AN227069

Getting Started with Bluetooth Mesh

Author: David Solda, Santhosh Vojjala, Sachin Gupta

Associated Part Family: CYBT-XX3XXX-0X, CYW20819, CYW20719, CYW20735, CYW20706

Software Version: ModusToolbox™ 2.1 or later

See Related Application Notes

AN227069 introduces you to Cypress' Bluetooth SIG-compliant Mesh solution. This application note covers Cypress' BLE Mesh architecture, code structure and how to make changes based on your application needs. By the end of this application note, you will be able to develop your own Mesh-enabled products using Cypress' certified module or silicon solution.

Contents

1 Introduction

This application note introduces Bluetooth Mesh and its key concepts and describes how to get started with Cypress' BLE Mesh solution to design your own BLE Mesh-enabled applications. Cypress is a world-leader in connectivity solutions for Internet of Things (IoT) applications, combining best-in-class silicon platforms with the industry's most broadly deployed Bluetooth stack. This combination allows you to deploy successful BLE Mesh-enabled products with minimum development costs and with a rapid cycle time.

2 More Information

This section provides a list of learning resources for developing Bluetooth Mesh products that can help you to get started and develop complete applications with Mesh-enabled Cypress EZ-BT™ Modules or silicon products. All documentation can be accessed at the Cypress Bluetooth Mesh website.

2.1 Mesh-Qualified Silicon Devices

Cypress offers a portfolio of Mesh-qualified Bluetooth devices. Table 1 provides the list of these devices and their capabilities. For the most up-to-date details on Cypress Mesh-qualified devices, visit the Cypress BLE Mesh Webpage.

Table 1. Mesh-Qualified Bluetooth Device Features

Silicon Device	BLE/BR/EDR	Bluetooth Core Specification	SIG Mesh Qualified	CPU Core	Flash (KB)	RAM (KB)	GPIOs (Maximum)	Tx Power (dB)	Temperature (Degree C)	Software Support
CYW20706	BLE/BR/EDR	5.0	Yes	96-MHz CM3	-	352	24	+10	-30 to + 85, -40 to +105	Modus Toolbox
CYW20719	BLE/BR/EDR	5.0	Yes	96-MHz CM4	1024	512	40	+4	-30 to + 85	Modus Toolbox
CYW20735	BLE/BR	5.0	Yes	96-MHz CM4	-	384	25	+10	-30 to + 85	Modus Toolbox
CYW20819	BLE/BR/EDR	5.0	Yes	96-MHz CM4	256	176	40	+4	-30 to + 85, -40 to +105	ModusToolbox
CYW20820	BLE/BR/EDR	5.0	Yes	96-MHz CM4	256	176	40	+10.5	-30 to + 85	Modus Toolbox

2.2 Mesh-Qualified EZ-BT Bluetooth Modules

Cypress provides a variety of certified Bluetooth Mesh-Qualified modules to reduce hardware development cost and time. Every module within the EZ-BT Module family has a unique Marketing Part Number (MPN) used for ordering. The MPN format is shown in Figure 1.

Figure 1. EZ-BT Module Marketing Part Numbering Format

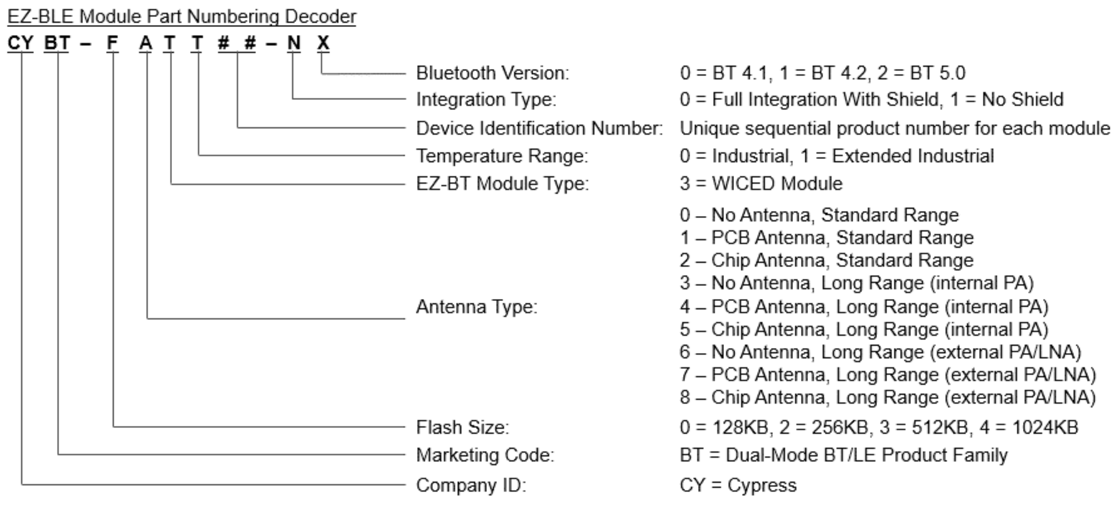

EZ-BLE Module Part Numbering Decoder
CY BT – F A T T # # – N X
Bluetooth Version: 0 = BT 4.1, 1 = BT 4.2, 2 = BT 5.0
Integration Type: 0 = Full Integration With Shield, 1 = No Shield
Device Identification Number: Unique sequential product number for each module
Temperature Range: 0 = Industrial, 1 = Extended Industrial
EZ-BT Module Type: 3 = WICED Module
Antenna Type:
0 – No Antenna, Standard Range
1 – PCB Antenna, Standard Range
2 – Chip Antenna, Standard Range
3 – No Antenna, Long Range (internal PA)
4 – PCB Antenna, Long Range (internal PA)
5 – Chip Antenna, Long Range (internal PA)
6 – No Antenna, Long Range (external PA/LNA)
7 – PCB Antenna, Long Range (external PA/LNA)
8 – Chip Antenna, Long Range (external PA/LNA)
Flash Size: 0 = 128KB, 2 = 256KB, 3 = 512KB, 4 = 1024KB
Marketing Code: BT = Dual-Mode BT/LE Product Family
Company ID: CY = Cypress

Table 2 summarizes the features and capabilities of each EZ-BT Module MPN available from Cypress. Table 2 details all modules that are sampling or in production. For the latest list of modules available, visit the Cypress Bluetooth website.

Table 2. EZ-BT Module MPN Features and Capabilities

Marketing Part Number	Silicon Device	Module Size (mm)	Range[1]	Regulatory Certification	BLE Standard	SIG Mesh Qualified	HomeKit Capable	Antenna Type	Package	GPIOs (Maximum)	Serial Flash (KB)	SRAM (KB)	I^2S/PCM	PDM	PWMs	ADC
CYBT-343026-01	CYW20706	14.52 x 19.2 x 1.95	225	Yes	5.0	Yes	Yes	Trace	24-SMT	11	512	352	Yes	No	4	Yes
CYBT-333032-02	CYW20706	12.00 x 13.5 x 1.95	250	Yes	5.0	Yes	Yes	RF Pad	24-SMT	11	512	352	Yes	No	4	Yes
CYBT-343047-02	CYW20706	12.00 x 13.5 x 1.95	250	Yes	5.0	Yes	Yes	u.FL	24-SMT	11	512	352	Yes	No	4	Yes
CYBT-343151-02	CYW20706	12.00 x 15.5 x 1.95	400	Yes	5.0	Yes	Yes	Trace	24-SMT	11	512	352	Yes	No	4	Yes
CYBT-353027-02	CYW20707	9.00 x 9.00 x 1.75	150	Yes	5.0	Yes	Yes	Chip	19-SMT	8	512	352	Yes	No	0	Yes
CYBT-423028-02	CYW20719	11.00 x 11.00 x 1.70	75	Yes	5.0	Yes	Yes	Chip	28-SMT	17	1024	512	Yes	Yes	6	Yes
CYBT-413034-02	CYW20719	12.00 x 16.30 x 1.70	75	Yes	5.0	Yes	Yes	Trace	30-SMT	17	1024	512	Yes	Yes	6	Yes
CYBT-483039-02	CYW20719	12.75 x 18.59 x 1.80	1000[2]	Yes	5.0	Yes	Yes	Chip	34-SMT	15	1024	512	Yes	Yes	6	Yes
CYBT-213043-02	CYW20819	12.00 x 16.61 x 1.70	75	Yes	5.0	Yes	Yes	Trace	35-SMT	22	256	176	Yes	Yes	6	Yes
CYBT-253059-02	CYW20820	11.00 x 11.00 x 1.70	75	Yes	5.0	Yes	Yes	Trace	35-SMT	22	256	176	Yes	Yes	6	Yes

2.3 Mesh-Qualified EZ-BT Module and Silicon Datasheets

EZ-BT WICED Module and silicon datasheets list the features, pinouts, device-level specifications, and fixed-function peripheral electrical specifications of Cypress module and silicon products. Visit the Cypress BLE and Bluetooth website to access all datasheets for EZ-BT WICED Modules and silicon devices.

The CYBT-213043-MESH kit referred to in this application note utilizes the CYBT-213043-02 certified module. The datasheet for this module can be accessed here. This datasheet can also be accessed on the Cypress BLE and Bluetooth website.

2.4 Bluetooth Mesh Evaluation Boards

CYBT-213043-MESH is a Mesh evaluation kit that features the CYBT-213043-02 EZ-BT certified module, based on the CYW20819 silicon device. This kit includes four Mesh evaluation boards, allowing you to develop a small Mesh network without the need to purchase multiple evaluation boards. Each board features an Ambient Light Sensor (ALS), a PIR motion sensor and lens, a thermistor, an RGB LED, a user switch and an on-board programmer (USB-to-UART). To learn more about the kit and available resources, visit the CYBT-213043-MESH EZ-BT Module Mesh Evaluation Kit Webpage.

If you are designing Bluetooth Mesh products based on a device or module other than the CYW20819, you can use the respective device's evaluation kit. Each EZ-BT WICED Module and silicon device offers a low-cost Arduino-compatible evaluation board, providing an easy-to-use vehicle to develop and evaluate Bluetooth Mesh using the EZ-BT WICED Modules and silicon without requiring custom hardware design. Visit the Cypress Bluetooth Mesh website to view and purchase Cypress evaluation boards.

[1] Measured in meters and is specified as Full Line-of-Sight ( ) in a Noise-Free environment.

[2] 1 km range is certified for US/FCC use only. European and Japan certifications are limited to 10 dBm (600 meters maximum).

2.5 Cypress Developer Community for Bluetooth Mesh and Technical Support

Whether you're a customer, partner, or a developer interested in designing Bluetooth Mesh products, the Cypress ModusToolbox Bluetooth Community offers you a place to learn, share, and engage with both Cypress experts and other embedded engineers around the world. If you have any technical questions, you can post them on Cypress Developer Community forums that are actively managed by Cypress engineers.

3 BLE Mesh Overview

Bluetooth Low Energy (BLE) has emerged as the preferred standard for IoT connectivity. The BLE standard's inherent low-power operation coupled with installed support on smart phones and personal computers have been a driving force behind its growth in IoT applications.

Traditional BLE applications use point-to-point communication, where each pair of devices sends data to and from each other. Each of these connections utilizes a GAP Central and a GAP Peripheral role to accomplish the communication.

In contrast, in a Mesh network, every device can communicate with every other device within the same network. This capability extends the overall communication range beyond the range of each individual device. In a Bluetooth Mesh implementation, messages are sent using advertising packets. By using advertising packets for network communication, the relaying of network messages requires no connections to be made to share information. Rather, data is broadcast by a transmitting device using these advertising packets and is received by any device that is part of the network and that is within the range of the transmitter. Devices that are provisioned into a BLE Mesh network are called "nodes". Bluetooth Mesh defines various types of nodes; some utilized for range extension and network coverage while others are optimized for low-power operation and only wake when they need to.

BLE Mesh is emerging as a preferred home automation technology driven by the fact that BLE Mesh nodes can be controlled directly from a mobile phone or PC without the need to install a network gateway. The BLE Mesh specification has been defined to provide standards-based operating models tailored to home and industrial automation use cases. Standard message formats for defined use cases (models) enable rapid deployment and interoperability assurance with other BLE Mesh products.

3.1 BLE Mesh Nodes and Feature Types

BLE Mesh has several node/feature types. Each node can include multiple features if desired. This section provides an overview of the different BLE Mesh node types and various features that can be employed on these nodes.

3.1.1 Relay Node

The Relay feature of a node includes the ability to relay messages over the advertising bearer. In a Mesh network, most nodes include this feature unless they are low-power nodes (LPNs). An example of a Relay feature in use is a smart light bulb or light switch. Typically, any node that is wall-powered is likely to be a Relay Node as they have the power needed to listen continuously for advertising packets and relay them to additional nodes in the network.

3.1.2 Low-Power Node (LPN)

A Low-Power Node is a critical feature for BLE Mesh-enabled applications. Low-Power Nodes operate in low duty cycles, by working and communicating in conjunction with Friend Nodes that collect messages on the LPN's behalf. The LPN will ping its Friend node at a defined interval to check for pending messages, and after communication with the Friend Node, it will go back to a low-power sleep state to conserve power. Any battery powered application, such as a glass door sensor or smoke detector can be a low-power node. With the BLE Mesh low-power feature, even a coin-cell battery is often sufficient to power a Low-Power Node for over a year.

3.1.3 Friend Node

A node with the Friend feature will listen to any messages that are relayed in the network and are intended for its associated LPNs. The Friend node will store these messages and deliver them to the associated LPNs when the LPNs wake and query the Friend node. Since a Friend Node needs to store messages for one or more LPN, the Friend Node may require more memory than other node types. The amount of memory required is dependent on the amount of data/commands required to be stored on the Friend node that will be communicated to the LPNs during a polling operation.

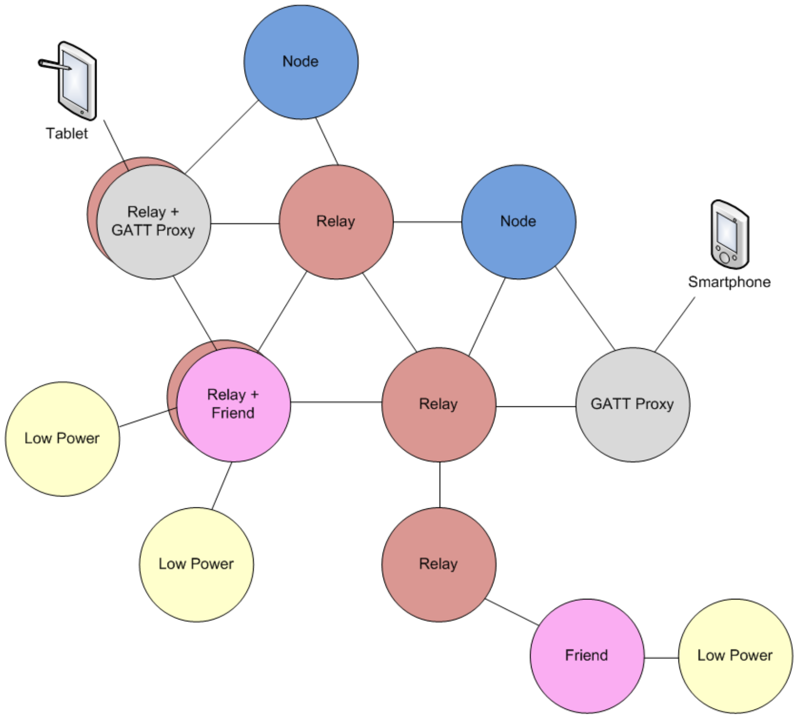
3.1.4 Proxy Node

BLE Mesh networks relay messages over an advertising bearer. The Proxy feature is the ability of a node to relay messages between the GATT (General ATTribute) and advertising bearers. This feature allows devices, such as smartphone, that support BLE but not BLE Mesh, to communicate with a Mesh network. A Proxy Node is the entry point into the Mesh network for those devices that don't directly support BLE Mesh. Any node that supports the Proxy feature can act as the interface for a smartphone/PC over a GATT connection.

3.1.5 Provisioner

The Provisioning feature is a key to enable a secure BLE Mesh network. Provisioning is the process of adding a new node to a network. When your consumer purchases a BLE Mesh-enabled device, they need to have the ability to add this device to their network. This requires several steps to ensure that unintended devices are not provisioned to the network. A device that is unprovisioned will send beacons at a predetermined interval, and the provisioning device will initiate the provisioning process once the unprovisioned device is found and selected. BLE Mesh uses the Elliptic Curve Diffie Hellman (ECDH) algorithm to exchange keys for secure communication to ensure that your network and the provisioning of devices into it is secure.

Figure 2. BLE Mesh Network and Node Types



3.2 BLE Mesh Operation and Key Concepts

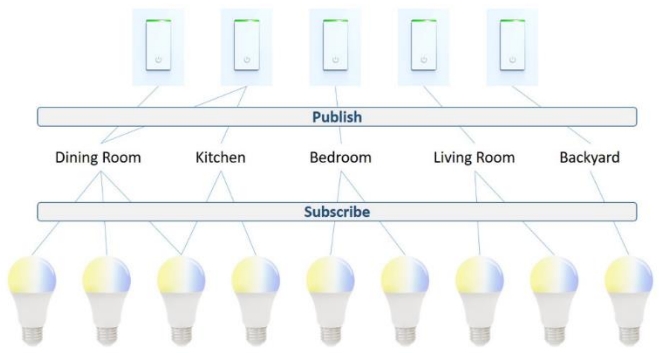
3.2.1 Managed Flood

BLE Mesh uses a managed flood operation to transfer messages. Managed flood is inherently a multi-path implementation that adds enough redundancy to ensure that a message reaches its destination. A brute force flood implementation is one where every node blindly relays (retransmits) any message that it has received. Conversely, the BLE managed flood operation is designed to prevent devices from relaying previously received messages by adding all messages to a cached list. When a message is received, it is checked against the list and ignored if already present. If the message has not been previously received, then it is added to the cache so that it can be ignored in the future. Each message includes a Time to Live (TTL) value that limits the number of times a message can be relayed. Each time a message is received and then relayed (up to a maximum of 126 times) by a device, the TTL value is decremented by 1.

3.2.2 Publish- and Subscription-Based Communication

A Publisher Node sends messages, and only nodes that have subscribed to the Publisher will act on these messages. This ensures that different types of products can coexist in a network without being bothered by messages from devices they do not need to listen to. An example of this operation is different rooms within your home, with each room subscribing to messages related to the specific light switches for that room. In addition, messages can be either unicast, multicast, and/or broadcast, meaning that a message can reach one, a few, or all nodes in the network.

Figure 3. Publish and Subscribe Example



3.2.3 Elements

Elements consist of entities that define a node’s functionality. Each node can have one or more Elements. Every node has at least one Element called the “Primary Element”. For example, a dimmable light bulb generally has one Element; this Element can expose one or more functionalities such as ON/OFF and level (brightness) control. In this example, the Light Lightness Server Model is used to achieve the ON/OFF and level control functionality.

Another example is a dimmable light bulb that also includes an occupancy sensor. In this example, the node will have two Elements: one for the lighting function and the other for the sensor function. The Primary Element in this case would be the lighting function, and the Light Lightness Server Model is used to access the lighting controls. The secondary element in this example is the sensor, where the occupancy sensor’s states can be accessed. Figure 4 shows a representation of both examples.

Every Element within a node has a unique address, known as a unicast address. This allows each Element to be addressed independently of other Elements within the same node.

3.2.4 Models

Models are used to define the functionality of a node – they bring together the concepts of states, transitions, bindings, and messages for an Element. Models are analogous to Services in regular Bluetooth devices. There are three types of Mesh Models – Client Models, Server Models, and Control models which implement both a Client and Server in a single model.

Server Model – A Server Model can have one or more states spanning one or more Elements. The Server Model defines messages a model may transmit and receive. It also defines the Element’s behaviors based on these messages. Said another way, Server Models expose the states of the Element that can be read or controlled by a Client. An example application of the Server Model is a sensor that exposes the sensor’s state or a light bulb that stores the current state of the light.

Client Model – A Client Model defines the set of messages to request and change the state of a Server. For example, an ON/OFF Switch (operating as a Client) can send a message to an ON/OFF Server to change the device’s state.

Control Model – An end application can use Server Models or Client Models, or both along with control logic. Any combination of Server and Client Models results in a Control Model.

3.2.4.1 Model Hierarchy

Models can (and often do) extend the functionality of other models. That is, models can be hierarchical. For example, the ‘Light Lightness’ model extends the ‘Generic OnOff Server’ model and the ‘Generic Level Server’ model. What that means is that if you implement the Light Lightness model in an application, you get all the Light Lightness functionality plus all the Generic Level and Generic OnOff functionality.

Models that do not extend other models are known as "root models".

3.2.5 States and Properties

3.2.5.1 States

Elements can be in various conditions; in Bluetooth Mesh, these conditions are stored in values called states. Each state is a value of a certain type contained within an element. In addition to the values, states have behaviors that are associated with that state. States are defined by the Bluetooth SIG.

For example, there is a state called ‘Generic OnOff’ which can have two values – ON or OFF. This is useful for devices like light bulbs or fan motors. The term ‘Generic’ is used to indicate that this state and its behaviors may be useful in different kinds of Mesh devices.

3.2.5.2 Properties

Properties also contain values relating to an element; however, unlike states, properties provide the context for interpreting states. For example, consider a device that wants to send a temperature state value. The temperature state may be "Present Indoor Ambient Temperature" or "Present Outdoor Ambient Temperature". In this case, a property would be used to provide the context for the temperature state value.

Properties can be manufacturer properties, which are read-only or Admin properties, which allow read-write access.

3.2.5.3 State Transitions

State transitions may be instantaneous or may execute over a period called the transition time.

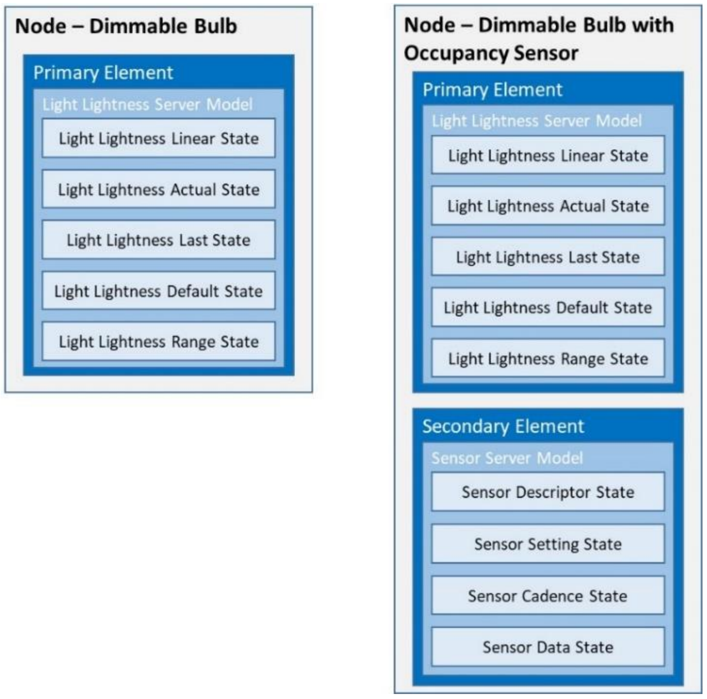
3.2.5.4 State Binding

States may be bound together such that a change in one state causes a change in the other. One state may be bound to multiple other states. For example, a light controlled by a dimmer switch will have one state called `Generic OnOff` and another called `Generic Level` to specify the brightness. If the light is in the ON state but is dimmed to the point that the Level becomes zero, then the OnOff state will transition to OFF. State binding is used in the Dimmable Light Bulb Code Example discussed later in this document.

State binding is defined by the models that contain the states in question. These can be found in the Bluetooth Mesh Specification.

Figure 4 illustrates the relationship between a Mesh node, elements, models and states.

Figure 4. Relationship Between Mesh Node, Elements, Models and States


Node – Dimmable Bulb
Primary Element
Light Lightness Server Model
Light Lightness Linear State
Light Lightness Actual State
Light Lightness Last State
Light Lightness Default State
Light Lightness Range State
Node – Dimmable Bulb with Occupancy Sensor
Primary Element
Light Lightness Server Model
Light Lightness Linear State
Light Lightness Actual State
Light Lightness Last State
Light Lightness Default State
Light Lightness Range State
Secondary Element
Sensor Server Model
Sensor Descriptor State
Sensor Setting State
Sensor Cadence State
Sensor Data State

3.3 Security and Privacy in BLE Mesh

Security and privacy are always a concern in wireless-connected devices. BLE Mesh incorporates several techniques to alleviate this concern. Table 3 details the security and privacy methods implemented in a BLE Mesh solution. To learn more about BLE security, refer to Bluetooth Mesh security overview at Bluetooth SIG website. To learn more about how they are implemented, refer to the BLE Mesh Networking Specification.

Table 3. Bluetooth Mesh Security and Privacy

Security / Privacy Feature	What does it do?
Encryption and Authentication	All Bluetooth mesh messages are encrypted and authenticated.
Separation of Concerns	Network security, application security, and device security are addressed independently by means of separate keys for each. • An **AppKey** is used to avoid any conflict of interest and differentiate one application type from another. For example, a sensor node cannot decode a message encrypted with the lighting AppKey. • Each node possesses one or more **NetKeys** based on the network and subnets they are part of. The NetKey separates each network and subnet from all others. • Each node has a unique **DevKey** that is used for provisioning and configuration which separates it from all other devices.
Area Isolation	A Bluetooth Mesh network can be divided into subnets, each cryptographically distinct and secure from the others. For example, each room in a hotel can be a subnet while the hotel is the complete network. Subnets allow guests from one room not to interfere with other rooms.
Key Refresh, Trashcan Attack Protection, and Node Removal	Security keys can be changed during the life of the Bluetooth mesh network via a Key Refresh procedure. When a node is removed from the network, the Mesh network runs the key refresh procedure to change the keys on all other nodes. Once keys are refreshed, the (old) keys stored on the node that were removed have no value. This feature addresses security concerns if someone gains access to a device that was present earlier in a BLE network – this type of attached is known as a trashcan attack.
Replay Attack Protection	Bluetooth mesh security protects the network against replay attacks. Replay attacks are one of the most common attacks in which an eavesdropper listens to a message and then replays it with the bad intent. Imagine someone listens to a door unlock message and then replays it in the middle of the night to break into a house. BLE Mesh provides a mechanism to avoid replay attacks by adding a sequence number in each message. Messages with the same or lower sequence number as a previous message are automatically ignored.
Message Obfuscation	Message obfuscation makes it difficult to track messages sent within the network and, as such, provides a privacy mechanism to make it difficult to track nodes.
Secure Device Provisioning	BLE Mesh provides a secure method to add a device to the network.

4 Software Development Tools and Helper Applications

Cypress provides easy-to-use tools and Helper Applications to enable evaluation and development of Bluetooth Mesh products.

1. ModusToolbox
2. iOS helper application
3. Android helper application
4. Windows helper application

4.1 ModusToolbox

ModusToolbox is a free state-of-the-art software development ecosystem that includes the Eclipse IDE for ModusToolbox and Software Development Kits (SDKs) such as the WICED® BT SDK (hereafter referred to as the BT SDK) to develop applications for Cypress IoT products. The Eclipse IDE for ModusToolbox is a multi-platform, Eclipse-based integrated development environment (IDE) used to create new applications, update application code, change middleware settings, and program/debug applications.

Using the IDE, you can enable and configure device resources and middleware libraries; write C source code; and program and debug the device. The IDE provides hooks for launching various tools provided by the BT SDK.

The BT SDK provides the core libraries for creating Bluetooth LE, Classic, and Mesh applications with CYW20xxx devices and EZ-BT modules based on them. It contains configuration tools, drivers, libraries, middleware, makefiles, as well as various utilities and scripts.

This application note uses the Eclipse IDE for ModusToolbox, but the ModusToolbox environment supports other options such as command line operation, Visual Studio Code, IAR, and Keil µVision IDEs. See the ModusToolbox user guide for more information.

4.1.1 Getting Started with ModusToolbox

Visit the ModusToolbox home page to download and install the latest version of ModusToolbox. Refer to the *ModusToolbox Installation Guide* document in the *Documentation* tab of ModusToolbox web page for information on installing the ModusToolbox software. After installing, launch ModusToolbox and navigate to the following items:

- **IDE Quick Start Guide**: Choose **Help** > **Eclipse IDE for ModusToolbox Documentation** > **Quick Start Guide**. This guide gives you the basics for using the Eclipse IDE for ModusToolbox.
- **IDE User Guide**: The detailed user guide for the IDE is under **Help** > **Eclipse IDE for ModusToolbox Documentation** > **User Guide**.
- **ModusToolbox User Guide**: Choose **Help** > **ModusToolbox General Documentation** > **ModusToolbox User Guide**. This guide gives you details of the complete ModusToolbox development environment.

4.1.1.1 Creating a Bluetooth SDK (BT SDK) Project

The Bluetooth SDK project (BT SDK) is required as a set of reference libraries to develop a Bluetooth Mesh application using ModusToolbox. The BT SDK not created by default with the ModusToolbox installation. You must create the BT SDK manually in the workspace before you can develop a Bluetooth Mesh-based application.

1. Open the Eclipse IDE for ModusToolbox.
2. Navigate to **File** > **New** > **ModusToolbox Application** or click “New Application” in Quick Panel
3. Choose the Board Support Package (BSP) for any WICED Bluetooth BSP such as “CYBT-213043-MESH” and click **Next**.
4. Choose “wiced_btsdk” as the application and click **Create**. Once project creation has completed, click “**Close**” button.
5. Wait for ModusToolbox to import the BT SDK project. It shows up as “wiced_btsdk” under the Project Explorer window. You will need to create the “wiced_btsdk” project just once in the working directory (i.e., Eclipse workspace).
Note: Do not change the name of the “wiced_btsdk” project. All Bluetooth application projects use this project name in their application makefiles.

4.1.2 Eclipse IDE for ModusToolbox

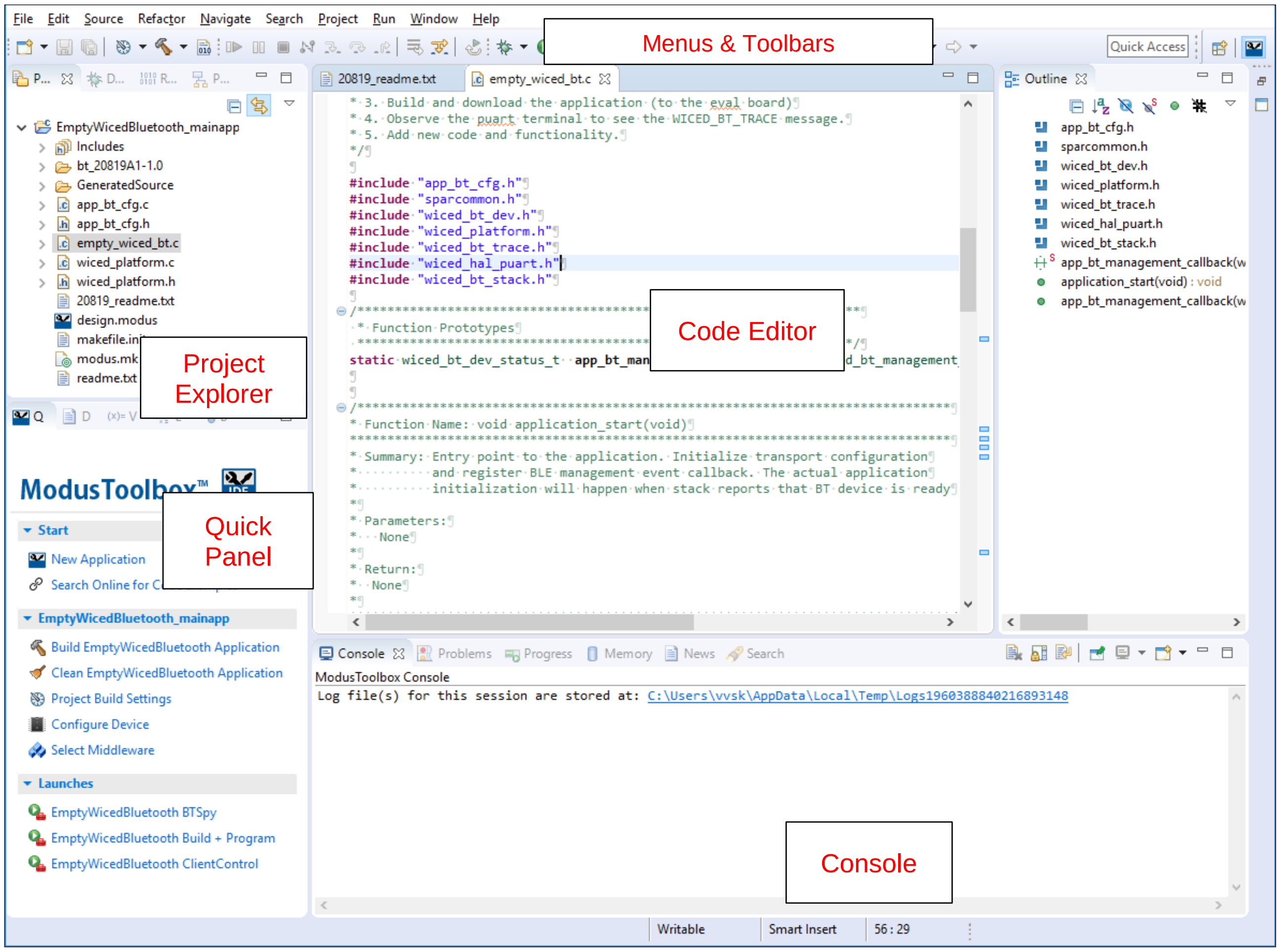
The Eclipse IDE for ModusToolbox is based on the Eclipse IDE "Oxygen" version. It uses several plugins, including the Eclipse C/C++ Development Tools (CDT) plugin. Cypress provides an Eclipse Survival Guide, which provides tips and hints for how to use the Eclipse IDE for ModusToolbox.

The IDE contains Eclipse-standard menus and toolbars, plus various panes such as the Project Explorer, Code Editor, and Console as shown in Figure 5. One difference from the standard Eclipse IDE is the "ModusToolbox Perspective." This perspective provides the **Quick Panel**, a **News View**, and adds tabs to the **Project Explorer**. In the IDE, the top-level entity that you ultimately program to a device is called an application. The application consists of one or more Eclipse projects. The IDE handles all dependencies between the projects automatically. It also provides hooks for launching various tools provided by the software development kits (SDKs).

With the IDE, you can:

1. Create a new application based on a list of starter applications, filtered by kit or device, or browse the collection of code examples online.
2. Configure device resources to build your hardware system design in the workspace.
3. Add software components or middleware.
4. Develop your application firmware.

Figure 5. ModusToolbox



```
 * 3. Build and download the application (to the eval board)
 * 4. Observe the puart terminal to see the WICED_BT_TRACE message.
 * 5. Add new code and functionality.
 */

#include "app_bt_cfg.h"
#include "sparcommon.h"
#include "wiced_bt_dev.h"
#include "wiced_platform.h"
#include "wiced_bt_trace.h"
#include "wiced_hal_puart.h"
#include "wiced_bt_stack.h"

/****************************************                **
 * Function Prototypes
 ****************************************                */
static wiced_bt_dev_status_t  app_bt_man                d_bt_management


/**************************************************************************
 * Function Name: void application_start(void)
 **************************************************************************
 * Summary: Entry point to the application. Initialize transport configuration
 *          and register BLE management event callback. The actual application
 *          initialization will happen when stack reports that BT device is ready
 *
 * Parameters:
 *   None
 *
 * Return:
 *  None
 *
```

```
ModusToolbox Console
Log file(s) for this session are stored at: C:\Users\vvsk\AppData\Local\Temp\Logs1960388840216893148
```

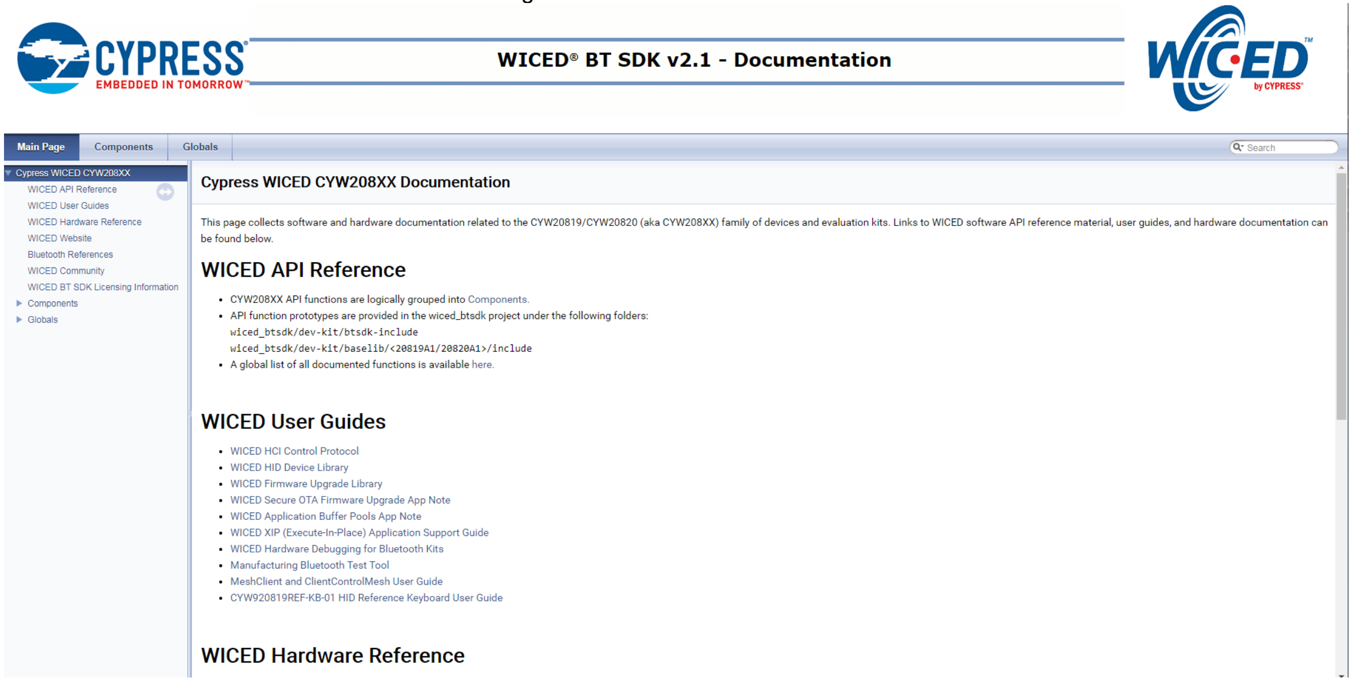
For more detailed information on the Eclipse IDE for ModusToolbox and how to create an application, see the Eclipse IDE for ModusToolbox User Guide.

4.1.3 SDK for ModusToolbox

SDKs provide the core of the ModusToolbox software. SDKs make it easier to develop firmware for supported devices without the need to understand all the intricacies of the device resources. An SDK contains configuration tools, drivers, libraries, and middleware as well as various utilities, Makefiles, and scripts. Application development for CYW20xxx silicon devices and EZ-BT modules based on these devices is enabled by the BT SDK within ModusToolbox. Important features of the BT SDK are listed below:

- Bluetooth Mesh stack
- BT stack and Profile-level APIs for embedded Bluetooth LE and Mesh application development
- APIs and drivers to access on-chip peripheral blocks such as SPI, UART, and ADC
- Bluetooth protocols supported include Generic Access Profile (GAP), Generic Attribute Profile (GATT), Security Manager Protocol (SMP), Radio Frequency Communication protocol (RFCOMM), Service Discovery Protocol (SDP), and BLE Mesh protocol
- BLE profile APIs, libraries, and sample applications
- Support for Over-The-Air (OTA) upgrade
- **SDK API Reference**: Choose **Help** > **ModusToolbox General Documentation** > **ModusToolbox Documentation Index**, search for Bluetooth Documentation and select the device you are working with such as CYW208xx. This HTML format guide gives you information on the SDK APIs related to the selected device. The API functions are logically grouped into components, as shown in Figure 6, for easy navigation and reference.

Figure 6. BT SDK API Reference



Cypress WICED CYW208XX Documentation

This page collects software and hardware documentation related to the CYW20819/CYW20820 (aka CYW208XX) family of devices and evaluation kits. Links to WICED software API reference material, user guides, and hardware documentation can be found below.

WICED API Reference

- CYW208XX API functions are logically grouped into Components.
- API function prototypes are provided in the wiced_btsdk project under the following folders:
  `wiced_btsdk/dev-kit/btsdk-include`
  `wiced_btsdk/dev-kit/baselib/<20819A1/20820A1>/include`
- A global list of all documented functions is available here.

WICED User Guides

- WICED HCI Control Protocol
- WICED HID Device Library
- WICED Firmware Upgrade Library
- WICED Secure OTA Firmware Upgrade App Note
- WICED Application Buffer Pools App Note
- WICED XIP (Execute-In-Place) Application Support Guide
- WICED Hardware Debugging for Bluetooth Kits
- Manufacturing Bluetooth Test Tool
- MeshClient and ClientControlMesh User Guide
- CYW920819REF-KB-01 HID Reference Keyboard User Guide

WICED Hardware Reference

Bluetooth Mesh API documentation is available on this page under **Components > Bluetooth > BLE Mesh**.

4.1.4 ModusToolbox Code Examples

ModusToolbox includes many code examples that are a great resource to understand the application code flow and to kickstart your application development. The code examples are grouped into fully functional demonstrations (Mesh-Demo group) and smaller code snippets demonstrate specific Mesh Models (Mesh-Snip group).

Fully functional demo code examples in the Mesh-Demo group compatible with the CYBT-213043-MESH EZ-BT Mesh Evaluation kit are:

- Color_light
- Dimmer

- Dimmer_self_config
- Embedded_provisionier
- Light_dimmable
- Light_smart
- Low_power_led
- On_off_switch
- Sensor_motion
- Sensor_temperature
- Switch_smart

Some of the code examples demonstrating specific Mesh Models in Mesh-Snip group compatible with the CYBT-213043-MESH EZ-BT Mesh Evaluation kit are:

- Mesh_battery_client
- Mesh_battery_server
- Mesh_level_client
- Mesh_level_server
- Mesh_light_hsl_server
- Mesh_light_hsl_client

Mesh code and snip examples are also available on GitHub. For more details, refer to Section 4.1.5

You can either browse the collection of starter applications during application creation through **File** > **New** > **ModusToolbox Application** or browse the collection of code examples on Cypress' GitHub repository.

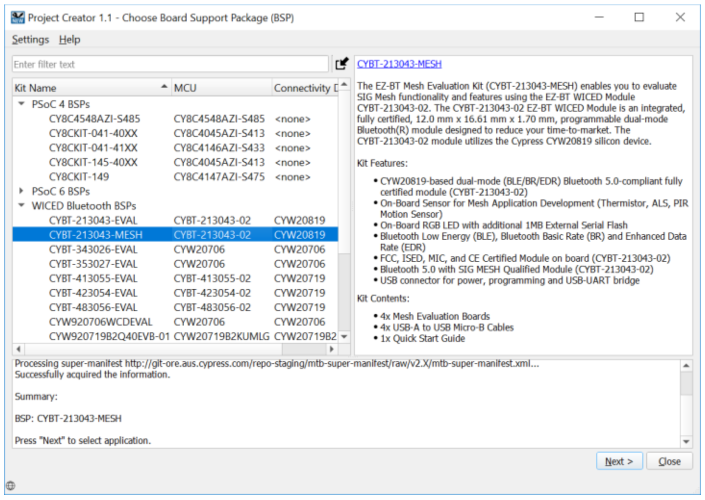
4.1.4.1 Creating Mesh Demo Code Examples

Ensure that you have already created the wiced_btsdk project before creating any other Mesh code example or application. Do the following to create the Mesh demo code example group from Eclipse IDE for ModusToolbox.

1. Select **File** > **New** > **ModusToolbox Application**.
2. Choose Board Support Package (BSP) as CYBT-213043-MESH as shown in Figure 7 and click **Next**.
3. Click on Mesh-Demo-213043MESH as shown in Figure 9 and click **Create**. Click **Close** after the code example download is complete to import the applications into the IDE.

The list of downloaded Mesh-Demo code examples can be seen in the Project Explorer window as shown in Figure 9. To learn how to create a project and program the EZ-BT Mesh evaluation board(s), refer to sections 8 and 9.

Figure 7. Board Support Package (BSP) Selection in Eclipse IDE for ModusToolbox



Kit Name	MCU	Connectivity D
PSoC 4 BSPs		
CY8C4548AZI-S485	CY8C4548AZI-S485	<none>
CY8CKIT-041-40XX	CY8C4045AZI-S413	<none>
CY8CKIT-041-41XX	CY8C4146AZI-S433	<none>
CY8CKIT-145-40XX	CY8C4045AZI-S413	<none>
CY8CKIT-149	CY8C4147AZI-S475	<none>
PSoC 6 BSPs		
WICED Bluetooth BSPs		
CYBT-213043-EVAL	CYBT-213043-02	CYW20819
CYBT-213043-MESH	CYBT-213043-02	CYW20819
CYBT-343026-EVAL	CYW20706	CYW20706
CYBT-353027-EVAL	CYW20706	CYW20706
CYBT-413055-EVAL	CYBT-413055-02	CYW20719
CYBT-423054-EVAL	CYBT-423054-02	CYW20719
CYBT-483056-EVAL	CYBT-483056-02	CYW20719
CYW920706WCDEVAL	CYW20706	CYW20706
CYW920719B2Q40EVB-01	CYW20719B2KUMLG	CYW20719B2

CYBT-213043-MESH

The EZ-BT Mesh Evaluation Kit (CYBT-213043-MESH) enables you to evaluate SIG Mesh functionality and features using the EZ-BT WICED Module CYBT-213043-02. The CYBT-213043-02 EZ-BT WICED Module is an integrated, fully certified, 12.0 mm x 16.61 mm x 1.70 mm, programmable dual-mode Bluetooth(R) module designed to reduce your time-to-market. The CYBT-213043-02 module utilizes the Cypress CYW20819 silicon device.

Kit Features:

- CYW20819-based dual-mode (BLE/BR/EDR) Bluetooth 5.0-compliant fully certified module (CYBT-213043-02)
- On-Board Sensor for Mesh Application Development (Thermistor, ALS, PIR Motion Sensor)
- On-Board RGB LED with additional 1MB External Serial Flash
- Bluetooth Low Energy (BLE), Bluetooth Basic Rate (BR) and Enhanced Data Rate (EDR)
- FCC, ISED, MIC, and CE Certified Module on board (CYBT-213043-02)
- Bluetooth 5.0 with SIG MESH Qualified Module (CYBT-213043-02)
- USB connector for power, programming and USB-UART bridge

Kit Contents:

- 4x Mesh Evaluation Boards
- 4x USB-A to USB Micro-B Cables
- 1x Quick Start Guide

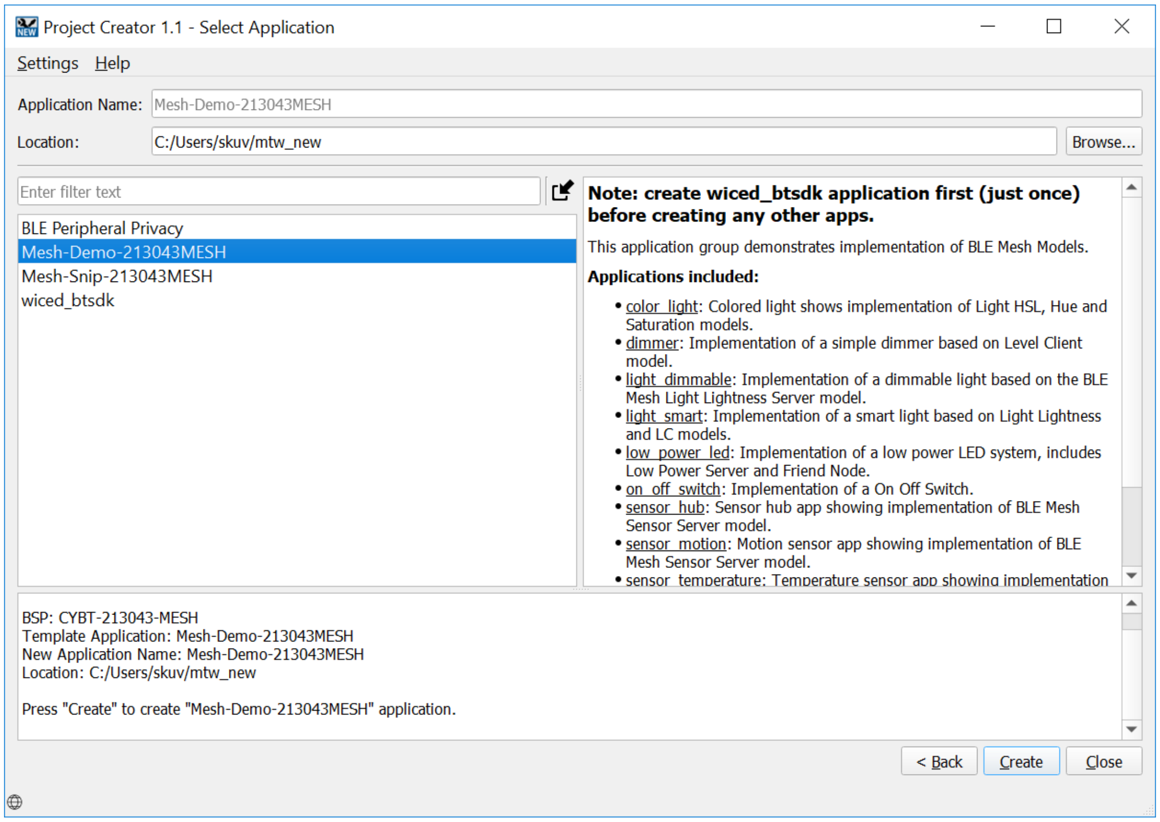
```
Processing super-manifest http://git-ore.aus.cypress.com/repo-staging/mtb-super-manifest/raw/v2.X/mtb-super-manifest.xml...
Successfully acquired the information.

Summary:

BSP: CYBT-213043-MESH

Press "Next" to select application.
```

Figure 8. Code Examples in Eclipse IDE for ModusToolbox



Project Creator 1.1 - Select Application

Settings Help

Application Name: Mesh-Demo-213043MESH

Location: C:/Users/skuv/mtw_new

Enter filter text

BLE Peripheral Privacy
Mesh-Demo-213043MESH
Mesh-Snip-213043MESH
wiced_btsdk

Note: create wiced_btsdk application first (just once) before creating any other apps.

This application group demonstrates implementation of BLE Mesh Models.

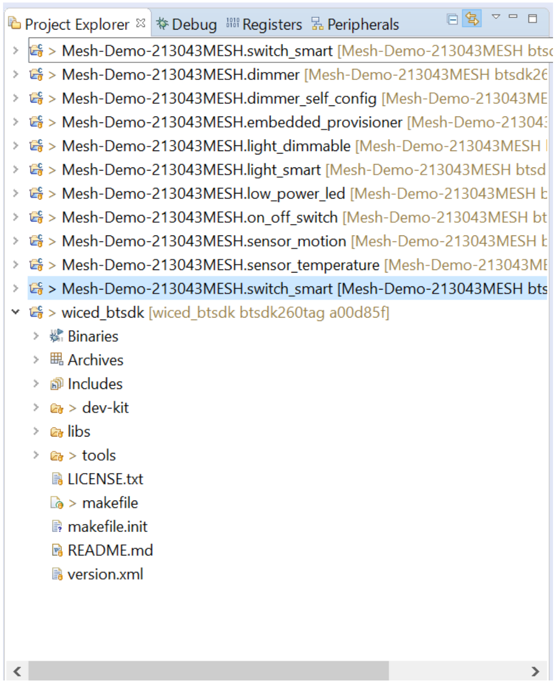
Applications included:

- color_light: Colored light shows implementation of Light HSL, Hue and Saturation models.
- dimmer: Implementation of a simple dimmer based on Level Client model.
- light_dimmable: Implementation of a dimmable light based on the BLE Mesh Light Lightness Server model.
- light_smart: Implementation of a smart light based on Light Lightness and LC models.
- low_power_led: Implementation of a low power LED system, includes Low Power Server and Friend Node.
- on_off_switch: Implementation of a On Off Switch.
- sensor_hub: Sensor hub app showing implementation of BLE Mesh Sensor Server model.
- sensor_motion: Motion sensor app showing implementation of BLE Mesh Sensor Server model.
- sensor_temperature: Temperature sensor app showing implementation

```
BSP: CYBT-213043-MESH
Template Application: Mesh-Demo-213043MESH
New Application Name: Mesh-Demo-213043MESH
Location: C:/Users/skuv/mtw_new

Press "Create" to create "Mesh-Demo-213043MESH" application.
```

Figure 9. Mesh-Demo Code Examples in Project Explorer window



```
> Mesh-Demo-213043MESH.switch_smart [Mesh-Demo-213043MESH btsc
> Mesh-Demo-213043MESH.dimmer [Mesh-Demo-213043MESH btsdk26
> Mesh-Demo-213043MESH.dimmer_self_config [Mesh-Demo-213043ME
> Mesh-Demo-213043MESH.embedded_provisioner [Mesh-Demo-21304
> Mesh-Demo-213043MESH.light_dimmable [Mesh-Demo-213043MESH
> Mesh-Demo-213043MESH.light_smart [Mesh-Demo-213043MESH btsd
> Mesh-Demo-213043MESH.low_power_led [Mesh-Demo-213043MESH b
> Mesh-Demo-213043MESH.on_off_switch [Mesh-Demo-213043MESH bt
> Mesh-Demo-213043MESH.sensor_motion [Mesh-Demo-213043MESH b
> Mesh-Demo-213043MESH.sensor_temperature [Mesh-Demo-213043MI
> Mesh-Demo-213043MESH.switch_smart [Mesh-Demo-213043MESH bts
> wiced_btsdk [wiced_btsdk btsdk260tag a00d85f]
    Binaries
    Archives
    Includes
    > dev-kit
    libs
    > tools
    LICENSE.txt
    > makefile
    makefile.init
    README.md
    version.xml
```

4.1.5 Code Examples on GitHub

Do one of the following:

- Click on the GitHub Mesh demo or GitHub Mesh snip repository link.
- Click the **Search Online for Code Examples** link in the **Quick Panel**.

4.1.6 Mesh-Related Application Settings

Help topics and resources provided in the previous section cover how to create an application using the IDE. This section covers Bluetooth Mesh-specific settings available in the makefile of the application projects that are needed when designing Bluetooth Mesh applications. To access these settings, go to the makefile of the desired Mesh application project (each Mesh application project has its own makefile).

Based on the application requirement, each of these settings can and may be changed.

BT_DEVICE_ADDRESS: This field allows the developer to set either a default Bluetooth device address (random 6-byte address), or a predefined Static Bluetooth Address. Bluetooth Mesh uses a Random Private Address (RPA) and the Mesh application code takes care of this for you. Therefore, for Mesh applications, this field should remain unchanged. This field is defined as “BT_DEVICE_ADDRESS?=default” in the makefile of Mesh applications; it means that a random Bluetooth device address is generated by default for Mesh applications.

UART: The default definition of this field is `UART?=AUTO` in the makefile of Mesh applications. This field allows you to select the UART to download the application. For example, 'COM6' on Windows or '/dev/ttyWICED_HCI_UART0' on Linux or '/dev/tty.usbserial-000154' on macOS. By default, the SDK will auto detect the port. This field is especially useful when multiple boards are connected to a computer and a specific port needs to be programmed.

ENABLE_DEBUG: This feature is disabled by default. This field is defined as `export ENABLE_DEBUG?=0` in the makefile of Mesh applications. To enable hardware debugging, set this field to '1'. To learn more about SWD debugging in CYW20819 and related modules, refer to the Hardware Debugging for CYW207xx and CYW208xx guide available at **Help** > **ModusToolbox General Documentation** > **ModusToolbox Documentation Index** > **Hardware Debugging for WICED devices**.

MESH_MODELS_DEBUG_TRACES: This option is disabled by default in the makefile (`MESH_MODELS_DEBUG_TRACES ?= 0`). When this field is enabled, it enables debug traces from the Mesh Models Library.

MESH_CORE_DEBUG_TRACES: This option is disabled by default in the makefile (`MESH_CORE_DEBUG_TRACES ?= 0`). When this field is enabled, it enables debug traces from the Mesh Core Library.

MESH_PROVISIONER_DEBUG_TRACES: This option is disabled by default (`MESH_PROVISIONER_DEBUG_TRACES ?= 0`) in the makefile. When this field is enabled, it enables debug traces from Mesh Provisioner Library.

Note: `MESH_MODEL_DEBUG_TRACES`, `MESH_CORE_DEBUG_TRACES`, and `MESH_PROVISIONER_DEBUG_TRACES` help to provide insights while debugging any Mesh-related issues. These flags enable traces in prebuilt Mesh libraries provided by Cypress. However, enabling these debug traces increases the application size and will limit the flash memory size available for user application code.

REMOTE_PROVISION_SRV: This feature is disabled by default (`REMOTE_PROVISION_SRV?=0`) in the makefile of Mesh applications. This field, when set to ‘1’, enables a device as a Remote Provisioning Server. It allows the provisioner to provision other unprovisioned devices through this node if a remote node is not directly reachable to the Provisioner. This is an in-development feature and is not a part of the Bluetooth SIG Mesh 1.0 specification. BT SDK version 2.x or later support this feature.

LOW_POWER_NODE: This field is disabled by default (`LOW_POWER_NODE ?= 0`) in the makefile of all the Mesh applications. When set to ‘1’, it enables the Low Power Node (LPN) feature on the device. Setting the value **in this field to ‘1’ will define the `LOW_POWER_NODE`** macro, which is used by the application to enable or disable low-power Mesh operation. Note that not all code examples that are provided in the BT SDK and on GitHub; use the `LOW_POWER_NODE` setting to enable low-power operation even though it is a global setting in the makefile. Also note that code examples that do use the `LOW_POWER_NODE` setting to enable low-power option does not enable it by default so, you must enable it in the makefile if low-power operation is desired for those nodes.

4.2 Helper Applications

Cypress provides Helper applications on Android, iOS, and Windows platforms that enable you to perform the following:

- Create and delete Mesh networks and groups
- Provision nodes
- Rename nodes
- Configure subscriptions
- Configure publications
- Publish Get and Set messages for lighting applications, including ON/OFF, level, Lightness, Lightness Hue Saturation, Lightness Color Temperature, Delta UV
- Get sensor states
- Set vendor data for vendor models
- Perform over-the-air firmware upgrade
- Reset a node to remove it from the network

In addition to these features, the Windows application provides a trace window that displays details on various activities.

4.2.1 iOS Helper App

The iOS Helper app allows you to use an iOS phone or tablet to act as the Mesh Provisioning Client. Cypress iOS MeshApp can be downloaded from Apple App Store using the QR code shown in Figure 10.

Figure 10. QR Code for iOS MeshApp

You can also use the Cypress-provided source code to customize the application based on the end product requirement and branding. See iOS Helper App Installation for instructions on building and downloading the source code using Apple MacBook.

Refer to Section 10 for instructions on how to use the Cypress iOS app and step-by-step details on testing the code examples covered in this application note.

4.2.2 Android Helper App

The Android Helper app allows you to use an Android device to act as the Mesh Provisioning Client. Cypress provides a pre-built application that can directly be installed on an Android device; in addition, it also provides the source code to enable you to customize the application based on your end product and branding.

OS requirements: Android version 7.1 or later

Install ModusToolbox and create the wiced_btsdk project as explained in Section 4.1.1. The installable Android Mesh App (.apk) and its source code are part of the BT SDK project.

- Android Mesh app (MeshController.apk): *<Workspace installation Folder>\wiced_btsdk\tools\btsdk-peer-apps-mesh\Android\src\bin\MeshController.apk*
- Android Mesh app source code: *<Workspace Folder>*\wiced_btsdk\tools\btsdk-peer-apps-*mesh\Android\src\MeshApp*

For the latest version of the Cypress Mesh Android app, download the source code from the GitHub repository.

To install the application on your Android device, copy the .apk file to your device. Then go to the device and browse to the .apk file from a file browser and tap to install it. Follow the on-screen instructions to install the application.

Refer to Using Android Helper App for instructions on using the Cypress Android app and step-by-step details on testing the example codes covered in this application note.

4.2.3 Windows Helper Applications

Cypress provides two different Windows Helper applications: **MeshClient** and **ClientControlMesh**.

Windows Helper applications are useful while developing Mesh applications as they provide ample details that can be useful while debugging applications.

The MeshClient and the ClientControlMesh applications provide a sample Windows implementation that show how to use interfaces exposed by Mesh libraries. MeshClient requires Windows 10 or higher. The MeshClient application uses the PC's built-in Bluetooth radio, or an external Bluetooth dongle to communicate with the Bluetooth Mesh network.

The MeshClient application implements all layers of the Mesh stack. The ClientControlMesh application implements only the application layer. It uses the Mesh Models and Mesh Core libraries residing on an embedded device, requiring a Cypress device (Mesh or other evaluation board) connected to the PC to act as a Client for Mesh operations. Any of the Cypress devices that support Bluetooth Mesh can be used for this application irrespective of the device used in the Mesh nodes. The ClientControlMesh application can be used with any version of Windows operating system. The `Mesh_provision_client` snip example must be programmed on a Cypress device so that it can operate as a Client to a Mesh network in conjunction with the ClientControlMesh application. The `mesh_provision_client` snip example can be downloaded from the GitHub repository. For detailed instructions on how to obtain snip examples and program the board, see Programing Code Examples.

See the MeshClient and ClientControlMesh App User Guide to find the application location, source code location, and instructions for using these apps. This user guide is also available as part of ModusToolbox installation and can be accessed from **Help** > **ModusToolbox General Documentation** > **ModusToolbox Documentation Index**.

5 CYBT-213043-02 EZ-BT Mesh Evaluation Kit

The EZ-BT Mesh Evaluation kit (CYBT-213043-MESH) enables the evaluation of the SIG Mesh functionality using the EZ-BT Bluetooth 5.0-qualified module CYBT-213043-02. The CYBT-213043-02s EZ-BT module is an integrated, fully-certified, 12.0 mm x 16.61 mm x 1.70 mm, programmable dual-mode Bluetooth (BLE/BR/EDR) module with BLE Mesh support.

The CYBT-213043-02 module utilizes the Cypress CYW20819 silicon device. CYW20819 is an ultra-low-power and highly integrated dual-mode Bluetooth 5.0-qualified device. CYW20819's low-power and integration capability addresses the requirements of both battery- and wall-powered applications that require BLE Mesh, such as sensor nodes, locks, lighting, blind controllers, asset tracking, and many more smart home applications.

CYBT-213043-MESH kit contains the following:

- 4x Mesh evaluation boards
- 4x USB-A to micro-B cable
- 1x Quick Start Guide

Figure 11 shows the top view of a BLE Mesh evaluation board and calls out the main components and connections available on the board. All four boards provided in the CYBT-213043-MESH Evaluation Kit are identical and offer the same features and functionality. The BLE Mesh evaluation board can be powered via the USB connector located on the top side of the board or can be optionally powered by a CR2032 coin cell battery using the coin cell battery holder on the bottom of the evaluation boards (Figure 12).

Figure 11. CYBT-213043-MESH Evaluation Board Top View

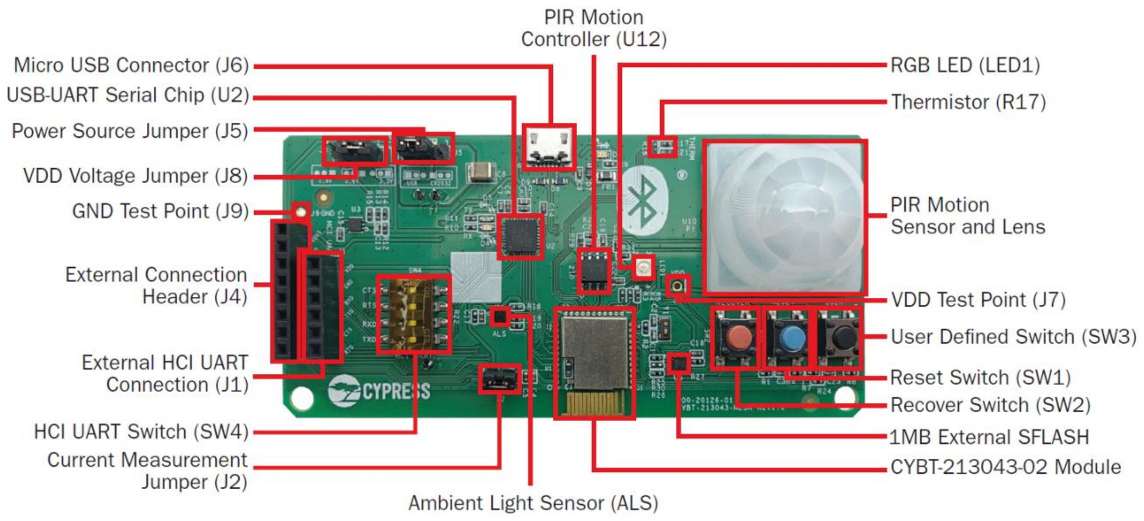
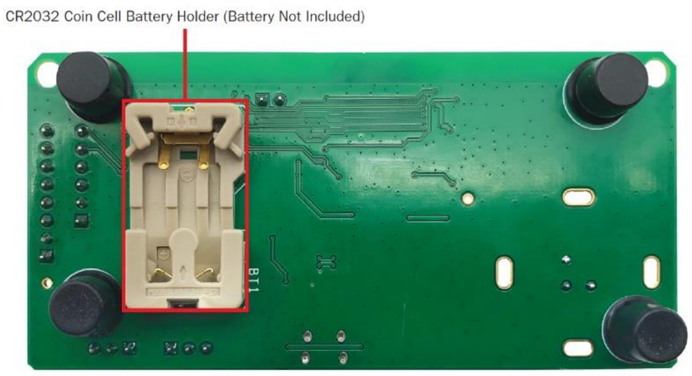


Figure 12. CYBT-213043-MESH Evaluation Board Bottom View



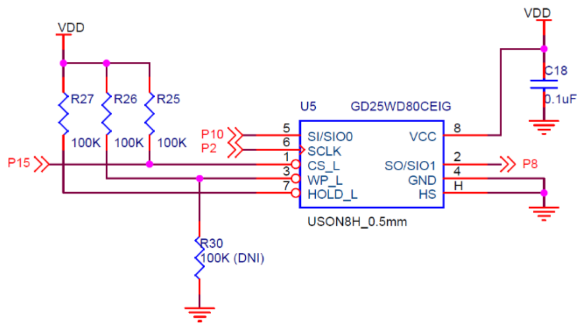
5.1 CYBT-213043-MESH Board Components

Each of the four EZ-BT Mesh evaluation boards includes several elements as described below.

5.1.1 Active Devices

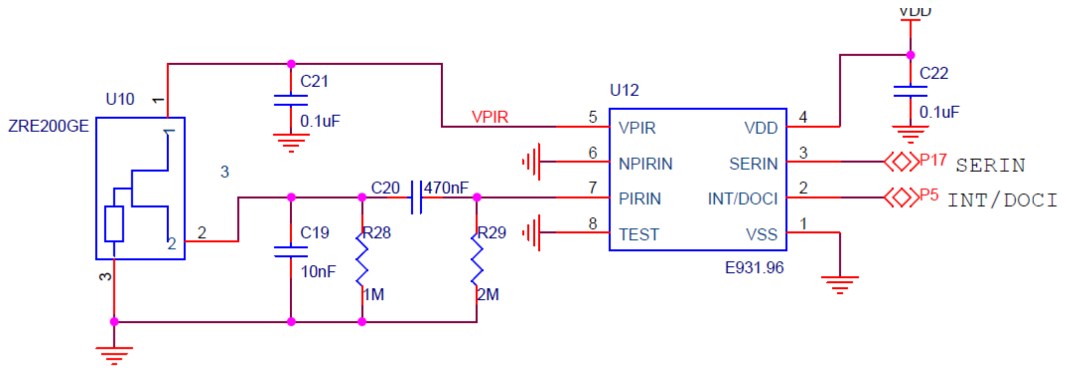
- **CYBT-213043-02 Module:** The CYBT-213043-02 EZ-BT WICED Module is an integrated, fully-certified (FCC, ISED, CE, and MIC), 12.0 mm x 16.61 mm x 1.70 mm, programmable dual-mode Bluetooth 5.0 module. This module is mounted to the Mesh evaluation board as shown in Figure 11. CYBT-213043-02 uses Cypress' CYW20819 silicon device, an ultra-low-power dual-mode Bluetooth 5.0 wireless MCU.
- **USB-to-UART Serial Chip (U2):** A USB-to-UART bridge device from Cypress is provided on the evaluation board to translate USB communication to UART communication (and vice-versa). For example, when HCI UART communication is configured to "On" on via SW4, the USB traffic from the host PC is connected to the HCI UART connections of the CYBT-213043-02 module. Additionally, the Peripheral UART (PUART) communication pins of the CYBT-213043-02 module are connected to another port of USB-to-UART bridge. This allows PUART communication to be used for either a user interface (command/control) or to print debug messages. Both UART (HCI UART and PUART) ports are available concurrently on the BLE Mesh evaluation boards.
- **1-MB external SFLASH (U5):** The Mesh evaluation board includes a 1-MB external serial flash device. This flash utilizes the SPI interface of the CYBT-213043-02 module. Solder pad P10 on the module is connected to MOSI, P8 is connected to MISO, P2 is connected to SCLK and P15 is connected to CS. This 1 MB flash can be used either for OTA upgrades or user data storage. The CYW20819 device includes 256 KB of on-chip flash memory. If your user application exceeds 50% of the available on-chip flash memory, the addition of external serial flash is provided to maintain OTA capability for complex applications.

Figure 13. External Flash Schematics



- **PIR motion controller (U12):** U12 is a low-power motion detector that processes the output of the on-board PIR sensor. This motion controller device allows the CYBT-213043-02 to remain in low-power mode while it constantly detects motion in the environment. When motion is detected, the PIR controller will interrupt the module using module solder pad P5.

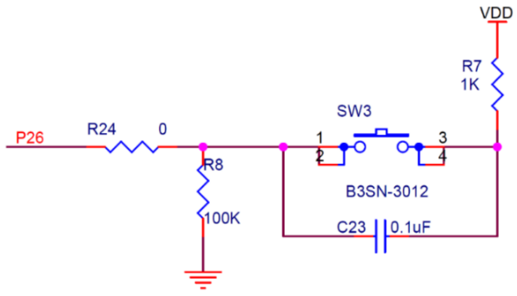
Figure 14. PIR Motion Controller and PIR Sensor Schematics



5.1.2 Switches and LEDs

- **RESET switch (SW1):** SW1 is a tactile switch that is connected directly to the XRES (External Reset) connection of the CYBT-213043-02 module. Activating (depressing) this switch will reset the EZ-BT WICED Module. This switch is also used if you need to recover the CYBT-213043-02 module to reprogram it.
- **RECOVER switch (SW2):** SW2 is provided to allow programming of a module that has the standard programming bootloader and/or application image corrupted or erased. It may also be necessary to allow programming a device that is in a low-power mode. While the kit has power applied, recovery mode can be entered by holding the SW2 button down and then pressing and releasing SW1 (RESET). SW2 is connected to the module's HCI UART CTS line (UART_CTS).
- **User defined Button (SW3):** SW3 is provided as a button that the user can configure as required. The SW3 element is connected to P26 on the CYBT-213043-02 module. The output of SW3 when pressed (activated) is high. Figure 15 shows the user defined button's circuit.

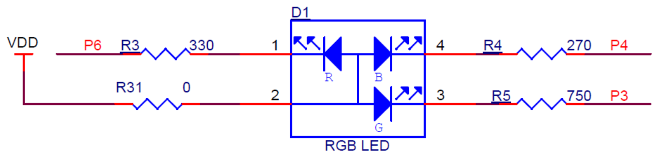
Figure 15. User Defined Switch Schematics



- **HCI UART switch (SW4):** SW4 controls the connection of USB data traffic to the HCI UART connections of the CYBT-213043-02 module. SW4 is a four-element switch (each element corresponding to one of the 4-wire UART signals – TX, RX, CTS, RTS) used to connect the HCI UART signals on the CYBT-213043-02 module to a host PC USB interface. To connect HCI UART signals from the CYBT-213043-02 module to the host PC, simply set each element of SW4 to the 'ON' position. To disconnect HCI UART communication from the CYBT-213043-02 module to the host PC, simply set each of the SW4 elements to the 'OFF' position. There is no High-Z configuration available on the SW4 elements; only On or Off positions. Flow control is supported in hardware on the HCI UART connection.
  - SW4 configuration is recommended to be set to the 'OFF' state if you need to use header J1 to connect an external microcontroller to the HCI UART connection of the CYBT-213043-02 module. Header J1 is provided as an optional HCI UART interface to the CYBT-213043-02 module.
  - SW4 configuration must be set to the 'ON' position if you need to program the CYBT-213043-02 module via a PC USB connection.
- **RGB LED (D1):** RGB LED is provided for user-configured behavior as required. RGB LED is an active LOW configuration, meaning that when the associated pin (R, G, or B) is pulled LOW, the LED will turn ON. The Red (R) element on the RGB LED is connected to P6 on the module, the Green (G) element is connected to P3 and the Blue (B) element is connected to P4.

  Note: Typical forward voltage for Green and Blue is 2.65 V and can be as high as 3.1 V. Therefore, the Green and Blue LED may not work at a power configuration setting of 1.8 V or while using a coin cell battery as the power source. The Red LED has typical forward voltage 1.8 V and maximum 2.1 V. So, it may not work on some boards at 1.8 V. If full RGB LED functionality is required in your application, use a 3.3 V or 3.6 V VDD configuration on jumper J8.

Figure 16. RGB LED Schematics

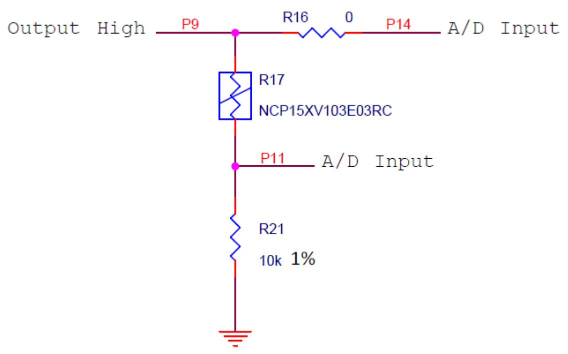


- D4 and D5: These two LEDs are provided to display programming or USB activity while in progress.
- D6: This LED is provided to show that USB power is applied to the Mesh evaluation board.

5.1.3 Sensors

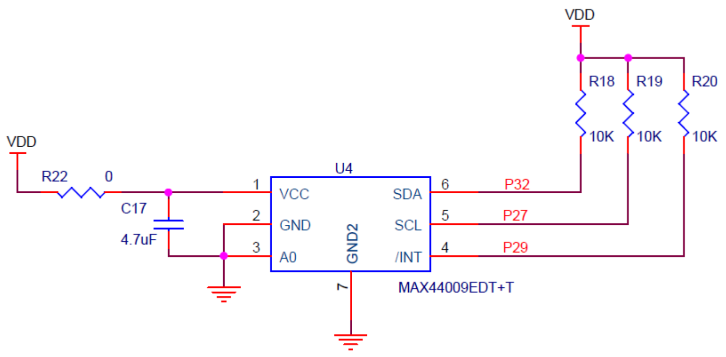
- **Thermistor (R17):** The Mesh evaluation board includes a 10-kΩ (resistance at 25 °C) NTC (negative temperature coefficient) thermistor. The thermistor is used in a typical resistor divider configuration with a 10-kΩ resistor at 1% accuracy (R21). Output of the resistor divider is connected to the CYBT-213043-02 module's A/D input on solder pad P11. The thermistor circuit is not excited all the time for the purpose of saving power. This circuit is excited using the CYBT-213043-02 module's pin P9. Your application will need to drive this pin High to provide power to this circuit. As there will be resistive loss in the GPIO driver, the output of P9 is connected to A/D input of the Cypress module on P14 for excitation voltage measurement via a 0-Ω resistor (R16).

Figure 17. Thermistor Schematics



- **Ambient Light Sensor (ALS) (U4):** U4 is an ambient light sensor that communicates to the Cypress module via I2C. The SDA line of sensor is connected to the module on P32 and the SCL line is connected on P27. The ALS sensor also provides an active LOW interrupt to indicate a change in Ambient Light of the environment. This line is connected to the Cypress module on P29. The I2C address for this sensor is 0x1001 010x.

Figure 18. Ambient Light Sensor (ALS) Schematics



- **PIR motion sensor (U10):** U10 is a Pyroelectric/passive infrared (PIR) motion sensor. This is an analog sensor and is connected to the PIR motion controller device (U12) on the Mesh evaluation board. The PIR motion controller is connected to the module and alerts the module whenever motion is detected.

Note: Minimum operating voltage for the PIR sensor controller is 2.5 V. So, it may not work at 1.8 V VDD setting using J8. Use 3.3 V or 3.6 V setting for reliable operation.

5.1.4 Connectors and Headers

- **CR2032 coin cell battery holder (BT1):** Use a CR2032 coin cell battery (not included with the kit) to power the Mesh evaluation board.

  Note: When operating from of a coin cell battery, the HCI-UART connection switch (SW4) should have all four elements set to the OFF position (disconnecting the HCI-UART connection). Peripheral UART (PUART) should also be disabled in the user application code to remove current consumption leakage that will occur between the on-board USB-to-UART bridge device and the Cypress CYBT-213043-02 EZ-BT module.

- **Micro-USB connector (J6):** The USB connection on the CYBT-213043-MESH evaluation board provides power to the evaluation board and also provides communication to the board via USB, which is translated to UART communication and routed to the HCI UART (depending on the configuration of SW4) and PUART connection on the CYBT-213043-02 Module on CYBT-213043-Mesh evaluation board.

- **Power source jumper (J5):** Use J5 to select one of the two power sources – coin cell (BT1) or USB power (J6). Table 4 provides the selection options for J5.

Table 4. J5 Jumper Power Source Selection Options

J5 Jumper Configuration	VDD Voltage Level
Short 1 & 2	USB Power (Default)
Short 2 & 3	Coin cell

- **Power voltage jumper (J8):** If USB power is configured as the power source in jumper J5, then J8, a 3-pin jumper, is used to configure the input to the module and on-board peripherals to either 1.8 V, 3.3 V or 3.6 V based on the user configuration. The power supply input is configured by shorting header positions or leaving them open on J8.
- Table 5 details the available power supply options and the associated header position connections required.

Table 5. J8 Header Power Supply Connection Options

J8 Jumper Configuration	VDD Voltage Level
Short 1 & 2	3.3 V (Default)
Short 2 & 3	3.6 V
Open 1 & 2 & 3	1.8 V

- **External HCI UART connection header (J1):** The J1 header provides all HCI UART communication lines to the user. This allows you to connect the CYBT-213043-02 Module to a host controller directly without having to connect through USB. The J1 direct HCI UART connection bypasses the SW4 (HCI UART to USB-to-UART bridge to Host PC) configuration. It is recommended that SW4 elements 1 ~ 4 are placed in the OFF position if the J1 direct HCI UART connection is used.
- **Power consumption measurement header (J2):** J2 is provided to allow for power consumption measurement with a multimeter or current measurement probe.
- **External connection header (J4):** J4 header provides access to 5V_VBUS (USB power) on the Mesh evaluation board. Additionally, J4 provides access to several module connections, including GND, VDD, HOST_WAKE, DEV_WAKE, XRES, P12 and P13. To use Hardware debug on the Mesh evaluation board, P12, P13 along with the GND connection on the J4 header can be used to connect to an external SWD debugger to the board.
- **GND test point (J9):** J9 test point provides access to GND on the Mesh evaluation board that can be used either for probing or connecting an external power source.
- **VDD test point (J7):** J7 test point provides access to VDD on the Mesh evaluation board that can be used either for probing or connecting an external power source.

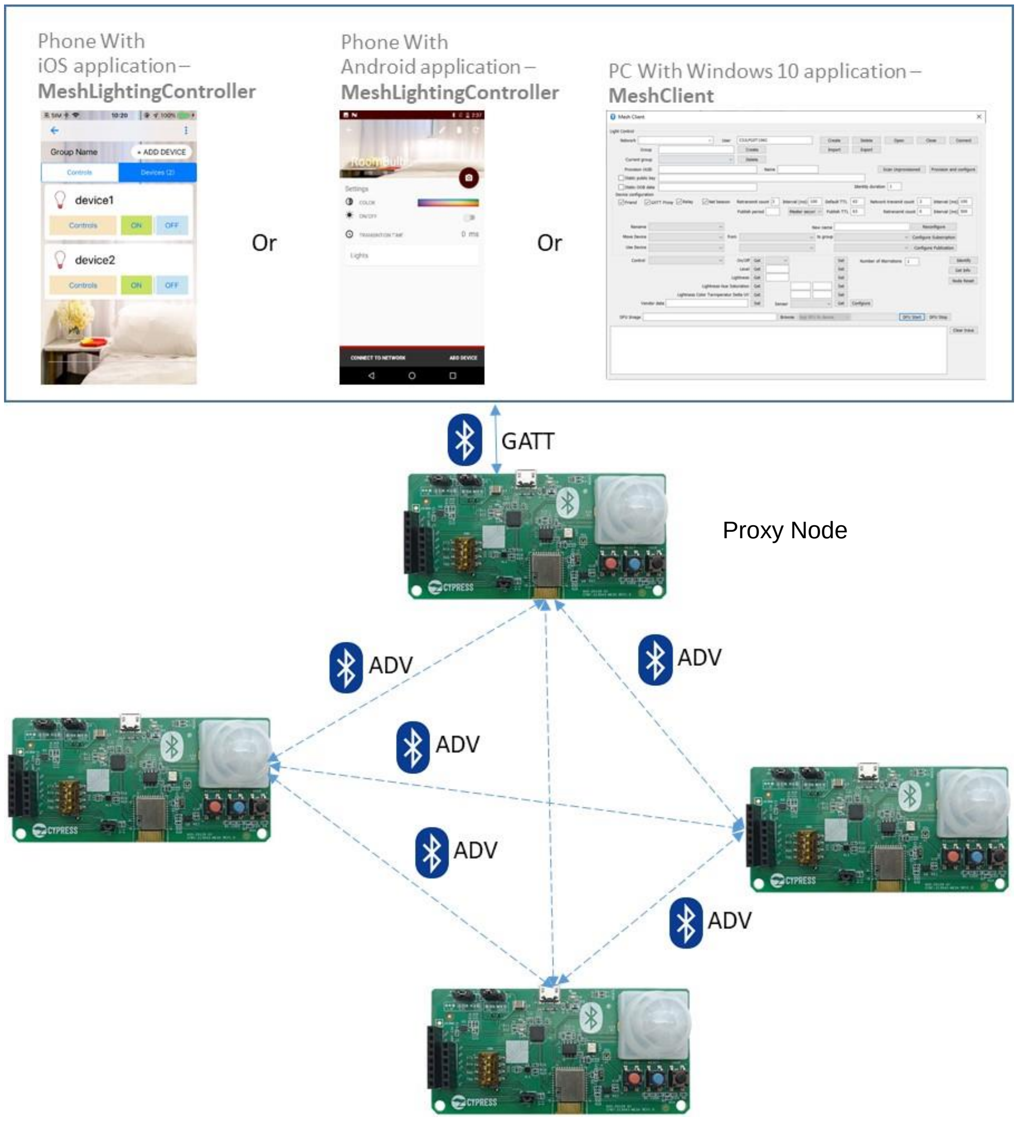
5.1.5 Programming the CYBT-213043-Mesh Evaluation Board

See Section 9 for information on how to program the BLE Mesh evaluation boards.

6 Bluetooth Mesh Development Setup

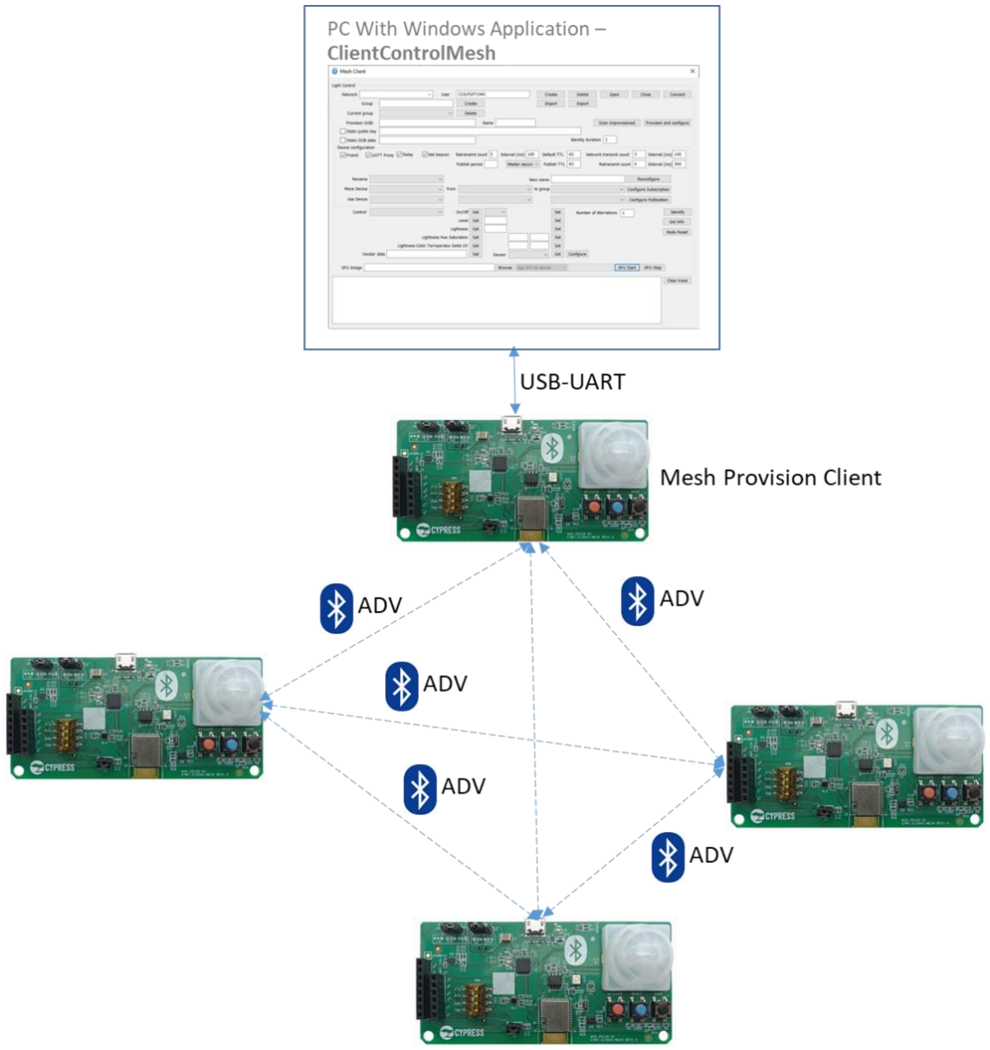
Figure 19 shows the hardware and software required for evaluating a Bluetooth Mesh design. CYBT-213043-MESH evaluation boards have a CYBT-213043-02 module that can talk to other Mesh boards/modules over an advertisement bearer and any of these boards can talk to a mobile phone running iOS or Android or even a PC running Windows 10. A Mesh device that talks directly to a PC or mobile phone is called a Proxy node. Communication between a Proxy node and a mobile phone/PC is over a GATT bearer; the Proxy then relays the information to/from the Mesh network over the Advertisement bearer. Each board needs to be programmed for the required Mesh functionality using ModusToolbox to work in a Mesh network.

Figure 19. Bluetooth Mesh Evaluation Setup With iOS/Android/Windows 10-based Devices with MeshClient Application


Phone With iOS application – MeshLightingController
Or
Phone With Android application – MeshLightingController
Or
PC With Windows 10 application – MeshClient
GATT
Proxy Node
ADV
ADV
ADV
ADV
ADV

If you are using the Windows ClientControlMesh application, one CYBT-213043-MESH board must be programmed as the Mesh Provisioning Client, because the ClientControlMesh Helper Application does not use the inbuilt Bluetooth communication on the Windows PC. Figure 20 shows the setup while using the ClientControlMesh application.

Figure 20. Bluetooth Mesh Evaluation Setup for Windows 10-based Devices Using ClientControlMesh Application



The Testing the Code Examples section of this application note will walk you through the steps on how create and test your Mesh network.

7 Cypress Bluetooth SIG Mesh Architecture Overview

This section describes the following details of the Cypress Mesh solution:

1. Overview of the Cypress Mesh implementation. Jump to Section 7.1.
2. Understand the User Application Code Flow. Jump to Section 7.2.

Within this section, the term "user application code" refers to the application code developed by you, the user, to define your specific Mesh application (i.e., this term does not refer to Cypress pre-built libraries). This code is analogous to the code that will be walked through in Section 8.

If you are already familiar with the Cypress Mesh architecture, libraries, and application code flow, you can skip to Section 8 to walk through how to design Mesh applications.

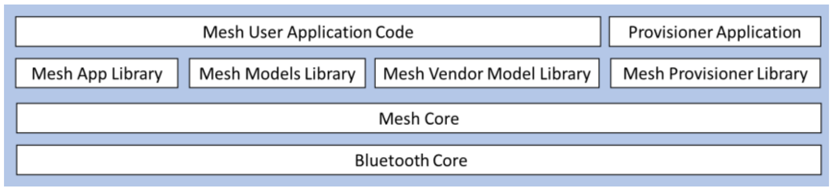
For additional details on the Cypress Mesh Application library, see Understanding the Flow of the Cypress Mesh Application Library in the appendix.

7.1 Cypress Mesh Implementation

Figure 21 shows the Bluetooth SIG Mesh architecture implemented in the BT-SDK. The Mesh Core library, Mesh Models library and Mesh App Library implementation does all the heavy lifting to be interoperable with a Bluetooth SIG Mesh network. The Mesh user application, as covered in code examples, just needs to define the Mesh configuration, hardware-specific configuration, and take necessary action when callbacks are received from Mesh app and Mesh model libraries. The BT-SDK also allows the developer to create vendor-specific Mesh models using the Mesh Vendor Model library. The Mesh Provisioner Library and Provisioner application help to implement a provisioner that creates a Mesh network, and provisions and configures new devices.

The following subsections provide details on each library.

Figure 21. Cypress Bluetooth SIG Mesh Architecture



7.1.1 Mesh User Application Code

The Mesh User Application code (code example or user application code) does not contain a Mesh-specific logic implementation. The Mesh User Application is the actual user application that defines the hardware actions and provides the methods for the application to exchange messages with other devices in the network. Cypress provides a number of Mesh code examples as part of BT-SDK as well as on GitHub for evaluating the Mesh functionality and to serve as a basis for your Mesh application development. Each code example provides a simple way to realize a specific functionality.

BT-SDK and GitHub code examples can be put in two different buckets:

7.1.1.1 Server or Client Mesh Code Examples

Server code examples leverage Mesh Server models. These code examples expose the states of the device to a Mesh Client. For example, the Dimmable Light code example uses the Light Lightness Server model and exposes the Server Model's states to a Client. Based on messages received from the Client, the application code in the Dimmable Light example modifies the LED state.

Client code examples leverage the Mesh Client models. Mesh Client code examples send commands to the network to change or read one or more Sever node's states. For example, the Dimmer code example uses the Generic Level Client Model to change the level in the Dimmable Light example. The application code for the Dimmer example monitors the user button state and accordingly sends a set command message to the Mesh network.

7.1.1.2 Provisioner Mesh Code Example

The Mesh Provisioner application code example uses the Mesh Provisioner library to create a Mesh network, and to provision and configure Mesh nodes. To enable the features of this code example, connect the evaluation board programmed with the Provisioner Mesh code example to the PC. Use ClientControlMesh Windows application on the PC to create a Mesh network, provision Mesh devices, and control Mesh nodes.

7.1.2 Mesh Application Library

The Mesh Application library does not implement any core Mesh functionality. It can be considered as a subset of the application layer with a Mesh Application code example as its counterpart. This library implements state machines and event handlers for common BLE Mesh functionality. The Mesh Application library abstracts the complexity for the user application code. It provides callback functions to the Mesh Application code (user application) to allow the user to implement hardware-level and application-based functionality. The Mesh Application library takes care of initializing the Mesh Core libraries and registers appropriate callback functions. Most application settings and interactions with the Mesh Core library are taken care of by the Mesh Application library.

7.1.3 Mesh Models Library and Vendor Model Library

The Mesh Models Library contains the implementation of all standard models defined in the Mesh Models specification. The library keeps the information about device states and provides a logical interface for the Mesh Application and Mesh Application library to control or send the status information to the peer device. It translates the inputs from the Mesh Application and converts them into appropriate Mesh messages. The Mesh Models library receives, parses, and validates the messages received over the air and sends appropriate events to the Mesh application.

The developer has an option to provide vendor-specific models not defined by the Mesh Models specification. In that case, Vendor-Specific Models will access the Mesh Core library interface directly. The interface between the application and the Mesh Vendor Model library is implementation-specific.

7.1.4 Mesh Provisioner Library

The Mesh Provisioner library contains the implementation of a provisioner as defined by the BLE Mesh Networking Specification. The Mesh Provisioner application code examples access the Mesh Provisioner library to create a new network, and to provision and configure Mesh devices.

7.1.5 Mesh Core Library

The Mesh Core library implements the core Mesh functionality from the Bluetooth Mesh profile as defined in the BLE Mesh Networking Specification and provides communication between devices in a Bluetooth Mesh network. The Provisioner assigns the device address and provides the security keys to the device. On the device side, everything is managed by the Mesh Core library. When the Mesh application code receives a callback to perform an action, the message is already validated to have the required permissions and the data is already decrypted by the Mesh Core library. Similarly, when the Mesh application needs to send the data out, the Mesh Core library encrypts the message with the keys provided during provisioning and sends it over the Bluetooth interface.

7.2 User Application Code Flow

The Mesh Application Library takes care of the interaction with the Mesh Models library and the Mesh Core library. The Mesh Application library sends relevant events and callback functions to the user application code to implement the application and hardware-specific functionality. Additional details on the Mesh Application Library can be found in Understanding the Flow of the Cypress Mesh Application Library.

The user application code is a specific implementation for every application design; a typical flow for user application code is as follows according to the design requirement:

1. Set up the Mesh Core configuration structure. This configuration defines the company ID, product ID, vendor ID, number of elements, and array of elements defined for the Mesh device. It also defines features supported by the Mesh device such as friend, low power node, relay, and proxy, and configuration parameters for friend and low power nodes. An example of a Mesh Core configuration is shown here:

```
wiced_bt_mesh_core_config_t  mesh_config =
{
    .company_id         = MESH_COMPANY_ID_CYPRESS,
    .product_id         = MESH_PID,
    .vendor_id          = MESH_VID,
    .replay_cache_size  = MESH_CACHE_REPLAY_SIZE,
    .features           = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND |
WICED_BT_MESH_CORE_FEATURE_BIT_RELAY |
                                      WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER,
    .friend_cfg         =
    {
        .receive_window        = 20,
        .cache_buf_len         = 300,
        .max_lpn_num           = 4
    },
    .low_power          =
    {
        .rssi_factor           = 0,
        .receive_window_factor = 0
        .min_cache_size_log    = 0,
        .receive_delay         = 0,
        .poll_timeout          = 0
    },
    .gatt_client_only          = WICED_FALSE,
    .elements_num  = (uint8_t)(sizeof(mesh_elements) / sizeof(mesh_elements[0])),
    .elements      = mesh_elements
     };
```

2. Set up the configuration structure for each Mesh element in the Mesh device. You can add multiple Mesh elements based on your design requirement. The parameters in this configuration structure define the element's location, default transition time, power up state, range, element properties, number of Mesh models, and array of models defined for each element in the Mesh device. An example of a Mesh element configuration is shown here:

```
wiced_bt_mesh_core_config_element_t mesh_elements[] =
{
    {
        .location = MESH_ELEM_LOC_MAIN,
        .default_transition_time = MESH_DEFAULT_TRANSITION_TIME_IN_MS,
        .onpowerup_state = WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE,
        .default_level = 0,
        .range_min = 1,
        .range_max = 0xffff,
        .move_rollover = 0,
        .properties_num = MESH_APP_NUM_PROPERTIES,
        .properties = mesh_element1_properties,
        .sensors_num = 0,
        .sensors = NULL,
        .models_num = MESH_APP_NUM_MODELS,
        .models = mesh_element1_models,
    },
};
```

3. Declare the Mesh models required. Multiple Mesh Models can be defined for each element. Every element should define its own set of Mesh models. It is mandatory to include the WICED_BT_MESH_DEVICE model in the primary element. The primary element may also include other Mesh models based on design requirement which help to achieve the desired functionality. Secondary elements (if there are any) should not include the WICED_BT_MESH_DEVICE model but can include other Mesh models based on user requirements. An example of Mesh model configuration is shown here:

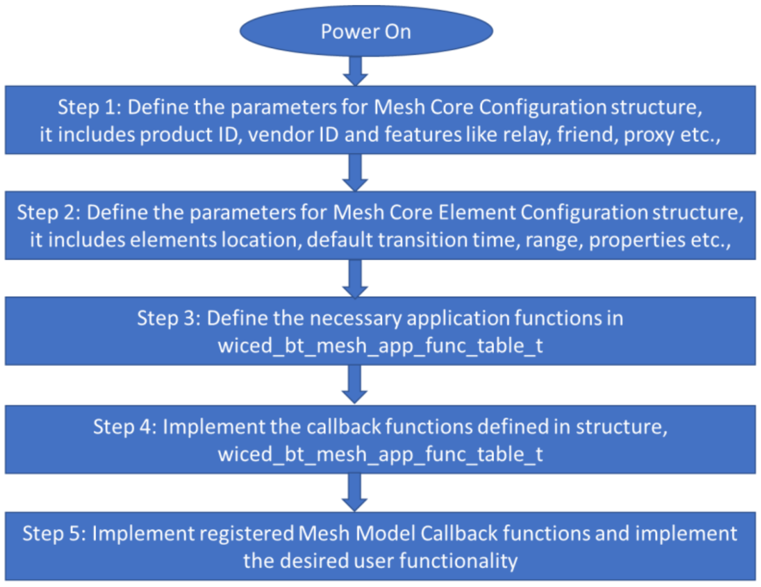
```
wiced_bt_mesh_core_config_model_t   mesh_element1_models[] =
{
    WICED_BT_MESH_DEVICE,
    WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER,
    WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER,
};
```

4. Register its own callback functions with the Mesh Application library; for example, the Dimmer Switch application needs to register a hardware initialization function in the `wiced_bt_mesh_app_func_table_t` structure to configure a button GPIO based on the hardware schematic.

5. Implement the callback functions registered in the previous step based on the product requirements. Also implement the functions to control the element(s) of the Mesh device based on the messages received from the Mesh Model library. For example, a Dimmable Light Bulb device must control the intensity of the LED by modifying the PWM parameters based on the messages received from the Mesh Models library.

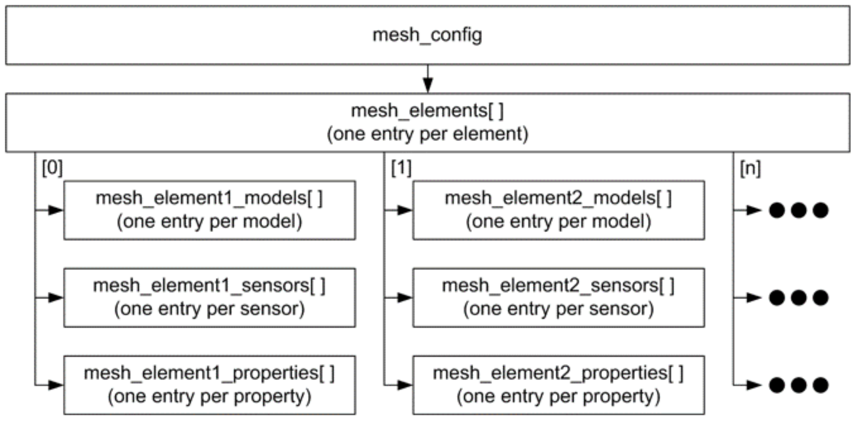
The typical flow of the user application code is broadly divided in to five steps as shown in Figure 22.

Figure 22. Typical Mesh User Application Code Flow


Power On
Step 1: Define the parameters for Mesh Core Configuration structure, it includes product ID, vendor ID and features like relay, friend, proxy etc.,
Step 2: Define the parameters for Mesh Core Element Configuration structure, it includes elements location, default transition time, range, properties etc.,
Step 3: Define the necessary application functions in wiced_bt_mesh_app_func_table_t
Step 4: Implement the callback functions defined in structure, wiced_bt_mesh_app_func_table_t
Step 5: Implement registered Mesh Model Callback functions and implement the desired user functionality

Before we get into the details of the steps that are part of developing the Mesh user application code, let us get familiar with the configuration variables that are needed for a Mesh device. Figure 23 illustrates how the different variables interact with each other to configure the mesh device. The following section provides details regarding each parameter that is part of Figure 23.

Figure 23. Typical Mesh Device Configuration Variables



7.2.1 Step 1: Define Mesh Core Configuration

The device configuration structure for the Mesh device, `wiced_bt_mesh_core_config_t`, defines the following parameters/features:

7.2.1.1 Mesh Company ID

The Bluetooth SIG defined company identifier should be assigned to this field. In this case, the Cypress company identifier is assigned (0x0131).

7.2.1.2 Mesh PID

This field is a vendor-assigned product identifier, which is 16 bits in length. Typically, each product group will have a unique value assigned to this field; for example, in the code examples, all Dimmable Light devices are assigned with a value “0x311F” and all Dimmer Switch devices are assigned with a value “0x3005”.

7.2.1.3 Mesh VID

This field is a vendor-assigned product version identifier, which is 16 bits in length. For all the code examples from Cypress, this value is defined as “0x0001”.

7.2.1.4 Maximum Size of Replay Protection List

This is determined by the maximum number of Mesh devices in a network which can send messages to this device. A value of ‘8’ is assigned by default for all the code examples which are part of the BT SDK and GitHub. The maximum size of the replay protection list is determined by the amount of available RAM space in your design. We have configured a value of ‘8’ corresponding to a typical use case (e.g., a dimmable light bulb node controlled by 3-5 mobile phones, and 2-3 dimmer/sensor nodes).

If the maximum number of devices is exceeded, then the node will ignore messages from Clients that are not already contained in the list of 8, until the list is cleared. If you have reached the maximum number of devices in the Replay Protection List, the node will automatically initiate an IV (Initialization Vector) index update procedure, which clears Replay Protection List, updates the index and allows new devices to be added to the list. An IV index update procedure can only occur once in a 96-hour period.

7.2.1.5 Features Supported by the Mesh Device

These include Friend, Relay, Proxy server, Low Power Node, Bearer support configuration (Advertisement Bearer or GATT Bearer).

You can enable specific features by modifying appropriate bits in the ‘features’ parameters as required. For example, use the following to enable the Friend and relay features (if required):

```
.features = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY
```

7.2.1.6 Friend Node and LPN Configuration

Parameters for the Friend node and LPN are defined in the structures `wiced_bt_mesh_core_config_t.friend_cfg` and `wiced_bt_mesh_core_config_t.low_power`. These structures are described in Sections 7.2.1.6.1 and 7.2.1.6.2.

7.2.1.6.1 wiced_bt_mesh_core_config_t.friend_cfg

The `wiced_bt_mesh_core_config_t.friend_cfg` parameters define the supported Friend features when a Mesh device is configured as a Friend node. These include the following:

- `.receive_window`: The time that the LPN listens for a response. A friend node should respond to the LPN request before the `receive_window` timeout. The LPN will stop listening after it receives a message from its Friend node. See Figure 24. The `receive_window` is configured in steps of milliseconds. A higher value for the `receive_window` provides additional time for the Friend node to process and respond to the LPN request, but this means that the LPN will consume more power as it continuously listens for a Friend response during this period. A lower value of `receive_window` requires the Friend node to respond quickly which saves power at the LPN as it can listen for a shorter time. A Friend node sets this parameter based on the processing time needed to respond to the LPN request. A value of 20 ms is assigned to all the code examples which are part of the BT SDK and GitHub.
- `.cache_buf_len:` The cache buffer length is the number of Bytes allocated for all the supported LPNs. The maximum value for this parameter depends on the free RAM available in the design. The default setting for this parameter in all code examples which support the Friend feature is 300 Bytes. Each Mesh message is 34 bytes long, which means that the code examples that support the Friend feature can store up to 8 Mesh messages. You can increase the value of this parameter based on the application needs and free RAM available in the device.
- `.max_lpn_num:` This parameter defines the maximum number of LPNs supported by the Friend node. The default value of this parameter for all code examples which support Friend feature is ‘4’. This value must be greater than ‘0’ if the node supports the Friend feature. The maximum value for this parameter depends on the free RAM available for the design.

7.2.1.6.2 wiced_bt_mesh_core_config_t.low_power

The `wiced_bt_mesh_core_config_t.low_power` parameters define the timing and cache size required by the LPN. This includes the following:

- `.rssi_factor`: This parameter defines the contribution of the RSSI measured by the Friend node used in the Friend Offer Delay calculations. This field is 2 bits in length and has a valid range between 1 to 2.5 in steps of 0.5. A low value of this parameter reduces the importance of RSSI contribution during Friend Offer Delay calculations. If the LPN wants to increase the significance of RSSI value, then this parameter should be set to the maximum value (3). This means that a Friend node with best RSSI sends the Friend offer the earliest. The default value of this parameter for all code examples which support LPN feature is ‘2’.

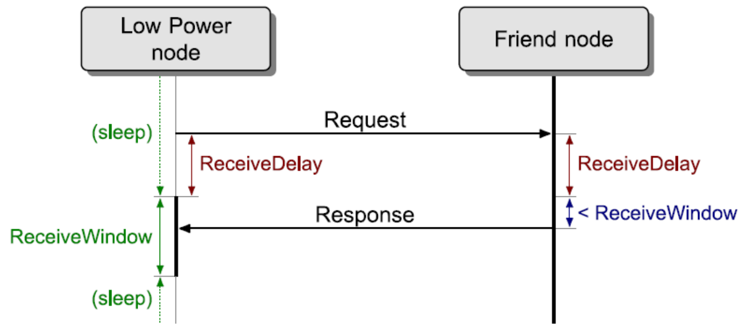
  Delay to send Friend offer = ReceiveWindowFactor * ReceiveWindow - RSSIFactor * RSSI

- `.receive_window_factor`: This parameter defines the contribution of the supported `receive_window` parameter used in Friend Offer Delay calculations. This field is 2 bits in length and has a valid range between 1 to 2.5 in steps of 0.5. If the LPN wants to decrease the significance of `receive_window`, then this parameter should be set to the maximum value (3). This means that a Friend node with smallest `receive_window` sends the Friend offer the earliest. The default value of this parameter for all code examples which support LPN feature is ‘2’.

  Delay to send Friend offer = ReceiveWindowFactor * ReceiveWindow - RSSIFactor * RSSI

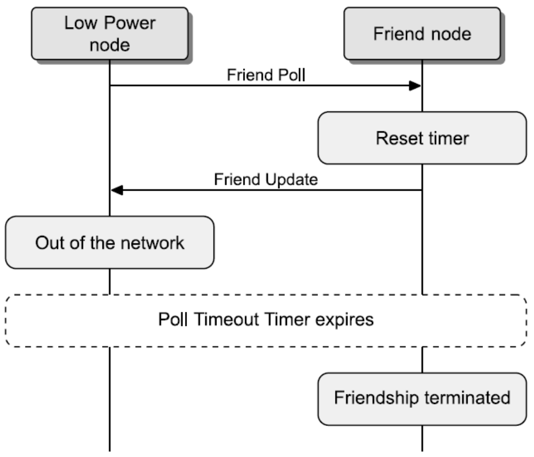
- `.min_cache_size_log`: This parameter defines the minimum number of messages that a Friend node must be able to store in its cache. The default value of this parameter for all code examples is ‘3’, indicating that the Friend node needs to store eight messages (2 to the power of 3) for the LPN node. The valid values for this parameter are 1 to 7, which means that an LPN can request from 2 to 128 messages.
- `.receive_delay`: This parameter defines the receive delay requested by the LPN. The `receive_delay` is the time between the LPN sending a request and listening for a response. This delay allows the Friend node time to prepare the response. See Figure 24. The value for this parameter can be set in steps of 1 millisecond. A lower value for this parameter provides less time for the Friend node to respond to an LPN request.

Figure 24. LPN and Friend Node Communication Timing (Bluetooth SIG, 2019)



- `.poll_timeout`: The `poll_timeout` timer is used to measure the time between two consecutive requests sent by the LPN. Friendship is considered terminated if the Friend does not receive a Friend poll from the LPN within the `PollTimeout` as shown in Figure 25. If a Friendship is terminated, the Friend node is aware of the termination, but the LPN is not informed of the termination. If this happens, the LPN will attempt to send a poll request to its Friend six times. After six unsuccessful attempts, the LPN will then terminate its Friendship with its Friend. It will then begin searching for Friend nodes by sending the Friend Request messages.

Figure 25. Poll Timeout and Friendship Termination (Bluetooth SIG, 2019)



To understand the parameters of Friend node and LPN, consider how an LPN establishes friendship with a Friend node.

The LPN node sends the RSSI Factor and RECEIVE WINDOW FACTOR to the adjacent Friend nodes as part of the Friend Request. Friend nodes receive the Friend Request and calculate a local delay using following formula:

*Local Delay = ReceiveWindowFactor * ReceiveWindow - RSSIFactor * RSSI*

ReceiveWindowFactor is a number from the Friend Request message sent by LPN.

ReceiveWindow is the value to be sent in the corresponding Friend Offer message by the Friend Node.

RSSIFactor is a value from the Friend Request message sent by LPN.

RSSI is the received signal strength of the received Friend Request message (sent by LPN) at the Friend node.

Potential Friend devices who receive the Friend Request calculate their Local Delay based on the received parameters and its own Receive Window parameter. If the calculated Local Delay is greater than 100, then the Friend Offer message is sent after the calculated Local Delay value in milliseconds. If the calculated Local Delay is less than 100, the Friend Offer message is sent after 100 milliseconds.

This means that the LPN receives Friend offers quicker from nodes that more closely match the LPN requirements, reducing the power consumed by the LPN when initially searching for a Friend node.

7.2.1.6.3 *Friend and Low-Power Configuration Example*

Consider the following the examples:

Case 1:

Consider an LPN and two Friend nodes with the following parameters:

Table 6. Case 1 Example of LPN Friendship Establishment

	LPN	Friend 1	Friend 2
RSSI Factor	2	–	–
Receive Window Factor	2	–	–
Receive Window	–	200	300
Measured RSSI by Friend	–	90 dBm	40 dBm

In this case:

- Friend 1 calculates the Local Delay as 220 (Local Delay = 2*200 – 2*90)
- Friend 2 calculates the Local Delay as 520 (Local Delay = 2*300 – 2*40)

The LPN will receive a Friend offer from Friend 1 and Friend 2 after 220 ms and 520 ms respectively.

Case 2:

If we have one LPN and two Friend nodes with below parameters

Table 7. Case 2 Example of LPN Friendship Establishment

	LPN	Friend 1	Friend 2
RSSI Factor	2	–	–
Receive Window Factor	2	–	–
Receive Window	–	200	100
Measured RSSI by Friend	–	90 dBm	40 dBm

In this case:

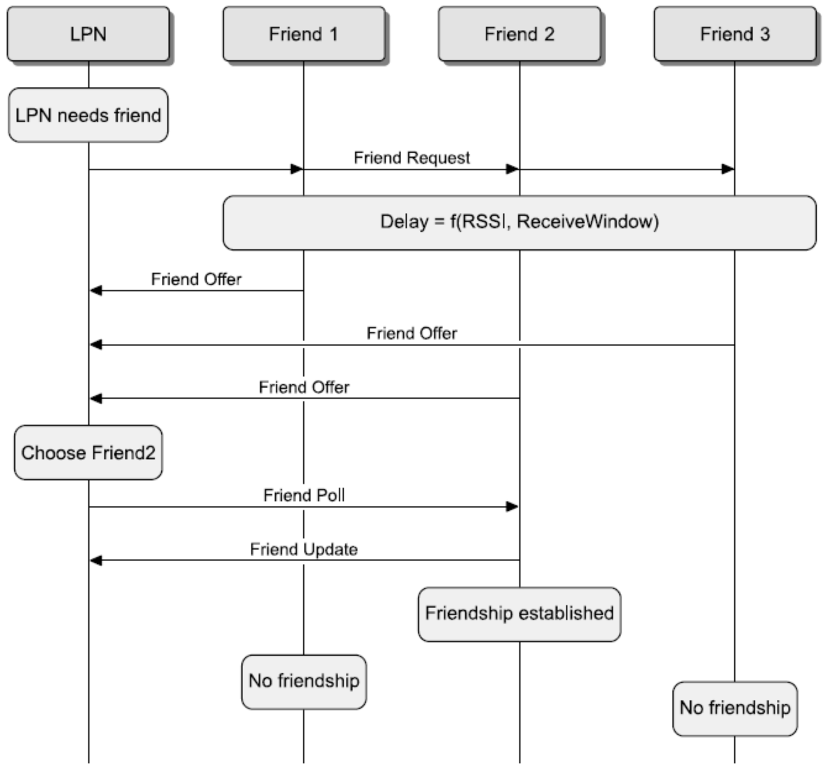
- Friend 1 calculates the Local Delay as 220 (Local Delay = 2*200 – 2*90)
- Friend 2 calculates the Local Delay as 120 (Local Delay = 2*100 – 2*40)

The LPN will receive a Friend offer from Friend 1 and Friend 2 after 220 ms and 120 ms respectively.

This means that the LPN receives Friend offers quicker from nodes that match the LPN requirements, reducing the power consumed by the LPN when initially searching for a Friend Node.

Figure 26 shows the Friendship establishment flow.

Figure 26. LPN and Friend Node Friendship Establishment Flow (Bluetooth SIG, 2019)



Code 1 shows a completed `wiced_bt_mesh_core_config_t` structure.

Code 1: Configuration of `wiced_bt_mesh_core_config_t` Structure

```
wiced_bt_mesh_core_config_t  mesh_config =
{
    .company_id         = MESH_COMPANY_ID_CYPRESS,                  // Company identifier assigned by the Bluetooth SIG
    .product_id         = MESH_PID,                                 // Vendor-assigned product identifier
    .vendor_id          = MESH_VID,                                 // Vendor-assigned product version identifier
    .replay_cache_size  = MESH_CACHE_REPLAY_SIZE,                   // Number of replay protection entries, i.e. maximum number of mesh devices that can send
application messages to this device.
    .features           = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY |
                                WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER,   // In Friend mode support friend, relay
    .friend_cfg         =                                           // Configuration of the Friend Feature(Receive Window in Ms, messages cache)
    {
        .receive_window        = 20,                               // Receive Window value in milliseconds supported by the Friend node.
        .cache_buf_len         = 300                                // Length of the buffer for the cache
        .max_lpn_num           = 4                                  // Max number of Low Power Nodes with established friendship. Must be > 0 if Friend
                                                                       feature is supported.
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 0,                                 // contribution of the RSSI measured by the Friend node used in Friend Offer Delay
calculations.
        .receive_window_factor = 0,                                 // contribution of the supported Receive Window used in Friend Offer Delay calculations.
        .min_cache_size_log    = 0,                                 // minimum number of messages that the Friend node can store in its Friend Cache.
        .receive_delay         = 0,                                 // Receive delay in 1 ms units to be requested by the Low Power Node.
        .poll_timeout          = 0                                  // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
    .gatt_client_only          = WICED_FALSE,                       // Can connect to mesh over GATT or ADV
    .elements_num  = (uint8_t)(sizeof(mesh_elements) / sizeof(mesh_elements[0])),   // number of elements on this device
    .elements      = mesh_elements                                  // Array of elements for this device
};
```

7.2.2 Step 2: Define Mesh Element Configuration

The Mesh element configuration structure, `wiced_bt_mesh_core_config_element_t`, defines the following parameters for a Mesh device. Because Mesh devices can contain more than one element, this structure is an array with one entry per element in the device.

- `.location`: This defines the location as defined in GATT Bluetooth Namespace Description. GATT Bluetooth Namespace Description is an optional enumeration used when you want to give some context to the Characteristic value. If, for example, you have a BLE Mesh Server device with two temperature sensors, one indoor and one outdoor, you may use 0x010F (indoor) and 0x0110 (outdoor) as Description values, which give context to the measured temperatures.
- `.default_transition_time`: This defines the transition time for Models of the Element in milliseconds. The default behavior is set to `MESH_DEFAULT_TRANSITION_TIME_IN_MS`, which is ‘0’, indicating an immediate transition time.
- `.onpowerup_state`: This defines the default behavior on power up. For example, in the light_dimmable code example, the default behavior is set to `WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE`. If a transition was in progress when the device was powered down, the element restores the target state when powered up. Otherwise, the element restores the state it was in when powered down. The supported values for this field are `WICED_BT_MESH_ON_POWER_UP_STATE_OFF`, `WICED_BT_MESH_ON_POWER_UP_STATE_DEFAULT`, `WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE.`
- `.default_level`, `.range_min`, and `.range_max`: These parameters define the default values for the `level`, `minimum`, and `maximum` variables supported by an element. For example, in the `light_dimmable` code example, these properties are set to 0, 1, and 0xffff respectively, which means that the default level is Not Applicable, and the level of the Dimmable light can go from 1 to 0xFFFF (decimal 65535).
- `.move_rollover`: The `move_rollover` parameter determines the action when a maximum value is reached. If this value is ‘true’ when the level reaches the maximum range, the level value rolls over to the minimum value; otherwise, it stops at the defined maximum value.
- `.properties`, `.properties_num`: These are arrays of element properties that are defined for each Mesh device. There will be a separate properties array for each element in the device that has properties and there will be one array entry per property in a given element. Each element is configured with specific properties based on application needs. `Properties` is an optional parameter. If not used, `properties_num` should be set to ‘0’ and `properties` should be set to NULL. The parameters for `properties` are described below.
    - `.id` defines the property. For example, it can be hardware revision, firmware revision etc.
    - `.type` defines if the property is a client, admin, manufacturer or user property.
    - `.user_access` defines the read/write permissions for the property.
    - `.max_len` defines the maximum length of the property.
    - `.value` is assigned with the actual value of the property

In the light_dimmable code example, the firmware version is defined as a user property for the primary element as shown in Code 2.

Code 2. Configuring Element Property Structure

```
wiced_bt_mesh_core_config_property_t mesh_element1_properties[] =
{
    {
        .id          = WICED_BT_MESH_PROPERTY_DEVICE_FIRMWARE_REVISION,
        .type        = WICED_BT_MESH_PROPERTY_TYPE_USER,
        .user_access = WICED_BT_MESH_PROPERTY_ID_READABLE,
        .max_len     = WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION,
        .value       = mesh_prop_fw_version
    },
};
```

- `.sensors, .sensors_num`: These are arrays of sensor configuration parameters defined for each sensor, which is part of the Mesh device. There will be a separate 'sensors' array for each element in the device that has sensors and there will be one array entry per sensor in a given element. The `sensors` configuration is optional and is only needed if you are designing a Mesh enabled sensor device. If not used, `sensors_num` should be set to '0' and `sensors` should be set to NULL. Details for the `sensors` configuration can be found in Sensor Configuration.
- `.models, .models_num`: These are an array of models used, which are defined for the Mesh device. There will be a separate 'models' array for each element in the device and there will be one array entry per model in a given element. The model `WICED_BT_MESH_DEVICE` is a mandatory model, which must be part of every Mesh application. For devices with more than one element, it should be included only in the primary element. Additional model(s) must be defined to achieve the user functionality. For example, the Dimmable Lightbulb code example uses `WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER` for achieving the Mesh dimmable lightbulb functionality. The values of the parameters `models` and `models_num` should be configured based on the application requirement and the number of models used by the application, respectively.

Code 3 shows the `light_dimmable` code example element initialization/assignment. This example contains properties and models but no sensors.

Code 3. Element Initialization in the `light_dimmable` Code Example

```
wiced_bt_mesh_core_config_element_t mesh_elements[] =
{
    {
        .location = MESH_ELEM_LOC_MAIN,                               // location description as defined in the GATT Bluetooth Namespace
Descriptors
                                                                         section of the Bluetooth SIG Assigned Numbers
        .default_transition_time = MESH_DEFAULT_TRANSITION_TIME_IN_MS,  // Default transition time for models of the element in
milliseconds
        .onpowerup_state = WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE,     // Default element behavior on power up
        .default_level = 0,                                            // Default value of the variable controlled on this element (for
example power,
                                                                         lightness, temperature, hue...)
        .range_min = 1,                                                // Minimum value of the variable controlled on this element (for
example power,
                                                                         lightness, temperature, hue...)
        .range_max = 0xffff,                                           // Maximum value of the variable controlled on this element (for
example power,
                                                                         lightness, temperature, hue...)
        .move_rollover = 0,                                            // If true when level gets to range_max during move operation, it
switches to
                                                                         min, otherwise move stops.
        .properties_num = MESH_APP_NUM_PROPERTIES,                     // Number of properties in the array models
        .properties = mesh_element1_properties,                        // Array of properties in the element.
        .sensors_num = 0,                                              // Number of sensors in the sensor array
        .sensors = NULL,                                               // Array of sensors of that element
        .models_num = MESH_APP_NUM_MODELS,                             // Number of models in the array models
        .models = mesh_element1_models,                                // Array of models located in that element. Model data is defined
by structure
                                                                         wiced_bt_mesh_core_config_model_t
    },
};
```

7.2.3 Step 3: Define Application Callback Functions

Hardware-specific initialization and application-specific callbacks are defined in the `wiced_bt_mesh_app_func_table_t` structure to enable the Mesh application library to route appropriate callbacks to the user application code. Details of the functions defined in the `wiced_bt_mesh_app_func_table_t` structure are provided in Table 8.

Table 8. Application Function Definitions

Application Callback Function	Comments
`wiced_bt_mesh_app_init_t`	This function is called in the `mesh_application_init()` function in *mesh_application.c*. The user application must initialize Mesh Server or Client Models in this function. You can also initialize any custom features based on application needs like PWM, Timers, etc. This is a mandatory callback for user application code. Every code example has this callback function defined in the `wiced_bt_mesh_app_func_table_t` structure and uses the name `mesh_app_init`.
`wiced_bt_mesh_app_hardware_init_t`	This function is called in the `APPLICATION_START()` function in *mesh_application.c*. Implementation of this function in the user application code is optional. Default hardware configuration for buttons, LEDs, and UART are initialized per the Mesh evaluation board if the user application code does not define a callback function for hardware initialization. The default behavior for the user button on the Mesh Evaluation board is a factory reset so if this callback is not specified, pressing/releasing the user button will remove all provisioning information from the device.
`wiced_bt_mesh_app_gatt_conn_status_t`	This function is called in the `mesh_gatts_callback()` function in *mesh_app_gatt.c.* Implementation of this function in the application is optional. This function allows the application to be notified of a change in the Bluetooth GATT connection status. This function can be used to perform a custom action if required (e.g., turn ON or turn OFF an LED) for a GATT connect/disconnect event.
`wiced_bt_mesh_app_attention_t`	This function is registered as a callback function with the Mesh Core in the `wiced_bt_mesh_core_init()` function in *mesh_application.c.* This callback function is triggered by the Mesh Core Library for the developer to implement logic that can alert the user (e.g., provide the user with a visual indication of which device is being provisioned). If this function is implemented, it should start a periodic seconds attention timer. For example, in the Dimmable Light bulb code example, the LED state is toggled (ON/OFF) every second in the callback function `attention_timer_cb` until the attention timer expires.
`wiced_bt_mesh_app_notify_period_set_t`	This function is called in the `mesh_publication_callback()` function in *mesh_application.c*. This function is used to change the timing for periodic notifications (i.e., publications). Implementation of this function in the user application code is required for Sensor Models as the publish period may need to be modified based on the `fast_cadence_period_divisor` setting.
`wiced_bt_mesh_app_proc_rx_cmd_t`	This function is called in the `mesh_application_proc_rx_cmd()` function in *mesh_app_hci.c.* Implementation of this function in the user application code is optional. If defined in the user application code, this function is called when an HCI UART command is received from the external host. This function is relevant in the case of a hosted MCU implementation; it is not applicable for a standalone chip implementation.
`wiced_bt_mesh_app_lpn_sleep_t`	This function is called in the `WICED_BT_MESH_CORE_STATE_LPN_SLEEP` event of the `mesh_state_change_cb` function in *mesh_application.c*. Implementation of this function in the user application code is mandatory for Mesh devices which support the LPN (Low Power Node) feature. In this function, the user application can put the Mesh device in to the lowest power state by calling `wiced_sleep_enter_hid_off`. The parameter passed is the maximum time that the device may be put to sleep. It is up to the user application to decide whether to not sleep, sleep for a reduced amount of time, or sleep for the maximum allowed time based on its own requirements.
`wiced_bt_mesh_app_factory_reset_t`	This function is called in the `mesh_application_factory_reset()` function in *mesh_application.c.* Default code in *mesh_application.c* file takes care of initializing the Mesh Core on a factory reset operation. Implementation of this callback function in the user application code is optional. The code placed in this function is executed before the Mesh Core rest code. For example, a Mesh code example can implement this function to delete data stored in NVRAM during a factory reset.

7.2.4 Step 4: Implement Application Callback Functions

The callback functions defined in Step 3 are implemented in this step. The following are examples of typical features and functions that are implemented in the user application code.

- Device name and Appearance is configured
- PWMs, Timers, Sensors, if used, are initialized
- Mesh Models are initialized, and callback functions are passed as the parameters to the initialization APIs
- LPN sleep, Factory reset, Publication period calculation etc. are implemented based on the application needs

7.2.5 Step 5: Implement the Mesh Model Callback Functions

During initialization, a callback function is registered for each model to handle the received messages for that model and perform an appropriate action. For example, in the case of a Dimmable Light bulb code example mesh_app_message_handler is registered during Model initialization for the Light Lightness Server Model. The Mesh Models library will trigger a callback to the function mesh_app_message_handler if it receives data for the registered Model. The User application code receives lightness/level information and accordingly control the level (brightness) of the LED. See Step5 of the Dimmable Light bulb section 8.2.2.2 for further details.

The user application code can also implement design-specific application functions to achieve the required functionality in this step. For example, in a Dimmer Switch application custom logic is implemented to handle the button press events.

If a given model does not require a callback function (i.e., it doesn't require the user application code to take any action for received messages), the callback can be specified as NULL in the Model initialization function.

8 Mesh Code Examples

The code examples described in this application note are designed for specific applications, such as a Dimmable Light Bulb, and Dimmer Switch. These are contained in the Mesh-Demo application group. More general code examples demonstrating each Mesh Model can be found in the Mesh-Snip application group. In this section, we will step through two use cases which implement a variety of Mesh features and functionalities using applications from the Mesh-Demo group.

Before you proceed to the remainder of this document, you should create Mesh-Demo group code examples as shown in Section 4.1.4.1. After successful creation of Mesh-Demo group code examples, a list of projects can be found in the “Project Explorer” window of the Eclipse IDE for ModusToolbox as shown in Figure 9. CYBT-213043-Mesh board support package (BSP) was used to create Mesh-Demo group code examples for this application note.

The two use cases that will be covered in this section are shown below:

1. **Use Case #1: Dimmer Switch Controlling Three Dimmable Light Bulbs** using the following code examples:

   - Dimmer Switch – `dimmer`, Section 8.2.1. The naming convention of code examples might vary across different versions and board support packages. In ModusToolbox 2.x with wiced_btsdk 2.1 or later and the CYBT-213043-Mesh BSP, look for a project named “Mesh-Demo-213043MESH.dimmer” in the “Project Explorer” window of the IDE.
   - Dimmable Light Bulb – `light_dimmable`, Section 8.2.2. The naming convention of code examples might vary across different versions and board support packages. In ModusToolbox2.x with wiced_btsdk 2.1 or later and the CYBT-213043-Mesh BSP, look for a project named “Mesh-Demo-213043MESH.light_dimmable” in the “Project Explorer” window of the IDE.

2. **Use Case #2: Low-Power ON/OFF Server with Three Dimmable Light Bulbs**

   - Low-Power ON/OFF Server – `low_power_led`, Section 8.3.1. The naming convention of code examples might vary across different versions and board support packages. In ModusToolbox 2.x with wiced_btsdk 2.1 or later and the CYBT-213043-Mesh BSP, look for a project named “Mesh-Demo-213043MESH.low_power_led” in the “Project Explorer” window of the IDE.
   - Dimmable Light Bulb – `light_dimmable`, Section 8.3.2. The naming convention of code examples might vary across different versions of the installation. In ModusToolbox2.x with wiced_btsdk 2.1 or later and the CYBT-213043-Mesh BSP look for a project named “Mesh-Demo-213043MESH.light_dimmable” in the “Project Explorer” window of the IDE.

8.1 Prerequisites

Before you get started with the implementation, ensure that you have the following software and hardware:

- ModusToolbox 2.1 or later
- One of the following Cypress **MESH Helper Applications**:
  - Android 7.1 or later with MeshLightingController app
  - iOS 10 or later with MeshLightingController app
  - MeshClient for Windows 10
  - ClientControlMesh application for Windows 7

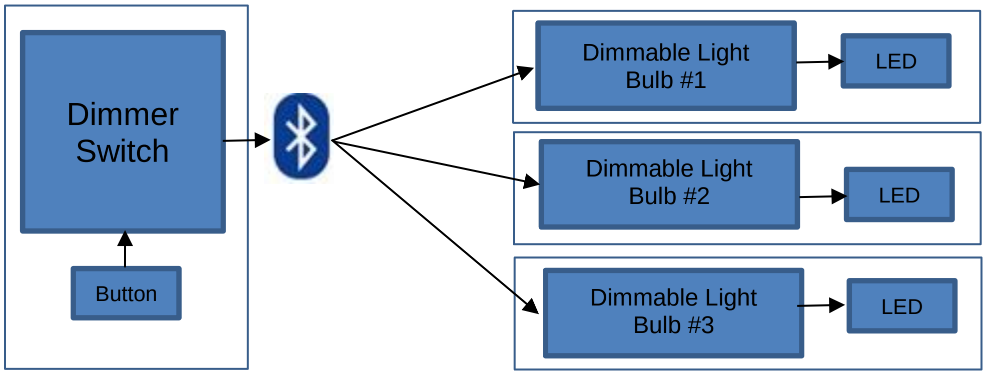
    Note: The ClientControlMesh application requires one Mesh kit to act as the Provisioning client. Therefore, for the code example in Use Case #1, only two Dimmable Light bulbs and one Dimmer Switch will be possible unless you have an additional kit to use for provisioning.

8.2 Use Case #1: Dimmer Switch Controlling Three Dimmable Light Bulbs

There are two example projects associated with Use Case #1, `dimmer` (Dimmer Switch) and `light_dimmable` (Dimmable Light Bulb). These code examples are detailed in sections 8.2.1 (Dimmer Switch) and 8.2.2 (Dimmable Light Bulb).

In this use case, a button press on the Dimmer Switch device will initiate a command that controls the level and ON/OFF state of the LED on the Dimmable Light Bulb nodes. The LED on the Dimmable Light Bulb nodes can also be controlled using the Mesh Helper applications (Android/iOS or Windows). Figure 27 summarizes the functions and assignments for each of the four Mesh evaluation boards used in this example.

Figure 27. Use Case 1: Dimmer Switch with Three Dimmable Light Bulbs



8.2.1 Dimmer Switch Code Example

8.2.1.1 Dimmer Switch Characteristics

- Contains one primary element, which enables the sending of ON/OFF and level commands within the Mesh network based on the user button status on the evaluation board.
- Uses the BLE Mesh `Generic Level Client` model to achieve ON/OFF and level control of the light bulbs within the network.
- The `dimmer` code example, which is included in ModusToolbox, implements this Dimmer Switch device.

8.2.1.2 `dimmer` Code Example

This section describes the five key steps required to implement this code example. These steps are already implemented in the provided code example and are described here for learning purposes.

Step 1: In this step, the Mesh Core configuration structure (`wiced_bt_mesh_config_t`) parameters are configured. The Dimmer Switch Core Configuration Structure used for this example code is shown in Code 4.

Each parameter in this structure is explained in Section 7.2.1. Key items configured in this structure for the Dimmer Switch code example are as follows:

The `LOW_POWER_NODE` macro is not enabled for the Dimmer Switch application as this device is generally connected to a main power supply. Enable the `LOW_POWER_NODE` macro in the application settings window if the Dimmer Switch device is battery powered.

- `.features`: The ‘features’ parameter of this structure is defined as ‘0’. This means that the Dimmer Switch does not support Proxy, Friend, Relay or LPN features. You can enable these features by changing the appropriate bits in the ‘features’ parameters as required. For example, make the following change to enable the Friend and Relay features:

```
.features = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY
```

- `.friend_cfg, .low_power`: As this device does not support Friend or LPN features, `friend_cfg` and `low_power` parameters are set to ‘0’.
- `.elements, .elements_num`: The number of elements and the details of each element used for the Dimmer Switch code example are defined in the `elements_num` and `elements` parameters of this structure. Step 2 describes the element configuration.

Code 4. Dimmer Switch Mesh Core Configuration Structure

```
wiced_bt_mesh_core_config_t  mesh_config =
{
    .company_id         = MESH_COMPANY_ID_CYPRESS,                  // Company identifier assigned by the Bluetooth SIG
    .product_id         = MESH_PID,                                 // Vendor-assigned product identifier
    .vendor_id          = MESH_VID,                                 // Vendor-assigned product version identifier
    .firmware_id        = MESH_FWID,                                // Vendor-assigned firmware version identifier
    .replay_cache_size  = MESH_CACHE_REPLAY_SIZE,                   // Number of replay protection entries, i.e. maximum number of mesh
devices that
                                                                       can send application messages to this device.
#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
    .features           = WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER, // A bit field indicating the device features. In Low Power mode no
Relay, no Proxy and no Friend
    .friend_cfg         =                                           // Empty Configuration of the Friend Feature
    {
        .receive_window = 0,                                        // Receive Window value in milliseconds supported by the Friend node.
        .cache_buf_len  = 0                                         // Length of the buffer for the cache
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 2,                                 // contribution of the RSSI measured by the Friend node used in Friend
Offer Delay
                                                                       calculations.
        .receive_window_factor = 2,                                 // contribution of the supported Receive Window used in Friend Offer
Delay calculations.
        .min_cache_size_log    = 3,                                 // minimum number of messages that the Friend node can store in its
Friend Cache.
        .receive_delay         = 100,                               // Receive delay in 1ms units to be requested by the Low Power Node.
        .poll_timeout          = 36000                              // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
#else
    .features           = 0,                                        // no, support for proxy, friend, or relay
    .friend_cfg         =                                           // Empty Configuration of the Friend Feature
    {
        .receive_window         = 0,                                // Receive Window value in milliseconds supported by the Friend node.
        .cache_buf_len          = 0                                 // Length of the buffer for the cache
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 0,                                 // contribution of the RSSI measured by the Friend node used in Friend
Offer Delay
                                                                       calculations.
        .receive_window_factor = 0,                                 // contribution of the supported Receive Window used in Friend Offer
Delay calculations.
        .min_cache_size_log    = 0,                                 // minimum number of messages that the Friend node can store in its
Friend Cache.
        .receive_delay         = 0,                                 // Receive delay in 1 ms units to be requested by the Low Power Node.
        .poll_timeout          = 0                                  // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
#endif
    .gatt_client_only          = WICED_FALSE,                       // Can connect to mesh over GATT or ADV
    .elements_num  = (uint8_t)(sizeof(mesh_elements) / sizeof(mesh_elements[0])),   // number of elements on this device
    .elements      = mesh_elements                                  // Array of elements for this device
};
```

Step 2: In this step, the Mesh Core Element configuration structure (`wiced_bt_mesh_core_config_element_t`) parameters are configured. The Mesh Core Element Configuration Structure used for the Dimmer Switch is shown in Code 5. This example contains one element.

Each parameter in this structure is explained in Section 7.2.2. Key parameters configured in this structure for the Dimmer Switch code example are described below:

- `.sensors, .sensors_num, .properties, .properties_num:` The dimmer code example project does not support element properties or sensors; therefore, these fields are configured to be '0' or NULL.
- `.models, .models_num`: Models for the Dimmer Switch code example are defined in the `wiced_bt_mesh_core_config_model_t` structure. The parameter `models_num` is configured based on the number of Mesh Models used for the design.
    - `WICED_BT_MESH_DEVICE`

      The Mesh Device includes Configuration Server and Health Server models. These models always need to be included in the primary element of a Mesh device.
    - `WICED_BT_MESH_MODEL_LEVEL_CLIENT`

      The Dimmer Switch device implements a Generic Level Client model. The device can be configured to send messages to any device that supports the Generic Level Server model (`WICED_BT_MESH_CORE_MODEL_ID_GENERIC_LEVEL_SRV`) including models that extend the Generic Level Server model.

For example, the model implemented in the Dimmer Switch code example can also control the following server models:

`WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER`,

`WICED_BT_MESH_MODEL_LIGHT_CTL_SERVER`,

`WICED_BT_MESH_MODEL_LIGHT_HSL_SERVER`, etc.

Code 5. Dimmer Switch Mesh Core Element Configuration Structure

```
wiced_bt_mesh_core_config_element_t mesh_elements[] =
{
    {
        .location = MESH_ELEM_LOC_MAIN,                                 // location description as defined in the GATT Bluetooth 
Namespace Descriptors section of 
                                                                           the Bluetooth SIG Assigned Numbers
        .default_transition_time = MESH_DEFAULT_TRANSITION_TIME_IN_MS,  // Default transition time for models of the element in 
milliseconds
        .onpowerup_state = WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE,     // Default element behavior on power up
        .default_level = 0,                                             // Default value of the variable controlled on this element (for 
example power, lightness, 
                                                                           temperature, hue...)
        .range_min = 1,                                                 // Minimum value of the variable controlled on this element (for 
example power, lightness, 
                                                                           temperature, hue...)
        .range_max = 0xffff,                                            // Maximum value of the variable controlled on this element (for 
example power, lightness, 
                                                                           temperature, hue...)
        .move_rollover = 0,                                             // If true when level gets to range_max during move operation, 
it switches to min, 
                                                                           otherwise move stops.
        .properties_num = 0,                                            // Number of properties in the array models
        .properties = NULL,                                             // Array of properties in the element.
        .sensors_num = 0,                                               // Number of sensors in the sensor array
        .sensors = NULL,                                                // Array of sensors of that element
        .models_num = MESH_APP_NUM_MODELS,                              // Number of models in the array models
        .models = mesh_element1_models,                                 // Array of models located in that element. Model data is 
defined by structure 
                                                                           wiced_bt_mesh_core_config_model_t
    },
};


wiced_bt_mesh_core_config_model_t   mesh_element1_models[] =
{
    WICED_BT_MESH_DEVICE,
    WICED_BT_MESH_MODEL_LEVEL_CLIENT,
};
```

Step 3: This step registers the application callback functions for the code example.

The Mesh application initialization and hardware initialization callback functions are defined in the `wiced_bt_mesh_app_func_table_t` structure. The Mesh Core performs default actions as defined in the *mesh_application.c* file for the items that do not have any callback function registered in the function table.

For example, the Dimmer Switch application project will not receive any callback on a GATT connection or disconnection. Because there is no callback function defined in this case, the Dimmer Switch application will not be able to process any custom actions (e.g., turn ON or turn OFF a LED) on a GATT connection or disconnection. A callback function can be added to this structure to enable the user to perform a custom action if desired.

The application function definitions that are used for the Dimmer Switch code example are shown in Code 6.

Code 6. Dimmer Switch Application Function Definitions

```
  wiced_bt_mesh_app_func_table_t wiced_bt_mesh_app_func_table =
{
    mesh_app_init,          // application initialization
    button_hardware_init,   // hardware initialization
    NULL,                   // GATT connection status
    NULL,                   // attention processing
    NULL,                   // notify period set
    NULL,                   // WICED HCI command
    NULL,                   // LPN sleep
    NULL                    // factory reset
};
```

Step 4: This step implements the application callback functions.

There are two application callback functions defined for the `dimmer` code example, `mesh_app_init()` and `button_hardware_init()`. Each is described below.

1. `mesh_app_init()`

   In this function shown in Code 7, the following actions are performed:

   - The device name is configured using the below API function:

     ```
     wiced_bt_cfg_settings.device_name = (uint8_t *)"Dimmer";
     ```

   - The device appearance is configured using the below API function:

     ```
     wiced_bt_cfg_settings.gatt_cfg.appearance = APPEARANCE_CONTROL_DEVICE_SLIDER;
     ```

   - The scan response data is configured for the Dimmer Switch code example using the `wiced_bt_mesh_set_raw_scan_response_data()` function. In this example, the Device Name and Appearance are sent as part of the scan response data.

   - The button default state is initialized using the `button_control_init()` function, which is defined in *button_control.c*.

   - The Mesh Level Client Model is initialized by calling `wiced_bt_mesh_model_level_client_init` with a callback function, `mesh_app_message_handler`, passed as a parameter so that the user application code can receive messages for this model. The Mesh Model Library will trigger the registered callback function, `mesh_app_message_handler`, when messages are received for this model. The `mesh_app_message_handler` callback function implementation is described in Step 5 of this section.

Code 7. `mesh_app_init()` Function

```
Void mesh_app_init(wiced_bool_t is_provisioned)
{
#if 0
    extern uint8_t wiced_bt_mesh_model_trace_enabled;
    wiced_bt_mesh_model_trace_enabled = WICED_TRUE;
#endif
    wiced_bt_cfg_settings.device_name = (uint8_t *)"Dimmer";
    wiced_bt_cfg_settings.gatt_cfg.appearance = APPEARANCE_CONTROL_DEVICE_SLIDER;

// Adv Data is fixed. Spec allows to put URI, Name, Appearance and Tx Power in the Scan Response Data.
    if (!is_provisioned)
    {
        wiced_bt_ble_advert_elem_t  adv_elem[3];
        uint8_t                     buf[2];
        uint8_t                     num_elem = 0;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_NAME_COMPLETE;
        adv_elem[num_elem].len         = (uint16_t)strlen((const char*)wiced_bt_cfg_settings.device_name);
        adv_elem[num_elem].p_data      = wiced_bt_cfg_settings.device_name;
        num_elem++;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_APPEARANCE;
        adv_elem[num_elem].len         = 2;
        buf[0]                         = (uint8_t)wiced_bt_cfg_settings.gatt_cfg.appearance;
        buf[1]                         = (uint8_t)(wiced_bt_cfg_settings.gatt_cfg.appearance >> 8);
        adv_elem[num_elem].p_data      = buf;
        num_elem++;

        wiced_bt_mesh_set_raw_scan_response_data(num_elem, adv_elem);
    }

    button_control_init();
    wiced_bt_mesh_model_level_client_init(MESH_LEVEL_CLIENT_ELEMENT_INDEX, mesh_app_message_handler, is_provisioned);
}
```

2. `button_hardware_init()`

   Code 8 shows the `button_hardware_init()` function. In this function, the User Button of the evaluation board is configured with an interrupt, and an interrupt callback function is registered. In this case, the User Button number (`WICED_PLATFORM_BUTTON_1`), User Button configuration (`GPIO_EN_INT_BOTH_EDGE`), and interrupt callback function (`button_interrupt_handler`) are passed as the parameters to `wiced_platform_register_button_callback`. The interrupt callback function, `button_interrupt_handler`, is triggered when the User Button is pressed or released.

Functions related to the User Button handling are implemented in *button_control.c*. The callback function `button_interrupt_handler` handles the User Button interrupts and implements the logic shown in Code 9. The function `button_set_level` is called in the `button_interrupt_handler`. The `wiced_bt_mesh_model_level_client_set()` API is called in the `button_set_level` function to publish Mesh messages. Code 10 shows the `button_set_level()` function, which is called by the `button_interrupt_handler()` function.

Code 8. `button_hardware_init()` Function

```
void button_hardware_init(void)
{
    /* Configure buttons available on the platform */
#if defined(CYW20706A2)
    wiced_hal_gpio_configure_pin(WICED_GPIO_BUTTON, WICED_GPIO_BUTTON_SETTINGS(GPIO_EN_INT_BOTH_EDGE), WICED_GPIO_BUTTON_DEFAULT_STATE);
    wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_BUTTON, button_interrupt_handler, NULL);
#elif (defined(CYW20735B0) || defined(CYW20719B0) || defined(CYW20721B0))
    wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_PIN_BUTTON, button_interrupt_handler, NULL);
    wiced_hal_gpio_configure_pin(WICED_GPIO_PIN_BUTTON, WICED_GPIO_BUTTON_SETTINGS, GPIO_PIN_OUTPUT_LOW);
#else
    wiced_platform_register_button_callback(WICED_PLATFORM_BUTTON_1, button_interrupt_handler, NULL, GPIO_EN_INT_BOTH_EDGE);
#endif
}
```

Code 9. `button_interrupt_handler()` Function

```
/*
 * Process interrupts from the button.
 */
void button_interrupt_handler(void* user_data, uint8_t pin)
{
    uint32_t value = wiced_hal_gpio_get_pin_input_status(pin);
    uint32_t current_time = wiced_bt_mesh_core_get_tick_count();
    uint32_t button_pushed_duration;

    if (value == button_previous_value)
    {
        WICED_BT_TRACE("interrupt_handler: duplicate pin:%d value:%d current_time:%d\n", pin, value, current_time);
        return;
    }
    button_previous_value = value;

    WICED_BT_TRACE("interrupt_handler: pin:%d value:%d current_time:%d\n", pin, value, current_time);

    if (value == platform_button[WICED_PLATFORM_BUTTON_1].button_pressed_value)
    {
        button_pushed_time = current_time;

        // if button is not released within 500ms, we will start sending move events
        wiced_start_timer(&button_timer, 500);
        return;
    }
    wiced_stop_timer(&button_timer);

    // button is released
    button_pushed_duration = current_time - button_pushed_time;
    if (button_pushed_duration < 500)
    {
        button_level_moving = WICED_FALSE;

        if (button_step == 0)
            button_step = NUM_STEPS - 1;
        else if (button_step == NUM_STEPS - 1)
            button_step = 0;
        else
            button_step = (button_direction ? NUM_STEPS - 1 : 0);
        button_set_level(WICED_TRUE, WICED_TRUE);
        return;
    }
    else if (button_pushed_duration < 15000)
    {
        // we were moving the level and button is released.
        // set message with ack
        if ((button_step != NUM_STEPS - 1) && (button_step != 0))
            button_set_level(WICED_FALSE, WICED_TRUE);
        return;
    }
    // More than 15 seconds means factory reset
    mesh_application_factory_reset();
}
```

Code 10. `button_set_level()` Function

```
void button_set_level(wiced_bool_t is_instant, wiced_bool_t is_final)
{
    wiced_bt_mesh_level_set_level_t set_data;

    set_data.level = button_level_step[button_step];
    set_data.transition_time = is_instant ? 100 : 500;
    set_data.delay = 0;

    WICED_BT_TRACE("Set level:%d transition time:%d final:%d\n", set_data.level, set_data.transition_time, is_final);

    wiced_bt_mesh_model_level_client_set(0, is_final, &set_data);
}
```

Step 5: This step implements the Mesh message Callback functions.

Because the Dimmer Switch is a client device, the purpose of the defined callback function is not to process and take any action, but to confirm that messages are sent into the network. The `mesh_app_message_handler` callback function registered during initialization in the Dimmer Switch application will receive callbacks from the Mesh Models Library, which provides information as to the current Light Level Status of the Dimmer Switch and the status of any Mesh network messages being sent from the Dimmer Switch.

Code 11. `mesh_app_message_handler()` Callback Function

```
Void mesh_app_message_handler(uint16_t event, wiced_bt_mesh_event_t *p_event, wiced_bt_mesh_level_status_data_t *p_data)
{
    WICED_BT_TRACE("level clt msg:%d\n", event);

    switch (event)
    {
    case WICED_BT_MESH_TX_COMPLETE:
        WICED_BT_TRACE("tx complete status:%d\n", p_event->tx_status);
        break;

    case WICED_BT_MESH_LEVEL_STATUS:
        WICED_BT_TRACE("level present:%d target:%d remaining time:%d\n", p_data->present_level, p_data->target_level, p_data-
>remaining_time);
        break;

    default:
        WICED_BT_TRACE("unknown\n");
        break;
    }
    wiced_bt_mesh_release_event(p_event);
}
```

8.2.2 Dimmable Light Bulb Code Example

8.2.2.1 Dimmable Light Bulb Characteristics

- Contains one element to support ON/OFF and level control of an LED based on Mesh messages received from the Generic OnOff Client, Generic Level Client, or Light Lightness Client (i.e., Dimmer Application or Helper Applications).
- Uses the BLE Mesh Light Lightness Server model, which enables the light bulbs to receive commands from the Dimmer Switch. Commands received are then processed locally on the light bulbs within the user application code.
- The Dimmable Light Bulb device controls the state of its LED based on the received commands from the Mesh network, acts as a proxy node, acts as a relay node to forward messages to other nodes in the vicinity, and acts as a friend node for nearby low power nodes.
- The `light_dimmable` code example, which is part of Mesh-Demo group code example, implements the Dimmable Light Bulb device.

8.2.2.2 Dimmable Light Bulb: light_dimmable

This section describes the five key steps required to implement this code example. These steps are already implemented in the provided code example and are described here for learning purposes.

Step 1: In this step, the Mesh Core configuration structure, `wiced_bt_mesh_config_t`, parameters are configured. The Dimmable Light Bulb Core Configuration Structure used for this example code is shown in Code 12.

Each parameter in this structure is explained in Section 7.2.1. Key items configured in this structure for the light_dimmable code example are as follows:

- `.features`: The Dimmable Light Bulb device supports Friend, Relay and GATT Proxy features of Bluetooth Mesh specification. Therefore, the corresponding bits are set in this code example. You can enable or disable features by changing the appropriate bits in the `features` parameter as desired.

  `.features = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY | WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER`

- `.friend_cfg`: The `friend_cfg` field parameters are configured with the `receive_window` of '20', `cache_buf_len` of '300' and `max_lpn_num` as '4'.
  - The `ReceiveWindow` is the time that the Low-Power Node listens for a response. In this case, the Friend node will send the response within 20 milliseconds after the LPN starts listening.
  - Maximum cache buffer length supported by the Dimmable Light device is configured to 300 bytes.
  - The maximum number of LPNs supported by the Friend node is configured as '4', which means that it can support up to four LPNs.

- `.low_power`: Because this device does not support low-power features, `low_power` parameters are set to '0'.

- `.elements, .elements_num`: Number of elements and details of each element used for the `light_dimmable` code example are defined in `elements_num` and `elements` parameter of this structure. We shall look at the element configuration in Step 2.

Code 12. Dimmable Light Mesh Core Configuration Structure

```
wiced_bt_mesh_core_config_t  mesh_config =
{
    .company_id         = MESH_COMPANY_ID_CYPRESS,                  // Company identifier assigned by the Bluetooth SIG
    .product_id         = MESH_PID,                                 // Vendor-assigned product identifier
    .vendor_id          = MESH_VID,                                 // Vendor-assigned product version identifier
    .replay_cache_size  = MESH_CACHE_REPLAY_SIZE,                   // Number of replay protection entries, i.e. maximum number of mesh devices that can send
                                                                       application messages to this device.
    .features           = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY | WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER,   // In
                                                                       Friend mode support friend, relay
    .friend_cfg         =                                           // Configuration of the Friend Feature(Receive Window in Ms, messages cache)
    {
        .receive_window        = 20,                               // Receive Window value in milliseconds supported by the Friend node.
        .cache_buf_len         = 300                                // Length of the buffer for the cache
        .max_lpn_num           = 4                                  // Max number of Low Power Nodes with established friendship. Must be > 0 if Friend feature is
                                                                       supported.
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 0,                                 // contribution of the RSSI measured by the Friend node used in Friend Offer Delay
                                                                        calculations.
        .receive_window_factor = 0,                                 // contribution of the supported Receive Window used in Friend Offer Delay calculations.
        .min_cache_size_log    = 0,                                 // minimum number of messages that the Friend node can store in its Friend Cache.
        .receive_delay         = 0,                                 // Receive delay in 1 ms units to be requested by the Low Power Node.
        .poll_timeout          = 0                                  // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
    .gatt_client_only          = WICED_FALSE,                       // Can connect to mesh over GATT or ADV
    .elements_num  = (uint8_t)(sizeof(mesh_elements) / sizeof(mesh_elements[0])),   // number of elements on this device
    .elements      = mesh_elements                                  // Array of elements for this device
};
```

Step 2: In this step, parameters for the Mesh Core Element configuration structure, `wiced_bt_mesh_core_config_element_t`, are configured. The Dimmable Light Bulb Mesh Core Element Configuration code is shown in Code 13. This example contains one element.

Each parameter in this structure is explained in Section 7.2.2. Key parameters configured in this structure for the LightDimmable code example are as follows:

- `.properties, .properties_num:` The parameters `properties_num` and `properties` define the number of properties and the details of the properties respectively for the Mesh element. In this case, we defined an 8-byte firmware version as a read-only property. Details of the definition can be found in the `mesh_element1_properties` structures as shown in Code 13. The value of the firmware version in the structure `mesh_element1_properties` is defined by the array `mesh_prop_fw_version`.

```
uint8_t mesh_prop_fw_version[WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION] = {
                                                              '0', '6', '.', '0', '2', '.', '0', '5' };
```

- `.sensors, .sensors_num: The parameters sensors_num` and `sensors` are configured as '0 and 'NULL' as the Dimmable Light code example project does not support sensors.

- `.models, .models_num`: The parameter `models_num` is configured based on the number of Mesh Models used for the design. The parameter `model` defines an array of models used for this code example. Models used for the light_dimmable code example are defined in the `wiced_bt_mesh_core_config_model_t` structure.
  - `WICED_BT_MESH_DEVICE`

    The Mesh Device includes Configuration Server and Health Server models. These models always need to be included in the primary element of a Mesh device.
  - `WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER`

    This model supports the Generic User Property server. The Generic User Property server defines state instances that represent device property of an element. Depending the defined device properties, this value may be read-only or read-write. In this case, the LightDimmable device firmware revision is configured as read-only as part of the `mesh_element1_properties` structure. This is an optional model that we have used to provide the firmware revision of the Dimmable Light Bulb to the Mesh network.
  - `WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER`

    The Light Lightness Server model extends the Generic Power OnOff Server Model and Generic Level Server Model. Therefore, the Dimmable Light Bulb device can be controlled by a Generic OnOff Client, a Generic Level Client (e.g. Dimmer Switch device), or a Light Lightness Client (e.g., Helper applications).

Code 13. Dimmable Light Mesh Core Element Configuration Structure and Mesh Core Element Property Structure

```
wiced_bt_mesh_core_config_element_t mesh_elements[] =
{
    {
        .location = MESH_ELEM_LOC_MAIN,                               // location description as defined in the GATT Bluetooth Namespace Descriptors section of
                                                                         the Bluetooth SIG Assigned Numbers
        .default_transition_time = MESH_DEFAULT_TRANSITION_TIME_IN_MS,  // Default transition time for models of the element in milliseconds
        .onpowerup_state = WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE,     // Default element behavior on power up
        .default_level = 0,                                           // Default value of the variable controlled on this element (for example power, lightness,
                                                                         temperature, hue...)
        .range_min = 1,                                               // Minimum value of the variable controlled on this element (for example power, lightness,
                                                                         temperature, hue...)
        .range_max = 0xffff,                                          // Maximum value of the variable controlled on this element (for example power, lightness,
                                                                         temperature, hue...)
        .move_rollover = 0,                                           // If true when level gets to range_max during move operation, it switches to min,
                                                                         otherwise move stops.
        .properties_num = MESH_APP_NUM_PROPERTIES,                    // Number of properties in the array models
        .properties = mesh_element1_properties,                       // Array of properties in the element.
        .sensors_num = 0,                                             // Number of sensors in the sensor array
        .sensors = NULL,                                              // Array of sensors of that element
        .models_num = MESH_APP_NUM_MODELS,                            // Number of models in the array models
        .models = mesh_element1_models,                               // Array of models located in that element. Model data is defined by structure
                                                                         wiced_bt_mesh_core_config_model_t

    },
};


wiced_bt_mesh_core_config_property_t mesh_element1_properties[] =
{
    {
        .id          = WICED_BT_MESH_PROPERTY_DEVICE_FIRMWARE_REVISION,
        .type        = WICED_BT_MESH_PROPERTY_TYPE_USER,
        .user_access = WICED_BT_MESH_PROPERTY_ID_READABLE,
        .max_len     = WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION,
        .value       = mesh_prop_fw_version
    },
};




wiced_bt_mesh_core_config_model_t   mesh_element1_models[] =
{
    WICED_BT_MESH_DEVICE,
    WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER,
    WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER,
};
```

Step 3: This step registers the application callback functions used for the code example.

The Mesh application initialization and Mesh application attention callback functions are defined in the `wiced_bt_mesh_app_func_table_t` structure. The Mesh Core Library performs default actions as defined in the *mesh_application.c* file for the items which do not have any callback function registered in this structure.

Code 14. Dimmable Light Application Function Definitions

```
wiced_bt_mesh_app_func_table_t wiced_bt_mesh_app_func_table =
{
    mesh_app_init,          // application initialization
    NULL,                   // Default SDK platform button processing
    NULL,                   // GATT connection status
    mesh_app_attention,     // attention processing
    NULL,                   // notify period set
    NULL,                   // WICED HCI command
    NULL,                   // LPN sleep
    NULL                    // factory reset
};
```

Step 4: This step implements the application callback functions.

There are two application callback functions defined for the light_dimmable code example, `mesh_app_init()` and `mesh_app_attention()`. Each is described below.

1. `mesh_app_init()`

   In this function shown in Code 15, the following actions are performed:

   - The device name is configured using the below API:

     `wiced_bt_cfg_settings.device_name = (uint8_t *)"Dimmable Light";`

   - The device appearance is configured using the below API:

     `wiced_bt_cfg_settings.gatt_cfg.appearance = APPEARANCE_LIGHT_CEILING;`

   - The scan response data is configured for the light_dimmable code example using the `wiced_bt_mesh_set_raw_scan_response_data()` function. In this example, the Device Name and Appearance are sent as part of the scan response data.
   - The LED is initialized by calling the `led_control_init` function. The Dimmable Light Bulb device uses a PWM to control the intensity of the LED. LED-specific functions, initialization, and level control logic for this application are implemented in the `led_control_init` function defined in *led_control.c*.
   - A Periodic Seconds Timer is initialized in this function. This timer is used for maintaining the attention timer timeout, which is described in the `mesh_app_attention()` function described in Step 5.
   - The Mesh Model Light Lightness server is initialized by passing a callback function, `mesh_app_message_handler`, to the `wiced_bt_mesh_model_light_lightness_server_init` function. The Mesh Models Library will trigger the registered callback function, `mesh_app_message_handler`, when the messages are received for this model. The `mesh_app_message_handler` function is described in Step 5 of this section.
   - The `Mesh_prop_fw_revision` properties array configures the desired values and the Mesh Model User Property Server Model is initialized using the `wiced_bt_mesh_model_property_server_init` function.

Code 15. `mesh_app_init()` Function

```
void mesh_app_init(wiced_bool_t is_provisioned)
{
#if 0
    extern uint8_t wiced_bt_mesh_model_trace_enabled;
    wiced_bt_mesh_model_trace_enabled = WICED_TRUE;
#endif
    wiced_bt_cfg_settings.device_name = (uint8_t *)"Dimmable Light";
    wiced_bt_cfg_settings.gatt_cfg.appearance = APPEARANCE_LIGHT_CEILING;

    mesh_prop_fw_version[0] = 0x30 + (WICED_SDK_MAJOR_VER / 10);
    mesh_prop_fw_version[1] = 0x30 + (WICED_SDK_MAJOR_VER % 10);
    mesh_prop_fw_version[2] = 0x30 + (WICED_SDK_MINOR_VER / 10);
    mesh_prop_fw_version[3] = 0x30 + (WICED_SDK_MINOR_VER % 10);
    mesh_prop_fw_version[4] = 0x30 + (WICED_SDK_REV_NUMBER / 10);
    mesh_prop_fw_version[5] = 0x30 + (WICED_SDK_REV_NUMBER % 10);
    mesh_prop_fw_version[6] = 0x30 + (WICED_SDK_BUILD_NUMBER / 10);
    mesh_prop_fw_version[7] = 0x30 + (WICED_SDK_BUILD_NUMBER % 10);

    // Adv Data is fixed. Spec allows to put URI, Name, Appearance and Tx Power in the Scan Response Data.
    if (!is_provisioned)
    {
        wiced_bt_ble_advert_elem_t  adv_elem[3];
        uint8_t                     buf[2];
        uint8_t                     num_elem = 0;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_NAME_COMPLETE;
        adv_elem[num_elem].len         = (uint16_t)strlen((const char*)wiced_bt_cfg_settings.device_name);
        adv_elem[num_elem].p_data      = wiced_bt_cfg_settings.device_name;
        num_elem++;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_APPEARANCE;
        adv_elem[num_elem].len         = 2;
        buf[0]                         = (uint8_t)wiced_bt_cfg_settings.gatt_cfg.appearance;
        buf[1]                         = (uint8_t)(wiced_bt_cfg_settings.gatt_cfg.appearance >> 8);
        adv_elem[num_elem].p_data      = buf;
        num_elem++;

        wiced_bt_mesh_set_raw_scan_response_data(num_elem, adv_elem);
    }

    led_control_init();

    wiced_init_timer(&attention_timer, attention_timer_cb, 0, WICED_SECONDS_PERIODIC_TIMER);

    // Initialize Light Lightness Server and register a callback which will be executed when it is time to change the brightness of the bulb
    wiced_bt_mesh_model_light_lightness_server_init(MESH_LIGHT_LIGHTNESS_SERVER_ELEMENT_INDEX, mesh_app_message_handler, is_provisioned);

    // Initialize the Property Server.  We do not need to be notified when Property is set, because our only property is readonly
    wiced_bt_mesh_model_property_server_init(MESH_LIGHT_LIGHTNESS_SERVER_ELEMENT_INDEX, NULL, is_provisioned);
}
```

2. `mesh_app_attention()`

   The callback function `mesh_app_attention,` shown in Code 16, is triggered by the Mesh Core Library for the developer to implement logic that can alert the user (e.g., provide the user with a visual indication of which device is being provisioned). A periodic seconds timer called `attention timer` is started in this function. In this example, the LED state is toggled (ON/OFF) every second in the callback function `attention_timer_cb` until the attention timer expires.

Code 16. `mesh_app_attention()` Function

```
void mesh_app_attention(uint8_t element_idx, uint8_t time)
{
    WICED_BT_TRACE("dimmable light attention:%d sec\n", time);

    // If time is zero, stop alerting and restore the last known brightness
    if (time == 0)
    {
        wiced_stop_timer(&attention_timer);
        led_control_set_brighness_level(last_known_brightness);
        return;
    }
    wiced_start_timer(&attention_timer, 1);
    attention_time = time;
    attention_brightness = (last_known_brightness != 0) ? 0 : 100;
    led_control_set_brighness_level(attention_brightness);
}
/*
 * Attention timer callback is executed every second while user needs to be alerted.
 * Just switch brightness between 0 and 100%
 */
void attention_timer_cb(TIMER_PARAM_TYPE arg)
{
    WICED_BT_TRACE("dimmable light attention timeout:%d\n", attention_time);

    if (--attention_time == 0)
    {
        wiced_stop_timer(&attention_timer);
        led_control_set_brighness_level(last_known_brightness);
        return;
    }
    attention_brightness = (attention_brightness == 0) ? 100 : 0;
    led_control_set_brighness_level(attention_brightness);
}
```

Step 5: This step implements the Mesh message callback functions.

When a message is received, the Dimmable Light Bulb node gets a callback to the function registered during initialization, `mesh_app_message_handler`. This callback function, shown in Code 17, is triggered by the Mesh Models Library on receipt of relevant Mesh messages for the BLE Mesh Light Lightness Server model. If the callback function receives the `WICED_BT_MESH_LIGHT_LIGHTNESS_SET` event, the `mesh_app_process_set` function (shown in Code 17) is called, which interprets the received data and performs the appropriate action. The Light Lightness Actual State (`p_status` to `lightness_actual_present`) is sent as the parameter to the callback function, which defines the level (brightness) value for the Dimmable Light node. As the Light Lightness Actual state can have values between 0 to 65535, we scale this value from 0 to 100% and pass it as the parameter to the `led_control_set_brightness()` function which is also shown in Code 17. In this function, the PWM values are modified according to the received level setting to control the intensity of the LED.

Code 17. `mesh_app_message_handler()`, `mesh_app_process_set()`, and `led_control_set_brightness_level()` Functions

```
void mesh_app_message_handler(uint8_t element_idx, uint16_t event, void *p_data)
{
    switch (event)
    {
    case WICED_BT_MESH_LIGHT_LIGHTNESS_SET:
        mesh_app_process_set_level(element_idx, (wiced_bt_mesh_light_lightness_status_t *)p_data);
        break;

    default:
        WICED_BT_TRACE("dimmable light unknown msg:%d\n", event);
        break;
    }
}
/*
 * Command from the level client is received to set the new level
 */
void mesh_app_process_set_level(uint8_t element_idx, wiced_bt_mesh_light_lightness_status_t *p_status)
{
    WICED_BT_TRACE("mesh light srv set level element:%d present actual:%d linear:%d remaining_time:%d\n",
        element_idx, p_status->lightness_actual_present, p_status->lightness_linear_present, p_status->remaining_time);

    last_known_brightness = (uint8_t)((uint32_t)p_status->lightness_actual_present * 100 / 65535);
    led_control_set_brighness_level(last_known_brightness);

    // If we were alerting user, stop it.
    wiced_stop_timer(&attention_timer);
}

void led_control_set_brighness_level(uint8_t brightness_level)

{
    pwm_config_t pwm_config;

    WICED_BT_TRACE("set brightness:%d\n", brightness_level);

    // ToDo.  For some reason, setting brightness to 100% does not work well on 20719B1 platform. For now just use 99% instead of 100.
    if (brightness_level == 100)
        brightness_level = 99;

    wiced_hal_pwm_get_params(PWM_INP_CLK_IN_HZ, brightness_level, PWM_FREQ_IN_HZ, &pwm_config);
    wiced_hal_pwm_change_values(PWM_CHANNEL, pwm_config.toggle_count, pwm_config.init_count);
}
```

8.3 Use Case #2: Low-Power ON/OFF Server with Three Dimmable Light Bulbs

There are two code examples associated with Use Case #2. These code examples are described in Sections 8.3.1 (Low-Power ON/OFF Server) and 8.3.2 (Dimmable Light Bulb).

Note: The Dimmable Light Bulb project is the same as is used in Use Case #1. There are no changes required to this code example for this use case.

In this use case, one of the Mesh evaluation boards is programmed as a Low-Power ON/OFF Server and the other three are programmed as Dimmable Light Bulbs. The Low-Power ON/OFF Server represents any LPN whose primary function is to receive information from the Mesh network and act on it locally

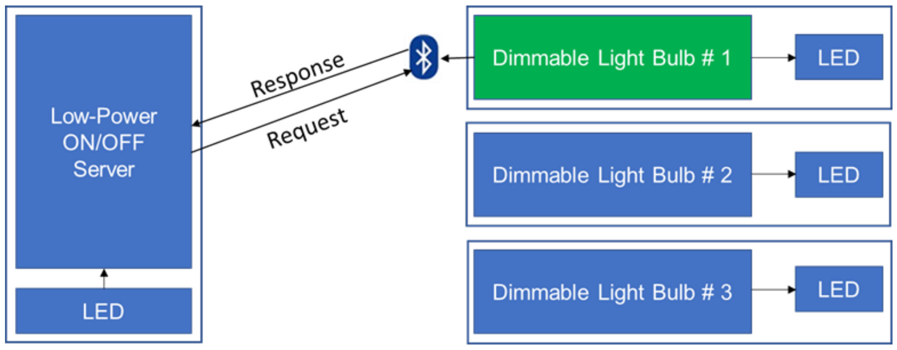
An example of the Low-Power ON/OFF Server would be a Door Lock, where the device is not required to be active all the time, but only at specific intervals. In this use case example, the Low-Power ON/OFF Server uses the Red LED on the Mesh evaluation board to represent an ON or OFF status (the Red LED ON represents the door being unlocked, and the Red LED OFF represents the door being locked). The Dimmable Light Bulbs use the Red LED of the Mesh evaluation board to represent a light bulb. The Low-Power ON/OFF Server code example uses `WICED_BT_MESH_MODEL_POWER_ONOFF_SERVER` model which allows the Provisioner to configure the power up state behavior. For example, you may always want the door to be locked (OFF) at power up while for another application, you may want the power up state to be ON. The model `WICED_BT_MESH_MODEL_POWER_ONOFF_SERVER` allows you to configure the power up state based on the application's requirement.

This use case demonstrates the LPN, Server, Relay, Friend, and Proxy features of Bluetooth Mesh.

The poll time interval (the time at which the LPN will wake and query its Friend) for the Low-Power ON/OFF Server node is defined in its application code locally. There is no interval configuration from the Helper application because the Low-Power ON/OFF Server LPN is not sending information into the network, it is only receiving.

Figure 28 details the setup for Use Case #2. In the illustration, Dimmable Light Bulb #1 is assumed to be the selected Friend node for the Low-Power ON/OFF Server node. In your setup, the Friend will be automatically selected based on the Friendship establishment process as documented in Section 7.2.1.6.

Figure 28. Low Power ON/OFF Server and Three Dimmable Light Bulbs



8.3.1 Low-Power ON/OFF Server Code Example

8.3.1.1 Low-Power ON/OFF Server Characteristics

- Contains one primary element to support the ON/OFF control functionality (represented by the status of the Red LED) based on the received messages from `Generic OnOff Client.`
- Uses the `WICED_BT_MESH_MODEL_POWER_ONOFF_SERVER` model, which enables the Low-Power ON/OFF Server node to receive messages from the Mesh network.
- Receives messages from the Mesh network and controls the state of the Red LED. As the device is configured for low power, it wakes up at regular intervals and polls for its messages from a Friend node. For this reason, the state change on the Low-Power ON/OFF Server node may have a delayed response.
- The `low_power_led` code example, which is part of Mesh-Demo group code examples, implements the Low-Power ON/OFF Server device.

8.3.1.2 low_power_led Code Example

This section describes the five key steps required to implement this code example. These steps are already implemented in the code example provided and are described here for learning purposes.

Note:

- The `LOW_POWER_NODE` macro is not enabled by default. We will enable the Low Power Node feature in this code example. To do so, in the low_power_led/makefile, set "LOW_POWER_NODE ?= 1", this will enable the low power feature of the 20819.
- Based on the `LOW_POWER_NODE` macro setting, if enabled, the Low-Power ON/OFF Server code example will support the LPN feature; otherwise, it will support Friend, Relay, and GATT Proxy features. This can be found in the `mesh_config` structure in *low_power_led.c*.

Step 1: In this step, parameters of the Mesh Core configuration structure, `wiced_bt_mesh_config_t`, are configured. The Low-Power ON/OFF Server Mesh Core Configuration code is shown in Code 18.

Each parameter in this structure is explained in Section 7.2.1. The key items configured in this structure for the Low-Power ON/OFF Server (low_power_led) code example are as follows:

Based on the status of the `LOW_POWER_NODE` macro, the Low-Power ON/OFF Server device can act as a `Power OnOff Server` with the LPN feature (`LOW_POWER_NODE` macro is enabled: target implementation) OR as a `Power OnOff Server` which supports Friend, Relay, and GATT Proxy features (if the `LOW_POWER_NODE` macro is not enabled).

- If `LOW_POWER_NODE = 1`, the Low-Power ON/OFF Server acts as a `Power OnOff Server` with support for the LPN feature.
- `.features`: The `features` field is configured to enable the LPN feature:

  ```
  .features = WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER
  ```

- `.friend_cfg`: Parameters of `friend_cfg` are initialized to '0' because the LPN node does not support the Friend feature.
- `.low_power`: To understand details related to the `low_power` parameters, see Section 7.2.1. Specific parameters that are configured for this example project are listed below:
  - `.rssi_factor`: The `rssi_factor` value is configured as '2'.
  - `.received_window_factor`: The `received_window_factor` value is configured as '2'

  RSSI Factor and Receive Window Factor are used by the Friend node to calculate the Friend Offer Delay (details in section 7.2.1). Each Friend in the network calculates the Friend Offer Delay and responds to the Friend request after the calculated Friend Offer Delay in milliseconds. With this approach, a Friend node who best meets these LPN requirements sends the Friend Offer the quickest, and therefore the best possible Friend is accepted by the LPN. This way, the LPN does not have to wait a longer time (relative to other potential Friend nodes) to receive messages, and therefore saves power.

  - `.min_cache_size_log`: The `min_cache_size_log` value is set to '3'.

  This defines the minimum number of messages a Friend node will store in its Friend cache. The value is specified as a power of 2 so a setting of 3 means 8 messages (2^3).

- `.receive_delay`: The `receive_delay` value is set to 100 ms.

  Receive delay is the time provided to the Friend node for its processing before starting to respond to the LPN node.

- `.poll_timeout`: The `poll_timeout` value is set to 200 ms.

  The PollTimeout timer is used to measure the time between two consecutive requests sent by the LPN. Friendship is considered terminated if the Friend node does not receive a Friend Poll from the LPN within the PollTimeout.

- `.elements, .elements_num`: The number of elements and details of the elements used for the Low-Power ON/OFF Server project are defined in the `elements_num` and `elements` parameters of this structure. We shall look at the element configuration in Step 2.

Code 18. Low-Power LED Bluetooth Mesh Core Configuration Structure

```
wiced_bt_mesh_core_config_t  mesh_config =
{
    .company_id         = MESH_COMPANY_ID_CYPRESS,                  // Company identifier assigned by the Bluetooth SIG
    .product_id         = MESH_PID,                                 // Vendor-assigned product identifier
    .vendor_id          = MESH_VID,                                 // Vendor-assigned product version identifier
    .firmware_id        = MESH_FWID,                                // Vendor-assigned firmware version identifier
    .replay_cache_size  = MESH_CACHE_REPLAY_SIZE,                   // Number of replay protection entries, i.e. maximum number of mesh devices that can send
                                                                       application messages to this device.
#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
    .features           = WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER, // A bit field indicating the device features. In Low Power mode no Relay, no Proxy and no
                                                                       Friend
    .friend_cfg         =                                           // Empty Configuration of the Friend Feature
    {
        .receive_window = 0,                                        // Receive Window value in milliseconds supported by the Friend node.
        .cache_buf_len  = 0,                                        // Length of the buffer for the cache
        .max_lpn_num    = 0                                         // Max number of Low Power Nodes with established friendship. Must be > 0 if Friend feature is
                                                                       supported.
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 2,                                 // contribution of the RSSI measured by the Friend node used in Friend Offer Delay
                                                                       calculations.
        .receive_window_factor = 2,                                 // contribution of the supported Receive Window used in Friend Offer Delay calculations.
        .min_cache_size_log    = 3,                                 // minimum number of messages that the Friend node can store in its Friend Cache.
        .receive_delay         = 100,                               // Receive delay in 1 ms units to be requested by the Low Power Node.
        .poll_timeout          = 200                                // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
#else
    .features = WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND | WICED_BT_MESH_CORE_FEATURE_BIT_RELAY | WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER,   // Supports
                                                                       Friend, Relay and GATT Proxy
    .friend_cfg         =                                           // Configuration of the Friend Feature(Receive Window in Ms, messages cache)
    {
        .receive_window        = 20,
        .cache_buf_len         = 300,                               // Length of the buffer for the cache
        .max_lpn_num           = 4                                  // Max number of Low Power Nodes with established friendship. Must be > 0 if Friend feature is
                                                                       supported.
    },
    .low_power          =                                           // Configuration of the Low Power Feature
    {
        .rssi_factor           = 0,                                 // contribution of the RSSI measured by the Friend node used in Friend Offer Delay
                                                                       calculations.
        .receive_window_factor = 0,                                 // contribution of the supported Receive Window used in Friend Offer Delay calculations.
        .min_cache_size_log    = 0,                                 // minimum number of messages that the Friend node can store in its Friend Cache.
        .receive_delay         = 0,                                 // Receive delay in 1 ms units to be requested by the Low Power Node.
        .poll_timeout          = 0                                  // Poll timeout in 100ms units to be requested by the Low Power Node.
    },
#endif
    .gatt_client_only          = WICED_FALSE,                       // Can connect to mesh over GATT or ADV
    .elements_num  = (uint8_t)(sizeof(mesh_elements) / sizeof(mesh_elements[0])),   // number of elements on this device
    .elements      = mesh_elements                                  // Array of elements for this device
};
```

Step 2: In this step, parameters of the Mesh Core Element configuration structure, `wiced_bt_mesh_core_config_element_t`, are configured. The Low-Power ON/OFF Server Mesh Core Elements Configuration code is shown in Code 19. This example has one element.

Each parameter in this structure is explained in Section 7.2.2. Key items configured in this structure for the `low_power_led` (Low-Power ON/OFF Server) code example are as follows:

- `.properties, properties_num`: The `properties` parameter defines the Mesh element properties. In this case, the firmware version of 8 bytes as a read-only property is defined. The `mesh_element1_properties` structure contains the properties defined for this element. The value of the firmware version in the `mesh_element1_properties` structure is defined by the `mesh_prop_fw_version` array.

  ```
  uint8_t mesh_prop_fw_version[WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION] = {
                                                 '0', '6', '.', '0', '2', '.', '0', '5' };
  ```

- .sensors, .sensors_num: The Low-Power ON/OFF Server code example project does not support sensors; therefore, the `.sensors_num` and `.sensors` fields are configured to be ‘0’ and NULL respectively.

- .models, .models_num: Models for the Low-Power ON/OFF Server code example are defined in the `wiced_bt_mesh_core_config_model_t` structure. Models used in this example project are described below:
    - `WICED_BT_MESH_DEVICE`

      The Mesh Device includes the Configuration Server and Health Server models. These models always need to be included in the primary element of the Mesh device.
    - `WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER`

      This model supports the Generic User Property server. The Generic User Property server defines state instances that represent the device property of an element. Depending the defined device properties, this value may be read-only or read-write. In this case, the Low Power LED device firmware revision is configured as read-only as part of the `mesh_element1_properties` structure. This is an optional model that we have used to provide the firmware revision of the Low Power LED device to the Mesh network.
    - `WICED_BT_MESH_MODEL_POWER_ONOFF_SERVER`

      The `Generic Power OnOff Server` model extends the `Generic OnOff Server` model. The Low-Power ON/OFF Server node can be controlled by `Generic OnOff Client` or `Generic Power OnOff Client`.

Code 19. Low-Power LED Bluetooth Mesh Core Configuration Element Structure

```
wiced_bt_mesh_core_config_element_t mesh_elements[] =
{
    {
        .location = MESH_ELEM_LOC_MAIN,                              // location description as defined in the GATT Bluetooth Namespace Descriptors section of
                                                                        the Bluetooth SIG Assigned Numbers
        .default_transition_time = MESH_DEFAULT_TRANSITION_TIME_IN_MS,  // Default transition time for models of the element in milliseconds
        .onpowerup_state = WICED_BT_MESH_ON_POWER_UP_STATE_RESTORE,     // Default element behavior on power up
        .default_level = 0,                                          // Default value of the variable controlled on this element (for example power, lightness,
                                                                        temperature, hue...)
        .range_min = 1,                                              // Minimum value of the variable controlled on this element (for example power, lightness,
                                                                        temperature, hue...)
        .range_max = 0xffff,                                         // Maximum value of the variable controlled on this element (for example power, lightness,
                                                                        temperature, hue...)
        .move_rollover = 0,                                          // If true when level gets to range_max during move operation, it switches to min,
                                                                        otherwise move stops.
        .properties_num = 0,                                         // Number of properties in the array models
        .properties = mesh_element1_properties,                      // Array of properties in the element.
        .sensors_num = 0,                                            // Number of sensors in the sensor array
        .sensors = NULL,                                             // Array of sensors of that element
        .models_num = (sizeof(mesh_element1_models) / sizeof(wiced_bt_mesh_core_config_model_t)),    // Number of models in the array models
        .models = mesh_element1_models,                              // Array of models located in that element. Model data is defined by structure
                                                                        wiced_bt_mesh_core_config_model_t

    },
};

wiced_bt_mesh_core_config_property_t mesh_element1_properties[] =
{
    {
        .id          = WICED_BT_MESH_PROPERTY_DEVICE_FIRMWARE_REVISION,
        .type        = WICED_BT_MESH_PROPERTY_TYPE_USER,
        .user_access = WICED_BT_MESH_PROPERTY_ID_READABLE,
        .max_len     = WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION,
        .value       = mesh_prop_fw_version
    },
};



wiced_bt_mesh_core_config_model_t   mesh_element1_models[] =
{
    WICED_BT_MESH_DEVICE,
    WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER,
    WICED_BT_MESH_MODEL_POWER_ONOFF_SERVER,
};
```

Step 3: This step registers application functions for the code example.

In this project, the Mesh application initialization and Mesh LPN Sleep callback functions are defined in the `wiced_bt_mesh_app_func_table_t` structure as shown in Code 20. The Mesh Core Library performs default actions as defined in *mesh_application.c* for the items that do not have any callback function registered in this structure.

Code 20. Low-Power LED Bluetooth Mesh Application Function Table

```
wiced_bt_mesh_app_func_table_t wiced_bt_mesh_app_func_table =
{
    mesh_app_init,          // application initialization
    NULL,                   // Default SDK platform button processing
    NULL,                   // GATT connection status
    NULL,                   // attention processing
    NULL,                   // notify period set
    NULL,                   // WICED HCI command
#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
    mesh_low_power_led_lpn_sleep,// LPN sleep
#else
    NULL,
#endif
    NULL                    // factory reset
};
```

Step 4: This step implements the application callback functions.

There are four application callback functions defined for the low_power_led code example, `mesh_app_init()`, `mesh_low_power_led_lpn_sleep()`, `wakeup_timer_cb()` and `mesh_low_power_led_sleep_poll()`. Each is described below.

1. `mesh_app_init()`

   This function shown in Code 21 performs the following actions:

   - `mesh_app_init()` function is executed for two reasons:
     1. Power cycle or POR (power on reset)
     2. Wakeup from HID Off mode, which involves a reset of the device similar to POR

     Use the `wiced_hal_mia_is_reset_reason_por()` API function to understand the reason for reset.
   - Configure the device name and device appearance. In this case, the device name is configured as "Low Power LED" and appearance is configured as "APPEARANCE GENERIC TAG"
   - The Power ONOFF Server Mesh Model is initialized by passing a callback function, `mesh_app_message_handler`, to `wiced_bt_mesh_model_power_onoff_server_init`. The Mesh Models Library will trigger the registered callback function, `mesh_app_message_handler` when messages are received for this model. The `mesh_app_message_handler` function is described in Step 5 of this section.
   - The `mesh_prop_fw_revision` properties array is configured with the desired values and the Mesh Model User Property Server model is initialized by using `wiced_bt_mesh_model_property_server_init`.
   - The LED is initialized in this function. The Low-Power ON/OFF Server node uses a GPIO to control the ON/OFF state of the LED. LED-specific functions, initialization, and control logic for this application are implemented in *led_control.c*. A GPIO is used to drive the state of the LED instead of a PWM to save power as this is an LPN implementation.
   - After configuring the required information in the steps above, the device is now in the idle state. Sleep configuration is done using the `wiced_sleep_configure()` API with the `wiced_sleep_config_t` structure as a parameter which defines wake GPIO, wake source, callback function for sleep permission, etc. The parameters for `wiced_sleep_config_t` are described below:
     - `sleep_mode` is configured to `WICED_SLEEP_MODE_NO_TRANSPORT`; this setting is used for HID use cases. Note that sleep is disallowed when a transport is connected, for example a UART. The user must disable all transports to allow the device to enter HID Off mode.
     - `device_wake_source` is configured as `WICED_SLEEP_WAKE_SOURCE_GPIO`.
     - `device_wake_gpio_num` is configured as `WICED_GPIO_PIN_BUTTON`, which in this case is `WICED_P00`.
     - `device_wake_mode` is configured as `WICED_SLEEP_WAKE_ACTIVE_HIGH`, which means that an active HIGH interrupt wakes up the device.
     - `host_wake` is configured to `WICED_SLEEP_WAKE_ACTIVE_HIGH`, which means that an active HIGH interrupt from the external host wakes up the device.
     - `sleep_permit_handler` is configured to `mesh_low_power_led_sleep_poll`. This callback function is called by the sleep framework to poll for sleep permission.
   - A millisecond timer, `lpn_wake_timer`, is initialized with `wakeup_timer_cb` configured as the callback function.

Code 21. Low-Power LED `mesh_app_init` Mesh Application Initialization Function

```
void mesh_app_init(wiced_bool_t is_provisioned)
{
#if 0
    extern uint8_t wiced_bt_mesh_model_trace_enabled;
    wiced_bt_mesh_model_trace_enabled = WICED_TRUE;
#endif
    wiced_result_t  result;

    // This means that device came out of HID off mode and it is not a power cycle
    if(wiced_hal_mia_is_reset_reason_por())
    {
        WICED_BT_TRACE("start reason: reset\n");
    }
    else
    {
#if CYW20819A1
        if(wiced_hal_mia_is_reset_reason_hid_timeout())
        {
            WICED_BT_TRACE("Wake from HID off: timed wake\n");
        }
        else
#endif
        {
            // Check if we wake up by GPIO
            WICED_BT_TRACE("Wake from HID off, interrupt:%d\n", wiced_hal_gpio_get_pin_interrupt_status(WICED_GPIO_PIN_BUTTON));
        }
    }

#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
    wiced_bt_cfg_settings.device_name = (uint8_t *)"Low Power LED";
#else
    wiced_bt_cfg_settings.device_name = (uint8_t *)"On/Off LED";
#endif
    wiced_bt_cfg_settings.gatt_cfg.appearance = APPEARANCE_GENERIC_TAG;

    // Adv Data is fixed. Spec allows to put URI, Name, Appearance and Tx Power in the Scan Response Data.
    if (!is_provisioned)
    {
        wiced_bt_ble_advert_elem_t  adv_elem[3];
        uint8_t                     buf[2];
        uint8_t                     num_elem = 0;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_NAME_COMPLETE;
        adv_elem[num_elem].len         = (uint16_t)strlen((const char*)wiced_bt_cfg_settings.device_name);
        adv_elem[num_elem].p_data      = wiced_bt_cfg_settings.device_name;
        num_elem++;

        adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_APPEARANCE;
        adv_elem[num_elem].len         = 2;
        buf[0]                         = (uint8_t)wiced_bt_cfg_settings.gatt_cfg.appearance;
        buf[1]                         = (uint8_t)(wiced_bt_cfg_settings.gatt_cfg.appearance >> 8);
        adv_elem[num_elem].p_data      = buf;
        num_elem++;

        wiced_bt_mesh_set_raw_scan_response_data(num_elem, adv_elem);
    }

    mesh_prop_fw_version[0] = 0x30 + (WICED_SDK_MAJOR_VER / 10);
    mesh_prop_fw_version[1] = 0x30 + (WICED_SDK_MAJOR_VER % 10);
    mesh_prop_fw_version[2] = 0x30 + (WICED_SDK_MINOR_VER / 10);
    mesh_prop_fw_version[3] = 0x30 + (WICED_SDK_MINOR_VER % 10);
    mesh_prop_fw_version[4] = 0x30 + (WICED_SDK_REV_NUMBER / 10);
    mesh_prop_fw_version[5] = 0x30 + (WICED_SDK_REV_NUMBER % 10);
    // convert 12 bits of BUILD_NUMMBER to two base64 characters big endian
    mesh_prop_fw_version[6] = wiced_bt_mesh_base64_encode_6bits((uint8_t)(WICED_SDK_BUILD_NUMBER >> 6) & 0x3f);
    mesh_prop_fw_version[7] = wiced_bt_mesh_base64_encode_6bits((uint8_t)WICED_SDK_BUILD_NUMBER & 0x3f);

    led_control_init(LED_CONTROL_TYPE_ONOFF);

    wiced_bt_mesh_model_power_onoff_server_init(MESH_LOW_POWER_LED_ELEMENT_INDEX, mesh_low_power_led_message_handler, TRANSITION_INTERVAL, is_provisioned);

#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
    if (!do_not_init_again)
    {
        WICED_BT_TRACE("Init once \n");

        // Configure to sleep as the device is idle now
        app_state.lpn_sleep_config.sleep_mode = WICED_SLEEP_MODE_NO_TRANSPORT;
        app_state.lpn_sleep_config.device_wake_mode = WICED_GPIO_BUTTON_WAKE_MODE;
        app_state.lpn_sleep_config.device_wake_source = WICED_SLEEP_WAKE_SOURCE_GPIO;
        app_state.lpn_sleep_config.device_wake_gpio_num = WICED_GPIO_PIN_BUTTON;
        app_state.lpn_sleep_config.host_wake_mode = WICED_SLEEP_WAKE_ACTIVE_HIGH;
        app_state.lpn_sleep_config.sleep_permit_handler = mesh_low_power_led_sleep_poll;
        app_state.lpn_sleep_config.post_sleep_cback_handler = NULL;

        if (WICED_BT_SUCCESS != wiced_sleep_configure(&app_state.lpn_sleep_config))
        {
            WICED_BT_TRACE("Sleep Configure failed\r\n");
        }

        wiced_init_timer(&app_state.lpn_wake_timer, wakeup_timer_cb, 0, WICED_MILLI_SECONDS_TIMER);

        do_not_init_again = WICED_TRUE;
    }
#endif}
```

2. `mesh_low_power_led_lpn_sleep()`

   The Mesh Core Library triggers this function for the Low-Power ON/OFF Server application project to let the application project decide whether it wants to enter a low-power state. The Low-Power ON/OFF Server project can go to a sleep state for a duration not to exceed the `max_sleep_duration` (time in milliseconds), which is passed as a parameter to this function.

HID-OFF or ePDS sleep mode will be executed based on the `max_sleep_duration` value, which is passed as a parameter to this function. Based on the experimental data, ePDS mode achieves optimal/lower current consumption for a sleep duration of up to two minutes. HID-Off mode provides optimal/lower current consumption beyond sleep duration of two minutes.

- Existing code in the application project configures the device to use ePDS mode if `max_sleep_duration` value is less than two minutes. In this case, `lpn_wake_timer` is configured to trigger an interrupt after `max_sleep_duration`.
- Existing code in the application project configures the device to use HID Off mode if `max_sleep_duration` is more than two minutes. In this case, the device enters HID Off mode by calling `wiced_sleep_enter_hid_off(max_sleep_duration, WICED_GPIO_PIN_BUTTON, 1))` as shown in Code 22. The device performs a soft reset after `max_sleep_duration` interval or with a button press.

3. `Wakeup_timer_cb()`

- In this application project, when the device enters ePDS mode, the `lpn_wake_timer` is configured to wake the device after `max_sleep_duration`. The callback function `wakeup_timer_cb()` is executed after the device exits ePDS sleep mode. In this callback function, the device state is set to `MESH_LPN_STATE_NOT_IDLE` to let the sleep framework know that the device is not ready to enter sleep mode. Now, the Mesh stack establishes a connection with its Friend node, receives data packets addressed to this node, and triggers the `mesh_low_power_led_lpn_sleep` function. In this function, the device is again configured to enter ePDS sleep mode (if `max_sleep_duration` is less than two minutes) with the device state modified to `MESH_LPN_STATE_IDLE`. The cycle explained in this paragraph repeats once the device exits ePDS sleep mode.
- In this application project, when the device enters HID-Off mode, the `wakeup_timer_cb()` callback function is not triggered because most of the device resources are turned OFF in the HID-Off state which includes timers. A soft reset of the entire device is triggered after the device exits HID-Off sleep mode.

4. `Mesh_low_power_led_sleep_poll()`

- `WICED_SLEEP_MAX_TIME_TO_SLEEP` is returned when the sleep framework requests the allowed sleep time with the device in the idle state (`MESH_LPN_STATE_IDLE`). Sleep requests from the sleep framework are rejected (`WICED_SLEEP_NOT_ALLOWED`) when the device is not in the idle state.
- `WICED_SLEEP_ALLOWED_WITHOUT_SHUTDOWN` is returned when the sleep framework requests permission to sleep when the device state is idle (`MESH_LPN_STATE_IDLE`). This means that sleep is allowed but shutdown mode is not allowed. Shutdown mode puts the device to sleep and turns off hardware along with most parts of SRAM, but as we have to be able to resume operation, `WICED_SLEEP_ALLOWED_WITHOUT_SHUTDOWN` is used for this project.

Code 22. Low-Power LED Sleep/HID-Off Function

```
#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
void mesh_low_power_led_lpn_sleep(uint32_t max_sleep_duration)
{
    // Generally speaking, if sleep timer bigger than 2mins, then hid-off will save more power. But it's up to your design.
    if (max_sleep_duration < 120000)//2mins
    {
        wiced_stop_timer(&app_state.lpn_wake_timer);
        wiced_start_timer(&app_state.lpn_wake_timer, max_sleep_duration);
        WICED_BT_TRACE("Get ready to go into ePDS sleep, duration=%d\n\r", max_sleep_duration);
        app_state.lpn_state = MESH_LPN_STATE_IDLE;
    }
    else
    {
        WICED_BT_TRACE("Entering HID-OFF for max_sleep_duration: %d\r\n", max_sleep_duration);
        if (WICED_SUCCESS != wiced_sleep_enter_hid_off(max_sleep_duration, WICED_GPIO_PIN_BUTTON, 1))
        {
            WICED_BT_TRACE("Entering HID-Off failed\n\r");
        }
    }
}

/*
 * wakeup timer callback.
 * ePDS is default sleep mode(current is about 10uA).
 */
static void wakeup_timer_cb(TIMER_PARAM_TYPE arg)
{
    WICED_BT_TRACE("ePDS wake up!!!\n");
    app_state.lpn_state = MESH_LPN_STATE_NOT_IDLE;
    wiced_stop_timer(&app_state.lpn_wake_timer);
}


/*
 * Sleep permission polling time to be used by firmware
 */
static uint32_t mesh_low_power_led_sleep_poll(wiced_sleep_poll_type_t type)
{
    uint32_t ret = WICED_SLEEP_NOT_ALLOWED;

    switch (type)
    {
    case WICED_SLEEP_POLL_TIME_TO_SLEEP:
        if (app_state.lpn_state == MESH_LPN_STATE_NOT_IDLE)
        {
            WICED_BT_TRACE("!");
            ret = WICED_SLEEP_NOT_ALLOWED;
        }
        else
        {
            WICED_BT_TRACE("@\n");
            ret = WICED_SLEEP_MAX_TIME_TO_SLEEP;
        }
        break;
    case WICED_SLEEP_POLL_SLEEP_PERMISSION:
        if (app_state.lpn_state == MESH_LPN_STATE_IDLE)
        {
            WICED_BT_TRACE("#\n");
            ret = WICED_SLEEP_ALLOWED_WITHOUT_SHUTDOWN;
        }

        break;
    }
    return ret;
}
#endif
```

Step 5: This step implements the Mesh message callback functions.

The Low Power ON/OFF Server node receives Mesh messages in the callback function, `mesh_app_message_handler`, registered during initialization. Details of the Mesh message for this model are received as parameters to this callback function. The `mesh_app_message_handler` is triggered by the Mesh Models Library on receipt of Mesh messages intended for the BLE Mesh Power OnOff Server model. If this callback function receives the `WICED_BT_MESH_ONOFF_SET` event, the `mesh_low_power_led_process_set` function is called, which interprets the received message and calls the `led_control_set_onoff` function with the desired LED state as the parameter. The `led_control_set_onoff` function controls the LED state based on the received Mesh messages.

Code 23. Low-Power LED Mesh Message Callback Functions

```
void mesh_low_power_led_message_handler(uint8_t element_idx, uint16_t event, void *p_data)
{
    switch (event)
    {
    case WICED_BT_MESH_ONOFF_STATUS:
        mesh_low_power_led_process_status(element_idx, (wiced_bt_mesh_onoff_status_data_t *)p_data);
        break;

    case WICED_BT_MESH_ONOFF_SET:
        break;

    default:
        WICED_BT_TRACE("unknown\n");
    }
}

/*
 * This function is called when command to change state is received over mesh.
 */
void mesh_low_power_led_process_status(uint8_t element_idx, wiced_bt_mesh_onoff_status_data_t *p_status)
{
    led_control_set_onoff(p_status->present_onoff);
}

void led_control_set_onoff(uint8_t onoff_value)
{
    if(onoff_value == 1)//led is on
    {
        wiced_hal_gpio_configure_pin(led_pin, GPIO_OUTPUT_ENABLE, GPIO_PIN_OUTPUT_LOW);
    }
    else if(onoff_value == 0)//led is off
    {
        wiced_hal_gpio_configure_pin(led_pin, GPIO_OUTPUT_ENABLE, GPIO_PIN_OUTPUT_HIGH);
    }
}
```

8.3.2 Dimmable Light Code Example

The Dimmable Light used in this use case is the same as the one used in Use Case #1. See Section 8.2.2. However, the Friend feature that is enabled in the code example is not utilized in Use Case #1 but it is used in Use Case #2. The Friend feature of the Dimmable Light Bulbs allows the Low-Power ON/OFF Server node to establish friendship with one of the Dimmable Light Bulbs in the network so that it can receive messages that are intended for the Low-Power ON/OFF Server node.

9 Programming the Evaluation Boards

This section details the steps required to program the CYBT-213043-MESH EZ-BT Bluetooth Mesh Evaluation Board. The CYBT-213043-02 module on the CYBT-213043-MESH Mesh evaluation boards include a UART-based bootloader in the on-chip ROM, and therefore does not require an external programmer. Boards are programmed over the UART interface using the on-board Cypress USB-UART device. For more details on the programming hardware included on the Mesh Evaluation boards, see sections 5.1.1 and 5.1.4.

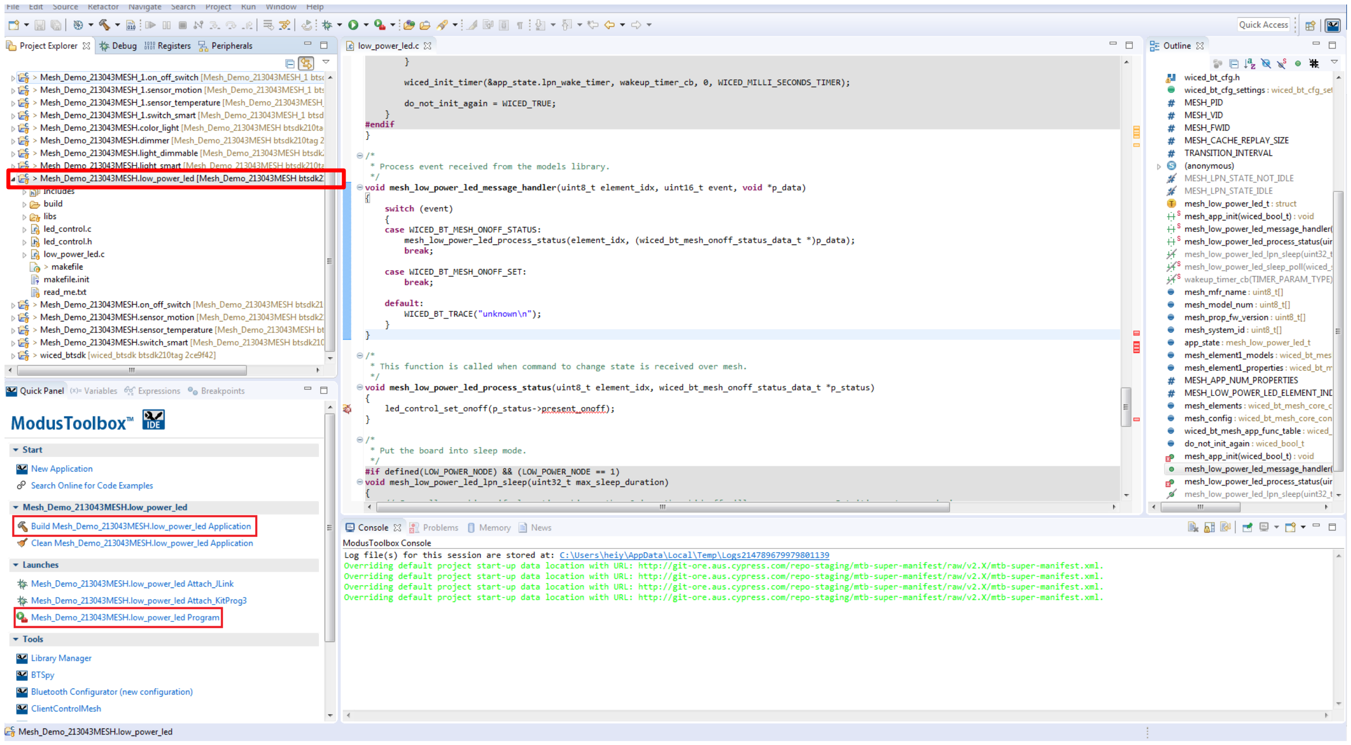
9.1 Programing Code Examples

You should first create the wiced_btsdk project and the Mesh Demo Application group. If you have already done these per sections 4.1.1 and 4.1.4, you can skip steps 1-6.

1. Open ModusToolbox.
2. Select **File** > **New** > **ModusToolbox Application**.
3. Choose the Board Support Package (BSP) as “CYBT-213043-MESH”, and click **Next**.
4. Choose “wiced_btsdk” and click **Create**.

   Note: Do not change the name of “wiced_btsdk” project. All Bluetooth application projects use this project name in their application makefiles.
5. When this step is completed, select “Mesh-Demo-213043MESH” and click **Create** again.
6. Click **Close**. All of the projects will be imported into the IDE.
7. Select the application that you want to build from the project explorer, build the application, and then program it as shown in Figure 29.

Figure 29. Build and Program Applications in Eclipse IDE for ModusToolbox


Project Explorer
Debug
Registers
Peripherals
Mesh_Demo_213043MESH.low_power_led [Mesh_Demo_213043MESH btsdk2
Quick Panel
Variables
Expressions
Breakpoints
ModusToolbox™
Start
New Application
Search Online for Code Examples
Mesh_Demo_213043MESH.low_power_led
Build Mesh_Demo_213043MESH.low_power_led Application
Clean Mesh_Demo_213043MESH.low_power_led Application
Launches
Mesh_Demo_213043MESH.low_power_led Attach_JLink
Mesh_Demo_213043MESH.low_power_led Attach_KitProg3
Mesh_Demo_213043MESH.low_power_led Program
Tools
Library Manager
BTSpy
Bluetooth Configurator (new configuration)
ClientControlMesh
low_power_led.c
Outline
Quick Access
Console
Problems
Memory
News

```
        }

        wiced_init_timer(&app_state.lpn_wake_timer, wakeup_timer_cb, 0, WICED_MILLI_SECONDS_TIMER);

        do_not_init_again = WICED_TRUE;
    }
#endif
}

/*
 * Process event received from the models library.
 */
void mesh_low_power_led_message_handler(uint8_t element_idx, uint16_t event, void *p_data)
{
    switch (event)
    {
    case WICED_BT_MESH_ONOFF_STATUS:
        mesh_low_power_led_process_status(element_idx, (wiced_bt_mesh_onoff_status_data_t *)p_data);
        break;

    case WICED_BT_MESH_ONOFF_SET:
        break;

    default:
        WICED_BT_TRACE("unknown\n");
    }
}

/*
 * This function is called when command to change state is received over mesh.
 */
void mesh_low_power_led_process_status(uint8_t element_idx, wiced_bt_mesh_onoff_status_data_t *p_status)
{
    led_control_set_onoff(p_status->present_onoff);
}

/*
 * Put the board into sleep mode.
 */
#if defined(LOW_POWER_NODE) && (LOW_POWER_NODE == 1)
void mesh_low_power_led_lpn_sleep(uint32_t max_sleep_duration)
{
```

```
ModusToolbox Console
Log file(s) for this session are stored at: C:\Users\heiy\AppData\Local\Temp\Logs2147896799798011139
Overriding default project start-up data location with URL: http://git-ore.aus.cypress.com/repo-staging/mtb-super-manifest/raw/v2.X/mtb-super-manifest.xml.
Overriding default project start-up data location with URL: http://git-ore.aus.cypress.com/repo-staging/mtb-super-manifest/raw/v2.X/mtb-super-manifest.xml.
Overriding default project start-up data location with URL: http://git-ore.aus.cypress.com/repo-staging/mtb-super-manifest/raw/v2.X/mtb-super-manifest.xml.
Overriding default project start-up data location with URL: http://git-ore.aus.cypress.com/repo-staging/mtb-super-manifest/raw/v2.X/mtb-super-manifest.xml.
```

9.2 Performing a Recovery Procedure

In some cases, the normal firmware download procedure fails even though all connections and switches are correct. This may happen as a result of flash corruption or power loss during a normal firmware download process. It may also happen if the device is in a low-power mode. If this happens, you may need to recover the device (i.e., force it into a mode where it is listening for programming instructions). Do the following to put the device into recovery mode.

1. Ensure that the HCI UART is connected to the module (via direct connection) or evaluation board (via configuration switches). This step is required only if you want to reprogram your module after the following recovery procedure has completed. The CYBT-213043-MESH kit has the configuration switches set for the HCI UART by default.
2. Press and hold the RECOVER button SW2 (Red button).
3. Press and release the RESET button SW1 (Blue button).
4. Release the RECOVER button (Red button).
5. Re-program the board in ModusToolbox.

10 Testing the Code Examples

This section describes how to test the two use cases outlined in this application note.

- Use Case #1: Dimmer Switch Controlling Three Dimmable Light Bulbs
- Use Case #2: Low-Power ON/OFF Server with Three Dimmable Light Bulbs

To test these use cases, make sure that you have all hardware and software as described in Section 8.1.

Note: In this application note, we will use the Cypress iOS Helper app for all three use cases. If you are using the Windows Helper Application(s), download MeshClient and ClientControlMesh App User Guide for instructions on how to use these applications. If you are using the Android Helper application, see Using Android Helper App.

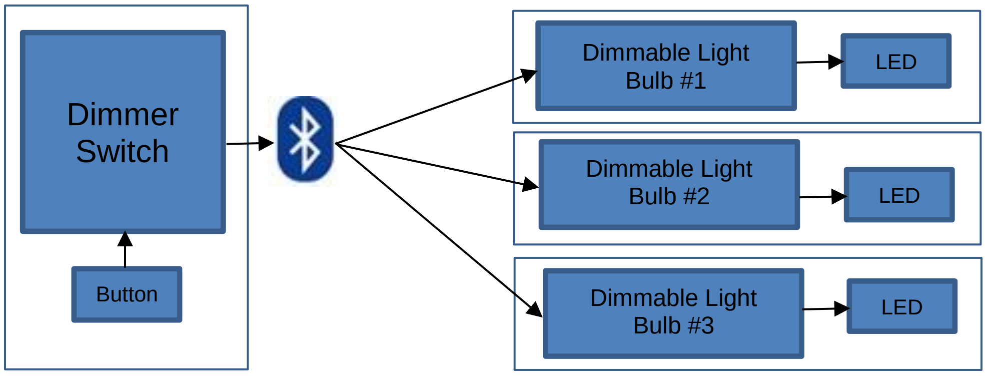
Before beginning the test, mark the four kits as MESH BOARD-1, MESH BOARD-2, MESH BOARD-3 and MESH BOARD-4.

10.1 Testing Use Case #1

Follow the steps below to download and demonstrate the Mesh functionality described in Use Case #1.

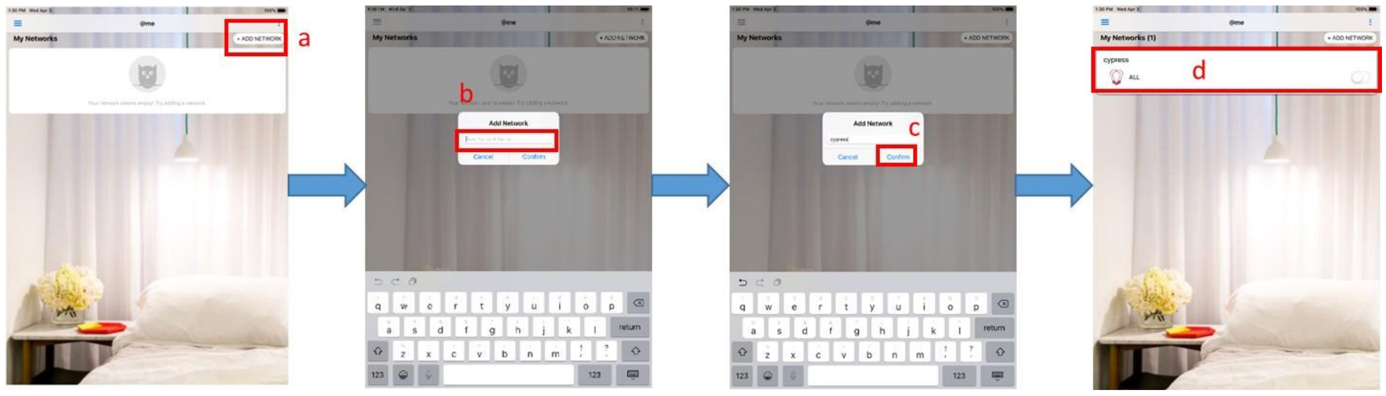
1. Program the CYBT-213043-MESH boards with following projects. Refer to programming instructions in Section 9 to program the Mesh evaluation boards.
   - `dimmer` – MESH BOARD-1
   - `light_dimmable` – MESH BOARD-2
   - `light_dimmable` – MESH BOARD-3
   - `light_dimmable` – MESH BOARD-4

Figure 30. Use Case #1 Setup: Dimmer Switch with Three Dimmable Light Bulbs



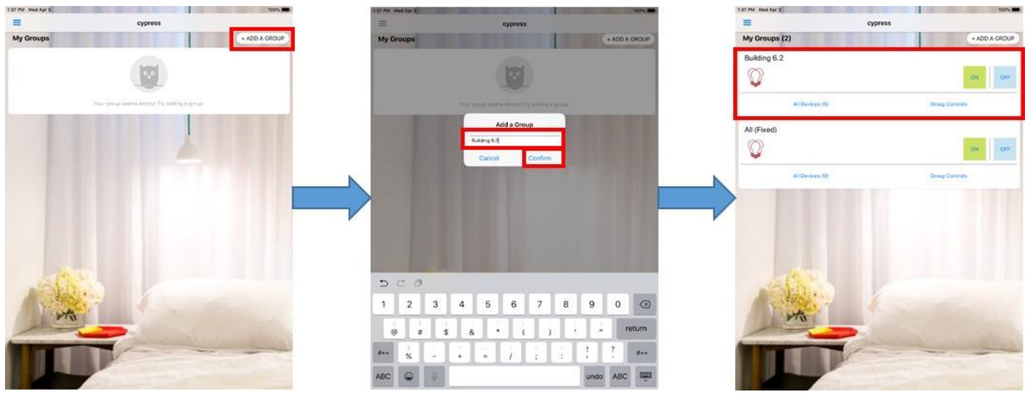
2. Make sure that your iOS device has the MeshLightingController application installed. For steps to install the MeshLightingController application on your iOS device, see iOS Helper App Installation.
3. Open the MeshLightingController app on your iOS device.
4. Do the following to create a Mesh network:
   a. Tap **+ADD NETWORK** located towards the top right of the MeshLightingController application.
   b. A new window pops up where a network can be created. In this case, the network name is assigned as “cypress”; you can use your own network name here as desired.
   c. Tap **Confirm**.
   d. Observe that a new network with the name “cypress” is created with a default group called “All”

Figure 31. Creating a Mesh Network



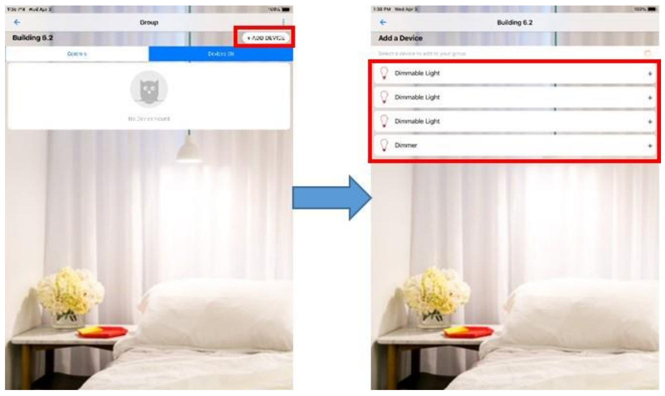
5. Next, tap on the **cypress** network on the MeshLightingController app home. A new screen will appear as shown below in Figure 32. Tap the **+Add A GROUP”** button located towards the top right side of the MeshLightingController application. Enter your desired name for this group and tap **Confirm**. In this example below, the name of the group is “Building 6.2”.

Figure 32. Creating a Group in the Mesh Network



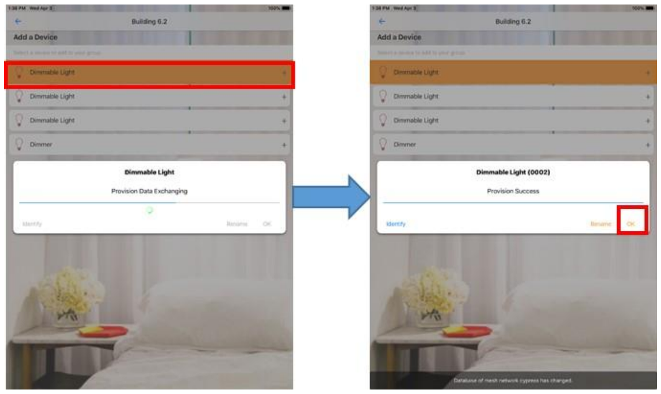
6. After creating the Mesh Network and Group, you can add the Mesh devices to the network/group. Tap on Building 6.2 group and then tap on the **+ADD DEVICE** button to see the unprovisioned Mesh devices in the vicinity as shown in Figure 33. You will see the advertisement of the four Mesh Evaluation boards that we programmed in Step 1 of this section.

Figure 33. Scan for Unprovisioned Mesh Devices



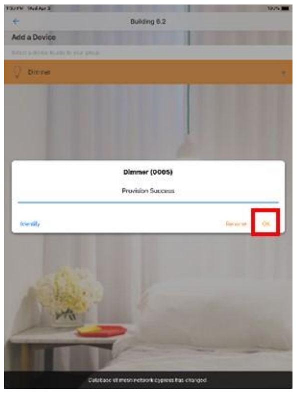
7. Tap on the Dimmable Light device to start the “Provisioning” and “Configuration” process. Observe the status message “Provision Success” after the process is complete. Now, the Dimmable Light Bulb with the name Dimmable Light (002) is part of the Mesh Group. The “(002)” shown in this example is an auto-generated post-fix representing the element unique address for the node.

Figure 34. Provision a Dimmable Light Bulb Node



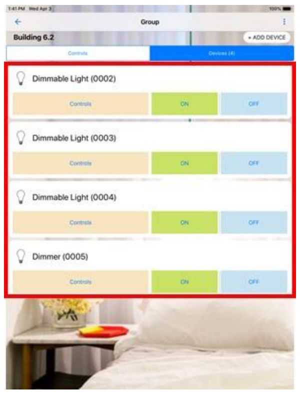
8. Repeat Step 6 and 7 to add the other two Dimmable Light Bulbs and the Dimmer Switch.

Figure 35. Provision Dimmer Switch Node



9. At this point, all four nodes are part of the Mesh Group (in this case, the name of the group is “Building 6.2”) as shown below. The Dimmer Switch is automatically configured to control the Dimmable Light Bulbs as both Dimmer and LightDimmable devices are part of the same group.

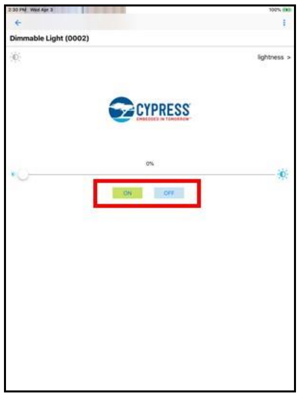
Figure 36. All Provisioned Nodes in The Group



10. The LED on the Dimmable Light Mesh nodes can be controlled in the following ways:

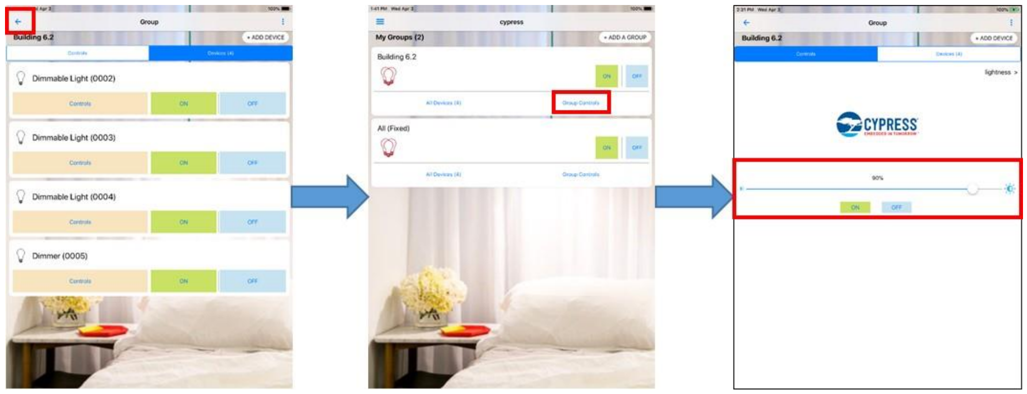
   a. Tap on the ON/OFF button corresponding to the Dimmable Light on the MeshLightController app and observe that each Dimmable Light Bulb can be controlled independently. Tap on any Dimmable Light device, in this case Dimmable Light (0002). The level setting of the LED on Dimmable Light Bulb can be controlled by varying the linear slider between 0 to 100%.

Figure 37. Controlling Dimmable Light Bulb



   b. For group control, tap twice on the blue back arrow located towards the top left side of the MeshLightingController app screen and select “Group Controls” to control a group’s ON/OFF status or level.

Figure 38. Controlling All Three Dimmable Light Bulbs in Group “Building 6.2”



   c. The Dimmer Switch node can also control the Dimmable Light Bulbs directly without the need of the Cypress iOS Helper app. The Dimmer Switch controls the LED on the three Dimmable Light Bulbs through the User Button on the Dimmer Switch evaluation board. The User Button actions and associated Mesh messages that are sent out from the Dimmer Switch are detailed below:

      I. A short press and release (500 ms or less) of the User Button on the Dimmer Switch node will toggle the ON/OFF state of the LED on the Dimmable Light Bulb nodes.

      II. A long press of User Button on the Dimmer Switch node sends a 12.5% level increase command every 500 milli-seconds. It takes 4 seconds for the level of the Dimmable Light Bulb Nodes to increase from 0% to 100%. The Dimmer Switch node will stop sending commands after 100% level setting is achieved.

      III. After the Dimmer Switch has reached the minimum or maximum level setting (0% or 100%), releasing the User Button and repeating a long press will repeat the inverse level controls to the Dimmable Light Bulbs. For example, in Step 10.c.ii above, the Dimmable Light Bulbs level setting were increased to 100%. After releasing the User Button and repeating the long press of the User Button, the level setting will decrease from 100% to 0% in 12.5% decrements every 500 milli-seconds.

      Note: If the button is held down for more than 15 seconds, a factory reset will be performed which will remove all provisioning information from the device. If this happens, it will need to be reprovisioned for it to control the dimmable light bulbs again.

10.2 Testing Use Case #2

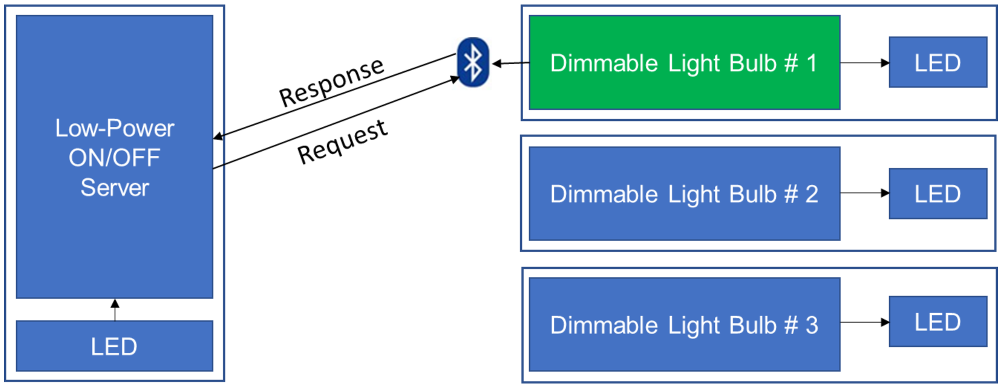
Follow the steps below to download and demonstrate the Mesh functionality described in Use Case #2.

1. Program the CYBT-213043-MESH boards with the following applications. Refer to instructions in Section 9 to program the Mesh evaluation boards.

   - `low_power_led` – MESH BOARD-1
   - `light_dimmable` – MESH BOARD-2
   - `light_dimmable` – MESH BOARD-3
   - `light_dimmable` – MESH BOARD-4

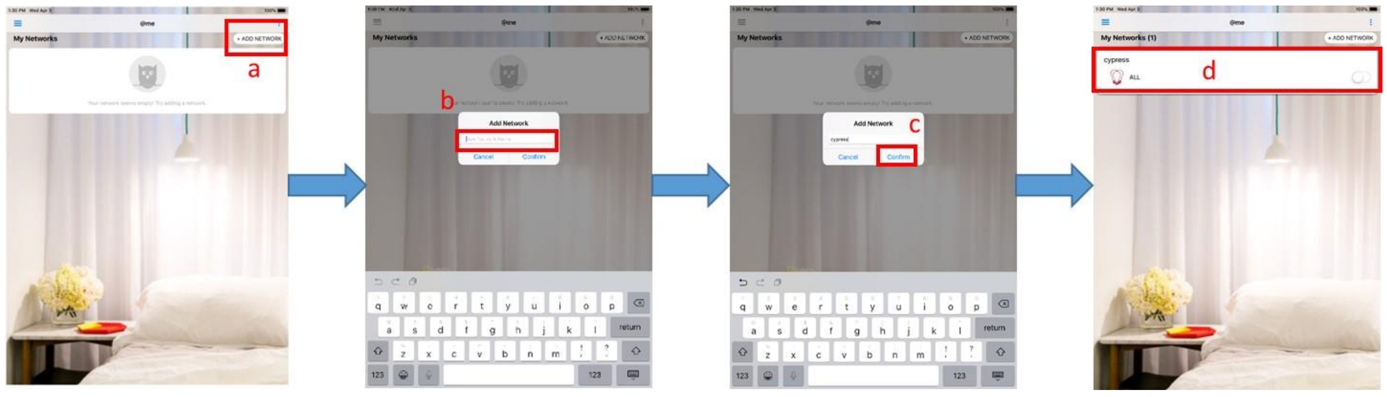
   Figure 39 details the setup for Use Case #2. In the illustration, Dimmable Light Bulb #1 is assumed to be the selected Friend node for the Low-Power ON/OFF Server node. In your setup, the Friend will be automatically selected based on the Friendship establishment process as documented in Section 7.2.1.6.

   Figure 39. Testing Use Case #2: Low-Power ON/OFF Server with Three Dimmable Light Bulbs



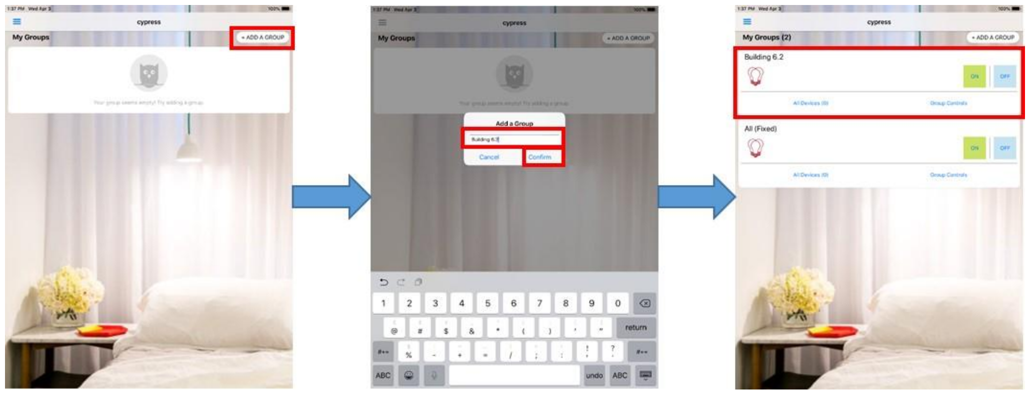
2. Make sure that your iOS device has the MeshLightingController app. Steps to install MeshLightingController app on your iOS device are detailed in iOS Helper App Installation.
3. Open the MeshLightingController app on your iOS device.
4. Follow below steps to create a Mesh network.
   a. Tap on the **+ADD NETWORK** button located towards the top right of the MeshLightingController app screen.
   b. A new window pops up where a network can be created. In this case, the network name is assigned as “cypress”; you can use your own network name here as desired.
   c. Tap confirm.
   d. Observe that a new network with the name “cypress” is created.

Figure 40. Creating a Mesh Network



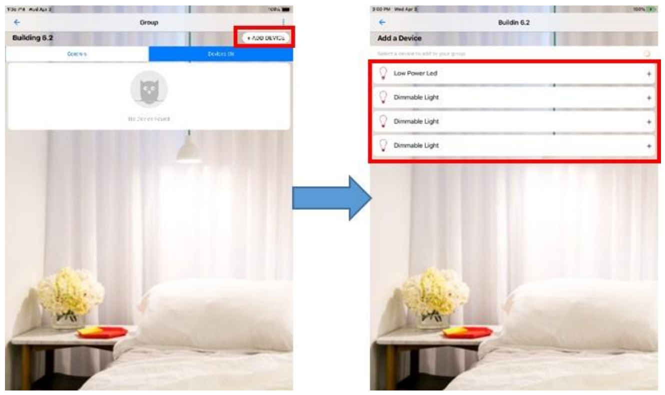
5. Next, tap on the “cypress” network on the MeshLightingController app home screen. A new screen will appear as shown in Figure 41. Tap on the **+Add A GROUP** button located towards the top right side of the MeshLightingController app screen. Enter your desired name for this group and tap **Confirm**. In this example below, the name of the group is “Building 6.2”.

Figure 41. Creating a Group in the Mesh Network



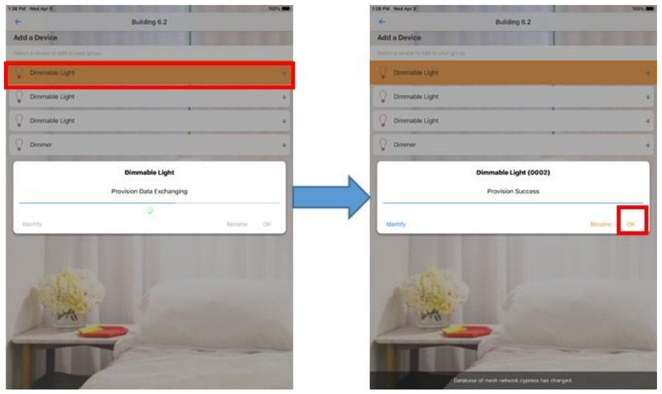
6. After creating the Mesh Network and Group, you can add the Mesh devices to the network/group. Tap on the Building 6.2 group and then tap on the **+ADD DEVICE** button to see the unprovisioned Mesh devices in the vicinity as shown in Figure 42. You will see the advertisement of the four Mesh Evaluation boards that programmed in Step 1 of this section.

Figure 42. Scanning Unprovisioned Mesh Devices



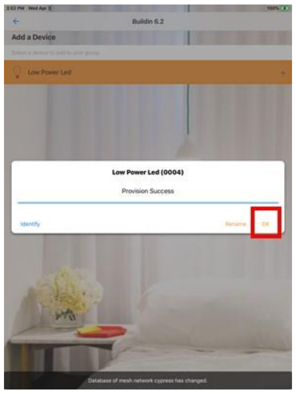
7. Tap on the Dimmable Light device to start the “Provisioning” and “Configuration” process. Observe that status message “Provision Success” after the process is complete. Now the Dimmable Light Bulb with the name Dimmable Light (002) is part of the Mesh network. The “(002)” shown in this example is an auto-generated post-fix representing the element unique address for the node.

Figure 43. Adding Dimmable Light Bulb to the Group



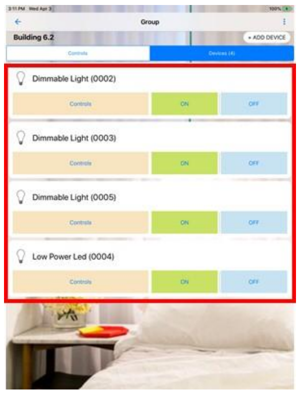
8. Repeat steps 6 and 7 to add other two Dimmable Light Bulbs and the Low-Power ON/OFF Server.

Figure 44. Adding Low-Power ON/OFF Server to the Group



9. At this point, all four nodes are part of the Mesh Group (in this case the name of the group is “Building 6.2”) as shown below. The Low-power ON/OFF Server automatically creates friendship with one of the Dimmable Light Bulbs. The Mesh Core Library will take care of the Friendship establishment process.

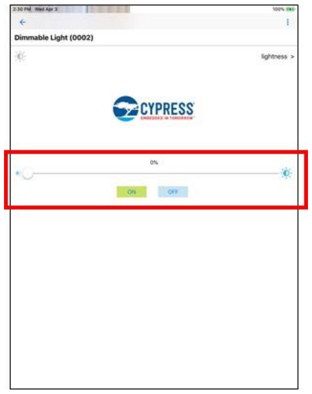
Figure 45. One Low-Power ON/OFF Server and Three Dimmable Light Bulbs in the Group



10. The LED on the Dimmable Light Bulbs and the Low-Power ON/OFF Server can be controlled in the following ways:

    a. Tap on the ON/OFF button corresponding to the Dimmable Light Bulb in the MeshLightController app screen and observe that the LED on the corresponding Dimmable Light Bulb boards can be controlled independently. Tap on any Dimmable Light, in this case Dimmable Light (0002), and observe a linear slider. The level setting of each Dimmable Light Bulb node can be controlled by varying the linear slider between 0 to 100%.

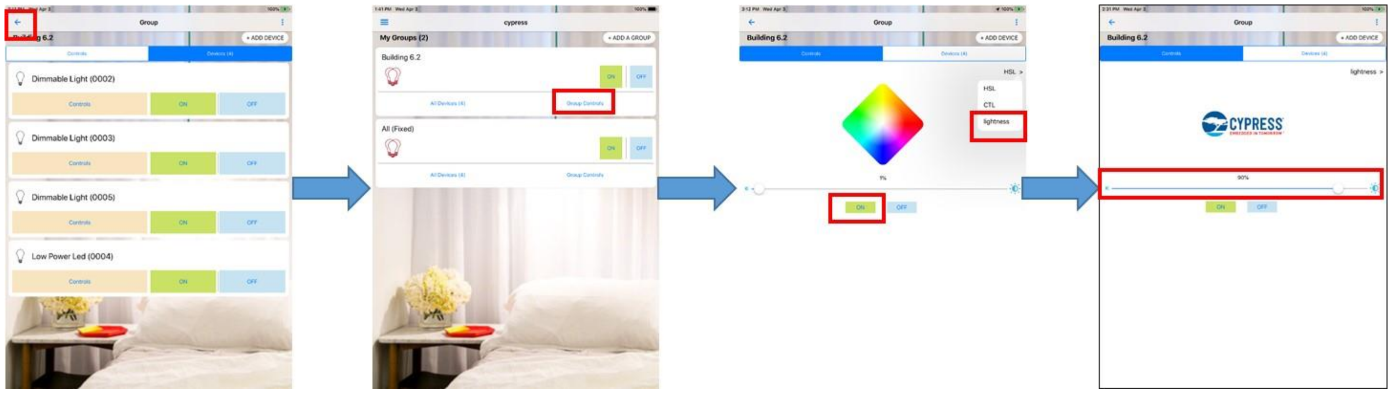
Figure 46. Controlling One Dimmable Light Bulb



    b. Tap on the **ON** button corresponding to Low-Power ON/OFF Server. Observe that there is a delay of up to 17 seconds for the LED to turn ON. This is due to the fact that the Low-Power ON/OFF Server will poll its Friend for any cached messages approximately every 17 seconds. The Low-Power ON/OFF Server will stay in the lowest possible power state during the period when it is not polling its Friend (Sleep period). The delay will not be constant since it depends when in the sleep cycle the message is sent. Likewise, click the OFF button and observe the delay in turning the LED OFF.

    c. For group control, tap twice on the blue back arrow located towards the top left side of the MeshLightingController app screen. Tap the **ON** button under group controls; you will observe that Dimmable Light Bulbs' LEDs turn ON instantly, but the LED on the Low-Power LED could take up to 17 seconds to turn ON. Tap on the lightness option to control the level of the LEDs on the Dimmable Light Bulbs. The LED on the Low-Power ON/OFF Server board uses a GPIO to control the LED; therefore, it cannot control the level setting whereas the LED on the Dimmable Light Bulbs are controlled by a PWM, providing the ability to control the level setting.

Figure 47. Sending a Set Message to All Nodes in Group "Building 6.2"



11 Summary

This application note introduces you to Cypress’ Bluetooth SIG-compliant Mesh solution. It provides an overview of Bluetooth Mesh and its key concepts as well as how to get started with Cypress’ BLE Mesh solution for your applications. This application note provides details on various software tools, including mobile applications, as well as the CYBT-213043-MESH evaluation kit offered by Cypress as part of its Bluetooth Mesh offering. Cypress offers many code examples to help evaluate and design Bluetooth Mesh applications. This application note covers Cypress’ BLE Mesh architecture, code structure, and how make changes based on your application needs.

12 Related Application Notes

- AN225684 – Getting Started with CYW20819
- AN225270 – CYW20819 Low-Power Guidelines
- AN226546 – CYW20819 Feature and Peripheral Guide

About the Authors

Name:	David Solda, Santhosh Vojjala, Sachin Gupta
Title:	Senior Business Unit Director, Principal Applications Engineer, Sr. Staff Product Marketing Engineer

Appendix A. Understanding the Flow of the Cypress Mesh Application Library

This appendix section takes you through the Mesh Application library code flow that includes initializing a Mesh GATT database, registering GATT callbacks, registering Mesh-specific callback functions with the Mesh Core, initializing the firmware upgrade library, setting the output TX power level, and other Mesh-related items which do not need hardware-specific knowledge. All initialization and actions which are specific to hardware must be implemented in the Mesh user application code as shown in the code examples.

The main blocks of the Mesh Application library implementation in the BT SDK in ModusToolbox are:

1. System Bluetooth Parameters
2. System initialization
3. Bluetooth stack event handlers
4. Bluetooth Mesh-specific initialization

The Mesh Application library takes care of the state machines and event handlers for most of the Bluetooth Mesh functionality. It acts as an interface between Mesh Core libraries and user application (code examples). Hooks are provided by the Mesh application library for the user application to register hardware-specific and implementation-specific callback functions, which are described in Section 7.2.3.

A.1 System Bluetooth Parameters

Before looking at the application initialization routines, let's look at *wiced_bt_cfg.c*. The configuration structures in this file allow simple control over advertisement/scan parameters for BLE and allocates buffer pools for the stack. The SDK provides APIs that allow the user to modify the settings based on the user's requirements. The default values provided in this file case be used as-is for Mesh implementations. Typically, the only parameter which needs modification for a Mesh application is the device name, because it is unique for each application. The device name parameter, `wiced_bt_cfg.settings.device_name`, can be modified in the user application code so it is not necessary to change it in `wiced_bt_cfg.c`.

For example, if a user is creating a Dimmable Light Mesh application, the device name can be modified in the user application code in the Mesh initialization function as follows:

```
wiced_bt_cfg_settings.device_name = (uint8_t *)"Dimmable Light";
```

All the parameters in this file are tuned for optimal performance of the Mesh device. So, it is recommended not to make any modification to this file.

Code 24. wiced_bt_cfg: Application Configuration Structure

```
/*****************************************************************************
 * wiced_bt core stack configuration
 ****************************************************************************/
wiced_bt_cfg_settings_t wiced_bt_cfg_settings =
{
    .device_name                        = (uint8_t *)"Mesh",                                  // Local device name (NULL terminated)
    .device_class                       = { 0x20, 0x07, 0x04 },                               // Local device class
    .security_requirement_mask          = BTM_SEC_NONE,                                       // Security requirements mask (BTM_SEC_NONE, or
                                     combinination of BTM_SEC_IN_AUTHENTICATE, BTM_SEC_OUT_AUTHENTICATE, BTM_SEC_ENCRYPT (see #wiced_bt_sec_level_e))
    .max_simultaneous_links             = 3,                                                  // Maximum number simultaneous links to different
devices

    .br_edr_scan_cfg =                                                                        // BR/EDR scan settings
    {
        .inquiry_scan_type              = BTM_SCAN_TYPE_STANDARD,                             // Inquiry scan type (BTM_SCAN_TYPE_STANDARD or
BTM_SCAN_TYPE_INTERLACED)
        .inquiry_scan_interval          = WICED_BT_CFG_DEFAULT_INQUIRY_SCAN_INTERVAL,         // Inquiry scan interval  (0 to use default)
        .inquiry_scan_window            = WICED_BT_CFG_DEFAULT_INQUIRY_SCAN_WINDOW,           // Inquiry scan window (0 to use default)

        .page_scan_type                 = BTM_SCAN_TYPE_STANDARD,                             // Page scan type (BTM_SCAN_TYPE_STANDARD or
BTM_SCAN_TYPE_INTERLACED)
        .page_scan_interval             = WICED_BT_CFG_DEFAULT_PAGE_SCAN_INTERVAL,            // Page scan interval  (0 to use default)
        .page_scan_window               = WICED_BT_CFG_DEFAULT_PAGE_SCAN_WINDOW               // Page scan window (0 to use default)
    },

    .ble_scan_cfg =                                                                           // BLE scan settings
    {
        .scan_mode                      = BTM_BLE_SCAN_MODE_PASSIVE,                          // BLE scan mode (BTM_BLE_SCAN_MODE_PASSIVE,
BTM_BLE_SCAN_MODE_ACTIVE, or BTM_BLE_SCAN_MODE_NONE)

        .high_duty_scan_interval        = 96,                 // High duty scan interval
        .high_duty_scan_window          = 96,                   // High duty scan window
        .high_duty_scan_duration        = 0,                                                  // High duty scan duration in seconds (0 for infinite)

        .low_duty_scan_interval         = 96,                                                 // Low duty scan interval
        .low_duty_scan_window           = 96,                                                 // Low duty scan window
        .low_duty_scan_duration         = 0,                                                  // Low duty scan duration in seconds (0 for infinite)
        /* Connection scan intervals */
        .high_duty_conn_scan_interval    = WICED_BT_CFG_DEFAULT_HIGH_DUTY_CONN_SCAN_INTERVAL,        // High duty cycle connection scan interval
        .high_duty_conn_scan_window      = WICED_BT_CFG_DEFAULT_HIGH_DUTY_CONN_SCAN_WINDOW,          // High duty cycle connection scan window
        .high_duty_conn_duration         = 30,                                                       // High duty cycle connection duration in seconds (0 for
infinite)

        .low_duty_conn_scan_interval     = WICED_BT_CFG_DEFAULT_LOW_DUTY_CONN_SCAN_INTERVAL,         // Low duty cycle connection scan interval
        .low_duty_conn_scan_window       = WICED_BT_CFG_DEFAULT_LOW_DUTY_CONN_SCAN_WINDOW,           // Low duty cycle connection scan window
        .low_duty_conn_duration          = 30,                                                       // Low duty cycle connection duration in seconds (0 for
infinite)

        /* Connection configuration */
        .conn_min_interval               = WICED_BT_CFG_DEFAULT_CONN_MIN_INTERVAL,                    // Minimum connection interval, 112 * 1.25 = 140ms.
        .conn_max_interval               = WICED_BT_CFG_DEFAULT_CONN_MAX_INTERVAL,                    // Maximum connection interval, 128 * 1.25 = 160ms.
        .conn_latency                    = WICED_BT_CFG_DEFAULT_CONN_LATENCY,                         // Connection latency, ~1sec
        .conn_supervision_timeout        = WICED_BT_CFG_DEFAULT_CONN_SUPERVISION_TIMEOUT              // Connection link supervsion timeout
    },

    /* BLE advertisement settings */
    {
        .channel_map                     = BTM_BLE_ADVERT_CHNL_37 |                                  // Advertising channel map (mask of BTM_BLE_ADVERT_CHNL_37,
BTM_BLE_ADVERT_CHNL_38, BTM_BLE_ADVERT_CHNL_39)
                                           BTM_BLE_ADVERT_CHNL_38 |
                                           BTM_BLE_ADVERT_CHNL_39,

        .high_duty_min_interval          = WICED_BT_CFG_DEFAULT_HIGH_DUTY_ADV_MIN_INTERVAL,          // High duty undirected connectable minimum advertising
interval 48 *0.625 = 30ms
        .high_duty_max_interval          = WICED_BT_CFG_DEFAULT_HIGH_DUTY_ADV_MAX_INTERVAL,          // High duty undirected connectable maximum advertising
interval
        .high_duty_duration              = 30,                                                       // High duty undirected connectable advertising duration in
seconds (0 for infinite)

        .low_duty_min_interval           = 1024,                                                     // Low duty undirected connectable minimum advertising
interval. 2048 *0.625 = 1.28s
        .low_duty_max_interval           = 1024,                                                     // Low duty undirected connectable maximum advertising
interval
        .low_duty_duration               = 60,                                                       // Low duty undirected connectable advertising duration in
seconds (0 for infinite)

        .high_duty_directed_min_interval = WICED_BT_CFG_DEFAULT_HIGH_DUTY_DIRECTED_ADV_MIN_INTERVAL, // High duty directed connectable minimum advertising interval
        .high_duty_directed_max_interval = WICED_BT_CFG_DEFAULT_HIGH_DUTY_DIRECTED_ADV_MAX_INTERVAL, // High duty directed connectable maximum advertising interval

        .low_duty_directed_min_interval  = WICED_BT_CFG_DEFAULT_LOW_DUTY_DIRECTED_ADV_MIN_INTERVAL,  // Low duty directed connectable minimum advertising interval
        .low_duty_directed_max_interval  = WICED_BT_CFG_DEFAULT_LOW_DUTY_DIRECTED_ADV_MAX_INTERVAL,  // Low duty directed connectable maximum advertising interval
        .low_duty_directed_duration      = 30,                                                       // Low duty directed connectable advertising duration in
seconds (0 for infinite)

        .high_duty_nonconn_min_interval  = WICED_BT_CFG_DEFAULT_HIGH_DUTY_NONCONN_ADV_MIN_INTERVAL,  // High duty non-connectable minimum advertising interval
        .high_duty_nonconn_max_interval  = WICED_BT_CFG_DEFAULT_HIGH_DUTY_NONCONN_ADV_MAX_INTERVAL,  // High duty non-connectable maximum advertising interval
        .high_duty_nonconn_duration      = 30,                                                       // High duty non-connectable advertising duration in seconds
(0 for infinite)

        .low_duty_nonconn_min_interval   = WICED_BT_CFG_DEFAULT_LOW_DUTY_NONCONN_ADV_MIN_INTERVAL,   // Low duty non-connectable minimum advertising interval
        .low_duty_nonconn_max_interval   = WICED_BT_CFG_DEFAULT_LOW_DUTY_NONCONN_ADV_MAX_INTERVAL,   // Low duty non-connectable maximum advertising interval
        .low_duty_nonconn_duration       = 0                                                         // Low duty non-connectable advertising duration in seconds (0
for infinite)
    },

    .gatt_cfg =                                                                                      // GATT configuration
    {
        .appearance                    = APPEARANCE_GENERIC_TAG,                                     // GATT appearance (see gatt_appearance_e)
        .client_max_links              = 1,                       // Client config: maximum number of servers that local client can connect to
        .server_max_links              = 1,                       // Server config: maximum number of remote clients connections allowed by the local
        .max_attr_len                  = 357,                     // Maximum attribute length; gki_cfg must have a corresponding buffer pool that can
                                                                    hold this length
#if ( defined(CYW20719B0) || defined(CYW20719B1) || defined(CYW20721B1) || defined(CYW43012C0) || defined(CYW20735B1)  || defined(CYW20819A1) )
        .max_mtu_size                  = 360                    // Maximum MTU size for GATT connections, should be between 23 and (max_attr_len + 5)
#endif
},

    .rfcomm_cfg =                                                                                    // RFCOMM configuration
    {
        .max_links                     = 0,                                         // Maximum number of simultaneous connected remote devices*/
        .max_ports                     = 0                                                           // Maximum number of simultaneous RFCOMM ports
    },

    .l2cap_application =                                                            // Application managed l2cap protocol configuration
    {
        .max_links                     = 0,                                         // Maximum number of application-managed l2cap links (BR/EDR and LE)

        /* BR EDR l2cap configuration */
        .max_psm                       = 0,                                                      // Maximum number of application-managed BR/EDR PSMs
        .max_channels                  = 0,                                                      // Maximum number of application-managed BR/EDR channels

        /* LE L2cap connection-oriented channels configuration */
        .max_le_psm                    = 0,                                                         // Maximum number of application-managed LE PSMs
        .max_le_channels               = 0,                                                         // Maximum number of application-managed LE channels
#if ( defined(CYW20719B1) || defined(CYW20721B1) || defined(CYW43012C0) || defined(CYW20735B1) || defined (CYW20819A1) )
        /* LE L2cap fixed channel configuration */
        .max_le_l2cap_fixed_channels     = 0                                                           // Maximum number of application managed fixed channels
supported (in addition to mandatory channels 4, 5 and 6).
#endif
    },

    .avdt_cfg =                                                                                      // Audio/Video Distribution configuration
    {
        .max_links                     = 0,                                                          // Maximum simultaneous audio/video links
    },

    . avrc_cfg =                                                                                     // Audio/Video Remote Control configuration
    {
        .roles                         = 0,                                                          // Mask of local roles supported
(AVRC_CONN_INITIATOR|AVRC_CONN_ACCEPTOR)
        .max_links                     = 0                                                           // Maximum simultaneous remote control links
    },
    .addr_resolution_db_size             = 5,                                                        // LE Address Resolution DB settings - effective only for pre
4.2 controller
#if ( defined(CYW20719B0) || defined(CYW20719B1) || defined(CYW20721B1) || defined(CYW43012C0) || defined(CYW20735B1) || defined(CYW20819A1) )
```

```
    .max_number_of_buffer_pools         = 4,                                                     // Maximum number of buffer pools in p_btm_cfg_buf_pools and
by wiced_create_pool
    .rpa_refresh_timeout                = 0,                                                     // Interval of  random address refreshing - secs
#if ( defined(CYW43012C0) || defined(CYW20735B1) || defined(CYW20819A1) )
    .ble_white_list_size                = 0                                   // Maximum number of white list devices allowed. Cannot be more than 128
#endif
#else
    .max_mtu_size                       = 360                     // Maximum MTU size for GATT connections, should be between 23 and (max_attr_len + 5)
#endif
};

/*****************************************************************************
 * wiced_bt  buffer pool configuration
 *
 * Configure buffer pools used by the stack  according to application's requirement
 *
 * Pools must be ordered in increasing buf_size.
 * If a pool runs out of buffers, the next  pool will be used
 *****************************************************************************/
const wiced_bt_cfg_buf_pool_t wiced_bt_cfg_buf_pools[WICED_BT_CFG_NUM_BUF_POOLS] =
{
/*  { buf_size, buf_count } */
    { 64,   42 },      /* Small Buffer Pool */
    { 140, 20 },       /* Medium Buffer Pool (used for HCI & RFCOMM control messages, min recommended size is 360) */
    { 360, 12 },       /* Large Buffer Pool  (used for HCI ACL messages) */
    { 1024, 3 },       /* Extra Large Buffer Pool - Used for avdt media packets and miscellaneous (if not needed, set buf_count to 0) */
};
```

A.2 System Initialization

A.2.1 APPLICATION_START() Function

The first visible entry point in to the Mesh application is the `APPLICATION_START()` function shown in Code 30. When the Mesh device starts up, it enters the `APPLICATION_START()` function located in *mesh_application.c*. This file is located at *CODE_EXAMPLE_NAME _mainapp/libraries/mesh_app_lib*.

The `APPLICATION_START()` function performs the following critical tasks:

- Initializes the UART transport and start a 1-second application timer –Code 25 and Code 26
- Initializes the hardware configuration –Code 27
- Initializes the Bluetooth Stack – using the below API shown in Code 30

  `wiced_bt_stack_init(mesh_management_cback, &wiced_bt_cfg_settings, wiced_bt_cfg_buf_pools)`
- Sets up NVRAM IDs which are used by Mesh Core and Model libraries with function call `mesh_setup_nvram_ids()` – see Code 29.

The user application code defines callback functions in the `wiced_bt_mesh_app_func_table` structure if it is required to perform hardware-specific initialization and develop custom application logic. If a hardware initialization function is defined, the user application code takes care of the hardware initialization; otherwise, the default initialization is completed based on the hardware/schematic of the Cypress evaluation kit. Code 30 shows the initialization performed in `APPLICATION_START()`. The following sections provide more detailed descriptions of each process in this function.

1. The transport method and related parameters are initialized in the `mesh_app_hci_init()` function. In the case of Mesh code examples, the default configuration is PUART with a baud rate of 921600 bps. The application debug information is sent over this interface. The baud rate can be configured using the `wiced_hal_puart_set_baudrate()` function as shown in Code 25. The `mesh_app_hci_init` function is defined in the *mesh_app_hci.c* file that is located under the following location:

CODE_EXAMPLE_NAME\bt_20819A1-1.0\components\BT-SDK\common\libraries\mesh_app_lib\ mesh_app_hci.c

Code 25. *mesh_app_hci.c*: Initializing Transport for Printing Debug Information

```
void mesh_app_hci_init(void)
{
#ifndef MESH_HOMEKIT_COMBO_APP
    wiced_transport_init(&transport_cfg);

    // create special pool for sending data to the MCU
    host_trans_pool = wiced_transport_create_buffer_pool(1024, 2);
#endif

#ifdef WICED_BT_TRACE_ENABLE
#ifndef _DEB_ENABLE_HCI_TRACE
    //For the 24 MHz board set the UART type as WICED_ROUTE_DEBUG_TO_PUART
    // For BCM920706V2_EVAL board make sure SW5.2 and SW5.4 are ON and all other SW5 are off
    wiced_set_debug_uart(WICED_ROUTE_DEBUG_TO_PUART);
#if (defined(CYW20706A2) || defined(CYW20735B0) || defined(CYW20719B0) || defined(CYW43012C0))
    wiced_hal_puart_select_uart_pads(WICED_PUART_RXD, WICED_PUART_TXD, 0, 0);
#endif
#ifndef CYW20706A2
```

```
    wiced_hal_puart_set_baudrate(921600);
#endif
#else
    // WICED_ROUTE_DEBUG_TO_WICED_UART to send debug strings over the WICED debug interface */
    wiced_set_debug_uart(WICED_ROUTE_DEBUG_TO_WICED_UART);
#endif

    /* Starting the app timer  */
    memset(&app_timer, 0, sizeof(wiced_timer_t));
    if (wiced_init_timer(&app_timer, mesh_app_timer, 0, WICED_SECONDS_PERIODIC_TIMER) == WICED_SUCCESS)
    {
        if (wiced_start_timer(&app_timer, MESH_APP_TIMEOUT_IN_SECONDS) != WICED_SUCCESS)
        {
            WICED_BT_TRACE("APP START Timer FAILED!!\n");
        }
    }
    else
    {
        WICED_BT_TRACE("APP INIT Timer FAILED!!\n");
    }
#endif
    wiced_bt_dev_register_hci_trace(mesh_hci_trace_cback);
}
```

A periodic one-second timer is initialized and started in the `mesh_app_hci_init()` function. Existing code in the timer callback function, `mesh_app_timer`, has a provision to transmit the buffer usage data and Mesh status data every 30 seconds over the PUART interface.

Code 26. *mesh_app_timer.c:* Timer Callback Function

```
void mesh_app_timer(uint32_t arg)
{
    static uint32_t app_timer_count = 1;
    app_timer_count++;

#ifdef _DEB_PRINT_BUF_USE
    /* dump wiced bt buffer statistics on every 10 seconds to monitor buffer usage */
    if (!(app_timer_count % _DEB_PRINT_BUF_USE))
    {
        _deb_print_buf_use();
    }
#endif
#ifdef _DEB_PRINT_MESH_STATS
    // dump mesh statistics on every _DEB_PRINT_MESH_STATS
    if ((app_timer_count % _DEB_PRINT_MESH_STATS) == 2)
    {
        _deb_print_mesh_stats();
    }
#endif
}
```

2. The Mesh Application library provides hooks for the Mesh user application code to register callback functions. Callback functions applicable/needed by the Mesh user application code should be populated in the `wiced_bt_mesh_app_func_table` structure so that the Mesh Application library knows that the user application code wants to perform custom hardware configuration instead of the default configuration in the Mesh Application library as shown in Code 27.

   For example, if the application project requires the use of a specific button, an entry should be added in `wiced_bt_mesh_app_func_table` as shown in Code 28. By doing this, the Mesh Application library knows that the application project is required to customize the implementation based on user requirements; it will not rely on the default button configuration in the Mesh Application library.

Code 27. *mesh_application.c:* Hardware Initialization Function

```
    if (wiced_bt_mesh_app_func_table.p_mesh_app_hw_init != NULL)
    {
        wiced_bt_mesh_app_func_table.p_mesh_app_hw_init();
    }
    else
    {
        /* Configure buttons available on the platform */
#if defined(CYW20706A2)
        wiced_hal_gpio_configure_pin(WICED_GPIO_BUTTON, WICED_GPIO_BUTTON_SETTINGS(GPIO_EN_INT_BOTH_EDGE), WICED_GPIO_BUTTON_DEFAULT_STATE);
        wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_BUTTON, mesh_interrupt_handler, NULL);
#elif (defined(CYW20735B0) || defined(CYW20719B0) || defined(CYW20721B0))
        wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_PIN_BUTTON, mesh_interrupt_handler, NULL);
        wiced_hal_gpio_configure_pin(WICED_GPIO_PIN_BUTTON, WICED_GPIO_BUTTON_SETTINGS, GPIO_PIN_OUTPUT_LOW);
#else
        wiced_platform_register_button_callback(WICED_PLATFORM_BUTTON_1, mesh_interrupt_handler, NULL, GPIO_EN_INT_BOTH_EDGE);
#endif
    }
```

Code 28. `Dimmer_mainapp`: Custom Hardware Initialization Function button_hardware_init

```
/*
 * Mesh application library will call into application functions if provided by the application.
 */
wiced_bt_mesh_app_func_table_t wiced_bt_mesh_app_func_table =
{
    mesh_app_init,          // application initialization
    button_hardware_init,   // hardware initialization
    NULL,                   // GATT connection status
    NULL,                   // attention processing
    NULL,                   // notify period set
    NULL                    // WICED HCI command
};
```

3. The Bluetooth stack is initialized by calling the `wiced_bt_stack_init` function.

   As part of the `wiced_bt_stack_init` function, Bluetooth configuration settings, `wiced_bt_cfg_settings`, are passed to the stack and the `mesh_management_cback` function is registered to receive BLE stack events. Note that `mesh_management_cback` function already exists and does not need to be added to your source file. The `wiced_bt_stack_init` function is declared in the *wiced_bt_stack.h* standard SDK library that is in the *include* folder of the project. The actual implementation of this function is in ROM.

4. Maximum number of network keys, application keys, scenes and scheduler events are configured to 4, 8, 10 and 16 respectively. The maximum values for these parameters depend on the available free RAM for a given application.

   - **Network key:** A NetKey secures the communication at the Network Layer and is shared across all nodes in the network or all nodes in a particular Subnet. Possession of the NetKey allows a node to decrypt and authenticate up to the network layer, allowing the relay of messages, but it does not allow application data decryption. The default value of network keys is set to ‘4’.
   - **Application Key:** An AppKey secures the communication at the Access Layer and is shared across all nodes that participate in a given Mesh application. A Provisioner is responsible for generating and distributing AppKeys. For example, a lighting system AppKey would be shared between light switches and light bulbs, but not with a thermostat or a motion sensor. The default value of application keys is set to ‘8’.
   - **Scenes:** Scenes serve as memory banks for storage of states (e.g., a power level or a light level/color). Scenes allow triggering of an action that can set multiple states on different Mesh nodes with a single Scene recall message. They can be triggered on demand or at a specified time. For example, a scene may be configured to set the temperature of a room to 60 degrees Fahrenheit and set the living room lights to a 50% level setting. The default value of scenes is set to ‘10’.
   - **Scheduler:** A Scheduler provides a means for an autonomous change of states of a device based on the time, calendar, and a register of defined time points with associated state changing actions. For example, a lamp may automatically turn OFF every day at 2 AM. The default value of scheduler events is set to ‘16’.
   - **NVRAM Allocation:** Memory is allotted in the VS (Vendor-Specific) section of the NVRAM (flash memory) for Mesh parameters that need to survive power cycles. This is accomplished by the `mesh_setup_nvram_ids()` function.

Code 29. *mesh_application.c:* Setup NVRAM IDs for Mesh Parameters

```
void mesh_setup_nvram_ids()
{
    uint16_t cfg_data_len = wiced_bt_mesh_get_node_config_size(&mesh_config);

    wiced_bt_mesh_light_lc_nvram_id_start           = WICED_NVRAM_VSID_END   - number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_LIGHT_LC_SRV);
    wiced_bt_mesh_light_hsl_nvram_id_start          = wiced_bt_mesh_light_lc_nvram_id_start -
                                                           number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_LIGHT_HSL_SRV);
    wiced_bt_mesh_light_ctl_nvram_id_start          = wiced_bt_mesh_light_hsl_nvram_id_start        -
                                                           number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_LIGHT_CTL_SRV);
    wiced_bt_mesh_light_xyl_nvram_id_start          = wiced_bt_mesh_light_ctl_nvram_id_start        -
                                                           number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_LIGHT_XYL_SRV);
    wiced_bt_mesh_light_lightness_nvram_id_start    = wiced_bt_mesh_light_xyl_nvram_id_start        -
                                                            number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_LIGHT_LIGHTNESS_SRV);
    wiced_bt_mesh_power_level_nvram_id_start        = wiced_bt_mesh_light_lightness_nvram_id_start  -
                                                          number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_GENERIC_POWER_LEVEL_SRV);
    wiced_bt_mesh_power_onoff_nvram_id_start        = wiced_bt_mesh_power_level_nvram_id_start      -
                                                          number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_GENERIC_POWER_ONOFF_SRV);
    wiced_bt_mesh_default_trans_time_nvram_id_start = wiced_bt_mesh_power_onoff_nvram_id_start      - 1;
    wiced_bt_mesh_scheduler_nvram_id_start          = wiced_bt_mesh_default_trans_time_nvram_id_start - wiced_bt_mesh_scheduler_events_max_num;
    wiced_bt_mesh_scene_nvram_id_end                = wiced_bt_mesh_scheduler_nvram_id_start        - 1;
    wiced_bt_mesh_scene_nvram_id_start              = wiced_bt_mesh_scene_nvram_id_end              - wiced_bt_mesh_scene_max_num;
    wiced_bt_mesh_scene_register_nvram_id           = wiced_bt_mesh_scene_nvram_id_start            - 1;

    wiced_bt_mesh_core_nvm_idx_node_data            = wiced_bt_mesh_scene_register_nvram_id     - 1;
    wiced_bt_mesh_core_nvm_idx_virt_addr            = wiced_bt_mesh_core_nvm_idx_node_data      - 1;
    wiced_bt_mesh_core_nvm_idx_frnd_state           = wiced_bt_mesh_core_nvm_idx_virt_addr      - 1;
    wiced_bt_mesh_core_nvm_idx_net_key_begin        = wiced_bt_mesh_core_nvm_idx_frnd_state     - wiced_bt_mesh_core_net_key_max_num;
    wiced_bt_mesh_core_nvm_idx_app_key_begin        = wiced_bt_mesh_core_nvm_idx_net_key_begin  - wiced_bt_mesh_core_app_key_max_num;
    wiced_bt_mesh_core_nvm_idx_health_state         = wiced_bt_mesh_core_nvm_idx_app_key_begin  - 1;
    wiced_bt_mesh_core_nvm_idx_cfg_data             = wiced_bt_mesh_core_nvm_idx_health_state   - ((cfg_data_len + 0xfe) / 0xff);
    mesh_nvm_idx_seq                                = wiced_bt_mesh_core_nvm_idx_cfg_data       - 1;

    WICED_BT_TRACE("setup nvram ids: net_key_max_num:%d app_key_max_num:%d nvm_idx_seq:%x %x-%x\n", wiced_bt_mesh_core_net_key_max_num,
wiced_bt_mesh_core_app_key_max_num, mesh_nvm_idx_seq, wiced_bt_mesh_core_nvm_idx_cfg_data, WICED_NVRAM_VSID_END);
}
```

Code 30. APPLICATION_START() Function

```
/*
*  Entry point to the application. Set device configuration and start BT
*  stack initialization.  The actual application initialization will happen
*  when stack reports that BT device is ready.
*/
#ifndef MESH_HOMEKIT_COMBO_APP
#if (defined(CYW20719B0) || defined(CYW20719B1) || defined(CYW20721B1) ||  defined(CYW20706A2))
APPLICATION_START()
#else
void application_start(void)
#endif
#else // MESH_HOMEKIT_COMBO_APP
void mesh_application_start()
#endif
{
    mesh_app_hci_init();

    // If application wants to control the hardware, call appropriate initialization function.
    // Otherwise use default processing of the mesh application library.
    if (wiced_bt_mesh_app_func_table.p_mesh_app_hw_init != NULL)
    {
        wiced_bt_mesh_app_func_table.p_mesh_app_hw_init();
    }
    else
    {
        /* Configure buttons available on the platform */
#if defined(CYW20706A2)
        wiced_hal_gpio_configure_pin(WICED_GPIO_BUTTON, WICED_GPIO_BUTTON_SETTINGS(GPIO_EN_INT_BOTH_EDGE), WICED_GPIO_BUTTON_DEFAULT_STATE);
        wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_BUTTON, mesh_interrupt_handler, NULL);
#elif (defined(CYW20735B0) || defined(CYW20719B0) || defined(CYW20721B0))
        wiced_hal_gpio_register_pin_for_interrupt(WICED_GPIO_PIN_BUTTON, mesh_interrupt_handler, NULL);
        wiced_hal_gpio_configure_pin(WICED_GPIO_PIN_BUTTON, WICED_GPIO_BUTTON_SETTINGS, GPIO_PIN_OUTPUT_LOW);
#else
        wiced_platform_register_button_callback(WICED_PLATFORM_BUTTON_1, mesh_interrupt_handler, NULL, GPIO_EN_INT_BOTH_EDGE);
#endif
    }
#ifndef MESH_HOMEKIT_COMBO_APP
    // Register call back and configuration with stack
    wiced_bt_stack_init(mesh_management_cback, &wiced_bt_cfg_settings, wiced_bt_cfg_buf_pools);
#endif

    // Currently we can support up to 4 network keys.
    wiced_bt_mesh_core_net_key_max_num = 4;
    wiced_bt_mesh_core_app_key_max_num = 8;
    wiced_bt_mesh_scene_max_num            = 10;
    wiced_bt_mesh_scheduler_events_max_num = 16; // PTS test uses index 15 (MMDL/SR/SCHS/BV-01-C )

    // setup NVRAM IDs which will be used by core and models
    mesh_setup_nvram_ids();

    WICED_BT_TRACE("Mesh Start\n");
}
```

A.3 Bluetooth Stack Event Handler

The Bluetooth event handler function, `mesh_management_cback`, is registered during initialization to receive Bluetooth stack-related callback events**.** Two Bluetooth events are of interest for a Mesh application:

1. The `BTM_ENABLED_EVT` event is triggered after Bluetooth stack initialization is successful. The Bluetooth Mesh library initialization and necessary callback function registration are done in this event.
2. The `BTM_BLE_ADVERT_STATE_CHANGED_EVT` event is received after an advertisement timeout period or after a successful connection. In this event, the Mesh Core is requested to restart advertisements if the Mesh device is in a disconnected state.

Code 31. *mesh_application.c:* Bluetooth Stack Callback Events

```
wiced_result_t mesh_management_cback(wiced_bt_management_evt_t event, wiced_bt_management_evt_data_t *p_event_data)
{
    wiced_bt_ble_advert_mode_t          *p_mode;
    wiced_result_t                      result = WICED_BT_SUCCESS;

    // WICED_BT_TRACE("mesh_management_cback: %x\n", event);
    //test_aes();

    switch (event)
    {
        /* Bluetooth  stack enabled */
    case BTM_ENABLED_EVT:
#ifdef _DEB_DELAY_START_SEC
        mesh_delay_start_init();
#else
#ifdef MESH_APPLICATION_MCU_MEMORY
        mesh_application_send_hci_event(HCI_CONTROL_EVENT_DEVICE_STARTED, NULL, 0);
#else
        mesh_application_init();
#endif
#endif
        break;

    case BTM_DISABLED_EVT:
        break;

    case BTM_BLE_ADVERT_STATE_CHANGED_EVT:
        p_mode = &p_event_data->ble_advert_state_changed;
        WICED_BT_TRACE("Advertisement State Changed:%d\n", *p_mode);
        if (*p_mode == BTM_BLE_ADVERT_OFF)
        {
            WICED_BT_TRACE("adv stopped\n");
            // On failed attempt to connect FW stops all connectable adverts. 20719B1 also receives that event in case of successfull connection
            // If we disconnected then notify core to restart them
            if (!mesh_app_gatt_is_connected())
            {
                wiced_bt_mesh_core_connection_status(0, WICED_FALSE, 0, 20);
            }
        }
        break;

    case BTM_BLE_SCAN_STATE_CHANGED_EVT:
        WICED_BT_TRACE("Scan State Change:%d\n", p_event_data->ble_scan_state_changed);
        break;

    case  BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT:
        result = WICED_BT_ERROR;
        break;

    default:
        break;
    }

    return result;
}
```

A.4 Mesh-Specific Initialization

The `mesh_application_init()` function is executed in the `BTM_ENABLED_EVT` event of the Bluetooth stack callback.

All Mesh-specific initialization occurs in this function. Detailed steps in this function are provided below and can be seen in Code 34.

1. Register a callback function, `mesh_gatts_callback` in `mesh_app_init()`, to receive Bluetooth GATT events.

Code 32. Bluetooth GATT Callback Registration

```
void mesh_app_gatt_init(void)
{
    wiced_bt_gatt_status_t gatt_status;
#ifndef MESH_HOMEKIT_COMBO_APP
    /* Register with stack to receive GATT callback */
    gatt_status = wiced_bt_gatt_register(mesh_gatts_callback);
    WICED_BT_TRACE("wiced_bt_gatt_register: %d\n", gatt_status);
    memset(&mesh_gatt_cb, 0, sizeof(mesh_gatt_cb_t));
#endif // MESH_HOMEKIT_COMBO_APP
}
```

2. Disable the standard Bluetooth pairing capability by using `wiced_bt_set_pairable_mode(WICED_FALSE, WICED_FALSE)`.
3. Check whether the Mesh device has Configuration Client Model and set a flag, `mesh_config_client`, accordingly. The Configuration Client model is used only in the Provisioner code example. This flag is set to 'false' for all other use cases.

   A Mesh device needs two random static Bluetooth device addresses; one is used during non-provisioned state and the other is used after the device is provisioned into a network. A Mesh device also needs a 128-bit (16-byte) UUID for virtual addresses. The generation of the Bluetooth device addresses and UUID values for the Mesh device occur when the device boots up for the very first time. These addresses and UUID are stored in the NVRAM. These addresses are retrieved from the NVRAM during subsequent power cycles and stored in local variables.

Code 33. Generate BD Address for Provisioned and Non-Provisioned State and UUID

```
if (mesh_nvram_access(WICED_FALSE, NVRAM_ID_LOCAL_ADDR, buffer, (2 * 6) + 16, &result) != ((2 * 6) + 16))
 {
     mesh_application_gen_bda(init.non_provisioned_bda, init.provisioned_bda, init.device_uuid, WICED_TRUE);
 }
 else
 {
     memcpy(init.non_provisioned_bda, &buffer[0], 6);
     memcpy(init.provisioned_bda, &buffer[6], 6);
     memcpy(init.device_uuid, &buffer[6 * 2], 16);
 }
```

5. Initialize the Mesh Core with the necessary parameters/structures/callback functions.

   Most of the necessary functions are implemented in *mesh_application.c* with a few exceptions with respect to configuration structures and user-specific functions. The Mesh Core initialization is described in more detail in Section A.4.1 below.
6. Initialize the GATT database based on the status of the device (provisioned or non-provisioned) using the function `mesh_app_gatt_db_init(node_authenticated)`. The GATT database of a Mesh device consists of multiple services like provisioning service, OTA service, proxy service, Device information service etc. The difference between provisioned and non-provisioned GATT data base is:
   - The provisioning service is enabled in the GATT database when the device is unprovisioned (`gatt_db_unprovisioned[]`).
   - The proxy service (if the node supports proxy feature) and mesh command service are enabled in the GATT database when the device is provisioned (`gatt_db_provisioned[]`).
7. Initialize the sequence number and RPL (Replay Protection List) list with `mesh_application_seq_init`. The sequence number and RPL are initialized to '0' and stored in NVRAM if the device is not provisioned; else the sequence number is updated and stored in NVRAM.
8. Initialize the firmware update library using the `wiced_ota_fw_upgrade_init` function.
9. Perform the Mesh application initialization defined in the application function, `wiced_bt_mesh_app_func_table.p_mesh_app_init(node_authenticated)`.

   Each code example (user application code) implements its own `wiced_bt_mesh_app_func_table`. Based on the configurations contained in `wiced_bt_mesh_app_func_table`, the appropriate functions are initialized and executed.
10. Call the Mesh Start function, `wiced_bt_mesh_core_start()`, after all of the above initializations are performed successfully.

Code 34. Mesh Application Initialization Function

```
void mesh_application_init(void)
{
    wiced_result_t              result;
    uint8_t                     buffer[(6 * 2) + 16];
    wiced_bt_mesh_core_init_t init = { 0 };

    WICED_BT_TRACE("## mesh_application_init free_bytes:%d ##\n", wiced_memory_get_free_bytes());

#ifndef MESH_HOMEKIT_COMBO_APP
    /* Initialize wiced app */
#if (!defined(CYW20735B1) && !defined(CYW20819A1))
    wiced_bt_app_init();
#endif
#endif

#ifndef MESH_HOMEKIT_COMBO_APP
    mesh_app_gatt_init();
    wiced_bt_set_pairable_mode(WICED_FALSE, WICED_FALSE);
#endif // MESH_HOMEKIT_COMBO_APP

    //remember if we are provisioner (config client)
    mesh_config_client = number_of_elements_with_model(WICED_BT_MESH_CORE_MODEL_ID_CONFIG_CLNT) > 0 ? WICED_TRUE : WICED_FALSE;

#ifdef PTS
    // initialize core
    // each node shall be assigned a 128-bit UUID known as the Device UUID.
    // Device manufacturers shall follow the standard UUID format and generation
    // procedure to ensure the uniqueness of each Device UUID.
    // for now device_uuid is local bda with hardcoded(0x0f) remaining bytes
    wiced_bt_dev_read_local_addr(init.non_provisioned_bda);
#ifdef _DEB_DF_TEST_BDA
    // remember local BD address for DF test mode
    memcpy(_deb_df_test_bda, init.non_provisioned_bda, 6);
#endif
    memcpy(init.provisioned_bda, init.non_provisioned_bda, sizeof(wiced_bt_device_address_t));
    memcpy(&init.device_uuid[0], init.non_provisioned_bda, 6);
    memset(&init.device_uuid[6], 0x0f, 16 - 6);
    // in PTS disable proxy on demand - always advertyse proxy service network if when appropriate
    wiced_bt_mesh_core_proxy_on_demand_advert_to = 0;
#else
    // We currently use 2 BDAs, one for the initial scenario. We also keep saved UUID
    if (mesh_nvram_access(WICED_FALSE, NVRAM_ID_LOCAL_ADDR, buffer, (2 * 6) + 16, &result) != ((2 * 6) + 16))
    {
        mesh_application_gen_bda(init.non_provisioned_bda, init.provisioned_bda, init.device_uuid, WICED_TRUE);
    }
    else
    {
        memcpy(init.non_provisioned_bda, &buffer[0], 6);
        memcpy(init.provisioned_bda, &buffer[6], 6);
        memcpy(init.device_uuid, &buffer[6 * 2], 16);
    }
#endif

    // Remove this line if MeshClient supports proxy on demand
    wiced_bt_mesh_core_proxy_on_demand_advert_to = 0;

#ifdef MESH_SUPPORT_PB_GATT
    mesh_config.features |= WICED_BT_MESH_CORE_FEATURE_BIT_PB_GATT;
#endif
    init.p_config_data = &mesh_config;
    init.callback = get_msg_handler_callback;
    init.pub_callback = mesh_publication_callback;
    init.proxy_send_callback = mesh_app_proxy_gatt_send_cb;
    init.nvram_access_callback = mesh_nvram_access;
    init.fault_test_cb = mesh_fault_test;
    init.attention_cb = wiced_bt_mesh_app_func_table.p_mesh_app_attention;
    init.state_changed_cb = mesh_state_changed_cb;
    init.scan_callback = mesh_start_stop_scan_callback;
    node_authenticated = WICED_BT_SUCCESS == wiced_bt_mesh_core_init(&init);

    mesh_app_gatt_db_init(node_authenticated);

    // Initialize own SEQ and RPL
    if (!mesh_application_seq_init())
        mesh_application_factory_reset();

    // Initialize OTA FW upgrade
    if (!wiced_ota_fw_upgrade_init(NULL, mesh_ota_firmware_upgrade_status_callback, mesh_ota_firmware_upgrade_send_data_callback))
    {
        WICED_BT_TRACE("mesh_application_init: wiced_ota_fw_upgrade_init failed\n");
    }

    wiced_bt_mesh_app_provision_server_init(pb_priv_key, NULL);

    mesh_app_gatt_init();

    if (wiced_bt_mesh_app_func_table.p_mesh_app_init)
    {
        wiced_bt_mesh_app_func_table.p_mesh_app_init(node_authenticated);
    }
    // Now start mesh picking up tx power set by app in wiced_bt_mesh_core_adv_tx_power
    wiced_bt_mesh_core_start();

    WICED_BT_TRACE("***** Free mem after app_init:%d\n", wiced_memory_get_free_bytes());
}
```

A.4.1 Mesh Core Initialization Structure

As part of the Mesh Core initialization process, the Mesh Application library provides the `wiced_bt_mesh_init_t` structure as a parameter to the API `wiced_bt_mesh_core_init()`. The parameters of the `wiced_bt_mesh_init_t` structure are shown in Code 35, the code snippet from the `mesh_applicaiton_init()` function that has details of the assigned callback functions and structures is shown in Code 36.

Table 9 provides details of the structures and callback functions that are required as part of the Mesh Core initialization process. These structures and callback functions are assigned to each field of the `wiced_bt_mesh_init_t` structure which is then passed as a parameter of for `wiced_bt_mesh_core_init()` to initialize the Mesh Core.

Any structure or callback function that is defined in the User Application Code (column 2 in Table 9) is the responsibility of the developer to define the appropriate configurations/actions to be completed. Any structure or callback function that is defined in the Mesh Application Library is already completed and is not recommended to change (these are provided in Table 9 for information and understanding of the Mesh Core Initialization process).

Code 35. Definition of wiced_bt_mesh_init_t Structure

```
typedef struct
{
    uint8_t device_uuid[16];                                   /**< 128-bit Device UUID. Device manufacturers shall follow the standard UUID format
                                                                    and generation procedure to ensure the uniqueness of each Device UUID */
    uint8_t non_provisioned_bda[6];                             /**< BD address in non-provisioned state */
    uint8_t provisioned_bda[6];                                 /**< BD address in provisioned state */
    wiced_bt_mesh_core_config_t *p_config_data;                 /**< Configuration data */
    wiced_bt_mesh_core_get_msg_handler_callback_t callback;     /**< Callback function to be called by the Core at received message to get message
                                                                     handler */
    wiced_bt_mesh_core_publication_callback_t pub_callback;     /**< Callback function to be called by the Core at time when publication is required by
                                                                     periodic publication or when configuration changes */
    wiced_bt_mesh_core_scan_callback_t scan_callback;           /**< Callback function to be called by the Core when scan needs to be started or
                                                                     stopped */
    wiced_bt_mesh_core_proxy_send_cb_t proxy_send_callback;     /**< Callback function to send proxy packet over GATT */
    wiced_bt_core_nvram_access_t nvram_access_callback;         /**< Callback function to read/write from/to NVRAM */
    wiced_bt_mesh_core_health_fault_test_cb_t fault_test_cb;    /**< Callback function to be called to invoke a self test procedure of an Element */
    wiced_bt_mesh_core_attention_cb_t attention_cb;             /**< Callback function to be called to attract human attention */
    wiced_bt_mesh_core_state_changed_callback_t state_changed_cb;   /**< Callback function to be called on any change in the mesh state */
} wiced_bt_mesh_core_init_t;
```

Code 36. Mesh Core Initialization Code Snippet from mesh_applicaiton_init() Function

```
    init.p_config_data = &mesh_config;
    init.callback = get_msg_handler_callback;
    init.pub_callback = mesh_publication_callback;
    init.proxy_send_callback = mesh_app_proxy_gatt_send_cb;
    init.nvram_access_callback = mesh_nvram_access;
    init.fault_test_cb = mesh_fault_test;
    init.attention_cb = wiced_bt_mesh_app_func_table.p_mesh_app_attention;
    init.state_changed_cb = mesh_state_changed_cb;
    init.scan_callback = mesh_start_stop_scan_callback;
    node_authenticated = WICED_BT_SUCCESS == wiced_bt_mesh_core_init(&init);
```

Table 9. Mesh Core Initialization Structure

Assigned Functions/Structures	Function/ Structure Location	Details
`wiced_bt_mesh_core_config_t   mesh_config`	User Application Code	The configuration structure is defined in the user application code (Code Examples). This structure contains the details such as feature capabilities (Friend, Relay, LPN etc.), definitions of Friend node parameters, definitions of Low Power Node parameters, elements, models, etc. Because these parameters are unique for each application, the Mesh Applications library allows the Mesh user application code to define its `mesh_core_config` structure. This structure is discussed in detail in Section 7.2.1
`get_msg_handler_callback`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). This function returns the message handler for the specific element requested by the Mesh Core library.
`mesh_publication_callback`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). The publication callback function (`mesh_publication_callback`) is registered with the Mesh Core for triggering periodic publication events. This function is also triggered when the publication period configuration is changed by the Provisioner, Mesh Application Library checks if `wiced_bt_mesh_app_func_table.p_mesh_app_notify_period_set` function is defined in the Mesh user application code (Code Examples) and triggers this callback function accordingly.
`mesh_app_proxy_gatt_send_cb`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_app_gatt.c* file). The Mesh Application GATT Proxy Callback function (`mesh_app_proxy_gatt_send_cb`) is registered with the Mesh Core. This callback function sends a packet over the GATT Proxy connection. Notifications are used when the Mesh node acts a Server and GATT write commands are used when the Mesh node acts as a Client.
`mesh_nvram_access`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). This function is used to read or write information from/to the NVRAM.

Assigned Functions/Structures	Function/ Structure Location	Details
`mesh_fault_test`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). It is used to initiate self-test procedure for an element.
`wiced_bt_mesh_app_func_table.p_mesh_app_attention`	User Application Code	The callback function is defined in the user application code (Code Examples). The Mesh Application library calls this registered callback function in the user application code when it receives the attention callback from Mesh Core libraries. For example, a custom logic to blink an LED every second (until the attention timeout expires) is implemented in the Dimmable Light Code example described in Step 4 of Section 8.2.2.
`mesh_state_changed_cb`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). The Mesh State Changed Callback function (`mesh_state_changed_cb`) is registered with the Mesh Core library. State changes in the Mesh device, sequence number, provisioning state, node reset status, LPN sleep, and friendship status are notified to the Mesh Application library by triggering specified events.
`mesh_start_stop_scan_callback`	Mesh Application Library	This function is defined in the Mesh Application library (*mesh_application.c* file). The Mesh Start Stop Callback function (`mesh_start_stop_scan_callback`) is registered with the Mesh Core library. Scanning for advertisement packets is either started or stopped in this function based on the received parameters.

Appendix B. Sensor Configuration

This appendix provides the `sensor` configuration parameters available for a Mesh enabled sensor device. Each Mesh device may contain one or more `sensors`.

The `wiced_bt_mesh_core_config_sensor_t` structure defines the details of a `sensor` configuration. Each parameter available in this structure is described below.

- `property_id`: The Sensor Property ID field is a 2-octet value referencing a device property that describes the meaning and the format of data reported by a sensor. The format of this property is Boolean as defined in Mesh Device Properties Specification.
- `prop_value_len`: This parameter represents the length of the value which is reported by this sensor.
- `descriptor` structure, `wiced_bt_mesh_sensor_config_descriptor_t,` describes the properties of the value reported by the sensor.
  - `.positive_tolerance`

    The Sensor Positive Tolerance parameter is a 12-bit value representing the magnitude of a possible positive error associated with the measurements that the sensor is reporting. For cases in which the tolerance information is not available, a value 0x00 is assigned to indicate that the positive tolerance field is unknown or not provided. Valid settings for this parameter are between 0x0001 to 0xFFF.
  - `.negative_tolerance`

    The Sensor Negative Tolerance parameter is a 12-bit value representing the magnitude of a possible negative error associated with the measurements that the sensor is reporting. For cases in which the tolerance information is not available, a value 0x00 is assigned to indicate that the negative tolerance field is unknown or not provided. Valid settings for this parameter are between 0x0001 to 0xFFF.
  - `.sampling_function`

    This Sensor Sampling Function parameter specifies the averaging operation or type of sampling function applied to the measured value. For example, this field can identify whether the measurement is an arithmetic mean value or an instantaneous value. For cases in which the sampling function is not available, a value 0x00 is assigned to indicate that the sampling function field is unknown or not provided. Valid settings for this parameter are show in below Table 10.

Table 10. `sampling_function` Parameter Values (Bluetooth SIG, 2019)

Value	Description
0x00	Unspecified
0x01	Instantaneous
0x02	Arithmetic Mean.
0x03	RMS
0x04	Maximum
0x05	Minimum
0x06	Accumulated. (See note below.)
0x07	Count. (See note below.)
0x08–0xFF	Reserved for Future Use

  - `.measurement_period`

    This Sensor Measurement Period parameter specifies a value, `n`, that represents the averaging time span, accumulation time, or measurement period in seconds over which the measurement is taken, using the formula shown below:

$$\text{Represented value} = 1.1^{n-64}$$

    For cases where the measurement period is not available, a value 0x00 is assigned to indicate that the sampling function field is unknown or not provided. Valid settings for this parameter are show in below Table 11.

Table 11. `measurement_period` Parameter Values (Bluetooth SIG, 2019)

Value n	Represented Value	Description
0x00	Not Applicable	Not Applicable
0x01–0xFF	1.1^{n-64}	Time period in seconds

- `.update_interval`

  The measurement reported by a sensor is internally refreshed at the frequency indicated in the Sensor Update Interval field (e.g., a temperature value that is internally updated every 15 minutes). This field specifies a value, `n`, that determines the interval (in seconds) between updates, using the formula:

  `Represented value = 1.1`$^{n-64}$

  For cases in which the measurement period is not available, the value 0x00 shall be assigned to indicate that the sampling function field is unknown or not provided. Valid settings for this parameter are show in Table 12.

Table 12. `update_interval` Parameter Values (Bluetooth SIG, 2019)

Value n	Represented Value	Description
0x00	Not Applicable	Not Applicable
0x01–0xFF	1.1^{n-64}	Update interval in seconds.

- `Data`: The sensor data variable or array address is assigned to this pointer.
- `Cadence`: The `Cadence` configuration structure, `wiced_bt_mesh_sensor_config_cadence_t`, contains the following parameters:
  - `.fast_cadence_period_divisor`

    The Fast Cadence Period Divisor parameter controls the increased cadence of publishing the Sensor Status messages. The fast publish period is calculated according to the formula below:

    ```
    mesh_sensor_fast_publish_period = mesh_sensor_publish_period / p_sensor-
    >cadence.fast_cadence_period_divisor;
    ```

    For example, if the `mesh_sensor_publish_period` is set to 320000 using a Mesh Helper application, then the `mesh_sensor_fast_publish_period` is 10 seconds (320000 / 32 milliseconds).
  - `.trigger_type_percentage`

    The Status Trigger Type field defines the unit and format of the Status Trigger Delta Down and the Status Trigger Delta Up parameters.

    - A value of ‘0’ means that the format of the data is as defined by the `property_id` parameter contained in the `sensors` element configuration structure.
    - A value of ‘1’ means that the unit is «unitless», the format type is set to 0x06 (uint16), and the value is represented as a percentage change with a resolution of 0.01 percent.

  - `.trigger_delta_down`

    The Status Trigger Delta Down parameter controls the negative change of a measured quantity that triggers publication of a Sensor Status message. The setting is calculated based on the value of the parameter `.trigger_type_percentage`

    - If the value of the `.trigger_type_percentage` parameter is ‘0’, the setting is calculated as defined by the `sensors property_id` parameter.
    - If the value of the `.trigger_type_percentage` parameter is ‘1’, the setting is calculated using the following formula:

`Represented value = .trigger_delta_down/ 100`

- `.trigger_delta_up`

  The Status Trigger Delta Up parameter controls the positive change of a measured quantity that triggers publication of a Sensor Status message. The setting is calculated based on the value of the parameter `.trigger_type_percentage`:

  - If the value of the `.trigger_type_percentage` parameter is '0', the setting is calculated as defined by the `sensors property_id` parameter.
  - If the value of the `.trigger_type_percentage` parameter is '1', the setting is calculated using the following formula:

  `Represented value = .trigger_delta_up/ 100`

- `.min_interval`

  The min interval parameter controls the minimum interval between publishing two consecutive sensor status messages. The value is represented as 2^n milliseconds.

  $\texttt{Min\_interval} = 2^n$ `milliseconds`

  The valid range of n is 0 to 26.

- `.fast_cadence_low`

  The Fast Cadence Low parameter defines the lower boundary of a range of measured quantities when the publishing cadence is increased as defined by the Fast Cadence Period Divisor field. The represented value is calculated as defined by the `sensors property_id` parameter.

  Note: The Fast Cadence Low may be set to a value higher than the Fast Cadence High. In such cases, the increased cadence will occur outside the range of the Fast Cadence High and Fast Cadence Low parameters.

- `.fast_cadence_high`

  The Fast Cadence High parameter defines the upper boundary of a range of measured quantities when the publishing cadence is increased as defined by the Fast Cadence Period Divisor field. The represented value is calculated as defined by the `sensors property_id` parameter.

  Note: The Fast Cadence High may be set to a value lower than the Fast Cadence Low. In such cases, the increased cadence will occur outside the range of the Fast Cadence High and Fast Cadence Low parameters.

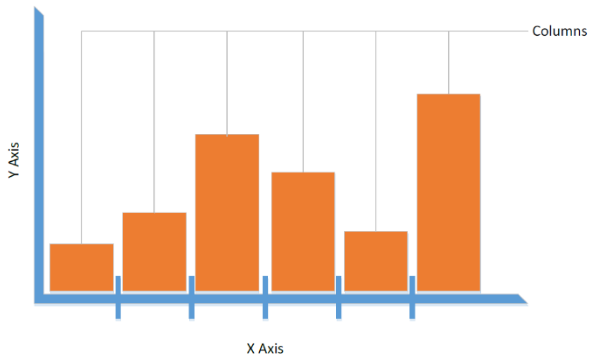
- `.num_series`: This parameter represents the number of `series_columns` defined for the Mesh device.
- `.series_columns`:

Values measured by sensors organized as arrays (and represented as series of columns, such as histograms, as illustrated by Figure 48). Table 13 summarizes the Sensor Series Column states. Each Sensor Series Column state represents a column of a series.

Table 13. Sensor `series_columns` Parameters (Bluetooth SIG, 2019)

Field	Size (octets)	Notes
Sensor Property ID	2	Property describing the data series of the sensor
Sensor Raw Value X	variable	Raw value representing the left corner of a column on the X axis
Sensor Column Width	variable	Raw value representing the width of the column
Sensor Raw Value Y	variable	Raw value representing the height of the column on the Y axis

Figure 48. Sensor Series Array (Bluetooth SIG, 2019)



- `.num_settings`: This parameter contains the number of settings present in the sensor.
- `.settings`: The `settings` configuration structure, `wiced_bt_mesh_sensor_config_setting_t`, contains the following parameters:
  - `setting_property_id:`

    The Sensor Setting Property ID field identifies the device property of a setting, including the size, format, and representation of the Sensor Setting Raw field.
  - `access:`

    The Sensor Setting Access field is an enumeration indicating whether the device property can be read or written.

    Table 14. `access` Parameter Values

Value	Meaning
0x00	Prohibited
0x01	The device property can be read.
0x02	Prohibited
0x03	The device property can be read and written.
0x04–0xFF	Prohibited

  - `value_len`:

    This parameter contains the length of the Sensor Setting Raw parameter value.
  - `value`:

    The Sensor Setting Raw parameter has a size and representation defined by the `setting_property_id` and represents a setting of a sensor.

Appendix C. iOS Helper App Installation

The iOS Helper app installation has the following requirements:

1. A MacBook with the latest version of BT SDK and Xcode IDE installed
2. An iPhone and a registered Apple ID and password

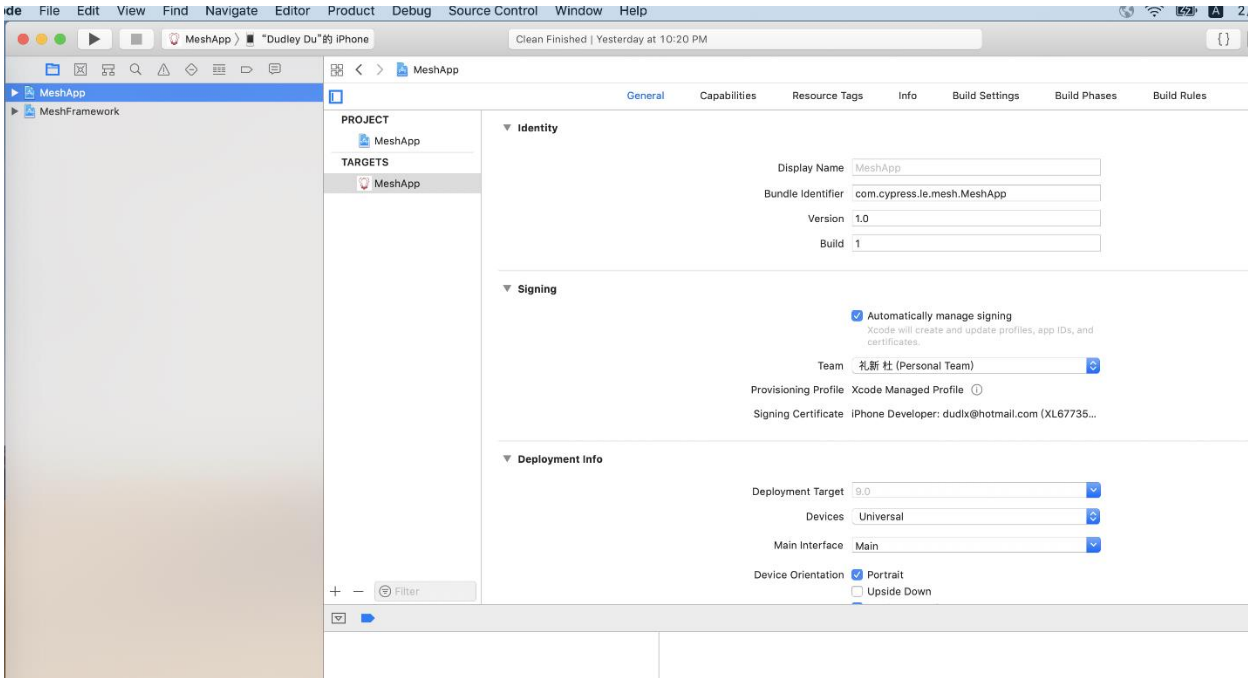
The latest version of the BT SDK can be found at the ModusToolbox BT SDK community page.

For the latest version, download the source code from the GitHub repository.

Once you have acquired all the pieces needed to install the iOS Helper app, follow these steps:

1. Connect the iPhone to the MacBook using phone’s data cable.
2. Go to *<Workspace Folder>\tools\btsdk-peer-apps-mesh\iOS\MeshApp* .
3. Double-click on the “MeshApp.xcworkspace” to open the MeshApp and MeshFramework projects in the Xcode IDE. The opened projects include the MeshApp and MeshFramework shown below.

Figure 49. Xcode IDE with MeshApp and MeshFramework Projects



Identity

Display Name: MeshApp
Bundle Identifier: com.cypress.le.mesh.MeshApp
Version: 1.0
Build: 1

Signing

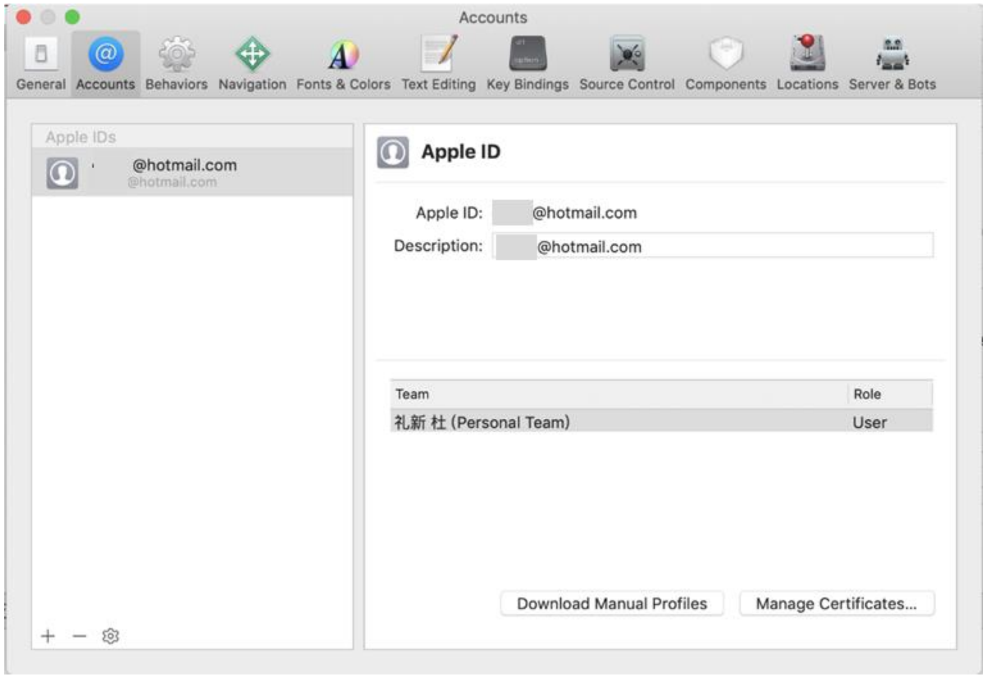
[✓] Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.
Team: 礼新 杜 (Personal Team)
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iPhone Developer: dudlx@hotmail.com (XL67735...

Deployment Info

Deployment Target: 9.0
Devices: Universal
Main Interface: Main
Device Orientation: [✓] Portrait, [ ] Upside Down

4. Click on **Xcode** > **Preferences…** in the Xcode menu to open the **Preferences** dialogue. Select the **Accounts** item. The Accounts page is like what is shown in Figure 50.

Figure 50. Xcode Accounts Page



@hotmail.com
@hotmail.com

Apple ID

Apple ID: @hotmail.com
Description: @hotmail.com

Team	Role
礼新 杜 (Personal Team)	User

Click the **+** button in the left bottom to add your Apple ID in Xcode, and then input your Apple ID and password to log in. After your Apple ID is added and you have logged in successfully, your Apple ID will be shown as displayed in Figure 50.

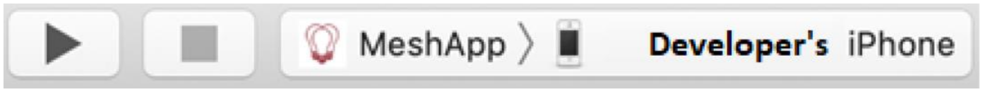
5. Click the MeshApp project on the left of Xcode as shown in Figure 49. Click the **General** item on the right side to display the Signing configuration.

6. As shown in Figure 49, enable **Automatically manage signing**, and then update the Team by selecting your Apple ID in the drop list.

7. Next, find the Identify configuration above the Signing configuration, and add a unique suffix string to the Build Identifier string, such as changing “com.cypress.le.mesh.MeshApp” to “com.cypress.le.mesh.MeshApp.Tom”. This is needed as the previous Build Identifier string may have been used, so you must use another unique Build Identifier to build and install the iOS MeshApp.

8. Make sure that the iOS MeshApp and the connected iPhone device have been selected to build and run the *app*. These selections can be done at the top of the Xcode IDE as shown in Figure 51.

Figure 51. Selecting the iOS MeshApp and the iPhone



9. Click **Product** > **Clean Build Folder** in the Xcode menu to clean the build system.

   Now, the build environment is ready.

10. Click the ▶ button on the top left of the Xcode IDE to build and install MeshApp automatically into the connected iPhone device. You will need to set the MacBook as a Trusted device in your iPhone settings, if it is not already done.

11. When you open the iOS app for the first time, it will require you to enter your Apple ID and Password. After this, your app is ready to create Mesh networks.

Appendix D. Using Android Helper App

For installation instructions and location of the .apk file and source code, see Section 4.2.2.

Note: Once installed, the app will appear on your phone as “MeshLighting”. When you open the app for the first time, it will ask for following four permissions:

1. Allow **MeshLighting** to take pictures and record video?
2. Allow **MeshLighting** to access this device’s location?
3. Allow **MeshLighting** to access photos, media, and files on your device?
4. Allow **MeshLighting** to access your contacts?

Select **Allow** for all these. If these are not allowed, the app will not work.

Once you have granted the permissions, you are set to use the app to create Mesh networks.

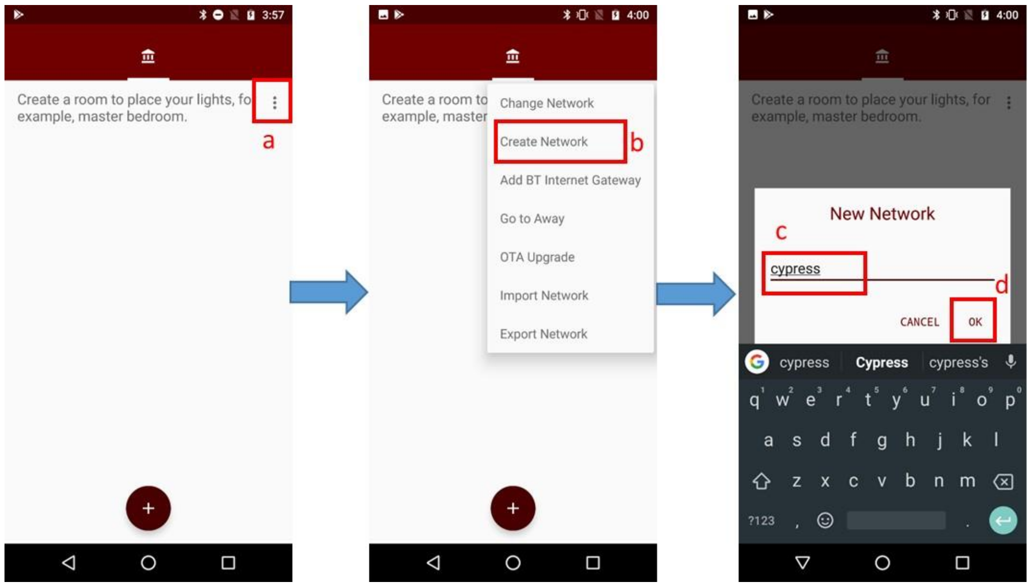
The Android app is very intuitive and does not require a detailed user guide. However, the following steps can be referred to while using the app. This section uses Use Case #1 (Mesh network with 1 Dimmer Switch and 3 Dimmable Light Bulbs) in this application note to demonstrate Android app usage.

1. Follow the below steps to create a Mesh network.
   a. On home screen, tap on the three dots next to **Create a room to place your lights, for example, master bedroom.**
   b. Select **Create Network**.

      A new window, *New Network*, opens where a network can be created. In this case, the network name is assigned as “cypress”. You can use a custom name for your network.
   c. Once the name is entered, select **OK**.

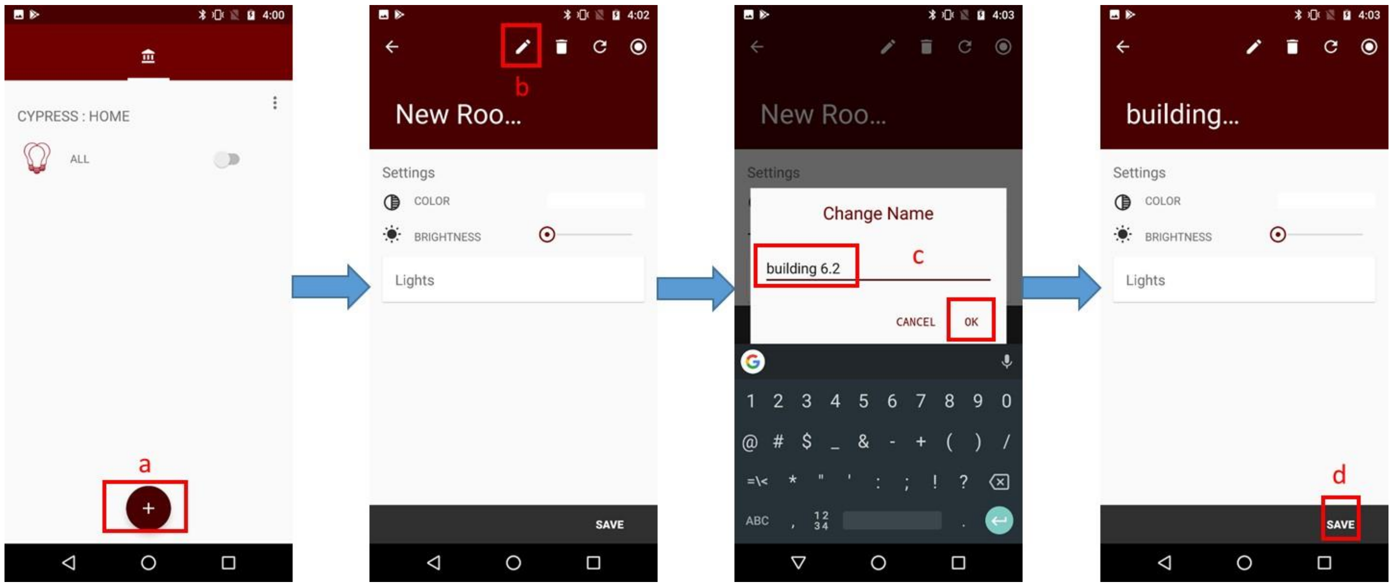
      Observe that a new network with the name “cypress” is created.

Figure 52. Creating a Mesh Network



2. Follow these steps to create a group.
   a. Tap on the **+ (**Add A GROUP) button located towards the bottom of the app screen.
   b. This brings you to a new screen with a default room named “New Room”. To change the room name, select Pencil symbol at the top of the screen.
   c. Enter the desired name of the group and tap **OK**. In this example, the name of the group is “building 6.2”
   d. To save group name and settings, tap **Save**.

Figure 53. Creating a Room and Updating the Name



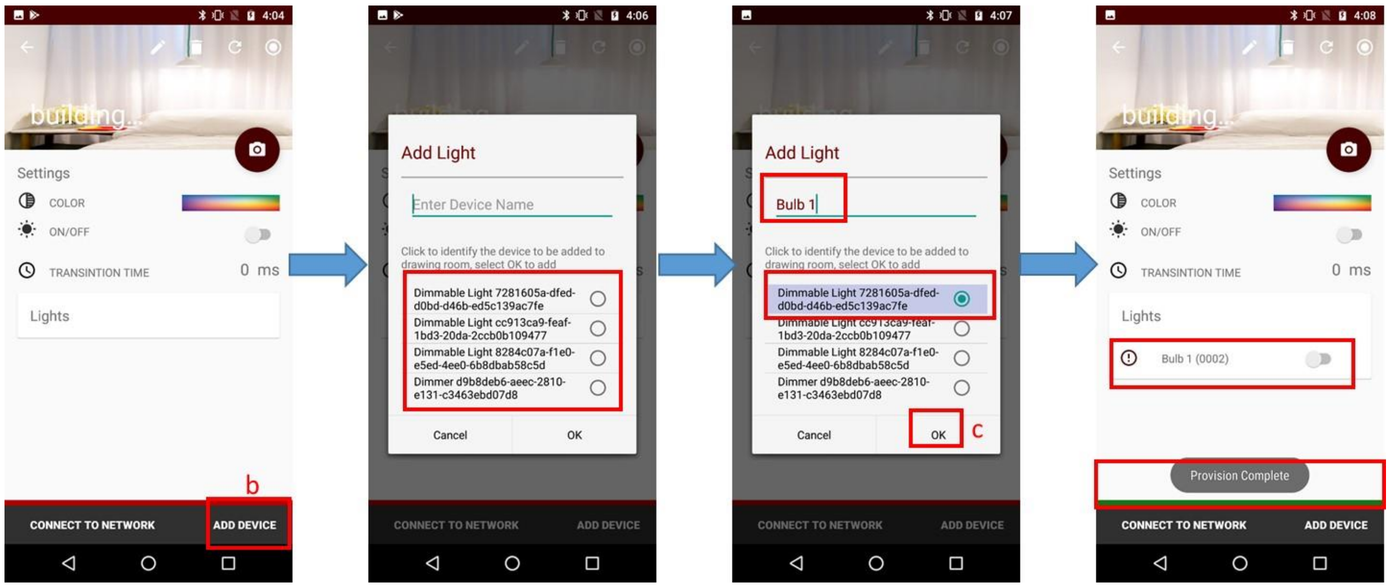
3. After the group is created, follow these steps to add nodes to the group “building 6.2”:
   a. Before you proceed with these steps, ensure that all evaluation boards are powered ON.
   b. Select **Add Device**.

      A new window will pop up displaying all unprovisioned Mesh devices that are advertising. In this case, as all devices are powered, it will show all four devices we have programmed. Dimmable Light Bulbs are shown as Dimmable Light with their associated Bluetooth address.
   c. Tap on one Dimmable Light device and then enter the name you want to give to this node. For this example, we will use name “Bulb 1”. Then select **OK** to start the “Provisioning” and “Configuration” process. Observe the status message “Provision Complete” after the process is complete. Do not take any action until you see this message. Once you see the message, the Dimmable Light device with the name “Bulb 1” is part of the Mesh network.
   d. Repeat these steps for the other three devices for Use Case #1. In this example, we have used names “Bulb 2” and “Bulb 3” for the other two Dimmable Light devices and “Dimmer” for Dimmer Switch device. You can use the name of your choice.

      Note: If you want to know exactly which node you are provisioning, you can power one node at a time and provision it. Then repeat it for other unprovisioned nodes.

      Note: After you have completed Step 3 and provisioned the three Dimmable Light Bulbs and the Dimmer Switch evaluation boards, you can control the Dimmable Light Bulbs using either 1) the User Button on the Dimmer Switch Evaluation board, or 2) the Helper Application. Refer to Step 4 for instructions on how to control the Dimmable Light Bulbs using the Dimmer Switch. Refer to Steps 5~6 for instructions on how to control the Dimmable Light Bulbs using the Android Helper app.

Figure 54. Node Provisioning – Adding the Dimmable Light Bulb and Dimmer to the Group



Click to identify the device to be added to drawing room, select OK to add

Dimmable Light 7281605a-dfed-d0bd-d46b-ed5c139ac7fe
Dimmable Light cc913ca9-feaf-1bd3-20da-2ccb0b109477
Dimmable Light 8284c07a-f1e0-e5ed-4ee0-6b8dbab58c5d
Dimmer d9b8deb6-aeec-2810-e131-c3463ebd07d8

4. The Dimmer Switch node can control the Dimmable Light Bulbs directly without the need of the Cypress Android Helper app. The Dimmer Switch controls the LED on the three Dimmable Light Bulbs through the User Button on the Dimmer Switch evaluation board. The User Button actions and associated Mesh messages that are sent out from the Dimmer Switch are detailed below:

   a. A short press and release of the User Button on the Dimmer Switch node will toggle the ON/OFF state of the LED on the Dimmable Light Bulb nodes.

   b. A long press of User Button on the Dimmer Switch node sends a 12.5% level increase command every 500 milliseconds. It takes 4 seconds for the level of the Dimmable Light Bulb Nodes to increase from 0% to 100%. The Dimmer Switch node will stop sending commands after 100% level setting is achieved.

   c. After the Dimmer Switch has reached the minimum or maximum level setting (0% or 100%), releasing the User Button and repeating a long press will repeat the inverse level controls to the Dimmable Light Bulbs. For example, in Step 4.b above, the Dimmable Light Bulbs level setting were increased to 100%. After releasing the User Button and repeating the long press of the User Button, the level setting will decrease from 100% to 0% in 12.5% decrements every 500 milli-seconds.

5. The Cypress Android Helper app can also be used to control the Dimmable Light Bulbs. To access and control the nodes in the group using the Android Helper app, the phone must be connected to the network. If a node supports the Proxy feature, by default the phone will remain connected to this Proxy node after it completes provisioning. If the phone is connected to the network, a green bar is shown at the bottom of the Android app. A red bar in this location indicates that the phone is not connected to the network. If the Dimmer Switch is the last node that is provisioned, the phone will disconnect from the network after provisioning as the Dimmer Switch node does not support the Proxy feature. Therefore, before you can control these nodes, ensure that the status bar is green. If the status bar is red, follow these steps to connect to the network:

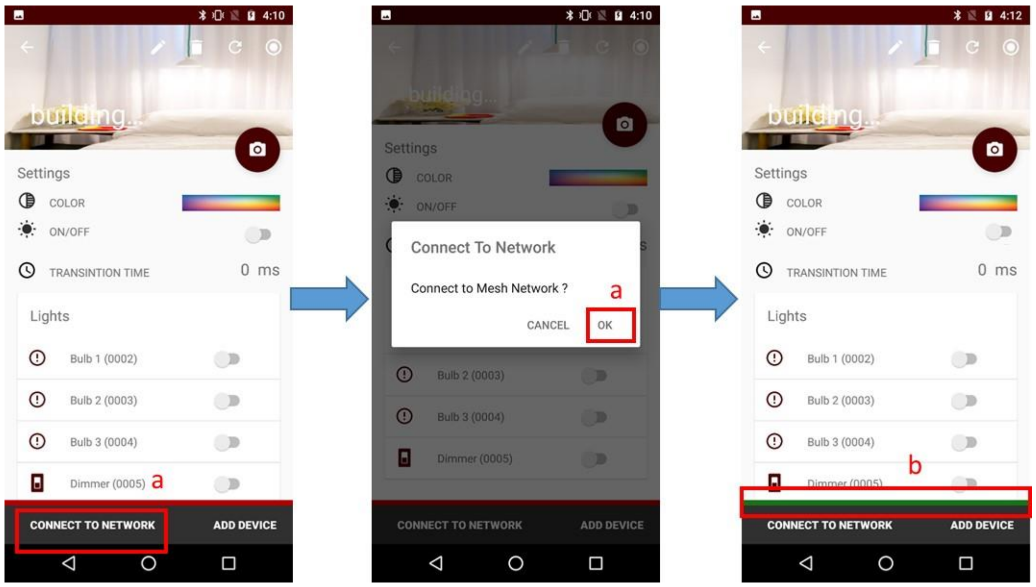
   a. Select the **Connect To Network** button on the bottom right hand side of the screen.

      A new screen will be prompted that will ask if you want to connect to the Mesh Network. Select **OK**.

   b. Wait until status bar turns green.

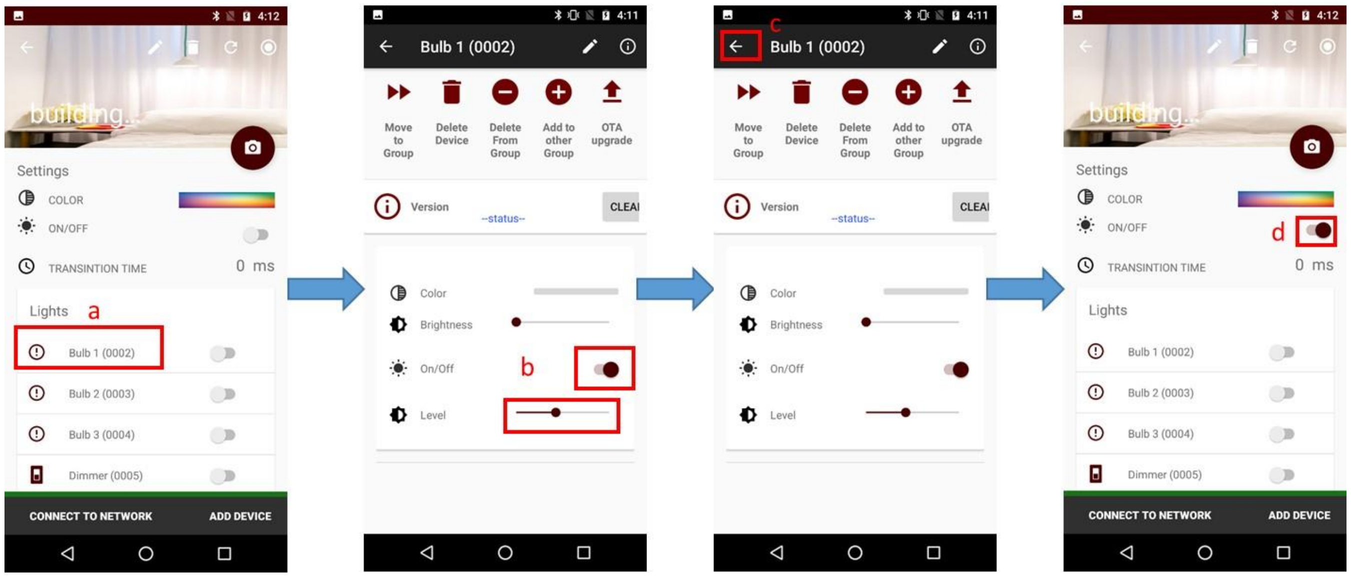
      Note that this will only work if at least one device that supports the Proxy feature is within range of the Android device.

Figure 55. Connecting to the Network



6. Follow these steps to access the nodes in the group.
   a. To control each node individually, select the individual node. This will bring up a screen where this node can be accessed.
   b. Tap on the **ON/OFF** button to change state from Off to On and vice-versa. You can control the LED intensity by changing the **Level** slider position. Observe the LED state and intensity while you modify the settings in the Android app.
   c. Once done, tap on the arrow on the top left of the screen to go back to the group.
   d. To control the entire group, select the **ON/OFF** button on the group's home screen. Observe that the LEDs on all Dimmable Light nodes will change according to your changes.

Figure 56. Controlling the Dimmable Light Bulbs in The Group



Document History

Document Title: AN227069 – Getting Started with Bluetooth Mesh

Document Number: 002-27069

Revision	ECN	Date	Description of Change
**	6557343	05/13/2019	New application note
*A	6885892	05/22/2020	Added support for ModusToolbox 2.1 and wiced_btsdk 2.1 or later Updated low_power_led project section

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com
www.infineon.com