AN760

USING THE STANDALONE BOOTLOADER

This application note describes the implementation of the Silicon Labs-proprietary Standalone Bootloader, a special firmware image intended to reside on a chip separately from the ZigBee application/stack firmware. It is designed as a simple, dedicated program to facilitate input of new application/stack firmware via Xmodem upload to a serial interface (SPI, UART, or USB) or via a proprietary, IEEE 802.15.4-based, single-hop MAC layer RF protocol.

New in This Revision

Update for EM359x release.

Contents

1.1 Modes - Serial / OTA

The standalone bootloader and its utility library support three basic modes for uploading an application image to a network device:

- Serial upload
- Over-the-air upload
- Hybrid mode uploads

Note: The standalone serial USB bootloader for EM358x/359x devices does not support over-the-air or hybrid upload modes.

1.1.1 Serial Upload

You can establish a serial connection between a PC and a target device's serial interface and upload a new software image to it using the XModem protocol, as shown in Figure 1. If you need information on the XModem protocol there is plenty of documentation online. A good place to start would be http://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up to date links to protocol documentation.

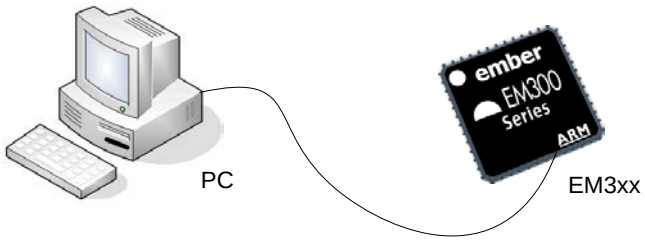

Figure 1. Serial Upload

1.1.1.1 Serial Upload

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

To open serial connections over USB (only available on EM358x/359x devices that support USB), the PC connects to the Windows COM port that the target device creates in the PC. When connecting to this Windows COM port, baud, data bits, parity bits, and stop bits are irrelevant.

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 is established:

1. The target device's bootloader sends output over its serial port after it receives a carriage return from the PC. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following options:

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

After listing the menu options, the bootloader's "BL >" prompt displays.

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.

1.1.1.2 Serial Upload: Uploading an Image

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 serial line, 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. After an image successfully uploads, the XModem transaction completes and the bootloader displays 'Serial upload complete' before redisplaying the menu.

1.1.1.3 Serial Upload: Errors

If an error occurs during the upload, the bootloader will display the message 'Serial upload aborted' followed by a more detailed message and one of the hex error codes shown in Table 7 2. It will then redisplay the bootloader menu.

Table 1. Serial Uploading Error Messages EM250 & EM35x

Hex code	Constant	Description
0x21	BLOCKERR_SOH	The bootloader encountered an error while trying to parse the start of header (SOH) character in the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected an invalid checksum in the XModem frame.

Hex code	Constant	Description
0x23	BLOCKERR_CRCH	The bootloader encountered an error while trying to parse the high byte of the CRC in the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while trying to parse the low byte of the CRC in the XModem frame.
0x25	BLOCKERR_SEQUENCE	The bootloader encountered an error in the sequence number of 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	GOT_DUP_OF_PREVIOUS	The bootloader encountered a duplicate of the previous XModem frame.
0x41	BL_ERR_HEADER_EXP	No .EBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Header failed CRC.
0x43	BL_ERR_CRC	File failed CRC.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in .EBL image.
0x45	BL_ERR_SIG	Invalid .EBL header signature.
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.
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	An invalid tag was found in the .EBL image.

1.1.1.4 Running the Application Image

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.

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

1.1.1.5 Obtaining Image Information

On the EM250, bootloader menu option 3 (ebl info) displays platform information about the uploaded image, in the following format:

platform-micro-phy-board

For example, the bootloader shows the following data about an image that is built for the Ember EM250 RCM and breakout board:

xap2b-em250-em250-dev0455

On the EM35x platform, the image info is customizable by the user, and can be specified as a string using the --imageinfo option in the em3xx_convert utility, which creates the EBL image from an s37 file. Menu option 3 then displays the information as a quoted string, similar to the following:

"custom image info"

For both the EM35x and EM250 the information displayed by these commands represents the image that is currently stored in the flash. This means that after a successful bootload this information should change to reflect the new application.

1.1.2 Over-the-Air Upload

You can upload images over the air to a target device in several ways:

- Passthrough
- Multi-hop Passthrough

In all cases, the source device must be within radio range of the target device. The uploaded image can originate from a PC or some other device that sends the image to a network device over a serial line.

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 devices that participate in the upload process continue to run an application that is based on the EmberZNet PRO stack.

Note: If a target device gets a carriage return from its serial port while it awaits over-the-air bootloader packets from another device, and if no over-the-air bootloader-formatted packets have arrived, the device's bootloader switches to serial mode and ignores any subsequent over-the-air packets.

1.1.2.1 Passthrough

When using over-the-air passthrough mode, the source node is connected to a PC by a serial cable. The source receives an image over the serial line and passes the image to the target node over the air, as shown in Figure 2.

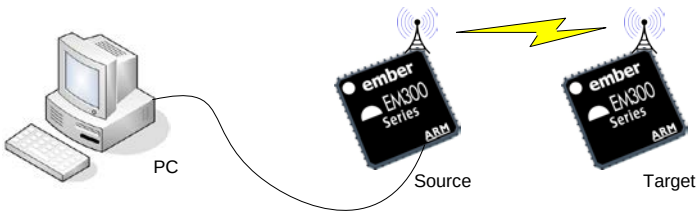

Figure 2. Over-the-air Passthrough Mode

The PC is expected to transfer the .EBL image to the source node using standard 128-byte XModem CRC packets. The source node application must split these packets into the special 64-byte XModem format that the Ember standalone bootloader uses for its over-the-air protocol.

1.1.2.2 Passthrough: Upload Process

As soon as the source node verifies that the target node is running the bootloader, it starts the upload process by calling `bootloadUtilStartBootload()` with the mode parameter set to `BOOTLOAD_MODE_PASSTHRU`.

After receiving the query response from the target node, the source node calls `emberIncomingBootloadMessageHandler()`, which directs the serial download by calling `XModemReceiveAndForward()`.

For more detailed information, see the sample code in the bootloader utility library (/app/util/bootload).

1.1.2.3 Multi-Hop Passthrough

Multi-hop passthrough mode is an extension of standard over-the-air passthrough, and is used when an image must travel longer distances, as shown in Figure 3. In this mode, a gateway device receives an image over the serial line, then uses standard networking protocols to forward the image to a network device. The device converts the ZigBee-formatted messages to the bootloader's over-the-air link-layer protocol, then forwards the image to the target.

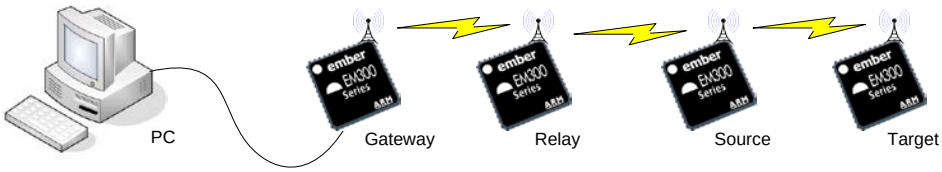

Figure 3. Over-the-air Multi-hop Passthrough Mode

1.1.3 Hybrid Mode Uploads

Upload operations can mix any of the modes previously described. For example, a source node might acquire an image by some other means—for example, from a set of images stored on an externally connected serial flash.

1.2 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 node. It remains in recovery mode until it successfully uploads the application image.

Note: The serial USB bootloader for EM358x/359x devices does not support over-the-air or hybrid upload modes.

If a hard reset occurs before the bootloader receives a new valid image, or the bootloader is launched by the hardware trigger, the target node enters bootload recovery mode. In this mode, the bootloader listens on the default channel (13) for a new upload to begin. The primary use of GPIO5 on the EM250 is PTI_DATA (Packet Trace Data) supplied to the Debug Adapter while the application is running. During module power up, holding GPIO5 low triggers the emergency recovery mode. Optionally, GPIO5 can be used as the ADC1 input. Document 120-0250-000, *EM250 Data Sheet*, provides full details on the hardware functionality of GPIO5.

The EM260 uses the nWAKE and PTI_DATA pins to trigger recovery mode on power up. See document 120-0260-000, *EM260 Data Sheet* (chapter 5.6.1), for details.

The EM35x uses PA5 as a hardware-based trigger for recovery mode when serial over UART is being used. As in the EM250, the primary use of this pin is 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.

Warning: On EM358x/359x chips there is no hardware-based trigger for the serial USB bootloader. The hardware-based trigger described for EM35x devices uses the FIB monitor which is serial over UART only. Recovery is only possible with software using other IO pins.

The EM35x platform can also be configured to use other IO pins or other schemes of activation by modifying the `bootloadForceActivation()` API in bootloader-gpio.c and rebuilding the bootloader from the provided project files. An example of utilizing PC6, which is connected to a button on the EM35x Breakout Board, is provided in bootloader-gpio.c and can be enabled by building the bootloader with USE_BUTTON_RECOVERY defined.

After the source node identifies a node that is in recovery mode, it resumes the upload process as follows:

4. The source application starts the download process by calling `bootloadUtilStartBootload()`. Before calling the function, the source node needs to ensure that it is on the same channel as the node to be recovered. For EM250 applications, the source node can leave the current channel and join or form the network on the recovering channel. For EM260 host applications, the source node can call `ezspOverrideCurrentChannel()` to change the channel to the recovering channel. In case of default channel recovery, the source node needs to be on bootload default channel (13).
5. The source node sends an XMODEM_QUERY message to the target node.
6. The target node bootloader extracts and saves the source node's destination address and PAN ID, and responds with a query response.
7. When the source node receives the query response in `emberIncomingBootloadMessageHandler()`, it checks the target node's EUI, protocol version, and whether the target node 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 node.

1.3 Bootloader Utility Library API

The bootloader utility library, in /app/util/bootload, provides APIs that source and target node applications can use to interact with a standalone bootloader. The library is supplied as source code.

For details on the bootloader utility, see the library source code and the supplied standalone-bootloader-demo application.

Note: Silicon Labs recommends that you do not modify the supplied utilities.

The following sections in this document discuss programming requirements for using the standalone bootloader.

- Library interfaces
- Application requirements
- Bootloader over-the-air launch
- Library constraints

The bootloader utility library contains the following interfaces, defined in bootload utils.h:

Note: Applications that use bootload utilities must define EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS and EMBER_APPLICATION_HAS_RAW_HANDLERS in their CONFIGURATION_HEADER.

1.3.1 Functions

```
void bootloadUtilInit(
   int8u appPort,
   int8u bootloadPort
);

EmberStatus bootloadUtilSendRequest(
   EmberEUI64 targetEui,
   int16u mfgId,
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   int8u encryptKey[BOOTLOAD_AUTH_COMMON_SIZE],
   int8u mode
);
```

```
void bootloadUtilSendQuery(
   EmberEUI64 target
);

void bootloadUtilStartBootload(
   EmberEUI64 target,
   bootloadMode mode
);

void bootloadUtilTick(void);
```

1.3.2 Callbacks

```
boolean bootloadUtilLaunchRequestHandler(
   int16u manufacturerId,
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   EmberEUI64 sourceEui
);

void bootloadUtilQueryResponseHandler(
   boolean bootloaderActive,
   int16u manufacturerId,
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   EmberEUI64 targetEui,
   int8u bootloaderCapabilities,
   int8u platform,
   int8u micro,
   int8u phy,
   int16u blVersion
);

#define IS_BOOTLOADING ((blState != BOOTLOAD_STATE_NORMAL) && \
                        (blState != BOOTLOAD_STATE_DONE))
```

1.3.3 Application Requirements

To enable over-the-air bootloader launch, network node applications must:

- Include bootload-utils.h in the application's .h file.
- Include the bootload-utils.c file in the project.
- Define EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS and USE_BOOTLOADER_LIB in the application's configuration file.
- Implement these handlers:
  - `bootloadUtilLaunchRequestHandler()`
  - `bootloadUtilQueryResponseHandler()`
- Call `bootloadUtilInit()` before `emberNetworkInit()` but after `emberInit()` when the application starts.
- **Note:** If port 1 serves as the bootloader port, `bootloadUtilInit()`changes the baud rate to 115200.Call `bootloadUtilTick()` in a heartbeat function.

The standalone bootloader demonstration application code and libraries allow building different configuration variants depending on what is needed by the application and the code space available. The default is to build using the complete solution.

The following configurations are available when configuring the standalone bootloader demo application and library code. If necessary, you can reduce flash requirements by eliminating features not needed. Typical uses are to remove V1 bootloader support (if not supporting bootloading of legacy Ember devices) or to remove passthrough support.

Modify the following definitions in bootloader-demo-v2-configuration.h:

USE_BOOTLOADER_LIB // always defined with the bootloader demo library

SBL_LIB_SRC_NO_PASSTHRU // define this to build a node without passthrough ability

SBL_LIB_TARGET // define this to build a target only node.

?EMBER_APPLICATION_HAS_RAW_HANDLER // configures V1 bootloader protocol support

EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS // leave this defined unless app does not supply incoming message handler. Not defined adds message and transmit complete stubs.

1.3.4 Bootloader Over-the-Air Launch

The bootloader utility library in /app/util/bootload provides the implementation of a standard mechanism for over-the-air bootloader launch. This mechanism is compatible with Ember Desktop, and restricts bootloader launch to trusted devices only. This process is summarized in Figure 4.

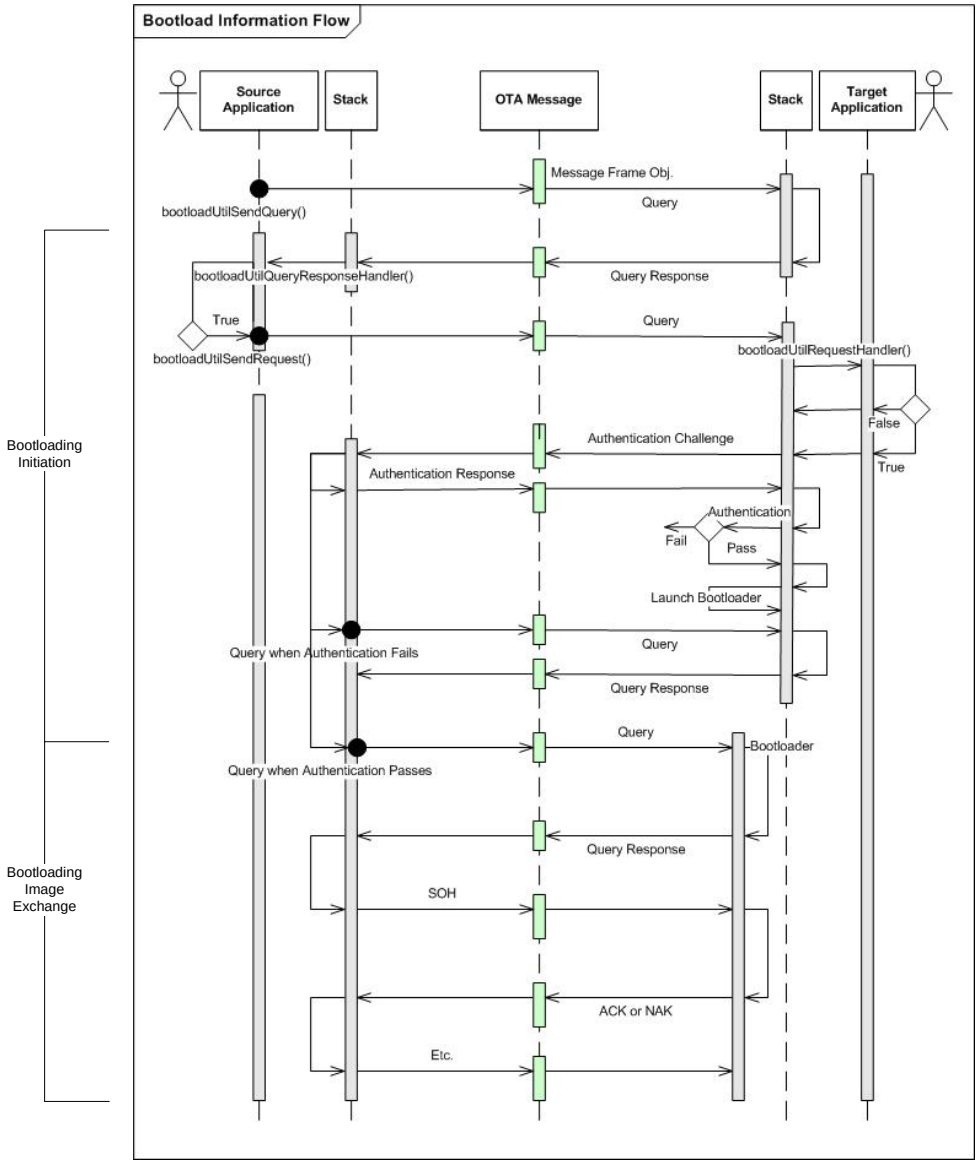

Figure 4. Standalone Bootloading Initial Information Flow

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:

8. The source node typically queries the network to determine which nodes require updating, by issuing an APS message to nodes of interest. Responding nodes return their application version. The source node evaluates this information and identifies potential target nodes accordingly.
9. The source node queries each potential target node by calling `bootloadUtilSendQuery()`. This function can initiate a unicast or broadcast message, depending on the argument supplied—NULL (0xFF) for broadcast, or the target node's EUI for unicast.

10. On each queried node, the application-supplied handler `bootloadUtilQueryResponseHandler()` is invoked, which returns the following information to the source node:
    - Whether it is in application or bootload mode
    - Device type, including platform, micro, phy, and board designations
11. Depending on the query results, the source node can send a bootloader launch request message to a target node by calling `bootloadUtilSendRequest()`.

    You supply the following arguments:
    - Target node's EUI64: Identifies the node to upgrade.
    - Manufacturer's ID: The manufacturer's unique product identifier.
    - Hardware tag: The manufacturer's unique hardware identifier.
    - Encryption key: Manufacturer-supplied, the target node uses this to verify that the source node is authorized to initiate the request.
    - Desired mode: Set to `BOOTLOAD_MODE_PASSTHRU`.
12. On receiving the launch request, the target node calls the bootloader launch handler `bootloadUtilLaunchRequestHandler()`. This handler examines the information supplied by the target node—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 node times out waiting for the authorization challenge.
13. If the target device launch handler returns true, the target application calls `bootloadSendAuthChallenge()`, which sends an authorization challenge message to the source device. This challenge contains the target device's EUI64 and random data.
14. When the source device receives the challenge, it calls `bootloadUtilSendAuthResponse()`, which uses the AES block cipher and the encryption key saved from the earlier request, and encrypts the challenge data.

1.3.5 Library Constraints

The following constraints apply to the bootloader utility library:

- The library does not support multi-hop downloads.
- The library code takes over the serial port on the source node during serial uploads. If the application uses the serial port, it might need to reconfigure this port when the upload is complete.
- Future releases might require changes to bootloader utility APIs.

1.4 Manufacturing Tokens

The bootloader requires you to 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 and the EM250 uses em2xx_patch.exe to set these tokens.

See document AN708, *Setting Manufacturing Certificates for the EM35x*, as well as documents 120-5031-000, *Bringing Up Custom Devices for the EM250 SoC Platform*, 120-5041-000, *Bringing Up Custom Devices for the EM260 Co-Processor*, and AN710, *Bringing Up Custom Devices for the EM35x SoC 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 nodes for their hardware tags and can determine which nodes to bootload accordingly.
- When a node calls `bootloadUtilSendRequest` to request that a target node 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 node 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 nodes to obtain their manufacturer ID, and decide whether to bootload a node accordingly.
- When a node calls `bootloadUtilSendRequest` to request that a target node 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 node 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) 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 the Bringing Up Custom documentation for your 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 too.

1.5 Example Standalone Bootloading Scenario

1.5.1 Standalone Bootloader

The standalone bootloader demo shows how to integrate the bootloader utility library into an application. It shows how to trigger all modes of operation, and uses a simple command interface to manually drive the bootloader. You can use the standalone bootloader demo as a development tool or starting point for your own application that will update your device images over the air.

The application features various commands over the serial port to exercise all available bootload features. It does not use any buttons.

1.5.2 Serial Baud Rates and Ports Used

The application uses a Debug Adapter feature called virtual UART to communicate with users. All commands are transmitted and received through (TCP) port 4900. Serial port 1 (TCP port 4901) is configured to be the bootload port at 115,200 bps. Table 2 lists the serial commands supported.

If you change the settings for serial port 1 in your Debug Adapter's administrative configuration interface, you might need to adjust your configuration to match what this application expects. Refer to document 120-2010-000, *Ember Debug Adapter Technical Specification*, for information about adjusting these settings.

Note: Both the application and bootload ports can be configured for the same port. However, application usage of the port needs to be limited when bootloading is in progress in order to maximize performance and to avoid any interruption to the bootload process.

Table 2. Serial Commands Supported

Command	Description
`default mode`	Recover nodes that fail bootloading on the default channel (13), where mode is set to 1 (passthrough).
`Form`	Form a network.
`Join`	Join the network as router.
`Leave`	Leave the network.
`query_neighbor`	Report bootload-related information about itself and its neighbor. This information helps determine which node to bootload.
`query_network`	Obtain application information about itself and other nodes in the network.
`recover target-EUI64 mode`	Recover nodes that fail bootloading on current channel, where mode is set to 1 (passthrough).
`remote target-EUI64`	Upload an application image to the specified remote node. The image is obtained from the serial port using XModem protocol.
`serial`	Put the node in serial bootload mode.

1.5.3 Usage Notes

15. To start remote bootloading, select the .EBL file to download using a terminal emulator program (such as Hyperterminal) connected to the node's serial port. The node signals the event by printing a stream of C characters to serial port 1.
16. This standalone bootloader demo sample application can be used together with the demo Java application BootloaderDemoV2.java (see section 7.3.9.4, Sample Upload Application).

1.5.4 Sample Upload Application

The Ember distribution provides a sample Java application for over-the-air uploads in /tool/standalone-bootloader. This application lets users gather information and perform over-the-air bootloading on network nodes through a gateway node as follows:

17. The application talks to the gateway node over TCP/IP.
18. The gateway node talks to other nodes in the network over-the-air.

This sample application supports the following tasks:

19. Querying neighbor nodes to determine which nodes require bootloading.
20. Serial bootloading on the gateway node.
21. Over-the-air passthrough bootloading on a remote node.
22. Recovery and default recovery on a remote node.

1.6 OTA Standalone Bootloader Packets

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

1.6.1 Broadcast Query

Table 3 describes the format of the broadcast query message.

Table 3. 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	

1.6.2 Common Packet Header

Many of the message packets use a common header format. Table 4 describes the format of this common header.

Table 4. 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

1.6.3 Query Packet

Table 5. Query Packet

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

1.6.4 Query Response

Table 6. 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	0x01 em250, 0x03 em357, 0x05 em351
1	PHY	0x02 em250, 0x03 em3xx
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	

1.6.5 Bootloader Launch Request

Table 7. 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	

1.6.6 Bootloader Authorization Challenge

Table 8. Bootloader Authorization Challenge

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

1.6.7 Bootloader Authorization Response

Table 9. Bootloader Authorization Response

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

1.6.8 XModem SOH

Table 10. 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	

1.6.9 XModem EOT

Table 11. XModem EOT

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

1.6.10 XModem ACK

Table 12. XModem ACK

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

1.6.11 XModem NACK

Table 13. XModem NACK

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

1.6.12 XModem Cancel

Table 14. XModem Cancel

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

1.6.13 XModem Ready

Table 15. XModem Ready

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

2 Creating a Standalone Bootload Application via AppBuilder

See the application note *AN728 - Over-the-Air Bootload Server and Client Setup Using EM35x Development Kits* for a full example of how to use the Standalone bootloader to bootload a client.

CONTACT INFORMATION

Silicon Laboratories Inc.

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

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