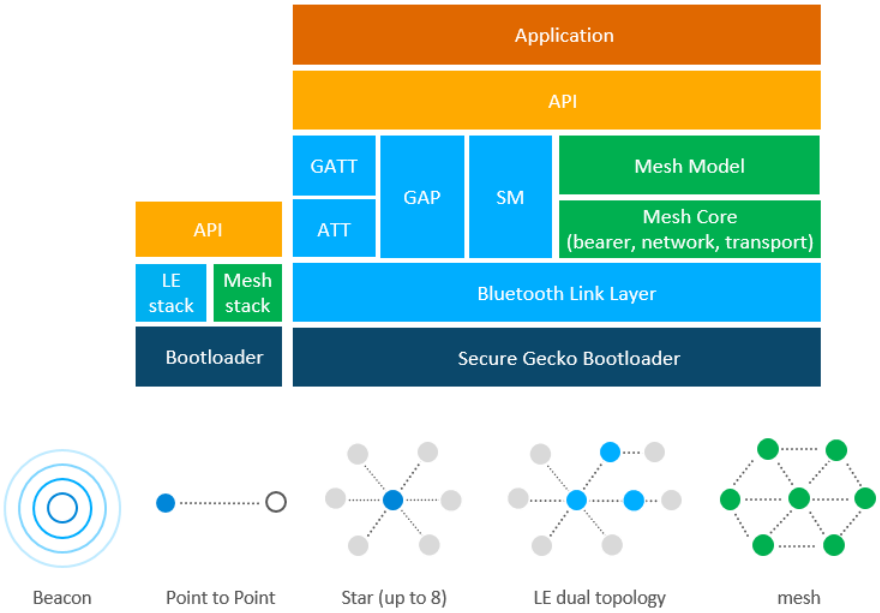
AN1098: Understanding the Silicon Labs *Bluetooth*® Mesh Lighting Demonstration in SDK v1.x

The Bluetooth mesh SDK comes with an example project that creates a wireless network of lights and switches using Bluetooth mesh technology. The example assumes usage of Silicon Labs WSTKs for switches and lights and an Android or iOS mobile phone for provisioning and controlling the network. In this document we discuss the basics of Bluetooth mesh required to understand the example, and walk through key aspects of the application source code.

KEY FEATURES

- Short introduction to Bluetooth mesh
- Lighting example application description and code walkthrough
- Silicon Labs Bluetooth mesh mobile application

This document assumes you have read *QSG148: Getting Started with the Silicon Labs Bluetooth Mesh Demonstration Software,* installed the Bluetooth mesh SDK, and successfully run the example.



Bluetooth LE and Mesh Stacks and Supported Topologies

1 Introduction

This document focuses on explaining the Bluetooth mesh lighting demo, installed as part of the Bluetooth mesh SDK. For the most part the documentation centers on the example application and its usage flow, explaining key parts of the source code and the Silicon Labs Bluetooth Mesh mobile application, but also includes a brief discussion of some concepts of the specification important for understanding the example.

In the following subsections, we briefly walkthrough the relevant aspects of the Bluetooth mesh technology. Section 2 Bluetooth Mesh Lighting Demonstration describes the features and functions of the Lighting Demonstration, section 3 Network Analyzer describes using Network Analyzer for packet capture, and section 4 Bluetooth Mesh Stack and Application for Smartphones focuses on the mobile application.

1.1 Bluetooth Mesh

Bluetooth mesh is a new topology available for Bluetooth LE devices and applications. Previously Bluetooth devices have been using point-to-point connectivity or broadcasting topologies to communicate with other devices. Bluetooth mesh extends that and allows both many-to-many device communications and using Bluetooth devices in a mesh topology. This enables multi-hop communications between Bluetooth devices and much larger-scale Bluetooth device networks than have been possible previously.

Bluetooth mesh uses Bluetooth LE advertising channels to send and receive messages between the Bluetooth mesh nodes, but it can also use Bluetooth connections and GATT services to communicate with devices that do not natively support Bluetooth mesh.

Bluetooth mesh also uses its own security architecture, which is separate from the normal Bluetooth LE security architecture, although the same AES-CCM 128-bit and Elliptic Curve Diffie Hellman (ECDH) security algorithms are used.

Bluetooth mesh also defines its own application layer called mesh model which is different than the GATT-based profiles and services non-mesh Bluetooth LE devices use. The new application layer was defined to address the requirements and needs of mesh-based topologies and also to make Bluetooth mesh a full stack solution and enable interoperable mesh devices to be built.

1.1.1 Bluetooth Mesh Network Roles and Node Features

The Bluetooth mesh network typically consists of multiple nodes. All nodes can transmit and receive mesh messages, but they can optionally also support one or more additional features. If a node does not implement any of the additional features it is considered just a node. Various node types are illustrated in the following figure.

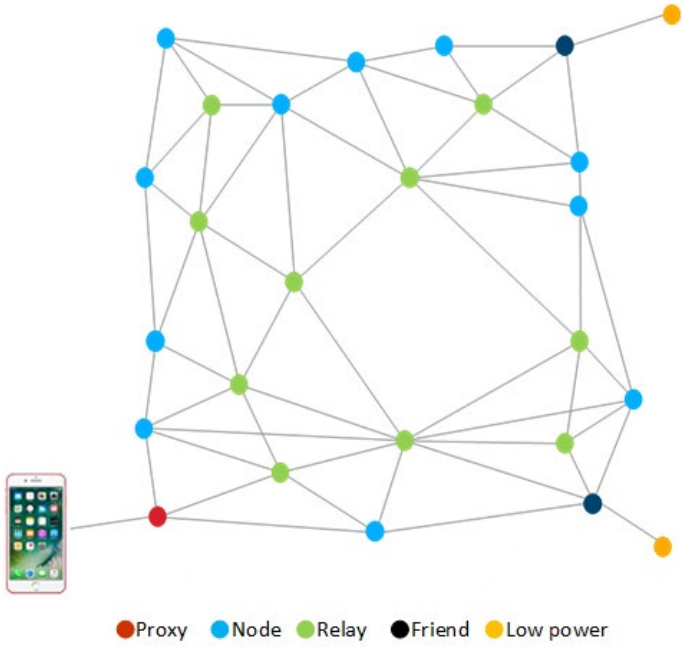


Figure 1-1. Node Types

The four types of specified node features are as follows:

Proxy feature: Enables message proxy between Bluetooth mesh and GATT and enables devices such as smartphones to connect to Bluetooth mesh.

Relay feature: Relays messages to extend the range and scale of a Bluetooth mesh network.

Friend feature: Implements an additional message cache to support nodes with the low power feature.

Low power feature: Allows sleeping and polling of messages from friend nodes at known time intervals.

For further information on these features and Bluetooth mesh technology, please go to the Silicon Labs Bluetooth mesh learning center.

1.1.2 Provisioning

Provisioning refers to the operation where devices that are not part of any Bluetooth mesh network are transformed into nodes that are part of one or more Bluetooth mesh networks. For example, provisioning happens when a new light bulb is installed and taken into use, so it can be controlled by switches or dimmers.

Provisioning is mainly a security process where the first level security keys are generated by the provisioner and transferred to the device that is being provisioned to make it part of a Bluetooth mesh network.

The provisioning process begins when a device starts to send unprovisioned Bluetooth beacon packets and the provisioner receives them. The provisioner then initiates the provisioning process, the devices exchange public keys, and both generate session keys. The session keys are used to secure the session, in the transfer of the actual network key, and the rest of the provisioning process. After provisioning, each device, now a node in the network, has the network key, a security parameter called the IV index, and its unicast address.

1.1.3 Publish and Subscribe

In Bluetooth mesh, communication to a group of devices is typically implemented through a publish and subscribe mechanism. This is an easy-to-understand concept which also simplifies the setup of Bluetooth mesh networks and adding and reconfiguring nodes.

Usually the Bluetooth mesh nodes are configured into groups, which may represent their physical location (kitchen or living room) or specific function (lights or window coverings). Usually the devices are also controlled as groups, so the same message is sent to all devices in a group. To accomplish this functionality Bluetooth mesh uses a concept called publish – subscribe, where nodes such as lights subscribe to messages groups and nodes like switches publish messages to those groups. At the network layer each group is assigned a group address and multicast messaging is used to send the messages to all devices in a specific group.

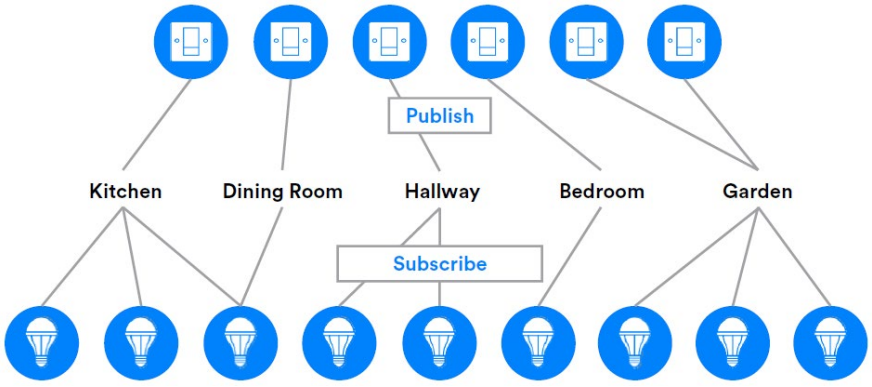


Figure 1-2: Publish and Subscribe

The benefit of publish and subscribe is that, when a new node is added or an existing node is removed or replaced, only that node needs to be provisioned and configured.

2 Bluetooth Mesh Lighting Demonstration

2.1 Requirements

- Simplicity Studio
  - Bluetooth Mesh SDK 1.7.0 or later, distributed through Simplicity Studio 4
  - The pre-built demo binaries and source code are included in the SDK.
  - Simplicity Studio has a network analyzer capable of capturing and decoding Bluetooth mesh packets.
  - The actual code development can be done with Simplicity Studio, IAR EWARM, or command line tools
- Silicon Labs Bluetooth mesh mobile application
  - Available for both iOS and Android
  - Used for discovering and provisioning devices over GATT.
  - Includes network, group and publish-subscribe setup.
  - Allows device configuration and control.
  - Requires iOS 10 or later.
  - Requires Android 6 (API23) or later.
- For the full experience at least 3 pcs of Silicon Labs Blue Gecko SoC Wireless Starter Kits are needed.
  - 2 kits are used as lights with proxy feature.
  - 1 kit is used as a switch.
  - EFR32BG12, EFR32MG12, EFR32BG13, EFR32MG13, EFR32xG21, and EFR32xG22 SoCs as well as the BGM13P, and BGM13S modules support Bluetooth mesh software.

See *QSG148: Getting Started with the Silicon Labs Bluetooth Mesh Lighting Demonstration* for more information on obtaining required hardware and software, and running the demonstration.

The demonstration setup can, in principle, consist of any number of switch nodes and light nodes. A single switch node can control an arbitrary number of light nodes by sending commands to a group address. Similarly, a light node can receive on/off commands from multiple switches.

2.2 Mesh Network Implementation

The demonstration implementation process can be divided into four main phases as follows:

1. Unprovisioned mode – After the demo firmware is installed, the device starts in unprovisioned mode.
2. Provisioning – The devices are provisioned to a Bluetooth mesh network and network security is set up.
3. Configuration – The group, publish and subscribe, and application security are configured.
4. Normal operation – The light node(s) can be controlled by the switch(es) and also from the smartphone application.

In the first phase, all the devices are unprovisioned and transmitting unprovisioned beacons. They do not have any network keys or application keys configured and publish and subscribe settings are not set. In this state the devices are simply waiting for the provisioner to assign them into a Bluetooth mesh network and configure publish and subscribe and mesh models. In this state the devices can be detected by the smartphone application.

In the provisioning phase the provisioner adds lights and switches to the Bluetooth mesh network. A network key is generated and distributed to the nodes and each node is assigned a unicast address.

In the configuration phase the provisioner configures: groups, publish and subscribe settings, application-level security, and mesh models.

After provisioning and configuration, the Bluetooth mesh network is operational and switches can be used to control the lights. The WSTK switch's buttons can be used to control all the lights in a group. The same functionality can be done with the smartphone application, but it can also control individual lights using unicast addressing.

2.3 Code Walkthrough

The Bluetooth mesh SDK includes separate example projects for the switch and for the light. These are named **BT Mesh – Light Example** and **BT Mesh – Switch Example**. Both examples are implemented using the same event-driven architecture that is used in plain Bluetooth (non-mesh) applications.

For information about Bluetooth C application development please see *UG136: Silicon Labs Bluetooth® C Application Developer's Guide*.

2.3.1 Unprovisioned Mode, Provisioning, and Configuration

In unprovisioned mode, both examples behave the same way. The unprovisioned device simply starts sending unprovisioned beacons and waits for a provisioner to provision and configure it.

After receiving the `system_boot` event (`gecko_evt_system_boot_id`), the application checks if a button is pressed. If yes, it calls the function `initiate_factory_reset()`, which closes connections if any exist and performs a factory reset by erasing PS storage. The factory reset is also done after receiving `mesh_node_reset` event (`gecko_evt_mesh_node_reset_id`). If no button is pressed, then the name of the device is set based on the Bluetooth address and the function `gecko_cmd_mesh_node_init()` is called to initialize the Bluetooth mesh node stack.

The event `gecko_evt_mesh_node_initialized_id` indicates that the Bluetooth mesh node stack initialization is complete. This event also includes information about the node status. The application first checks the provisioning status. If the node is not provisioned (the default state when the device is first powered up after programming) then the application starts unprovisioned beaconing by calling `gecko_cmd_mesh_node_start_unprov_beaconing()`.

The API call `gecko_cmd_mesh_node_start_unprov_beaconing` takes one parameter (`bearer`) that selects which bearers are used (PB-ADV, PB-GATT, or both). In this example, both bearers are used. Because the PB-GATT bearer is enabled, the device will begin advertising its provisioning GATT service. This allows the smartphone application to detect unprovisioned nodes.

When unprovisioned beaconing has been started the application waits for the provisioner (in this case, the smartphone app) to start provisioning. Start of provisioning is indicated with the event `mesh_node_provisioning_started`.

During provisioning, no actions are required from the user application. The configuration of network keys and other operations are handled automatically by the Bluetooth mesh stack. Both the light and the switch application simply start blinking the two LEDs on the WSTK to indicate that provisioning is in progress. Then they wait for the event `gecko_evt_mesh_node_provisioned_id` that indicates provisioning is complete.

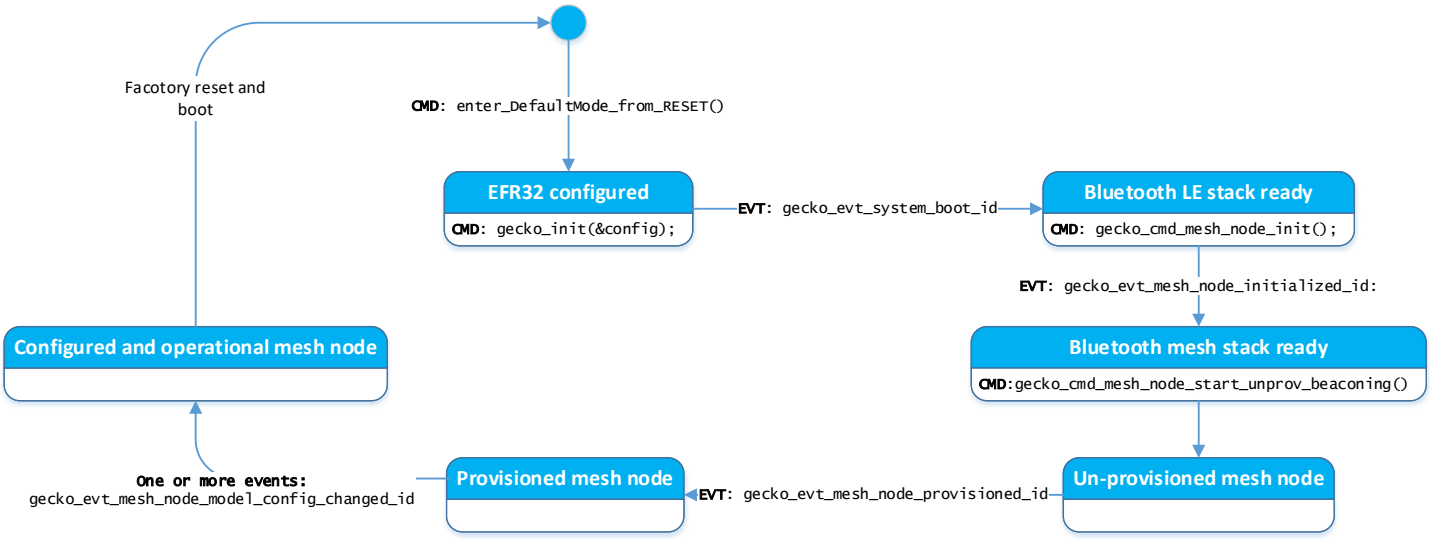


Figure 2-1: Life Cycle of the Application

The next step after provisioning is configuration of the node. As explained in *QSG148: Getting Started with the Silicon Labs Bluetooth Mesh Lighting Demonstration*, the smartphone app is used to configure a node either as a switch or a light and the node is assigned to a group. The configuration procedure consists of following steps:

- Provisioner distributes an application key to the node.
- Application key is bound to the selected Bluetooth mesh model.
- Publish address and settings are configured.
- Subscribe address and settings are configured.

The configuration phase is mostly handled between the Mesh stack and the provisioner and it does not require any involvement from the user application in the node. The following events are generated by the stack to give status information about the ongoing configuration:

- `gecko_evt_mesh_node_key_added_id`: generated when the provisioner has sent a new key (network or application)
- `gecko_evt_mesh_node_model_config_changed_id`: indicates that the provisioner has modified configuration of the local model (either publish or subscribe settings changed)

The node application includes some debug prints to keep track of the configuration process, but no action is needed in the application code when either of these events is raised.

Up to this point, the code in the examples **BT Mesh – Light Example** and **BT Mesh – Switch Example** is almost identical.

2.3.2 Switch Node Example

This section describes basic operation of the **BT Mesh – Switch Example.** It is assumed that the node is already provisioned and publish - subscribe settings have been configured by the smartphone app. The switch node has one simple task: listen for push-button presses and, based on the button press length, control the brightness, color temperature, or on/off state of the lights in the group or recall previously stored scenes. Short button presses (less than 250 ms) are used to adjust light brightness up (PB1) and down (PB0). Medium button presses (more than 250 ms and less than 1 s) are used to adjust light color temperature up (PB1) and down (PB0). A long press (more than 1 s and less than 5 s) turns the light on (PB1) or off (PB0). A very long press (more than 5 s) recalls scene number 1 (PB0) or scene number 2 (PB1).

For boards that support only one button, the behavior is slightly different. Short button press (less than 250 ms) is used to adjust light brightness up. When it achieves maximum, the next press sets brightness to minimum. Medium button press (more than 250 ms and less than 1 s) is used to adjust light color temperature up. When it achieves maximum, the next press sets temperature to minimum. A long press (more than 1 s and less than 5 s) toggles the light. A very long press (more than 5 s) recalls the scene number 1.

The on/off control uses the **Generic OnOff Client** model, the brightness control uses the **Light Lightness Client** model, the color temperature control uses the **Light CTL Client** model, and the scene recalls use the **Scene Client** model (see the Bluetooth Mesh Model specification for more details on the Scene model). The switch example also demonstrates the low-power node (LPN) feature. When the switch is provisioned into the network, it will start looking for a friend so that it can enter low-power mode. When a friendship is established, the switch can go to deep sleep and it will wake up periodically to poll the friend node for any incoming messages.

Upon receiving the `gecko_evt_mesh_node_initialized_id` event, the node application enables GPIO interrupts for development kit buttons PB0 and PB1 (if the board supports it). Next it initializes the mesh library with the `mesh_lib_init()` function. After that the **Low Power Node** feature is initialized (`gecko_cmd_mesh_lpn_init()` function), configured (`gecko_cmd_mesh_lpn_configure()` function) and run (`gecko_cmd_mesh_lpn_establish_friendship()` function). The configuration has two parameters. The first represents the minimum queue length that a friend must support. The second is the poll timeout, which specifies the longest time between two consecutive queries from the LPN to the friend. The LPN could sleep between these queries and it allows to save the energy. The LPN feature status is displayed on the WSTK LCD display.

The LPN feature is mostly implemented in the mesh stack, so the application can receive only a few events:

- `gecko_evt_mesh_lpn_friendship_established_id`: The friendship was successfully established. On receipt, the application displays “LPN with friend” on the WSTK LCD.
- `gecko_evt_mesh_lpn_friendship_failed_id`: Friendship establishment failed. On receipt, the application starts a timer and after 2 seconds tries to establish the friendship again. It displays “no friend” on the WSTK LCD.
- `gecko_evt_mesh_lpn_friendship_terminated_id`: Friendship was terminated for some reason. On receipt, the application starts a timer and after 2 seconds tries to establish the friendship again. It displays “friend lost” on the WSTK LCD.

If a GATT connection is opened, the friendship is terminated and the LPN is de-initialized (`gecko_cmd_mesh_lpn_deinit()` function). In this case the WSTK LCD displays “LPN off”. After closing all GATT connections, the application re-initializes the LPN using the `lpn_init()` function.

The interrupt handler code measures the duration of the button press to distinguish short, medium, long, and very long presses. The number of RTCC ticks at the time of button press and release are recorded and the difference between these time stamps is used to determine the press duration.

Handling of long button presses is implemented in function `handle_long_press()`. For each long button press, the application publishes three consecutive on/off requests to the group address that has been set by the smartphone app. The request is sent multiple times for increased reliability. Note that it is the application's responsibility to choose a suitable strategy for reliable communications. In this example, multiple application message transmissions were chosen. Retransmissions can also be configured to be added automatically at the network level.

Sending a single on/off request is implemented in the function `send_onoff_request()`. A soft timer is used to trigger three calls to `send_onoff_request()` with a 50 ms delay between each call.

The mesh stack API call used to send one on/off transaction is `gecko_cmd_mesh_generic_client_publish()`. This is a common API call that is used to publish data for several client models. It is not limited to the generic on/off client only. For example, publishing data as a generic transition time client would be done using the same API call. The first parameter `model_id` selects which model is being used.

In addition to the desired on/off status, the publish API call has some additional parameters such as **transaction identifier**, **transition time** and **delay**.

The transaction identifier is a running number that is incremented for each transaction. In this example, each on/off state change triggers three consecutive on/off requests. The transaction identifier is same for each of these requests so that, at the receiving end, duplicate requests can be filtered out. In other words, all the three published messages are part of the same transaction and they will trigger only one event at the receiving light node.

The delay parameter can be used to indicate that the on/off transition should not be executed immediately but after a given delay. In this example, the delay parameter is set to values of 100 ms / 50 ms / 0 in the first, second and third request, respectively. The purpose is to ensure that all lights in the target group change their state simultaneously, regardless of which of the three on/off requests was captured on the receiving side.

Handling of short button presses is implemented in function `handle_short_press()`. Short presses are used to adjust light brightness up and down. The application sends a request using the **Light Lightness** model. The last level that has been set is stored in a variable (type uint16) and the level is adjusted up or down each time a short button press is detected.

Sending a single light lightness request is implemented in function `send_ligthness_request()`, which is very similar to `send_onoff_request()` that is used for on/off requests. Both of these use the same API call `gecko_cmd_mesh_generic_client_publish()` to publish the request. The difference is in the model ID that is passed as argument and the parameter data type.

Handling of medium button presses is implemented in function `handle_medium_press()`. Medium presses are used to adjust light color temperature up and down. The application sends a request using the **Light CTL** model. The last temperature that has been set is stored in a variable (type uint16) and the temperature is adjusted up or down each time a medium button press is detected.

Sending a single light CTL request is implemented in function `send_ctl_request()`, which is very similar to `send_lightness_request()` that is used for light brightness requests. Both of these use the same API call `gecko_cmd_mesh_generic_client_publish()` to publish the request. The difference is in the model ID that is passed as argument and the parameter data type.

Handling of very long button presses is implemented in function `handle_very_long_press()`. Very long presses are used to recall scenes. The application sends a request using the **Scene Client** model.

Sending a single scene recall request is implemented in function `send_scene_recall_request()`, which is similar to other requests. The difference is that it uses a dedicated `gecko_cmd_mesh_scene_client_recall()` function for publishing. This function is generally used for sending scene recall requests. If the destination address is set to a prohibited 0 address, the function publishes the recall message.

The application code that implements the light switch functionality is relatively simple because many aspects are automatically handled by the mesh stack. For example, the switch does not need to know anything about the light nodes that it is controlling. Any number of light nodes can be subscribed to the on/off requests that are published by the switch node.

The switch node does not need to know the group address that has been configured by the provisioning application. It simply publishes the on/off requests using the `gecko_cmd_mesh_generic_client_publish()` API call and the stack automatically sends the requests using the group address that has been configured by the provisioner.

2.3.3 Light Node Example

This section describes basic operation of the **BT Mesh – Light Example.** It is assumed that the node is already provisioned and that the publish and subscribe settings have been configured by the smartphone app.

The main feature in the light node is that the development kit LEDs are turned on or off based on the requests that are received from switch nodes or from the smartphone application. The brightness of the LEDs can also be controlled. On/off control is based on the Bluetooth mesh Generic OnOff model and the brightness control is based on the Light Lightness model. The Light CTL model supports color temperature requests. Color temperature changes are shown on the WSTK LCD display. The light node also supports the friend feature. It can establish friendship with the low-power switch nodes in the network so that the switch nodes can enter low-power mode.

The light node supports following states:

- Generic OnOff
- Generic Level
- Generic OnPowerUp
- Generic Default Transition Time
- Light Lightness
- Light CTL
- Scenes
- Time
- Schedule
- All Light LC states
- All Light LC Property states

The OnPowerUp state enables configuration of the default state after power is applied to the light node. The possible settings are listed below.

OnPowerUp Setting	Description (light node)
OFF	Light is off after power up
ON	Light is on after power up
RESTORE	The state before light was powered down is restored at next power up

The transition time model makes it possible to configure how long it takes for the light to transition from one state to another.

To support all the states listed above, the light node must store its internal state permanently so that it is preserved over reboots and power cycles. The state information is kept in structs named `lightbulb_state`, `lc_state`, `lc_property_state` and in the stack. Struct `lightbulb_state` contains the following fields.

Struct Member Name	Description
onoff_current	Current state of light (ON or OFF)
onoff_target	Target state of light (ON or OFF)
transtime	Default transition time
onpowerup	Light state after power up (possible values OFF/ON/RESTORE)
lightness_current	Current brightness (possible values from 0 to 65535)
lightness_target	Target brightness
lightness_last	Last non-zero brightness
lightness_default	Default brightness
lightness_min	Minimum lightness value
lightness_max	Maximum lightness value
pri_level_current	Current generic level on primary element (possible values from -32768 to 32767)
pri_level_target	Target generic level on primary element
temperature_current	Current color temperature (possible values from 800 to 20000)

Struct Member Name	Description
temperature_target	Target color temperature
temperature_default	Default color temperature
temperature_min	Minimum color temperature
temperature_max	Maximum color temperature
deltauv_current	Current value of delta UV (possible values from -32768 to 32767)
deltauv_target	Target value of delta UV
deltauv_default	Default value of delta UV
sec_level_current	Current generic level on secondary element (possible values from -32768 to 32767)
sec_level_target	Target generic level on secondary element

The light state initialization is implemented in function `lightbulb_state_init()`. This function initializes the mesh library by calling `mesh_lib_init()`. The mesh library is an adaptation layer between the mesh stack and the application code that enables using multiple models with a small set of generic API calls.

After initializing the mesh library, the friend functionality is initialized to enable the friend feature implemented in the stack. After successful initialization friend requests from Low Power Nodes can be accepted. The friend feature is mostly implemented in the stack, so after initialization the application only receives the status events:

- `gecko_evt_mesh_friend_friendship_established_id`: Friendship was established. The WSTK LCD displays “FRIEND”.
- `gecko_evt_mesh_friend_friendship_terminated_id`: Friendship was terminated. The WSTK LCD displays “NO LPN”.

After that, LC (Light Controller) models are initialized (see the Bluetooth Mesh Model specification for more details on the LC model). The LC state is kept in `lc_state` struct that contains following fields.

Struct Member Name	Description
mode	Current Light LC Mode state
occupancy_mode	Current Light LC Occupancy Mode state
light_onoff	Current Light LC OnOff state
onoff_current	Current generic state of LC (ON or OFF)
onoff_target	Target generic state of LC (ON or OFF)

The LC Property states are kept in `lc_property_state` struct that contains the following fields.

Struct Member Name	Description
time_occupancy_delay	Delay between receiving a sensor occupancy message and changing the Light LC Occupancy state
time_fade_on	Transition time from a standby state to a run state
time_run_on	Duration of the run state after last occupancy was detected
time_fade	Transition time from a run state to a prolong state
time_prolong	Duration of the prolong state
time_fade_standby_auto	Transition time from a prolong state to a standby state when the transition is automatic
time_fade_standby_manual	Transition time from a prolong state to a standby state when the transition is triggered by a manual operation
lightness_on	Lightness level in a run state
lightness_prolong	Lightness level in a prolong state
lightness_standby	Lightness level in a standby state
ambient_luxlevel_on	Required Ambient LuxLevel level in the Run state
ambient_luxlevel_prolong	Required Ambient LuxLevel level in the Prolong state
ambient_luxlevel_standby	Required Ambient LuxLevel level in the Standby state
regulator_kiu	Integral coefficient of PI light regulator when increasing output

Struct Member Name	Description
regulator_kid	Integral coefficient of PI light regulator when decreasing output
regulator_kpu	Proportional coefficient of PI light regulator when increasing output
regulator_kpd	Proportional coefficient of PI light regulator when decreasing output
regulator_accuracy	Accuracy of PI light regulator

Both structs are used for saving and restoring the LC states between the resets. Most of the LC functionality is implemented in the stack, so events are mostly used for saving state and informative purposes. Application use `gecko_evt_mesh_lc_server_linear_output_updated_id` to update the LED state according to the LC output generated by controller based on the LC properties values, state of the LC and received sensor readings.

Next the Scene Server and Scene Setup Server are initialized. The Scenes models behaviors are implemented in the stack, so application receives only informative events. The scene recall automatically recall the LC states and generate the `gecko_evt_mesh_generic_server_state_recall_id` events for other models which state is stored with scene.

Next the Time and Scheduler Server models are initialized. The behavior of these models is also implemented in the stack, so the application receives only informative events.

The light node registers callback functions for each of the models that it supports. This is done in the function `init_models()` as shown for three models in the following image.

```
static void init_models(void)
{
  mesh_lib_generic_server_register_handler(MESH_GENERIC_ON_OFF_SERVER_MODEL_ID,
                                           0,
                                           onoff_request,
                                           onoff_change,
                                           onoff_recall);
  mesh_lib_generic_server_register_handler(MESH_GENERIC_POWER_ON_OFF_SETUP_SERVER_MODEL_ID,
                                           0,
                                           power_onoff_request,
                                           power_onoff_change,
                                           NULL);
  mesh_lib_generic_server_register_handler(MESH_GENERIC_TRANSITION_TIME_SERVER_MODEL_ID,
                                           0,
                                           transtime_request,
                                           transtime_change,
                                           NULL);
```

The first parameter passed to function `mesh_lib_generic_server_register_handler()` is the model ID. The light node registers handlers for eleven models:

- Generic OnOff Server
- Generic PowerOnOff Server
- Generic Default Transition Time Server
- Light Lightness Server
- Light Lightness Setup Server
- Generic Level Server (on primary element)
- Light CTL Server
- Light CTL Setup Server
- Light CTL Temperature Server
- Generic Level Server (on secondary element)
- Generic OnOff Server (on secondary element)

On the server side, the mesh library works as follows. When any generic request from a client is received the event `gecko_evt_mesh_generic_server_client_request_id` is raised. The application then calls function `mesh_lib_generic_server_event_handler` from the mesh library and passes the event as parameter. The mesh library decodes the model ID from the event and invokes the callback function that has been registered for that model.

For example, in the light node a Generic OnOff request will invoke the callback function `onoff_request()`

The `onoff_request()` function is called whenever an on/off request is received either from one of the switch nodes or from the smartphone app. This is the piece of code in the light node that turns lights on and off.

If the request does not specify any transition time or delay then the light state is changed immediately. Alternatively, the client may have requested a delay and/or a transition time, meaning that the transition does not happen instantly. In such case, the light node application starts a soft timer with the given delay. The light state is not changed until the soft timer expires.

Light Lightness requests are handled in function `lightness_request()`. The lightness request includes a parameter of type uint16 that indicates the light brightness on a scale of 0. - 65535. The example code uses pulse-width modulation (PWM) to drive the LEDs. The PWM is implemented using a 16-bit timer and the requested brightness value is directly mapped to the value of the Compare/Capture register of the timer. For example, value 32768 will result in 32768/65536 ~ 50% brightness / PWM duty cycle. The duty cycle of the PWM signal is displayed on the LCD so that it is easy to compare the brightness that has been requested and the brightness that is currently set in the light node.

The Generic OnOff state is bound with the Light Lightness state. This means that, if we turn off the light using an on/off request, the last value of brightness is saved by the application and is recovered after the application receives an on/off request that turns on the light. If brightness is set to 0 using lightness request, the generic on/off state is set to OFF. If brightness is set to a positive value, the generic on/off state is set to ON.

Brightness could be also changed using Generic Level requests handled in function `pri_level_request()`. The generic level request includes a parameter of type int16 that indicates brightness level. The conversion from level to lightness is made by adding 32768 to the level value.

Light CTL requests are handled in function `ctl_request()`. The ctl request includes three parameters that indicate the light brightness, color temperature, and delta UV. The first two parameters are of type uint16, and the third is of type int16. Actual color temperature and delta UV are displayed on WSTK LCD below the lightness. Color temperature is limited by spec to scale 800.- 20000 K. Limits could be changed by `ctl_setup_request()` with type of request set to *ctl_temperature_range*. Also the default values for ctl state could be changed using `ctl_setup_request()` with type of request set to *ctl_default*.

2.3.4 HSL Light Node Example

This section describes the basic operation of the **BT Mesh – HSL Light Example.** This example was introduced in Bluetooth Mesh SDK 1.7.2 to demonstrate new HSL model capabilities. It is similar to the Light Example, so it is recommended to read section 2.3.3 Light Node Example first. The only difference is that CTL models are replaced by HSL models.

The main feature in the HSL light node is that the device can change LED color based on Lightness, Hue, and Saturation values that are handled by the HSL model. As the WSTK has only one-color LEDi, the values of these parameters are displayed on the LCD. On Thunderboard Sense 2 the RGB LEDs represent the HSL color. Basic functionality like on/off control or the brightness control work the same way as in the Light Example.

The HSL light node supports the following states:

- Generic OnOff
- Generic Level
- Generic OnPowerUp
- Generic Default Transition Time
- Light Lightness
- Light HSL
- Scenes
- Time
- Schedule
- All Light LC states
- All Light LC Property states

As the HSL model has three different elements, the dcd also must have three elements. The secondary and tertiary elements are used for hue and saturation server models with corresponding generic level models. The state structure for LC models are the same as in the light example. The difference is only in `lightbulb_state` structure.

Struct `lightbulb_state` in the HSL example contains the following fields.

Struct Member Name	Description
onoff_current	Current state of light (ON or OFF)
onoff_target	Target state of light (ON or OFF)
transtime	Default transition time
onpowerup	Light state after power up (possible values OFF/ON/RESTORE)
lightness_current	Current brightness (possible values from 0 to 65535)
lightness_target	Target brightness
lightness_last	Last non-zero brightness
lightness_default	Default brightness
lightness_min	Minimum lightness value
lightness_max	Maximum lightness value
pri_level_current	Current generic level on primary element (possible values from -32768 to 32767)
pri_level_target	Target generic level on primary element
hue_current	Current hue (possible values from 0 to 65535, which are mapped to 0° to 360°)
hue_target	Target hue
hue_default	Default hue
hue_min	Minimum hue
hue_max	Maximum hue
sec_level_current	Current generic level on secondary element (possible values from -32768 to 32767)
sec_level_target	Target generic level on secondary element
saturation_current	Current saturation (possible values from 0 to 65535)
saturation_target	Target saturation
saturation_default	Default saturation
saturation_min	Minimum saturation
saturation_max	Maximum saturation
ter_level_current	Current generic level on tertiary element (possible values from -32768 to 32767)
ter_level_target	Target generic level on tertiary element

The next difference between light and HSL examples is the number of models that use the mesh library. The HSL node registers handlers for thirteen models:

- Generic OnOff Server
- Generic PowerOnOff Server
- Generic Default Transition Time Server
- Light Lightness Server
- Light Lightness Setup Server
- Generic Level Server (on primary element)
- Light HSL Server
- Light HSL Setup Server
- Light HSL Hue Server
- Generic Level Server (on secondary element)
- Generic OnOff Server (on secondary element)
- Light HSL Saturation Server
- Generic Level Server (on tertiary element)

Currently no HSL client example is provided either for a mobile phone or an embedded board, but it is easy to modify the Switch example to control HSL instead of CTL.

3 Network Analyzer

Network Analyzer is a packet capture, decoder software and visualization application and is part of Silicon Labs Simplicity Studio. Network Analyzer has support for Bluetooth LE and Bluetooth mesh packet capture and the latest version of Simplicity Studio also has decoders to decode the Bluetooth LE and mesh traffic.

The EFR32 SoCs have a dedicated Packet Trace Interface (PTI), which outputs all the radio traffic sent and received by a specific EFR32 device, and Network Analyzer is able to capture this traffic. On the EFR32 the PTI functionality can be enabled or disabled at the source-code level so it can be enabled during development, but can then be disabled for production software.

Silicon Labs Wireless Starter Kits (WSTKs) support PTI packet capture either over USB, which is useful for capturing packets from a few WSTKs at a time, or over an Ethernet connection. The Ethernet connection also provides access to PTI, and this functionality enables building and debugging a network of WSTKs and large-scale testing environments for Bluetooth mesh.

The easiest way to start a Network Analyzer session for a specific application is to use ***Profile -> Profile AS -> Network Analyzer Target***

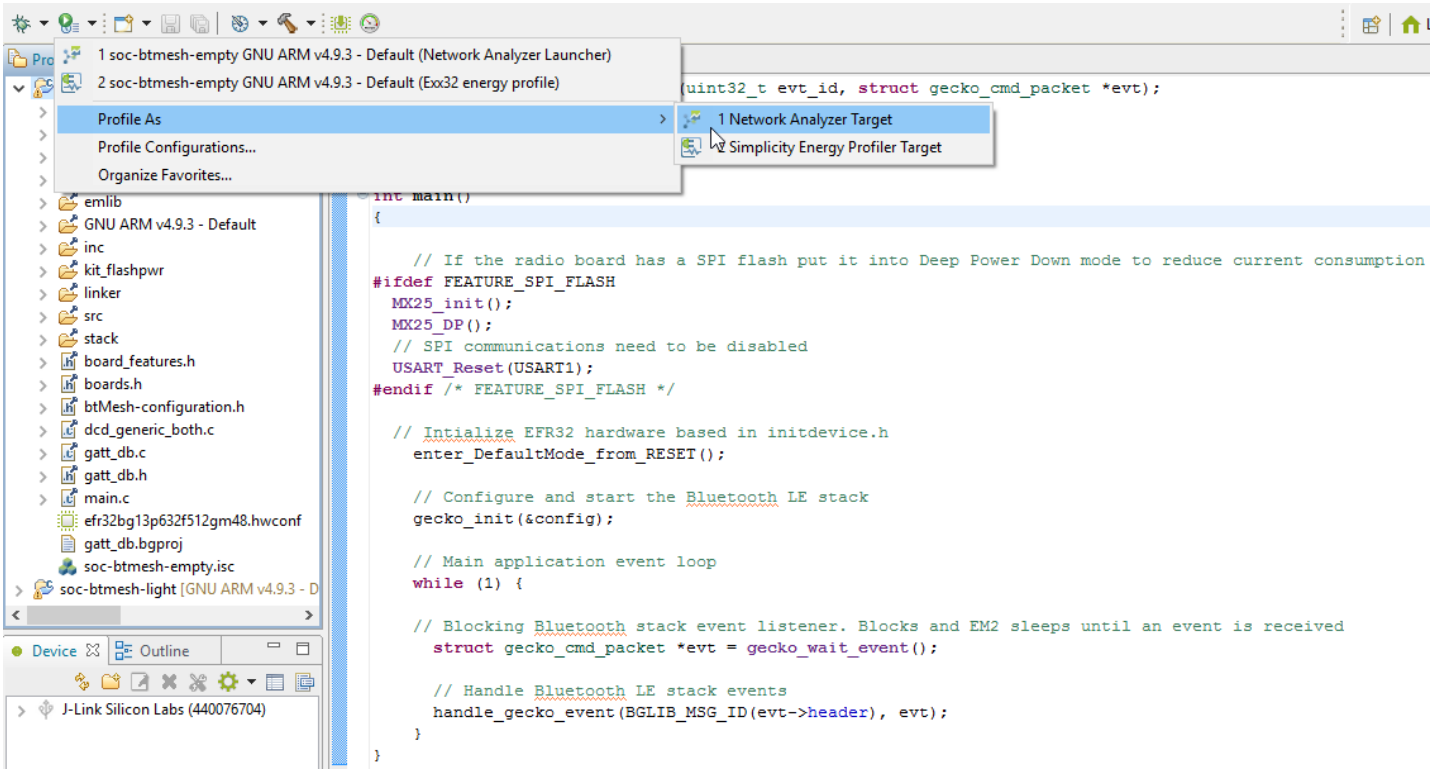


Figure 3-1: Starting Silicon Labs Network Analyzer

4 Bluetooth Mesh Stack and Application for Smartphones

Silicon Labs also provides a Bluetooth mesh stack and a reference application for smartphones. The application can be used to provision mesh-capable Bluetooth devices as nodes that are part of a Bluetooth mesh network, as well as configure the nodes, set up groups, and the publish subscribe settings for nodes. At the time of writing this document the application supports one physical network, multiple groups and Lighting mesh models, but the application will be constantly updated for new features and functionality.

As the smartphones at the time of writing this document do not natively support Bluetooth mesh, Silicon Labs also provides the Bluetooth mesh stack for the phones. The mesh stack is needed for the phone to be able to provision, configure, and control the Bluetooth mesh nodes over the GATT bearer. The figure below illustrates the architecture and the relationship between the Bluetooth stack on the phone operating system and the Silicon Labs Bluetooth mesh stack, as well as how the application relates to this.

The Bluetooth mesh stack will be available as a binary library for phone application developers. A reference application implementing the Bluetooth mesh stack, provisioning, configuration, and device control is available on the Google Play and Apple App Stores.

Contact your local Silicon Labs sales office for more information.

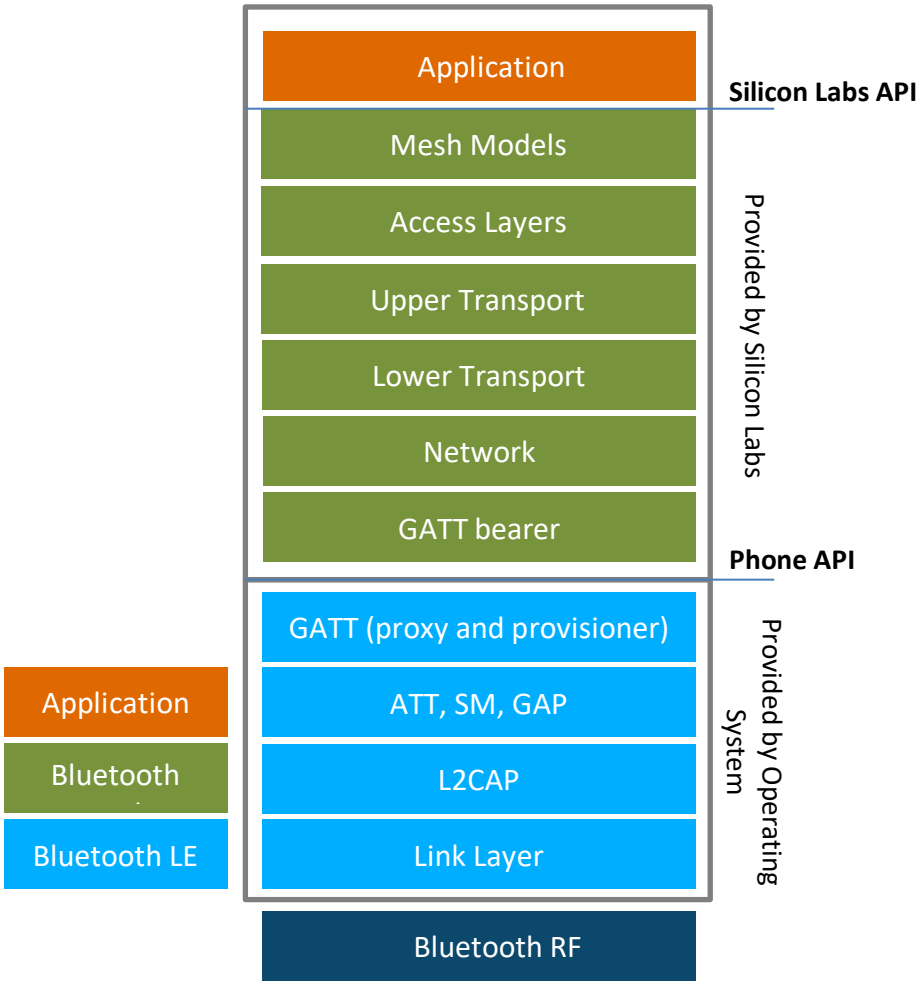


Figure 4-1: Bluetooth Stacks and Application Architecture on Smartphones

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com