BLE Mesh
BL654 Sample smartBASIC Application

Application Note v2.1.1 (Build 10 or newer) - rel 3

1 INTRODUCTION

In July of 2017, the Bluetooth SIG released Mesh Profile Specification v1.0 describing a Mesh Profile running on top of any BLE device which is v4.0 or newer.

The following are the goals of this document:

▪ Provide an overview of BLE mesh from an application perspective by introducing you to some early beta BLE mesh functionality in the Laird BL654 module
▪ Demonstrate how to utilize it in a sample smartBASIC application by testing the functionality over the UART using a light switch client and server example.

Given the smartBASIC implementation, the mesh explanation is in the context of the event-driven smartBASIC programming paradigm.

The mesh functionality described in this application note is for testing and demonstrative purpose only. It is not fit for production as it is built using the v2.1.1 release of the BLE mesh SDK from Nordic Semiconductor. This release allows provisioning of devices using either an iOS or Android device using an application from Nordic called nRF Mesh.

As this is based on the v2.1.1 release of the SDK from Nordic, Laird reserves the right to change the specifics of the API that is used to expose the SDK API and is described in this application note. Potential changes will be in line with any changes that Nordic may introduce as they continue to work on the stack towards an eventual feature-rich production release which they intend to have approved by the Bluetooth SIG. This implies that Laird will inherit that certification and pass it along to customers who use a Laird module that incorporates mesh.

2 REQUIREMENTS

To use this sample application, you need the following:

▪ DVK-BL654 (development kits) – Minimum of three; ideally a total of five to view the interaction with multiple on/off servers
  One dev kit is used as a sniffer for mesh adverts to get a better understanding of mesh operation because it shows reassuring activity (such as unprovisioned beacons). The sniffer device can be a BL652 devkit because the smartBASIC application runs as-is on both platforms.
▪ PC with spare USB ports (using a USB hub, if appropriate)
▪ UwTerminalX – available for Windows, Linux and Mac: https://github.com/LairdCP/UwTerminalX/releases
▪ Engineering mesh-capable firmware for the BL654
  The response to the AT command “AT I 3” will have the word ‘MESH211’.
▪ Sample command manager smartBASIC applications demonstrating mesh functionality called $autorun$.mesh.light.switch.proxy.client.sb and $autorun$.mesh.light.switch.proxy.server.sb which are included in the firmware zip file.
▪ A MeshSniff smartBASIC application called $autorun$.mesh.sniff.sb which is included in the firmware zip file. This is used to sniff and display mesh-related advert packets.
▪ An iOS or Android smartphone with the latest OS and the latest application called nRF Mesh installed from either Apple Store or Google Play respectively.

Note: For the purposes of this document, we assume you have familiarised yourself with compiling/loading smartBASIC applications
The switches and jumpers on the BL654 devkits shall be configured as per the photograph below.

![BL654 development kit](image)

**Figure 1: BL654 development kit**

**WARNING:** If the four jumpers on the bottom right of the board are missing then the four LEDs that are below them will not operate. This mesh demo relies on showing those LED being switched on and off.

### 3 Release Specific Notes

This application note describes mesh functionality exposed via Laird's smartBASIC programming language implemented on top of Nordic Semiconductor’s Mesh SDK which is at release level 2.1.1. This release is not fully functional and offers the advert bearer and relay functionality and enough aspects of Proxy provisioning and Proxy relaying. There is no ability to offer Friend or Low Power Node capabilities.

Please refer to Nordic Semiconductor’s release notes for more details about the 2.1.1 SDK.

### 4 Demo Description

The examples demonstrated in this application note are light switch client and server devices with provisioning of all devices using an iOS or Android smartphone.

The client device implements the client behavior of a Nordic Semiconductor custom Light Switch model and likewise the server device implements the server behavior of the Light Switch model.

In the overview section, a model is described as an array of opcode and associated handlers. The Nordic custom Light Switch model consists of the following opcodes and the recipient for each opcode.
Table 1: Nordic custom light switch model opcodes and opcode recipients

<table>
<thead>
<tr>
<th>Opcode Name</th>
<th>Opcode</th>
<th>Role</th>
<th>Msg Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLE_ON_OFF_OPCODE_SET</td>
<td>0xC10059</td>
<td>Server</td>
<td>0/1</td>
</tr>
<tr>
<td>SIMPL__ON_OFF_OPCODE_GET</td>
<td>0xC20059</td>
<td>Server</td>
<td>tid</td>
</tr>
<tr>
<td>SIMPLE_ON_OFF_OPCODE_SET_UNRELIABLE</td>
<td>0xC30059</td>
<td>Server</td>
<td>0/1</td>
</tr>
<tr>
<td>SIMPLE_ON_OFF_OPCODE_STATUS</td>
<td>0xC40059</td>
<td>Client</td>
<td>0/1</td>
</tr>
</tbody>
</table>

* 0059 – Nordic Semiconductor Company ID

tid – Transaction ID

The tid data field is just an incrementing number which wraps at 0xFF to 0x00; Nordic did not attach any specific meaning to it. It is incremented each time a SET or SET_UNRELIABLE message is sent.

The example demonstrated in this appnote shows provisioning of up to three light switch servers and a single light switch client.

The example is best described using the following figure (Figure 2).

![Figure 2: Demo client/server structure](image)

**Figure 2: Demo client/server structure**

Figure 2 shows four devkits labelled Client, Server 0, Server 1, and Server 2. Each server contains a single element implementing Nordic’s custom Light Switch Model server roles and the client contains four elements each containing a single Light Switch Model client.

When the client devkit is powered up, it registers four elements, each containing the same model (Nordic Light Switch Client Model, Model ID=0x00590001). It starts to advertise an unprovisioned beacon and GATT Mesh Provisioning Service advert and contains the same device UUID in both which always remains the same.

When an unprovisioned server is powered up, it registers a single element with a single model (Nordic Light Switch Server Model, Model ID=0x00590000). It starts to advertise an unprovisioned beacon and GATT Mesh Provisioning Service advert and contains the same device UUID in both which always remains the same.

The app nRF Mesh on an iOS or Android device is used to provision and configure the client and all servers.

The client’s first model is provisioned so that it has a node address of 0x0004, publish address of 0x0001, and subscribes to group address 0xC001 (all addresses in range 0xC000 to 0xFF00 are group addresses). The second model is provisioned so that it has a node address of 0x0005, publish address of 0x0002, and subscribes to the same group address 0xC001 as the first model. The third model is provisioned so that it has a node address of 0x0006, publish address of 0x0003, and subscribes to the same group address 0xC001 as the first model. Finally, the fourth model is provisioned so that it publishes to group address 0xC000 (to which all servers will subscribe) and does not subscribe to any address.
- Server 0 is provisioned so that it has a node address of 0x0001, publish address of 0xC001, and subscribes to group address 0xC000 (which is the publish address of the fourth model in the client).
- Server 1 is provisioned so that it has a node address of 0x0002, publish address of 0xC002, and subscribes to group address 0xC000 (which is the publish address of the fourth model in the client).
- Server 2 is provisioned so that it has a node address of 0x0003, publish address of 0xC003, and subscribes to group address 0xC000 (which is the publish address of the fourth model in the client).

The provisioning means first model in client controls server 0, second model in client controls server 1, third model in client controls server 2, and finally fourth model in client controls all three servers because they subscribe to its publish address.

At any time, if the client wants to set the on/off state of a server, it publishes a message with opcode SIMPLE_ON_OFF_OPCODE_SET or SIMPLE_ON_OFF_OPCODE_SET_UNRELIABLE. The former results in a response message with opcode SIMPLE_ON_OFF_OPCODE_STATUS but the latter does not.

This means that when a SIMPLE_ON_OFF_OPCODE_SET is sent, the client sees two SIMPLE_ON_OFF_OPCODE_STATUS messages. One as a direct response to the message and the other as a result of the server publishing the new state. This means that SIMPLE_ON_OFF_OPCODE_SET_UNRELIABLE results in a single response arising from the publish.

The smartBASIC sample app $autorun$.mesh.light.switch.proxy.client.sb is an application that implements the client device and is programmed so that the Laird devkit BUTTON1 controls the LED on server 0, BUTTON2 controls the LED on server 1, BUTTON3 controls the LED on server 2, and BUTTON4 controls the LED on all servers at the same time.

The smartBASIC sample app $autorun$.mesh.light.switch.proxy.server.sb is an application that implements the server device and is programmed so that the Laird devkit BUTTON1 locally toggles the state of LED and also publishes the state of the LED so that all subscribers are informed of the local state.

5 BL654 Development Kit Firmware Load

To set up each development kit with the engineering mesh firmware, locate the mesh firmware zip file, unzip into a folder, and follow these steps for all of the dev kits:

1. Connect your BL654 development kit to your PC via the USB Micro cable. The power LED illuminates when the board is receiving power.
2. Open UwTerminalX.
3. In the Config tab, set the parameters and COM port associated with your development board.

<table>
<thead>
<tr>
<th>Port Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device: BL652</td>
</tr>
<tr>
<td>Refresh: Auto</td>
</tr>
<tr>
<td>Port: COM3</td>
</tr>
<tr>
<td>Baudrate: 115200</td>
</tr>
<tr>
<td>Parity: None</td>
</tr>
<tr>
<td>Stop Bits: 1</td>
</tr>
<tr>
<td>Data Bits: 8</td>
</tr>
<tr>
<td>Handshaking: CTS/RTS</td>
</tr>
</tbody>
</table>

4. Click OK to advance to the Terminal tab.
5. Use UwTerminalX to return the BL654 to factory defaults using the command `at&f*` as shown in Figure 3.

If you are using a new development board with the sample application, you may need to remove the autorun jumper on J12 and press the reset button to exit out of the sample application and then issue the `at&f*` command to erase the file system and all non-volatile data.
6. Close UwTerminalX.
7. In the folder where the Mesh firmware was unzipped, locate the `_DownloadFirmwareUart.bat` file and launch it.

8. In the COM field, enter the same comport number as was used in step 3 and confirm that the message `COM port is invalid. should be between 1 and 255` disappeared.
9. Click `OK`.

10. Click `Proceed`.
11. When the upgrade is complete, click `Quit`. 
Open UwTerminalX.

13. In the Config tab, set the parameters and COM port associated with your development board.

14. Click OK to advance to the Terminal tab.

15. Send the command AT I 3 and confirm the following response, where nn is 10 or higher:

10 3 29.1.1.0-MESH211-nn

6  BL654 DEVELOPMENT KIT SMARTBASIC APP LOAD

If you have five boards then label them: Client, Server 0, Server 1, Server 2, and Sniff.

For boards labelled Client, Server 0, Server 1 (optional), and Server 2 (optional), perform the following steps:

1. For the Client board, load the Mesh smartBASIC example application. Use the right-click menu and select XCompile + load.

   ![Figure 6: Select XCompile + Load](image)

   Select the $autorun$.mesh.light.switch.proxy.client.sb file which is located in the MeshApps subfolder.

   It should take approximately ten seconds for the mesh program to load. Run the mesh example by typing at+run "$autorun$" followed by Enter or pressing the reset button.

   ![Figure 7: Running the mesh program](image)
For the board labelled Sniff, perform the following steps:

1. Load the Mesh Sniff smartBASIC example application. Use the right-click menu and select XCompile + Load.

   ![Figure 8: Select XCompile + Load](image)

2. Select the $autorun$.mesh.sniff.sb file which is located in the MeshApps subfolder.

   It should take approximately ten seconds for the mesh program to load. Run the sniff example by typing $autorun$ followed by Enter or pressing the reset button.

---

7 LAUNCH AND TEST THE MESH EXAMPLE

7.1 Overview

This section describes a step-by-step guide to creating and provisioning a mesh of up to four devices (we recommend four, but only two are required) implementing Nordic’s Light Switch example as per their SDK, but implemented in Laird’s easy-to-use event driven smartBASIC programming environment.

The devices can be provisioned using either an iOS or an Android application which is developed by Nordic Semiconductor called nRF Mesh. Only step-by-step instructions for provisioning using an iOS device are described in this application note.

**WARNING:**

Before you begin, please take a note of the revision of the BL654 devkit you are using.

The production release has the silkscreen label DVK-BL654-1.0 at the corner where the DC jack CON1 is located.

You may have the following earlier versions: DVK-BL654-A1 or DVL-BL654-B0. These earlier DVKs do not have the correct labels for the four buttons; trial and error is required to locate which button is which.

When you run this example, server DVKs only use BUTTON 1 and the client DVK uses all four buttons.

In each, when the appropriate button is pressed, a debug message prints on the UwTerminalX screen to identify which button was pressed. Experiment to locate the correct button. We then recommend you make a mark on the devkit appropriately for future reference.

---

7.2 Launch and Test Process Initial Steps

To launch and test the mesh example, follow these steps:

1. Connect all boards to your PC.
2. Open as many UwTerminalX instances as there are boards using the comport that your PC exposes for each board.
3. Reset each board via the reset button on the devkit. Confirm that you see the following messages on the client board (Figure 9), server boards (Figure 10), and sniff board (Figure 11).
If necessary, revert the boards into unprovisioned and clean states.

**Note:** If this is the first time you are running this test and the boards are already in a clean, unprovisioned state, you can skip this step.

- If, at any time, you think the boards may have non-volatile mesh information (aren't in an unprovisioned state) or you are uncertain, it is best practice to revert all the boards.
- If you clean one board, clean all the boards.

a. In the UwTerminalX toolbar, untick the DTR checkbox.
b.Tick/untick the BREAK checkbox.
c. This resets the module and ensures that the smartBASIC $autorun$ application does not automatically launch.
   In this mode, the module accepts AT commands.
d. Confirm this by sending AT and seeing a 00 response.
e. Send the AT&F 0x100000 command. This erases all flash sectors used by the Mesh stack. This is interpreted as an unprovisioned state.
f. In UwTerminalX, click Clear.
g. Tick the DTR checkbox.
h. Tick/untick the BREAK checkbox to reset the board.
i. Confirm that the board’s UwTerminalX displays the following:

```
## MESH_STATE : Wait_For_Provisioning
```

**Figure 12: Board is waiting for provisioning**

5. On the Sniff board UwTerminalX screen, tick/untick the BREAK checkbox after ensuring that the DTR checkbox is ticked.

6. Switch off the client board and all the server boards as you will be powering up and provisioning them using a smartphone one by one. This sequential startup is not necessary for the provisioning process, but it helps to make this step-by-step guide unambiguous. As you get more familiar with the nRF Mesh smartphone app, you will be able to ignore this and provision selected devices at will.

The following sections instruct you on actual provisioning.
### 7.3 Setting Up Server 0

To set up Server 0 for provisioning, follow these steps:

1. **Switch on the Server 0 board and observe the following sniff board traffic:**

   ![Sniff board traffic](image)

   **Figure 13: Sniff board traffic**

   This traffic shows that Server 0 started advertising, that it is unprovisioned, and that it can accept provisioning either over adverts (PB-ADV) or via a BLE connection (PB-GATT).

   Note that both advert forms contain the same UUID and OOB data. In your case, the UUID value will be very different as each mesh device has a unique value.

   The following applies to each row:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Field 1</strong></td>
<td>The Bluetooth address of the unprovisioned mesh device</td>
</tr>
<tr>
<td><strong>Field 2</strong></td>
<td>PB GATT([-50]) or PB ADV([-58]) &lt;br&gt;This means this is an unprovisioned mesh beacon and the ([-50]) or ([-58]) is the RSSI value for the beacon/advert that arrived.</td>
</tr>
<tr>
<td><strong>Field 3</strong></td>
<td>(DevUUID=}9F0CE6498091A00B4865DD9386892175 &lt;br&gt;The device UUID that is factory programmed into the device and is always constant for that particular device. This will vary for you.</td>
</tr>
<tr>
<td><strong>Field 4</strong></td>
<td>(OOB=}0000 &lt;br&gt;The out-of-band bit mask which conveys how the authentication phase of the provisioning will take place. The bit mask is reproduced from the spec as follows:</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Other</td>
</tr>
<tr>
<td>1</td>
<td>Electronic/URI</td>
</tr>
<tr>
<td>2</td>
<td>2D machine-readable code</td>
</tr>
<tr>
<td>3</td>
<td>Bar code</td>
</tr>
<tr>
<td>4</td>
<td>Near Field Communication (NFC)</td>
</tr>
<tr>
<td>5</td>
<td>Number</td>
</tr>
<tr>
<td>6</td>
<td>String</td>
</tr>
<tr>
<td>7</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>8</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>9</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>10</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>11</td>
<td>On box</td>
</tr>
<tr>
<td>12</td>
<td>Inside box</td>
</tr>
<tr>
<td>13</td>
<td>On piece of paper</td>
</tr>
<tr>
<td>14</td>
<td>Inside manual</td>
</tr>
<tr>
<td>15</td>
<td>On device</td>
</tr>
<tr>
<td><strong>Field 5</strong></td>
<td>(URIhash=) &lt;br&gt;This is empty and may contain an eight-hex digit hash value of a URL that would be advertised by this mesh device in a normal advert (arranged via the GATT stack). It can also be used to direct the user to a website for installation or product details. See smartBASIC function BleAdvertStart() in the BL654 smartBASIC Extensions Guide for more details.</td>
</tr>
</tbody>
</table>
7.4 Provisioning Server 0 by Phone

7.4.1 iOS

To provision your iOS Server 0, follow these steps:

1. Launch the nRF Mesh application.
2. From the bottom tabs, select Settings.
3. From the Settings page Reset Mesh State row, select/tap Forget Network and confirm in the pop-up dialog box. This erases all states from the smartphone and allows you to start a new mesh network. It also recreates a new random network and creates three new random app keys.
4. Locate the Application Version row and ensure that the version is at least v1.0.2.
5. From the bottom tabs, select Scanner. The scanner screen scans for all devices that are in an unprovisioned state and offers mesh provisioning service via GATT.
6. Select the LS P-Server device to advance to the Node Provision screen.
7. Ensure the following:
   - the Name row shows LS P-Server
   - the Unicast address shows 0x0001
   - there is a value in the Appkey 1 row

   **Note:** You can edit any of these rows by tapping it. This is especially relevant if you want to allocate a node address different from the one displayed.

8. On the top right, tap Identify to view additional information that is pertinent to the provisioning step. From that window, tap Provision from the top right to open the UwTerminalX window for that device. This window displays a BLE connection being established and then some `## MESH STATE` messages that show the progress of the provisioning process.

   When the new Progress row at the bottom of the phone screen reaches 100%, the screen automatically changes to the Network screen.

   **Note:** If the progress gets stuck before it reaches 100%, kill the phone app, power-cycle the module, and start again. Ensure that, in the Node Provision screen, the unicast address to be allocated is still correct.

   On the UwTerminal screen, the following message confirms that the address of the first node is 1 (==0x0001) and that there is one element: `## MESH_STATE : Provisioned <Addr=1 Count=1>`

   The Network screen displays all the nodes in this network. For now, there is only one – LS P-SERVER : 0001 where 0001 is the node address allocated to it.

   It also has one element and three models. In the loaded smartBASIC application (`$autorun.mesh.light.switch.proxy.server.sb`), we only registered one light switch server model in function `ls_server();` the other two are the configuration and health foundation models which all devices inherit by default.

   `LS P-SERVER` corresponds to the `#define DEVICE_NAME` that exists in the loaded smartBASIC app.

9. Tap the LS P-Server box to advance to the Node Configuration screen. This screen shows details of the three models.
10. Tap the Simple OnOff Server row to advance to the Simple OnOff Server window.
11. Tap the APPKEY BINDING row and select Appkey 1. The APPKEY BINDING value now displays Key Bound with index 0000.
12. Because Server 0 publishes to the group address 0xC001 (see Figure 2), do the following:

   a. In the Simple OnOff Server screen, tap to add the value for the PUBLICATION ADDRESS row to advance to the Publication Settings screen. On this screen, you see that the first row has a default publication address of 0xCEEF.
   b. Tap 0xCEEF.
   c. In the pop-up dialog box, enter 0xC001 and tap Set to return to the Publication Settings screen.
   d. On the top right of the screen, tap Apply Publication to accept and return to the previous screen.
13. Because the subscription address is 0xC000 (see Figure 2), do the following:
   a. Tap **Add Subscription Address**.
   b. Enter 0xC000 in the new dialog box.
   c. Tap **Add**.
   d. Confirm that the Simple OnOff Server window lists C001 as a subscription address.
   e. On the top left of the window, tap **Back**.
      In the Node Configuration window, an icon in the third row shows two vertical arrows and a horizontal line. This shows that the model was assigned a publication and a subscription address.

14. On the top left of the window, tap **Network**.

15. On the top right of the Network window, tap **Disconnect** to advance to the appropriate UwTerminalX window and see confirmation of a BLE disconnection.

   The Sniffer UwTerminalX window now displays a new type of adverts called PROXY(NET_ID). This is basically the device now advertising a Mesh Proxy Service for the phone to get back into the network via a GATT connection, if necessary.

### 7.4.2 Android

TBD. Awaiting a refreshed nRF Mesh that has same iOS functionality.

### 7.5 Provisioning Server 1 by Phone

This step is recommended but can be skipped if you have only one server.

Turn on board Server 1 and observe the sniff board traffic that is similar to what you saw in a previous step.

#### 7.5.1 iOS

To provision your iOS Server 1, follow these steps:

1. Launch the `nRF Mesh` application if it’s not still running from the previous step.
2. From the bottom tabs, select **Scanner**. The scanner screen scans for all devices that are in an unprovisioned state and offers mesh provisioning service via GATT.
3. Select the **LS P-Server** device to advance to the Node Provision screen.
4. Ensure the following:
   - the Name row shows **LS P-Server**
   - the Unicast address shows 0x0002

**Note:** You can edit any of these rows by tapping it. Change the unicast address to 0x0002, if necessary.

5. On the top right, tap **Identify** to view additional information that is pertinent to the provisioning step.
6. From that window, tap **Provision** from the top right to open the UwTerminalX window for that device. This window displays a BLE connection being established and then some `## MESH STATE` messages that show the progress of the provisioning process.

   When the new Progress row at the bottom of the phone screen reaches 100%, the screen automatically changes to the Network screen.

   **Note:** If the progress gets stuck before it reaches 100%, kill the phone app, power-cycle the module, and start again. Ensure that, in the Node Provision screen, the unicast address to be allocated is still correct.

The Network screen displays all the nodes in this network. For now, there is only one – **LS P-SERVER : 0002** where 0002 is the node address allocated to it.
It also has one element and three models. In the loaded smartBASIC application
(Sautorn$.mesh.light.switch.proxy.server.sb), we only registered one light switch server model in function ls_server();
the other two are the configuration and health foundation models which all devices inherit by default.

LS P-SERVER corresponds to the #define DEVICE_NAME that exists in the loaded smartBASIC app.

7. Tap the LS P-SERVER : 0002 box to advance to the Node Configuration screen. This screen shows details of the three models.
8. Tap the Simple OnOff Server row to advance to the Simple OnOff Server window.
9. Tap the APPKEY BINDING row and select Appkey 1. The APPKEY BINDING value now displays Key Bound with index 0000.
10. Because Server 1 publishes to the group address 0xC002 (see Figure 2), do the following:
    a. In the Simple OnOff Server screen, tap to add the value for the PUBLICATION ADDRESS row to advance to the Publication Settings screen. On this screen, you see that the first row has a default publication address of 0xCEEF.
    b. Tap 0xCEEF.
    c. In the pop-up dialog box, enter 0xC002 and tap Set to return to the Publication Settings screen.
    d. On the top right of the screen, tap Apply Publication to accept and return to the previous screen.
11. Because the subscription address is 0xC000 (see Figure 2), do the following:
    a. Tap Add Subscription Address.
    b. Enter 0xC000 in the new dialog box.
    c. Tap Add.
    d. Confirm that the Simple OnOff Server window lists C000 as a subscription address.
    e. On the top left of the window, tap Back.
12. On the top left of the window, tap Network.
13. On the top right of the Network window, tap Disconnect to advance to the appropriate UwTerminalX window and see confirmation of a BLE disconnection.

The Sniffer UwTerminalX window now displays more adverts called PROXY(NET_ID) with different MAC addresses. This is basically the device now advertising a Mesh Proxy Service for the phone to get back into the network via a GATT connection, if necessary.

7.5.2 Android
TBD. Awaiting a refreshed nRF Mesh that has same iOS functionality.

7.6 Provisioning Server 2 by Phone
This step is recommended but can be skipped if you have only two servers.

Turn on board Server 2 and observe the sniff board traffic that is similar to what you saw in a previous step.

7.6.1 iOS
To provision your iOS Server 2, follow these steps:
1. Launch the nRF Mesh application if it’s not still running from the previous step.
2. From the bottom tabs, select Scanner. The scanner screen scans for all devices that are in an unprovisioned state and offers mesh provisioning service via GATT.
3. Select the LS P-Server device to advance to the Node Provision screen.
4. Ensure the following:
   ▪ the Name row shows LS P-Server
   ▪ the Unicast address shows 0x0003

Note: You can edit any of these rows by tapping it. Change the unicast address to 0x0003, if necessary.
5. On the top right, tap **Identify** to view additional information that is pertinent to the provisioning step.

6. From that window, tap **Provision** from the top right to open the UwTerminalX window for that device. This window displays a BLE connection being established and then some **#MESH STATE** messages that show the progress of the provisioning process.

When the new Progress row at the bottom of the phone screen reaches 100%, the screen automatically changes to the Network screen.

**Note:** If the progress gets stuck before it reaches 100%, kill the phone app, power-cycle the module, and start again. Ensure that, in the Node Provision screen, the unicast address to be allocated is still correct.

The Network screen displays all the nodes in this network. For now, there is only one – **LS P-SERVER : 0003** where **0003** is the node address allocated to it.

It also has one element and three models. In the loaded **smartBASIC** application ($authorun$.mesh.light.switch.proxy.server.sb), we only registered one light switch server model in function **ls_server();** the other two are the configuration and health foundation models which all devices inherit by default.

**LS P-SERVER** corresponds to the **#define DEVICE_NAME** that exists in the loaded **smartBASIC** app.

14. Touch Tap the LS P-SERVER: 0003 box to advance to the Node Configuration screen. This screen shows details of the three models.

15. Tap the Simple OnOff Server row to advance to the Simple OnOff Server window.

16. Tap the APPKEY BINDING row and select **Appkey 1**. The APPKEY BINDING value now displays **Key Bound** with index **0000**.

17. Because Server 2 publishes to the group address **0xC003** (see Figure 2), do the following:
   a. In the Simple OnOff Server screen, tap to add the value for the PUBLICATION ADDRESS row to advance to the Publication Settings screen. On this screen, you see that the first row has a default publication address of **0xCEEF**.
   b. Tap **0xCEEF**.
   c. In the pop-up dialog box, enter **0xC003** and tap **Set** to return to the Publication Settings screen.
   d. On the top right of the screen, tap **Apply Publication** to accept and return to the previous screen.

18. Because the subscription address is **0xC000** (see Figure 2), do the following:
   a. Tap **Add Subscription Address**.
   b. Enter **0xC000** in the new dialog box.
   c. Tap **Add**.
   d. Confirm that the Simple OnOff Server window lists **C000** as a subscription address.
   e. On the top left of the window, tap **Back**.
      In the Node Configuration window, an icon in the third row shows two vertical arrows and a horizontal line. This shows that the model was assigned a publication and a subscription address.

19. On the top left of the window, tap **Network**.

20. On the top right of the Network window, tap **Disconnect** to advance to the appropriate UwTerminalX window and see confirmation of a BLE disconnection.

The Sniffer UwTerminalX window now displays more adverts called PROXY(NET_ID) with different MAC addresses. This is basically the device now advertising a Mesh Proxy Service for the phone to get back into the network via a GATT connection, if necessary.

**7.6.2 Android**

TBD. Awaiting a refreshed nRF Mesh that has same iOS functionality.
7.7 Provisioning the Client by Phone

Turn on the board Client and observe the sniff board traffic that is similar to what you saw in a previous step.

7.7.1 iOS

To provision your iOS Client, follow these steps:

1. Launch the nRF Mesh application if it’s not still running from the previous step.
2. From the bottom tabs, select Scanner. The scanner screen scans for all devices that are in an unprovisioned state and offers mesh provisioning service via GATT.
3. Select the LS P-CLIENT device to advance to the Node Provision screen.
4. Check the following:
   - the Name row shows LS P-CLIENT
   - the Unicast address now shows 0x0004

   **Note:** You can edit any of these rows by tapping it.

5. Change the Unicast address to 0x0028.
6. On the top right, tap Identify to view additional information that is pertinent to the provisioning step.
7. From that window, tap Provision from the top right to open the UwTerminalX window for that device. This window displays a BLE connection being established and then some ## MESH STATE messages that show the progress of the provisioning process.

   When the new Progress row at the bottom of the phone screen reaches 100%, the screen automatically changes to the Network screen.

   **Note:** If the progress gets stuck before it reaches 100%, kill the phone app, power-cycle the module, and start again. Ensure that, in the Node Provision screen, the unicast address to be allocated is still correct.

The UwTerminal window displays the following message:

## MESH_STATE : Provisioned <Addr=40 Count=4>. This confirms that the address of the first node is 40 (==0x0028) and that there are four elements.

The Network screen displays all the nodes in this network. For now, there is only one – LS P-CLIENT : 0028 where 0028 is the node address allocated to it.

It also has four elements and six models. In the loaded smartBASIC application ($autorun.mesh.light.switch.proxy.client.sb), we only registered four light switch client models in function ls_client(); the other two are the configuration and health foundation models which all devices inherit by default.

The LS P-CLIENT corresponds to the #define DEVICE_NAME that exists in the loaded smartBASIC application.

8. Tap the LS P-CLIENT : 0028 box to advance to the Node Configuration screen. This screen shows details of the six models.
9. Tap the Simple OnOff Server row to advance to the Simple OnOff Server window.
10. Tap the APPKEY BINDING row and select Appkey 1. The APPKEY BINDING value now displays Key Bound with index 0000.
11. Because the client’s first model publishes to the address 0x0001, do the following: (see Figure 2), do the following:
   a. In the Simple OnOff Server screen, tap to add the value for the PUBLICATION ADDRESS row to advance to the Publication Settings screen. On this screen, you see that the first row has a default publication address of 0xCEEF.
   b. Tap 0xCEEF.
   c. In the pop-up dialog box, enter 0x0001 and tap Set to return to the Publication Settings screen.
   d. On the top right of the screen, tap Apply Publication to accept and return to the previous screen.
12. Because the subscription address is $0xC001$(see Figure 2), do the following:
   a. Tap Add Subscription Address.
   b. Enter $0xC001$ in the new dialog box.
   c. Tap Add.
      d. Confirm that the Simple OnOff Client window lists C001 as a subscription address.
      e. On the top left of the window, tap Back.
      f. Tap the second row (Simple OnOff Client) to advance to the Simple OnOff Client window.
      g. Tap the APPKEY BINDING row and select Appkey 1. The APPKEY BINDING value now displays Key Bound with an index 0000.

13. Because the client’s second model publishes to the address $0x0002$ (see Figure 2), do the following:
   a. In the Simple OnOff Client window, tap to add the value for the PUBLICATION ADDRESS row. This reveals the Publication Settings window where the first row has a default publication address of $0xCEEF$.
   b. Tap $0xCEEF$.
   c. In the pop-up dialog box, enter $0x0002$.
   d. Tap Set to return to the Publication Settings window.
   e. On the top right of the window, tap Apply Publication.

14. Because the client’s second model has the subscription address $0xC002$ (see Figure 2), do the following:
   a. Tap Add Subscription Address.
   b. Enter $0xC002$.
   c. Tap Add.
      d. Confirm that the Simple OnOff Client window lists C002 as a subscription address.
      e. On the top left of the window, tap Back.
      f. Tap the third row (Simple OnOff Client) to advance to the SimpleOnOff Client window.
      g. Tap the APPKEY BINDING row and select Appkey 1. This returns you to the previous window and the APPKEY BINDING value now shows Key Bound with index 0000.

15. Because the client’s third model publishes to the address $0x0003$ (see Figure 2), do the following:
   a. In the Simple OnOff Client window, tap to add the value for the PUBLICATION ADDRESS row. This reveals the Publication Settings window where the first row has a default publication address of $0xCEEF$.
   b. Tap $0xCEEF$.
   c. In the pop-up dialog box, enter $0x0003$.
   d. Tap Set to return to the Publication Settings window.
   e. On the top right of the window, tap Apply Publication.

16. Because the client’s third model has the subscription address $0xC003$ (see Figure 2), do the following:
   a. Tap Add Subscription Address.
   b. Enter $0xC003$.
   c. Tap Add.
      d. Confirm that the Simple OnOff Client window lists C003 as a subscription address.
      e. On the top left of the window, tap Back.
      f. Tap the fourth row (Simple OnOff Client) to advance to the SimpleOnOff Client window.
      g. Tap the APPKEY BINDING row and select Appkey 1. This returns you to the previous window and the APPKEY BINDING value now shows Key Bound with index 0000.

17. Because the client’s forth model publishes to the address $0xC000$ (see Figure 2), do the following:
   a. In the Simple OnOff Client window, tap to add the value for the PUBLICATION ADDRESS row. This reveals the Publication Settings window where the first row has a default publication address of $0xCEEF$.
   b. Tap $0xCEEF$.
   c. In the pop-up dialog box, enter $0xC000$.
   d. Tap Set to return to the Publication Settings window.
   e. On the top right of the window, tap Apply Publication.
   f. On the top left of the window, tap Back.

In the Node Configuration window, an icon in the first three Simple OnOff Client model rows shows two vertical arrows and a horizontal line. This shows that the model was assigned a publication and a subscription address.
The fourth has an icon with only an up arrow and the horizontal line which implies that it has a publication address but not a subscription address.

18. On the top left of the window, tap Network.
19. On the top right of the Network window, tap Disconnect to advance to the appropriate UwTerminalX window and see confirmation of a BLE disconnection.

The Sniffer UwTerminalX window now displays more adverts called PROXY(NET_ID) with different MAC addresses. This is basically the device now advertising a Mesh Proxy Service for the phone to get back into the network via a GATT connection, if necessary.

7.7.2 Android

TBD. Awaiting a refreshed nRF Mesh that has same iOS functionality.

7.8 Testing Nordic’s Simple OnOff Server and Client Mesh Model

In all four devkits, locate the four LEDs. They are located near the area where there are two USB sockets close together. There are four two-pin headers with the following labels: J26, J37, J45, and J39. Ensure that all four of these have a jumper which connects the four LEDs to the appropriate GPIO pins on the module.

All four devices are now provisioned so you can test Nordic’s Simple OnOff server and client mesh model. To do this, follow these steps:

1. On the client, press BUTTON 1. Its UwTerminal window displays the following

   ```
   ## BUTTON 1
   ## Sent Mesh Message
   ## SIMPLE_ON_OFF_OPCODE_STATUS
   ## SIMPLE_ON_OFF_OPCODE_STATUS
   ```

   This confirms that BUTTON 1 was pressed and that a mesh message was sent. It also displays two status messages: 1) the direct replay to the SET message (reliable) and 2) the status that was published so that all subscribers are aware of the change.

   LED1 should now be ON.

   The following displays on Server 0’s UwTerminal window:

   ```
   ## SIMPLE_ON_OFF_OPCODE_SET
   ```

   This displays when a message with the opcode SET is received.

   LED 1 should now be ON.

2. On Server 0, press BUTTON 1. It’s UwTerminal window then displays the following:

   ```
   ## BUTTON 1
   ## Sent Mesh Message
   ```

   This confirms that BUTTON 1 was pressed and that a mesh message was sent.

   LED1 on that devkit should now be OFF.

   The following displays on the client’s UwTerminal window:

   ```
   ## SIMPLE_ON_OFF_OPCODE_STATUS
   ```

   This displays when a STATUS message is received.

   LED 1 should now be OFF.
Note the following:

- Pressing BUTTON 2 on the client devkit controls LED1 on Server 1.
- Pressing BUTTON 3 on the client devkit controls LED1 on Server 2.
- Pressing BUTTON 4 on the client devkit controls LED1 on all server devkits and locally LED1 to LED3 follow those states.

### 8 Mesh-related smartBASIC Functions and Events

This section describes the functions and events that were added to this engineering firmware release. Because it is based on v2.1.1 of the Nordic Mesh SDK, Laird reserves the right to change or delete any functions and events listed in this section.

#### 8.1 Mesh-related AT Commands

##### 8.1.1 AT&F 0x100000

This AT command is used to delete all mesh-related flash sectors so that all state information is deleted. This results in which the device reverting to the unprovisioned state and so it begins sending unprovisioned beacons.

### 8.2 Mesh-related Result Codes

Many of the new functions return a result code. There is a lookup feature in UwTerminalX that describes what those failure result codes mean. The new mesh-related result codes for this alpha release are as follows:

<table>
<thead>
<tr>
<th>Result Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_OPCODEID</td>
<td>0x60C0</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_TOO_MANY_MODELS</td>
<td>0x60C1</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_OPCODE_TABLE_FULL</td>
<td>0x60C2</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_MODEL_NOT_ADDED</td>
<td>0x60C3</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_PREV_MODEL_EMPTY</td>
<td>0x60C4</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_PREV_ELEMENT_EMPTY</td>
<td>0x60C5</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_CURRENT_MODEL_EMPTY</td>
<td>0x60C6</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_TOO_MANY_ELEMENTS</td>
<td>0x60C7</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_TABLE EMPTY</td>
<td>0x60C8</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_LAST_MODEL_EMPTY</td>
<td>0x60C9</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_DUPLICATE_OPCODEID</td>
<td>0x60CA</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_MODELHANDLE</td>
<td>0x60CB</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_MODELINDEX</td>
<td>0x60CC</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_PACKED_OPCODE</td>
<td>0x60CD</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_REPLYINFO</td>
<td>0x60CE</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_ALREADY_STARTED</td>
<td>0x60CF</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_CANT_BEPROVISIONER</td>
<td>0x60D0</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_DATALEN</td>
<td>0x60D1</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INVALID_TIMEOUT</td>
<td>0x60D2</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_INV_STATIC_AUTH_DATA</td>
<td>0x60D3</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_MODELS_NOTALLOWED</td>
<td>0x60D4</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_ATT_MTU_TOO_SMALL</td>
<td>0x60D5</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_PROVISIONER_BUSY</td>
<td>0x60D6</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_ATT_MTU_TOO_SMALL</td>
<td>0x60D7</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_NOPENDING_PUB_REL</td>
<td>0x60D8</td>
</tr>
<tr>
<td>UWRESULTCODE_BLE_MESH_NOPENDING_PUB_REL</td>
<td>0x60D9</td>
</tr>
</tbody>
</table>
8.3 Mesh-related Functions

8.3.1 BleMeshSchemaNew

When a mesh is started, it must know the number of elements the device will expose as well as the models and opcodes each of those elements will host. The element/mesh/opcode information can be viewed as a tree structure of information.

This function is used to create a container with a single empty element which has the index 0. It takes a single integer argument – the location value as defined in the specification. That value is conveyed to a provisioner during provisioning that it can display to the user to give context about the element as part of the composition data.

Note: For those familiar with how a USB device works when plugged into a host, it sends configuration data describing itself. The composition data serves a similar function in Mesh provisioning.

BleMeshSchemaNew( nLocation )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x0607 : Location value not in range 0x0000 to 0xFFFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>nLocation byVAL</td>
</tr>
</tbody>
</table>

Specifies the location description as defined in the GATT Bluetooth Namespace Descriptors which can be found here and is a value in the range 0x0000 to 0xFFFF

8.3.2 BleMeshAddSigModel

An element in a device (the default added with BleMeshSchemaNew()) shall have one or more models. Use this function to add a model using a 16-bit SIG adopted identifier to the mesh schema. It is added to the most recently added element. A model in turn contains opcodes; a function detailed later is used to do that.

BleMeshAddSigModel( nModelId, handleModel )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x0607 : nModelId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x60C4 : Previously added model has no opcodes attached</td>
</tr>
<tr>
<td></td>
<td>0x60C1 : Too many models have been defined in total</td>
</tr>
<tr>
<td></td>
<td>0x60CC : handleModel is not recognised as a model handle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>nModelId byVAL</td>
</tr>
</tbody>
</table>

Specifies a value in the range 0x0000 to 0xFFFF which is model ID as adopted by the Bluetooth SIG and described in the specification “Mesh Model Specification”. For example that specification defines 0x1000 as a Generic OnOff Server and 0x1001 as a Generic OnOff Client.

handleModel byREF     |

On Entry if this model is going to be an extension of another earlier added model then it shall be the handle of that model obtained when BleMeshAddSigModel() or BleMeshAddVendorModel() was called, otherwise it shall contain 0.

On Exit, this is an opaque handle value that the smartBASIC app uses to describe a model when an API will interact with a model or when a message arrives, this value will be presented to enable the developer to channel the behaviour accordingly. We recommend that you store it in a global smartBASIC variable.
8.3.3 BleMeshAddVendorModel

An element in a device (the default added with BleMeshSchemaNew()) has one or more models. Use this function to add a Model using a 32-bit vendor identifier to the mesh schema. It is added to the most recent-added element. A model in turn contains opcodes; a function detailed later is used to do that.

BleMeshAddVendorModel( nCompanyId, nModelId, handleModel )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x0607 : nCompanyId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x0608 : nModelId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x60C4 : Previously added model has no opcodes attached</td>
</tr>
<tr>
<td></td>
<td>0x60C1 : Too many models have been defined in total</td>
</tr>
<tr>
<td></td>
<td>0x60CC : handleModel is not recognised as a model handle</td>
</tr>
</tbody>
</table>

**Arguments:**

- **byVAL nCompanyId** AS INTEGER.
  Specifies a value in the range 0x0000 to 0xFFFF which is a company ID. A member of the Bluetooth SIG can request one for free.
  For a full list of company identifiers see here. Where you will see for example, 0x0059 is for Nordic Semiconductor.
  It is VERY important that if you create a new custom model you use your own company ID and not someone else as you risk collision and thus confuse a provisioner.
  Also please note that if you want to interact with a Nordic defined Model then it is perfectly valid to use their company identifier here.

- **byVAL nModelId** AS INTEGER.
  Specifies a value in the range 0x0000 to 0xFFFF which is model ID as adopted by the Bluetooth SIG and described in the specification "Mesh Model Specification".

- **byREF handleModel** AS INTEGER.
  On Entry if this model is going to be an extension of another earlier added model then it shall be the handle of that model obtained when BleMeshAddSigModel() or BleMeshAddVendorModel() was called, otherwise it shall contain 0.
  On Exit, this is an opaque handle value that the smartBASIC app shall use to describe a model when an API will interact with a model or when a message arrives, this value will be presented to enable the developer to channel the behaviour accordingly
  We recommend that you store it in a global smartBASIC variable.

8.3.4 BleMeshAddOpcode

A model in a device will have one or more opcodes for messages registered so that incoming messages can be processed. Use this function to add a packed opcode which is a value in up to 3-bytes long.

**Note:** If this function fails with BLE_MESH_DUPLICATE_OPCODEID (0x60CB) then it implies that your mesh structure is faulty. If you need a duplicate opcode, then you must add another element to the device for it to again be a unique entry. Then, since an element gets its own node address, the node address is used to differentiate which instance of opcode is being referenced.

BleMeshAddOpcode( nPackedOpcode )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
</tbody>
</table>
0x60C3: No models have been added to the current element
0x60C2: Too many opcodes have been added. Limit will be exceeded
0x60CE: nPackedOpcode is invalid
0x60CB: Current element already has this opcode added

**Arguments:**

<table>
<thead>
<tr>
<th>nPackedOpcode</th>
<th>byVAL nPackedOpcode AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For a SIG defined opcode this shall be a value in the range 0x0000 to 0xFFFF.</td>
</tr>
<tr>
<td></td>
<td>For a vendor defined opcode the value shall be 0xPPVVVV where PP is a value in the range 0xC0 to 0xFF and VVVV is the companyID.</td>
</tr>
</tbody>
</table>

### 8.3.5 BleMeshAddElement

Use this function to add another element to the container started with BleMeshSchemaNew() to which is added more instances of models and op-codes. Each element ends up getting a unique node address by the provisioner. As mentioned in the description for BleMeshAddOpcode() a new element is needed if a device will end up with multiple instances of opcodes. The Mesh specification mandates that an element SHALL have only one instance of an opcode.

**BleMeshAddElement( nLocation )**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x0607 : Location value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x60C5 : Previous element empty</td>
</tr>
<tr>
<td></td>
<td>0x60C6 : Current model empty</td>
</tr>
<tr>
<td></td>
<td>0x60C7 : Too many elements. Limit will be exceeded.</td>
</tr>
</tbody>
</table>

**Arguments:**

<table>
<thead>
<tr>
<th>nLocation</th>
<th>byVAL nLocation AS INTEGER.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specifies the location description as defined in the GATT Bluetooth Namespace Descriptors which can be found here and is a value in the range 0x0000 to 0xFFFF</td>
</tr>
</tbody>
</table>

### 8.3.6 BleMeshStart

Once an Element/Model/Opcode tree has been defined using the functions described above, it must be registered with the Mesh stack and started. This function does that and is always done even if the device is provisioned and configured. When the mesh stack starts, it checks if the non-volatile information matches the structure defined in the tree. It will also know how to fork from there. If the non-volatile data is missing or does not match, then it puts the device into unprovisioned state and starts unprovisioned adverts. Otherwise it resumes mesh operation as a full member of a network.

Some of the parameters supplied in this function are used to configure the composition data – the information that is supplied to a provisioner so that it knows more about this device and more of that composition data is configured using the functions BleMeshConfigInt() and BleMeshConfigStr() which are described later.

**BleMeshStart( nCompanyId, nProductId, nVersionId, staticAuth$ )**

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x0608 : nCompanyId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x0609 : nProductId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x060A : nVersionId value not in range 0x0000 to 0xFFFF</td>
</tr>
<tr>
<td></td>
<td>0x060C : nDefaultTTL value not in range 0 to 127</td>
</tr>
<tr>
<td></td>
<td>0x60D0 : The mesh stack has already been started</td>
</tr>
<tr>
<td></td>
<td>0x60C9 : The mesh table tree is not empty</td>
</tr>
<tr>
<td></td>
<td>0x60D1 : This device cannot be a provisioner</td>
</tr>
<tr>
<td></td>
<td>0x60C8 : The mesh table tree is empty</td>
</tr>
<tr>
<td></td>
<td>0x60CA : The last model is empty in the tree</td>
</tr>
</tbody>
</table>
Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCompanyld</td>
<td>byVAL nCompanyld AS INTEGER. Specifies a value in the range 0x0000 to 0xFFFF which is a company ID. A member of the Bluetooth SIG can request one for free. For a full list of company identifiers see <a href="#">here</a>. Where you will see for example, 0x0059 is for Nordic Semiconductor. It is VERY important that you use your own company ID so that a provisioner better understands how to configure your device. Think of this value and the nProductId as the equivalent of the plug and play VID/PID information presented by a USB device.</td>
</tr>
<tr>
<td>nProductId</td>
<td>byVAL nProductId AS INTEGER. Specifies a value in the range 0x0000 to 0xFFFF which is a product ID. This can be any value that you wish as you maintain a list of all the different mesh products that your produce. This is very similar to the PID value in USB world</td>
</tr>
<tr>
<td>nVersionld</td>
<td>byVAL nVersionld AS INTEGER. Specifies a value in the range 0x0000 to 0xFFFF which is a version ID. This can be any value that you wish.</td>
</tr>
<tr>
<td>staticAuth$</td>
<td>byREF staticAuth$ AS STRING. This is a 16 byte string containing a key which will be randomly generated</td>
</tr>
</tbody>
</table>

### 8.3.7 BleMeshConfigInt

This function is used to specify more integer configuration parameters before the mesh functionality is started using the function `BleMeshStart()`.  

**BleMeshConfigInt( nConfigld, nConfigValue )**

Returns

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>resultCode</td>
</tr>
<tr>
<td>0x0000</td>
<td>Success</td>
</tr>
</tbody>
</table>

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nConfigld</td>
<td>byVAL nConfigld AS INTEGER. See 'nConfigValue' for more details.</td>
</tr>
</tbody>
</table>
| nConfigValue | byVAL nConfigValue AS INTEGER. For Configld=0 Default Time to Live which will be a value in the range 2 to 127  
For Configld=1 A bit mask which specifies which mesh features the device will expose as follows:-  
  - Bit 0 : Relay Capability  
  - Bit 1 : Proxy Capability  
  - Bit 2 : Friend Capability  
  - Bit 3 : Low Power Node Capability  
  - Bits 4 to 31 : Reserved for future use and should be set to 0  
For Configld=2 16 bit OOB Info field in unprovisioned beacons  
  - Range : 0..xFFFF (Bit Mask as follows)  
    - Bit Description  
    - 0 Other  
    - 1 Electronic / URI  
    - 2 2D machine-readable code |
3  Bar code
4  Near Field Communication (NFC)
5  Number
6  String
7  Reserved for Future Use
8  Reserved for Future Use
9  Reserved for Future Use
10 Reserved for Future Use
11 On box
12 Inside box
13 On piece of paper
14 Inside manual
15 On device

For ConfigId=3
Output OOB size  (Table 5.21 in Mesh Profile spec)
Range : 0..8  (0==Device does not support output OOB)

For ConfigId=4
Output OOB Action
Range : 0..15

For ConfigId=5
Input OOB size  (Table 5.23 in Mesh Profile spec)
Range : 0..8  (0==Device does not support input OOB)

For ConfigId=6
Input OOB Action
Range : 0..15

8.3.8 BleMeshConfigStr

This function is used to specify string configuration parameters before the mesh functionality is started using the function BleMeshStart().

BleMeshConfigStr( nConfigId, sConfigValue$ )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>nConfigId</td>
</tr>
<tr>
<td>byVAL</td>
</tr>
<tr>
<td>nConfigId</td>
</tr>
<tr>
<td>AS INTEGER.</td>
</tr>
<tr>
<td>See ‘sConfigValue$’ for more details.</td>
</tr>
</tbody>
</table>

| sConfigValue$ |
| byREF        |
| sConfigValue$ |
| AS STRING.   |

For ConfigId=0
Provisioning Beacon Device URI

8.3.9 BleMeshPublish

This function is used to publish a message with the opcode and data specified using the preconfigured publish details (like destination address) of the model specified by the handleModel provided (the handle that was returned when either
BleMeshAddSigModel() or BleMeshAddVendorModel() were called. It uses the appkey and netkey bound to the model identified by ‘handleModel’.

BleMeshPublish( handleModel, nPackedOpcode, sData$ )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x60CC : handleModel is not recognised as a model handle</td>
</tr>
<tr>
<td></td>
<td>0x60CE : nPackedOpcode is invalid</td>
</tr>
<tr>
<td></td>
<td>Other : Nordic stack specific</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>handleModel</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>nPackedOpcode</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>sData$</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

8.3.10 BleMeshPublishReliable

This function is used to publish a reliable message with all the parameters as described for the function BleMeshPublish() but in addition, it takes two more parameters which correspond to the opcode of the message to expect that acknowledges receipt of this method and the maximum time to wait for that ack.

If the function returns a successful resultcode, then a reliable publish transaction shall be assumed to have started. It will terminate when an event message EVBLEMESH_PUBREL_RESULT is received with a status code which conveys whether it was successful or a timeout occurred or cancelled because the function BleMeshPublishRelCancel() had subsequently been called.

While a reliable publish transaction is in progress, this function cannot be called for the same handleModel.

Note: If the model publishes to a group address, then a message is sent but the transaction is deemed to be completed immediately – do not expect to wait for the EVBLEMESH_PUBREL_RESULT event.

BleMeshPublishReliable( handleModel, nPackedOpcode, nExpectedOpcode, nTimeoutsec, sData$ )

<table>
<thead>
<tr>
<th>Returns</th>
<th>INTEGER : resultCode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000 : Success</td>
</tr>
<tr>
<td></td>
<td>0x60CC : handleModel is not recognised as a model handle</td>
</tr>
<tr>
<td></td>
<td>0x60CE : nPackedOpcode is invalid</td>
</tr>
<tr>
<td></td>
<td>Other : Nordic stack specific</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>handleModel</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>nPackedOpcode</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
**8.3.11 BleMeshPublishRelCancel**

This function is used to cancel a reliable publish transaction that was initiated for a model using the function `BleMeshPublishReliable()`.

`BleMeshPublishRelCancel(handleModel)`

Returns  
INTEGER : resultCode  
0x0000 : Success  
0x60CC : handleModel is not recognised as a model handle  
Other : Nordic stack specific

Arguments:

- **handleModel**  
  byVAL handleModel AS INTEGER.  
  This is the handle of a model that was registered using `BleMeshAddSigModel()` or `BleMeshAddVendorModel()`.

**8.3.12 BleMeshReply**

This function is used to send a response to an incoming message with the opcode and data specified using the destination details (like destination address) embedded in the opaque parameter `sReplyData$`. This was supplied when the incoming message arrived via the event `EVBLEMESH_OPC_MSG` which is described later. The `sReplyInfo$` will also contain the appkey that was used by the incoming message and so the response needs to use the same one.

**Note:** handleModel and nPackedOpcode are also supplied in the `EVBLEMESH_OPC_MSG` event when the incoming message arrived

`BleMeshReply(handleModel, nPackedOpcode, sData$, sReplyInfo$)`

Returns  
INTEGER : resultCode  
0x0000 : Success  
0x60CC : handleModel is not recognised as a model handle  
0x60CE : nPackedOpcode is invalid  
0x60CF : sReplyInfo$ is invalid  
Other : nordic stack specific

Arguments:

- **handleModel**  
  byVAL handleModel AS INTEGER.  
  This is the handle of a model that was registered using `BleMeshAddSigModel()` or `BleMeshAddVendorModel()`. The destination address, appkey comes from whatever was configured for the model by a provisioner.

- **nPackedOpcode**  
  byVAL nPackedOpcode AS INTEGER.  
  For a SIG defined opcode this shall be a value in the range 0x0000 to 0xFFFF.
For a vendor defined opcode the value shall be 0xPPVVVV where PP is a value in the range 0xC0 to 0xFF and VVVV is the companyID.

$sData$$ byREF $sData$ AS STRING.
This contains the data that will be sent as payload for the message. The specification allows this to be from 0 to 380-bytes. It will be appropriately lower if the opcode is 3-bytes long.

$sReplyInfo$$ byREF $sReplyInfo$ AS STRING.
This will have been supplied in the EVBLEMSG_OPC_MSG event and MUST be supplied unmodified from there. It is an opaque object and will be checked for modification and if so will result in a failure to send a response.

8.4 Mesh-related Events

8.4.1 EVBLEMESH_STATE
This event occurs when the mesh state of the device changes.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nNewState$ byVAL $nNewState$ AS INTEGER. This contains the new state</td>
</tr>
<tr>
<td>$sContext$$ byREF $sContext$ AS STRING. This contains context data for the new state</td>
</tr>
</tbody>
</table>

The values for $nNewState$ and associated context string will be as per the table below. If the context column is 'none' then the string will be empty.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Context Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>WAIT_FOR_PROVISIONING</td>
<td>none</td>
</tr>
<tr>
<td>150</td>
<td>PROV_PRE_STACK_DISABLE</td>
<td>None (will be received just before the stack is disabled so that the GATT table which contains the Mesh Provisioning Service can be destroyed and on re-enabling a new GATT table is created with the Mesh Proxy Service)</td>
</tr>
<tr>
<td>160</td>
<td>PROV_STACK_ENABLED</td>
<td>none (happens after the stack is restarted with Mesh Proxy Service)</td>
</tr>
<tr>
<td>200</td>
<td>PROVISIONED</td>
<td>First 2-bytes = First Element Node address Second 2-bytes = Number of Elements &lt;&lt;Note: 2-bytes entities are little endian&gt;&gt;</td>
</tr>
<tr>
<td>300</td>
<td>KEY_REFRESH_NOTIFICATION</td>
<td>none</td>
</tr>
<tr>
<td>400</td>
<td>IV_UPDATE_NOTIFICATION</td>
<td>none</td>
</tr>
</tbody>
</table>

8.4.2 EVBLEMESH_OPC_MSG
This event occurs when a message arrives and needs to be processed and may result in zero or more outgoing messages.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nElementIndex$ byVAL $nElementIndex$ AS INTEGER. This contains the element index 0 to N and will correspond to the elements that were added using BleMeshElementAdd()</td>
</tr>
<tr>
<td>$handleModel$ byVAL $handleModel$ AS INTEGER. This contains the handle that was returned by BleMeshAddSigModel() or BleMeshAddVendorModel()</td>
</tr>
<tr>
<td>$nPackedOpcode$ byVAL $nPackedOpcode$ AS INTEGER. This contains the packed opcode. For a SIG defined opcode this shall be a value in the range 0x0000 to 0xFFFF.</td>
</tr>
</tbody>
</table>
For a vendor defined opcode the value shall be 0xPPVVVV where PP is a value in the range 0xC0 to 0xFF and VVVV is the companyID

**sData**

byREF sData$ AS STRING.

This contains the data that arrived in the message associated with the opcode.

**sReplyInfo**

byREF sReplyInfo$ AS STRING.

This contains context data that will be used if BleMeshReply() is called and should be supplied to that function unmodified. This data MUST NOT be modified in anyway by the application when it is supplied in BleMeshReply().

It is expected that the smartBASIC application the handler will switch on the nPackedOpcode value (using the Select statement) and then call an appropriate function to handle the data.

### 8.4.3 EVBLEMESH_PUBREL_RESULT

This event occurs to signal the end of a reliable publish transaction which was initiated using the function BleMeshPublishReliable().

#### Parameters:

- **nElementIndex**
  - byVAL nElementIndex AS INTEGER.
  - This contains the element index 0 to N and will correspond to the elements that were added using BleMeshElementAdd().

- **handleModel**
  - byVAL handleModel AS INTEGER.
  - This contains the handle that was returned by BleMeshAddSigModel() or BleMeshAddVendorModel().

- **nResult**
  - byVAL nResult AS INTEGER.
  - This contains the result outcome as follows:
    - 0 = Transaction successfully completed
    - 1 = Transaction failed due to a timeout
    - 2 = Transaction Aborted due to BleMeshPublishRelCancel() being called

### 9 SMARTBASIC APP CODE WALKTHROUGH

In a smartBASIC BLE Mesh application, the developer only must code for creating the mesh models and then sending and receiving mesh messages; the developer only has to deal with what triggers a mesh message to be sent and what happens when a mesh message is received. With that in mind, smartBASIC provides functions such as BleMeshPublishXXXX() and BleMeshReply() for sending messages; since it is an event-driven language, there is an event EVBLEMESH_OPCODE_MESH that is thrown to the application which in turn has a handler written by the developer to process it.

In terms of which node to send to or where a message comes from, that is entirely dealt with when a device is provisioned and configured by a provisioner like a smartphone. In fact, all the coding of that aspect is completely opaque to the application layer so is not even exposed to the smartBASIC app developer.

With that in mind, the relevant portions of the Client and Server applications are described in the following sections.

#### 9.1 Client : $autorun$.mesh.light.switch.proxy.client.sb

On start, the function GpioInit() on line circa 620 which is defined at line circa 244 where the gpio for all 4 buttons on the devkit are configured as inputs using GpioSetFunc() and all 4 LEDs are configured as outputs, again using GpioSetFunc() and then finally all 4 button gpios are configured to generate the EVGPIOCHANx event so that the handlers HandlerOnButtonx() are called when they change state.

Then the function ls_client() is called (see the second last line of code) which in turn registers 4 mesh elements, each with a single ‘simple onoff client’ model which you see in the ‘for’ loop, and then the mesh stack is started by calling the function BleMeshStart().

For incoming mesh messages, the following statement, towards the end of the file, ensures that the function HandlerMeshOpcMsg() is called.

```c
OnEvent EVBLEMESH_OPCODE_MESH call HandlerMeshOpcMsg
```
Locate the **HandlerMeshOpcMsg()** function and you will see it is passed the element index, the model handle, the opcode of the message and the payload. There you see a SELECT statement to switch on the opcode value and in this case the model only listens for the STATUS message and for this demo we call the function **onStatus()** and print a message that the message has arrived.

Locate the **onStatus()** function where you will see that the data in the message is decoded to extract the state of the LED at the server that sent it and that it calls **GpioWrite()** to reflect the state of the remote LED on one of the local LEDs.

Locate the **HandlerOnButtonX()** (where X= 1 or 2 or 3 or 4) and you will see that a debug message is printed to indicate that BUTTON X has been pressed and then the function **OnButton()** is called. That function is defined at circa line which in turn calls **SendSetMsg()** which in turn creates and sends the SIMPLE_ON_OFF_OPCODE_SET message.

### 9.2 Server : $autorun$.mesh.light.switch.proxy.server.sb

On start, the function **GpioInit()** on line circa 557 which is defined at line circa 235 where the gpio for button 1 on the devkit is configured as input using **GpioSetFunc()** and all 4 LEDs are configured as outputs, again using **GpioSetFunc()** and then finally button 1 gpio is configured to generate the EVGPIOCHAN0 event so that the handlers **HandlerOnButton1()** is called when it changes state.

Then the function **ls_server()** is called (see the second last line of code) defined at line circa 307 which in turn registers 1 mesh element, with a single 'simple onoff server model at line circa 316. Then the three opcodes are registered at lines circa 320, 323 and 326 and then finally the mesh stack is started by calling the function **BleMeshStart()**.

For incoming mesh messages, the following statement, towards the end of the file, ensures that the function **HandlerMeshOpcMsg()** is called.

```
OnEvent EVBEMESH_OPC_MSG call HandlerMeshOpcMsg
```

Locate the **HandlerMeshOpcMsg()** function and you will see it is passed the element index, the model handle, the opcode of the message and the payload. There you see a SELECT statement to switch on the opcode value and in this case the model listens for the GET, SET and SET_UNRELIABLE messages and for this demo we call the functions **OnGet(), OnSetRel()** and **OnSetUnrel()** respectively and also print a message that the message has arrived.

Locate the **OnGet()** function where you will see that the STATUS message is published – and note there is no information about who it publishes to as that is provided at provisioning time.

Locate the **OnSetRel()** function where you will see that STATUS message is sent in a reply (the destination node address is provided in the respInfo$ parameter) and also another STATUS message is published – again to a recipient that will have been configured when the device was provisioned.

Locate the **OnSetUnrel()** function where you will see that STATUS message is published – again to a recipient that will have been configured when the device was provisioned. Note there is no reply to the sender.

Locate the **HandlerOnButton1()** and you will see that a debug message is printed to indicate that BUTTON 1 has been pressed and then the locate variable **ledstate[0]** is toggled and the **GpioWrite()** is used to update the satet of LED1 as per the state of that variable and then finally the new LED state is published using **BleMeshPublish()** so that all subscribers are made aware of that change of state.

### 10 REFERENCES

The following documents are also accessible from the BL654 product page of the Laird website (Documentation tab):

- BL654 smartBASIC Extension Manual
- BL654 Datasheet
- UwTerminalX

The following documents are also accessible from the Bluetooth SIG website:

- Mesh Profile Specification v1.0
- Mesh Model Specification v1.0
- Mesh Device Properties v1.0
# 11 Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Notes</th>
<th>Approver</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10.0/rel10</td>
<td>5 Dec 2017</td>
<td>Initial Release</td>
<td>Jonathan Kaye</td>
</tr>
<tr>
<td>2.1.1/8</td>
<td>17 Jul 2018</td>
<td>Updated to reflect firmware based on Nordic Mesh SDK V2.1.1</td>
<td>Jonathan Kaye</td>
</tr>
<tr>
<td>2.1.1/8-rel 2</td>
<td>16 Aug 2018</td>
<td>All sections updated as per the functionality</td>
<td>Jonathan Kaye</td>
</tr>
<tr>
<td>2.1.1 rel 3</td>
<td>24 Sep 2018</td>
<td>Typo correction</td>
<td>Mahendra Tailor</td>
</tr>
</tbody>
</table>