UG235.03: Architecture of the Silicon Labs Connect Stack

This chapter of the *Connect User's Guide* describes the architecture of the Silicon Labs Connect stack. The Connect stack is delivered as part of the Silicon Labs Flex SDK. The *Connect User's 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. Refer to *UG235.01: Developing Code with Silicon Labs Connect* for an overview of the chapters in the *Connect User's Guide*.

KEY POINTS

- Introduces Silicon Labs Connect.
- Lists the stack requirements.
- Describes the stack's modes.
- Describes physical layer limitations
- Describes the stack's MAC layer.
- Discusses the stack's network layer.
- Discusses the stack's configuration.

The *Connect User's Guide* is a series of documents that provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. If you are new to Connect and the Flex SDK, see *QSG138: Getting Started with the Silicon Labs Flex Software Development Kit for the Wireless Gecko (EFR32™) Portfolio*.

Connect is supported for EFR32FG, EFR32MG1x, and EFR32BG1x.

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 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 Medium Access Control (MAC) 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.

2 Stack Peripheral Requirements

The Connect stack itself requires some MCU peripherals to work:

- RF Transceiver
- HFXO—dependency of the RF transceiver
- TIMER0—used for high-precision (µs) timers.
- RTCC, compare channel 1 through `rtcdrv.c`—drives the low-precision (ms) timers for the event handler. It is also possible to use rtcdrv-based timers from the application.
- LFRCO or LFXO—dependency of the RTCC (most features were only tested with LFXO)
- Part of the internal flash memory is used for token (NVM) storage see *UG235.08: Using Other Services with Silicon Labs Connect*, *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 SoC Platforms* and *AN1154: Using Tokens for Non-Volatile Data Storage* for details. Also available to application through tokens.

The MCU hardware is accessed through emlib and RAIL, unless stated otherwise. This also introduces a dependence on some common emlib files (for example, `em_core.c` or `em_cmu.c`).

The stack also has this optional requirement:

- PTI—RF packet output for Network Analyzer (enabled by default)

Some plugins may require further peripherals. For more information, see *UG235.04: Using Plugins with Silicon Labs Connect*.

3 Connect Modes

Connect supports three distinct modes of operation. Only one mode is allowed in any single network and there is no simple way to upgrade from one mode to the other. As a result, you should select the mode carefully early in the design process.

3.1 Extended Star Mode

In this mode, Connect supports extended star topology networks.

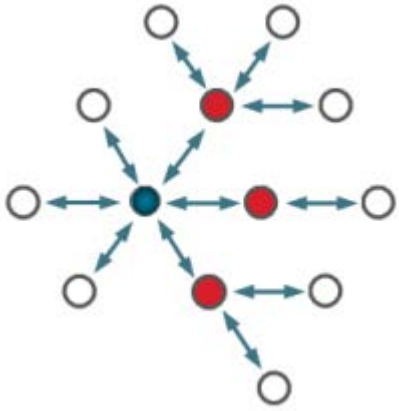


Figure 3-1. Extended Star Topology

Data message routing between any two devices is supported by the network layer in this mode. End devices can be configured to be sleepy, which means they do not keep their radio in receive when idle. The network layer also provides endpoints for messages which is set by the sender and seen by the receiver, and can be used similarly to a TCP/IP port.

Extended star topology is a centralized network. Joining to it must be accepted by the PAN coordinator and short address allocation can be handled by the PAN coordinator.

This mode is not fully IEEE 802.15.4 compliant.

3.2 Direct Mode

In this mode, Connect only provides connections between devices that are in range of each other.

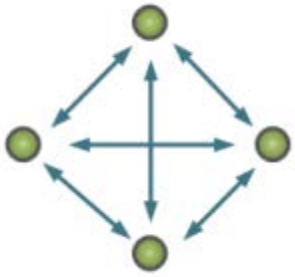


Figure 3-2. Direct Topology

The Connect network layer is still enabled in this mode, but it does not provide routing, only endpoints. However, routing protocols can be implemented in the application layer.

This is not a centralized topology. Any device can join the PAN by setting the right PAN parameters. Short address allocation is not provided by the stack and address duplication must be avoided by the application.

This mode is not fully IEEE 802.15.4 compliant.

3.3 MAC Mode

MAC mode is a fully IEEE 802.15.4 compliant setup of the Connect MAC layer. The Connect network layer is not enabled, which renders some plugins unusable in this mode (because some plugins require endpoints). The API is more complex compared to Direct mode and requires some knowledge of the IEEE 802.15.4 standard.

To make it fully IEEE 802.15.4 compliant, make sure to set up a 15.4-compliant radio configuration.

4 Connect Stack Layers

4.1 Physical Layer

Connect uses what was configured in the radio configurator. For details on how to use this tool, see *AN971: EFR32 Radio Configurator Guide*.

However, there are some restrictions when using the radio configurator used with Silicon Labs Connect:

- Connect will only use the first configured protocol
- You must use the Connect profile. This will remove some frame configuration options, since Silicon Labs Connect needs the frame's length configuration to be 802.15.4 compatible.
- Above 500kbps bitrate, the hardware-based address filter (which is required by Silicon Labs Connect) is not guaranteed to work. Using higher bitrates are only recommended after PER testing.

Silicon Labs Connect comes with a number of preconfigured PHY setups, designed to work within regional regulations.

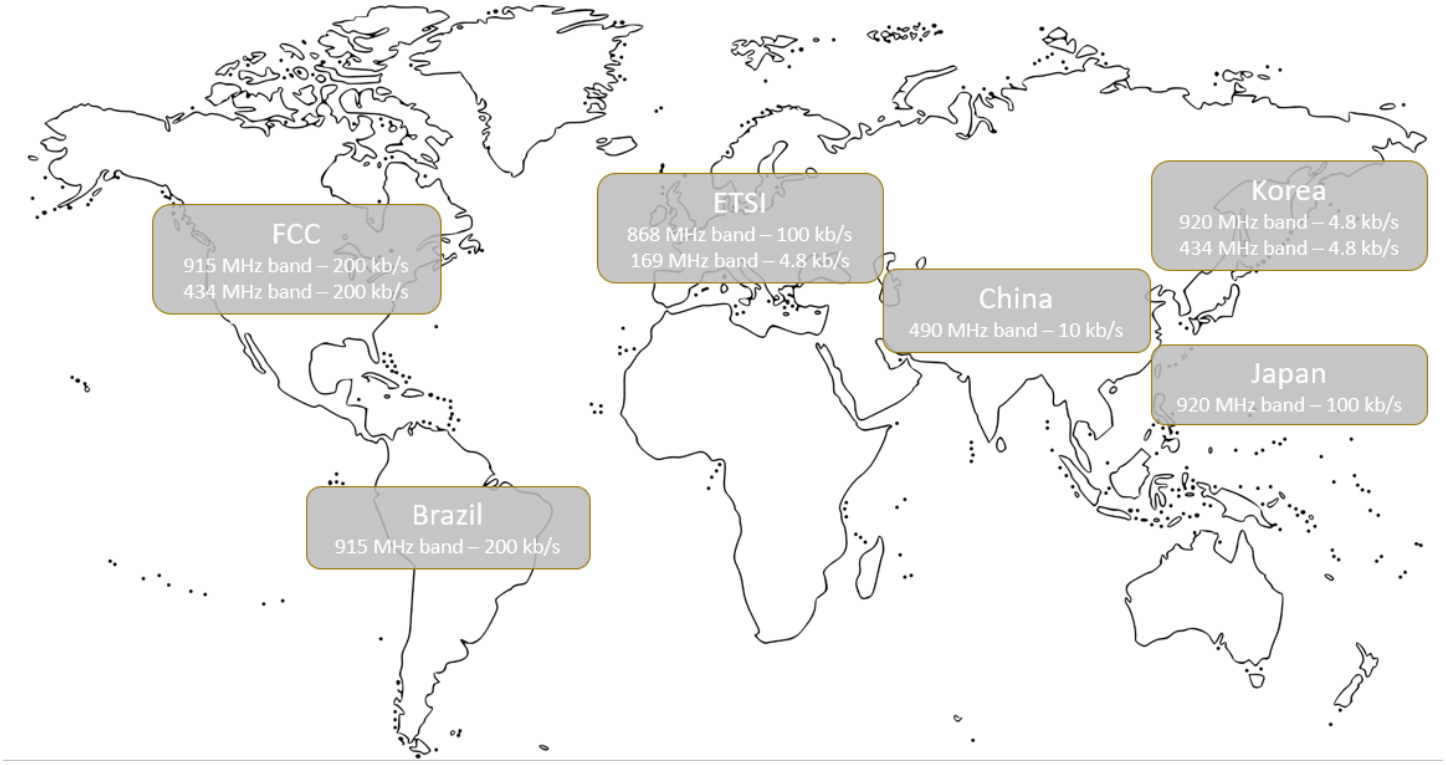


Figure 4-1. Silicon Labs Connect Configurations for Regions

4.1.1 FCC Regulations, DSSS, and Frequency Hopping

FCC requires steps be taken to "widen" the band that a given application uses. Silicon Labs Connect supports Direct Sequence Spread Spectrum (DSSS) and Frequency-hopping Spread Spectrum (FHSS) to comply with this requirement. Generally, Silicon Labs recommends the use of DSSS.

DSSS is supported by the hardware and requires no extra communication to work. The MAC layer can be fully 15.4-compatible, and 15.4 defines some standard DSSS PHY setups. On the other hand, frequency hopping requires non-15.4-compatible MAC commands and regular message exchange between devices to keep them synchronized.

Setting up a DSSS radio configuration is not simple, but Silicon Labs provides three preconfigured setups for 2.4GHz and 915MHz. Setting up the radio for frequency hopping is not difficult.

It is possible to set up frequency hopping with a narrower band (and lower bitrate) radio configuration which yields better sensitivity.

4.2 MAC Layer

The Silicon Labs Connect MAC layer is based on RAIL's IEEE 802.15.4-specific API, which provides the interface to the hardware implementation for

- Address filtering
- ACK
- CSMA/CA (Carrier-Sense Multiple Access with Collision Avoidance)
- Data Request/frame pending bit setup

Above the RAIL API, Silicon Labs Connect implements an 802.15.4-like MAC. This section describes the implemented features and highlights the deviations from the IEEE 802.15.4-2011 standard.

4.2.1 CSMA/CA

The CSMA/CA parameters can be configured using `EMBER_RADIO_CCA_THRESHOLD` (defaults to -65dBm) and `emberSetMacParams()`. The defaults are documented in the API documentation.

4.2.2 Device Types

Devices type is determined based on which API is used to join the network. Some device types are not available in every mode.

4.2.2.1 Coordinator (Extended star and MAC mode)

Coordinator forms the network with the API `emberFormNetwork()` or `emberMacFormNetwork()`. In Extended Star mode, the coordinator performs various tasks to maintain the network: It allocates addresses to joining devices, maintains routing tables, and so on. In MAC mode, it has no specific role, but it will report itself as PAN coordinator. The coordinator always has the short address 0x0000.

4.2.2.2 Range Extender (Extended Star only)

Range extender is a device that supports routing between the coordinator and end devices. It joins the network using `emberJoinNetwork()` with the nodeType set to `EMBER_STAR_RANGE_EXTENDER`.

4.2.2.3 End Devices and Sleepy End Devices (Extended Star and Direct Mode only)

End device is a simple device that can communicate in the network. A sleepy end device is the same, but it turns off its radio when in idle so it can only receive messages through message polling or some application layer implementation, such as the Mailbox plugin.

End devices join the network using `emberJoinNetwork()` or `emberJoinNetworkExtended()` in Extended Star mode and using `emberJoinCommissioned()` in Direct mode.

All of these have the nodeType parameter which selects the actual device type:

- `EMBER_STAR_END_DEVICE`
- `EMBER_STAR_SLEEPY_END_DEVICE`
- `EMBER_DIRECT_DEVICE`

4.2.2.4 MAC Mode Device and Sleepy MAC Mode Device (MAC Mode only)

A MAC mode device can have any role in an 802.15.4 network. A sleepy MAC mode device is the same, but it turns off its radio when in idle, so it can only receive messages through message polling or some application layer implementation. (This clears the "Receiver On When Idle" bit in the association request command.)

MAC mode devices can join the network with either `emberJoinNetwork()`, `emberJoinNetworkExtended()`, or `emberJoinCommissioned()`.

All of these have the nodeType parameter which selects the actual device type:

- `EMBER_MAC_MODE_DEVICE`
- `EMBER_MAC_MODE_SLEEPY_DEVICE`

4.2.3 MAC Layer in Extended Star and Direct Mode

Extended Star and Direct modes use the same MAC implementation, but with a few differences that will be highlighted in the following sections.

4.2.3.1 Messaging and Addressing

Apart from network maintenance, Extended Star mode only supports short-addressed, intra-PAN messages.

4.2.3.2 Short Address Allocation (Extended Star Mode only)

Short address allocation is always handled by the coordinator. The coordinator starts assigning addresses with 0x0001 and increments from there with each new access request. The last assigned address is stored in a token (nonvolatile memory), so the uniqueness of the assigned addresses is preserved even if the coordinator restarts. The coordinator cannot allocate an address which it had previously assigned to a device. Therefore, after it has allocated the (largest) address—0xFFFD—it will not accept any additional access requests from any device.

4.2.3.3 Security

In these modes, Silicon Labs Connect supports security very similar to 802.15.4's mode-5 security, but it is not fully-compliant. It uses different endianness and calculates the nonce slightly differently, but these do not impact the security itself.

The frame counter is stored in a token (which is a persistent element Silicon Labs Connect provides through one of multiple available non-volatile memory libraries). To further reduce wear on the flash memory, the frame counter is only saved after every 16384 (0x4000) increments. To ensure the frame counter is incremented even after the exact value in RAM is lost by a device reset or power cycle event, at boot Silicon Labs Connect will increment the base value stored in the persistent token (which will forfeit/skip any unused remaining increments of the original 16384 in the previous base value).

4.2.3.4 Non-standard Command Messages

Silicon Labs Connect uses some non-standard MAC commands:

- 0x7D: Extended association request. For details, see Section4.2.3.5.3 Device Joins with Extended Association Request.
- 0x7E: Frequency hopping info request. For details, see Section 4.4 Frequency Hopping (Extended Star and Direct Mode only).
- 0x7F: Frequency hopping info. For details, see Section 4.4 Frequency Hopping (Extended Star and Direct Mode only).

4.2.3.5 Association (Extended Star Mode only)

4.2.3.5.1 Device Joins Range Extender

In Extended Star mode, the coordinator allocates the short addresses. So if a device joins a range extender, the range extender must request a short address from the coordinator. For this, the range extender uses network layer command frames. (For details, see Section 5.3 Network Layer Commands.) This process is depicted in Figure 4-2.

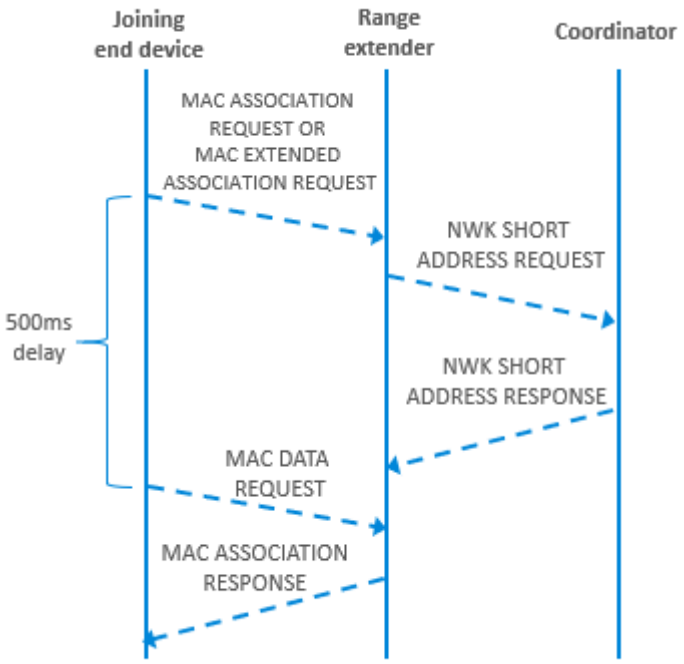


Figure 4-2. Join through Range Extender

4.2.3.5.2 Device Joins Coordinator

When a (sleepy) end device joins a coordinator, the association process is almost the same as the standard IEEE 802.15.4. (See *UG235.02: Using Silicon Labs Connect with IEEE 802.15.4* for a short description.) The only difference is the 500ms delay between association request and data request commands, as seen above.

4.2.3.5.3 Device Joins with Extended Association Request

Extended association request differs from the standard in one aspect: The joining device can select its own short address. In this case, the joining device is responsible for the uniqueness of the address—the coordinator will not check it. In every other detail, the join process is the same.

If the association was requested from a range extender, it still sends a short address request to the coordinator with the requested address. Although this is basically always approved by the coordinator, this allows for the ability to handle address allocation of extended association requests differently in the future.

4.2.3.5.4 Joining Device Receives Multiple Beacons

If a joining device receives multiple beacons during the join process, it will

- Join to the coordinator, if it received a beacon from a coordinator.
- Join to the first range extender to respond, if a range extender responded.

Note: This only applies to end devices. A range extender can only join to the coordinator.

4.2.4 Commissioning (Direct Mode only)

When commissioning, a device sets up all of its parameters (PAN ID, short address, and device type) without any communication to the network. This means that the commissioning device is responsible for ensuring the selected short address is unique.

4.3 MAC Layer in MAC Mode

4.3.1 Messaging and Addressing

In MAC mode, Silicon Labs Connect supports all addressing modes available in IEEE 802.15.4. The application can only send data frames. The stack still handles MAC commands, beacons, and ACKs.

4.3.2 Short Address Allocation

All non-sleepy MAC mode devices handle short address allocation. They assign random addresses and the network is responsible for detecting address duplication.

4.3.3 Association

Association in MAC mode implements the association described in IEEE 802.15.4. (See *UG235.02: Using IEEE 802.15.4 with Silicon Labs Connect* for a short description.) A MAC mode end device can join any other non-sleepy MAC mode device (coordinator or PAN coordinator).

4.3.3.1 Extended Association

The API `emberJoinNetworkExtended()` works differently compared to Extended Star or Direct mode. It still sends out the standard association request, but it can be only used to request the short address 0xFFFE, which means that the device wants to communicate to the network using its long address.

4.3.4 Commissioning

When commissioning, a device sets up all of its parameters (PAN ID, short address, and device type) without any communication to the network. This means that the commissioning device is responsible for selecting a unique short address.

4.4 Frequency Hopping (Extended Star and Direct Mode only)

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).

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 Figure 4-3.

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

Figure 4-3. Pseudo Random Sequence Index

To enable frequency hopping, on the Plugins tab, Connect Stack group, check the Frequency Hopping plugin, shown in Figure 4-4.

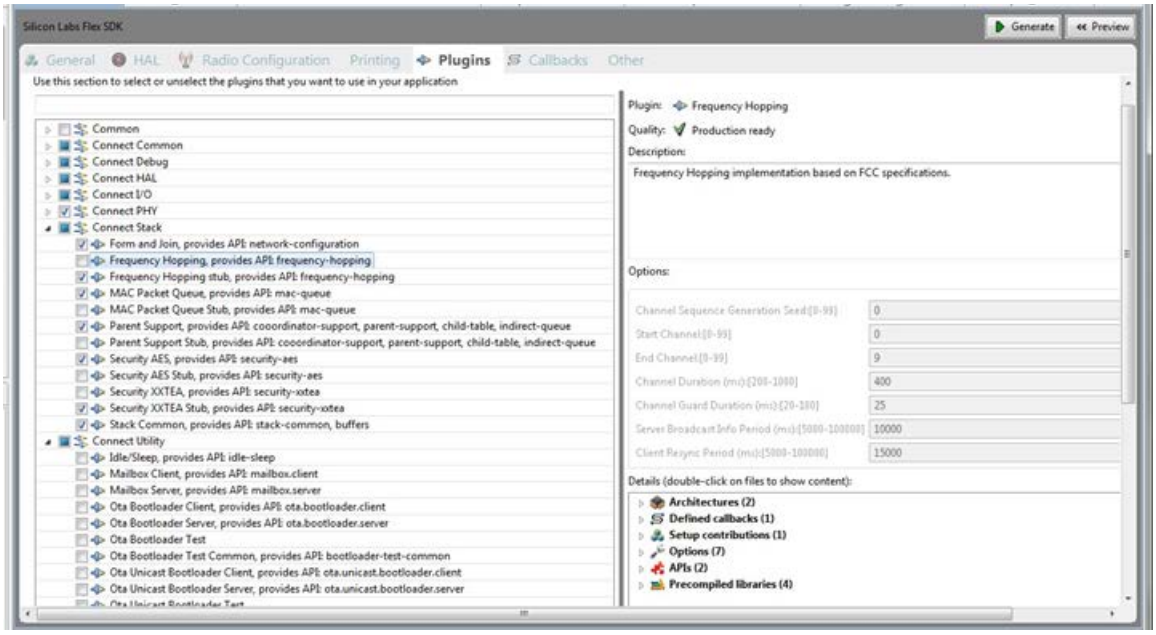


Figure 4-4. Frequency Hopping Plugin

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 Figure 4-4 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 Figure 4-4, a node changes channels, waits 25 ms, transmits for 350 ms, waits 25 ms, and 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 10000ms, 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.

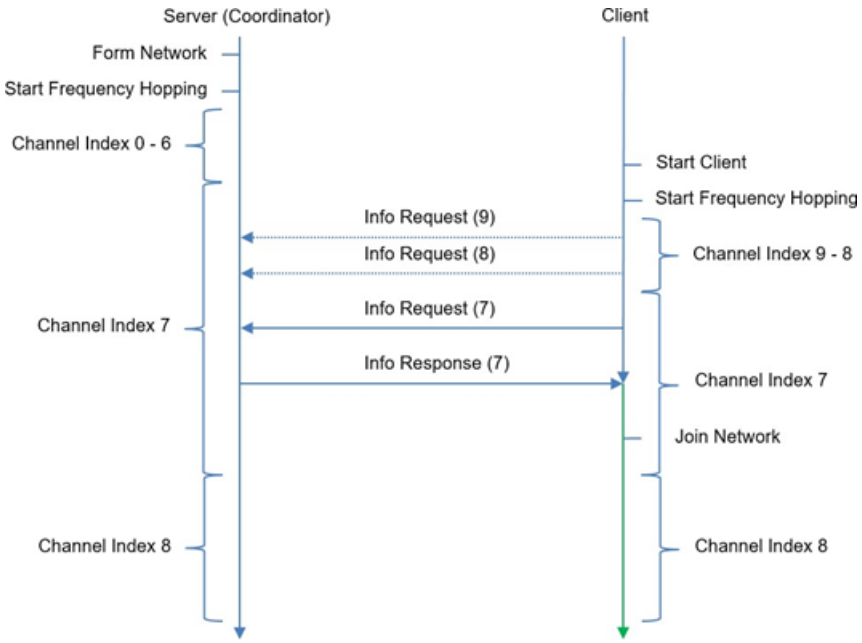


Figure 4-5. Network Form and Join with Frequency Hopping

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.

5 Network Layer

The Silicon Labs Connect network layer is only available in Extended Star and Direct mode. It is responsible for routing (Extended Star only) and endpoint handling. Endpoints implement channel sharing between protocols, similar to TCP/IP's port concept. For example, the Mailbox plugin uses endpoint 15, OTA uses endpoint 14, and other endpoints are sent to the application with the help of the application framework.

5.1 Network Layer Header

The Network layer header is illustrated in

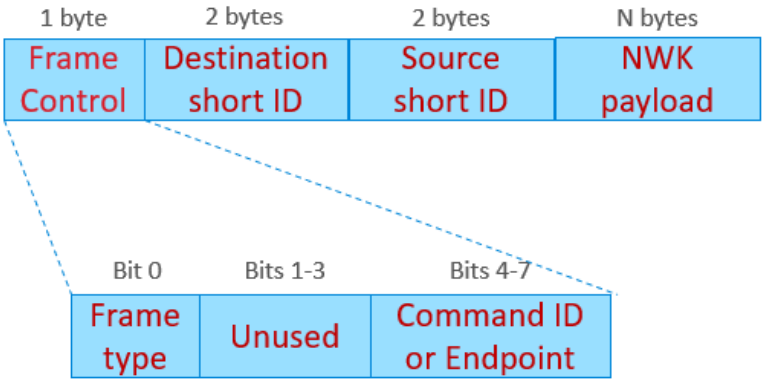


Figure 5-1. Network Layer Header

The Network layer header is included in all MAC data frames.

If the frame type bit is set, the frame is a network command message and handled by the network layer. If the frame type is cleared, the message is a data frame. A data frame is either routed towards its destination, or if it has been received by the destination, it is forwarded to the application layer with the `emberIncomingMessageHandler()` callback (which includes the NWK payload, the source short ID, and the endpoint as arguments).

5.2 Routing (Extended Star only)

5.2.1 Routing Tables

The routing protocol is centralized and the coordinator knows a route to all devices in the network. This is achieved with the following routing data stored on the devices:

- End device stores the address of its parent.
- Range extender stores the address of all children connected to it (child table).
- Coordinator stores the address of all children connected to it (child table) and keeps a table for each range extender with their children's addresses (coordinator routing tables, RAM only).

5.2.1.1 Aging

The child tables also store the timestamp of the last received message and entries are removed if a child does not communicate for "Child timeout" amount of time (configurable in the parent support plugin, 1 hour by default).

5.2.1.2 Child Table Limitations

The child table has the following limitations:

- The maximum number of children for coordinators is 64 (limited by the maximum child table size).

- The maximum number of children for range extender is 32 (limited by the format of the range extender update command).

5.2.2 Forwarding Rules

The routing protocol can be expressed in these forwarding rules:

- End Device: always forward to the parent (coordinator or range extender)
- Range Extender: if the final destination is in the child table, forward the packet to the final destination. Otherwise, forward to the coordinator.
- Coordinator: if the final destination is in the child table, forward the packet to the final destination. Otherwise, look up the final destination in the routing table and forward it to the corresponding range extender.

5.3 Network Layer Commands

The network layer supports the following commands:

- 0x01: Short address request
- 0x02: Short address response
- 0x03: Range extender update request
- 0x04: Range extender update

Short address request and Short address response are used to get a short address from the coordinator if a device joins a range extender. (For details, see Section 4.2.3.5.1 Device Joins Range Extender.)

Range extender updates are used to maintain the routing tables on the coordinator. Their payload is an array of short addresses. Each range extender periodically sends this command to the coordinator, every 60 seconds by default or can be configured with the compile time define `EMBER_NWK_RANGE_EXTENDER_UPDATE_PERIOD_SEC`. This command is also the response to range extender update requests.

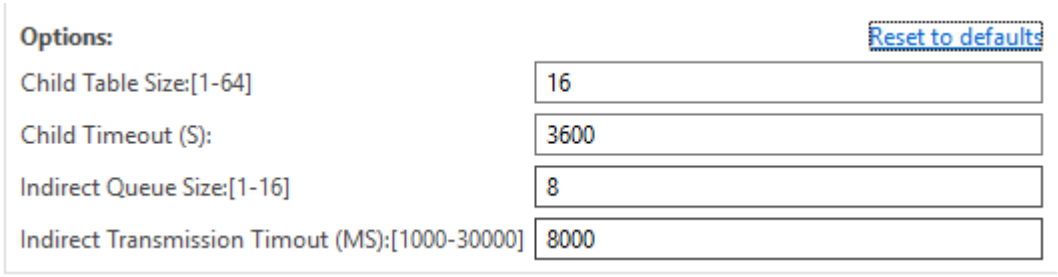
Range extender update requests are used if the coordinator needs to update its routing tables. Currently this is only used when a coordinator reboots.

6 Stack Configuration

The Connect stack loads various configuration values at bootup. These values can be configured in three different places.

6.1 Plugin Options in the Connect Stack Group

Many plugins in the Connect stack group have configuration options like this for the Parent Support plugin:

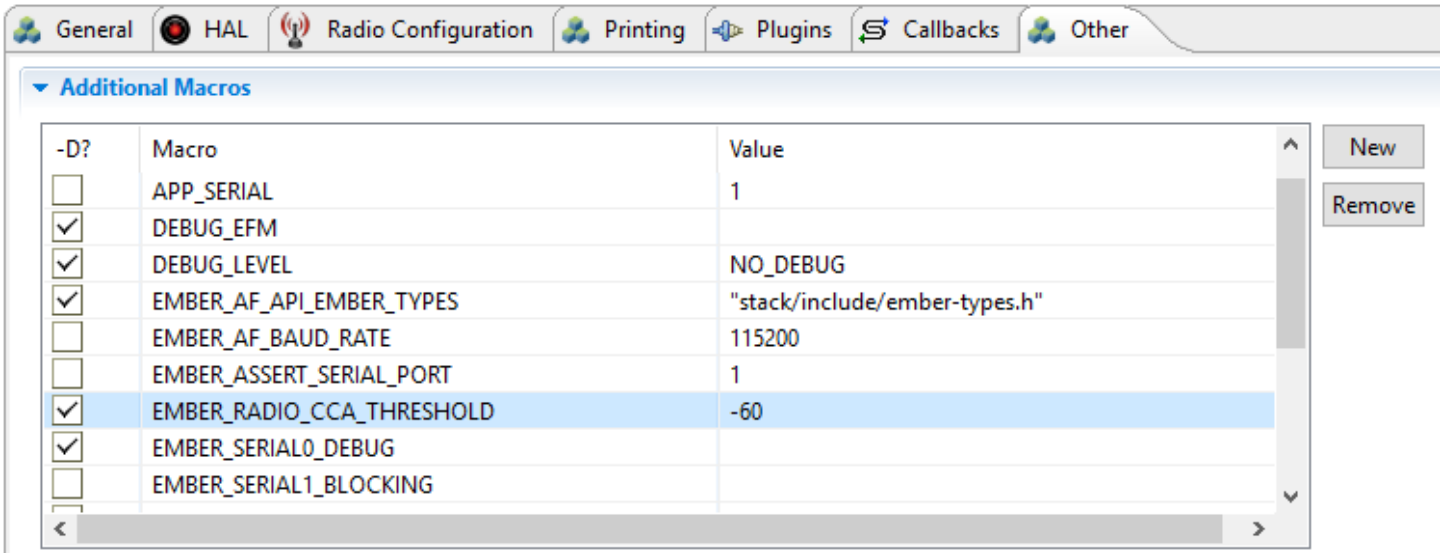


The configuration options are generated into the `flex-configuration.h` file in the project root as macros.

6.2 Compile-time Macros

The Connect stack uses several macros that are not configurable from the application builder. These are documented in the Connect API documentation. If a configuration macro is not defined at compilation, a default value will be used from `protocol/flex/connect/plugins/stack/config/ember-configuration-defaults.h`.

Compile-time macros can be added on the Application Builder's Other tab, using the Additional Macros pane with the -D option:



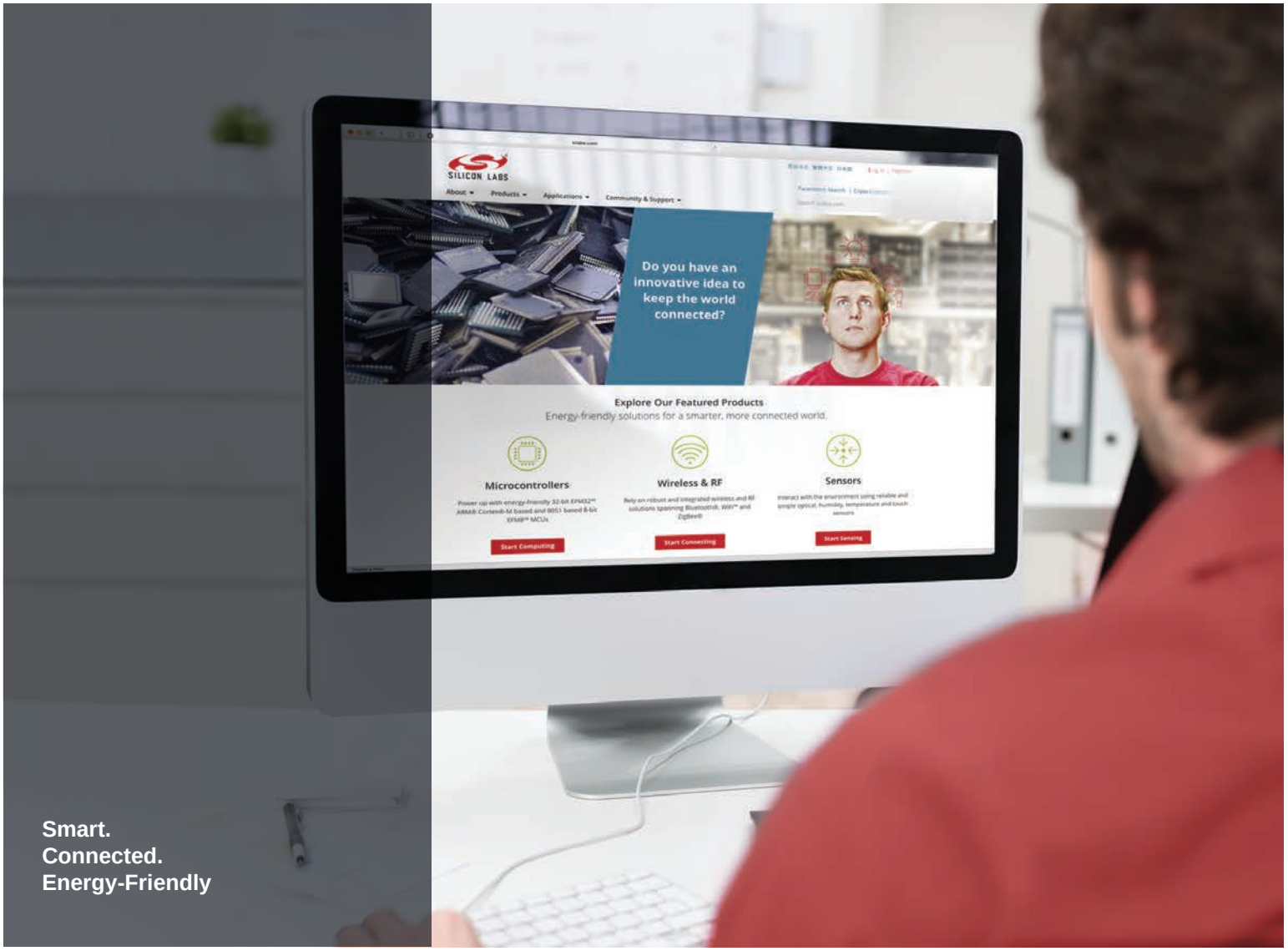
6.3 Manufacturing Tokens

The Connect stack also checks several manufacturing tokens to configure itself. These are tokens that are usually written once in the lifetime of the application, but can be written independently from the firmware. For details, see *AN961: Bringing Up Custom Devices for the Mighty Gecko and Flex Gecko Families* and *AN1154: Using Tokens for Non-Volatile Data Storage.*

The following manufacturing tokens are supported:

- TOKEN_MFG_CUSTOM_EUI_64
- TOKEN_MFG_CTUNE
- TOKEN_MFG_SECURE_BOOTLOADER_KEY (for Gecko bootloader)
- TOKEN_MFG_SIGNED_BOOTLOADER_KEY_X (for Gecko bootloader)
- TOKEN_MFG_SIGNED_BOOTLOADER_KEY_Y (for Gecko bootloader)

For more information on each manufacturing token, see *AN961: Bringing Up Custom Devices for the Mighty Gecko and Flex Gecko Families.*



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