UG435.03: Architecture of the Silicon Labs Connect Stack v3.x

This chapter of the *Connect v3.x User's Guide* describes the architecture of the Silicon Labs Connect stack v3.x. The Connect stack is delivered as part of the Silicon Labs Proprietary Flex SDK v3.0 and higher. The *Connect v3.x 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 *UG435.01: About the Connect v3.x User's Guide* for an overview of the chapters in the *Connect v3 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 v3.x 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 Proprietary Flex SDK, see *QSG168: Proprietary Flex v3.x Quick Start Guide*.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

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
- sleeptimer component:

  Can be configured to work with various timer sources, typically RTCC.

  All configurable timer sources depend on an LF oscillator, typically LFXO.
- Part of the internal flash memory is used for token (NVM) storage. For more information, see *AN1154: Using Tokens for Non-Volatile Data Storage* for details. Also available to application through tokens.
- While not actually used by Connect, the configuration of component dependencies pulls in a microsecond timer which uses TIMER0 by default.

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 Connect stack also has this optional requirement:

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

Some components may require further peripherals. For more information, see *UG435.04: Customizing Applications with Silicon Labs Connect v3.x*.

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 as illustrated in the following figure.

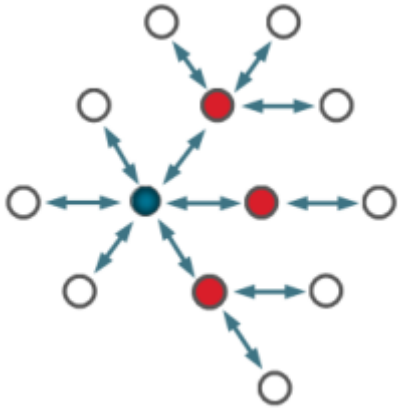


Figure 3.1. Extended Star Topology

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 are 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 as illustrated in the following figure.

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 components unusable in this mode (because some components 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 more information on how to use this tool, see *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5*.

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

- Connect will only use the first configured protocol.
- You should use the Connect profile. This will remove some frame configuration options, because 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 Connect) is not guaranteed to work. Using higher bitrates are only recommended after PER testing.

Connect comes with several preconfigured PHY setups, designed to work within regional regulations as illustrated in the following figure.

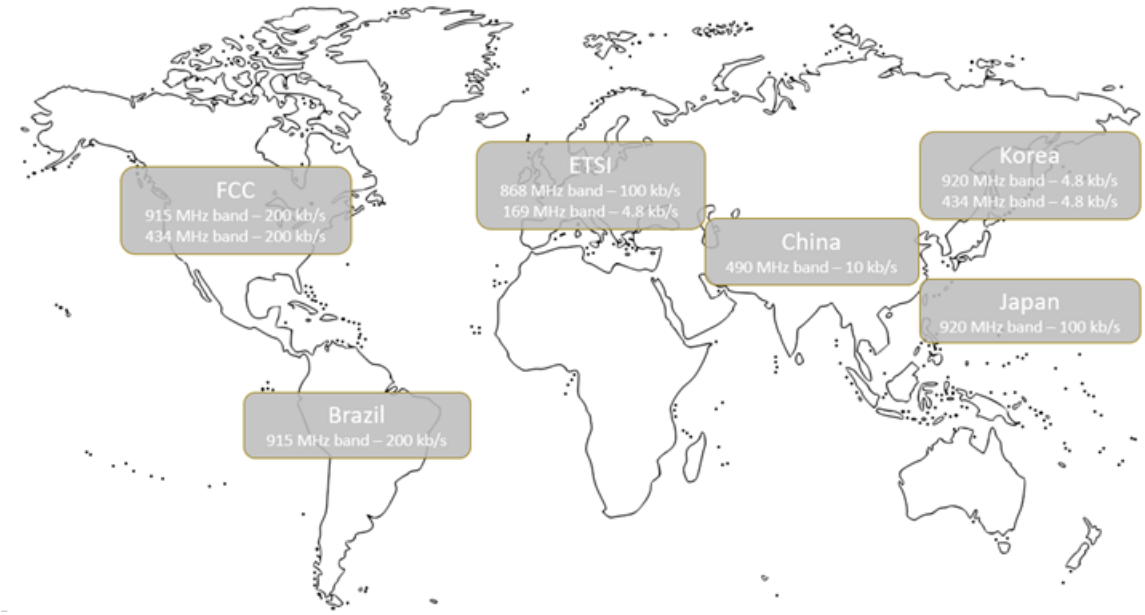


Figure 4.1. Silicon Labs Connect Configurations for Regions

4.1.1 FCC Regulations, DSSS, and Frequency Hopping

FCC requires steps be taken to "widen" the band that a given application occupies. 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 requires some RF knowledge, but Silicon Labs provides three preconfigured setups for 2.4GHz and 915MHz. Frequency hopping does not require a special radio configuration, only a well configured channel spacing and number of channels.

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

4.2 MAC Layer

The 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, 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 Acknowledgments

Acknowledgments are transmitted according to the IEEE 802.15.4 specification. However, acknowledgement timing and retransmit count can deviate from the standard in these ways:

- The turnaround time—that is, the delay between the end of reception and the start of transmission of the acknowledgement—is always **12 symbol time**.
- The timeout until a device waits for an ACK packet can be configured with the definition of `EMBER_MAC_ACK_TIMEOUT_MS`, or with the `ackTimeout` argument of `emberSetMacParams()`. The default is 25ms. Silicon Labs recommends fine tuning this parameter after choosing a PHY.
- The number of transmission retries from the sender if it does not receive an ACK can be configured with the `maxRetries` argument of `emberMacSetParams()`. The default is 3 defined as a maximum of 4 transmits: 3 retries after the first transmit.

4.2.2 CSMA/CA

The CSMA/CA parameters can be configured using `emberSetMacParams()`. The defaults are documented in the API documentation and are essentially the parameters recommended by IEEE 802.15.4 for the 2.4GHz OQPSK PHY. You can also configure the threshold under the *Stack Common* component configuration.

4.2.3 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.

4.2.3.1 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. A coordinator requires the *Parent support* component.

4.2.3.2 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`. A range extender requires the *Parent support* component.

4.2.3.3 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.

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`

4.2.3.4 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. (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`

4.2.4 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.

4.2.4.1 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.

4.2.4.2 Short Address Allocation (Extended Star Mode only)

Short address allocation is always handled by the coordinator. By default, the coordinator starts assigning addresses with 0x0001. This can be modified in the *Parent support* component, to reserve part of the address space for manual commissioning.

The allocated address increments from the first assigned address 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.

4.2.4.3 Security

In these modes, 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 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 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).

4.2.4.4 Non-standard Command Messages

Connect uses some non-standard MAC commands:

- 0x7D: Extended association request. Uses secure communication if the requester supports it. For more information, see section 4.2.4.5.3 Device Joins with Address Selected on the Joining Device.
- 0x7E: Frequency hopping info request. For For more information, see section 4.4 Frequency Hopping (Extended Star and Direct Mode only).
- 0x7F: Frequency hopping info. For For more information, see section 4.4 Frequency Hopping (Extended Star and Direct Mode only).

4.2.4.5 Association (Extended Star Mode only)

4.2.4.5.1 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 more information, see section 5.4 Network Layer Commands.) This process is depicted in the following figure.

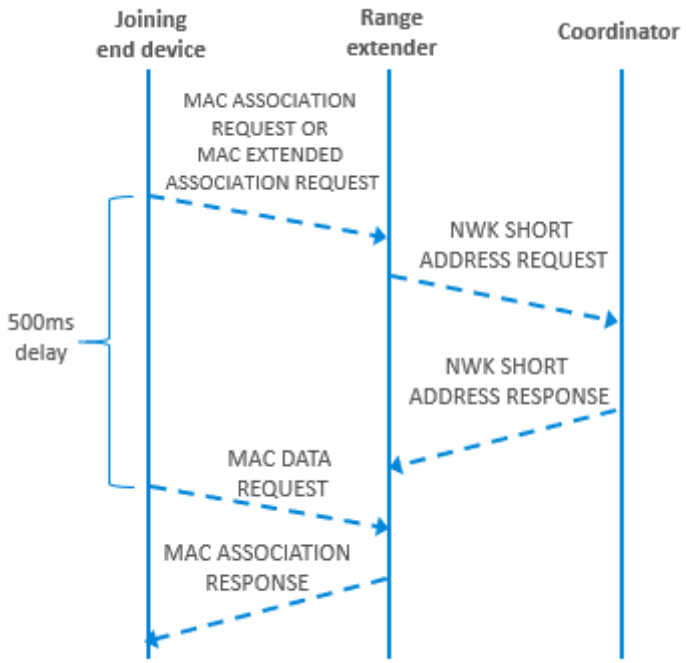


Figure 4.2. Join through Range Extender

4.2.4.5.2 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 *UG435.02: Using Silicon Labs Connect v3.x with IEEE 802.15.4* for a short description.) There are, however, are some differences in timing as follows:

- 500ms delay between association request and data request commands is added, as seen in the above figure.
- The active scan duration—the amount of time a joining device is waiting for beacons—is configured according to IEEE 802.15.4 and calculated from symbol time. However, on low-symbol rate PHYs this can slow down the join process significantly. You can configure it with `emberSetActiveScanDuration()`.

4.2.4.5.3 Device Joins with Address Selected on the Joining Device

The joining device can select its own short address. Because the standard association request does not have this requested address field, Connect uses extended association request for this feature. 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 with address request differently in the future.

4.2.4.5.4 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.

4.2.4.5.5 Selective Joining

In some cases it is required to select the coordinator/range extender on the joining device. This is enabled in Connect with the selective join payload, configurable with the `emberSetSelectiveJoinPayload()` and `emberClearSelectiveJoinPayload()` APIs. When enabled, the association request of the joining device will be only accepted on coordinators and range extenders that have the same join payload configured as the joining device. Because the standard association request does not allow this field, Connect will use extended association requests, even when the address was not requested.

4.2.5 Commissioning (Direct Mode only)

When commissioning, a device sets up all 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 Security

MAC mode supports the same mode-5 security as other modes of the Connect stack. Security requires the long address of the source to decode a message, so the application should provide address translation when short addressed frames are received. This is handled through a lookup table that can be populated and updated through the API `emberMacAddShortToLongAddressMapping()`.

4.3.4 Association

Association in MAC mode implements the association described in IEEE 802.15.4. (For a short description, see *UG435.02: Using Silicon Labs Connect v3.x with IEEE 802.15.4*.) A MAC mode end device can join any other non-sleepy MAC mode device (coordinator or PAN coordinator).

4.3.4.1 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.

4.3.5 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. Frequency hopping is implemented in a client-server model, in which the server coordinates the hopping. The server and the clients must be in the same PAN and must be programmed with the same frequency hopping parameters. These parameters are highlighted with **bold text** in the following sections.

4.4.1 Frequency Hopping Messages

Frequency hopping is implemented with the non-standard *Frequency hopping info* and *Frequency hopping info request* MAC command messages. A client receiving a *Frequency hopping info* message will result in synchronization between the server and the client, while *Frequency hopping info request* requests the info message from the server. After a sleepy end device sends a request, it will wait for an answer, similarly to data polls. These messages are sent without CSMA/CA.

4.4.2 Hopping Sequence

The hopping sequence is generated when the hopping system is started by the application using `emberFrequencyHoppingStartServer()` or `emberFrequencyHoppingStartClient()`. By default, all channels are used between **Start channel** and **End channel** and you can remove channels using `emberFrequencyHoppingSetChannelMask()` before starting the system. Next, the channels are reordered using a pseudo-random sequence generator seeded by the **Channel Sequence Generation Seed** component parameter as shown in the following figure.

Figure 4.3. Pseudo-Random Sequence Index

The radio will use these channels in a fixed order for a period specified by **Channel duration** time—a defining a channel slot. At the start and end of these channel slots, transmission is not allowed for the time configured with **Channel Guard Duration** to avoid communication problems caused by slightly desynchronized devices as shown in the following figure.

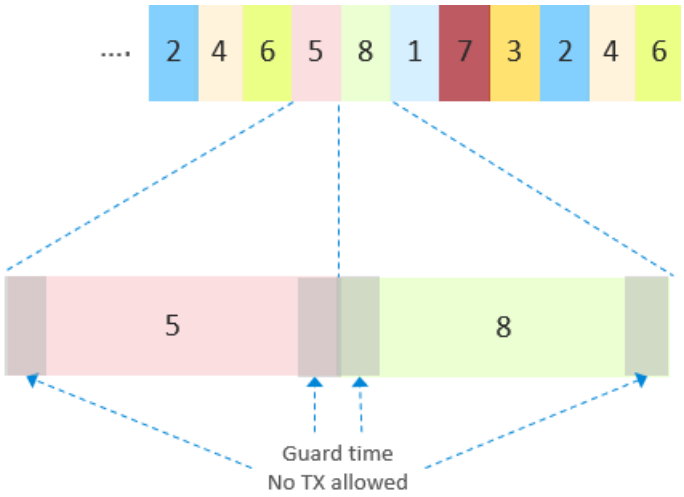


Figure 4.4. Guard Time

4.4.3 Full Synchronization

When a client wants to join to an already frequency hopping server (or if a client loses the synchronization), the client starts the full synchronization process: It will start sending *Frequency hopping info request* messages quickly hopping through the channels in reverse order until it receives a *Frequency hopping info*, at which point the client is resynchronized with the server. Full synchronization works the same if the client is a sleepy end device as shown in the following figure.

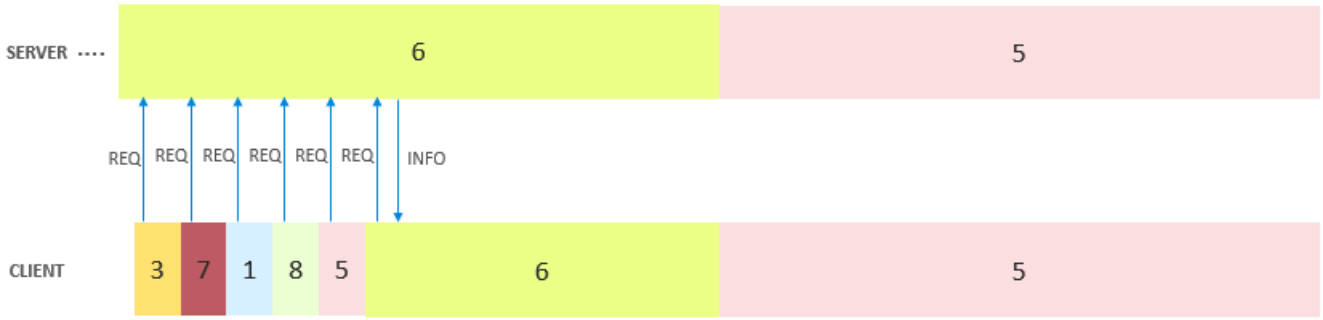


Figure 4.5. Full Synchronization

If the client loops through all the channels five times without receiving *Frequency hopping info*, it will call `emberStackStatusHandler()` with `EMBER_MAC_SYNC_TIMEOUT`.

4.4.4 Passive Synchronization

Passive synchronization is used to keep non-sleepy devices in sync. The server periodically broadcasts a *Frequency hopping info* message and if a client receives it, the client will adjust its synchronization. The period of these broadcast is configured with **Server Broadcast Info Period**. However, the period is not exact. The message will always be sent in the middle of the next slot after the period is passed to reach clients that are out of sync by as much as half a **Channel duration** as shown in the following figure.

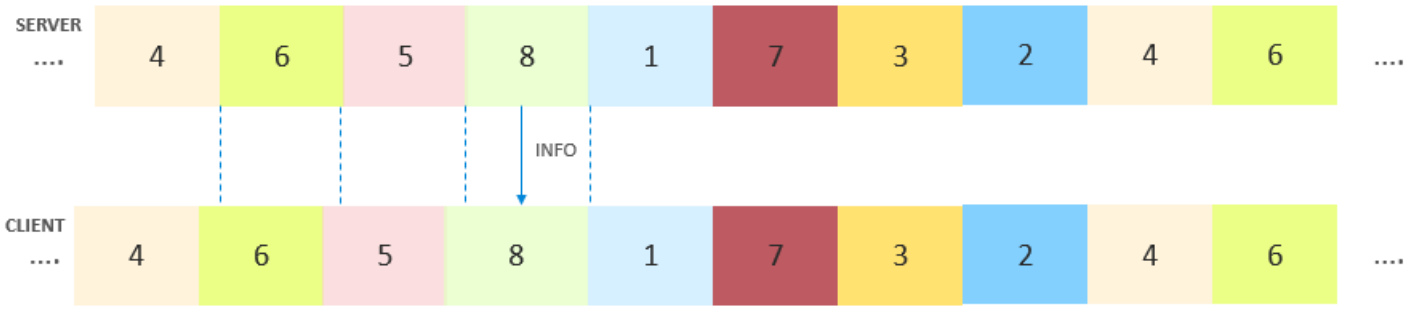


Figure 4.6. Passive Synchronization

If a non-sleepy client does not receive *Frequency hopping info* messages for **Client synchronization timeout**, it will call `emberStackStatusHandler()` with `EMBER_MAC_SYNC_TIMEOUT`.

4.4.5 Active Synchronization

Active synchronization is used to keep sleepy devices in sync. If a sleepy client has not received *Frequency hopping info* message for **Client resync period**, the next transmit attempt will be preceded by an active synchronization: The client will send a *Frequency hopping info* request in the middle of the next slot as shown in the following figure.

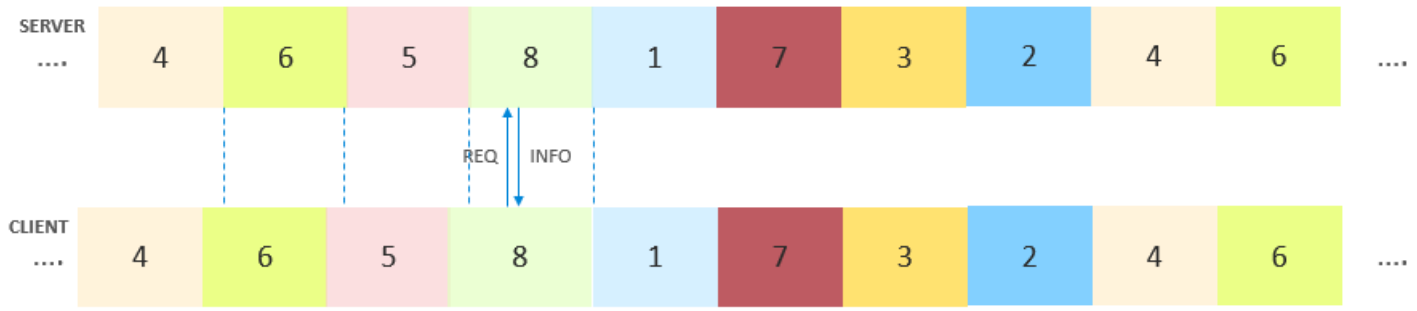


Figure 4.7. Active Synchronization

If the client receives no *Frequency hopping info* message upon its request, it will trigger a full synchronization.

4.4.6 Server Advertising

When the server starts, it will start with an advertising period. The goal of this period is to resynchronize clients that are already running. This could happen, for example, if the server restarted after a watchdog reset. This mode is basically the inverse of full synchronization: The server quickly hops through all channels in reverse order, sending a *Frequency hopping info* message on each channel. The server loops through all channels **Server advertisement sequence count** times as shown in the following figure.

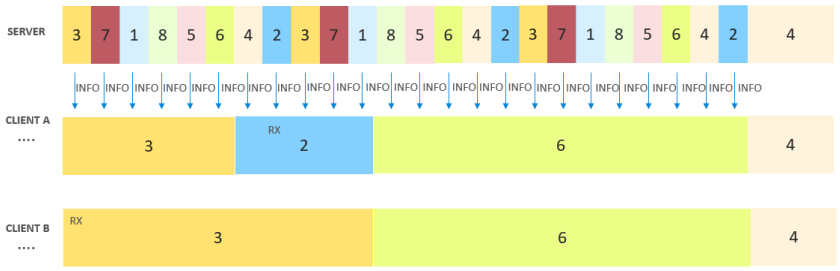


Figure 4.8. Server Advertising

In the above figure, Client B received the first Info message sent, but Client A did not (for example, because of some interfering signal). Therefore, Client A only synchronizes the next time it matches it channels with the server.

4.4.7 Summary of Component Parameters

The parameters for the components are as follows:

- **Channel Sequence Generation Seed:** The seed number which generates the pseudo-random sequence.
- **Start channel:** The start or minimum of the pseudo-random sequence channel range. Configured inclusively, that is, start channel will be in the sequence.
- **End channel:** The end or maximum of the pseudo-random sequence channel range. Configured inclusively, that is, end channel will be in the sequence.
- **Channel Duration:** How long in milliseconds the nodes will spend on each channel.
- **Channel Guard Duration:** How long in milliseconds the nodes will not transmit when entering and leaving a channel 'slot'. Transmit requests during this time from the application will be deferred until the guard duration is over. The guard duration is inside the channel duration. For example, with 400ms channel duration and 25ms guard duration, a node changes channels, waits 25ms, transmits for 350ms, waits 25ms, 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 re-sync.
- **Client Resync Period:** The time in seconds after which a sleepy client requests server information if no re-sync happened.
- **Client synchronization timeout:** The time in seconds after which a non-sleepy client stops and reports hopping sync failure if no re-sync happened.
- **Server advertisement sequence count:** The number of times the server loops through all the channels at startup to re-synchronize already running children.

5. Network Layer

The Connect network layer is only available in Extended Star mode and Direct mode. It is responsible for routing (Extended Star mpode only) and endpoint handling. Endpoints implement channel sharing between protocols, similar to TCP/IP's port concept. For example, the Mailbox component 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 the following figure.

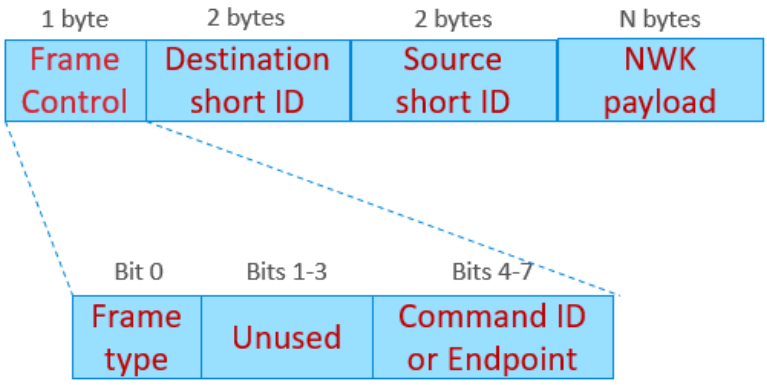


Figure 5.1. Network Layer Header

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 Mode 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. If the child table is full and a new child wants to join, the coordinator can remove a child which has not communicated for "Child timeout" amount of time (configurable in the *Parent support* component, 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).

5.2.1.3 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 Leaving the Network (Extended Star Mode only)

A child (end device, sleepy end device, or range extender) can leave the network (without notifying its parent) by calling `emberResetNetworkState()`. This action will burden the network with failed message routing attempts until the device times out from the routing and child tables. To avoid this problem, a device can notify its parent before leaving the network by calling `emberNetworkLeave()` (instead of `emberResetNetworkState()`). This will trigger a *Leave notification* message to be sent. Upon receiving the message, the parent will remove the child from its child table. If the parent does not acknowledge the *Leave notification*, the child will try to send it 7 more times (with 200ms delay between the messages) before leaving the network.

A parent can also request that a child be removed from the network by calling `emberRemoveChild()`. This will immediately mark the child ID in the child table to be reusable. (Hence, if the child table is full, it may be reassigned to any new child that is allowed to join.) The function will also cause a *Leave request* message to be sent. For a non-sleepy device, this will happen immediately. For a sleepy device, it will be sent as a response to data polls. The child will be removed from the child table if it acknowledges the *Leave request* message, or if it does not respond to 8 *Leave request* messages. *Leave request* messages will be sent when the child tries to communicate with the parent.

5.4 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
- 0x05: Leave request
- 0x06: Leave notification

All of these commands use secure communication if both sides support it.

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 more information, see section 4.2.4.5.1 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.

5.5 Network Layer Payload

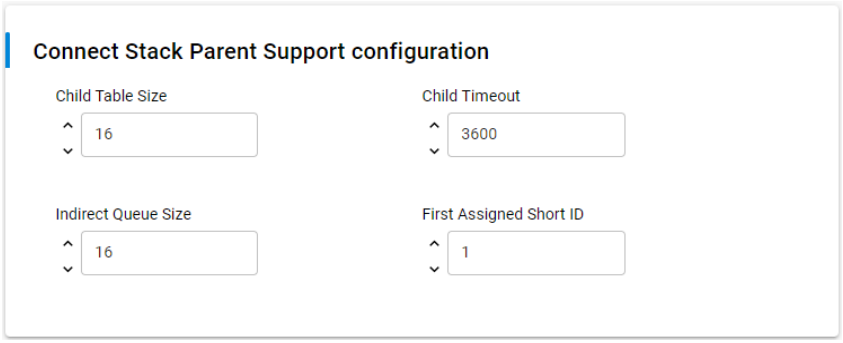
The only requirement on the payload is that it must exist: 0 length messages are not supported. Messages with no payload are usually used for beaconing. Silicon Labs recommends using data poll messages for beaconing.

6. Stack Configuration

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

6.1 Component Options in the Connect Stack Group

Many components in the Connect stack group have configuration options like this for the Parent Support component.

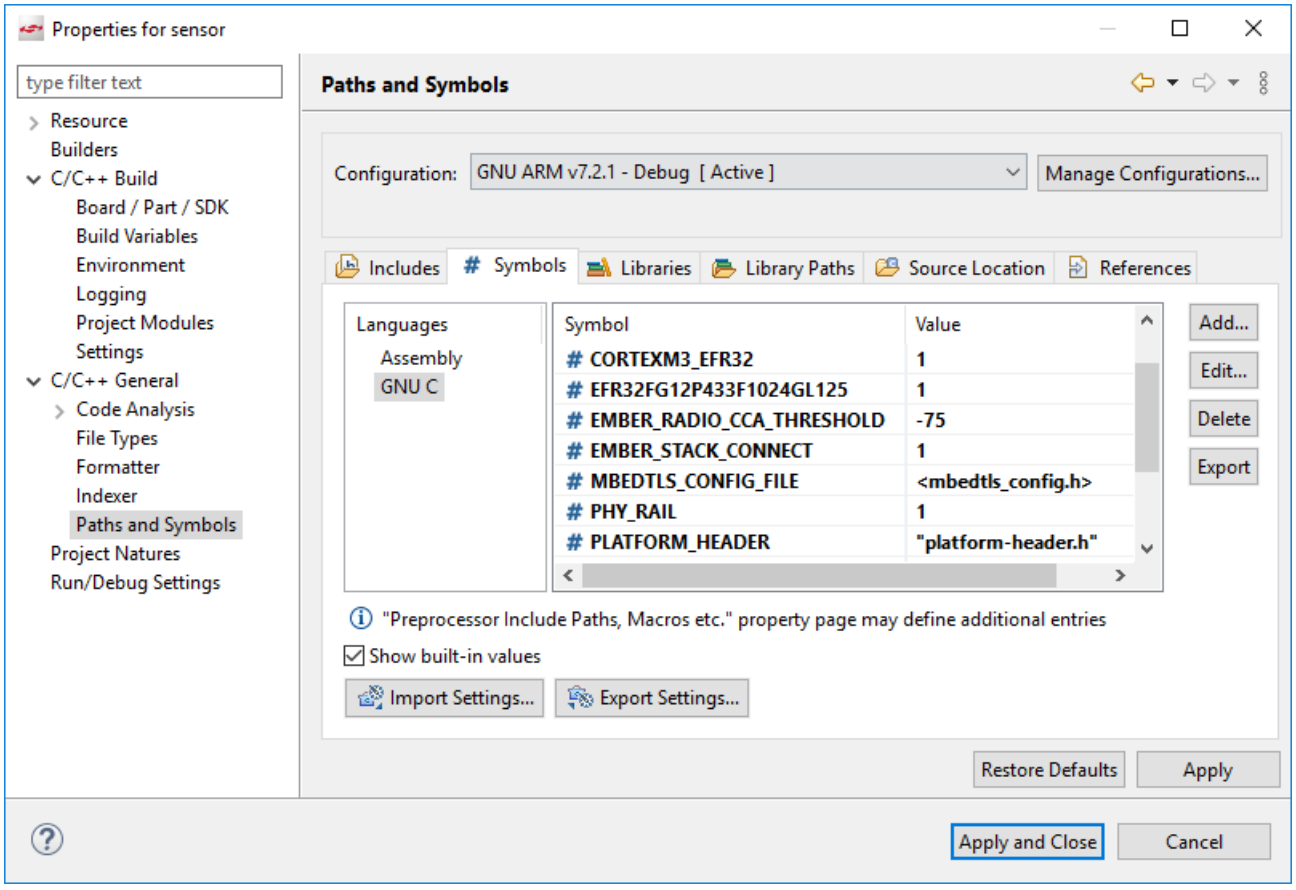


The configuration options are generated into a logically named header (for example, parent-support-config.h) under the config folder of the project.

6.2 Compile-time Macros

The Connect stack uses several macros that are not configurable as component options. 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/stack/config/ember-configuration-defaults.h`.

Compile-time macros can be added under the project properties as shown in the following figure.



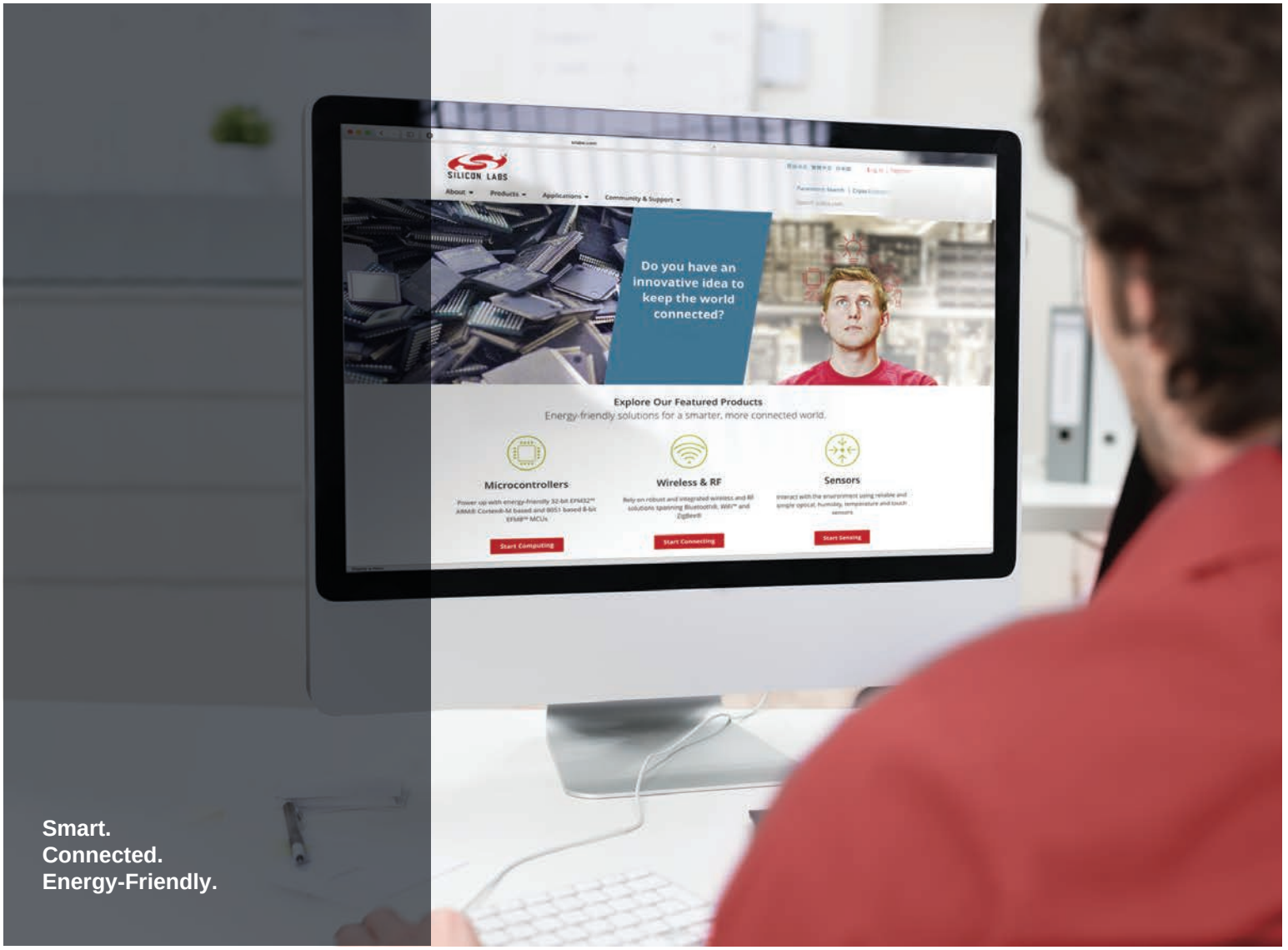
6.3 Manufacturing Tokens

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 more information, see *AN961: Bringing Up Custom Devices for the EFR32MG and EFR32FG 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 EFR32MG and EFR32FG 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