Bluetooth Mesh Gateway and Mobile App

# Introduction

This document focuses on explaining the Bluetooth mesh gateway demo, installed as part of the Bluetooth mesh SDK. For the most part, this document focuses on an explanation of the source code flow to solve the requirement of the mock project:

* Gateway: Communicates to end user by UART via command interface. Use can send command to individual control Light by its address. Receive and notify all motion sensors and switches whenever they have an event.
* Mobile app: Use can connect to any Light on a mesh network by PB-GATT and able to control other lights by their unicast address.

The source code can take from: *<https://github.com/tru5tzz/group1-mock.git>*.

## 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 that 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.

## 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.

A diagram of a network

Description automatically generated

The four types of specified node features are as follows:

**Figure 1‑1 Node Types**

**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.

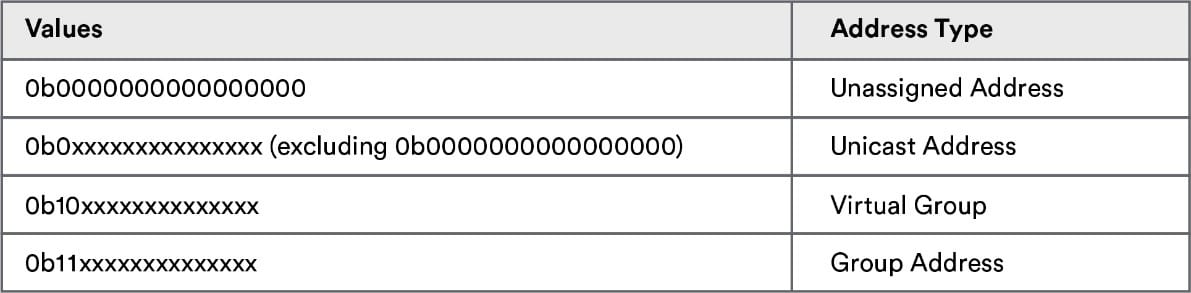
For further information on these features and Bluetooth mesh technology, please go to the Silicon Labs Bluetooth mesh learning center.

## How does Bluetooth mesh communication work?

Messages are exchanged in a Bluetooth mesh network via the publish-subscribe pattern. Publishing is the act of sending a message. Subscribing is a configuration used to allow select messages to be sent to specific addresses for processing. Typically, messages are addressed to group or virtual addresses.

* 1. Addresses

There are four types of addresses; three of these types are used in messaging: unicast, virtual, and group addresses. The fourth is known as an unassigned address. Addresses are 16 bits in length and are encoded as defined below:



* **Unassigned Address** – Unconfigured elements, or elements without designated addresses, have an unassigned address. Given these elements do not have a unique address, they may not be used in messaging.
* **Unicast Address** – During provisioning, a provisioner allocates a unicast address to each element in a node for the lifetime of that node on the network. Unicast addresses may appear in a message's source address field and/or the destination address field. Messages sent to unicast addresses are only processed by one element.
* **Virtual Address** – Virtual addresses are a set of elements associated with a specific Label UUID; these addresses may be published or subscribed to. The Label UUID is a 128-bit value associated with multiple elements that may come from one or more nodes.

For virtual addresses, bits 15 and 14 are set to 1 and 0 respectively (Figure 2); bits 13 – 0 are set to a hash value (providing 16,384 hash values). The hash is derived from the Label UUID. Checking the full 128-bit UUID with a subscribing element is inefficient, especially as the UUID may span more than one message segment. The hash values provide a more efficient way of determining which messages go to which elements.

* **Group Address** – Group addresses are another type of multicast address found in Bluetooth mesh networking. Representing multiple elements from one or more nodes, there are two types of group addresses:
  + *Dynamically assigned* -> 0xC000-0xFEFF
  + *Fixed addresses* – Assigned by Bluetooth SIG and divided into five segments:
    - *Reserved for Future Use (RFU)* –> 0xFF00-0xFFFB
    - *All-proxies* -> 0xFFF: Sent to all nodes with proxy functionality enabled.
    - *All-friends* -> 0xFFFD: Sent to all nodes with friend functionality enabled.
    - *All-relays* -> 0xFFFE: Sent to all nodes with relay functionality enabled.
    - *All-nodes* -> 0xFFFF: Sent to all nodes.

All messages sent to fixed nodes are processed by the primary element of the node.

* 1. Models

Models define a set of States, State Transitions, State Bindings, Messages, and other associated behaviors. An Element within a Node must support one or more models, and it is the model or models that define the functionality that an Element has.

States are data items that indicate the condition of the device, such as on/off or high/low. States may be simple, containing only a single value, or composite, containing multiple fields

**Model categories**

This client-server architecture gives rise to three types of models:

* Server Model
  + Composed of one or more states spanning over one or more elements.
  + Defines messages the model may transmit/receive and defines the element’s behavior given these messages.
  + Examples:
    - On/Off switch – may expose the state of a switch
    - Sensors – may expose the state of the sensor (Potentially a value for temperature or full/refill/empty for sensor measuring)
    - Power level – may expose the state of power (Levels 1-10)
* Client Model
  + Defines the set of messages used by the client to request, change or consume corresponding states of a server.
  + Examples:
    - On/Off switch – Client sends a message of on or off
    - Power level – Messages defining the state of power (0-10)
* Control Model
  + Control Models have multiple functionalities and may contain one or more of the following:
    - Client Model(s)
    - Server Model(s)
    - Control logic (rules and behaviors) coordinating interactions between models it connects with.
  + Example: We may use a Control Model for a pump that circulates liquid coolant around a machine.
    - Scenario – A temperature sensor records the temperature of a machine. If the machine gets above a defined temperature, the cooling pump turns on.
    - Control Module for cooling pump
      * A client to the temperature sensor (accepts temperature value)
      * A server to the On/Off switch (turns pump on or off)
      * Control logic (rules and behaviors) – defines that if a temperature sensor is over a defined value, turn the pump on.

**Model communication and behaviors**

Models talk to each other by sending and receiving messages

Messages either communicate a state value to other devices or change a state value, eliciting a response, often visible, from a device

Models can have specified dependencies on other models. A model may extend another model, a process whereby the first model adds states to the second model. A model may also require that a model which extends it be present. Models that do not extend other models are known as *root models*.

**Foundation Models: Configuration Server model**

The Configuration Server is a root model (i.e., it does not extend any other models). This model is used to represent a mesh network configuration of a device.

The model shall be supported by a primary element and shall not be supported by any secondary elements. The application-layer security on the Configuration Server model shall use the device key established during provisioning.

The model defines the state instances as shown in table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Instance | Model | State | Instance |
| Secure Network Beacon | Primary | - | - | - |
| Composition Data | Primary | - | - | - |
| Default TTL | Primary | - | - | - |
| GATT Proxy | Primary | Configuration Server | Node Identity | Primary |
| Friend | Primary | - | - | - |
| Relay | Primary | - | - | - |
| Model Publication | Primary | - | - | - |
| Subscription List | Primary | - | - | - |
| NetKey List | Primary | - | - | - |
| AppKey List | Primary | - | - | - |
| Model to AppKey List | Primary | - | - | - |
| Node Identity | Primary | - | - | - |
| Key Refresh Phase | Primary | - | - | - |
| Heartbeat Publish | Primary | - | - | - |
| Heartbeat Subscription | Primary | - | - | - |
| Network Transmit | Primary | - | - | - |
| Relay Retransmit | Primary | - | - | - |

In the table above, there are some important state in bold. Those are the states we should go into detail first.

* Composition Data state: The Composition Data state contains information about a node, the elements it includes, and thesupported models. The Composition Data is composed of a number of pages of information. Composition Data Page 0 is mandatory. All other pages are optional
* Model Publication: The Model Publication state is a composite state that controls parameters of messages that are published by a model. The state includes a Publish Address, a Publish Period, a Publish AppKey Index, a Publish Friendship Credential Flag, a Publish TTL, a Publish Retransmission Count, and a Publish Retransmit Interval Steps. Within an element, each model has a separate instance of Model Publication state. It is highly recommended that models defined by higher layer specifications use instances of the Model Publication state to control the publishing of messages
* Subscription List: The Subscription List state is a list of group addresses and Label UUIDs. Within an element, each model has a separate instance of a Subscription List, unless the model extends another model on that element. Instances of models that extend other models (i.e., all models within an extension relation tree) shall share a single instance of a Subscription List per element.
* NetKey List: The NetKey List state is an indexed list of NetKeys. Each entry in the NetKey List holds up to two key values: the old key value and the new key value. The use of the old key and the new key values is described in the Key Refresh procedure. The NetKey List shall contain a minimum of one NetKey.
* AppKey List: The AppKey List state is an indexed list of AppKeys. Each entry in the AppKey List holds an AppKey Index and up to two key values: the old key value and the new key value. The use of the old key and the new key values is described in the Key Refresh procedure
* Model to AppKey List: The Model to AppKey List state is a list of relationships between models and AppKeys. A model may be associated with one or more AppKeys.
* Composition Data State: The Composition Data state contains information about a node, the elements it includes, and the supported models. The Composition Data is composed of a number of pages of information. Composition Data Page 0 is mandatory. All other pages are optional. All Composition Data Pages not defined in this specification are reserved for future use. The size of the state shall not exceed the maximum useful access payload size.

**Composition Data Page 0**

The format of the Composition Data Page 0 is defined in the table below.

|  |  |  |
| --- | --- | --- |
| Field | Size (octets) | Notes |
| CID | 2 | Contains a 16-bit company identifier assigned by the Bluetooth SIG (the list is available at [6]) |
| PID | 2 | Contains a 16-bit vendor-assigned product identifier |
| VID | 2 | Contains a 16-bit vendor-assigned product version identifier |
| CRPL | 2 | Contains a 16-bit value representing the minimum number of replay protection list entries in a device |
| Features | 2 | Contains a bit field indicating the device features |
| Elements | variable | Contains a sequence of element descriptions |

The format of the Features field is defined in the table.

|  |  |  |
| --- | --- | --- |
| Bit | Feature | Notes |
| 0 | Relay | Relay feature support: 0 = False, 1 = True |
| 1 | Proxy | Proxy feature support: 0 = False, 1 = True |
| 2 | Friend | Friend feature support: 0 = False, 1 = True |
| 3 | Low Power | Low Power feature support: 0 = False, 1 = True |

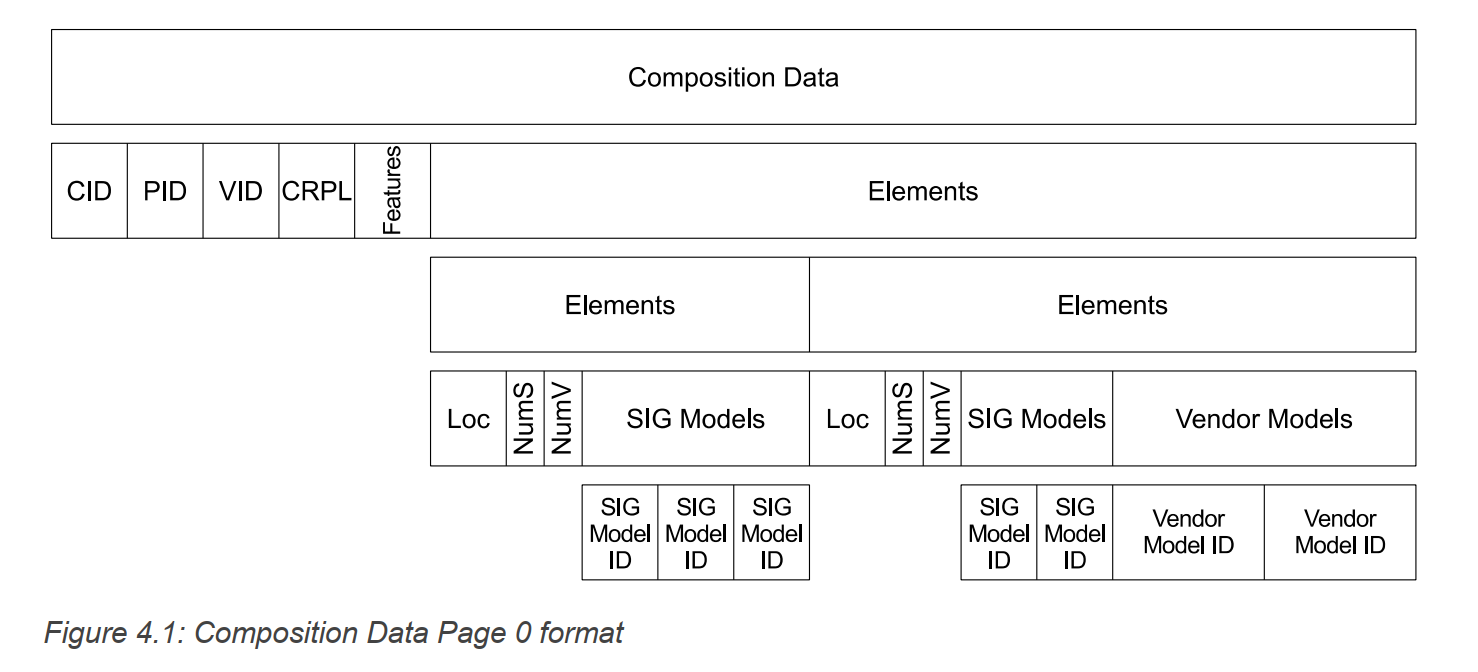
The Elements field contains a sequence of one or more element descriptions. The format of each element description is defined in the table below:

|  |  |  |
| --- | --- | --- |
| Field | Size (octets) | Notes |
| Loc | 2 | Contains a location descriptor |
| NumS | 1 | Contains a count of SIG Model IDs in this element |
| NumV | 1 | Contains a count of Vendor Model IDs in this element |
| SIG Models | variable | Contains a sequence of NumS SIG Model IDs |
| Vendor Models | variable | Contains a sequence of NumV Vendor Model IDs |

The Loc field contains a location description as defined in the GATT Bluetooth Namespace Descriptors section of the Bluetooth SIG Assigned Numbers [4]. Values not defined in the GATT Units table are Reserved for Future Use.

The SIG Models field contains a sequence of NumS SIG Model IDs. For each extended model included in this sequence, all models it extends shall also be included.

The Vendor Models field contains a sequence of NumV Vendor Model IDs.



The example in Figure 4.1 shows a Composition Data Page 0 with two elements. Each element includes the location, the number of SIG Model IDs, and the number of Vendor Model IDs. In this example, the first element has three SIG Model IDs and no Vendor Model IDs, and the second element has two SIG Model IDs and two Vendor Model IDs.

* 1. Types of Bluetooth mesh messages

There are three types of messages in Bluetooth mesh, each of which is defined by a unique opcode (operation code):

* **A GET message**: a message to request the state from one or more nodes.
* **A SET message**: a message to change the value of a given state.
* **A STATUS message**: A status message is used in different scenarios:
  + In response to a GET message, containing the state value.
  + In response to an acknowledged SET message.
  + Sent independently of any message to report the element’s status. One example is a message that’s triggered by a timer running on the element sending this message.

Some messages require an acknowledgment message to be sent by the receiver of the original message. An acknowledgment message serves two purposes:

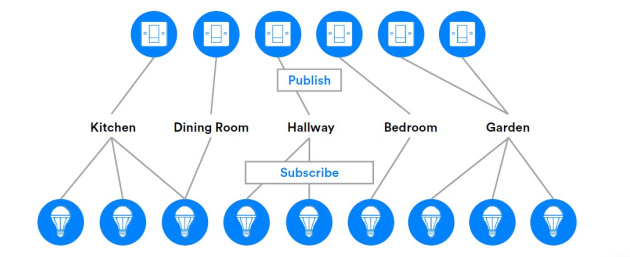
* Confirmation of receipt of the message.
* Return of data related to the message received.

If a response to the message is not received by the sender, or an unexpected response is received, the sender may resend the message. Multiple acknowledged messages received by a node do not affect the behavior (it’s as if the message was received once).

* 1. 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 that 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 or sensor clients subscribe to messages groups, and nodes like switches or sensor servers 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.



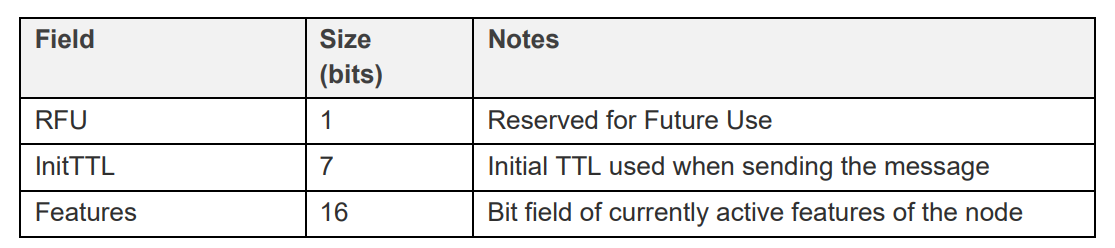
**Figure 1‑2 Publish and Subscribe**

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.

* 1. HEARBEAT MESSAGE

The Heartbeat message is sent by a node to let other nodes determine that a node is still active within a mesh network and how far nodes are apart from each other.

The Heartbeat message parameters:



The Heartbeat message is sent periodically, as configured by the Configuration Server model.

The Heartbeat message can be sent a limited number of times or they can be sent indefinitely.

Heartbeat messages are sent to a configured destination, and it is recommended that a group address is used for sending Heartbeat messages. The messages can also be configured with a specific TTL value.

When Heartbeat messages are received, they are counted. The number of Heartbeat messages received can help determine the reliability of the mesh network for delivering messages from the node sending Heartbeat messages.

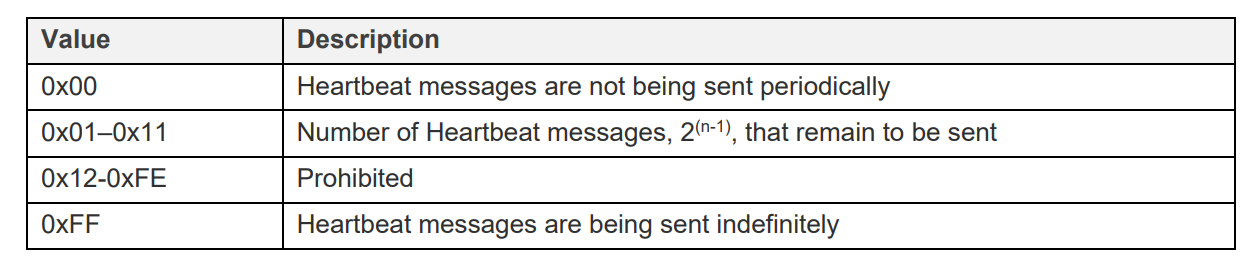
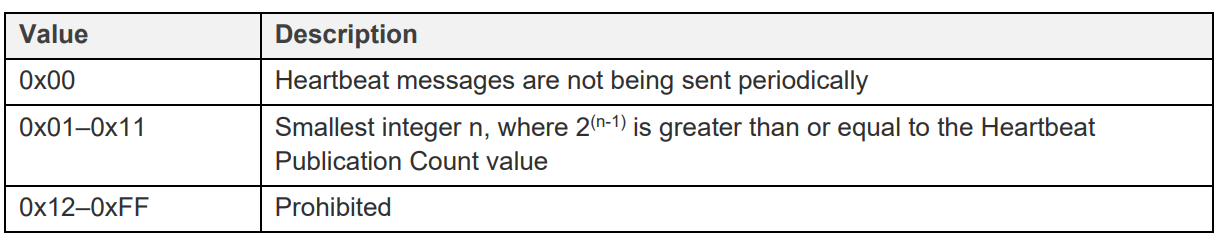
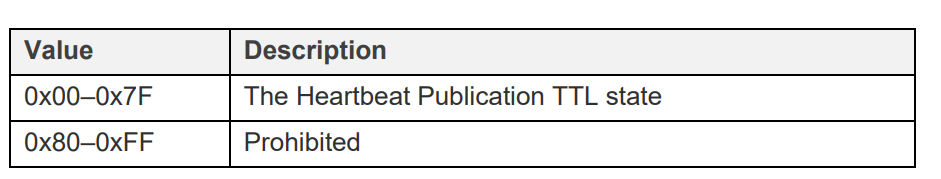
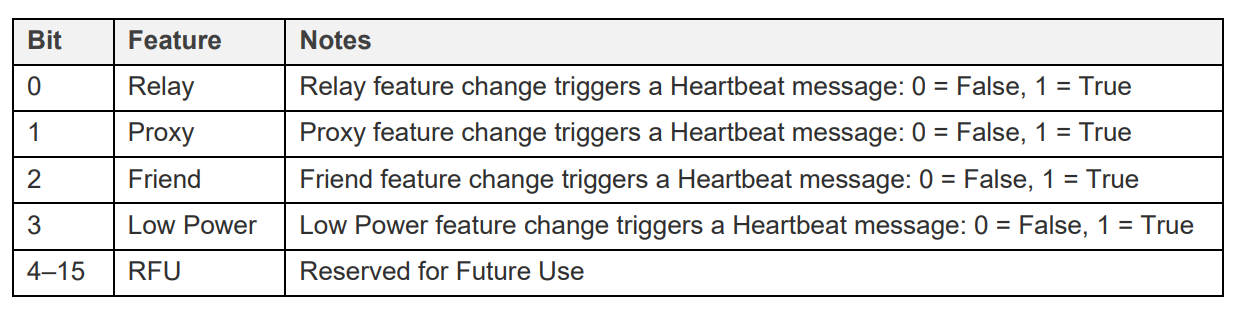
Each Heartbeat message includes the initial TTL value used when sending the original Heartbeat message. This allows a receiving device to determine the number of times this message was retransmitted, known as the number of hops, and a record of the minimum and maximum number of hops can also be used to determine how reliable the mesh network is.

The use of Heartbeat messages can therefore be used to determine the best TTL value to use to address a given node.

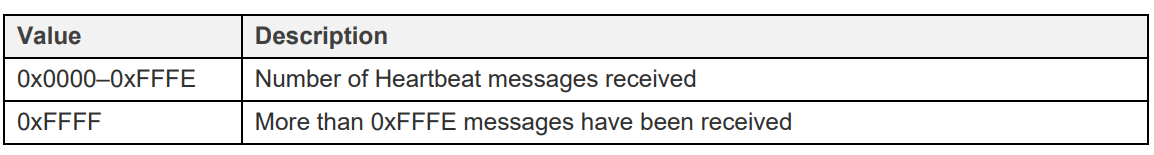
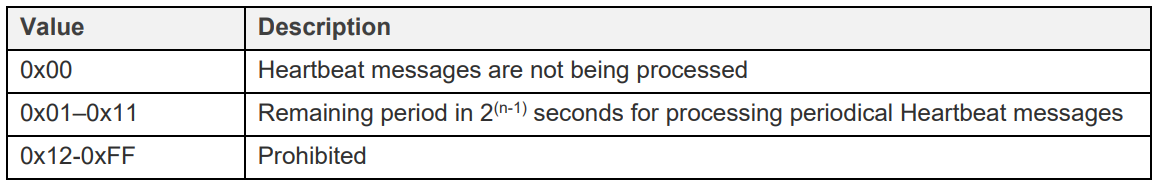
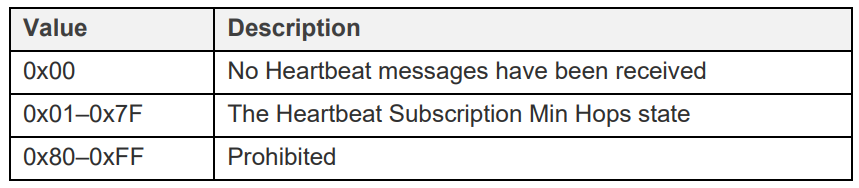
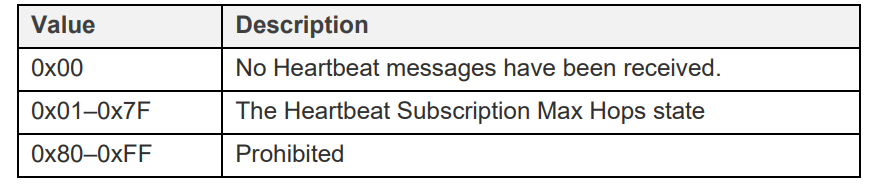
Heartbeat messages also include the features of the node that are currently in use. A node can be configured to send a Heartbeat message when various features are enabled or disabled. This allows the features available on various nodes within a mesh network to be determined.

* + 1. Publishing and Receiving Heartbeat messages
       1. Publishing Heartbeat messages

Publishing of Heartbeat messages is controlled by the Heartbeat Publication state. The Heartbeat Publication state is a composite state that controls sending of periodical Heartbeat transport control messages

* The Heartbeat Publication Destination state: determines the destination address for Heartbeat messages. shall be the unassigned address, a unicast address, or a group address, all other values are Prohibited.
* Heartbeat Publication Count state: is a 16-bit value that controls the number of periodical Heartbeat transport control messages to be sen
* Heartbeat Publication Period Log state: is an 8-bit value that controls the cadence of periodical Heartbeat transport control messages. 
* Heartbeat Publication TTL state: determines the TTL value used when sending Heartbeat messages. 
* The Heartbeat Publication Features state: determines the features that trigger sending Heartbeat messages when changed. 
* Heartbeat Publication NetKey Index state: determines the global NetKey Index of the NetKey used to send Heartbeat messages.
  + - 1. Receiving Heartbeat messages

Receiving of Heartbeat messages is controlled by the Heartbeat Subscription state. The Heartbeat Subscription state is a composite state that controls receiving of periodical Heartbeat transport control messages

* The Heartbeat Subscription Source state: determines the source address for Heartbeat messages a node shall process, shall be the unassigned address or a unicast address, all other values are Prohibited
* Heartbeat Subscription Destination state: determines the destination address for Heartbeat messages. This can be used by nodes to configure a proxy filter to allow them to receive Heartbeat messages
* The Heartbeat Subscription Count state: is a 16-bit counter that controls the number of periodical Heartbeat transport control messages received since receiving the most recent Config Heartbeat Subscription Set message. The counter stops counting at 0xFFFF. 
* The Heartbeat Subscription Period state: is a 16-bit value that controls the period for processing periodical Heartbeat transport control messages. 
* The Heartbeat Subscription Min Hops state: determines the minimum hops value registered when receiving Heartbeat messages since receiving the most recent Config Heartbeat Subscription Set message 
* The Heartbeat Subscription Max Hops state: determines the maximum hops value registered when receiving Heartbeat messages since receiving the most recent Config Heartbeat Subscription Set message. 
  + 1. Configuration
       1. Config Heartbeat Publication Get

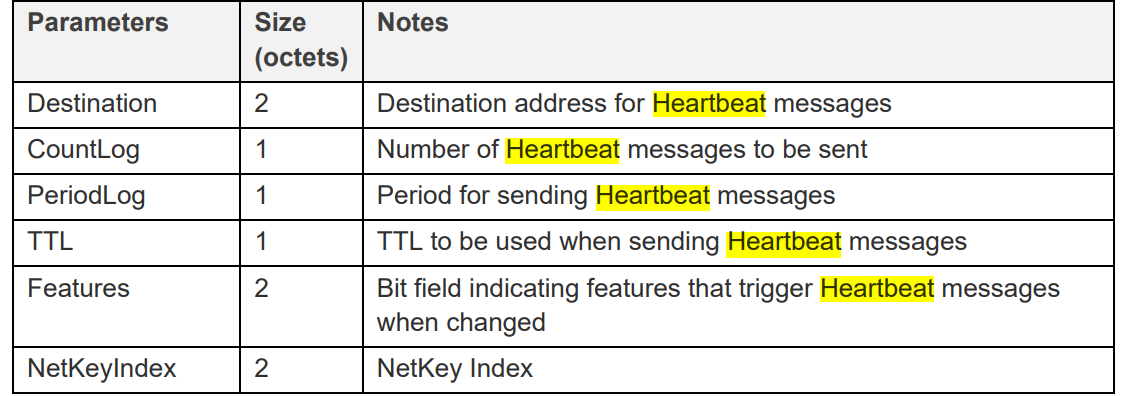
The Config Heartbeat Publication Get is an acknowledged message used to get the current Heartbeat Publication state of an element.

The response to a Config Heartbeat Publication Get message is a Config Heartbeat Publication Status message.

The message has no parameters.

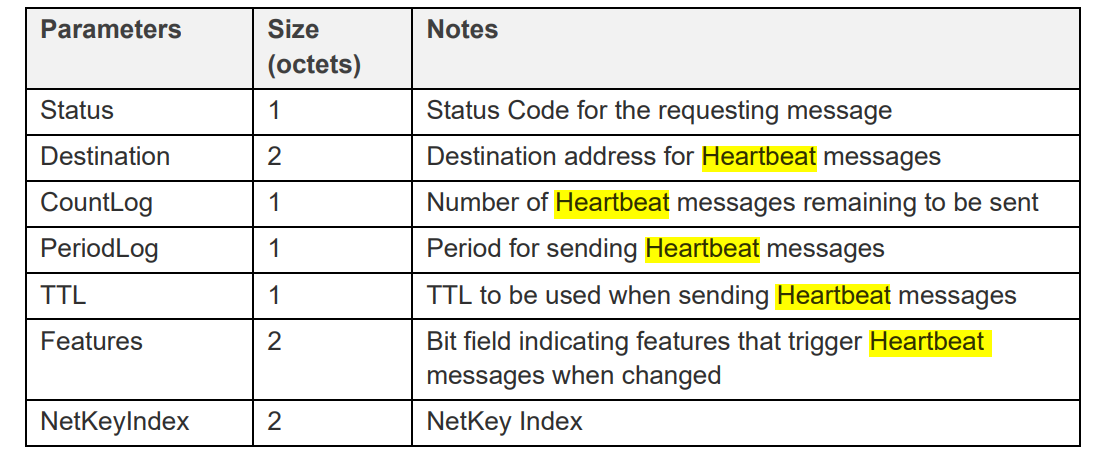
* + - 1. Config Heartbeat Publication Set

The Config Heartbeat Publication Set is an acknowledged message used to set the current Heartbeat Publication state of an element.

The response to a Config Heartbeat Publication Set message is a Config Heartbeat Publication Status message. 

* + - 1. Config Heartbeat Publication Status

The Config Heartbeat Publication Status is an unacknowledged message used to report the Heartbeat Publication state of a node.



* + - 1. Config Heartbeat Subscription Get

The Config Heartbeat Subscription Get is an acknowledged message used to get the current Heartbeat Subscription state of an element.

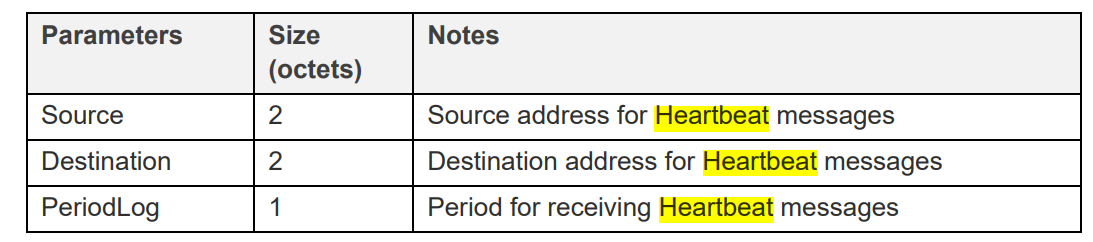
The response to a Config Heartbeat Subscription Get message is a Config Heartbeat Subscription Status message.

The message has no parameters.

* + - 1. Config Heartbeat Subscription Set

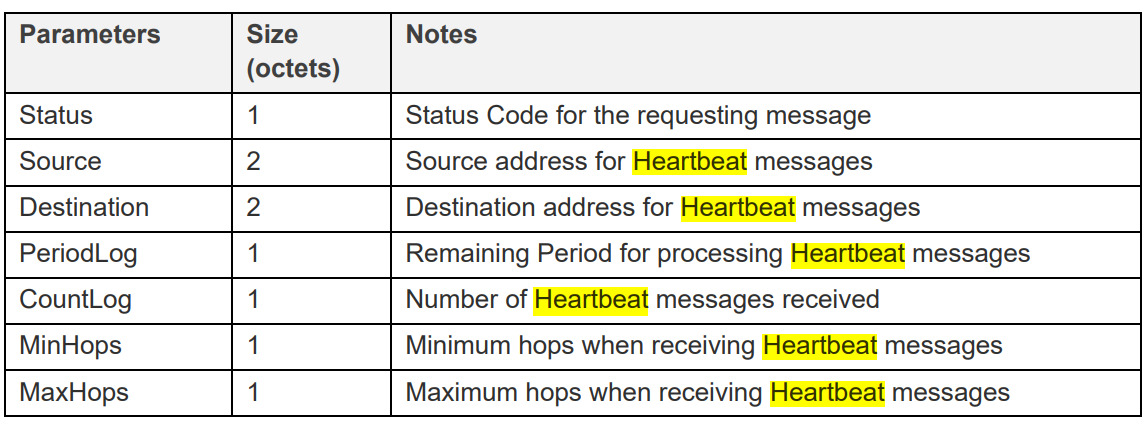
The Config Heartbeat Subscription Set is an acknowledged message used to set the current Heartbeat Subscription state of an element.

The response to a Config Heartbeat Subscription Set message is a Config Heartbeat Subscription Status message.



* + - * 1. Config Heartbeat Subscription Status

The Config Heartbeat Subscription Status is an unacknowledged message used to report the Heartbeat Subscription state of a node.



* + 1. Proceduce

Node have a Configuration Client Model ( such as Provisioner ) will determine Heartbeat Publication/ Subscription state of an other node using all configuration command ( GET/SET ) above, and the other node will response with a message contain its Heartbeat Publication /Subscription Status.

* 1. The Provisioning process

The provisioning process is one of the most important concepts in Bluetooth mesh. It is used for adding devices to the mesh network. A device that gets added to the network is called a node, and the device used to add a node to the network is called the provisioner (usually a tablet, smartphone, or PC).

This process involves five steps:

**Step 1: Beaconing**

Step 1 involves what’s called beaconing, where the unprovisioned device announces its availability to be provisioned by sending the mesh beacon advertisements in the advertisement packets. This is a new type of advertisement data type introduced in the Bluetooth mesh standard.

If an unprovisioned device supports the PB-ADV bearer, it advertises as an unprovisioned device beacon. This involves a specified packet format and is used by the unprovisioned device to allow them to be discovered by a provisioner.

When the PB-GATT bearer is used by the unprovisioned device, a GATT service called the Mesh Provisioning Service supports the overall provisioning procedure and interactions with the provisioner. In the beaconing phase, the unprovisioned device broadcasts advertising packets that include the UUID of the Mesh Provisioning Service. It is discovered by the provisioner through the standard Bluetooth Low Energy scan procedure.

**Step 2: Invitation**

When the provisioner discovers the unprovisioned device via the beacons that were sent, it sends an invitation to this unprovisioned device. This uses a new type of PDU introduced in Bluetooth mesh called the provisioning invite PDU.

The unprovisioned device then responds with information about its capabilities in a provisioning capabilities PDU, which includes:

* The number of elements the device supports.
* The set of security algorithms supported.
* The availability of its public key using an Out-of-Band (OOB) technology.
* The ability of this device to output a value to the user.
* The ability of this device to allow a value to be input by the user.

**Step 3: Public key exchange**

Security in Bluetooth mesh involves the use of a combination of symmetric and asymmetric keys, such as the Elliptic-curve Diffie-Hellman (ECDH) algorithm. In ECDH, public keys are exchanged between the provisioner and the device to be provisioned. This is done either directly over BLE or via an out-of-band (OOB) channel.

**Step 4: Authentication**

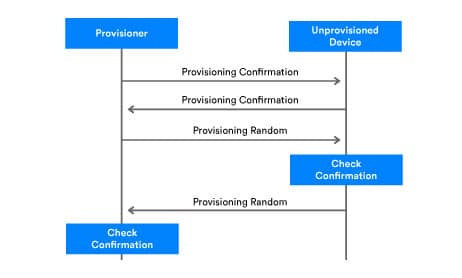
The next step is to authenticate the unprovisioned device. This usually requires an action by the user to interact with both the provisioner and the unprovisioned device. The authentication method depends on the capabilities of both devices used.

In one case, called the output OOB, the unprovisioned device could output a random single- or multiple-digit number to the user in some form, such as blinking an LED several times. That number is then input into the provisioning device via some input method. Other cases include an input OOB, where the number is generated by the provisioner and entered into the unprovisioned device, a static OOB, or no OOB at all.

Regardless of the authentication method used, the authentication also includes a confirmation value generation step and a confirmation check step.

**Confirmation Value Check**

When confirmation values are ready, they are exchanged by the two devices and each checks the integrity of the received value.



The confirmation process starts with the provisioner sending its random number, RandomProvisioner, to the unprovisioned device. The unprovisioned device uses it to recalculate the confirmation value and verify it by comparing it with the confirmation value it previously received:

* If the confirmation value calculated by the unprovisioned device does not match the received ConfirmationProvisioner, then the provisioning process will be aborted.
* If the confirmation value calculated by the unprovisioned device matches the received ConfirmationProvisioner, then the unprovisioned device sends its RandomDevice value to the provisioner.

The provisioner then uses the same process to recalculate the confirmation value and verify by comparing the calculated value with the previously received value.

**Step 5: Provisioning data distribution**

After authentication is complete, each device derives a session key using its private key, and the public key is sent to it from the other device. The session key is then used to secure the connection for the exchange of additional provisioning data, including the network key (NetKey), a device key, a security parameter known as the IV index, and a unicast address which is assigned to the provisioned device by the provisioner. After this step, the unprovisioned device becomes known as a node.

|  |  |  |
| --- | --- | --- |
| Field | Size(octets) | Notes |
| Network Key |  | NetKey for short. NetKey secures communication at the network layer and is shared across all nodes in the network. Possession of a given NetKey is what defines membership of a given Bluetooth mesh network or subnet. Equipping a device with a network’s NetKey is one of the primary outcomes of the provisioning process. The provisioner creates the NetKey when provisioning the first device to be added to the network. |
| Device Key |  | A unique security key possessed only by the provisioner and the device being provisioned. |
| Key Index |  | NetKeys are too long to be transported in single segment messages. To make messaging as efficient as possible, keys are allocated a globally unique 12-bit index value known as the key index, which acts as a short identifier for the key. Messages include key index values which may be referenced against key lists maintained by Configuration Clients. |
| Flags |  | Flag bitmask – indicates the status of the associated key. |
| [IV Index](https://www.notion.so/Bluetooth-Mesh-Technical-i-C-ng-e1283f29ddbd496fa4a5e2cfeffbe487?pvs=21) |  | The IV (Initialization Vector) Index is a 32-bit value which is shared by all nodes in a network. Its purpose is to provide entropy (randomness) in the calculation of message nonce values. |
| Unicast Address |  | Unicast address of the primary element of the new node. |

# Project requirements & Design

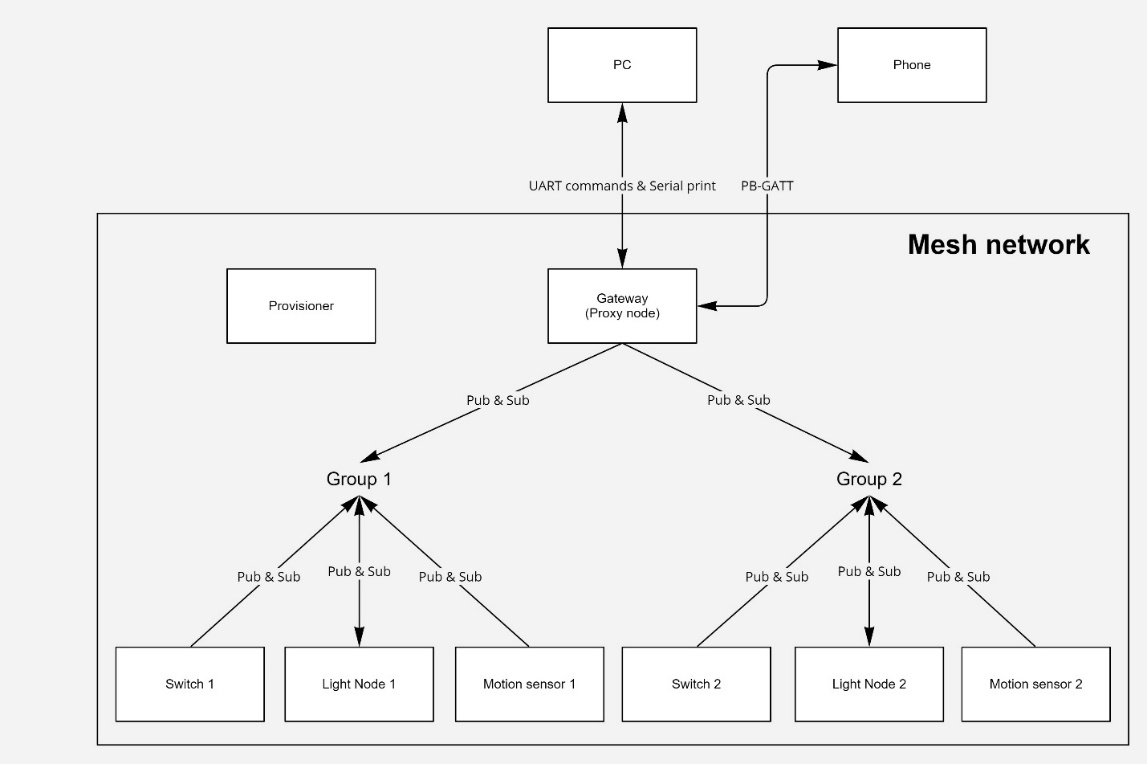
1. System Requirements

Build a Bluetooth Mesh network with components: provisioner, light, motion sensors, switch, gateway

* Provisioner: Responsible for managing and grouping devices in the mesh network
  + Ability to group unprovisioned devices in the network into two distinct groups (using 2 buttons).
  + Status message via LED:
    - During provisioning, the LED blinks slowly.
    - If the provision is successful, the LED light stays on for 3 seconds
    - If an error occurs, the LED blinks rapidly for 3 seconds
* Light: Controls one or more light elements
  + Use generic state on/off to control lights
  + Support CTL (Color Turn-able Light) light element and another light element
  + Send a heart bit signal to the gateway in range
* Motion: Motion detection sensor
  + When motion is detected (using a button), send a light-on request to every light element in the group, the light stays on for a period pre-set by the user.
  + Always active at EM2
* Switch: Manually switch the light on/off
  + Change the state of all light elements in the group
  + Always active at EM2
* Gateway: Announce events that have just happened in the mesh network and send commands
  + Transmit and receive data over UART
  + Enable command passing over UART connection to change the state of light elements according to their unicast address
  + Acting as an intermediary device to receive events from the switch and motion sensor and then print the message on the computer screen through the UART connection.
* Mobile App
  + User can control each light element via unicast address via PB-GATT. communication

User can perform Secure OTA for all 5 types of devices mentioned above: provisioner, light, motion, switch, gateway.

1. System Architecture



Grouping is done through two separate group addresses. Devices in each group will only send and receive messages to and from the group address to which it subscribes or publishes.

To simplify the installation process, all models available on all three devices Switch, Light, and Motion Sensor will all be set to the same subscribe and publish address as the group address it is in. This will cause the device to send or receive unnecessary messages in exchange for simplicity in the setup process.

Gateway, because it's not in a fixed group, will work a little differently. Instead of just publishing or subscribing to a single group address, the models present on the gateway will publish and subscribe to both group addresses 1 and 2. Thus, it will receive all messages sent by nodes in both groups.

In addition, the gateway also has a feature that allows to control each Light via its Unicast address, control commands can be sent from two sources. The first source is the phone. The phone will communicate with the gateway by writing and reading to the GATT characteristic.

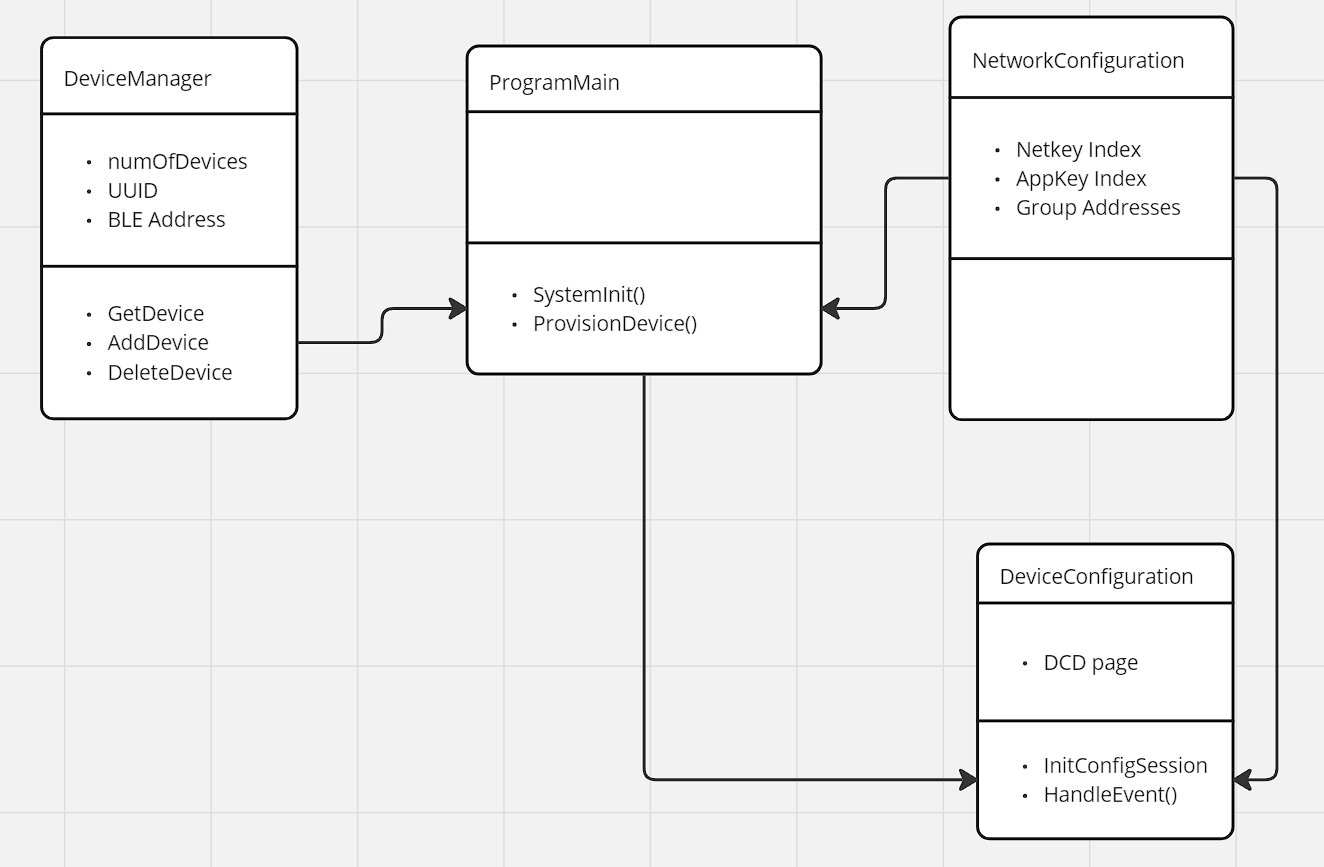
The second command source is from the computer, via the serial line.

# Provisioner

1. System Design

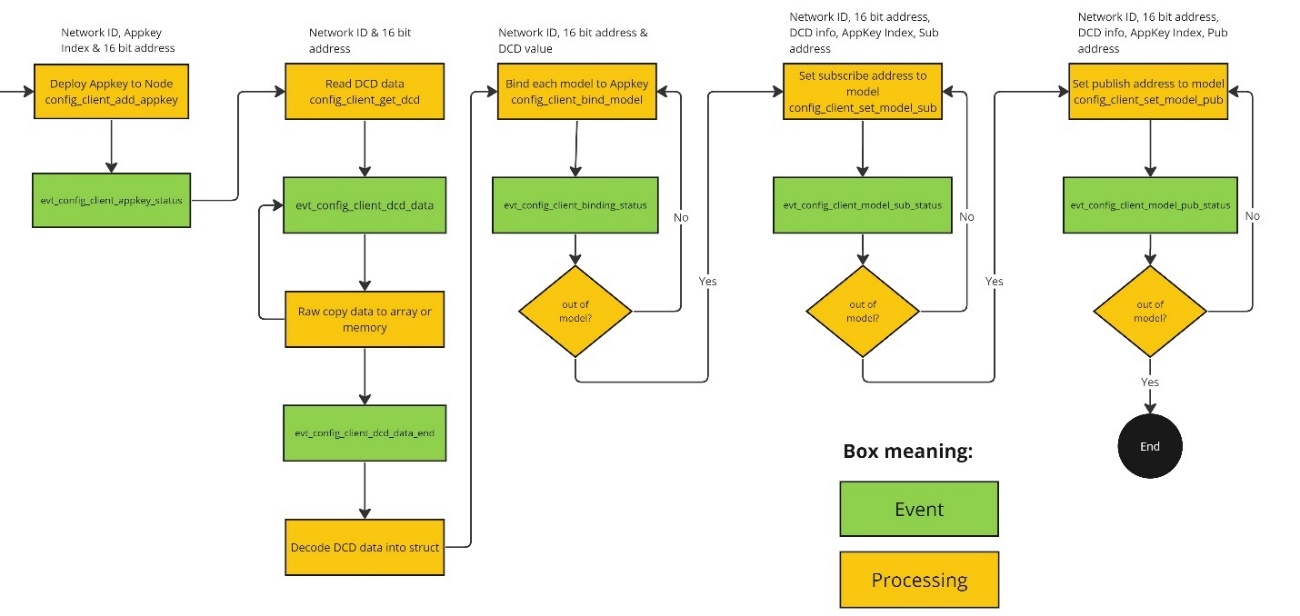
In each Bluetooth Mesh network, the Provisioner acts as a device to manage the network, and the devices in that network. As can be seen, the tasks of the provisioner in this project include:

* 1. Network Initialization
  2. Add nodes to the network
  3. Set the model of each node to match the proposed architecture

In order to simplify the code and increase the ease of editing, the provisioner program has been divided into four modules as shown in the figure:

1. DeviceManager: This is a module that helps manage devices that are sending provisioning beacons. Devices that are transmitting beacons will be saved as a list for later consideration for addition to the network.
2. ProgramMain: This is the program body, which initializes the device, and provides the device provisioning feature to the devices available in the list saved by DeviceManager.
3. NetworkConfiguration: This is the smallest module, containing only a few general settings of the mesh network so that it can be quickly looked up by other modules.
4. DeviceConfiguration: The module assumes the role of setting up each model of each node according to the settings included in NetworkConfiguration.

In addition, the Bluetooth library provided by Silicon Labs is designed with event handling in mind. This can help to chain the entire provisioning process and set up the models present on the node.



However, when processing in this sequential fashion, generating an error at a certain step may result in the device not being set up correctly, so a mechanism is needed to reset the device when the process is interrupted. There are many ways to fix errors and prevent errors, one strategy can be mentioned as redo the error step up to a fixed number of times, if the error continues, remove the device from the network, and put it through the whole provisioning process.

Not only that, once the complexity of the program is larger, the interwoven processing threads will lead to software confusion, and especially limited in modification, upgrade, or expansion.

Indeed, throughout the development process, the modular design failed to fulfill its mission for the whole program.

1. Code Walkthrough

When the device boots, if Button 0 is being pressed down, the system will activate the factory reset feature to erase all data stored on the flash memory and restart automatically. If no key is pressed, the system will skip the flash erase step and proceed to generate a new network key and application key.

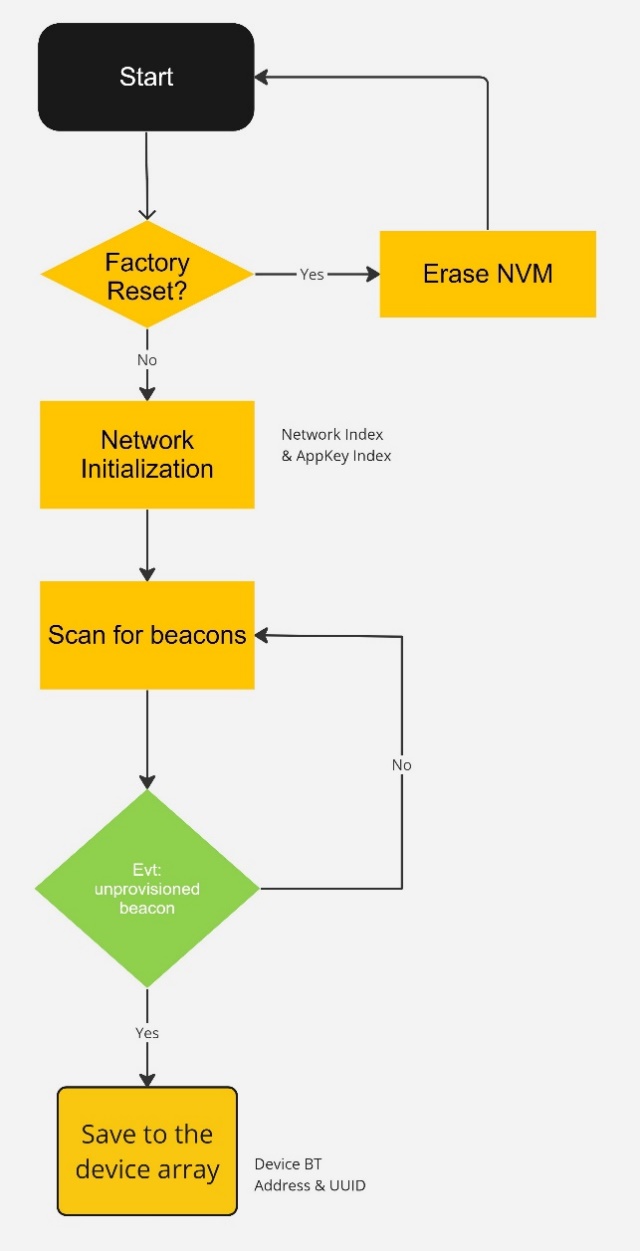
Generating a new network key and application key can cause an error if the key already exists in flash memory, although the mesh is still considered to be initialized successfully. Finally, the system turns on beacon scanning to receive provisioning beacons sent from devices that are not on the network.

Figure 3‑1 Device Bootup sequence

Once a new device's beacon is detected, the device's information is saved into an array pending provisioning.

You can the the whole process in the picture on the right.

To activate the provisioning process, users only need to press the 1 or 0 button to select the group for the device after entering the network.

The software will perform the provisioning procedure, if successful, the program will assign the application key to the device that has been successfully provided.

The last step is to set up the model of node. Each model needs to be associated with an application key and must be published or subscribed to at least one address to be considered active.

With devices with many elements and models, the setup will be repeated so that the all the models have the desired setup.

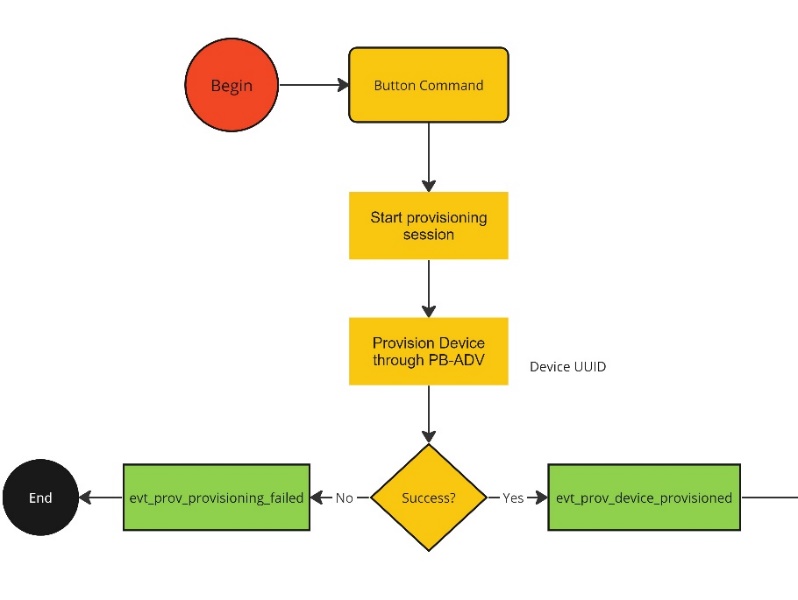


Figure 3‑2 Device received button command

1. Results

Recall the system requirements placed on the provisioner device including:

|  |  |
| --- | --- |
| **Requirement** | **Completion level** |
| Divide devices in the network into 2 groups | Finished |
| Status message via LED | Finished |

It can be said that all requirements have been fulfilled. However, considering the ease of use, fault tolerance or flexibility of the system, there is still a lot of room for improvement.

Especially, it is impossible not to mention the structure of the whole system: there must be a new design that is sophisticated enough, strong enough to clearly divide the functions and roles of each part in the system to avoid overlapping, and conflicts with each other.

# Bluetooth Mesh Gateway

## Requirements

*Communicates to end user by UART via command interface. Use can send command to individual control Light by its address.* *Receive and notify all motion sensors and switches whenever they have an event.*

**Simplicity Studio**

* Bluetooth Mesh SDK 2.1.0 or later, distributed through Simplicity Studio 5.
* 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 mock project about sensor models**

* 1 kit BGM220P is used as a gateway with proxy features.
* 1 kit BGM220P is used as light node with proxy features.

Note that EFR32xG22, BGM220P, and BGM220S only support limited Bluetooth mesh features.

See QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide for more information on obtaining required hardware and software, and running the demonstration.

## Starting with code

A screenshot of a computer

Description automatically generatedThis section summarizes how to start the demonstration from example projects. See QSG176: Silicon Labs Bluetooth® Mesh SDK v2.x

**Figure 4‑1 Create example gateway**

Quick-Start Guide for an introduction to configuring and building your own projects, and for a guide to additional resources. Open Simplicity Studio 5 with a compatible SoC wireless kit connected to the computer. Select the part in Debug Adapters view to open the Launcher perspective. Click the Example Projects & Demos tab. Under Technology Type, filter on Bluetooth Mesh. Next to the Bluetooth Mesh - SoC Sensor Switch project, click Create.

Modify project settings, and click Finish to create the project. Select the project .slcp file if it is not already selected, click the Software

Components tab, and expand the Bluetooth Mesh components group to see the installed features.

A screenshot of a computer

Description automatically generatedProject files autogenerate, with progress reflected in the lower right of the Simplicity IDE. Build and flash the project. Connect your other device and repeat for the Bluetooth Mesh-Gateway example.

**Figure 4‑2 Config in software components**

After that, follow *readme.md* file in [*https://github.com/tru5tzz/group1-mock.git*](https://github.com/tru5tzz/group1-mock.git) to change some operation code of the gateway to solve the requirement.

## Code Walkthrough

### 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 can be controlled by the gateway node through UART via the command interface. Gateway can scan all address devices of lights, switches, and sensors in mesh and know the main models of each address. Gateway can receive and notify all motion sensors and switches whenever they have an event.

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 to a Bluetooth mesh network and to 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 a gateway 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, publishes and subscribes settings, application-level security, and mesh models.

After provisioning and configuration, the Bluetooth mesh network is operational, and the gateway can be used to control the light. The gateway’s buttons can be used to scan all the address lights in the mesh.

### Unprovisioned Mode, Provisioning and Configuration

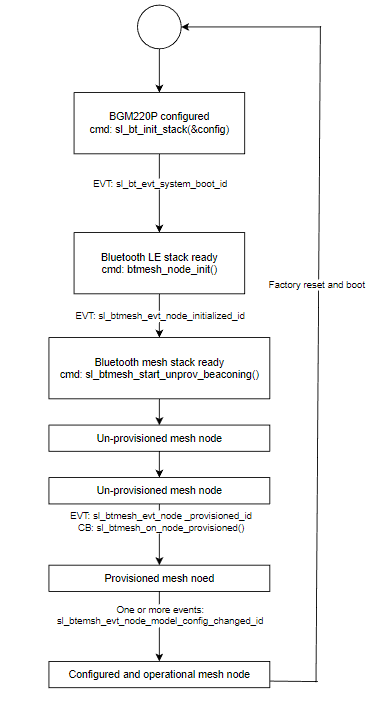
In unprovisioned mode, both gateway and light examples behave the same way. The unprovisioned device simply starts sending unprovisioned beacons and waits for a provisioner to provision and configure it.

After receiving the system\_boot event (*sl\_bt\_evt\_system\_boot\_id*), the application checks if a button is pressed. If yes, it calls the function *sl\_btmesh\_initiate\_full\_reset(),* which halts the system and performs a factory reset by erasing PS storage. The factory reset is also done after receiving a node\_reset event (*sl\_btmesh\_evt\_node\_reset\_id*). If no button is pressed, then the name of the device is set based on the Bluetooth address, and the function *sl\_btmesh\_node\_init()* is called to initialize the Bluetooth mesh node stack.

The event *sl\_btmesh\_evt\_node\_initialized\_id* indicates that the Bluetooth mesh node stack initialization is complete. When this event is raised, the callback function *sl\_btmesh\_on\_provision\_init\_status()* is called to provide 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 *sl\_btmesh\_node\_start\_unprov\_beaconing().*

The API *sl\_btmesh\_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. The start of provisioning is indicated with the event *sl\_btmesh\_evt\_node\_provisioning\_started\_id*. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioning\_started()* is called.

During provisioning, no actions are required from the user application. The Bluetooth mesh stack automatically handles network key configuration and other operations. The c application simply starts blinking the LEDs to indicate that provisioning is in progress. Then they wait for the event *sl\_btmesh\_evt\_node\_provisioned\_id* that indicates provisioning is complete. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioned()* is called.

**Figure 4‑3 Life Cycle of the Application**

The next step after provisioning is the configuration of the node. As explained in *QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide*, the smartphone app is used to configure a node either as a gateway or a light and assign it to a group. The configuration procedure consists of the following steps:

* Provisioner distributes an application key to the node.
* The 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 Bluetooth 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:

* *sl\_btmesh\_evt\_node\_key\_added\_id:* generated when the provisioner has sent a new key (network or application)
* *sl\_btmesh\_evt\_node\_model\_config\_changed\_id:* indicates that the provisioner has modified the configuration of the local model (either publish or subscribe settings changed)

The result after this step shows in the picture below.

**Figure 4‑4 The result of provisioner**

### A screenshot of a computer Description automatically generatedGateway Example

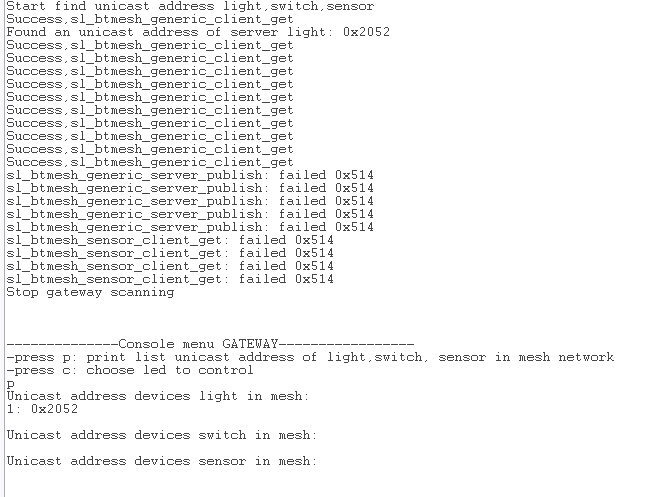
This section describes the basic operation of the code **Gateway**. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app. To solve the requirement, some models in the gateway are required: Generic On/Off Server, Generic On/Off Client, and Sensor Client.

After that, the console shows a commend to require users to push button PB0 to scan the address of the device in a configurable duration time:

* Light node: when PB0 press will run command *sl\_btmesh\_generic\_client\_get.* This command will get the current state of a server model or models in the network. Besides the immediate result code, the response or responses from the network will generate server state report events for the replies received. An *event* *sl\_btmesh\_evt\_generic\_client\_server\_status\_id*  for the response of the light server will capacity address of it.
* Switch node: when button switch node PB0 is pressed will creat event *sl\_btmesh\_evt\_generic\_server\_client\_request\_id.* This contains the address of the switch node and the address group it publishes
* Sensor node: when button sensor node PB0 is pressed will creat event *sl\_btmesh\_evt\_sensor\_client\_status\_id.* This contains the address of the sensor node and the address group it publishes

After time out for scanning, a console menu display to control light through the command line in USART. The character input from the console through the command *getchar().* The console menu is shown in the figure below:

A diagram of a flowchart

Description automatically generatedFor the control light, will choose the address light want to control and the state that wants to be set to light by entering the character from the console menu. After that will send the command *sl\_btmesh\_generic\_client\_set* to change the state of light.

**Figure 4‑5 Console menu**

# Mobile app

## Requirements

*Use can connect to any Light on mesh network by PB-GATT and able to control other lights by their unicast address*

## Mobile app example

Connect with the device gateway through Bluetooth by proxy features enable. In the GATT config, create a service control light. This service has 4 characteristics:

* A screenshot of a phone

  Description automatically generatedNumber Light: Show the number of lights in the mesh
* Set Light: change state light
* Light Connected: address light is connected
* Connect Light: connect the address light in the list shown by the gateway by the number in the list.

A diagram of a light and mobile phone

Description automatically generatedDown below is the sequence diagram of the turning indiviual light on or off operation:

# Bluetooth Mesh Sensor

## Requirements

*The demo motion sensor turn on all light in its group mesh by pushing a button that can configurable time duration of light on, always running in mode EM2*

**Simplicity Studio**

* Bluetooth Mesh SDK 2.1.0 or later, distributed through Simplicity Studio 5.
* 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 mock project about sensor models**

* 1 kit BGM220P is used as a sensor client with proxy features.
* 1 kit BGM220P is used as a sensor server with proxy features.

Note that EFR32xG22, BGM220P, and BGM220S only support limited Bluetooth mesh features.

See QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide for more information on obtaining required hardware and software, and running the demonstration.

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

## 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 sensor client node(s) can be controlled by the sensor server node(s).

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 to a Bluetooth mesh network and to 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 a sensor client and sensor server 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, publishes and subscribes settings, application-level security, and mesh models.

After provisioning and configuration, the Bluetooth mesh network is operational, and the sensor server can be used to control the sensor client. The sensor server’s buttons can be used to control all the sensor clients in a group.

## Starting with code

A screenshot of a computer

Description automatically generatedThis section summarizes how to start the demonstration from example projects. See QSG176: Silicon Labs Bluetooth® Mesh SDK v2.x

Quick-Start Guide for an introduction to configuring and building your own projects, and for a guide to additional resources. Open Simplicity Studio 5 with a compatible SoC wireless kit connected to the computer. Select the part in Debug Adapters view to open the Launcher perspective. Click the Example Projects & Demos tab. Under Technology Type, filter on Bluetooth Mesh. Next to the Bluetooth Mesh - SoC Sensor Client project, click Create.

**Figure 6‑1 Create example sensor model**

Modify project settings, and click Finish to create the project. Select the project .slcp file if it is not already selected, click the Software

Components tab, and expand the Bluetooth Mesh components group to see the installed features.

A screenshot of a computer

Description automatically generatedProject files autogenerate, with progress reflected in the lower right of the Simplicity IDE. Build and flash the project. Connect your other device and repeat for the Bluetooth Mesh - SoC Sensor Server example.

**Figure 6‑2 Config in software components**

After that, follow *readme.md* file in [*https://github.com/tru5tzz/group1-mock.git*](https://github.com/tru5tzz/group1-mock.git) to change some operation code of the sensor server and sensor client to solve the requirement.

## Code Walkthrough

### Unprovisioned Mode, Provisioning and Configuration

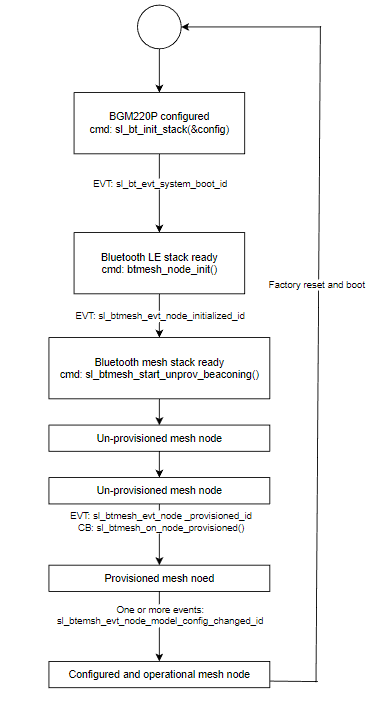
In unprovisioned mode, both sensor client and sensor server examples behave the same way. The unprovisioned device simply starts sending unprovisioned beacons and waits for a provisioner to provision and configure it.

After receiving the system\_boot event (*sl\_bt\_evt\_system\_boot\_id*), the application checks if a button is pressed. If yes, it calls the function *sl\_btmesh\_initiate\_full\_reset(),* which halts the system and performs a factory reset by erasing PS storage. The factory reset is also done after receiving a node\_reset event (*sl\_btmesh\_evt\_node\_reset\_id*). If no button is pressed, then the name of the device is set based on the Bluetooth address, and the function *sl\_btmesh\_node\_init()* is called to initialize the Bluetooth mesh node stack.

The event *sl\_btmesh\_evt\_node\_initialized\_id* indicates that the Bluetooth mesh node stack initialization is complete. When this event is raised, the callback function *sl\_btmesh\_on\_provision\_init\_status()* is called to provide 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 *sl\_btmesh\_node\_start\_unprov\_beaconing().*

The API *sl\_btmesh\_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. The start of provisioning is indicated with the event *sl\_btmesh\_evt\_node\_provisioning\_started\_id*. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioning\_started()* is called.

During provisioning, no actions are required from the user application. The Bluetooth mesh stack automatically handles network key configuration and other operations. The sensor client and the sensor server application simply start blinking the LEDs to indicate that provisioning is in progress. Then they wait for the event *sl\_btmesh\_evt\_node\_provisioned\_id* that indicates provisioning is complete. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioned()* is called.

**Figure 6‑3 Life Cycle of the Application**

The next step after provisioning is the configuration of the node. As explained in *QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide*, the smartphone app is used to configure a node either as a sensor client or a sensor server and assign it to a group. The configuration procedure consists of the following steps:

* Provisioner distributes an application key to the node.
* The 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 Bluetooth 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:

* *sl\_btmesh\_evt\_node\_key\_added\_id:* generated when the provisioner has sent a new key (network or application)
* *sl\_btmesh\_evt\_node\_model\_config\_changed\_id:* indicates that the provisioner has modified the configuration of the local model (either publish or subscribe settings changed)

The result after this step shows in the picture below.

**Figure 6‑4 The result of provisioner**

### A screenshot of a computer Description automatically generatedSensor Server Example

This section describes the basic operation of the code **Sensor Server**. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app.

In this demo, the sensor server supports one type of sensor: a People Count sensor. The People Count sensor is simulated by the button on the started kit: PB0 push will public signal by cmd: *sl\_btmesh\_sensor\_server\_send\_status* and create an event on sensor client *sl\_btmesh\_evt\_sensor\_client\_status\_id* that makes the light turn on.

Upon receiving the *sl\_btmesh\_evt\_node\_initialized\_id* event and call to the corresponding callback *handle\_node\_initialized\_event()*, the sensor server node initializes the sensors by calling *sl\_btmesh\_sensor\_server\_node\_init()*. This occurs in the btmesh\_sensor\_server component (see *sl\_btmesh\_sensor\_server.c*). This function sets the people count to 0. Next, it enables GPIO interrupts for device buttons PB0. Otherwise, the node starts unprovisioned beaconing and waits for a provisioner.

Also when receiving the *sl\_btmesh\_evt\_node\_initialized\_id* event. The Low Power Node (LPN) feature is then initialized and configured. After the LPN feature is initialized, the callback function *sl\_btmesh\_lpn\_on\_init()*. The LPN configuration has five parameters: lpn\_queue\_length, lpn\_poll\_timeout, lpn\_receive\_delay, lpn\_request\_retries, and lpn\_retry\_interval. The settings of these parameters are defined in *sl\_btmesh\_lpn\_config.h* as LPN\_MIN\_QUEUE\_LENGTH, LPN\_POOL\_TIMEOUT, LPN\_RECEIVE\_DELAY, LPN\_REQUEST\_RETRIES, and LPN\_RETRY\_INTERVAL, respectively. After that, the node starts finding a friend node.

The LPN feature is mostly implemented in the mesh stack, so only a few informative events can be raised to the application through corresponding callback functions:

* sl\_btmesh\_lpn\_on\_friendship\_established: A friendship was successfully established.
* sl\_btmesh\_lpn\_on\_friendship\_failed: The Friendship establishment failed, and the node tries to establish a friendship again in 2 seconds.
* sl\_btmesh\_lpn\_on\_friendship\_terminated: The friendship was terminated for some reason, and the node tries to establish a friendship again in 2 seconds.

If a GATT connection is opened, the friendship is terminated and the LPN is de-initialized. In this case, the callback function *sl\_btmesh\_lpn\_on\_deinit()* is called. After all GATT connections are closed, the LPN feature is re-initialized.

Once provisioned and initialized, the sensor server node simply publishing sensor data is handled by *handle\_sensor\_server\_publish\_event()* when the publish period expires or PB0 is pressed. Data from the sensor is published.

### A diagram of a sleep mode Description automatically generatedSensor Client Example

**Figure 6‑5 Flow code sensor server**

This section describes the basic operation of the Sensor Client. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app. The main purpose of the sensor client is to receive publish event from the sensor server to set led turn on and after a configurable time duration led turn off.

Upon receiving the *sl\_btmesh\_evt\_mesh\_node\_initialized\_id* event, the sensor client node performs the following actions:

* Initializes the sensor client model by calling sl\_btmesh\_sensor\_client\_init().
* When PB0 of sensor server push, event *sl\_btmesh\_evt\_sensor\_client\_status\_id* appears. Indicates that a sensor status message has been received. In this event, the led is turned on, and the set timer to call back start *sl\_simple\_timer\_start*  for a configurable duration time turns off led.

Also when receiving the *sl\_btmesh\_evt\_mesh\_node\_initialized\_id* event. The friend functionality is then 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, only a few informative events can be raised to the application through corresponding callback functions:

* sl\_btmesh\_friend\_on\_friendship\_established: a friendship was established.
* A diagram of a computer program

  Description automatically generatedsl\_btmesh\_friend\_on\_friendship\_terminated: the friendship was terminated.

## Bluetooth Mesh Node Power Consumption

**Figure 2‑6 Flow sensor client**

Bluetooth mesh nodes (other than Low Power nodes) are always listening for messages from the other nodes. Because the nodes are always on and receiving, they consume quite a lot of power and are typically connected to mains or respective power sources (for example, coin or button cell batteries might have too short a lifespan for actual usage). You can learn about the radio/power operation and consumption in the datasheets for your part, SoC, or module.

The Mesh Profile Bluetooth® Specification Revision: v1.0.1 defines a Low Power Node, Friend node, and the concept of friendship as follows:

* Low Power Node: The ability to operate within a Bluetooth mesh network at significantly reduced receiver duty cycles. Minimizing the time the radio receiver is on leads to lower power consumption with the node only enabling the receiver when strictly necessary. LPNs achieve this by establishing a friendship with a Friend node.
* Friend Node: The ability to help an LPN operate by storing messages destined for the LPN and only forwarding them to it when the LPN explicitly requests messages from the Friend node.

### Low Power Node Settings

The following subsections describe the LPN settings related to power consumption.

### PollTimout

The PollTimeout setting has the following characteristics:

* Establishes a maximum time that may elapse between two consecutive requests sent by the LPN to its Friend node. If no requests from an LPN are received by the Friend node before the PollTimeout timer expires, the Friend terminates the friendship.
* Duration: 10 to 3,455,999 in 100 ms = 1 sec to 345,600 sec = 5,760 min = 96 hours = 4 days
* The longer the PollTimeout, the more the LPN can sleep. On the other hand, this also means that the LPN receives messages less
* often and it takes more time for other nodes or the Provisioner to communicate with the LPN.
* This is the most important parameter when controlling the LPN power consumption.

### ReceiveDelay

The ReceiveDelay setting has the following characteristics:

* The time which elapses between the LPN sending a request to the Friend node and the Friend node starting to listen for a response.

This allows the Friend node time to prepare its response and send it back.

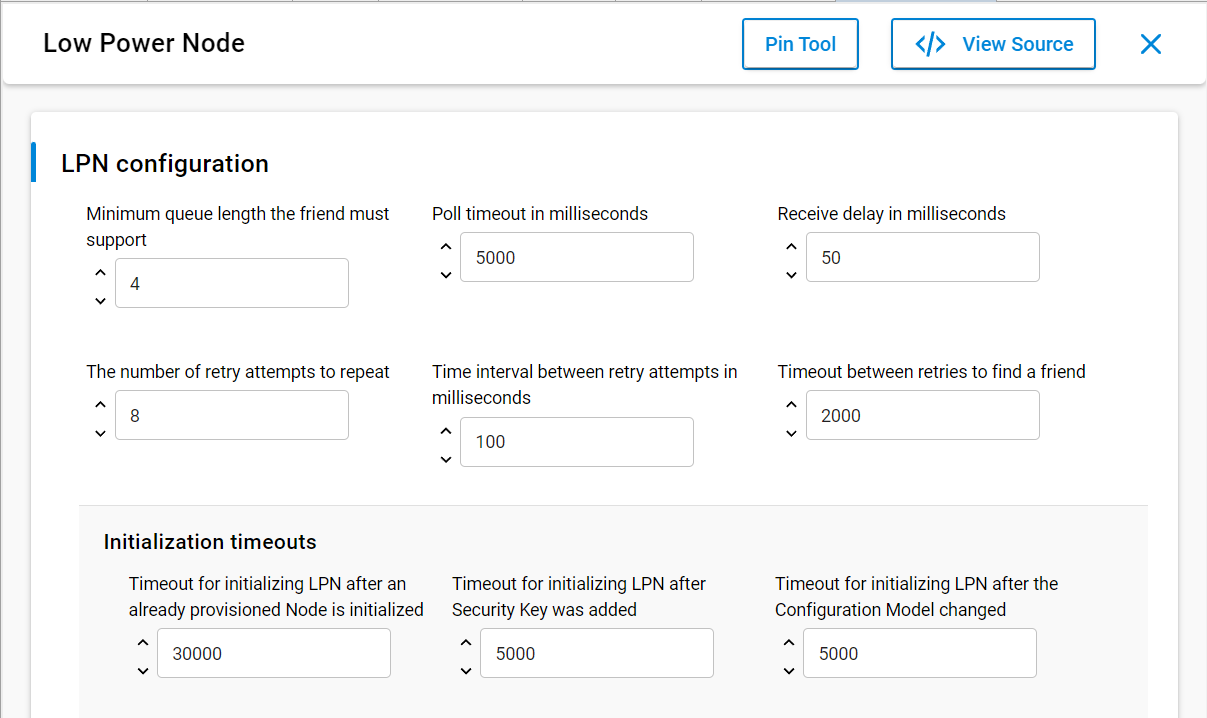
* Duration: 10 to 255 ms
* Requested by the LPN.
* If it is too small, the Friend node might not send a Friend Offer.
* Can be adjusted by the application.
* Does not have much effect on LPN power consumption.

### Power Consumption-Related Guidelines

* Proxy feature uses a lot of power so avoid using this frequently.
* Transmit as few messages as possible as transmission always increases power consumption noticeably.
* Subscribe to as few messages as possible.
* Do not change subscriptions often.
* Target fewer unicasts to the LPN.
* Be sure that the LPN receives a Friend node, and the friendship stays (that is, the signal is sufficient). An LPN that is just on the signal edge could make a friendship, then lose it, and make it again. Going back and forth like this is not very efficient because establishing every friendship consumes more power than being in a friendship.
* Do not restart the LPNs or Friend nodes often as the friendship has to be established again.
* When an LPN is establishing a friendship, do not blindly keep trying in a busy loop. Have a backoff policy if no acceptable Friend
* Offers are received.
* Acknowledged messages and especially segmented acknowledged messages for other nodes might be slow as the acknowledgments return via the Friend node. This also might have an effect on power consumption.
* Do not keep the node in an unprovisioned state because it will remain awake and consume high current, accordingly.

### Setup

If you create a source code example, the Low power node settings are in the Studio Component named Low Power Node (Bluetooth Mesh->Features->Low Power Node) as shown in the following figure.



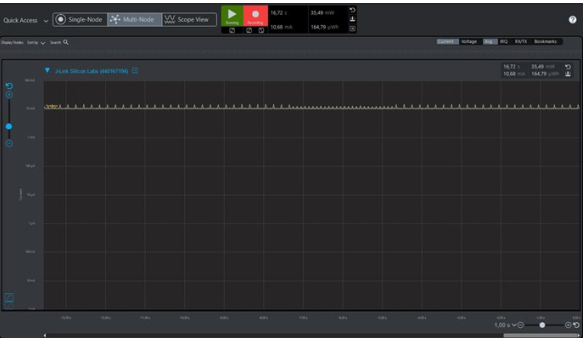
**Figure 3‑1 Low Power Node**

You can also find these additional power consumption-related settings:

* During friendship: PollTimeout, Poll timeout in ms, LPN\_POLL\_TIMEOUT, default 5000 ms
* When finding friendship: Timeout in ms between retries to find a friend, LPN\_FRIEND\_FIND\_TIMEOUT, default 2000 ms

### Measurements

* + - 1. Unprovisioned Low Power Node

The current measurement in unprovisioned looks like the following figure

**Figure 3‑2 Current Profile for EFR32MG12, unprovisioned**

The current measurement is on a steady high level (average 10.68 mA in the example) because the node is running and waiting for provisioning. When using a device family other than the EFR32MG12, the current might be different.

* + - 1. Low Power Node Looking for a Friend Node

After provisioning, remember to return to the mobile application's main level so that the Bluetooth connection with the mobile phone is closed. You should then see a current profile like the following figure. At this point, provision only the Switch and do not change any Device Configuration settings at this time.

A screenshot of a computer

Description automatically generatedOn average the current measurement is now much lower (3.94 mA in this example). The approximately one-second lasting high current part occurs when the LPN is looking for a Friend node by sending Friend Request messages. The lower current part with spikes that take approximately two seconds occurs when the device is sleeping (no Bluetooth mesh communication) and using the Proxy feature (Bluetooth LE device connectivity)

**Figure 3‑3 Current Profile for EFR32MG12, Looking for a Friend Node, with Proxy**

* + - 1. Low Power Node Friendship Established

The provision Sensor client also acts as a Friend node. Now the sensor server LPN establishes a friendship with the sensor client node.

A screenshot of a computer

Description automatically generatedThe power measurements look like the following figure.

**Figure 3‑4 . Current Profile for EFR32MG12, Friendship established, with Proxy**

The current consumption is now only 736 µA on average which is much less than the original over 10 mA measurement. You can change the current consumption by changing the interval advertiser of the proxy, with an interval of 100ms, the current consumption is over 500 µA.

* + - 1. Low Power Node Friendship Established with Proxy Turned Off

When turning off the feature proxy, The current measurement is now only 6.6 µA. There is no communication visible as the poll period (PollTimeout) is so long that it is not detected. A longer—over 120-second measurement—appears in the following figure.

A screenshot of a video editing software

Description automatically generated

# SWITCH NODE

1. Requirements about switch node

* Control all lights in a group of mesh.
* Always run in low power node ( EM2 )

1. Requirements about software and hardware

**Software:**

* Simplicity Studio 5: Bluetooth Mesh SDK 2.1.0 or later, the pre-built demo binaries and source code are included in the SDK.
* Silicon Labs Bluetooth mesh mobile application

**Hardware:**

* 1 kit BGM220P is used as a switch node with proxy features, 1 kit BGM220P is used as a light node with proxy features. ( switch node can control light node )
  + Note that EFR32xG22, BGM220P, and BGM220S only support limited Bluetooth mesh features.

See QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide 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 sensor light node can receive commands from multiple switch nodes.

1. Mesh network implementation

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

* Unprovisioned mode – After the demo firmware is installed, the device starts in unprovisioned mode.
* Provisioning – The devices are provisioned to a Bluetooth mesh network and network security is set up.
* Configuration – The group, publish and subscribe, and application security are configured.
* Operation – The switch node(s) can control by multiple light node in group.

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 to a Bluetooth mesh network and to 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 a switch node 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, publishes and subscribes settings, application-level security, and mesh models.

After provisioning and configuration, the Bluetooth mesh network is operational, and the switch node can be used to control the light nodes.

1. Starting with code

* Open Simplicity Studio 5 with a BGM220P kit connected to computer ( host )
* Open Launcher tab, click on START button, switch to Example Projects & Demos tab.
* Under Technology Type, filter on Bluetooth Mesh. Next to the Bluetooth Mesh - SoC Empty project, click Create.

1. Necessary components

Open file .slcp in project, switch to Software component tab install the below components:

* Application -> Button Press
* Application -> Log
* Application -> Timer
* Application -> Device Information GATT Service
* Bluetooth Mesh -> Features -> Low power node
* Bluetooth Mesh -> Model -> Generic -> Generic Base
* Bluetooth Mesh -> Model -> Lighting -> Lightness Client
* Bluetooth Mesh -> Stack -> Model -> Generic Client Models
* Platform -> Driver -> Button -> Simple Button

After that, follow readme.md file in [*https://github.com/tru5tzz/group1-mock.git*](https://github.com/tru5tzz/group1-mock.git) to change some operation code of the sensor server and sensor client to solve the requirement.

1. Code Walkthrough

### Unprovisioned Mode, Provisioning and Configuration

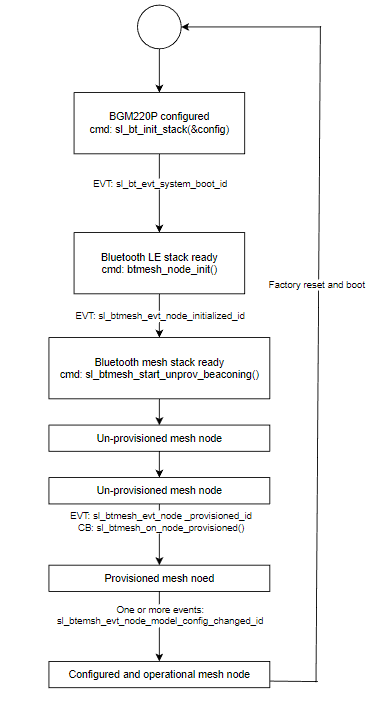
BLE and BLE Mesh handle related operations by storing them as events, each event has a event id to distinguish.

Event *sl\_bt\_evt\_system\_boot\_id* : when this event is raised, the application checks if a button is pressed. If yes, it calls the function *sl\_btmesh\_initiate\_full\_reset(),* which halts the system and performs a factory reset by erasing PS storage. The factory reset is also done after receiving a node\_reset event (*sl\_btmesh\_evt\_node\_reset\_id*). If no button is pressed, then the name of the device is set based on the Bluetooth address, and the function *sl\_btmesh\_node\_init()* is called to initialize the Bluetooth mesh node stack.

Event *sl\_btmesh\_evt\_node\_initialized\_id* : this event indicates that the Bluetooth mesh node stack initialization is complete. When this event is raised, the callback function *sl\_btmesh\_on\_provision\_init\_status()* is called to provide information about the node status. Then the necessary models are initialized

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 *sl\_btmesh\_node\_start\_unprov\_beaconing().*

The API *sl\_btmesh\_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. The start of provisioning is indicated with the event *sl\_btmesh\_evt\_node\_provisioning\_started\_id*. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioning\_started()* is called. During provisioning, no actions are required from the user application. The Bluetooth mesh stack automatically handles network key configuration and other operations.

Then they wait for the event *sl\_btmesh\_evt\_node\_provisioned\_id* that indicates provisioning is complete. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioned()* is called.

**Note**: After the event *sl\_btmesh\_evt\_node\_provisioned\_id* is raised, low power node feature is initialized and try to find friend node.

The next step after provisioning is the configuration of the node. As explained in *QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide*, the smartphone app is used to configure a node either as a sensor client or a sensor server and assign it to a group. The configuration procedure consists of the following steps:

* Provisioner distributes an application key to the node.
* The 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 Bluetooth 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:

* *sl\_btmesh\_evt\_node\_key\_added\_id:* generated when the provisioner has sent a new key (network or application)
* *sl\_btmesh\_evt\_node\_model\_config\_changed\_id:* indicates that the provisioner has modified the configuration of the local model (either publish or subscribe settings changed)

### Switch Node operation

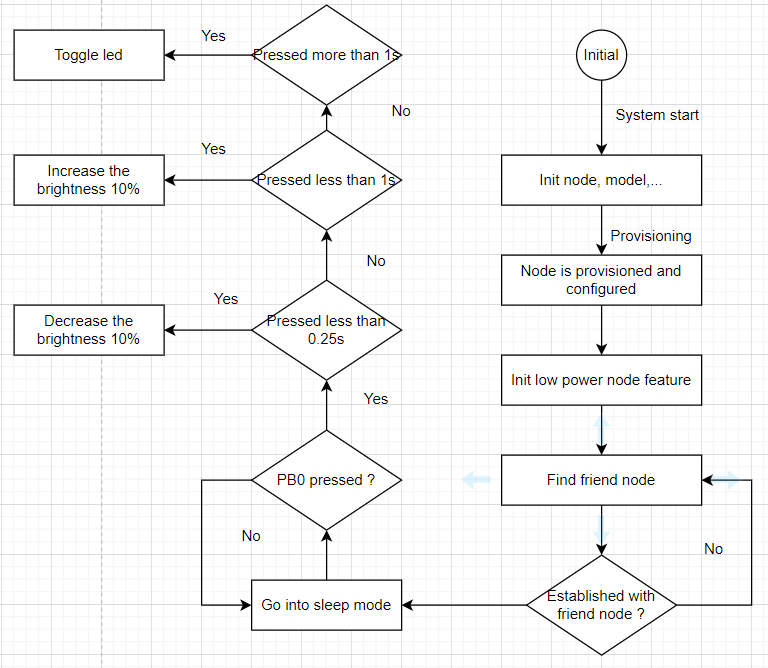
This section describes the basic operation of the code **Switch node**. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app.

In this project, switch node support three type:

* On/off all led in group ( toggle led ),
* Increase the lightness of all node in group,
* Decrease the lightness of all node in the group.

The main purpose of switch node is controlling all the light in a group which it is belong to.

The below diagram describe the work flow of switch node:



When PB0 button is pressed, it call to callback function app\_button\_press\_cb():

* if the button is pressed for less than 0.25s, the brightness will be set to decrease by 10% and function sl\_btmesh\_set\_lightness() is called to transmit the command to the light node.
* if the button is pressed for more than 0.25s and less than 1s, the brightness will be set to increase by 10% and function sl\_btmesh\_set\_lightness() is called to transmit the command to the light node.
* if the button is pressed for more than 1s, if the brightness is non-zero, the light will be turn off, and if the brightness is zero, the light will be turn on.

The brightness value is stored in global variable *lightness\_percent* and control by function *wrap\_add()* to always be in range 0-100.

The callback function sl\_btmesh\_set\_lightness() , this function creates struct **mesh\_generic\_request** which contain:

* **kind of request: *mesh\_lighting\_request\_lightness\_actual*,**
* **value of request: brightness**

and invoke to *mesh\_lib\_generic\_client\_publish()* .

Function *mesh\_lib\_generic\_client\_publish()* will choose:

* model\_id: MESH\_LIGHTING\_LIGHTNESS\_CLIENT\_MODEL\_ID model in client side to send message control lightness in BLE Mesh ( in Light Node, there is LIGHT LIGHTNESS SERVER MODEL which receive message ).
* element\_index: element which Light Lightness client model is belong to
* And more parameter to send in message.

…and call *sl\_btmesh\_generic\_client\_publish()* to publish the message which contain brightness level to all light node in the group.

**Note:** It use Light Lightness Client Model to send message so only light node with Light Lightness Server Model can catch the message and raise an event to process it.

# Light Node In the Mesh network

This section describes the basic operation of the code **Light Node**. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app.

1. Requirement for light node:

* *The Light is controlled by Generic On/Off Model.*
* *Support CTL (Color Turn-able Light) light element and another light element.*
* *Send heartbeat to mobile app every 5 minues.*

1. 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 node(s).

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 to a Bluetooth mesh network and to 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 a switch and light 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, publishes and subscribes settings, application-level security, and mesh models.

After provisioning and configuration, the Bluetooth mesh network is operational, and the switch can be used to control the light. The switch buttons can be used to control all the light in a group.

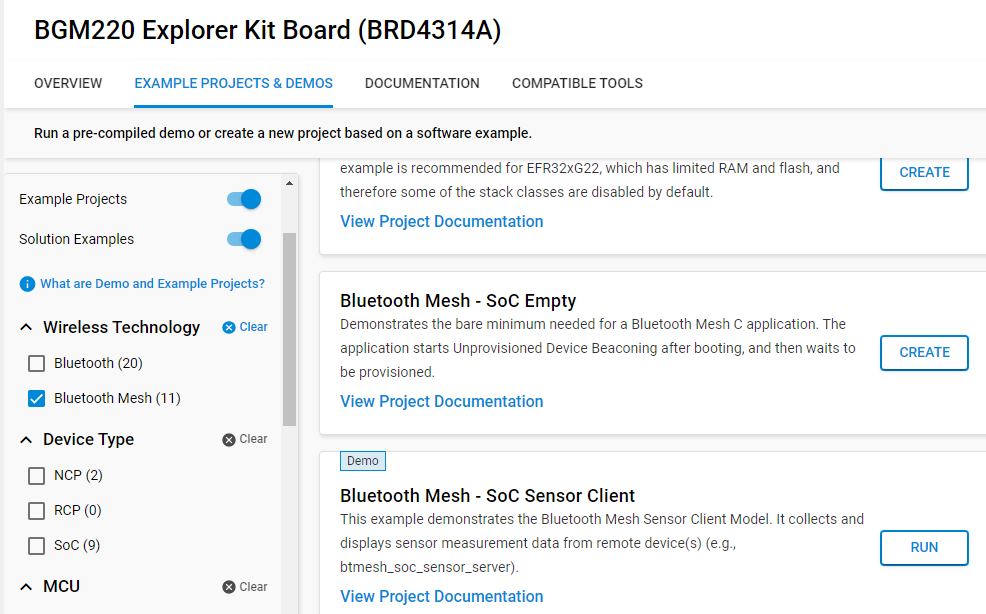
1. Starting with code.

The SDK version 4.3 of Simplisity Studio 5 has a SoC Light demo. But it doesn't fit the hardware of the BGM220. So this example is rewritten based on example SoC Empty and adds the components and models needed to fit the device.

Quick-Start Guide for an introduction to configuring and building your own projects, and for a guide to additional resources. Open Simplicity Studio 5 with a compatible SoC wireless kit connected to the computer. Select the part in Debug Adapters view to open the Launcher perspective. Click the Example Projects & Demos tab. Under Technology Type, filter on Bluetooth Mesh. Next to the Bluetooth Mesh - SoC Empty Project, click Create.

Modify project settings, and click Finish to create the project. Select the project .slcp file if it is not already selected, click the Software

After Finish, the files in the project will be autogenerated by the system.



After the above steps, you need to add components in order for the light node to work. The components are as follows:

* Button press
* Lightness server
* Event Logging
* Factory Reset
* Provision Decorator
* PWM (led0).

After the above steps, you need to add components in order for the light node to work. The components are as follows:

* Lightness server.
* Generic on/off server.

More specifically written in readme in: [group1-mock/LightNode at main · tru5tzz/group1-mock (github.com)](https://github.com/tru5tzz/group1-mock/tree/main/LightNode)

1. Code Walk Through
   1. Unprovisioned Mode, Provisioning and Configuration

In unprovisioned mode, both sensor client and sensor server 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 (*sl\_bt\_evt\_system\_boot\_id*), the application checks if a button is pressed. If yes, it calls the function *sl\_btmesh\_initiate\_full\_reset(),* which halts the system and performs a factory reset by erasing PS storage. The factory reset is also done after receiving a node\_reset event (*sl\_btmesh\_evt\_node\_reset\_id*). If no button is pressed, then the name of the device is set based on the Bluetooth address, and the function *sl\_btmesh\_node\_init()* is called to initialize the Bluetooth mesh node stack.

The event *sl\_btmesh\_evt\_node\_initialized\_id* indicates that the Bluetooth mesh node stack initialization is complete. When this event is raised, the callback function *sl\_btmesh\_on\_provision\_init\_status()* is called to provide 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 *sl\_btmesh\_node\_start\_unprov\_beaconing().*

The API *sl\_btmesh\_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. The start of provisioning is indicated with the event *sl\_btmesh\_evt\_node\_provisioning\_started\_id*. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioning\_started()* is called.

During provisioning, no actions are required from the user application. The Bluetooth mesh stack automatically handles network key configuration and other operations. The sensor client and the sensor server application simply start blinking the LEDs to indicate that provisioning is in progress. Then they wait for the event *sl\_btmesh\_evt\_node\_provisioned\_id* that indicates provisioning is complete. When this event is raised, the callback function *sl\_btmesh\_on\_node\_provisioned()* is called.

The next step after provisioning is the configuration of the node. As explained in *QSG176: Bluetooth Mesh SDK v2.x Quick-Start Guide*, the smartphone app is used to configure a node either as a sensor client or a sensor server and assign it to a group. The configuration procedure consists of the following steps:

* Provisioner distributes an application key to the node.
* The 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 Bluetooth 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:

• *sl\_btmesh\_evt\_node\_key\_added\_id:* generated when the provisioner has sent a new key (network or application)

• *sl\_btmesh\_evt\_node\_model\_config\_changed\_id:* indicates that the provisioner has modified the configuration of the local model (either publish or subscribe settings changed)

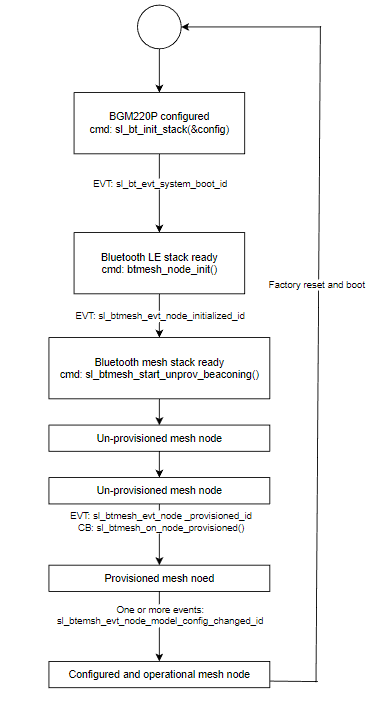


Figure 8‑13 Life Cycle of the Application

* 1. Light Node operation

This section describes the basic operation of the Light Server. It is assumed that the node is already provisioned and publish-subscribe settings have been configured by the smartphone app. The main purpose of the light server is to receive publish event from the light client (switch) to set the lightness.

When the light server receives a request from the light client, *the lightness\_request (...)* function is called, from which the function refers to the *sl\_btmesh\_lighting\_set\_level(...)*, in which the call is to the function *sl\_btmesh \_lighting \_level\_pwm\_cb(current\_level);*

From this function the lighness parameter is obtained. Then you can use that parameter to set the brightness of the light.

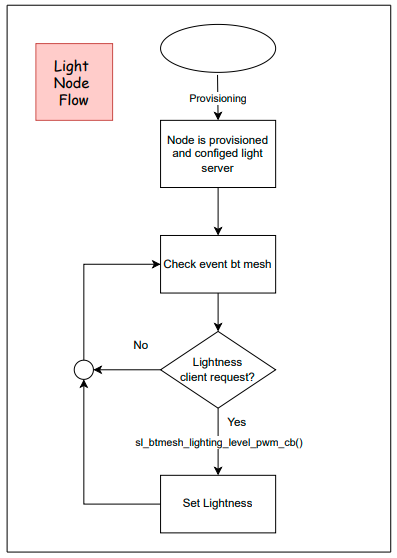


Figure 2‑6 Flow Light Server

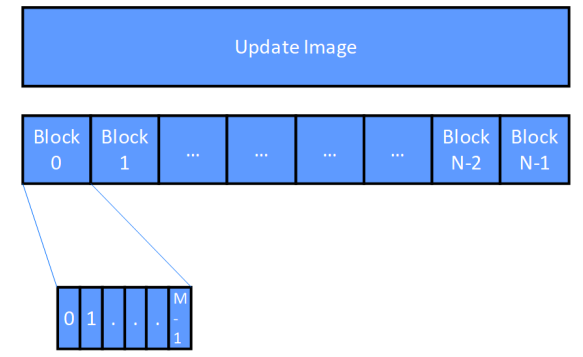
# Device Update Firmware.

There are two main types of Bluetooth Mesh Device Firmware Update (DFU): Mesh DFU and Over-the-Air (OTA) DFU.

1. Mesh Device Firmware Update.

Bluetooth Mesh DFU and BLOB Transfer: Bluetooth Mesh Device Firmware Update is designed as a two-layer architecture to run different protocols. The DFU layer runs the DFU protocol for higher-level decision making that manages and coordinates firmware updates among different types of nodes in the network. The BLOB Transfer layer runs the underlying data transport protocol for handling the transfer of large binary object files. In principle, the BLOB Transfer component can be used standalone to transfer any data files larger than the maximum Access Layer PDU size between devices in the network.

The Bluetooth Mesh BLOB Transfer (MBT) is a component on a node transferring or receiving binary large objects (BLOBs) over a Bluetooth Mesh network. The MBT client sends a BLOB to one or multiple nodes and the MBT server receives the BLOB. Bluetooth Mesh Device Firmware Update uses the MBT component to transfer update images. The following figure illustrates the concept of how the MBT component transfers an update image. The MBT client breaks an update image into suitably-sized blocks based on the capabilities of the MBT servers and transfers these blocks to the servers. Each block is composed of identically sized chunks of data, except for the last chunk which may be smaller than the other chunks if the block’s size is not an integer multiple of the chunk’s size. A single message carries only a single chunk.



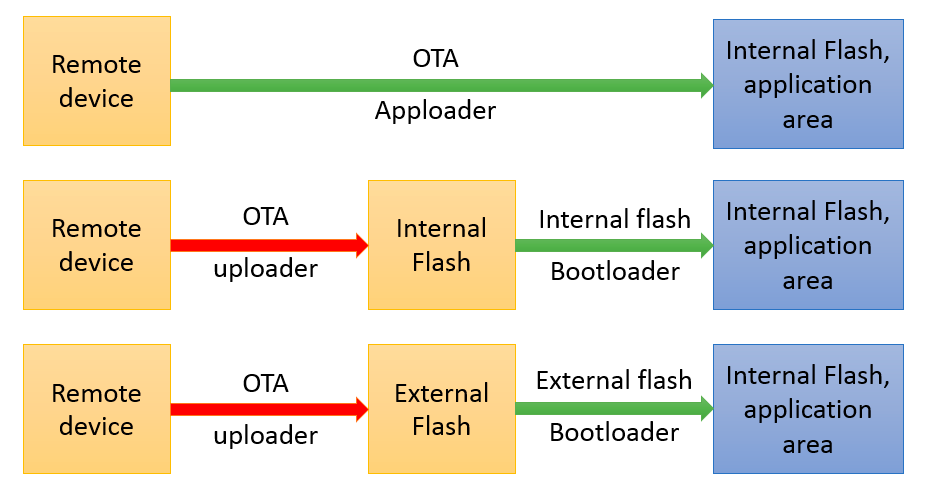
* 1. Over The Air Device Firmware Update (OTA DFU).

Gecko Bootloader can load application images into the application area from the following sources (OTA):

* Internal flash (if the internal flash is big enough to accommodate both the application code and an upgrade image)
* External flash

OTA DFU (Over-The-Air Device Firmware Upgrade) is not implemented in the Gecko Bootloader. To upload images OTA, you will need a dedicated Bluetooth application. You can either use Apploader or user application, as follows:

* Apploader. The Apploader is a simple application, which is separated from the main application and has a minimal Bluetooth stack that handles the upload process.
* User application extended with the ability to upload firmware images to the flash.



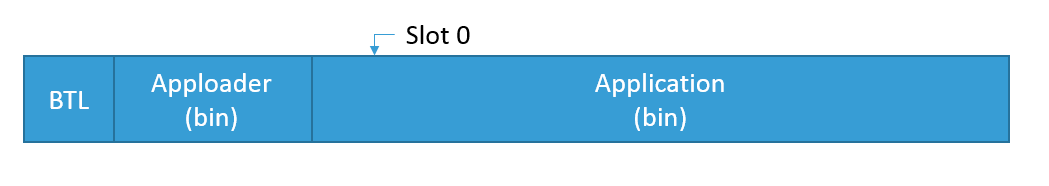
* 1. Apploader

Bluetooth application developed with Silicon Labs Bluetooth SDK comprises two parts: AppLoader and the user application. AppLoader is a small standalone application that is required to support in-place OTA updates. AppLoader can run independently of the user application. It contains a minimal version of the Bluetooth stack, including only those features that are necessary to perform the OTA update. Any Bluetooth features that are not necessary to support OTA updates are disabled in AppLoader to minimize the flash footprint.

The AppLoader features and limitations are summarized below:

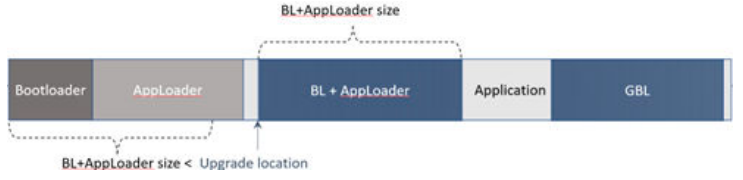
* Enables OTA updating of user application.
* The AppLoader itself can also be updated.
* Only one Bluetooth connection is supported, GATT server role only.
* Encryption and other security features such as bonding are supported.
* PTI is not enabled so it is not possible to use the Network Analyzer with the AppLoader.

The user application is placed in code flash after AppLoader. The default linker script provided in the SDK places the user application so that it starts at the next flash sector following AppLoader. The user application contains a full-featured version of the Bluetooth stack, and it can run independently of the AppLoader. If in-place OTA update does not need to be supported, then the AppLoader can be removed completely to free up flash for other use (code space or data storage).

****

* 1. OTA – DFU Process.

After example “Bluetooth AppLoader OTA DFU” is downloaded, the base address of the bootloader upgrade image needs to be updated in the Bootloader Core component. The recommended value is 0x12000 (for BGM220), so that bootloader upgrade location will not overlap the bootloader with AppLoader. However, the projects can use any appropriate values based on their bootloader and application size.



AppLoader supports two types of update:

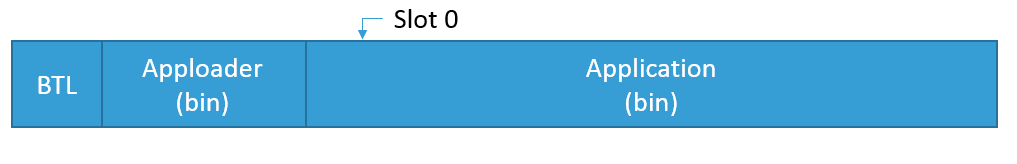
* Full update: both AppLoader and the user application are updated
* Partial update: only the user application is updated.
  + 1. Full update

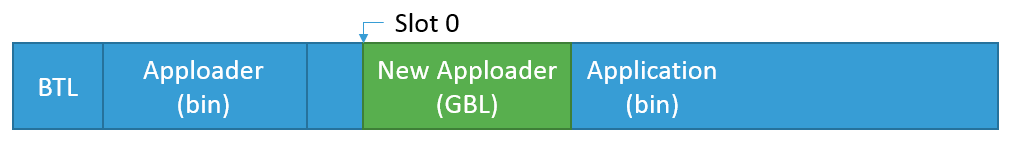
Full update enables updating both the AppLoader and the user application. Full update is done in two steps. Updating the AppLoader always erases the user application and therefore AppLoader update must always be followed by application update.

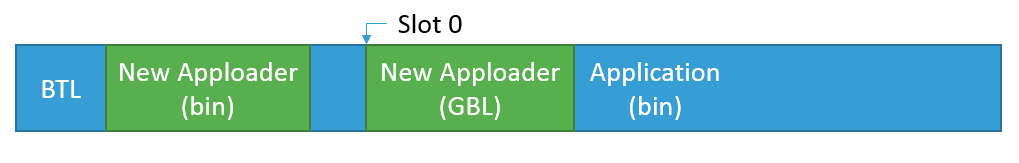
The first phase of full update updates the AppLoader and it consists of following steps:

1. OTA client connects to target device.
2. Client requests target device to reboot into DFU mode.
3. After reboot, client connects again.
4. During the 2nd connection, target device is running AppLoader (not the user application).
5. New AppLoader image (apploader.gbl) is uploaded to the target.
6. AppLoader copies the image into the download area (specified in Gecko bootloader configuration).
7. When upload is finished and connection closed, AppLoader reboots and requests Gecko Bootloader to install the downloaded image.
8. Gecko Bootloader updates AppLoader using the downloaded image and reboots.
9. After reboot, the new AppLoader is started.

At the end of the AppLoader update, the device does not contain a valid user application and therefore AppLoader will remain in DFU mode. To complete the update, a new user application is uploaded following the same sequence of operations that were described for the partial update.





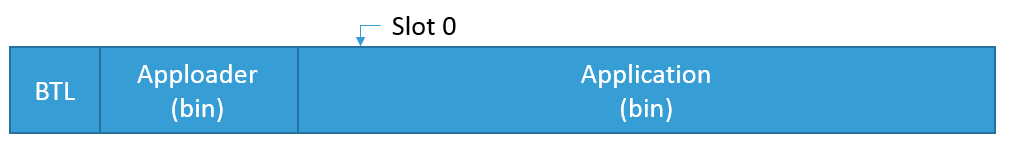




* + 1. Partial Update

The partial update process using AppLoader consists of following steps:

1. OTA client connects to target device.
2. Client requests target device to reboot into DFU mode.
3. After reboot, client connects again.
4. During the 2nd connection, target device is running AppLoader (not the user application).
5. New firmware image (application.gbl) is uploaded to the target.
6. AppLoader copies the new application on top of the existing application.
7. When upload is finished and connection closed, AppLoader reboots back to normal mode.
8. Update complete.

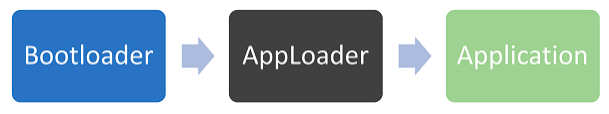




With partial update, it is possible to update the Bluetooth stack and user application. AppLoader is not modified during partial update.

* + 1. OTA DFU Secure

When employing secure boot, the AppLoader and application must be signed individually for the application to be allowed to run. This is because the AppLoader authenticates the application before allowing it to run, just as the bootloader authenticates the AppLoader before allowing it to run.



This section describes how to work with this capability in a production environment

* + - 1. Creating a Single Signed Image with a Batch File

A signed GBL file can be produced by executing create\_bl\_files.bat/create\_bl\_files.sh with a private signing key file, app\_sign\_key.pem, in the same folder. The resulting GBL file, full.gbl, can be flashed directly to the target device for a successful boot. This method is convenient for testing secure boot during the development process, but is not secure for production since it requires the private signing key to be available in plain PEM format, rather than isolating it in a Hardware Security Module (HSM) and does not support the use of bootloader certificates.

* + - * 1. Signing Firmware Images for Production

To sign a firmware image using an HSM, the image must first be separated into the AppLoader and application parts as follows.

1. Extract the AppLoader portion with the following command: objcopy -O srec -j .text\_apploader\* apploader.s37.
2. Sign the AppLoader for secure boot. For specific instructions on signing images with an HSM, see ‘Signing an Application for Secure Boot using a Hard Security Module’ in UG162. It is also possible to sign the AppLoader with a certificate although direct signing is sufficient for most use cases.
3. Extract the application with the following command: objcopy -O srec -R .text\_apploader\* -R .text\_signature\* application.s37.
4. Sign the application for secure boot. The instructions for signing the apploader in step 2 above also apply to the application.
5. Combine the signed apploader and application into a single image as follows: commander convert .s37 .s37 –outfile signed\_fw\_image.s37.
6. Optionally, see “Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module” and “Creating a Signed GBL File Using a Hardware Security Module” in UG162.
   1. Basic Steps to Update Firmware from the User Application.

The basic steps are summarized below.

1. Application initializes the Gecko bootloader by calling *bootloader\_init();*
2. The download area is erased by calling *bootloader\_eraseStorageSlot(0);*
3. The update image (full GBL file) is received either over-the-air or through some physical interface like UART, application writes the received bytes to the download area by calling *bootloader\_writeStorage()*
4. (optional) Application can verify the integrity of the received GBL file by calling *bootloader\_verifyImage()*
5. Before rebooting, call *bootloader\_setAppImageToBootload(0)* to specify the slot ID where new image is stored
6. Reboot and instruct Gecko bootloader to perform the update by calling *bootloader\_rebootAndInstall()*.

It is assumed here that only one download area is configured and therefore the slot index in the above function calls is set to 0. Note that the erase procedure in step 2) above takes several seconds to complete. If the new image is downloaded over a Bluetooth connection then the supervision timeout must be set long enough to avoid connection drops. Alternatively, the download area can be erased in advance, before the Bluetooth connection is opened. A third alternative is to erase the download area one flash page at a time while the writing progresses. This can be done using bootloader\_eraseRawStorage().