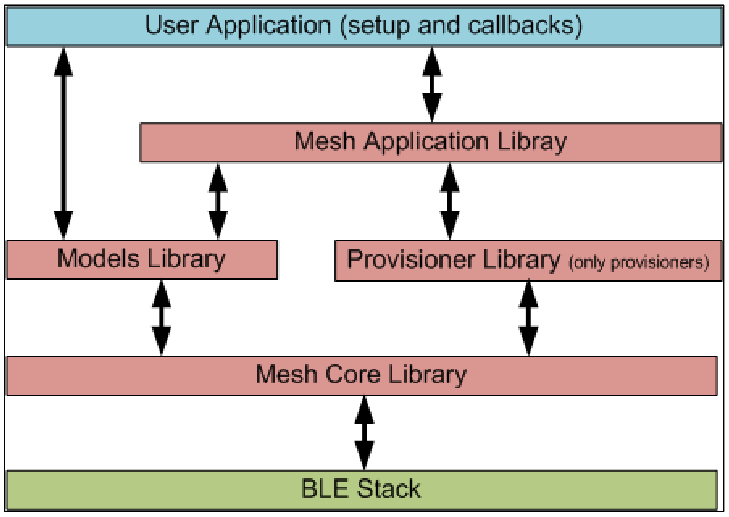
# Chapter 7C: Bluetooth Mesh Firmware

This chapter covers mesh code examples and mesh firmware architecture. It also covers client applications that can be used to provision and interact with mesh devices.

## 7C.1 Firmware Architecture


User Application (setup and callbacks)
Mesh Application Libray
Models Library
Provisioner Library (only provisioners)
Mesh Core Library
BLE Stack

### 7C.1.1 Mesh Application and Libraries

To access the mesh functionality, it is necessary to include the Mesh library (*btsdk-mesh*) in the application's dependencies. In the *Makefile*, you must include components for `mesh_app_lib` and either the release or debug versions of `mesh_core_lib` and `mesh_models_lib`. For example:

```
COMPONENTS += mesh_app_lib

# prebuilt libs - link release libs by default, or debug trace enabled if flag
set
ifeq ($(MESH_CORE_DEBUG_TRACES),1)
COMPONENTS += mesh_core_lib_debug
else
COMPONENTS += mesh_core_lib
endif
ifeq ($(MESH_MODELS_DEBUG_TRACES),1)
COMPONENTS += mesh_models_lib_debug
else
COMPONENTS += mesh_models_lib
endif
```

The Mesh functionality is implemented in a set of files located in:

*wiced_btsdk/dev-kit/libraries/btsdk-mesh/<version>/COMPONENT_mesh_app_lib*

The user does not typically need modify any of the files in *mesh_app_lib*. They are:

| | |
|---|---|
| *mesh_application.c/.h* | Top-level file containing `application_start`, stack initialization, Bluetooth management callback function, and other helper functions. |
| *mesh_app_gatt.c* | GATT database setup, GATT callback function, and all other GATT related functions. |
| *wiced_bt_cfg.c* | Bluetooth configuration including advertisement settings. |
| *mesh_app_hci.c* | Functions used for sending/receiving data to/from a host when the mesh device is used in HCI mode. |
| *mesh_app_provision_server.c* | Functions that allow the mesh device to be provisioned onto a network. |

Note that the lower level BT mesh core, client, and models library functions are in pre-compiled libraries so that source code is not provided. However, the API can be found in header files at:

*wiced_btsdk/dev-kit/btsdk-include/<version>/wiced_bt_mesh_*.h*

The easiest way to get to these files from inside the IDE are to right-click on an item such as a datatype for a structure and select **Open Declaration** [**F3**].

### 7C.1.2 User Application

The user application includes the Bluetooth and Mesh header files, sets up several variables and structures to configure mesh behavior, then registers and defines callback functions for various events that will be called from the mesh firmware.

Note that the user application does NOT contain the `application_start` function. It is pre-defined in the *mesh_application.c* file in the mesh library and should not be modified. All user application functionality is handled in the various callback functions which will be discussed below.

#### Includes

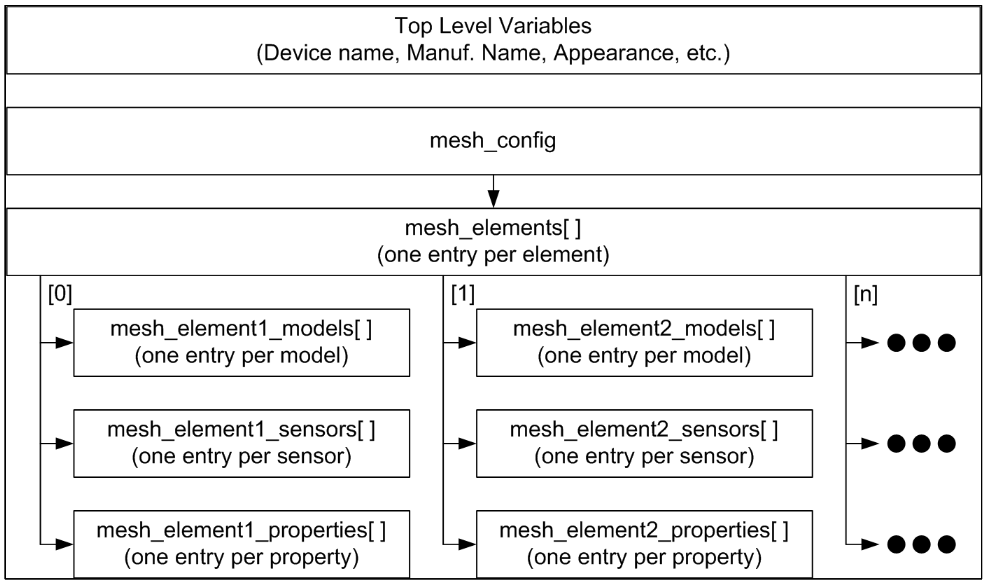
The user application starts with at least the following includes to get access to the Bluetooth, mesh, trace, HCI library functions and BT configuration from the mesh library:

```
#include "wiced_bt_ble.h"
#include "wiced_bt_gatt.h"
#include "wiced_bt_mesh_models.h"
#include "wiced_bt_trace.h"
#include "wiced_bt_mesh_app.h"

#ifdef HCI_CONTROL
#include "wiced_transport.h"
#include "hci_control_api.h"
#endif

#include "wiced_bt_cfg.h"
```

There may be other header files included depending on the application's functionality.

## Configuration Variables

After the include files, there is a set of variables that are used to configure the application. These variables configure names, models, properties, elements, and features such as relay, GATT proxy, friend, and low power. The configuration is done hierarchically, and it looks like the following. Details of each item are discussed below (from the bottom up).



### *Top Level Variables*

The first set of variables set up the manufacturer name, model number, etc. as shown with an example here:

```
uint8_t mesh_mfr_name
    [WICED_BT_MESH_PROPERTY_LEN_DEVICE_MANUFACTURER_NAME] = {'C','y','p','r','e','s','s',0};
uint8_t mesh_model_num
    [WICED_BT_MESH_PROPERTY_LEN_DEVICE_MODEL_NUMBER]       = {'A','1','9',0};
uint8_t mesh_prop_fw_version
    [WICED_BT_MESH_PROPERTY_LEN_DEVICE_FIRMWARE_REVISION] = {'0','6','.','0','2','.','0','5'}; // this is overwritten during init
uint8_t mesh_system_id
    [8]                                                    = { 0xbb,0xb8,0xa1,0x80,0x5f,0x9f,0x91,0x71};
```

### *Models*

The next item is an array that specifies the models that will be implemented in the device. In fact, you need one of these arrays for each element in the device. So, if the device has two elements, you would have two arrays which each specify the models used in one of the two elements.

For the primary element, the array must contain `WICED_BT_MESH_DEVICE`. It will also contain other models to implement the required functionality of the primary element. For example, if the primary element has a user property server, the array will have an entry for `WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER`. Secondary elements (if there are any) do not require `WICED_BT_MESH_DEVICE` but will contain any other models required for the element's functionality.

An example of a model array for a primary element with three models 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,
};
```

### Sensors

If your device implements sensor models, then you will need an array for each element containing sensors to configure each sensor. Each array has one structure entry per sensor in the given element. If your device does not implement any sensor models, then these arrays are not required and can be deleted.

An example for a configuration array for a single temperature sensor is shown here:

```
wiced_bt_mesh_core_config_sensor_t mesh_element1_sensors[] =
{
    {
        .property_id = WICED_BT_MESH_PROPERTY_PRESENT_AMBIENT_TEMPERATURE,
        .prop_value_len = WICED_BT_MESH_PROPERTY_LEN_PRESENT_AMBIENT_TEMPERATURE,
        .descriptor =
        {
            .positive_tolerance = MESH_TEMPERATURE_SENSOR_POSITIVE_TOLERANCE,
            .negative_tolerance = MESH_TEMPERATURE_SENSOR_NEGATIVE_TOLERANCE,
            .sampling_function  = MESH_TEMPERATURE_SENSOR_SAMPLING_FUNCTION,
            .measurement_period = MESH_TEMPERATURE_SENSOR_MEASUREMENT_PERIOD,
            .update_interval    = MESH_TEMPERATURE_SENSOR_UPDATE_INTERVAL,
        },
        .data = (uint8_t*)&mesh_sensor_sent_value,
        .cadence =
        {
          // Value 1 indicates that cadence does not change depending on the measurements
            .fast_cadence_period_divisor = 1,
            .trigger_type_percentage     = WICED_FALSE,
            .trigger_delta_down          = 0,
            .trigger_delta_up            = 0,
            .min_interval                = (1 << 12), // minimum interval for sending data by
default is 4 seconds
            .fast_cadence_low            = 0,
            .fast_cadence_high           = 0,
        },
        .num_series     = 0,
        .series_columns = NULL,
        .num_settings   = 1,
        .settings       = sensor_settings,
    },
};
```

### Properties

If any of the elements have properties associated with them, they are specified in an array of property structures. Each entry in the array specifies one property.

Each element in the device that contains properties has its own array to specify those properties just like with the model and sensor arrays. If there are no properties for a given element, then this array is not required and can be deleted.

An example of a property array for an element with one property (firmware revision) is shown here:

```
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
    },
};
```

### *Elements*

This is one array that configures the elements in the device. There is one entry in the array for each element in the device. The entries define the element's location, default transition time, power up state, etc. Each entry also points to the model array, sensor array (if there is one) and property array (if there is one) for that element.

An example element array for a device with one element is shown here. Note the pointers provided to `mesh_element1_properties` and `mesh_element1_models`. In this case there is no sensor so `sensors_num` is set to `0` and `NULL` is specified for the pointer. The code example applications include comments explaining what each element means.

```
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,
    },
};
```

### *Device Configuration*

The final structure contains information such as product ID, vendor ID, etc., as well as features such as relay, GATT proxy, friend, and low power. This structure also specifies how many replay caches are available, which limits how many other devices can send application messages to this device. If the node is a friend node or a low power node then those settings are specified in this structure (details below). Finally, a pointer to the element array created previously is provided.

The following is an example mesh configuration structure. Note the pointer provided to the `mesh_elements` array. The code example firmware includes comments explaining what each entry means.

```
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
};
```

The Friend configuration parameters are:

- **`receive_window`**: The time window (in ms) that an LPN must listen for a response from its Friend. Lower values allow LPN nodes to turn off quicker thereby saving more power since they don't need to listen in as wide a window but that gives the Friend node less time to prepare a response (see the figure below `receive_delay`)
- **`cache_buf_len`**: The number of bytes allocated to store messages for all LPNs supported by the Friend. Each message is 34 bytes long.
- **`max_lpn_num`**: Maximum number of LPNs that can be supported by a Friend.

The LPN configuration parameters are:

- **`rssi_factor`** and **`receive_window_factor`**: These are weights that are used to determine how "good" a potential friend will be. Both have a range of 1 – 2.5 in steps of 0.5. They allow the application to prioritize the importance of signal strength vs. delay of response. A potential Friend sees a friend request from an LPN and sends its friend offer after a delay calculated using this formula:

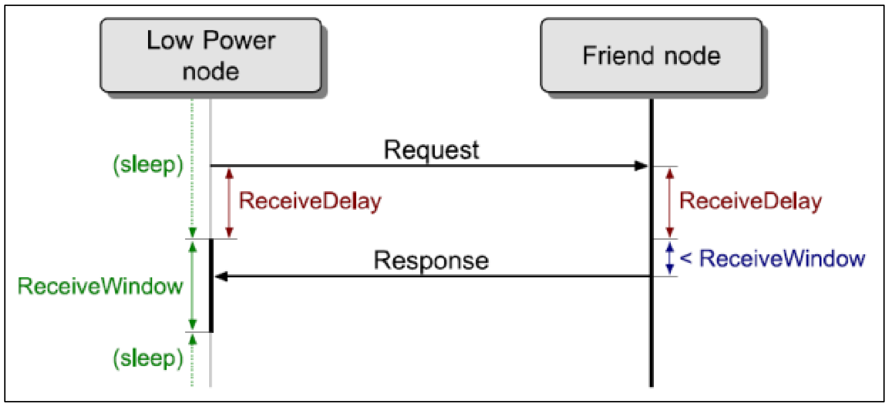
  Friend offer delay = (receive_window_factor * receive_window) – (rssi_factor * RSSI)

  An LPN will pick the first friend to respond, since it has the lowest calculated value.

- **`min_cache_size_log`**: The minimum number of messages (not bytes) the Friend must be able to hold for the LPN.

- **`receive_delay`**: This is a time (in ms) requested by the LPN before the Friend can respond to a request. This allows the LPN to stop listening for a time while the Friend prepares its response.

  The relationship between `receive_delay` and `receive_window` can be seen here:

  (This figure is taken from the Bluetooth Specification)

- **`poll_timeout`**: This is the maximum time (in ms) between consecutive requests from an LPN. If a Friend node doesn't get a request within `poll_timeout`, it will terminate the friendship. The LPN will not know the friendship is terminated until it sends a request 6 times and does not receive a response. At that point the LPN will begin searching for a new Friend.

## User Function Callbacks

After the configuration variables, there must be a structure defined that contains a table of pointers to the functions that will be called by various mesh events. The structure looks like the following. Note that `NULL` can be specified for any of the functions that are not required by the user application functionality. Each of the items from the table are discussed in more detail below.

```
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
};
```

### *Application Initialization*

This function is called once the mesh stack is running and it is ready for the user application to perform any initialization that it requires.  The initialization function takes one `wiced_bool_t` parameter called `is_provisioned` which is passed in by the stack. That is, the prototype for the application initialization function is:

```
typedef void(*wiced_bt_mesh_app_init_t)(wiced_bool_t is_provisioned);
```

The application initialization function generally does the following things:

1. (Optional) If provisioned, the device may remove provisioning information after a set number of reset cycles.

2. Set the device name appearance, firmware version, etc. For example:

```
    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);
     // 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);
```

3. (Optional) If not provisioned, setup a scan response packet that a provisioning tool can look at.

4. Do any required initialization for the user's application.

5. Initialize any models, sensors, or property servers that the device requires (other than the required core mesh models). For the element model example shown in the previous section, the following would be placed in the application initialization function:

```
// 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);
```

The arguments to these initialization functions are:

- Element Index: This is the index of the entry in the element array.
- Callback function: The function that is called when a message is received for the model (use `NULL` if the user application doesn't need a message callback)
- `is_provisioned`: This indicates whether a device has been provisioned or not. If a device has not been provisioned then configuration data for the model is deleted from NVRAM.

### *Button processing*

The function specified here is a callback to the user application to configure button functionality. Typically, the user would configure the buttons with interrupts and then would use interrupt callback functions to process button events. The prototype for this function is:

```
typedef void(*wiced_bt_mesh_app_hardware_init_t)(void);
```

If the platform being used as the target device contains a push button but this callback is specified as `NULL`, the button is configured in the *mesh_application.c* library file to perform a factory reset on the device whenever the button is pressed for 3 or more seconds and then released. Factory reset will return the device to an unprovisioned state.

### *GATT connection status*

The function specified here is called when a GATT connection event occurs. That is, when there is a connection up or down GATT callback event. It is passed a structure with the GATT connection status of type `wiced_bt_gatt_connection_status_t`. That is, the prototype for the function is:

```
typedef void(*wiced_bt_mesh_app_gatt_conn_status_t)(wiced_bt_gatt_connection_status_t *p_status);
```

### *Attention processing*

This function is called by the stack whenever it wants the device to alert the user of its presence. For example, it may be used to identify the device that is being provisioned so that the user can visually identify it from a group of unprovisioned nodes.

It is up to the hardware to determine how that alert should be done – e.g. blink and LED, sound a buzzer, etc. The stack provides the index of the element that should be used and a length of time for the alert in seconds from `0` (stop alert) up to `0xFF`. The prototype for the alert function is:

```
typedef void (*wiced_bt_mesh_app_attention_t)(uint8_t element, uint8_t time);
```

Since the stack provides a length of time for the attention signal to occur, it is typical for the callback function to use a "seconds" timer to control when the attention signal stops.

### *Notify Period Set*

Depending on the model, this function may be called to indicate to the application that it needs to periodically send updates. The function prototype is:

```
typedef wiced_bool_t (*wiced_bt_mesh_app_notify_period_set_t)(uint8_t element,
                    uint16_t company_id,
                    uint16_t model_id, uint32_t time);
```

The arguments provided by the stack are the `element`, `company_id`, `model_id`, and the time for periodic updates to be sent in seconds. If the model is a BT SIG defined model, the `company_id` will be `MESH_COMPANY_ID_BT_SIG`. If it is a vendor specific model, the `company_id` will be specific to that vendor (e.g. `MESH_COMPANY_ID_INFINEON`).

If the value of time indicated by the stack is 0 then the user application should NOT send periodic updates.

The return value is a `wiced_bool_t` which should be returned as `WICED_TRUE` if the application will take care of periodic publications for the specified `element`, `company_id`, and `model_id`. This allows the application to handle periodic status updates for some models but not others.

It is mandatory to implement this functionality on sensors (i.e. `WICED_BT_MESH_MODEL_SENSOR_SERVER`). If a device does not have sensors, support for this callback is optional. If a device does not implement this callback it can just specify `NULL`. Applications that do not want to handle this callback should report all changes to the local states to the Mesh Models library. This will be discussed in a minute.

#### *WICED HCI command*

This function is used if the application needs to interpret HCI commands received from a host device.

```
typedef uint32_t (*wiced_bt_mesh_app_proc_rx_cmd_t)
                 (uint16_t opcode, uint8_t *p_data, uint32_t length);
```

As you can see, the function takes three arguments: the command opcode, a pointer to the command parameters, and the length of the command parameters.

#### *Low Power Node (LPN) Sleep*

The stack will call this function when it is safe to enter a low power mode. The stack provides the length of time in milliseconds that it does not require any processing, so the application can decide if it wants to go to sleep, and what sleep mode is appropriate. It is up to the user application to determine whether to enter a low power mode or not, and to wake up at the appropriate time. The prototype for this function is:

```
typedef void (*wiced_bt_mesh_app_lpn_sleep_t)(uint32_t duration);
```

For example, an application may go into HID-Off if the sleep time is longer than 2 minutes and ePDS if the sleep time is shorter than 2 minutes. The template application Mesh Demo Low-Power LED demonstrates this approach.

#### *Factory Reset*

If a function is provided here, it will be executed before a factory reset is performed. This is necessary if the user application has NVRAM data that needs to be erased when a factory reset is done. After it returns, a factory reset will be performed to return the device to an unprovisioned state. The prototype for this function is:

```
typedef void (*wiced_bt_mesh_app_factory_reset_t)(void);
```

## Sending Messages

The model initialization functions described above allow the user to specify a callback function when a message is received. On the other hand, when the application has a message that it needs to send, it must call a function to send the appropriate message. For example, a Generic OnOff client may send a message using:

```
wiced_result_t wiced_bt_mesh_model_onoff_client_set(uint8_t element_idx,
       wiced_bt_mesh_onoff_set_data_t* p_data);
```

Likewise, a sensor server may send a status message using:

```
wiced_result_t wiced_bt_mesh_model_sensor_server_data(uint8_t element_idx,
             uint16_t property_id,
             void *p_ref_data);
```

More examples of messages will be seen in the mesh demo applications in the next section.

### Notifying the Mesh Model Library of Local Changes

In addition to sending messages over the network, it is also necessary in some cases to notify the Mesh Model Library of changes to local values. That is, if a value can be changed locally, the change should be reported so that the stack will be able to send the correct value when a client on the network requests it. For example, if a light on a light lightness server device can be turned on or off by the local hardware, it should use the following function to keep the library synchronized whenever the local value changes:

```
void wiced_bt_mesh_model_light_lc_onoff_changed(uint8_t element_idx,
                                              wiced_bt_mesh_onoff_set_data_t *p_status);
```

This is not required for sensor models because in that case the stack will call a report callback function whenever it receives a get message from a client, so it will always get the most recent sensor value before sending it out.

## 7C.2 MESH Demo Starter Applications

ModusToolbox contains a wealth of demo and snip applications for mesh networking. Mesh Demo applications contain fully-featured applications that demonstrate a complete mesh device. Mesh Snip applications contain smaller examples each of which demonstrate one of the SIG BLE mesh models. At the time of this writing, every SIG defined Mesh Model has a snip example.

The remainder of this section coves details of four of the mesh demo applications and explains how they interact with each other.

### 7C.2.1 Mesh Demo Dimmable Light (Server)

Most of the code snippets shown as examples up to now have been from the Mesh Demo Dimmable Light application so most of the configuration is already familiar to you. This application controls an LED which can be turned on/off and can also be dimmed via a PWM.

#### Configuration

The configuration sets up models, properties and features using the variables and structures as described in the previous section. The models, sensors, properties and features implemented are as follows:

| | |
|---|---|
| Models | WICED_BT_MESH_DEVICE<br>WICED_BT_MESH_MODEL_USER_PROPERTY_SERVER<br>WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER |
| Sensors | N/A |
| Properties | WICED_BT_MESH_PROPERTY_DEVICE_FIRMWARE_REVISION |
| Features | WICED_BT_MESH_CORE_FEATURE_BIT_FRIEND<br>WICED_BT_MESH_CORE_FEATURE_BIT_RELAY<br>WICED_BT_MESH_CORE_FEATURE_BIT_GATT_PROXY_SERVER |

## User Application Functionality

The top-level file for the user application is *light_dimmable.c*. There are helper functions to control the LED in *led_control.c/.h*.

Two of the mesh callback functions are registered for this application: application initialization (`mesh_app_init`) and attention processing (`mesh_app_attention`).

**`mesh_app_init`**: The application initialization function does the following:

- Factory reset the device (i.e. remove provisioning info) if it is turned ON/OFF 5 times with an ON time less than 5 seconds each time
- Set the device name, appearance, and firmware version
- Start advertisements if the device isn't provisioned
- Call a function to set up a PWM that will be used to drive the LED (`led_control_init`)
- Initialize a 1 second timer (including specifying the timer callback function (`attention_timer_cb`) that will be used for attention processing
- Initialize the two models:
    - The light lightness server model registers a message callback function (`mesh_app_message_handler`)
    - The property server model does not register a callback because the only property is read only so there is no action required for received messages. That is, the stack will handle sending the fixed property value when it is requested.

**`mesh_app_message_handler`**: The message handler for the light lightness server model only responds to one event - `WICED_BT_MESH_LIGHT_LIGHTNESS_SET`. For that event, the message handler calls the function `mesh_app_process_set_level` which takes the value of `lightness_actual_present` and converts it to a percentage brightness as an 8-bit value. The value is then sent to the function `led_control_set_brighness_level` which sets the PWM duty cycle to achieve the desired brightness.

**`mesh_app_attention`**: The attention callback starts the timer (or stops it if the attention time is 0). Every 1 second, the timer callback is called, and it does whatever is necessary including decrementing the remaining time.

**`mesh_app_fast_power_off_execute:`** This function looks to see if the device has been powered ON/OFF a total of 5 times in a row with the ON time being less than 5 seconds each time. If that condition is satisfied, the provisioning information is removed from the device. This is useful for Mesh devices that don't have a button or other way to reset them to the factory default (e.g. a light bulb).

**Button Processing**: Since no callback function is provided for button processing, the default behavior is used. That is, if the target kit has a button defined in its platform, pressing the button for >3 seconds then releasing will result in a factory reset that removes all mesh provisioning information from the device.

### 7C.2.2 Mesh Demo On-Off Switch (Client)

This application demonstrates a basic on/off client. It can be used in conjunction with a kit that is running the Mesh Demo Smart Light application so that it can control the LED on that kit over the mesh network.

#### Configuration

The configuration sets up models, properties and features using the variables and structures as described in the previous section. The models, sensors, properties and features implemented are as follows:

| | |
|---|---|
| Models | `WICED_BT_MESH_DEVICE`<br>`WICED_BT_MESH_MODEL_ONOFF_CLIENT` |
| Sensors | `N/A` |
| Properties | `N/A` |
| Features | `WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER`<br>(if enabled in application settings) |

#### User Application Functionality

The top-level file for the user application is *on_off_switch.c*.

Two of the mesh callback functions are registered for this application: application initialization (`mesh_app_init`) and button processing (`mesh_app_hardware_init`).

**`mesh_app_init`**: The application initialization function in this case only needs to:

1. Set the device name and appearance.
2. Start advertisements if the device is not provisioned.
3. Initialize the onoff client model. The onoff client model has no callback function because it does not check the result of transmission messages or status messages.

**`mesh_app_hardware_init`**: This function sets up the default button on the kit to have an interrupt on both edges with the callback function `button_interrupt_handler`. It also stores the default state of the button to a global variable. All other application functionality is handled by the interrupt callback.

**`button_interrupt_handler:`** This function does the following:

- Check if the button state has changed. If it has:
  - If the button is pressed, increment a counter
  - If the button is released, calculate the time that it was pressed:
    - If the button was pressed more than 15 seconds, do a factory reset.
    - Otherwise, call the `process_button_push` function.

**`process_button_push:`** This function sends a mesh message to toggle the on/off state of whatever device(s) it is controlling. It uses the default transition time and a delay of 0.

### 7C.2.3 (Advanced) Mesh Demo Dimmer (Client)

This application demonstrates a level client. It can be used in conjunction with a kit that is running the Mesh Demo Dimmable Light application so that it can control both on/off and dimming of the LED on that kit over the mesh network.

#### Configuration

The configuration sets up models, properties and features using the variables and structures as described in the previous section. The models, sensors, properties and features implemented are as follows:

| | |
|---|---|
| Models | `WICED_BT_MESH_DEVICE`<br>`WICED_BT_MESH_MODEL_LEVEL_CLIENT` |
| Sensors | `N/A` |
| Properties | `N/A` |
| Features | `WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER`<br>(if enabled in application settings) |

#### User Application Functionality

The top-level file for the user application is *dimmer.c*. There are helper functions to handle the button in *button_control.c/.h.*

Two of the mesh callback functions are registered for this application: application initialization (`mesh_app_init`) and button processing (`button_hardware_init`).

**`mesh_app_init`**: The application initialization function:

1. Set the device name and appearance.
2. Start advertisements if the device is not provisioned.
3. Calls the helper function `button_control_init`.
4. Initializes the level client model. The model has a callback function called `mesh_app_message_handler` that is used to act on messages that come back from the server.

**`mesh_app_message_handler`**: The message callback function for the level client handles two events:

> `WICED_BT_MESH_TX_COMPLETE`: This event occurs when the light lightness model responds that a transmission is complete.
>
> `WICED_BT_MESH_LEVEL_STATUS`: This event occurs when a status message is received from the light lightness model. The callback function prints out the current level, the target level, and the time remaining in the transition.

**`button_control_init`**: This function initializes a timer and registers the callback function `button_timer_callback`. It then saves the current button state.

**`button_hardware_init`**: This function just sets up the default button on the kit to have an interrupt on both edges with the callback function `button_interrupt_handler`. All other application functionality is handled by the interrupt callback.

**`button_interrupt_handler`**: This function does the following:

- If the button state has not changed, just return.
- If the button is pressed, start a 500 ms timer.
- If the button was released after being held down for less than 500 ms, call `button_set_level` to send a message to toggle the LED on or off.
- If the button has been held down for more than 500ms but less than 15 seconds, call `button_set_level` to send a final message to adjust the brightness of the LED.

**`button_timer_callback`**: The 500 ms timer callback calculates the next brightness value for the LED and calls `button_set_level` to send a message to the server to adjust the brightness of the LED.

**`button_set_level`**: This function sends a level message to the server using `wiced_bt_mesh_model_level_client_set`.

## 7C.2.4 (Advanced) Mesh Demo Temperature Sensor

This application demonstrates a sensor server. In this case, the sensor measures temperature. It sends status messages to report values on a periodic basis.

### Configuration

The configuration sets up models, properties and features using the variables and structures as described in the previous section. The models, sensors, properties and features implemented are as follows:

| | |
|---|---|
| Models | `WICED_BT_MESH_DEVICE`<br>`WICED_BT_MESH_MODEL_SENSOR_SERVER` |
| Sensors | `WICED_BT_MESH_PROPERTY_PRESENT_AMBIENT_TEMPERATURE` |
| Properties | `N/A` |
| Features | `WICED_BT_MESH_CORE_FEATURE_BIT_LOW_POWER`<br>(if enabled in application settings) |

### User Application Functionality

The top-level file for the user application is *sensor_temperature.c*.

Four of the mesh callback functions are registered for this application:

- Application initialization (`mesh_app_init`)
- Notify period set (`mesh_app_notify_period_set`).
- Low power node sleep (`mesh_app_lpn_sleep`)
- Factory reset (`mesh_app_factory_reset`)

**`mesh_app_init`**: The application initialization function:

- Sets up a pointer to the sensor configuration structure
- Sets the device name, appearance, and FW version
- If the device isn't provisioned yet, it starts advertising and returns.
- If the device is provisioned, it:
    - Calls `thermistor_init` to initialize the ADC
    - Reads the temperature using `mesh_sensor_get_temperature_8`
    - Initializes a timer and registers the callback function `mesh_sensor_publish_timer_callback`
    - Reads the stored sensor cadence value from NVRAM
    - Initializes the sensor server and registers two callback functions – one for the sensor report handler and one for the configuration change handler
        - `mesh_sensor_server_report_handler`
        - `mesh_sensor_server_config_change_handler`
    - Sends the initial temperature value by calling `wiced_bt_mesh_model_sensor_server_data`

**`mesh_sensor_publish_timer_callback`**: The timer callback function is used to send periodic temperature updates. It uses the cadence value set by the client to determine how often to send an update.

**`mesh_sensor_server_report_handler`**: This is the sensor server callback function for report messages. It only implements the case `WICED_BT_MESH_SENSOR_GET`. For that case, it gets the temperature value using `mesh_sensor_get_temperature_8` and sends it to the client using `wiced_bt_mesh_model_sensor_server_data`.

**`mesh_sensor_server_config_change_handler`**: This is the sensor server callback function for configuration change messages. It implements two cases:

> `WICED_BT_MESH_SENSOR_CADENCE_SET`**:** This event occurs when the client requests a change to the sensor cadence. For this case, the user application calls `mesh_sensor_server_process_cadence_changed`. That function updates the cadence, stores its value in NVRAM, and then restarts the publish timer so that cadence changes take effect.

> `WICED_BT_MESH_SENSOR_SETTING_SET`**:** This event occurs when the client requests a change to the sensor settings. For this case, the user application calls `mesh_sensor_server_process_setting_changed` which just prints a UART message that the change has been received.

**`mesh_app_notify_period_set`**: This function is called when the client requests a notify period change. The function first makes sure that the request is for the correct element (`MESH_TEMPERATURE_SENSOR_INDEX`), company ID (`MESH_COMPANY_ID_BT_SIG`), and model (`WICED_BT_MESH_CORE_MODEL_ID_SENSOR_SRV`). If it is, the publish period is updated and the timer is restarted.

**`mesh_app_lpn_sleep`**: This function calls `wiced_sleep_enter_hid_off` with the timeout period that is provided by the mesh stack so that the device wakes up at the appropriate time.

**`mesh_app_factory_reset`**: This function deletes the cadence information from NVRAM before the factory reset occurs.

## 7C.3 (Advanced) OTA/DFU

The mesh library GATT definition includes the BLE OTA (over-the-air) upgrade service by default. Therefore, it is possible to update the firmware over BLE on an unprovisioned mesh device. However, the 20819 does not have enough on-chip flash to hold two copies of the application so an external flash must be used to support OTA.

In addition, code to support the `MESH_DFU_SUPPORTED` option is included in the Mesh Demo On-Off Server snip application (it is disabled by default). This snip demonstrates how to support DFU (device firmware update) over a mesh network.

## 7C.4 (Advanced) Apple HomeKit

Apple HomeKit uses a completely different protocol. It is not based on Bluetooth Mesh but rather on a proprietary protocol that uses WiFi and BLE. Our mesh library will allow devices that support Bluetooth Mesh to also support HomeKit in the following ways:

1. Now: Allow a single device to advertise as both an unprovisioned mesh device and an unpaired HomeKit device. The user can then choose to use it as a Bluetooth Mesh device or a HomeKit device but not both at the same time. This feature can be enabled by defining `MESH_HOMKIT_COMBO_APP`.
2. Future: Allow a single device to be controlled simultaneously by Mesh clients and HomeKit clients.

Note that HomeKit libraries are not available through general SDK due to licensing restrictions. They can be provided to companies that prove that they have an MFi license.

## 7C.5 Exercises

### Exercise 7C.1 Add an On-Off Switch to your Network

In this exercise you will add an on/off switch to your mesh network that can control LEDs on the light_dimmable kit(s).

1. Remove one of your **Dimmable Light** kits from your mesh network using the Android or iOS app.
2. Create an application using the Mesh Demo On-Off Switch template application.

   Name it **ch07c_ex01_on_off**.
3. Open the file *on_off_switch.c* and find the `device_name`.

   Change the name so that it has your initials in it (e.g. "<Inits> Switch").

   **Hint** If you don't see this file in the project, right click and select **Refresh**.
4. Program the application into the mesh kit.

   **Hint** You may want to label the kits to keep track of which one is programmed with each application. Remember if you accidentally hold down the user button on a Dimmable Light kit for 5 seconds, it will perform a factory reset so that kit will need to be re-provisioned.
5. Provision the kit with the On-Off Switch application to your network.
6. Press the user button on the On-Off Switch kit to toggle the LEDs on the Dimmable Light kits.
7. Experiment with changing the Assignment for the On-Off Switch to control different lights or rooms.

   **Hint** You can't move switches to different rooms, rather you Assign the device or rooms that the switch will control.
8. Note that you can still control the lights using the app.

### Exercise 7C.2 Add a Dimmer to the Network

In this exercise you will add a Dimmer to your mesh network. This new device will be able to turn the LED on/off as well as control the brightness of the LED on a Dimmable Light kit. The On-Off Switch from the previous exercise will be able to control the same LED.

1. Remove one of your Dimmable Light kits from your mesh network using the Android or iOS app.

☐ 2. Create an application using the Mesh Demo Dimmer template application.

Name it **ch07c_ex02_dimmer**.

☐ 3. Open the file *dimmer.c* and find the `device_name`.

Change the name so that it has your initials in it (e.g. "<Inits> Dimmer").

☐ 4. Program the application into the kit.

**Hint** You may want to label the kits to keep track of which one is programmed with each application. Remember if you accidentally press the user button on the Dimmable Light kit, it may perform a factory reset.

☐ 5. Provision the Dimmer kit to your network.

☐ 6. Press the user button on the Dimmer kit to toggle the LEDs on the Dimmable Light kits.

☐ 7. Press and hold the user button on the Dimmer kit to adjust the brightness of the LEDs.

**Hint** If you hold the button for longer than 15 seconds a factory reset will be performed and the Dimmer kit will no longer be associated with the mesh network.

☐ 8. Verify that the On-Off Switch kit and the app can still control the LED.

☐ 9. Experiment with Assigning the Dimmer kit to different lights or rooms.

## Exercise 7C.3 (Advanced) Add a 2nd Element for the Green LED to light_dimmable

In this exercise, you will add a new element to the Dimmable Light application so that you can control the Red and Green LEDs on the kit individually. To do this:

☐ 1. Use the app to remove one of the Dimmable Light kits from your network.

☐ 2. Create a new application using the Dimmable Light template application.

Name it **ch07c_ex03_red_green**.

☐ 3. In the *light_dimmable.c* file:

a. Add another element to the design. This new element will have one `WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER` model and no properties.

- **Hint** You will need to create the `mesh_element2_models` array.
- **Hint** you will need to add a set of entries to the `mesh_elements` array for the new element.
- **Hint** Make sure you set the number of models and number of properties in the `mesh_elements` array to the correct values for this new element.

b. In the `mesh_app_init` function, initialize the new light lightness server.

- **Hint** You can use the same callback for both light lightness servers since it is passed the element index when it is called.

☐ 4. In *led_control.c/led_control.h*:

a. Add a define for another PWM channel and add code to `led_control_init` to initialize the new PWM.

- **Hint** The configurator was not used in the Dimmable Light demo application so for consistency you can set up the PWM the same way.
- **Hint** Copy the code for PWM0 and update it to use PWM1.
- **Hint** the BSP has a `#define` for `LED_GREEN` that you can use.
- **Hint** you can use the same `pwm_config` structure for both PWMs – just call `wiced_hal_gpio_select_function` and `wiced_hal_pwm_start` two times each – once for each PWM.

b. Add an additional parameter to the function `led_control_set_brighness_level` so that it knows which element a message is intended for.

- **Hint** `unit8_t element_idx`.
- **Hint** remember to update the function prototype in *led_control.h* too.

c. Update the `led_control_set_brighness_level` function so that it looks at the `element_idx` input and updates the appropriate PWM.

d. Update the `led_control_set_onoff` function in the same way as `led_control_set_brighness_level`.

☐ 5. Back in *light_dimmable.c*:

a. Change the variable `last_known_brightness` to a 2-entry array so that you can store values for both LEDs. Add the index of `[elemnt_idx]` everywhere that variable is used in the code.

b. Search for calls to `led_control_set_brighness_level` (yes, the 't' is missing in brightness) and add the `element_idx` parameter.

- **Hint** The timer callback function `attention_timer_cb` doesn't have access to `element_idx`, but when you initialize the timer you can set it to pass a `uint32_t` to the callback. Therefore, you can:
  1. Set up a global `uint32_t` to hold the index value.
  2. Pass that variable as an argument when you initialize the attention timer
  3. Update the value of the variable with the `element_idx` value just before starting the attention timer.

4. Inside the callback, use the passed argument instead of `element_idx`.

- **Hint** If you don't want to deal with the above, you can just hard code the `element_idx` to 0 in `mesh_app_attention` and `attention_timer_callback`.

☐ 6. Program your kit.

☐ 7. Provision your device onto your network.

**Hint** Once it is provisioned, you should see 2 devices show up for this kit instead of just one – one for each element.

☐ 8. Control each of the LEDs from the app individually using the two separate devices

Note that the On-Off Switch and Dimmer control both LEDs simultaneously because they control everything in the group at once.

## Exercise 7C.4 (Advanced) Update light_dimmable to use the HSL Model

In this exercise, we will change the light lightness model to the Light HSL model. This model extends the light lightness model by adding the ability to control Hue and Saturation. We will use all 3 LEDs in the RGB LED for this exercise.

☐ 1. Use the app to remove one Dimmable Light device from your network.

☐ 2. Create a new application using the Dimmable Light template application.

Name it **ch07c_ex04_rgb**.

☐ 3. In *light_dimmable.c*:

a. Remove the `mesh_app_attention` callback functionality to simplify the changes required.

b. Change the model from `WICED_BT_MESH_MODEL_LIGHT_LIGHTNESS_SERVER` to `WICED_BT_MESH_MODEL_LIGHT_HSL_SERVER`.

c. Find all instances of `mesh_light_lightness` and replace them with `mesh_light_hsl` (look for both lower and uppercase and maintain the case).

d. Add 2 additional elements, each containing one model. The required models are:

i. `WICED_BT_MESH_MODEL_LIGHT_HSL_HUE_SERVER`

ii. `WICED_BT_MESH_MODEL_LIGHT_HSL_SATURATION_SERVER`

e. Change the function `mesh_app_process_set_level` to `mesh_app_process_set_hsl` and make the changes to get the Hue, Saturation and Lightness values from `p_status`.

   **Hint** the data provided in `p_status` will be of type `wiced_bt_mesh_light_hsl_status_data_t`. rather than `wiced_bt_mesh_light_hsl_status_t`.

   **Hint** Use **Open Declaration** on that datatype to find out what it contains. If that doesn't work, the definition can be found in *wiced_btsdk/dev-kit/btsdk-include/<version>/wiced_bt_mesh_models.h*.

   **Hint** Change the calls to `led_control_set_brighness_level` to pass the Hue, Saturation, and Lightness instead of `last_known_brightness`.

   **Hint** Hue, Saturation and Lightness are of type `unit16_t`.

f. Change the call to `led_control_init` to use `LED_CONTROL_TYPE_COLOR`.

☐ 4. In *led_control.c*, set up two additional PWMs.

One for the Green LED and one for the Blue LED.

**Hint** Use PWM1 and PWM2.

**Hint** the BSP has `#defines` for `LED_GREEN` and `LED_BLUE` that you can use.

a. Initialize the PWMs in the `LED_CONTROL_TYPE_COLOR` section of the `led_control_init` function.

b. Update `led_congtrol_set_brightness_level` to set all three PWMs. Update the parameters so that it is passed hue, saturation, and lightness values. You will need to convert the HSL values to RGB values using the function provided in the next step.

c. Use the function shown below to convert the HSL values to RGB values (In future releases of the SDK, this may be provided as a middleware library).

   **Hint** The HSL inputs can be passed in directly from what the model provides. The outputs are provided as pointers to three `uint8_t` variables (r, g, and b).

```
/* Convert HSL values to RGB values */
/* Inputs range 0-65535 */
/* Output range: 0-100 */
void HSL_to_RGB(uint16_t hue, uint16_t sat, uint16_t light, uint8_t* r, uint8_t* g,
uint8_t* b)
{
    uint16_t v;
    /* Formulas expect input in the range of 0-255 so we need to convert
       the input range which is 0-65535. 65535/255 = 257 */
    hue = hue/257;
    sat = sat/257;
    light = light/257;
    v = (light < 128) ? (light * (256 + sat)) >> 8 :
          (((light + sat) << 8) - light * sat) >> 8;
    if (v <= 0) {
        *r = *g = *b = 0;
    } else {
        int m;
        int sextant;
        int fract, vsf, mid1, mid2;
        m = light + light - v;
        hue *= 6;
        sextant = hue >> 8;
        fract = hue - (sextant << 8);
        vsf = v * fract * (v - m) / v >> 8;
        mid1 = m + vsf;
        mid2 = v - vsf;
        // Convert output range of 0-255 to 0-100
        v = (v*100)/255;
        m = (m*100)/255;
        mid1 = (mid1*100)/255;
        mid2 = (mid2*100)/255;
        switch (sextant) {
           case 0: *r = v; *g = mid1; *b = m; break;
           case 1: *r = mid2; *g = v; *b = m; break;
           case 2: *r = m; *g = v; *b = mid1; break;
           case 3: *r = m; *g = mid2; *b = v; break;
           case 4: *r = mid1; *g = m; *b = v; break;
           case 5: *r = v; *g = m; *b = mid2; break;
        }
    }
}
```

☐ 5. Program your kit.

☐ 6. Provision your device onto your network.

☐ 7. Use the app to adjust the light color and intensity.

Also note that if you turn the light on/off using an on/off switch (either in a kit or in the app), the device remembers the last value so that when you turn it back on the color and brightness are the same.