 09 14
41 00 3E 25 1F 14 41 00 30 25 0E 15 FA 27 0E 11 11 14 FA 3C 50 9D 7A 00
D8 15 02 3C 0C 00 4C 9C 40 14 42 00 44 25 03 14 42 00 42 25 41 00 4A 25
FA 00 0E DE A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 01 00 A7]
TX: [FD 85 01 02 FD D6 15 0D 00 B0 9C 00 84 06 F4 01 00 2D 10 03 00 24
14 46 9D FA 00 D7 11 08 A0 08 A4 E2 21 E2 15 0D 00 47 9C 01 14 41 00 46
25 18 00 00 14F 00 B2 9C FF 10 FF 14 0F 00 A8 01 A2 E_00 7C 9C 01 14
0F 00 90 9C 01 14 41 00 14 25 46 00 1A 15 06 B5 46 00 1A 25 04 00 F3 14
46 00 1E 0 1C 25 FF 00 4D 9C 00 14 04 3C F 0F 42 _B A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 02 00 A7]
826 more data block transmissions follow; omitted here for brevity.
The frames following immediately below illustrate how the last data
block is padded to 128 bytes and how the transmission is concluded
successfully with the EOT and a final query transaction...
TX: [FD 85 01 3F C0 01 00 01 00 FF FF FF 00 FF 00 FF 00 AB CD C1 10 FF
00 FC 04 00 04 5A 0C BA B8 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 9F 61 A7]
RX: [FD 01 19 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 3F 00 A7]
TX: [FD 01 04 A7]
RX: [FD 01 17 A7]
TX: [FD 01 51 A7]
RX: [FD 03 06 40 00 A7]
NCP resets back into EZSP at this point, so normal reset sequence
occurs and frames begin to use 0xFE as SPI byte...
TX: [0B A7]
TX: [0B A7]
TX: [0A A7]
TX: [0B A7]
TX: [0B A7]
TX: [0A A7]
```

```
TX: [FE 04 07 00 00 02 A7]
RX: [FE 07 07 80 00 02 02 40 32 A7]
TX: [FE 04 08 00 52 01 A7]
RX: [FE 06 08 80 52 00 21 00 A7]
TX: [FE 06 09 00 53 01 18 00 A7]
RX: [FE 04 09 80 53 00 A7]
TX: [FE 04 0A 00 52 02 A7]
RX: [FE 06 0A 80 52 00 10 00 A7]
TX: [FE 04 0B 00 03 A7]
RX: [FE 06 0B 80 52 00 0A 00 A7]
EZSP startup process continues from this point.
```

2.1.4 Errors and Status Codes

If an error occurs during the upload, the UART or USB serial bootloader displays the message 'Serial upload aborted,' followed by a more detailed message and a hex error code. Some of the more common errors are shown in the following table. A more complete list is in **bootloader-common.h**. The UART or USB serial bootloader then redisplays the bootloader menu. If an error occurs during the SPI serial bootload, the target produces an error response followed by an XModem Cancel frame and a reboot.

The following table describes the normal status codes, error conditions, and special characters or enumerations used by some or all of the standalone bootloader variants.

Table 2.1. Serial Uploading Statuses, Error Messages, and Special Characters

Hex code	Constant	Description
0x00	BL_SUCCESS	Default success status.
0x01	BL_ERR	General error processing packet.
0x1C	BLOCK_TIMEOUT	The bootloader timed out waiting for some part of the XModem frame.
0x21	BLOCKERR_SOH	The bootloader did not find the expected start of header (SOH) character at the beginning of the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected the sequence check byte of the XModem frame was not the inverse of the sequence byte.
0x23	BLOCKERR_CRCH	The bootloader encountered an error while comparing the high bytes of the received and calculated CRCx of the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while comparing the low bytes of the received and calculated CRCs of the XModem frame.
0x25	BLOCKERR_SEQUENCE	The bootloader did not receive the expected sequence number in the current XModem frame.
0x26	BLOCKERR_PARTIAL	The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost).
0x27	BLOCKERR_DUPLICATE	The bootloader encountered a duplicate of the previous XModem frame.
0x40	BL_ERR_MASK	Bitmask for any bootloader error codes returned in CAN or NAK frame.
0x41	BL_ERR_HEADER_EXP	No .EBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Failed to write header or CRC.
0x43	BL_ERR_CRC	File or written image failed CRC check.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in .EBL image.
0x45	BL_ERR_SIG	Invalid .EBL header contents.
0x46	BL_ERR_ODD_LEN	Trying to flash odd number of bytes.
0x47	BL_ERR_BLOCK_INDEX	Indexed past end of block buffer.
0x48	BL_ERR_OVWR_BL	Attempt to overwrite bootloader flash.
0x49	BL_ERR_OVWR_SIMEE	Attempt to overwrite SIMEE flash.
0x4A	BL_ERR_ERASE_FAIL	Flash erase failed.
0x4B	BL_ERR_WRITE_FAIL	Flash write failed.
0x4C	BL_ERR_CRC_LEN	End tag CRC wrong length.

Hex code	Constant	Description
0x4D	BL_ERR_NO_QUERY	Received data before query request/response.
0x4E	BL_ERR_BAD_LEN	An invalid length was detected in the .EBL image.
0x4F	BL_ERR_TAGBUF	Insufficient tag buffer size or an invalid length was found in the .EBL image.
Special Characters Used in Packet Types		
0x01	SOH	Start of Header.
0x03	CTRL_C	Cancel (from sender).
0x04	EOT	End of Transmission.
0x06	ACK	Acknowledged.
0x15	NAK	Not acknowledged.
0x18	CAN	Cancel
0x43	C	ASCII ‘C’.
0x51	QUERY	ASCII ‘Q’.
0x52	QRESP	ASCII ‘R’.
Status Codes Returned in a Synchronous Response		
0x16	TIMEOUT	Bootloader timed out expecting characters.
0x17	FILEDONE	EOT process successfully.
0x18	FILEABORT	Transfer aborted prematurely.
0x19	BLOCKOK	Data block processed OK.
0x1A	QUERYFOUND	Successful query.

2.1.5 Running the Application Image

For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu. For SPI-based variants, which don’t use a menu, the application is run immediately upon ACKing the EOT frame from the source device.

Note: Because option 2 always resets the target device, the bootloader operating over USB will disconnect and then re-enumerate.

2.1.6 Obtaining Image Information

An application’s image information is a free text string that can be used by customers to provide additional details about the application. In EBL files, the image information (“image info” field) is customizable by the user. When creating an EBL file from an s37 image with the em3xx_convert utility, the image information can be specified as a string using the `--imageinfo` option. Menu option 3 then displays the information as a quoted string, similar to the following:

"custom image info"

The information displayed by these commands represents the image that is currently stored in the flash. This means that after a successful bootload, the image information should change to reflect the new application.

Note: Simplicity Commander does not yet support customizing the image info field for created EBL files, so em3xx_convert from the ISA3 Utilities must be used if custom image info is desired.

2.2 Over-the-Air Upload

For standalone bootloaders with OTA transfer capability (those with “ota” in their filenames), you can upload images over the air from a source device to a target device via a Silicon Labs-proprietary, IEEE 802.15.4-based MAC layer protocol. This protocol’s packets can be sent and received by the lower layers of Silicon Labs’ wireless stacks, as well as by the OTA-enabled standalone bootloaders themselves.

Note: Over-the-air target functionality is **not** available for any of the USB-based standalone bootloaders for EM358x/9x, although these devices are capable of acting as sources for image transfer.

In all cases, the source device must be within radio range of the target device, although there is no requirement for both devices to be joined to the same network in advance of the transfer since the bootloader’s OTA protocol can be considered “out of band” from the normal networking protocol that the application uses. The source device uses a simplified MAC-based protocol to communicate with the target, which can only travel one hop. This protocol is based on XModem CRC but uses 64-byte data blocks that can fit in a single 802.15.4 packet.

During over-the-air upload, only the target device actually runs the bootloader. The source device and any intermediary passthrough devices that participate in the upload process continue to run their normal application firmware with full networking capabilities.

Although OTA bootloading uses the same EBL file format as serial bootloading, the provided sample code (plugins) for standalone OTA bootloading uses the Zigbee Cluster Library (ZCL) standard OTA file format for wrapping the image in a non-proprietary format and storing it in the source device’s storage drivers. For more details about this file format, refer to Zigbee document #07-5123, *ZigBee Cluster Library Specification*, revision 6 or higher, specifically the “Over-the-air Upgrading” chapter. This document is available from http://www.zigbee.org. For information on creating .ota files from .ebl files using silicon Labs’ Image Builder tool, see *AN716: Instructions for Using Image Builder*.

3. Upload Recovery

If an image upload fails, the target node is left without a valid application image. Typically, failures are related to over-the-air transmission errors. When an error occurs, the bootloader restarts and continues to listen on the same channel for any retries by the source device. It remains in recovery mode until it successfully uploads the application image.

Note: If a hard reset occurs before the bootloader receives a new valid image, or the bootloader is launched by the hardware trigger, the target device enters bootload recovery mode. In this mode, the bootloader does not automatically boot into the application firmware's reset vector but instead waits indefinitely for a new image transfer to begin. Additionally, if the device's standalone bootloader has OTA target capability, it will activate the radio and begin listening on the default channel (IEEE 802.15.4 channel 13, centered at 2.415 GHz) for a new upload to begin.

3.1 Initiating Recovery Mode Manually

Regardless of whether the device's standalone bootloader firmware has OTA target capability, and regardless of the serial interface supported by your standalone bootloader, a GPIO-based trigger can be used to facilitate recovery mode via serial upload. On EM35x devices, a hardware-level function for this kind of recovery is provided via PA5 (nBOOTMODE), a GPIO whose primary use is typically PTI_DATA. Holding this pin low during power-up or across a reset and then sending a carriage return at 115200 baud launches the standalone bootloader.

In USB-based bootloaders for EM358x/9x, no hardware-based trigger can be used. This is because the hardware-based trigger described above for EM35x devices uses the FIB Monitor Mode, which is serial over UART only. In these cases, forced serial recovery is only possible with software using other IO pins, as described below.

You can configure your standalone bootloader to use a software-based GPIO pin check or other schemes of recovery mode activation by modifying the `bootloadForceActivation()` API in bootloader-gpio.c (or bootloader-gpio-ezsp-spi.c in the case of the EZSP-SPI bootloaders) and rebuilding the bootloader from the provided project files. An example is provided in bootloader-gpio.c utilizing PC6 for EM35x, which is connected to a button on the EM35x Breakout Board. This button-driven recovery code can be enabled by building the bootloader with `USE_BUTTON_RECOVERY` defined.

Note: For EZSP-SPI-based bootloaders, `USE_BUTTON_RECOVERY` is already defined (and the code included) by default, but the default recovery pin in those cases is based on the GPIO normally used for nWAKE in the EZSP-SPI protocol, for example PB6 for EM35x. See bootloader-gpio-ezsp-spi.c for customization.

3.2 Recovery of OTA targets

After the source device identifies a device that is in recovery mode (usually via broadcast Query mechanism), it resumes the upload process as follows:

1. The source application starts the download process again from the Query. However, the source device first needs to ensure that it is on the same channel as the device to be recovered.
2. The source device sends an XMODEM_QUERY message to the target device.
3. The target device bootloader extracts and saves the source device's destination address and PAN ID, and responds with a query response.
4. When the source device receives the query response in `emberIncomingBootloadMessageHandler()`, it checks the target device's EUI, protocol version, and whether the target device is already running the bootloader. The library handles the process of reading the programmed flash pages for the current application image and sends them to the target device.

4. Standalone Bootloader Plugin API

4.1 Bootloader Over-the-Air Launch

The Standalone Bootloader Client, Common, and Server plugins in the Silicon Labs Zigbee application framework provide client-side (target) and server-side (source) implementations of a standard mechanism for the standalone bootloader's proprietary, over-the-air launch and transfer process. This mechanism is compatible with Simplicity Studio's AppBuilder and restricts bootloader launch to trusted devices only. This process is summarized in the figure on the following page.

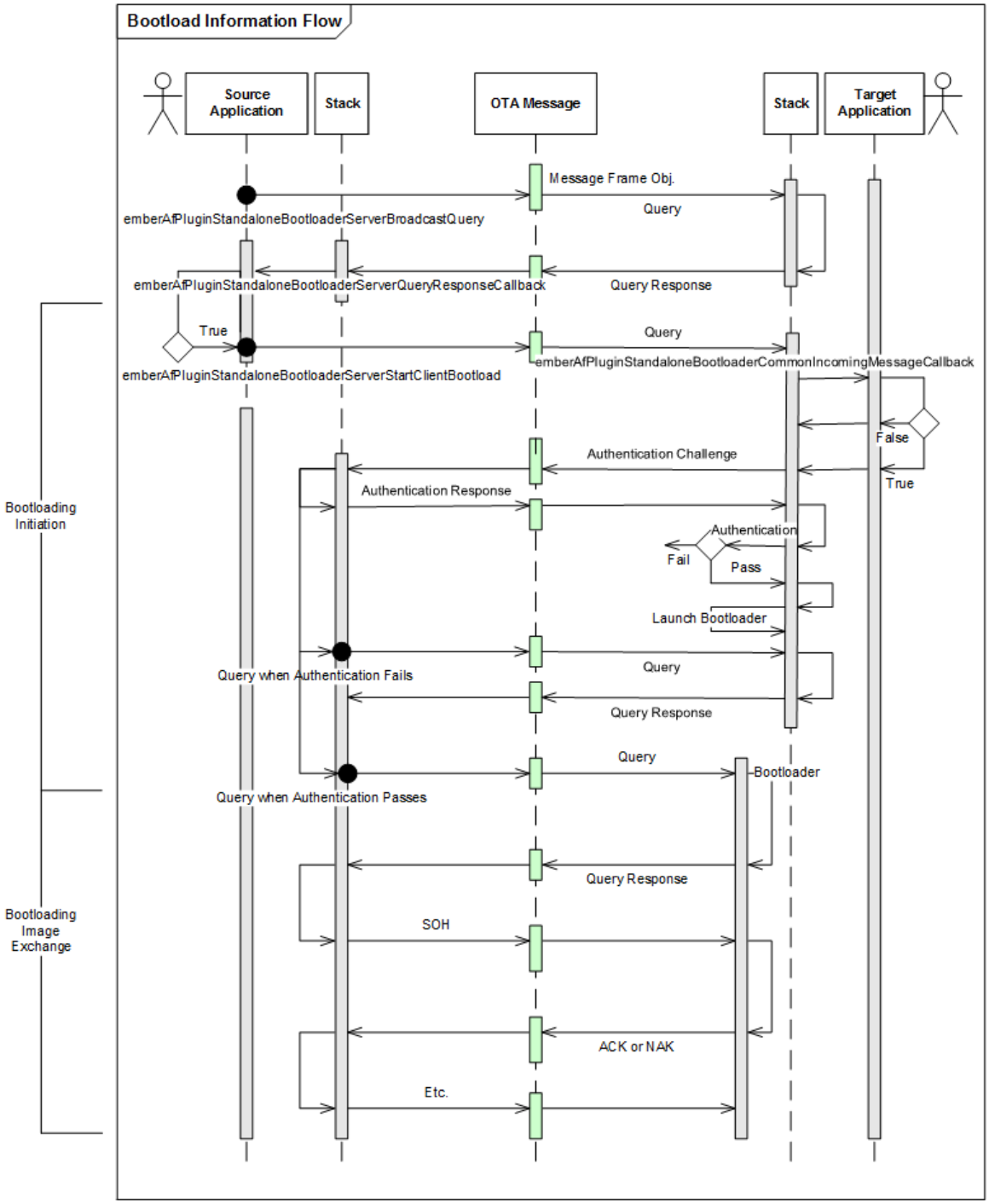


Figure 4.1. Standalone Bootloading Initial OTA Information Flow

Before you can update the image on a device that has an application running, its bootloader must be launched. The process typically follows these steps:

1. The source device queries the network to determine which devices require updating, by issuing an APS message to nodes of interest. Responding devices return their application version. The source device evaluates this information and identifies potential target devices accordingly.
2. The source device queries each potential target device by calling into the Standalone Bootloader Server plugin APIs with `emberAfPluginStandaloneBootloaderServerBroadcastQuery` or `emberAfPluginStandaloneBootloaderServerStartClientBootload`. These functions can initiate a broadcast or unicast query message, depending on whether the target device's EUI is known.

3. For each queried device, the Standalone Bootloader Server plugin's `emberAfPluginStandaloneBootloaderServerQueryResponseCallback()` is invoked, which returns the following information to the source device:
   - Whether the target is in application or bootload mode.
   - Device type for the target, including platform, micro, phy, and board designations.
4. Depending on the query results, the source device can send a bootloader launch request message to a target device by calling `emberAfPluginStandaloneBootloaderServerStartClientBootload()`.

   You supply the following arguments:
   - Target device's EUI64: Identifies the device to upgrade.
   - An image ID for an OTA image file residing on the source's storage system.
   - An image data tag referencing a specific firmware image within the referenced .ota image file.
5. On receiving the launch request, the target device calls the bootloader launch handler code from within the Standalone Bootloader Client plugin's `emberAfPluginStandaloneBootloaderCommonIncomingMessageCallback`. This code examines the information supplied by the target device—manufacturer and hardware IDs, radio signal strength, or other metrics—and determines whether the application should allow the request. The handler returns either true (launch the bootloader) or false. If false, the transaction completes when the source device times out waiting for the authorization challenge.
6. If the target device launch handler returns true, the Standalone Bootloader Client plugin on the target calls `sendChallenge()`, which sends an authorization challenge message to the Standalone Bootloader Server (source device). This challenge contains the target device's EUI64 and random data.
7. When the source device receives the challenge, it calls the Standalone Bootloader Server's `processChallenge()` subroutine, which uses the AES block cipher and the encryption key saved from the earlier request, and encrypts the challenge data.
8. Assuming the challenge authorization came back successfully from the source device, the target calls the Standalone Bootloader Client's `emberAfPluginStandaloneBootloaderClientAllowBootloadLaunchCallback()` to give the application the change to accept or deny the already-authorized launch request, which, if accepted, will cause the device to exit the normal application firmware and enter the standalone bootloader in OTA target mode.

4.2 Plugin Constraints

The following constraints apply to the Standalone Bootloader plugins:

- The Client and Server plugins do not support multi-hop transfers.
- The Server code requires that the firmware file for the target be an OTA file formatted as per the Zigbee Cluster Library (ZCL) specification, with the low-level EBL file data contained within the OTA file as an image data tag with tag ID 0x0000. For more details about this file format, refer to Zigbee document #07-5123, *Zigbee Cluster Library Specification*, revision 6 or higher, specifically the "Over-the-air Upgrading" chapter. This document is available from http://www.zigbee.org. For information on creating OTA files from .ebl files using the Silicon Labs Image Builder tool, see document *AN716: Instructions for Using Image Builder*.
- The Server code requires that the OTA file be accessible to the Server application (on the source device for the bootload) over some kind of locally accessible storage mechanism, such as a filesystem, external EEPROM, or serial dataflash. Existing storage driver implementations can be found as plugins in the Silicon Labs Zigbee application framework. These plugins contain the word "Storage" in the name and are grouped under the Zigbee OTA Bootloading category. See *UG391: Zigbee Application Framework Developer's Guide* for more details about available storage driver plugins. Alternatively, developers may author their own storage mechanisms in place of these plugins, as long as the same storage API callbacks are implemented to facilitate image access.

5. Manufacturing Tokens

If OTA target functionality is supported by your chosen standalone bootloader, and if you intend to use it, then you must set several manufacturing tokens. Note that a special area of flash is used to store these tokens, so they cannot be written by an application at runtime. The EM35x uses em3xx_load.exe to set these tokens.

See document *AN708: Setting Manufacturing Certificates and Installation Codes for the EM35x SoC and Coprocessor Platforms*, and *AN710: Bringing up Custom Devices for the Ember® EM35x SoC or NCP Platform*, for more information on setting manufacturing tokens. The tokens that need to be set are:

`TOKEN_MFG_BOARD_NAME` - Synonymous with the hardware tag used to identify nodes during the bootloader protocol. This tag serves two purposes:

- Applications can query devices for their hardware tags and can determine which devices to bootload accordingly.
- When a device performs a Launch Request to request that a target device switch to bootloader mode, it supplies the target's hardware tag as an argument. The target can use this tag to determine whether to refuse to launch its bootloader if it believes the requesting device is trying to program it with software for another hardware type. Each customer is responsible for programming this value.

`TOKEN_MFG_MANUF_ID` - A 16-bit (2-byte) string that identifies the manufacturer. This tag serves two purposes:

- Applications can query devices to obtain their manufacturer ID, and decide whether to bootload a device accordingly.
- When a device performs a Launch Request to request that a target device switch to bootloader mode, it supplies the target's manufacturer ID as an argument. The target can refuse to launch its bootloader if it believes the requesting device is trying to program it with software for another manufacturer.

Each customer is responsible for programming this value. Customers are encouraged to use the 16-bit manufacturer's code assigned to their organization by the Zigbee Alliance. This value is typically also used with the EmberZNet PRO stack's `emberSetManufacturerCode()` API call (stack/include/ember.h) or `ezspSetManufacturerCode()` in the EZSP host API to set the manufacturer ID used as part of the Simple Descriptor by the Zigbee Device Object (ZDO).

`TOKEN_MFG_PHY_CONFIG` - Configures operation of the alternate transmit path of the radio, which is sometimes required when using a power amplifier. This token should be set as described in *AN710: Bringing up Custom Devices for the Ember® EM35x SoC or NCP Platform*, or else the bootloader may not operate correctly in a recovery scenario. Each customer is responsible for programming this value.

`TOKEN_MFG_BOOTLOAD_AES_KEY` - The 16-byte AES key used during the bootloader launch authentication protocol. Each customer is responsible for programming this value and keeping it secret. Silicon Labs ships with the AES key set to all 0xFF. The sample application also uses this value. If the value is changed, be sure to modify the application as well.

6. OTA Standalone Bootloader Packets

The following sections describe the format of the packets used during over the air bootloading.

6.1 Broadcast Query

The following table describes the format of the broadcast query message.

Table 6.1. Broadcast Query Message Format

# bytes	Field	Description/notes
1	Length	Packet length (does not include the length byte)
2	Frame control field	Short destination, long source, inter PAN, command frame
1	Sequence number	
2	Destination PAN ID	Always set to broadcast address 0xFFFF
2	Destination address	Always set to broadcast address 0xFFFF
2	Source PAN ID	
8	Source EUI64	
1	MAC command type	Always set to 0x7C (an invalid 15.4 command frame chosen for bootload packets)
2	Signature	Always set to em; used as further validation in addition to mac command type
1	Version	Version of the bootloader protocol in use, currently set to 0x0001
1	Bootloader command	Always set to 0x51 for query
2	Packet CRC	

6.2 Common Packet Header

Many of the message packets use a common header format. The following table describes the format of this common header.

Table 6.2. Common Header for All Other Message Types

# bytes	Field	Description/notes
1	Length	Packet length (does not include the length byte)
2	Frame control field	Long destination, long source, intra PAN, ACK request, command frame
1	Sequence number	A unique identifier for each MAC layer transaction
2	Destination PAN ID	
8	Destination EUI64	
8	Source EUI64	
1	MAC command type	Always set to 0x7C (an invalid 15.4 command frame chosen for bootload packets)
2	Signature	Always set to em; used as further validation in addition to MAC command type
1	Version	Version of the bootloader protocol in use, currently set to 0x0001
n	Data	Remainder of packet

6.3 Query Packet

Table 6.3. Query Packet

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x51 query
2	Packet CRC	

6.4 Query Response

Table 6.4. Query Response

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x52 query response
1	Bootloader active	0x01 if the bootloader is currently running; 0x00 if an application is running
2	Manufacturer ID	
16	Hardware tag	
1	Bootloader capabilities	0x00
1	Platform	0x02 xap2b, 0x04 Cortex-M3
1	Micro	0x03 em357, 0x05 em351
1	PHY	0x03 em3x
2	blVersion	Optional field. Contains the remote standalone bootloader version. The high byte is the major version; low byte is the build.
2	Packet CRC	

6.5 Bootloader Launch Request

Table 6.5. Bootloader Launch Request

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x4C launch request
2	Manufacturer ID	
16	Hardware tag	
2	Packet CRC	

6.6 Bootloader Authorization Challenge

Table 6.6. Bootloader Authorization Challenge

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x63 authorization challenge
16	Challenge data	
2	Packet CRC	

6.7 Bootloader Authorization Response

Table 6.7. Bootloader Authorization Response

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x72 authorization response
16	Challenge response data	
2	Packet CRC	

6.8 XModem SOH

Table 6.8. XModem SOH

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x01 XModem SOH
1	Block number	
1	Block number one's complement	
64	Data	
2	Block CRC	
2	Packet CRC	

6.9 XModem EOT

Table 6.9. XModem EOT

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x04 XModem EOT
2	Packet CRC	

6.10 XModem ACK

Table 6.10. XModem ACK

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x06 XModem ACK
1	Block number	
2	Packet CRC	

6.11 XModem NACK

Table 6.11. XModem NACK

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x15 XModem NACK
1	Block number	
2	Packet CRC	

6.12 XModem Cancel

Table 6.12. XModem Cancel

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x18 or 0x03 XModem cancel (from source)
2	Packet CRC	

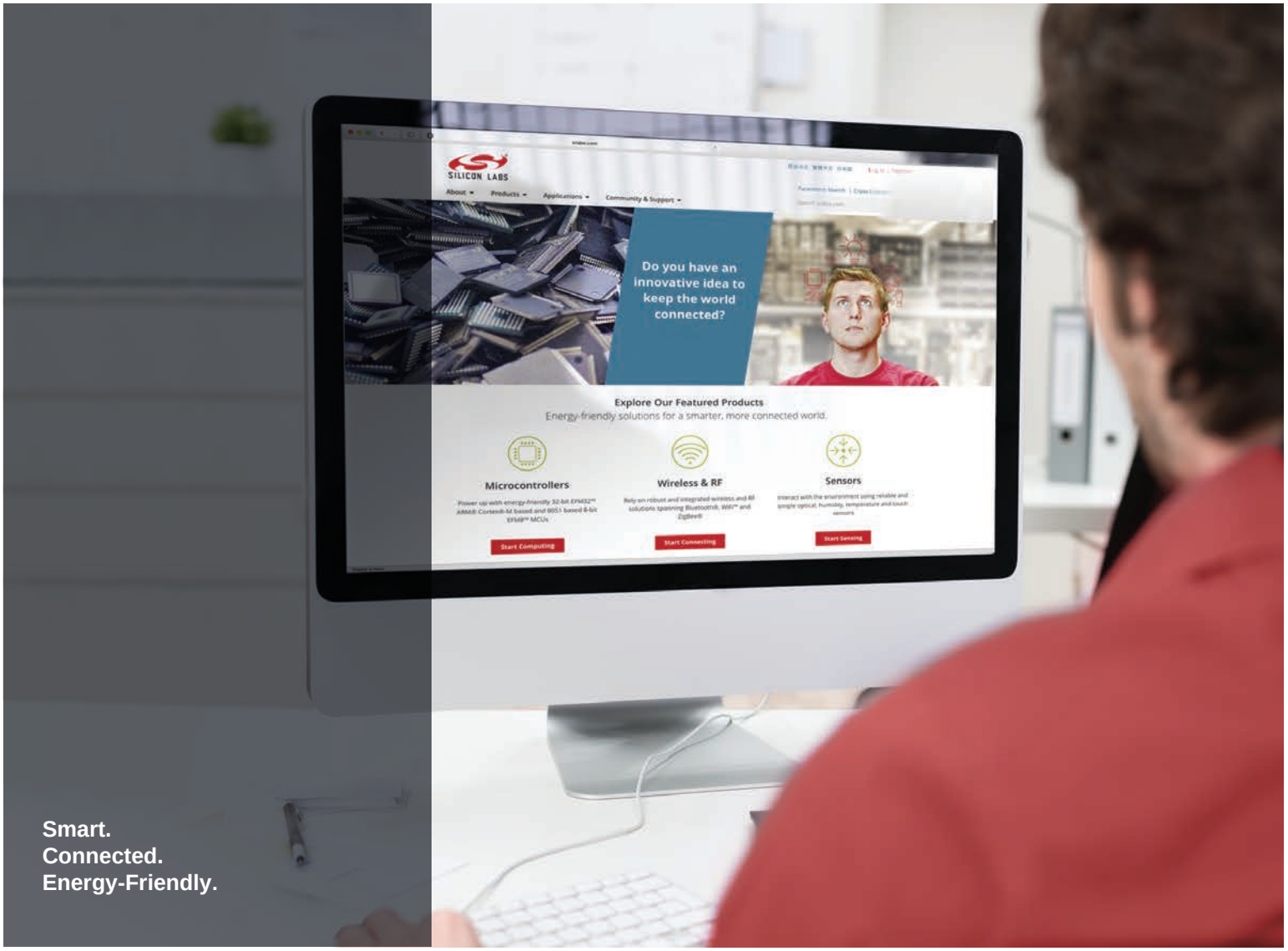
6.13 XModem Ready

Table 6.13. XModem Ready

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x43 XModem ready
2	Packet CRC	

7. Creating an Ember Standalone Bootload Application Using AppBuilder

See *AN728: Over-the-Air Bootload Server and Client Setup* for a full example of how to use the Ember standalone bootloader to bootload a client.



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