UG435.07: Energy Saving with Silicon Labs Connect v3.x

This chapter of the *Connect v3.x User's Guide* discusses techniques to reduce power consumption of network applications based on the Silicon Labs Connect Stack. 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.x User's Guide*.

KEY POINTS

- Discusses Silicon Labs Connect energy modes.
- Describes the support for sending data to sleepy end devices.
- Describes additional methods for reducing

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 SDK 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 to Energy Modes

In battery-powered microcontroller applications, saving energy is essential. By reducing current consumption, the application's effective battery life can be significantly increased.

The Wireless Gecko (EFR32™) portfolio supports five Energy Modes:

- Run Mode (Energy Mode 0)
- Sleep Mode (Energy Mode 1)
- Deep Sleep Mode (Energy Mode 2)
- Stop Mode (Energy Mode 3)
- Hibernate Mode / Shut Off Mode (Energy Mode 4)

Of these five energy modes, Silicon Labs Connect supports EM0, EM1, and EM2.

1.1 Energy Mode 0

This is the default mode. In this mode, the CPU fetches and executes instructions from flash or RAM, and all peripherals are available.

1.2 Energy Mode 1

In sleep mode, the clock to the CPU is disabled. All peripherals, as well as RAM and flash, remain available. By using the Peripheral Reflex System (PRS) and DMA, several operations can be performed autonomously. This helps save power by halting the main loop of the application but does not interfere with the timely handling of interrupts. For example, the radio can still receive packets in EM1.

1.3 Energy Mode 2

In deep sleep mode, no high-frequency oscillators run, which means that only asynchronous and low-frequency peripherals are available. This mode further improves energy efficiency while still allowing for a range of activities. In this mode, the radio is shut down and the node will not receive packets.

1.4 Power Manager Component

Power Manager is a platform-level software module that manages the system's energy modes. In case of the example applications, the Power Manager component is enabled by default. Silicon Labs recommends using Power Manager to control the energy modes of the applications.

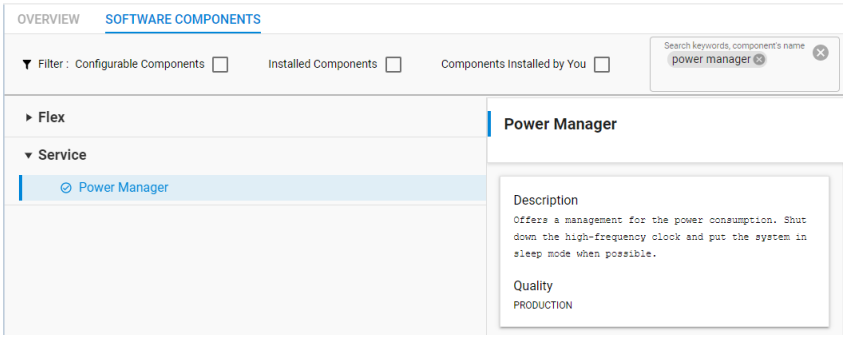


Figure 1.1. Silicon Labs Connect Power Manager Component

The application and the components can control the lowest allowed energy mode through the following power manager API calls:

```
void sl_power_manager_add_em_requirement(sl_power_manager_em_t em);
void sl_power_manager_remove_em_requirement(sl_power_manager_em_t em);
```

1.5 Support for Sending Data to Sleepy End Devices

When a sleepy end device is in sleep mode, the MCU is stopped and the radio is turned off. As a result, the device is not able to receive data from other parties. To overcome this issue, when a sleepy device wakes up it check whether messages are waiting to receive. Silicon Labs Connect supports two ways to send data to sleepy end devices:

- Indirect Queue
- Mailbox (uses Indirect Queue)

Note: If a device is configured as a sleepy end device, the radio is always in idle mode except when it transmits or during some limited period when it waits for specific message such as an acknowledge or pending frame. Conversely, non-sleepy devices are always in reception mode except during transmit periods.

1.5.1 Indirect Queue

Indirect queue is a component within parent support, which is implicitly enabled in coordinators (or range extenders) in star (or extended star) topologies due to the Parent Support component. On the device side, the Poll component must be enabled to retrieve information from the coordinator—for example ,in cases when a request needs longer processing time, an end device may choose to sleep (to allow processing to complete) and retrieve the information when it wakes up later.

The coordinator holds such data until it is requested by the relevant end device. In this case, to obtain data from the coordinator an end device must first poll the coordinator to determine whether any data is available. To achieve this, the device sends a data request which the coordinator acknowledges. Then, the coordinator determines whether it has any data for the requesting device. If it does, it sends a data packet which the receiving device may acknowledge.

1.5.1.1 Coordinator/Extender Settings

There are two parameters in the Parent Support component that modify the indirect queue behavior:

- Indirect Queue Size determines how many packets can be in the queue at a time. If the queue is full, no additional items can be added to the queue until space in the queue is recovered—either by items being sent to the recipient device, or by items aging out (timeout reached).
- The Indirect Transmission Timeout value is specified in milliseconds. Messages older than the specified timeout will be dropped by the coordinator.

The following figure shows where to select the Silicon Labs Connect Parent Support component.

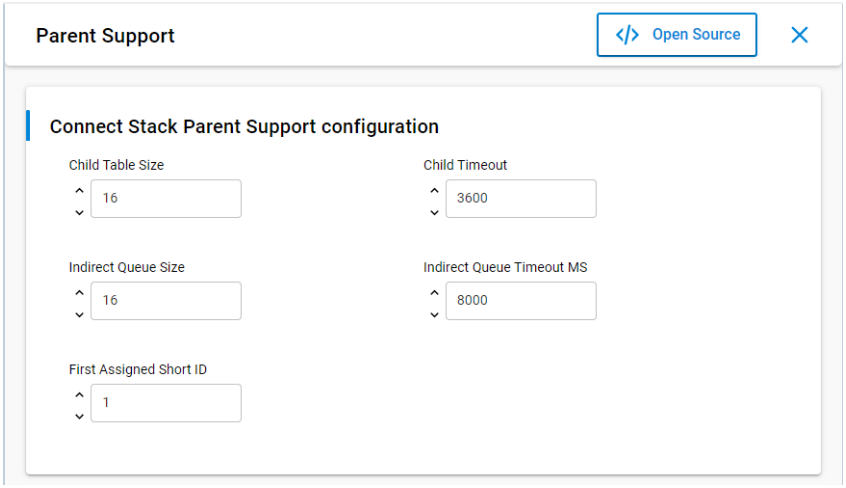


Figure 1.2. Silicon Labs Connect Parent Support Component

On the coordinator side, no additional effort need be taken. In Connect, the stack handles the messages sent to a sleepy device.

1.5.1.2 Sleepy End Device Settings

On a sleepy end device, the Poll component should be enabled to receive stored messages from the coordinator. The following figure shows the available settings of the Poll component.

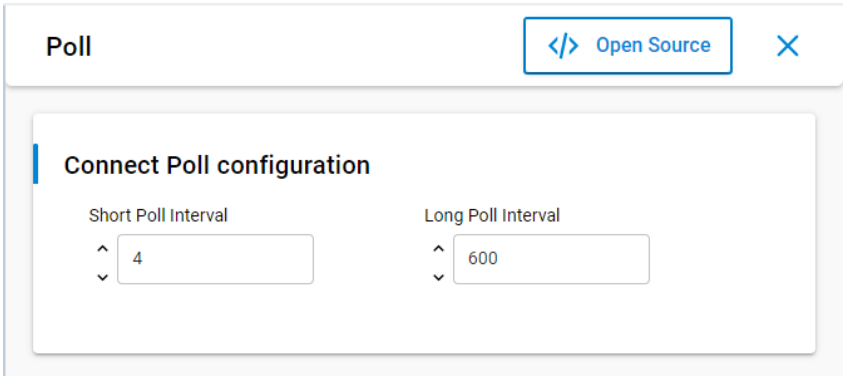


Figure 1.3. Silicon Labs Connect Poll Component

When the sleepy end device polls for messages, it sends a Data request to the coordinator and waits for the Acknowledge. If there is pending data addressed to the sleepy end device, the coordinator sets the Frame Pending bit in the Frame Control Field. Otherwise, that bit is not set. The sleepy end device checks the Frame Pending bit and waits for the next message from the coordinator. If the Frame Pending bit is not set, there is no pending message for the device and therefore it can return to sleep mode.

The Poll component supports short and long poll intervals. Short poll is used to obtain data from the coordinator while long poll is used for "keep alive" purposes to make sure the sleepy end device is kept in the coordinator's child table.

Switching between short and long polling is possible with the following Silicon Labs Connect API call:

```
void emberAfPluginPollEnableShortPolling(bool enable);
```

To poll for data the Poll component uses this API call:

```
EmberStatus emberPollForData(void);
```

which sends a data request command to the parent node.

If there is a message waiting in the indirect queue, the device will receive it through this callback:

```
void emberAfIncomingMessageCallback(EmberIncomingMessage *message)
```

With a single poll, a device can retrieve only a single message. Therefore, to receive all pending messages, it is necessary to poll repeatedly until no additional messages are received.

1.5.2 Mailbox

The Mailbox function is implemented at the application level. Mailbox is used to send information to a sleepy end device from the coordinator or from another device. In the case of sleepy devices, Mailbox uses the Indirect Queue feature of the stack to retrieve messages. Mailbox consists of two parts:

- Mailbox Client
- Mailbox Server

With Mailbox, assignment of the client or server role is flexible. (For example, it is possible to add the server to an end device and the client to the coordinator.) Mailbox uses endpoints to transfer messages between clients and server. The endpoint selected for the protocol is set in both the server and client component, and they must match. On the server side, two other parameters can be set: the maximum number of packets the server can store and their timeout. These values can be set in the Universal Configurator (see the following two figures).

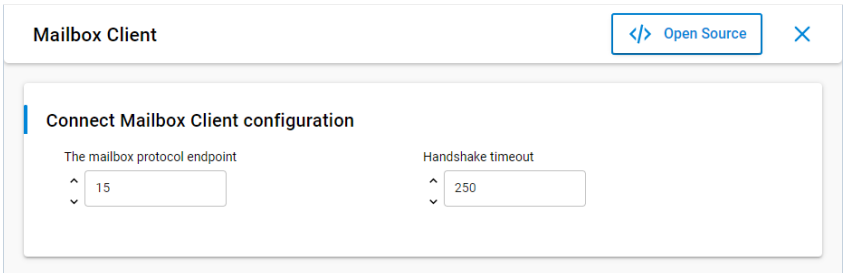


Figure 1.4. Silicon Labs Connect Mailbox Client Component

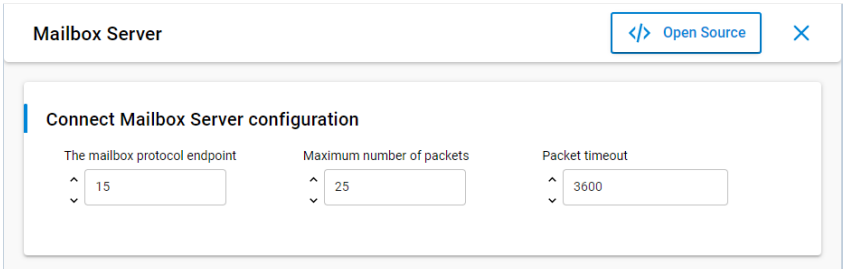


Figure 1.5. Silicon Labs Connect Mailbox Server Component

Mailbox is not supported in MAC mode because it uses endpoints which are implemented in the Silicon Labs Connect Network layer. However, it works in Connect Direct mode where the application should implement sleep.

The main advantage of using Mailbox is the server will notify clients that submit messages when a message was delivered or when it could not be delivered due to an error.

1.5.2.1 Mailbox Client

The Mailbox client can submit messages and check the inbox. In both cases, the result is delivered through callback functions.

Check-related API calls:

```
emberAfPluginMailboxClientCheckInbox()
emberAfPluginMailboxClientCheckInboxCallback()
```

Submit-related API calls:

```
emberAfPluginMailboxClientMessageSubmit()
emberAfPluginMailboxClientMessageSubmitCallback()
emberAfPluginMailboxClientMessageDeliveredCallback()
```

Note: The default value of the short poll interval is 4 quarter seconds (1s) while the Handshake timeout of the Mailbox Client is set to 250ms. Thus, it can happen that the Mailbox Client timeouts during waiting to poll for the pending message. Silicon Labs recommends setting these values to suit the application. This usually means that the Handshake timeout should be higher than the short poll interval.

1.5.2.2 Mailbox Server

On the server side, to set up the Mailbox system, you only need to enable the Connect Mailbox Server component. No additional code is necessary. However, if desired, there are available support functions that enable the server to add messages directly to the mailbox using these API calls:

```
emberAfPluginMailboxServerAddMessage()
emberAfPluginMailboxServerMessageDeliveredCallback()
```

1.6 Additional Methods for Reducing Energy Consumption

To fully minimize power consumption, disable unnecessary features and unused peripherals.

1.6.1 SPI Flash

All Silicon Labs EFR32 radio boards contain SPI Flash. Although these flash devices consume only 8-10µA in standby state, this constant drain will increase the apparent sleep current and reduce effective battery life. However, these flash devices have a deep power down mode which is accessed by issuing an SPI command. In the case of Silicon Labs radio boards, the default configuration is that the firmware puts the external SPI flash into deep power down mode through the MX25 Flash Shutdown component.

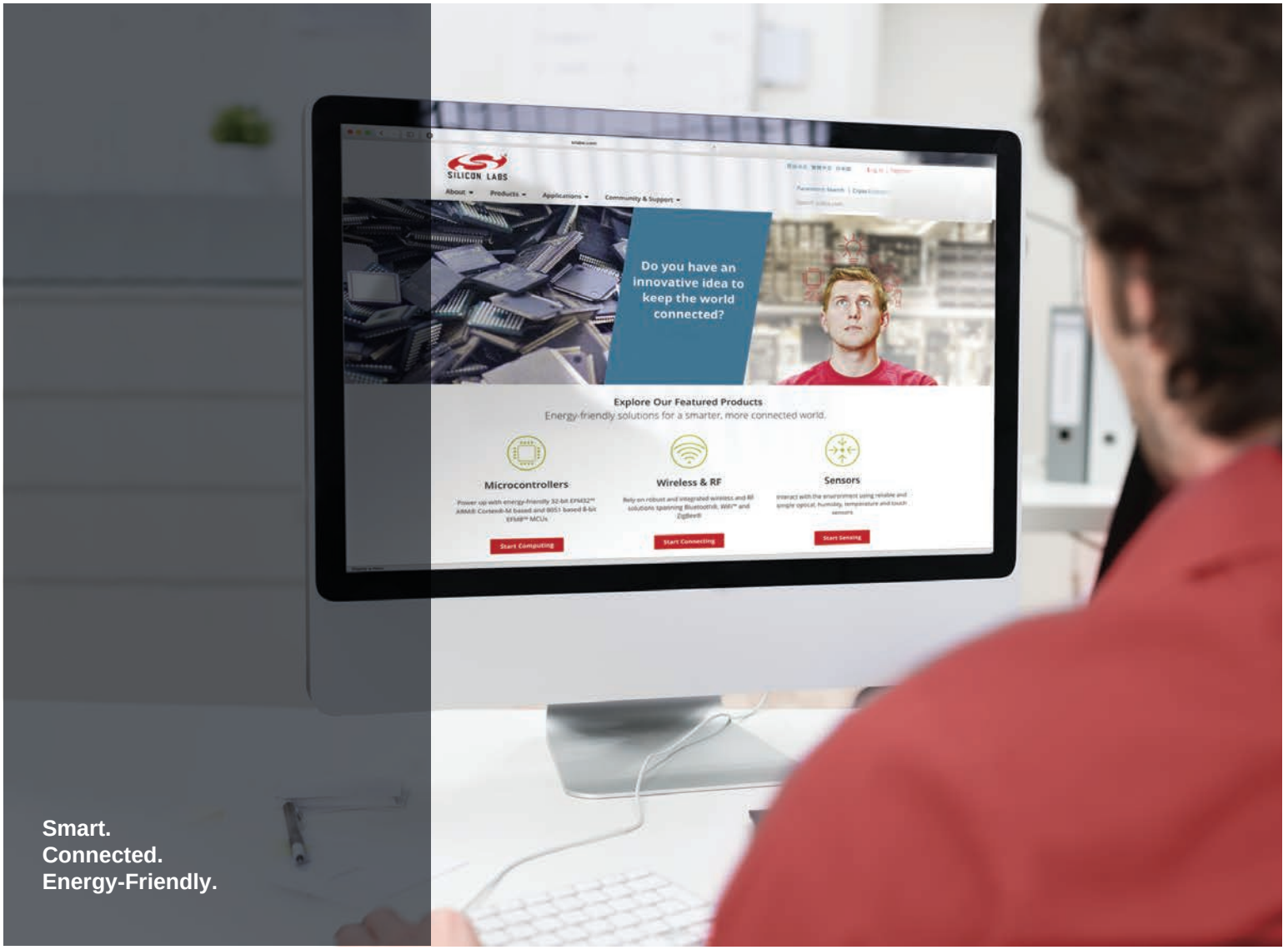
1.6.2 Network Up LED

By default, some Connect examples use an LED to show the network stack status. This LED is turned on when the stack is up.

```
void emberAfTickCallback(void)
{
  if (emberStackIsUp()) {
    sl_led_turn_on(&sl_led_led0);
  } else {
    sl_led_turn_off(&sl_led_led0);
  }
}
```

To save energy, remove the `sl_led_turn_on(&sl_led_led0);` line or change it to `sl_led_turn_off(&sl_led_led0);`.

As a rule of thumb, the aim is to spend as much time in sleep mode as possible and wake up only for short periods when necessary. Applying the above techniques, the sleep-mode current consumption of Silicon Labs Connect-based applications can be reduced to 2-4 µA.



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