UG235: Silicon Labs Connect User's Guide

This user guide provides in-depth information for developers who are using the Silicon Labs Connect stack as a foundation for their application development on the Wireless Gecko (EFR32™) portfolio or custom hardware. The Connect stack is delivered as part of the Silicon Labs Flex SDK. This guide assumes that you have already installed the Simplicity Studio development environment and the Flex SDK, and that you are familiar with the basics of configuring, compiling, and flashing Connect-based applications.

If you are just getting started with Connect and the Flex SDK, see *QSG138: Getting Started with the Silicon Labs Flex Software Development Kit for the Wireless Gecko (EFR32™) Portfolio* for an introduction.

KEY POINTS

- About Connect plugins.
- Using Bootloaders.
- Using the Serial Bootloader.
- Using the Connect OTA Broadcast and Unicast Bootloaders.
- About Frequency Hopping.
- Using the Mailbox plugin.
- Working with sleepy devices.
- Working with direct devices.
- Using a custom board header.

1. Introduction

The Silicon Labs Connect stack provides a fully-featured, easily-customizable wireless networking solution optimized for devices that require low power consumption and are used in a simple network topology. Connect is configurable to be compliant with regional communications standards worldwide. Each RF configuration is designed for maximum performance under each regional standard.

The Silicon Labs Connect stack supports many combinations of radio modulation, frequency and data rates. The stack provides support for end nodes, coordinators, and range extenders. It includes all wireless MAC (Medium Access Control) layer functions such as scanning and joining, setting up a point-to-point or star network, and managing device types such as sleepy end devices, routers, and coordinators. With all this functionality already implemented in the stack, users can focus on their end application development and not worry about the lower-level radio and network details.

The Silicon Labs Connect stack should be used in applications with simple network topologies, such as a set of data readers feeding information directly to a single central collection point (star or extended star topology), or a set of nodes in the same range exchanging data in a single-hop fashion (direct devices). It does not provide a full mesh networking solution such as that provided by the EmberZNet PRO or Silicon Labs Thread stack.

The Silicon Labs Connect stack supports efficient application development through its “building block” plug-in design. When used with Silicon Labs Application Builder, developers can easily select which functions should be included in the application. The resulting applications are completely portable, in that they can be recompiled for different regions, different MCUs, and different radios. For an overview of Connect and its features, see *UG103.12: Application Development Fundamentals: Silicon Labs Connect*.

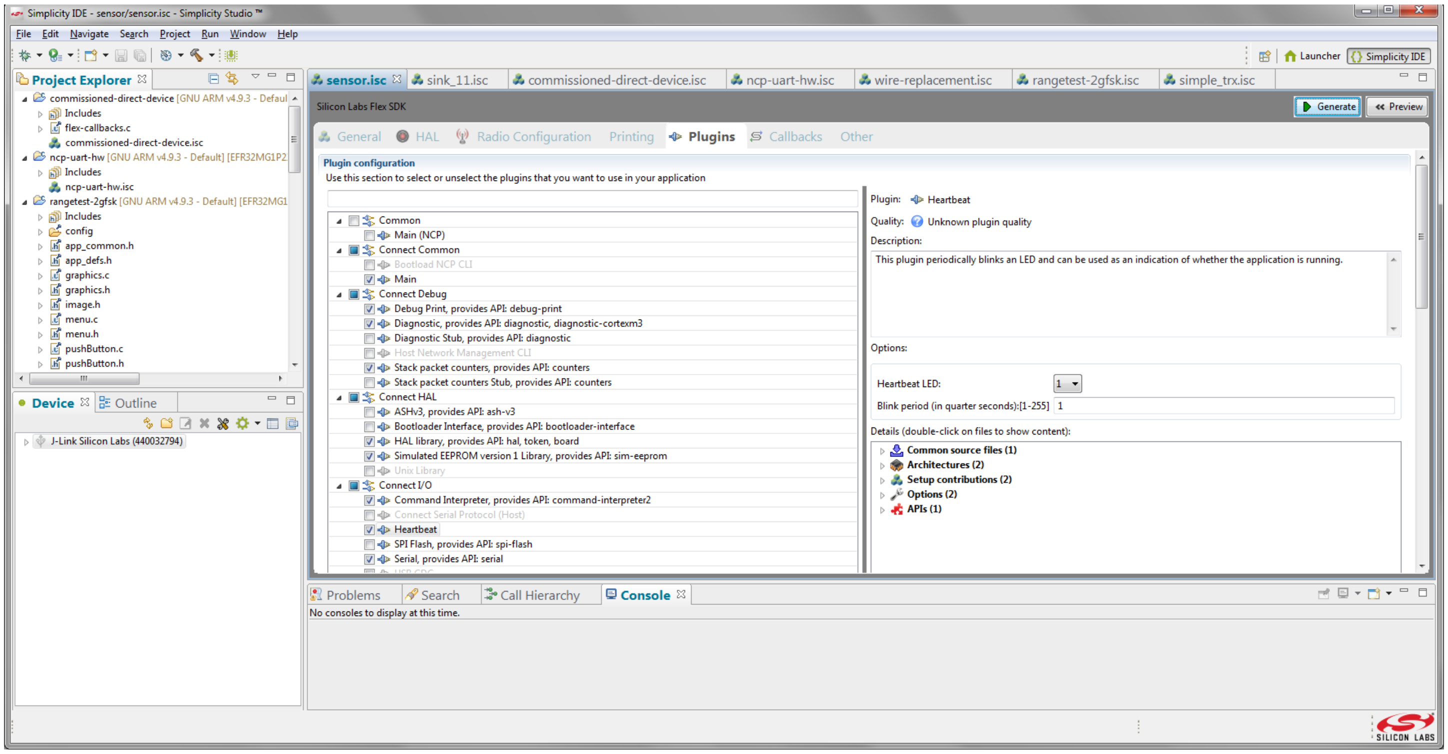
For more information about the 802.15.4 terms used in this document, see http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf.

This User Guide discusses the following topics:

- About Connect Plugins
- Using bootloaders
- Using the Serial Bootloader
- Using the Connect OTA Broadcast and Unicast Bootloaders
- Using Frequency Hopping
- About the Mailbox plugin
- Working with sleepy devices
- Working with direct devices
- Using a custom board header

2. Connect Plugins

The Connect stack code has been structured to support optional functionality blocks called plugins. Each plugin is provided as a stand-alone library or set of source code. The corresponding stub library is also provided, and is compiled in place of the full library if the plugin has not been selected during development. Underlying code checks at run time whether libraries are present or not. Common stack code performs some actions conditionally if a library is present or not. Plugins may depend on other plugins. For example, the network formation plugin depends on the scan plugin. Plugin dependencies are managed by Simplicity Studio's Application Builder. This section describes some of the plugins available. For a complete list see the Plugins tab of a current Connect release. Click on a plugin to see information about it.



commissioned-direct-device [GNU ARM v4.9.3 - Defaul
 Includes
 flex-callbacks.c
 commissioned-direct-device.isc
ncp-uart-hw [GNU ARM v4.9.3 - Default] [EFR32MG1P2
 Includes
 ncp-uart-hw.isc
rangetest-2gfsk [GNU ARM v4.9.3 - Default] [EFR32MG1
 Includes
 config
 app_common.h
 app_defs.h
 graphics.c
 graphics.h
 image.h
 menu.c
 menu.h
 pushButton.c
 pushButton.h

Plugin configuration

Use this section to select or unselect the plugins that you want to use in your application

- Common
  - [ ] Main (NCP)
- Connect Common
  - [ ] Bootload NCP CLI
  - [✓] Main
- Connect Debug
  - [✓] Debug Print, provides API: debug-print
  - [✓] Diagnostic, provides API: diagnostic, diagnostic-cortexm3
  - [ ] Diagnostic Stub, provides API: diagnostic
  - [ ] Host Network Management CLI
  - [✓] Stack packet counters, provides API: counters
  - [ ] Stack packet counters Stub, provides API: counters
- Connect HAL
  - [ ] ASHv3, provides API: ash-v3
  - [ ] Bootloader Interface, provides API: bootloader-interface
  - [✓] HAL library, provides API: hal, token, board
  - [✓] Simulated EEPROM version 1 Library, provides API: sim-eeprom
  - [ ] Unix Library
- Connect I/O
  - [✓] Command Interpreter, provides API: command-interpreter2
  - [ ] Connect Serial Protocol (Host)
  - [✓] Heartbeat
  - [ ] SPI Flash, provides API: spi-flash
  - [✓] Serial, provides API: serial

Plugin: Heartbeat

Quality: Unknown plugin quality

Description:

This plugin periodically blinks an LED and can be used as an indication of whether the application is running.

Options:

Heartbeat LED: 1

Blink period (in quarter seconds):[1-255] 1

Details (double-click on files to show content):

- Common source files (1)
- Architectures (2)
- Setup contributions (2)
- Options (2)
- APIs (1)

No consoles to display at this time.

Figure 2.1. The Plugins Tab of a Connect Example

2.1 MAC and Network Layer Optional Plugins

MAC and Network layer optional plugins, found in the Connect Stack group on the Plugins tab, include the following:

- Form and Join
- Frequency Hopping
- MAC packet queue
- Parent support
- Security plugins
  - XXTEA
  - AES

Form and join: Performs over-the-air network form and join operations. It also enables nodes to perform active scans, send 802.15.4 beacon request commands, and collect beacons. If this plugin is not selected, network parameters will be pre-commissioned by the application.

Frequency Hopping: Allows nodes to communicate while rapidly switching channels in a pseudo-random fashion, thereby reducing channel interference. It is currently not supported for Extended Star network topology.

MAC packet queue: Some applications need to submit multiple packets to the Connect stack. If this plugin is not selected, the stack can only handle one packet at a time. This also provides dynamic memory allocation functionality.

Parent support (requires MAC packet queue)**:** Provides parent functionalities such as indirect communication (communication with sleepy devices), child table and routing table, and should be included for any coordinator or range extender node intended to support multiple end-device and/or sleepy end-device nodes.

- **Child table** allows a star coordinator or a star range extender to support multiple children. Children are aged and eventually removed. The child information table is stored in NVM (non-volatile memory).
- **Indirect queue** buffers packets destined to sleepy children.
- **Routing table** is needed for star coordinator applications if the network includes star range extenders. It stores the information collected from star range extenders.

XXTEA-based security: Enables nodes to exchange secured messages using an XXTEA software-implemented encryption algorithm. This plugin enables security for 8-bit parts that have no hardware AES acceleration block. It should be included in applications for 32-bit parts that need to interoperate with 8-bit parts. The Connect over-the-air packet format makes it possible for nodes to support multiple security schemes and distinguish between them at run time.

AES-based security: Enables nodes to exchange secured messages based on the CCM* (Counter with CBC-MAC variant for 802.15.4) standard encryption/authentication scheme. This takes advantage of the AES hardware acceleration block available on 32-bit parts.

2.2 Application Framework Plugins

The Connect Application Framework plugins are used to manage application layer functionality, as follows:

- Connect Common group
  - Main
  - Main (NCP)
- Connect Debug group
  - Diagnostic
  - Debug Print
  - Stack packet counters
- Connect Utility group
  - Idle / Sleep
  - Polling
- Connect I/O group
  - Serial
  - Command Interpreter
  - Heartbeat
  - WSTK (Wireless Starter Kit) sensors

Connect Common - Main: Defines the `main()` function for SoC applications. It calls all the required initialization functions. It also implements the stack handlers and dispatches them to every plugin that subscribes to them by calling into the bookkeeping auto-generated callbacks. If the diagnostic plugin was selected, prints out reset information upon reset.

Connect Common - Main (NCP): Defines the `main()` function for NCP applications. It calls all the required initialization functions.

Connect Debug - Diagnostic: Provides program counter diagnostic functions and, if the software crashes and the system resets, prints out on the serial port reset information including the call stack.

Connect Debug - Debug print: Manages each plugin's `printf()` debug APIs. These APIs are auto-generated by the Application Framework. The application can also define its own debug printf types. This makes it possible to easily configure which plugins have debug print routines included or excluded and turned on or off at runtime. If this plugin is not selected, the API calls have no effect.

Connect Debug - Stack packet counters: Provides stack packet counters functionality. If this plugin is enabled, the stack shall keep track of successful and failed transmissions as well as successful received packets and dropped incoming packets.

Connect Utility - Idle/sleep: Manages idle and sleep mode.

- **Idle mode:** The main application loop is halted while the radio stays on. An incoming packet causes the node to get out of idle mode. Other interrupts are also served.
- **Sleep mode:** The MCU processing is halted and the radio is disabled.

The plugin includes the logic for determining if and when the device can idle or sleep and initiates idle/sleep whenever it is possible. It attempts to sleep first, but if that is not possible, then it attempts to idle. It queries the stack, Application Framework plugins, and the application.

Connect Utility - Polling: Manages periodic polling for end devices. Regular end devices need to exchange some sort of traffic with the parent as a "keep-alive" mechanism, also referred to as a "long poll interval." Star sleepy end devices need to poll the parent for incoming packets, also referred to as a "short poll interval." Star end devices are in long poll mode by default. The application can switch to short poll mode when appropriate using the plugin. For instance, a star sleepy end device sends out a packet that expects a response. The application then switches to short poll mode until the expected response is received.

Connect I/O - Serial: Provides high-level read/write serial communication functionality, including `readByte()`, `readData()`, `readLine()`, `writeByte()`, `writeHex()`, `writeString()`, `writeData()`, `writeBuffer()`, `printf()`, `guaranteedPrintf()`, `printfLine()`, `printCarriageReturn()`, and `printVarArg()`. Relies on the HAL low-level UART APIs.

Connect I/O - Command interpreter: Provides a common framework for defining CLI commands and for parsing serial input. Each CLI command is defined by the command string, the set of parameters, and the corresponding function to be called when a command and its parameters are successfully parsed. Application developers can easily define a custom set of CLI commands.

Connect I/O - Heartbeat: For use with the WSTK. Periodically toggles an LED on the WSTK board. The application can set which LED to toggle and the toggling period. Uses the HAL APIs to toggle board LEDs.

Connect I/O - WSTK sensors: The WSTK board is equipped with a temperature and humidity sensor. This plugin provides APIs to read humidity and temperature values. It initializes the sensor and reads values using the low level HAL APIs.

3. 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 can be used with EFR32MG1/ EFR32BG1 (EFR32xGI) and EFR32xG1+Flash, however, beginning with the EFR32MG12/ EFR32BG12/ EFR32FG12 (EFR32xG12) platform, it and all future Mighty Gecko, Flex Gecko, and Blue Gecko releases will use the Gecko Bootloader only. For more information about the Gecko Bootloader see *UG266: Silicon Labs Gecko Bootloader User's Guide*. For general background on bootloaders see *UG103.06: Application Development Fundamentals: Bootloading*.

Connect includes precompiled bootloader images, as well as the functionality for developers to build their own bootloader images.

Application images for use with the Gecko Bootloader have the extension .gbl and are referred to as GBL files. Application images for use with legacy bootloaders have the extension .ebl and are referred to as EBL files.

3.1 Using Precompiled Bootloaders

Precompiled Gecko Bootloader images for the most common of the EFR32MG12 platforms are provided under <Simplicity Studio Installation>\developer\sdks\gecko_sdk_suite\<version>\platform\bootloader\sample-apps.

Precompiled legacy serial-uart-bootloader.s37 and app-bootloader-spiflash.s37 bootloader images are available under <Simplicity Studio Installation>\developer\sdks\gecko sdk suite\<version>\protocol\Flex_<version>\tools\bootloaders\<board version>. Connect includes legacy precompiled Bootloader images for all supported boards.

3.2 Using Custom Bootloaders

For instructions on customizing and compiling Gecko Bootloaders for use with Connect, see *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*.

For legacy bootloaders, Connect provides the ability for developers to compile bootloaders for a custom board. Precompiled libraries (app_bootloader_library, standalone_bootloader_library) that are consumed by the Bootloader makefiles are provided in the stack.

Here is an example of recompiling a bootloader for a Silicon Labs board. From the Flex directory, execute the following:

```
make –f tools/bootloaders/serial_uart_bootloader.mak FAMILY=EFR32MG1P ARCHITECTURE=EFR32MG1P233F256GM48
BOARD=BRD4152A KIT=EFR32MG1_BRD4152A
```

Here is an example of compiling a bootloader for a custom board. From Flex directory, execute the following:

```
make –f tools/bootloaders/serial_uart_bootloader.mak FAMILY=EFR32MG1P ARCHITECTURE=EFR32MG1P233F256GM48
BOARD=custom_board BOARD_HEADER_PATH=../custom/board KIT_PATH=../custom/kit
```

The bootloader image will be compiled in the build directory.

Note: If KIT_PATH is given, KIT will not be used, as KIT refers to a Silicon Labs-provided kit.

4. Serial Bootloader

A serial bootloader is used to load a new image onto an MCU from a serial connected host process. There are four distinct parts to the serial bootloader system.

- **Host Application:** The host application runs on a different MCU from the network co-processor. This could be either an embedded application or a larger Linux type application. The host application generally presents the Command Line Interface to the user and is the starting point for invoking the serial bootload process.
- **Bootloader Application:** This is a separate process from the host process. It runs on the host processor and interacts with the embedded serial bootloader in order to load the new NCP firmware image into memory.
- **Serial Bootloader image:** The serial bootloader image takes up the first portion of flash memory on the NCP. It is used by the NCP to load the NCP firmware image into the right location in memory.
- **NCP firmware image:** This is the GBL/EBL NCP image that is loaded onto the MCU during the serial bootload process.

The serial bootloader may be used from the shipped Connect stack in the following steps:

Step 1:

In order to use the serial bootloader, the serial bootloader image must first be flashed onto the NCP. Connect provides both UART and SPI bootloader images for each of the supported boards.

As described in section 3.1 Using Precompiled Bootloaders, Gecko bootloader images are located in the platform installation under /bootloader/sample-apps/. The SPI bootloader is under bootloader-storage-spiflash, and the UART bootloader is under bootloader-uart-xmodem.

Legacy bootloader images are located in the protocol installation under tools/bootloader/<your-board>/serial-uart-bootloader/serial-uart-bootloader.s37 and tools/bootloader/<your-board>/app-bootloader-spiflash/app-bootloader-spiflash.s37. First find the serial bootloader image that you need for your EFR32 and flash it onto your NCP device using the flash loader or Simplicity Studio.

Step 2:

On the host, compile the bootloader application. The bootloader application is used by the host application to interact with the serial bootloader on the NCP. It is essentially a driver for the serial bootload process. In the Connect stack you can compile the bootloader application with the following command:

```
make -f connect/plugins/serial-bootloader/bootload-ncp-uart-app.mak
```

Note: This “make” call should be executed from the stack root directory.

Step 3:

Create, generate and compile your host application. The host application can be generated in Simplicity Studio and compiled on the command line, or by whatever method you are using to compile applications your host processor. In order to interact with your host application and use the serial bootloader, you need to include the Bootload NCP CLI (Command Line Interface) plugin. This provides the host application functionality and CLI for interacting with the bootloader application.

Step 4:

Convert your NCP image into the GBL/EBL format, if it is not in that format already. .

Step 5:

Run your host application normally. Use the CLI command `bootloader launch` to put your host application into bootload mode. To load the image, use the command:

```
bootloader load-image <location-of-bootloader-app> <new-image-path> <begin-offset> <length> <serial options>
```

The GBL/EBL image will be transferred over to the NCP. Restarting the NCP will launch with the new image.

5. Broadcast and Unicast Over-the-Air Bootloader

Connect includes the following plugins. Unless otherwise noted, these are in the Connect Utility group.

- **OTA Bootloader Server Plugins**: The server side of the over-the-air bootloader protocol. It includes all the functionality to distribute an image to one or multiple target devices (clients) and to instruct one or more target devices (clients) to perform an image bootload operation at a certain time in the future.
  - **OTA Bootloader Server Plugin**: The broadcast server version, meant for multiple clients.
  - **OTA Unicast Bootloader Server Plugin**: The unicast server version, for a single client.
- **OTA Bootloader Client Plugins**: The client side of the over-the-air bootloader protocol. It includes all the functionality to download an image from an OTA bootloader server and to be instructed to perform an image bootload at a certain time in the future.
  - **OTA Bootloader Client Plugin**: The broadcast client version that assumes that there are multiple clients receiving the same image simultaneously.
  - **OTA Unicast Bootloader Client Plugin**: The unicast client version that assumes that it is only one client getting the image from the server.
- **Bootloader Interface Plugin** (in the Connect HAL group): Provides a set of APIs for interacting with the Ember Application Bootloader.
- **SPI Flash Plugin** (in the Connect I/O group): Provides a set of APIs to perform read/write/erase operations on external flash parts. Supported parts are:
  - Macronix MX25R8035F (8 Mbit)
  - Macronix MX25R6435F (64 Mbit).
- **OTA Bootloader Test Plugins**: Provides test code to demonstrate how to perform flash read/write/erase operation locally and how to use the OTA Bootloader Server/Client plugins to perform an over-the-air bootloading operation. This plugin comes with a set of CLI commands as well.
  - **OTA Bootloader Test Common Plugin**: The common part of the bootloader tests, contains the CLI commands for the external Flash part operations as well as functions that work the same way for broadcast and unicast (e.g. setting the image tag on the client side etc.). This plugin can also provide a test image that is added to the image of the Sample Application during build. It is a standard wire-replacement demo program and servers can send this image to the target(s).
  - **OTA Bootloader Test Plugin**: The broadcast-specific bootloader test functions, like CLI commands for setting up the target devices, starting the distribution and requesting a bootload from the targets.
  - **OTA Unicast Bootloader Test Plugin**: The unicast-specific bootloader test functions, the same as above but different CLI command with some differences to those in the broadcast versions.

5.1 Building an Application with a Broadcast and/or Unicast Bootloader

This section describes how to use the stock example application **commissioned direct device** to demonstrate over-the-air bootloading functionality. The application described here can act both as OTA (broadcast or unicast) Bootloader Server and OTA (broadcast or unicast) Bootloader Client.

1. Open the stock commissioned direct device example application in Simplicity Studio.
2. In the HAL Configuration tab, click the Bootloader drop-down menu and select **Application**.
3. In the Plugins tab, make sure the following are checked (other plugins may be checked as well):
   - OTA Bootloader Server and/or OTA Unicast Bootloader Server (one of the two or even both is possible)
   - OTA Bootloader Client and/or OTA Unicast Bootloader Client (one of the two or even both is possible)
   - Bootloader Interface
   - SPI Flash
   - OTA Bootloader Test Common
   - OTA Bootloader Test (if broadcast plugins were selected)
   - OTA Unicast Bootloader Test (if unicast plugins were selected)
4. Generate and build your application.

Please note that having both a broadcast and a unicast bootloader and being a client and a server at the same time (in extreme cases having all plugins including the test plugins), the size of the included images other than the provided test image may present excessive code memory usage or may not even fit into the memory of the device.

At this point Simplicity Studio builds your application and calls a post-build step where it creates a binary by combining the application together with the provided application bootloader. The generated binary file (your_app_with_bl.bin) can be programmed to your nodes.

5.2 Testing OTA Bootloading

The steps (e.g. CLI calls) below apply to both the broadcast and unicast versions of the bootloader unless indicated otherwise.

1. Flash the **your_app_with_bl.bin** image to N nodes. Identify one of your nodes as SERVER and the other N-1 nodes as CLIENT_X.
2. Bring up each node via application specific commands. Make sure the nodes have all the same PAN ID and channel, but have different node IDs.
3. Erase the external flash storage on each node via following commands:

   ```
   bootloader init
   bootloader flash-erase
   ```
4. The OTA Bootloader Test Common plugin comes with a 58 kB wire-replacement test application, targeted for BRD4150B, stored in const space (flash). The goal is to write that image in external flash at the SERVER node. To do so, issue a `bootloader write-test-ebl` command. (Section 5.3 Using a Different Image describes how to replace this test image with an GBL/EBL of your choice.)
5. In order for the SERVER to distribute the wire-replacement test application to the client nodes, first set the image tag of interest at each client, so that the clients will accept the incoming image from the server. On CLIENT(S), issue `bootloader set-tag 0xAA`, where 0xAA is an arbitrary 1-byte tag (you can choose any value you like).
6. At the SERVER, set the target(s). Depending on which type(s) of bootloaders you have chosen, steps are listed below.
   - Broadcast servers: For each CLIENT_X issue at the server a `bootloader set-target X CLIENT_X_ID` command, where X is the client index (0, 1, etc.) and CLIENT_X_ID is the node ID of CLIENT_X. For instance, if you have three client nodes, with node IDs 0xC000, 0xC001 and 0xC002, issue at the server the following three commands:

     ```
     bootloader set-target 0 0xC000
     bootloader set-target 1 0xC001
     bootloader set-target 2 0xC002
     ```
   - Unicast servers: since a unicast server has only one target at a time, the server issues a `bootloader unicast-set-target CLIENT_X_ID` command, where CLIENT_X_ID is the node ID of CLIENT_X. For instance, if you have the client 0xC000, issue at the server the following three command:

     ```
     bootloader unicast-set-target 0xC000
     ```
7. Start the image distribution process at the server.
   - Broadcast server: Issue the command `bootloader distribute <size> 0xAA 3`, where `<size>` is the size in bytes of the wire-replacement test application, 0xAA is the tag set at each client in step 5, and 3 is the number of targets set in the target list. The server should now distribute the image to the three clients in broadcast fashion.
   - Unicast server: Issue the command `bootloader unicast-distribute <size> 0xAA`, where `<size>` is the size in bytes of the wire-replacement test application, and 0xAA is the tag set at each client in step 5. The server should now distribute the image to the client in unicast fashion.

   The test code implements the client and server callbacks. At the client, the provided implementation writes the incoming segments in external flash.
8. Once the distribution process completes, the clients have downloaded the wire-replacement test image and written it in external flash. The server can now instruct the target(s) to bootload the downloaded image at a certain time in the future.
   - Broadcast servers: Issue the command `bootloader request-bootload 10000 0xAA 3`, where 10000 is the delay in milliseconds, 0xAA is the image tag specified in step 5, and 3 is the number of targets in the target list. Each CLIENT will be instructed to perform an image bootload. It will also adjust the delay so that all nodes should (roughly) perform the bootload operation at the same time.
   - Unicast servers: issue the command `bootloader unicast-request-bootload 10000 0xAA`, where 10000 is the delay in milliseconds, and 0xAA is the image tag specified in step 5. The CLIENT will be instructed to perform an image bootload.

   You should see the CLIENT(S) rebooting and then running the “wire-replacement” application. Optionally, you can also have the SERVER bootload the wire-replacement test app by issuing the `bootloader flash-image` command.

5.3 Using a Different Image

To use an image other than the provided wire-replacement test GBL/EBL:

1. In Simplicity Studio, create the application that will become the image to distribute.
2. If your preferred application happens to use the OTA Bootloader Test plugin, uncheck the **Test EBL file inclusion** plugin option, so the size of your application is not increased unnecessarily by another image.
3. Generate and build your application. At this point Simplicity Studio builds your applications and calls a post-build step where it creates an GBL/EBL file out of the output binary (your_app.ebl). This becomes a part of the OTA Bootloader Test plugin in the next step.
4. To convert the .gbl/.ebl binary file produced in the previous step to a C header file containing a byte array that can be consumed as an image by your application, use the 'xxd' tool provided by any Linux (either Cygwin) distributions.

```
xxd –i your_app.ebl bl-test-application-ebl.h
```

Finally, make sure that the generated array can be built in your application.

1. Open the generated header file in a text editor.
2. Add the 'const' keyword before the first 'unsigned' keyword in the generated header file. For example:

```
const unsigned char __wire_replacement_ebl[] = {
```

3. If the application name is anything other than 'wire-replacement,' rename the array and the length word in the header file. If your application name is 'wire-replacement' you can skip these steps.
   - Rename your array to: `__wire_replacement_ebl`
   - Rename your length word (located at the very end of the generated header file) to: `__wire_replacement_ebl_len`
4. Copy the generated and edited header file to the folder of the ota-broadcast-bootloader-test plugin, where FLEX_SDK indicates the location of the Flex SDK on your local machine.

```
copy bl-test-application-ebl.h $FLEX_SDK\connect\plugins\ota-broadcast-bootloader-test\
```

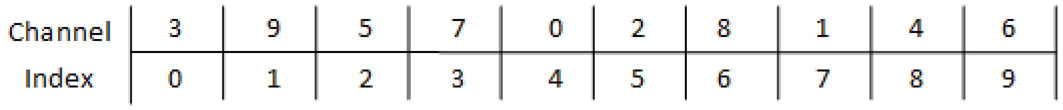
5. Any of your generated applications that use the OTA Bootloader Test plugin will bundle the new image as the test application file.

6. Frequency Hopping

Frequency hopping allows two nodes to communicate while rapidly switching channels in a pseudo-random fashion, thereby reducing channel interference.

Note: Frequency Hopping is only supported for direct device networks and simple star networks (star topologies with no range extenders). See *UG103.12: Silicon Labs Connect Fundamentals* for more information about network configurations.

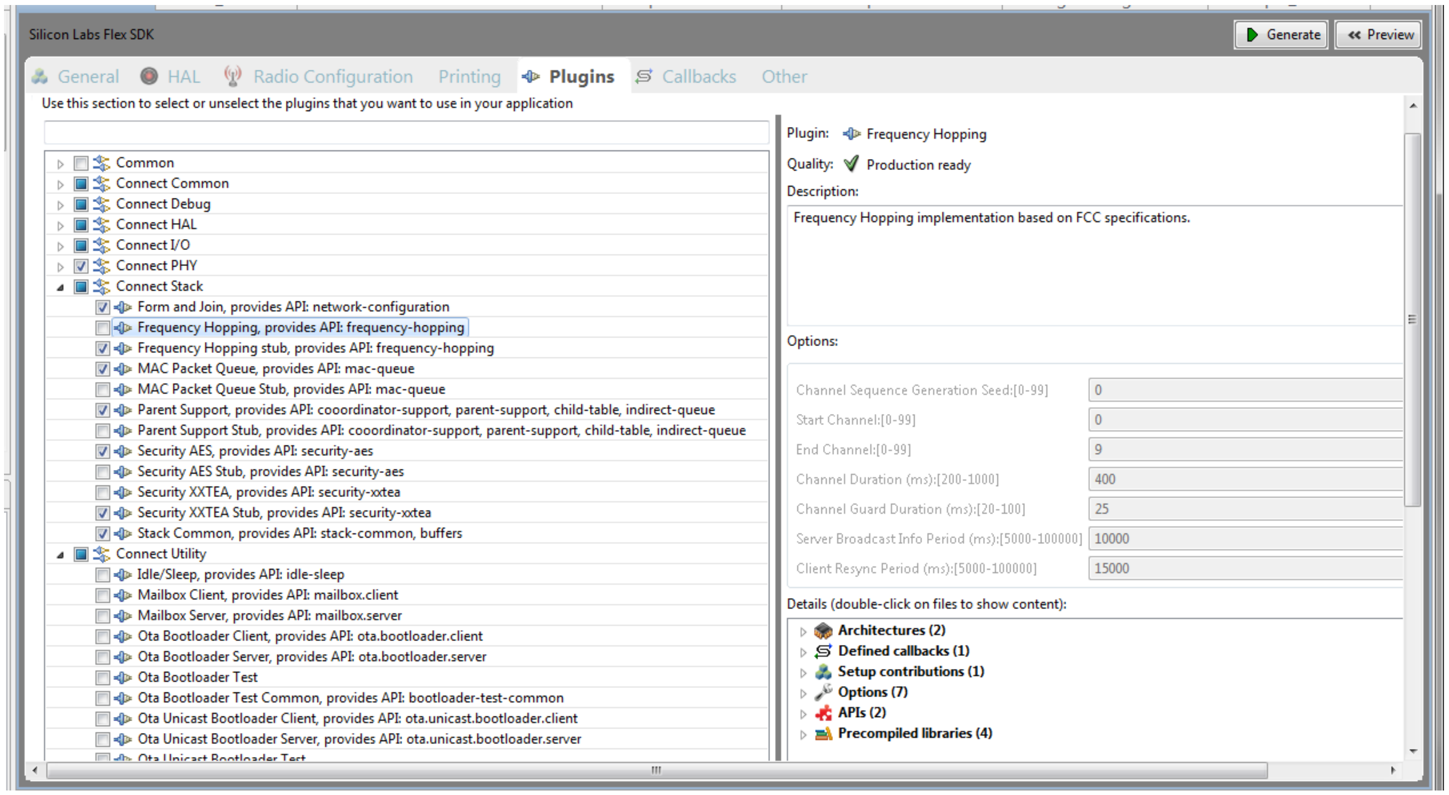
Frequency hopping is implemented in a client-server model, in which the server acts as the coordinator. The server and the clients are programmed with the same starting point and the same range. On both nodes, the pseudo-random sequence generator then assigns the same index number to each channel ‘slot’, as shown in the following figure.



Channel	3	9	5	7	0	2	8	1	4	6
Index	0	1	2	3	4	5	6	7	8	9

Figure 6.1. Pseudo Random Sequence Index

To enable frequency hopping, on the Plugins tab, Connect Stack group, check the Frequency Hopping plugin, shown in the following figure.



Use this section to select or unselect the plugins that you want to use in your application

- ☐ Common
- ■ Connect Common
- ■ Connect Debug
- ■ Connect HAL
- ■ Connect I/O
- ☑ Connect PHY
- ■ Connect Stack
  - ☑ Form and Join, provides API: network-configuration
  - ☐ Frequency Hopping, provides API: frequency-hopping
  - ☑ Frequency Hopping stub, provides API: frequency-hopping
  - ☑ MAC Packet Queue, provides API: mac-queue
  - ☐ MAC Packet Queue Stub, provides API: mac-queue
  - ☑ Parent Support, provides API: coordinator-support, parent-support, child-table, indirect-queue
  - ☐ Parent Support Stub, provides API: coordinator-support, parent-support, child-table, indirect-queue
  - ☑ Security AES, provides API: security-aes
  - ☐ Security AES Stub, provides API: security-aes
  - ☐ Security XXTEA, provides API: security-xxtea
  - ☑ Security XXTEA Stub, provides API: security-xxtea
  - ☑ Stack Common, provides API: stack-common, buffers
- ■ Connect Utility
  - ☐ Idle/Sleep, provides API: idle-sleep
  - ☐ Mailbox Client, provides API: mailbox.client
  - ☐ Mailbox Server, provides API: mailbox.server
  - ☐ Ota Bootloader Client, provides API: ota.bootloader.client
  - ☐ Ota Bootloader Server, provides API: ota.bootloader.server
  - ☐ Ota Bootloader Test
  - ☐ Ota Bootloader Test Common, provides API: bootloader-test-common
  - ☐ Ota Unicast Bootloader Client, provides API: ota.unicast.bootloader.client
  - ☐ Ota Unicast Bootloader Server, provides API: ota.unicast.bootloader.server
  - ☐ Ota Unicast Bootloader Test

Plugin: Frequency Hopping

Quality: ✓ Production ready

Description:

Frequency Hopping implementation based on FCC specifications.

Options:

Channel Sequence Generation Seed:[0-99]	0
Start Channel:[0-99]	0
End Channel:[0-99]	9
Channel Duration (ms):[200-1000]	400
Channel Guard Duration (ms):[20-100]	25
Server Broadcast Info Period (ms):[5000-100000]	10000
Client Resync Period (ms):[5000-100000]	15000

Details (double-click on files to show content):

- Architectures (2)
- Defined callbacks (1)
- Setup contributions (1)
- Options (7)
- APIs (2)
- Precompiled libraries (4)

Figure 6.2. Frequency Hopping Plugin

The parameters for the plugin are as follows:

- **Channel Sequence Generation Seed:** The seed number which generates the pseudorandom sequence.
- **Start channel:** The start or minimum of the pseudo random sequence channel range.
- **End channel:** The end or maximum of the pseudo random sequence channel range.

  In the figure above the pseudo random sequence is set to cover channels 0 through 9, inclusive.
- **Channel Duration:** How long in milliseconds the nodes will spend on each channel.
- **Channel Guard Duration:** How long in milliseconds the nodes will wait when entering and leaving a channel ‘slot’. The channel guard duration provides time in which transmission is not allowed. Packets transmitted during this time will wait until the guard duration is over.

Using the parameters in the previous figure, a node changes channels, waits 25 ms, transmits for 350 ms, waits 25 ms, then changes channels again.

- **Server Broadcast Info Period:** The time in milliseconds after which the server broadcasts the index number of the channel it is on and how long it has been on it. This gives clients that have gotten out of sync with the server the opportunity to resync,

  Using the parameters in the previous figure, the server frequency hops for 10000 ms, then transmits its information. If clients need to update their index and duration they do so now.
- **Client Resync Period:** The time in milliseconds without a resync, after which the client requests server information. This is particularly useful for sleepy devices, which may not be awake for a routine server broadcast.

In summary, the frequency hopping methodology is as follows:

1. The server forms the network and starts frequency hopping.
2. The client starts frequency hopping.
   a. It sends Frequency Hopping Info Request command on each channel in reverse pseudorandom order. This command requires a PAN ID and a Server Node ID for the network the client wishes to join, so that the client does not join a server from a different network.
   b. If an ACK is received on a channel, it waits for Frequency Hopping Info from the server.
   c. The server then sends the Frequency Hopping Info with information on which slot it is in and how many milliseconds it has been there.
   d. After some calculations to allow for any delays, the client syncs.
3. Once they are on the same channel, the client uses the join command to join the network.

The sequence is the same if the client is a commissioning device, except that in step 3 it commissions instead of joins.

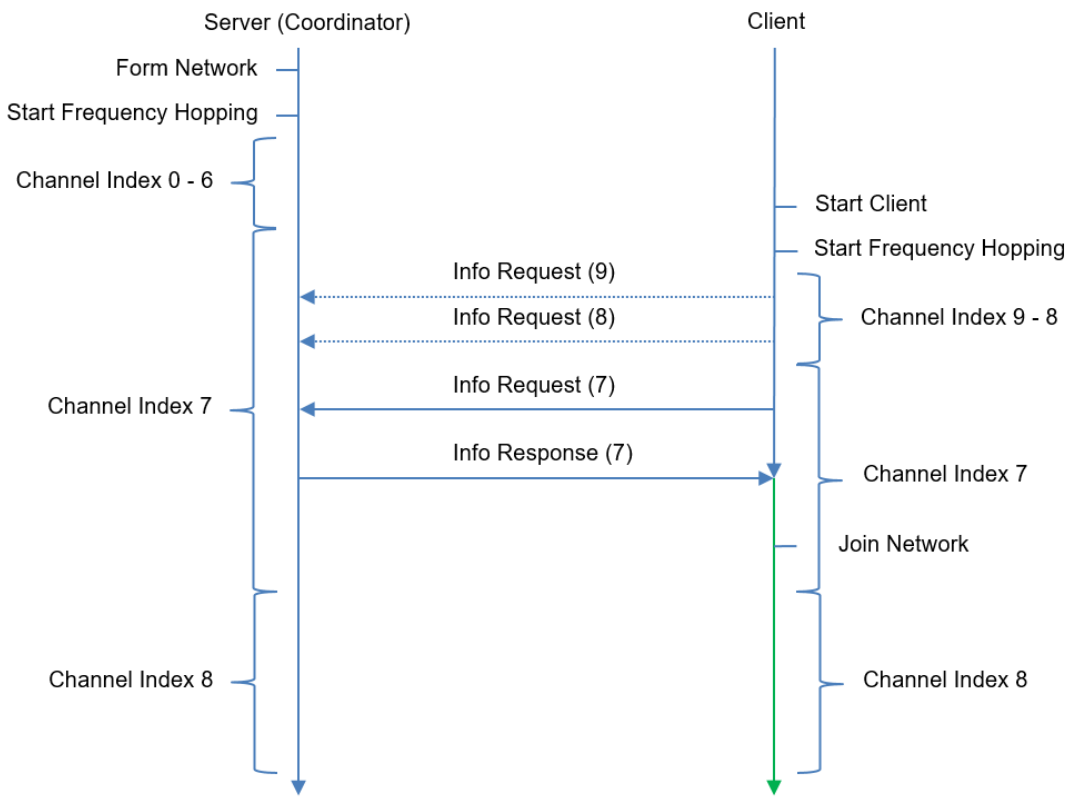


Figure 6.3. Network Form and Join with Frequency Hopping

The Frequency Hopping plugin includes three CLI commands that you can use to interact with an example application:

`start-fh-server` - Starts frequency hopping on the server.

`start-fh-client Node_ID PAN_ID` - Starts frequency hopping on the client, with the server Node ID and the network PAN ID.

`stop-fh` - Stops frequency hopping on the client or server.

7. Mailbox

The Mailbox plugin functionality consists of a Client and a Server. Simply put, the Server is responsible for storing and timing out messages that it is storing. The Client is responsible for submitting messages into the mailbox and retrieving messages intended for itself.

Mailbox Server

The mailbox server is able to receive messages from the client and hold them until the final destination retrieves them, or until some agreed upon timeout is reached. Technically, the final destination for a message can be any device. However, in a typical use case the final destination for a message is a sleepy device that is using the mailbox server as a holding point for messages that may arrive for it while it is asleep.

The agreed-upon mailbox timeout is configured in the plugin options.

Mailbox Client

The mailbox client has two functions. It can send messages to a mailbox server that are intended for another mailbox client. It can also retrieve messages intended for itself that are being stored on the mailbox server. When a message for another client is successfully retrieved by that client, the first client will receive a notification indicating that the message was successfully transmitted.

The process by which a client submits and another client retrieves a message is outlined in the following figure.

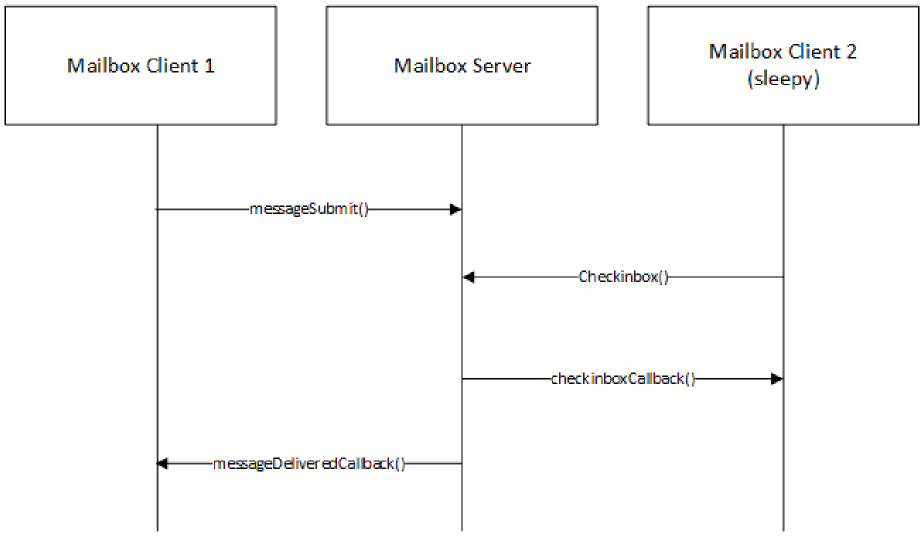


Figure 7.1. Mailbox Processes

8. Sleepy Devices

Sleepy devices can be implemented in two different ways.

- In a star network topology, a device may join the network as a sleepy end device.
- In direct device network topology, the application may determine when to put a device to sleep.

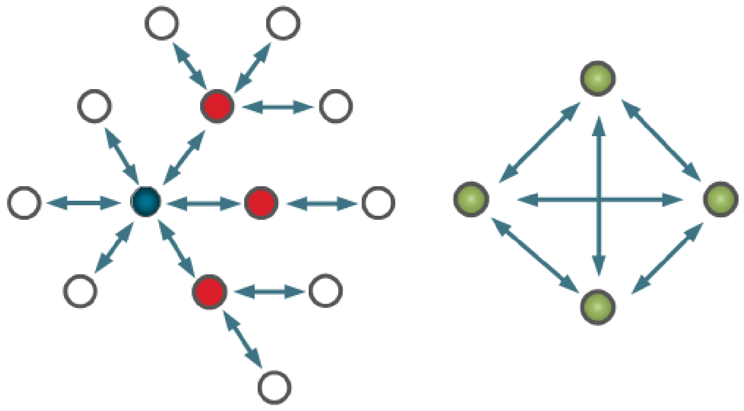


Figure 8.1. Star and Direct Device Network Topologies

In a star network topology, the device that wishes to join as a sleepy end device may simply join the network using the following API, where the nodeType is EMBER_STAR_SLEEPY_END_DEVICE:

```
EmberStatus emberJoinNetwork(EmberNodeType nodeType,
                             EmberNetworkParameters *parameters);
```

The sensor example uses this API if it gets the CLI command “join-sleepy”. This will cause the device to automatically join the network via an assigned parent, although the parent must enable joining of end devices beforehand. It will automatically poll its parent on a regular interval for messages that are stored there for it. In some cases, joining the star network as a sleepy end device obviates the need for the mailbox client and server mechanism. The parent of a sleepy device automatically stores messages for its child and provides them during the polling process. However, if the polling interval of the child is slower than the indirect message timeout of the parent, the mailbox client / server may still be useful.

The process of putting the device to sleep is handled by the idle-sleep plugin. The idle-sleep plugin checks with the stack on a regular interval to see if it is ok to put the device to sleep and for how long. It then idles the process, which is then awakened at the time of the next event. When the idle sleep plugin wakes, the plugin brings the stack back up and calls the `idleSleepWakeupCallback`.

The poll plugin is responsible for actually polling for data on a regular interval to keep the EMBER_STAR_SLEEPY_END_DEVICE in touch with its parent and check if the parent has any data for the child.

9. Direct Device

A direct device is a device type that is suited for flat network topologies. Each device can only communicate with other devices in a single hop radius, as shown in Figure 8.1 Star and Direct Device Network Topologies on page 14. As a result, no routing is performed.

In a direct device topology, the application is responsible for commissioning all network parameters. The application must make sure that each direct device is given its own unique short ID. Provided it is on the same PAN and channel as other direct devices, it will be able to communicate with them by addressing them using their unique short IDs.

The following brief narrative describes using the direct device sample application included in the Silicon Labs Connect stack. It offers some detail on what APIs are called from the associated CLI.

First, create a “commissioned-direct-device” sample application. Load it onto two devices using Simplicity Studio.

Next, using the “commission” CLI command, commission the first device (device_1) to have unique id 0x1111 and the second device (device_2) to have unique id 0x2222. Also, the devices should be on the same PAN (0xABCD) and on channel 15 at power 0.

Use the following command on each device:

```
device_1> commission 0x1111 0xABCD 15 0
device_1> Network up
device_2> commission 0x2222 0xABCD 15 0
device_2> Network up
```

This CLI command configures the stack for each device and starts the network. The commission CLI command simply calls the “ember-JoinCommissioned” API:

```
EmberStatus emberJoinCommissioned(EmberNodeType nodeType,
                                  EmberNodeId nodeId,
                                  EmberNetworkParameters *parameters);
```

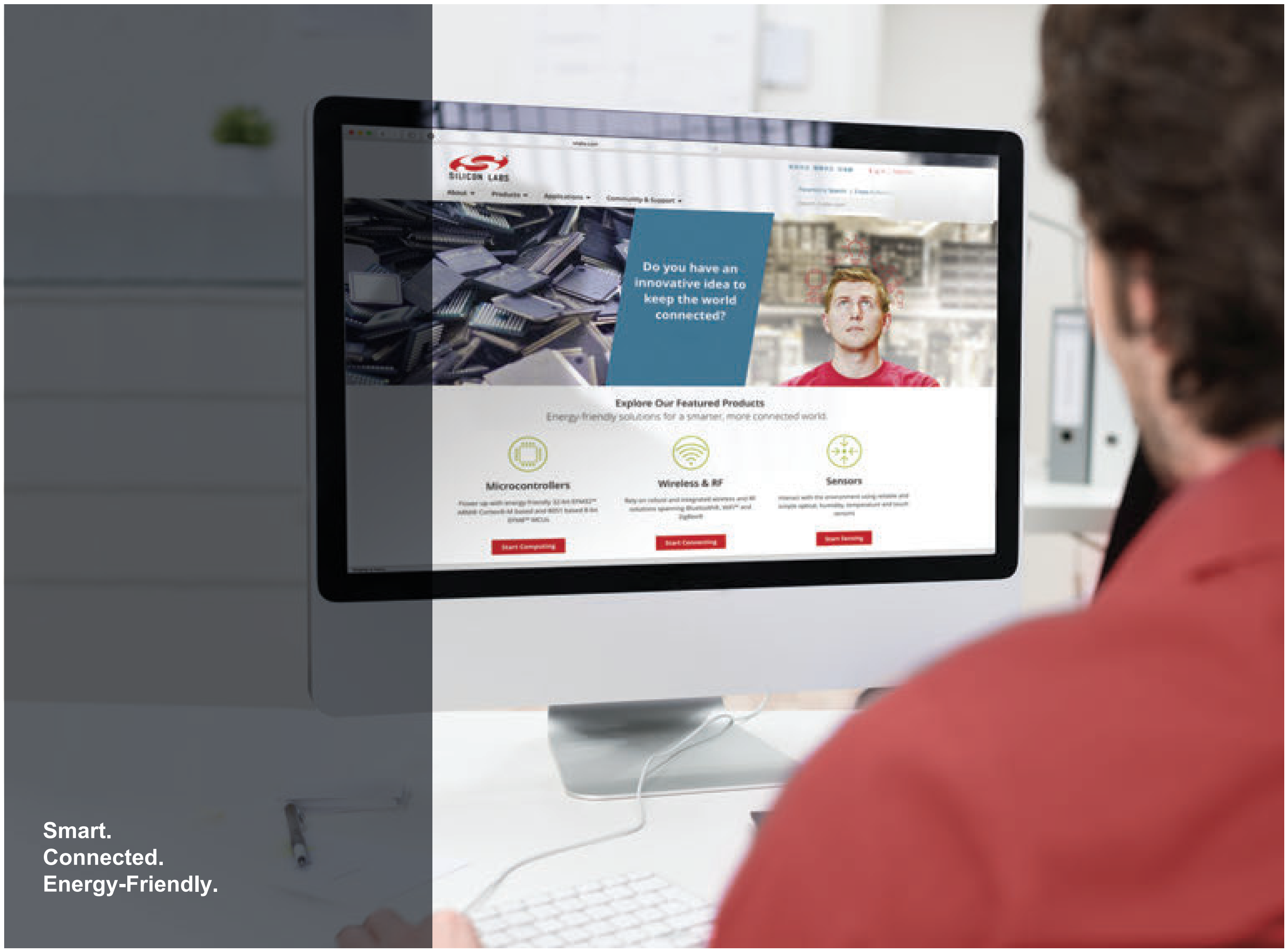
The parameters argument includes the PAN ID, Channel, and power settings for the device. Once the devices are commissioned they should return a status code indicating that the network has been formed. With the network in place you can send data between the two commissioned direct devices using the “data” command.

```
device_1> data 0x2222 “this is a message”
device_1> TX: Data to 0x2222: {74 68 69 73 20 69 73 20 61 20 6D 65 73 73 61 67 65}: status=0x00
device_2> RX: Data from 0x1111: {74 68 69 73 20 69 73 20 61 20 6D 65 73 73 61 67 65}
```

This is the simplest of network topologies. Each device must be able to hear and talk to all its neighbors. However, despite its simplicity, the direct device device type can be used very efficiently in conjunction with the sleep functionality and the mailbox plugin to handle a variety of complex use cases.

10. Using a Custom Board Header

When it is time to move an application over to a custom board, you can specify the custom header under the “Board Header Configuration” page of the HAL tab via “Custom created header” field.


Smart.
Connected.
Energy-Friendly.
Do you have an innovative idea to keep the world connected?
Explore Our Featured Products
Microcontrollers
Wireless & RF
Sensors

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