# Embedded provisioner application for ST Bluetooth® mesh

## Introduction

This application note describes the functionality of the embedded provisioner application for ST Bluetooth® mesh as well as the steps to set up and use it for provisioning, that is, to add unprovisioned devices to a Bluetooth® mesh network.

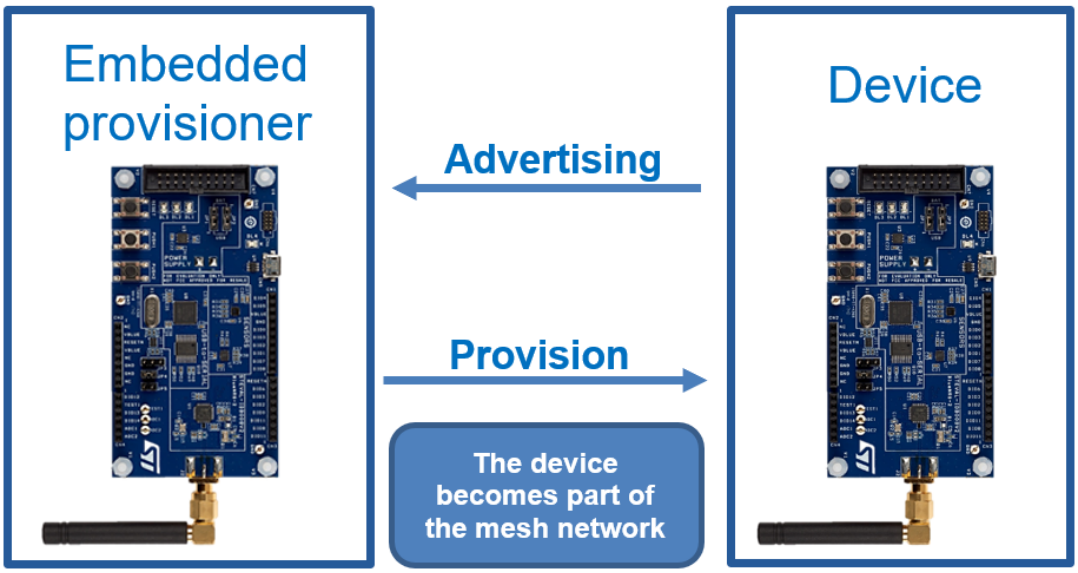
A provisioner initiates the provisioning procedure to create a mesh network. During the provisioning process, the provisioner sends the provisioning data to the unprovisioned device that allows it to become a part of a mesh network. The provisioning data includes the network key, net key index, flags, current IV index, and the node unicast address.

To provision a device, establish the provisioning bearer between the provisioner and a device. The provisioner can identify the device through its UUID.

The Bluetooth® mesh profile specifications allow provisioning by using the PB-GATT protocol and the PB-ADV bearer. Typically, a mobile phone acts as a provisioner using the PB-GATT bearer protocol. The embedded provisioner example described in this document uses the PB-ADV bearer, which is not supported yet on smartphones.

In this embedded provisioner application, which is an approach to provision a Bluetooth® mesh device without using a mobile phone, the Bluetooth® SoC board acts as a provisioner to perform the provisioning process and add unprovisioned devices to the network using the PB-ADV bearer.

**Figure 1. Embedded provisioner application**




The embedded provisioner application:

- creates the mesh network between nodes;
- uses the PB-ADV bearer for communication;
- saves the information of the provisioned node in the flash memory;
- auto-increments to assign a unique unicast address to new nodes.

# 1 System requirements

## 1.1 Hardware requirements

You can use the following boards to evaluate the embedded provisioner application using the ST BlueNRG-Mesh solution.

**Table 1. Hardware requirements**

| Bluetooth® Low Energy device | Evaluation boards | Description |
|---|---|---|
| BlueNRG-2 | STEVAL-IDB008V1/STEVAL-IDB008V2 | Evaluation platform based on the BlueNRG-2 |
| BlueNRG-LP | STEVAL-IDB011V1 | Evaluation platform based on BLUENRG-355MC system-on-chip |

To connect the STEVAL-IDB008V1/STEVAL-IDB008V2 (BlueNRG-2 evaluation board)/STEVAL-IDB011V1 (BlueNRG-LP evaluation board) to a PC, you need a USB port to provide the power supply to the board.

To connect the ST-LINK/V2 debugger, you need an additional USB port. Various LEDs, switches, connectors, and test points are available on the board as shown in the figures below.

**Figure 2. Devices and connectors available on the STEVAL-IDB008V2 evaluation board**

1. BlueNRG-2
2. XTAL = 32 MHz
3. Arduino connector
4. Arduino connector
5. Micro-USB connector for power supply
6. PUSH2 button
7. PUSH1 button
8. RESET button
9. JTAG connector
10. Blue LED (DL3), red LED (DL2), and yellow LED (DL1)
11. Power LED (DL4)

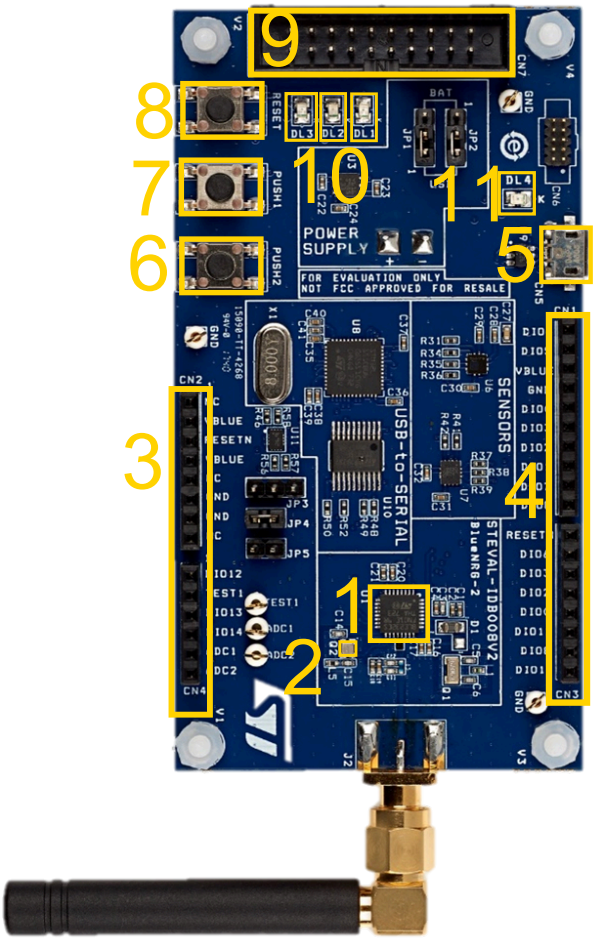
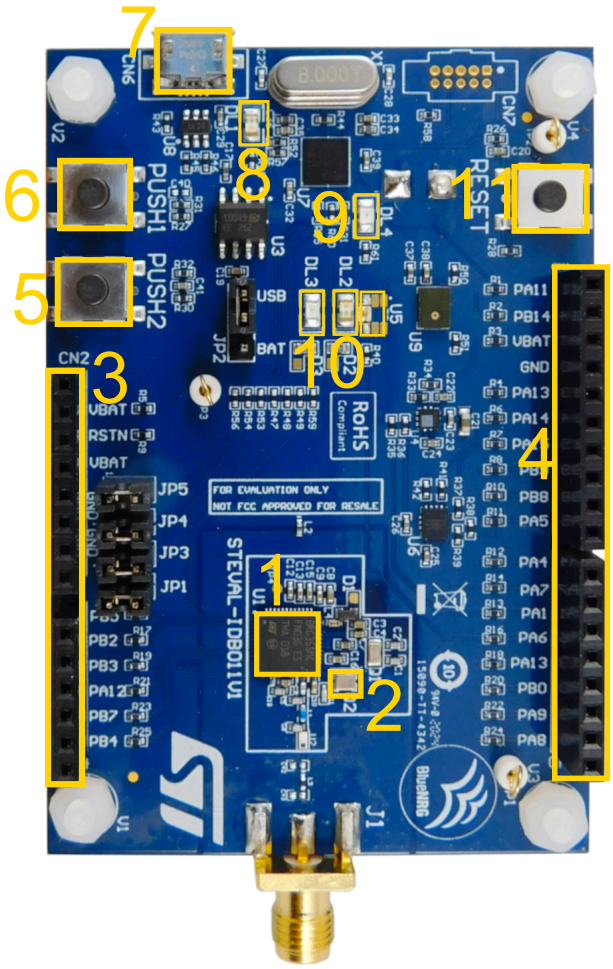


**Figure 3. Devices and connectors available on the STEVAL-IDB011V1 evaluation board**

1. BlueNRG-LP
2. XTAL = 32 MHz
3. Arduino connector
4. Arduino connector
5. PUSH2 button
6. PUSH1 button
7. Micro-USB connector for power supply
8. Power LED (DL1)
9. USB_CMSISDAP red LED (DL4)
10. Red LED (DL3), green LED (DL2), and blue LED (U5)
11. RESET button




*Note: Refer to UM2295 for further details on the board configuration and ST Bluetooth® mesh application.*

## 1.2 PC requirements

The minimum requirements to set up the software environment and run the ST BlueNRG-Mesh provisioner demo application are:

- a PC with an Intel or AMD processor running one of the following Microsoft operating systems: Windows XP/Vista/Windows7/Windows 10
- At least 128 MB of RAM
- Two USB ports
- 40 MB of hard disk space

- Development toolchains and compilers:
  - Keil® µVision v5.32.0.0
  - IAR Embedded Workbench v8.50.5

  Follow the system requirements and setup information provided by the IDE provider.

## 1.3 STSW-BNRG-Mesh and STSW-BNRGLP-Mesh installation

**Step 1.** For the BlueNRG-2, download the software from the STSW-BNRG-Mesh web page.

**Figure 4. Screenshot of the STSW-BNRG-Mesh software available in the web page**


**Step 2.** For the BlueNRG-LP, download the software from the STSW-BNRGLP-Mesh web page.

**Figure 5. Screenshot of the STSW-BNRGLP-Mesh software available in the web page**


**Step 3.** Extract the contents of the package to a temporary directory.

**Step 4.** Launch the installer and follow the instructions.

**Step 5.** Install the software into a suitable folder on your disk drive.

## 1.4 Connecting the boards to a PC

You can connect the boards to a PC via USB connection. You can use any terminal software (HyperTerminal, Hercules, Putty, etc.) to open the serial communication port on the PC to check the messages from the board.

The board controller UART is connected to the PC via a VCOM (virtual communication) port. The settings to open the communication port are the following:

- baud rate: 115200
- data size: 8
- parity: none
- stop bits: 1
- no hardware control

After the board is connected to the PC, open the terminal software. Then, press the board reset button. If the flashed provisioning firmware starts successfully, the following messages appear on the VCOM window showing the node as a "provisioner node".

Figure 6. **Provisioner node VCOM window**

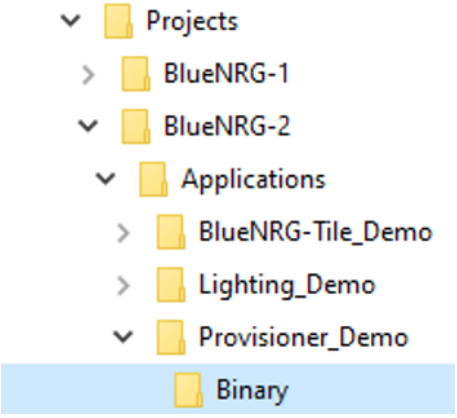
# 2 Using the embedded provisioner application

Precompiled binaries are available to run quickly the embedded provisioner application for the supported boards. If the application is modified, a workspace for both IAR Workbench and Keil® is available.

There are different methods of running the embedded provisioner application for the BlueNRG-2 platform by using:

- the precompiled binaries available in the STSW-BNRG-Mesh software at:
  Projects\BlueNRG-2\Applications\Provisioner_Demo\Binary

**Figure 7. Path for precompiled binaries in the BlueNRG-2 platform**




or

- the editable firmware workspaces for IAR or Keil® available at \Projects\BlueNRG-2\Applications\Provisioner_Demo\EWARM\STEVAL-IDB008Vx or \Projects\BlueNRG-2\Applications\Provisioner_Demo\MDK-ARM\ STEVAL-IDB008Vx folder, respectively

**Figure 8. Path for the IAR project for the BlueNRG-2 platform**

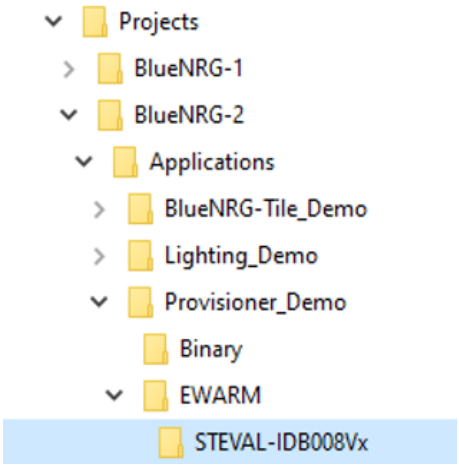
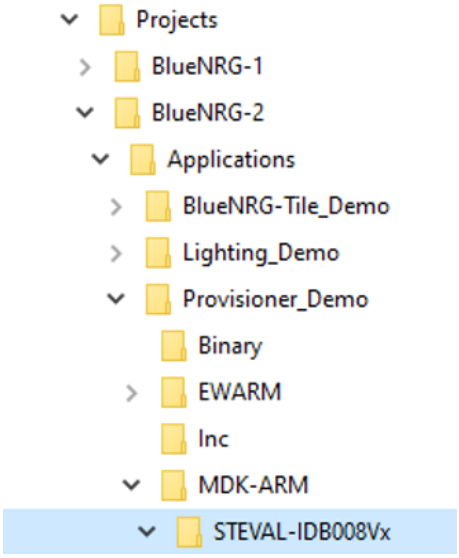



**Figure 9. Path for the Keil project for the BlueNRG-2 platform**




You can also use the embedded provisioner application for the BlueNRG-LP platform using editable firmware workspaces for IAR or Keil® available at \Projects\BlueNRG-LP\Applications\Lighting_Demo\EWARM\STEVAL-IDB0011Vx or \Projects\BlueNRG-LP\Applications\Lighting_Demo\ MDK-ARM \STEVAL-IDB0011Vx, respectively.

Figure 10. **Path for the IAR project for the BlueNRG-LP platform**

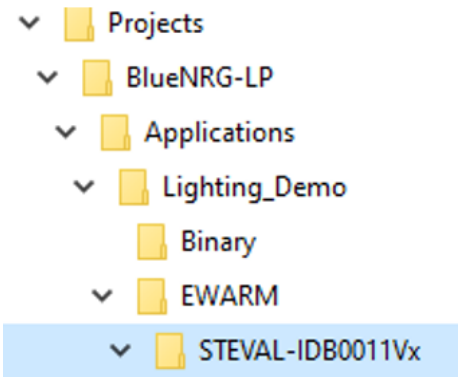
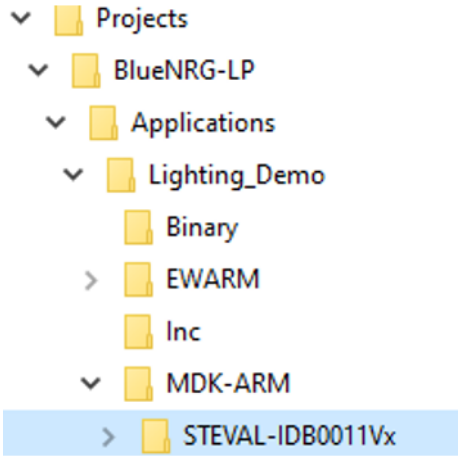



Figure 11. **Path for the Keil project for the BlueNRG-LP platform**

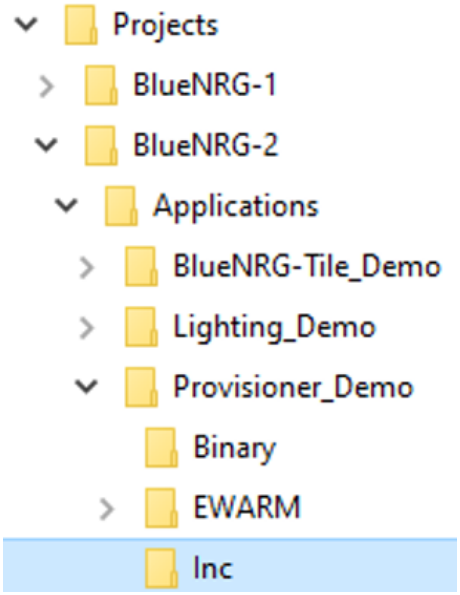


# 3 Firmware initialization and configuration

In the ST Bluetooth® mesh firmware, various macros are available to enable or disable certain features and debugging logs. These macros are available in “mesh_cfg_usr.h” file located at:

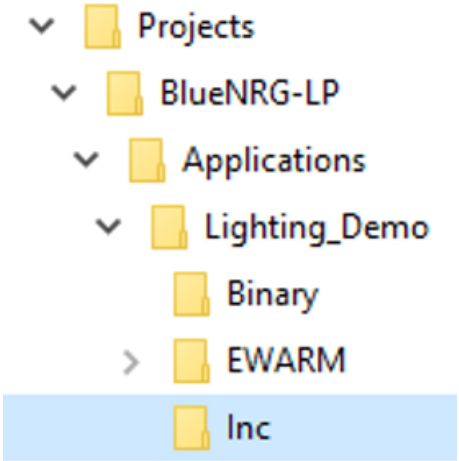
- \Projects\BlueNRG-2\Applications\Provisioner_Demo\Inc folder for the BlueNRG-2 platform

**Figure 12. Path for mesh_cfg_usr.h in the BlueNRG-2 platform**




- \Projects\BlueNRG-LP\Applications\Lighting_Demo for the BlueNRG-LP platform

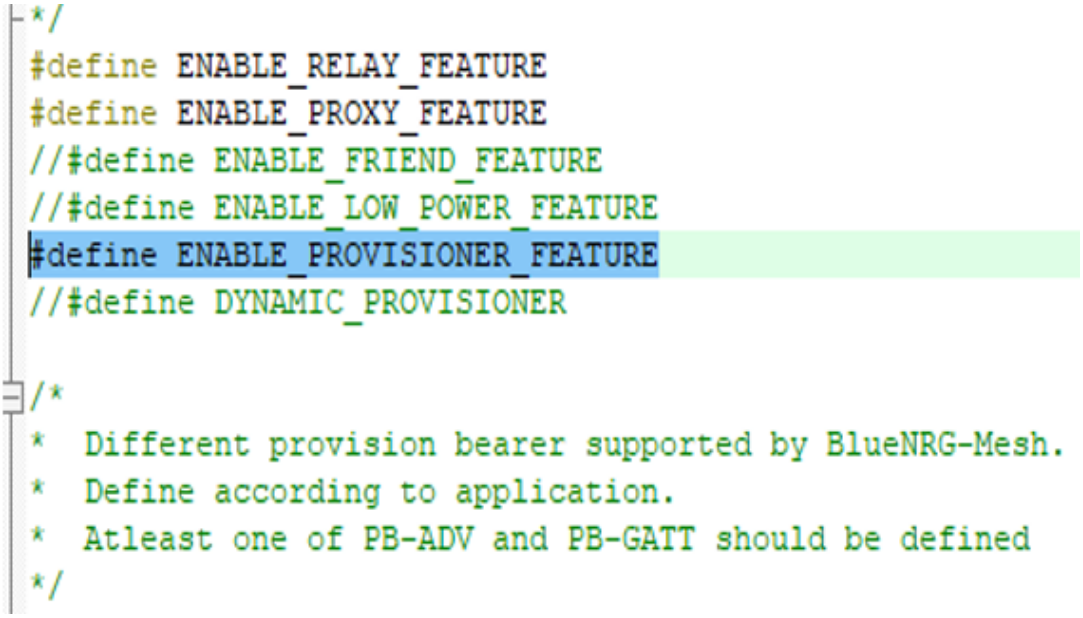
**Figure 13. Path for mesh_cfg_usr.h in the BlueNRG-LP platform**




## 3.1 Enabling the PB-ADV bearer and provisioning feature

The embedded provisioner application uses the PB-ADV bearer to provision the unprovisioned devices. Use the `ENABLE_PROVISIONER_FEATURE` macro in the “mesh_cfg_usr.h” file to enable the provisioner feature and the PB-ADV bearer as shown below.

Figure 14. **Enabling advertising and provisioning feature**



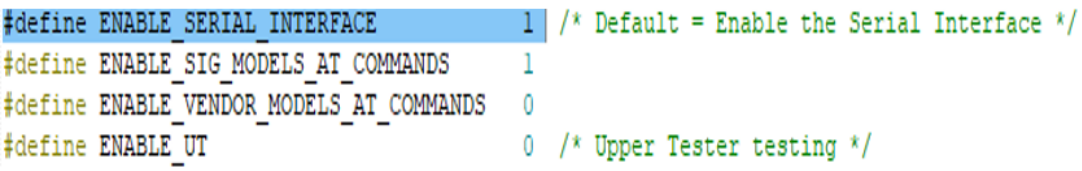
```
*/
#define ENABLE_RELAY_FEATURE
#define ENABLE_PROXY_FEATURE
//#define ENABLE_FRIEND_FEATURE
//#define ENABLE_LOW_POWER_FEATURE
#define ENABLE_PROVISIONER_FEATURE
//#define DYNAMIC_PROVISIONER

/*
*  Different provision bearer supported by BlueNRG-Mesh.
*  Define according to application.
*  Atleast one of PB-ADV and PB-GATT should be defined
*/
```

## 3.2 Enabling the serial interface

The embedded provisioner node takes all inputs in the form of user commands, using a serial interface through a UART peripheral. You can enable this serial interface by using a macro in `ENABLE_SERIAL_INTERFACE` in the “mesh_cfg_usr.h” file as shown below.

Figure 15. **Serial provision interface configuration**



```
#define ENABLE_SERIAL_INTERFACE          1  /* Default = Enable the Serial Interface */
#define ENABLE_SIG_MODELS_AT_COMMANDS    1
#define ENABLE_VENDOR_MODELS_AT_COMMANDS 0
#define ENABLE_UT                        0  /* Upper Tester testing */
```

## 3.3 Console configuration

This procedure initializes the UART interface to communicate with the PC for a console or to print data on a serial terminal. In the “mesh_cfg_usr.h” file, you must call the `TF_PROVISION` and `TF_SERIAL_CTRL` macros to enable the serial interface and to get the provisioning logs. Enable all the default traces to get the logs as shown below.

Figure 16. **Console configuration**

```
/* Enabled by default */
#define TF_GENERIC                          1
#define TF_GENERIC_CLIENT                   1
#define TF_SENSOR                           1
#define TF_LIGHT                            1
#define TF_LIGHT_CLIENT                     1
#define TF_LIGHT_LC                         1
#define TF_VENDOR                           1
#define TF_CONFIG_CLIENT                    1
#define TF_CONFIG_SERVER                    1
#define TF_LPN_FRND                         1
#define TF_PROVISION                        1
#define TF_HANDLER                          1
#define TF_INIT                             1
#define TF_MISC                             1
/* Disabled by default */
#define TF_COMMON                           1
#define TF_GENERIC_M                        1
#define TF_GENERIC_CLIENT_M                 0
#define TF_SENSOR_M                         0
#define TF_LIGHT_M                          1
#define TF_LIGHT_CLIENT_M                   0
#define TF_SENSOR_CLIENT_M                  0
#define TF_LIGHT_LC_M                       0
#define TF_VENDOR_M                         0
#define TF_CONFIG_CLIENT_M                  0
#define TF_NEIGHBOUR                        0
#define TF_MEMORY                           0
#define TF_BEACON                           0
#define TF_SERIAL_CTRL                      1 /
#define TF_VENDOR_APPLI_TEST                0
#define TF_NVM                              0
```

# 4 Using the embedded provisioner to provision the devices

The embedded provisioner application uses a state machine to handle the different provisioning process states. It starts with scanning the Bluetooth® Low Energy beacon state.

In this state, the provisioner:

- scans the unprovisioned device beacon
- establishes a link with the chosen device
- performs the provisioning procedure
- assigns the address to the device
- saves the device information in the flash memory

The embedded provisioner handles the provisioning sequence as described in the following sections to complete the provisioning process of an unprovisioned device.

## 4.1 Commands for the embedded provisioner

The embedded provisioner takes input commands from the user through a serial console. The USB port available on the board communicates with a PC to send commands and receive logs from the board. All the commands and parameters are case insensitive (both capital and regular characters can be used).

The embedded provisioner supports various commands to handle various functionalities required during the provisioning process.

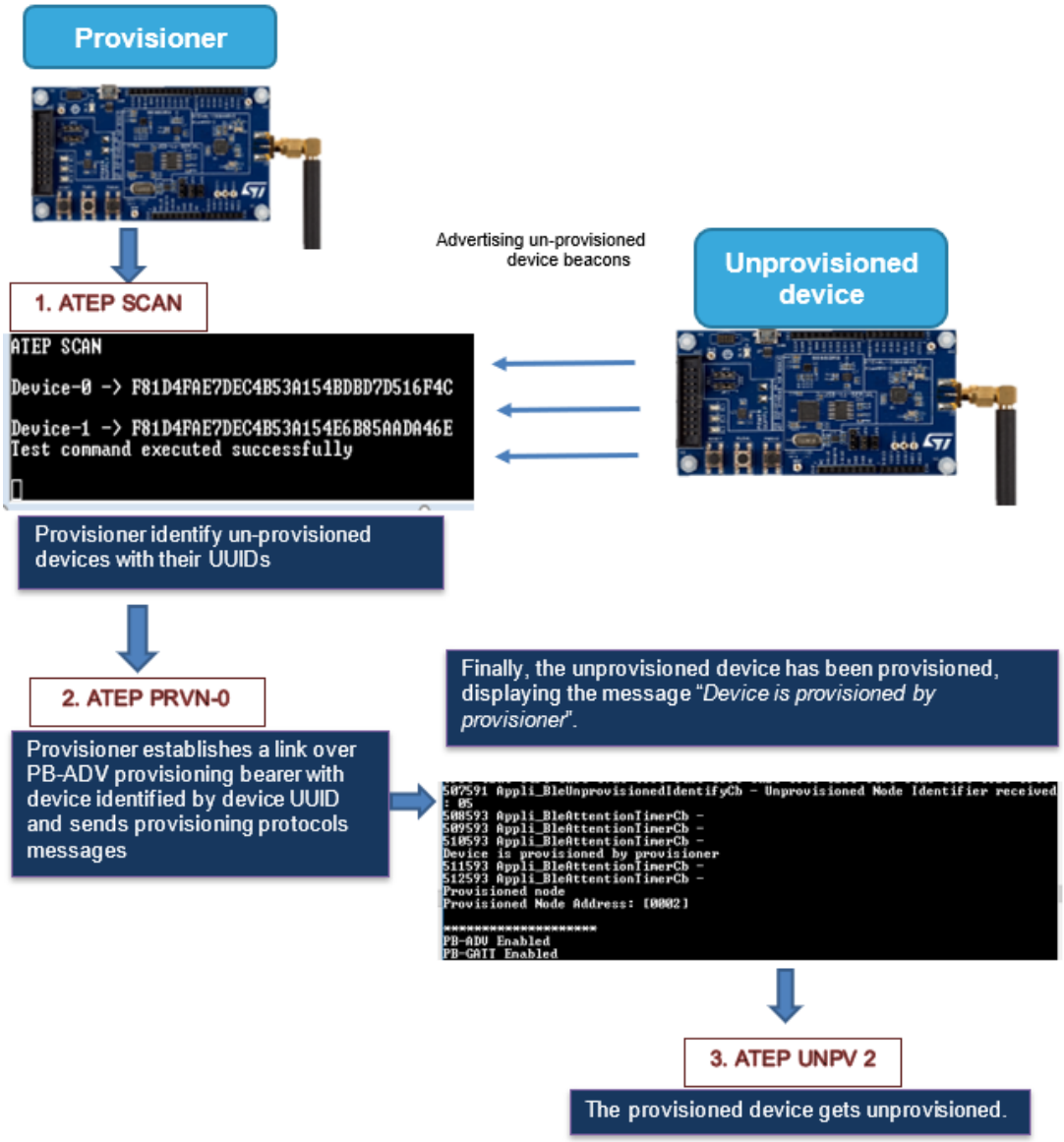
**Table 2. Command set for the embedded provisioner**

| Command string (case insensitive) | Description/functionality |
|---|---|
| `ATEP SCAN` | This command starts scanning all the unprovisioned devices in the radio range of the embedded provisioner. |
| `ATEP PRVN-<X>` | This command starts the provisioning of the unprovisioned devices found through the scanning command.<br>X is the offset of the device in the list of the unprovisioned commands received after scanning. This parameter is in decimal.<br>Example: `ATEP PRVN-0` is used to provision the very first device. It appears in the scanned device list. |
| `ATEP UNPV <X>` | This command unprovisions the provisioned node present in the Bluetooth® Low Energy mesh network.<br>X is the node address assigned to the device after the provisioner provisioning.<br>Example: `ATEP UNPV 2`, where 2 is the provisioned node address. This parameter is in decimal. |

*Note: The next section provides a detailed description of the commands above with some related examples.*

## 4.2 Embedded provisioner overview

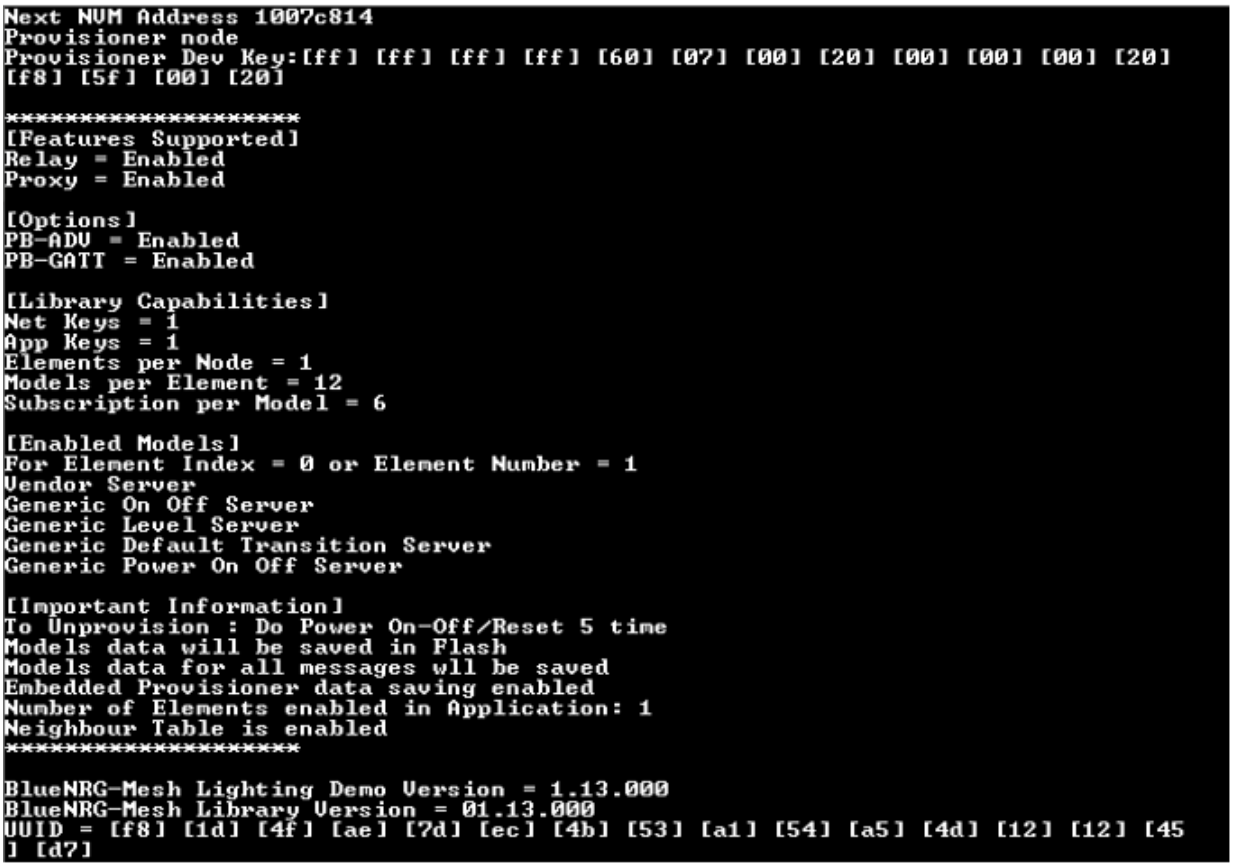
Figure 17. **Node provisioned and unprovisioned by the provisioner**




## 4.3 Powering on the embedded provisioner

After powering on the embedded provisioner node for the first time, it creates the network credentials, such as the network key, application key, IV index, etc. These credentials are required to send and receive the data packets in the mesh network. After their successful creation, a message appears on the serial interface that shows the node as a provisioner node (see the figure below). If the node does not appear as a provisioner node, reflash the firmware into the board.

Figure 18. **Embedded provisioner initialization**




## 4.4 Scanning the Bluetooth® mesh unprovisioned devices

In a Bluetooth® mesh, unprovisioned devices use the unprovisioned device beacon to allow the provisioner to discover them. These devices broadcast their own unique device UUID for their identification. The embedded provisioner node scans the unprovisioned device beacons to discover the presence of any unprovisioned device in the radio range. The scanning process can be initiated by sending a scan command defined as `ATEP SCAN` on the serial terminal.

After receiving this command, the provisioner node scans and provides a list of all the unprovisioned devices in the range. A single scan command can discover a maximum of five devices in a single scan command, which can be added to the network after provisioning. The figure below shows the scan command and the related response for two unprovisioned devices, that is, Device-0 and Device-1 in the network with their corresponding UUIDs.

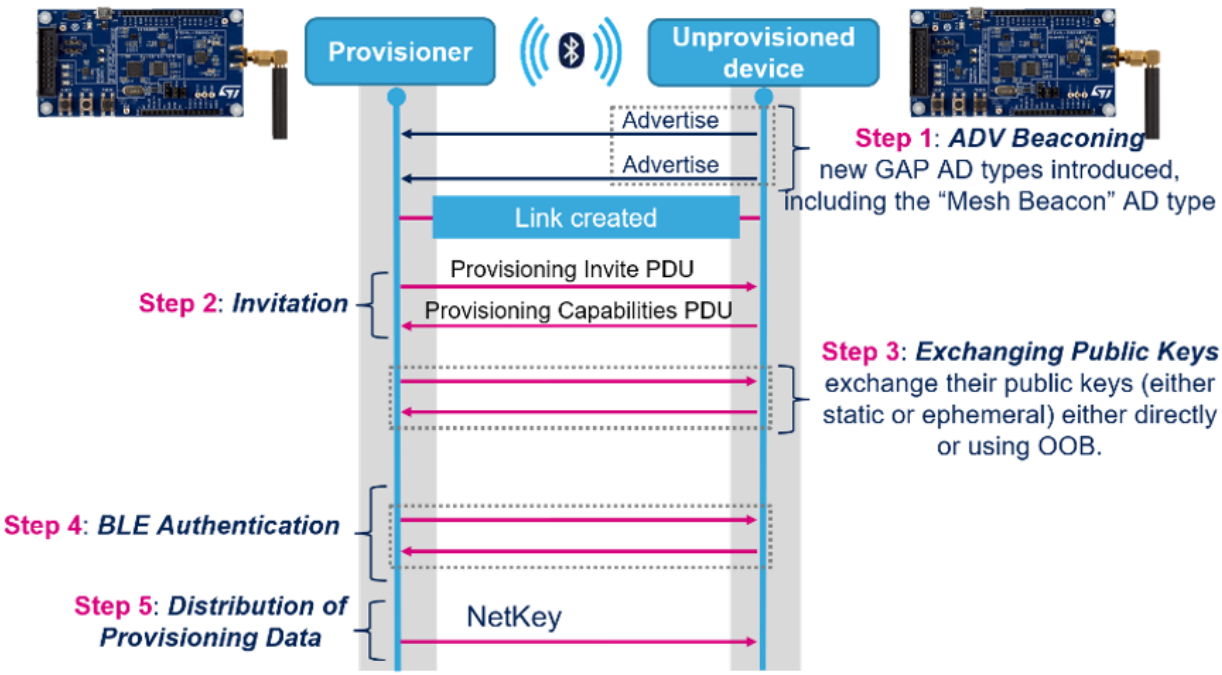
Figure 19. **Scan command to get device UUIDs**

## 4.5 Provisioning and configuration of devices

After scanning the unprovisioned devices, provision and configure them. To start provisioning any unprovisioned device, use the `ATEP PRVN-0` command. 0 is the offset parameter of the list of scanned devices starting from 0. So, here 0 is the very first device of the list. To provision any other device, change the command to reflect that device offset. For example, use `ATEP PRVN-2` to provision the third device in the list.

After issuing the provisioning command, the provisioner node establishes a link over the PB-ADV provisioning bearer with the device identified by its UUID. Over this link, all the required provisioning protocol messages have been sent using the provisioning PDUs. After the device is successfully provisioned, the PB ADV link is closed. The firmware provisioning process is a multistep procedure as per the Bluetooth mesh specification.

**Figure 20. Provisioning process**




Then, the provisioner prints a provisioning successful message along with the calculated device key of the node. After getting this message from the device, the provisioner closes the established link as shown below.

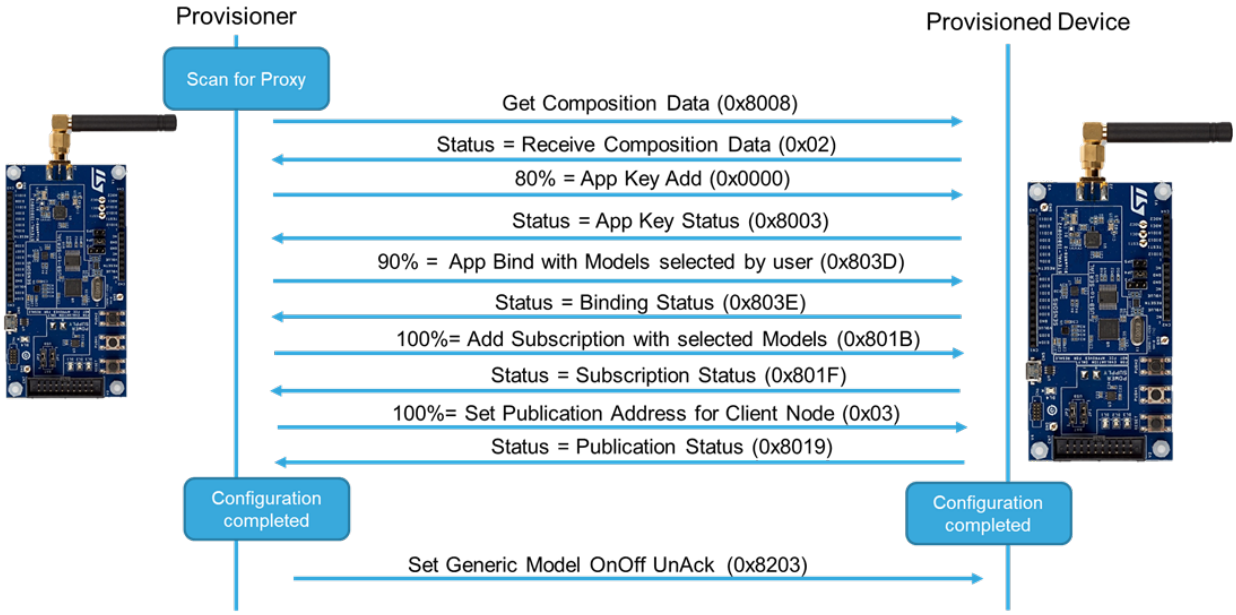
**Figure 21. Provisioning command to provision the devices**

```
ATEP PRVN-0
Test command executed successfully

19168 BluenrgMesh_PbAdvLinkOpenCb - PB-ADV Link opened successfully
21753 BluenrgMesh_PvnrDataInputCallback - Device Key: 92 6c d7 f1 10 26 68 c6 c
 56 6a 5c 23 50 29 d2 21756 BluenrgMesh_PvnrDataInputCallback - App Key: a7 9f
5 66 11 35 4d 2e b5 61 86 d5 5d a6 98 ce 21765 BluenrgMesh_PvnrDataInputCallbac
 -
21769 BluenrgMesh_PvnrDataInputCallback - Node Address Assigned = 2
Device is provisioned by provisioner
```

After successful provisioning, the provisioner starts the configuration process. Through this process, the provisioner node configures the mesh node using configuration model messages. This is a multistep procedure as shown below.

Figure 22. **Configuration process**




After successful configuration, the “** Node is configured**” message indicates that the node has become part of the mesh network as shown below.

Figure 23. **Node configuration after provisioning**

```
57145 BluenrgMesh_PvnrDataInputCallback - Node Address Assigned = 2
evice is provisioned by provisioner
61505 BluenrgMesh_PbAdvLinkCloseCb - PB-ADV Link Closed successfully
61506 AppliPrvnNvm_Process - Saving in SubPage[1007c814] =
75435 Appli_ConfigClient_ConfigureNode - **Node is configured**
```

## 4.6 Unprovisioning the node

To unprovision the provisioned node, that is, to remove it from the mesh network, send the `ATEP UNPV <#Node address>` command from the serial terminal. The node address is the unicast address that the provisioner assigns during the provisioning. This address is unique for each mesh network node.

For example, in the `ATEP UNPV 2` command, "2" is the provisioned node address. This value is in decimal.

The “Device is unprovisioned by provisioner” message appears on the node terminal to signal that the node does not belong to the mesh network anymore.

Figure 24. **Node unprovisioned**

```
Device is provisioned by provisioner
5354 Appli_BleAttentionTimerCb -
6354 Appli_BleAttentionTimerCb -
7354 Appli_BleAttentionTimerCb -
Device is unprovisioned by provisioner
Unprovisioned device

**********************
PB-ADV Enabled
PB-GATT Enabled
Feature: Relay Enabled
Feature: Proxy Enabled
```

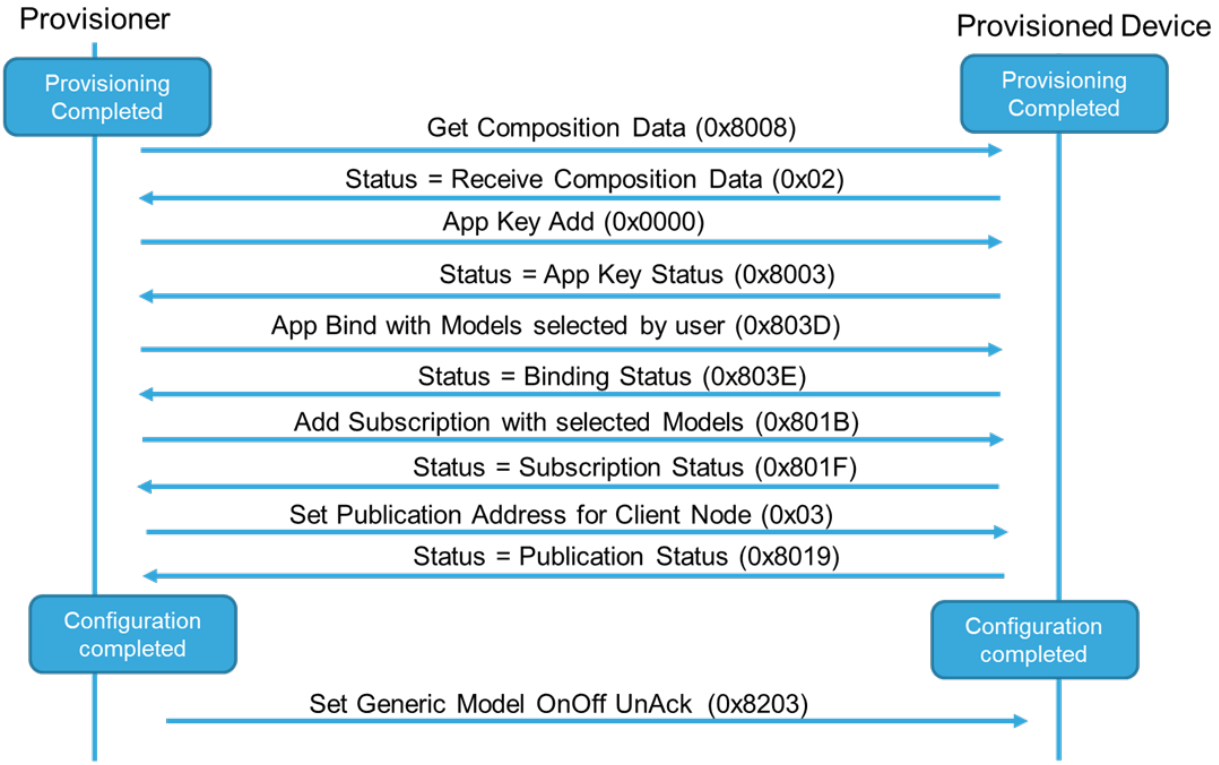
# 5 Node configuration

After provisioning the nodes, configure them with the minimum parameters that enable them to accept and process the mesh model messages.

## 5.1 Node configuration process

Once the configuration is completed, any model message can be sent from the provisioner/configuration client message to the configured node. For example, the following picture shows the `Set Generic Model OnOff` message.

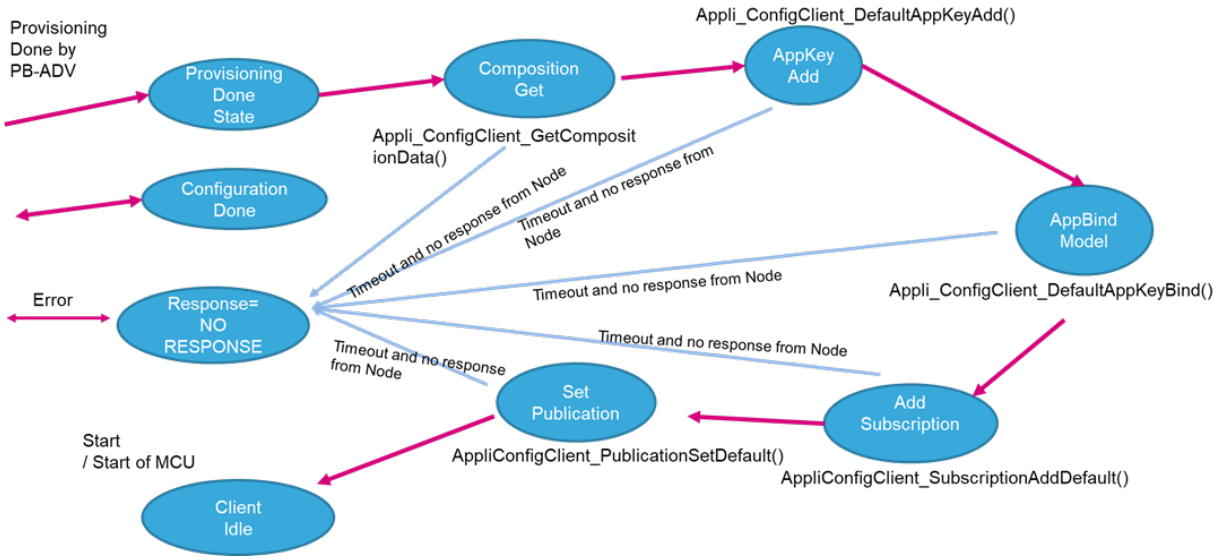
**Figure 25. Node configuration messages**




## 5.2 Firmware state machine

The firmware implements a configuration state machine. This process allows the embedded provisioner to configure the node and assigns the groups, bind them, and assign the publish address.

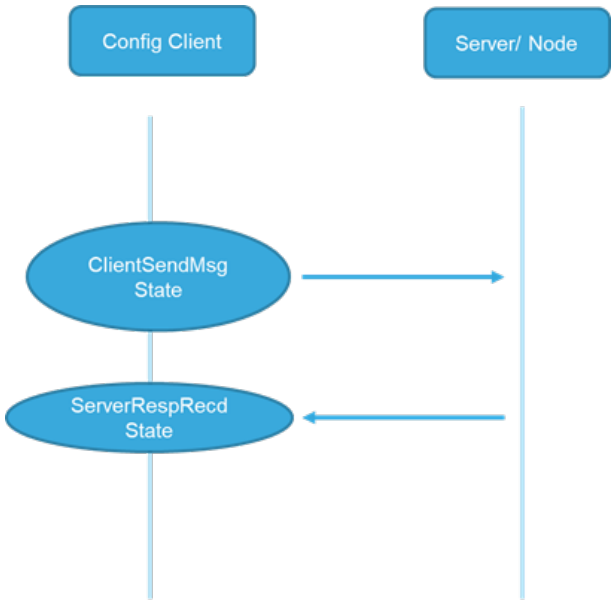
Figure 26. **Configuration state machine**




## 5.3 Configuration client message flow

Nodes are configured through a series of messages from the configuration client to the server node. As per the specifications, the configuration client messages are acknowledged messages. The figure below shows the firmware message and the acknowledgement flow.

Figure 27. **Configuration message flow**




## 5.4 Provisioner self-configuration

The node provisioner can also be used as a network node if it is able to receive and generate the model messages. This requires that the provisioner and the other nodes have the same AppKey to decrypt the model messages. The key needs to be configured in the provisioner. Similarly, the provisioner can subscribe to group information. You can also configure publish addresses.

So, the provisioner has dual role, that is, to provision and configure the other nodes in the network and to configure itself through the `Appli_SelfConfigurationProcess();` function.

The provisioner self-configuration is implemented in the firmware using the following functions:

1. `ApplicationSetNodeSigModelList()`

2. `Appli_ConfigClient_SelfDefaultAppKeyBind()`
3. `AppliConfigClient_SelfSubscriptionSetDefault();`
4. `AppliConfigClient_SelfPublicationSetDefault();`

# 6 Embedded provisioner flash memory management

The embedded provisioner uses the device flash memory to save the information of the provisioned devices. For each provisioned device, 20 bytes of memory is used, which includes:

- 2 bytes for the node address
- 2 bytes for the element number
- 16 bytes for the device key

**Table 3. Provisioned node information saved in the flash memory**

| Byte | Parameters | Endianness |
|---|---|---|
| 2 | Primary element address | Little-endian |
| 2 | Number of the element on the node | Little-endian |
| 16 | Device key of the node | Little-endian |

The total flash memory size available in the BlueNRG-2 or BlueNRG-LP devices is 256 kB. The flash memory starts from the address 0x10040000 and ends at 0x1007FFFF.

In the embedded provisioner application, the 2 kB memory is reserved for saving the information about the provisioned nodes.

The information of the offset and size of the used memory is available in the appli_nvm.h file.

The base address or initial address of this flash memory is at 0x1007D800 and is defined as: `PRVN_NVM_BASE_OFFSET 0x1007D800.`

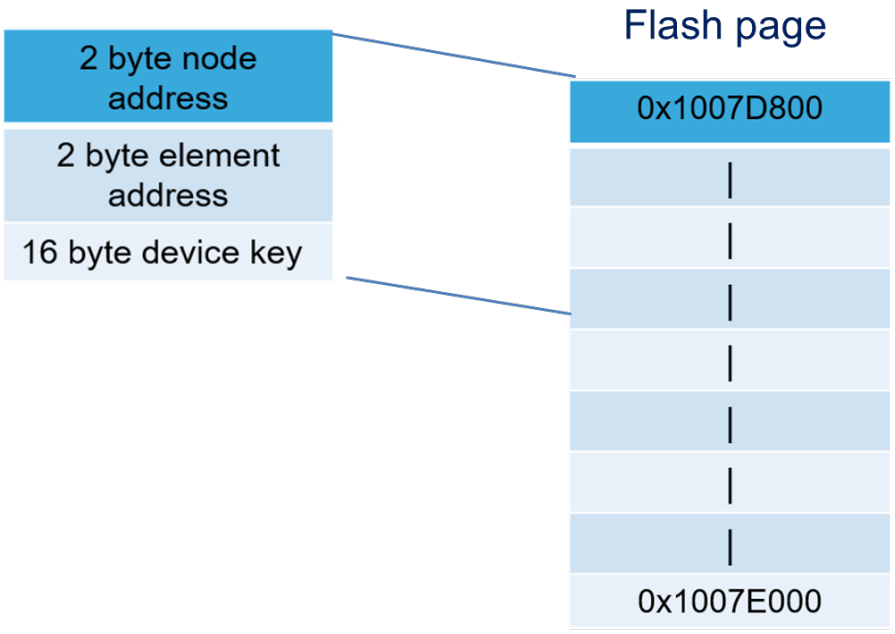
The page size of the flash memory reserved for saving the nodes data is 2 kB and is defined as: `PRVN_NVM_PAGE_SIZE 2048 byte.`

The chunk size or number of bytes to be allocated for each unprovisioned node is 20 bytes and is defined as: `PRVN_NVM_CHUNK_SIZE 20 bytes.`

The total amount of unprovisioned node information that can be saved in the total reserved flash memory is 102 as: `PRVN_NVM_MAX_SUBPAGE = PRVN_NVM_PAGE_SIZE / PRVN_NVM_CHUNK_SIZE = 2048 / 20 = 102 Nodes.`

As per requirement, the reserved page size can be increased using the `PRVN_NVM_PAGE_SIZE` macro. The following figure shows the flash memory addressing that uses a 2 kB page.

Figure 28. **20-byte node data saved in the flash memory**




You can modify this approach to save further information that corresponds to the provisioned nodes. For example, you can modify it to save the node UUID as an additional parameter.

## 6.1 Flash memory update

When the provisioner node boots for the first time, it saves its own information, such as its address, element number, and device key. The location is 0x1007D800 of the flash memory as shown below.

Figure 29. **Embedded provisioner flash memory before node provisioning**

```
0x1007d7f0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0x1007d800  01 00 01 00 e1 94 8d f5 4d 31 f8 3a 98 ca 10 50
0x1007d810  78 a1 b7 69 ff ff ff ff ff ff ff ff ff ff ff ff
```

After the successful provisioning of the unprovisioned devices, the provisioner saves their information, such as its address, element number, and device key. The information is saved in the flash memory at the next location as shown below.

Figure 30. **Embedded provisioner flash memory after node provisioning**

```
0x1007d800  01 00 01 00 e1 94 8d f5 4d 31 f8 3a 98 ca 10 50
0x1007d810  78 a1 b7 69 02 00 01 00 00 87 26 16 c1 5e ad d7
0x1007d820  eb 63 38 e6 a3 ea 75 87 ff ff ff ff ff ff ff ff
```

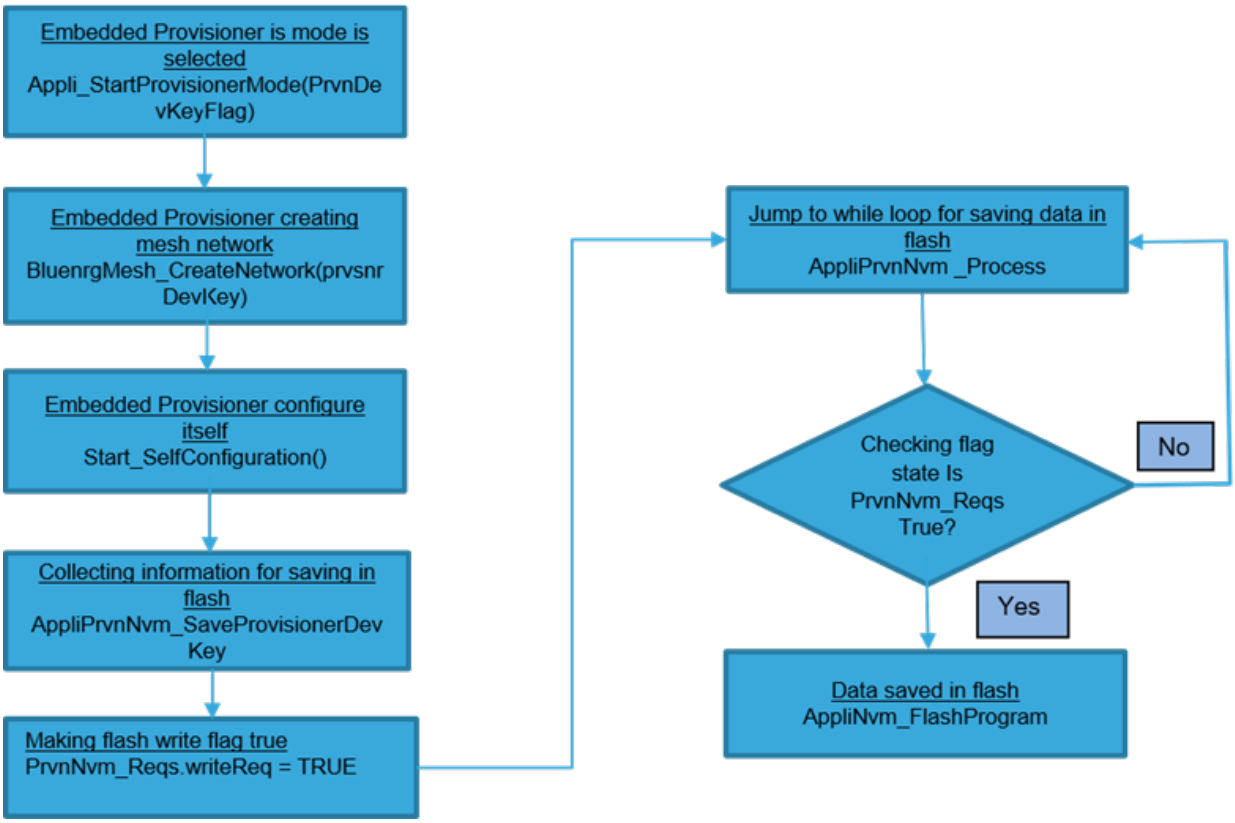
The node address is 02 00. The element number is 01 00 and the device key is a 16-byte value shared by the node, which is saved in the flash memory.

*Note:*

- *In the case of a multielement node, use the primary node address to unprovision the node and avoid an invalid primary address and an "unprovisioned node" error.*
- *Use the node addresses in the range of node number, which can be saved into the flash memory to avoid a "memory full" error.*
- *In the current implementation, the unprovisioned node addresses are not assigned to any new device.*
- *Currently, the node device key is reset to 0x0 after unprovisioning the node, while the node address and the element number remain in the flash memory.*

The figure below shows the flow diagram for the embedded provisioner.

Figure 31. **API flow diagram**



# 7 APIs

In the embedded provisioner application, several APIs are available through which the mesh library takes input from the user and provides the required information.

## 7.1 APIs used for the provisioning process

These APIs are used during the provisioning process and are available in the appli_mesh.c file. The following sections detail the various available APIs.

### 7.1.1 Appli_ProvisionerInit

This API is called once at the beginning of the process to initialize the provisioner node parameters. It is called with the application initialization.

**Table 4. Appli_ProvisionerInit prototype**

| Function | Description |
|---|---|
| Prototype | `Appli_ProvisionerInit` |
| Behavior description | Initializes various provisioner node parameters. It is called with the application initialization. |
| Input parameter | Void |
| Output parameter | Void |

### 7.1.2 SaveProvisionedNodeAddress

This API is called to save the information of the node provisioned by the provisioner to keep record of all the provisioned nodes inside the flash memory. This is called after the successful provisioning of the node.

**Table 5. SaveProvisionedNodeAddress prototype**

| Function | Description |
|---|---|
| Prototype | `SaveProvisionedNodeAddress` |
| Behavior description | Saves the information of the node provisioned by the provisioner to keep record of all the provisioned nodes inside the flash memory |
| Input parameter | Void |
| Output parameter | Void |

### 7.1.3 GetNewProvNodeDevKey

This API is called by the configuration client application to get the device key of the node provisioned by the provisioner. The device key is used by the configuration client to send the configuration messages to the node. This is called after the successful provisioning of the node.

**Table 6. GetNewProvNodeDevKey prototype**

| Function | Description |
|---|---|
| Prototype | `GetNewProvNodeDevKey` |
| Behavior description | Gets the device key of the node provisioned by the provisioner |
| Input parameter | Void |
| Output parameter | `MOBLEUINT8*` device key of the provisioned node |

### 7.1.4 GetProvNodeDevKey

This API is called by the configuration client application to get the device key of any of the nodes provisioned by the provisioner. The node address of the provisioned node is used as a parameter to select the node whose device key is needed. The device key is used by the configuration client to send the configuration messages to the node. This is called after the successful provisioning of the node.

**Table 7. GetProvNodeDevKey prototype**

| Function | Description |
|---|---|
| Prototype | `GetProvNodeDevKey` |
| Behavior description | Gets the device key of any of the nodes provisioned by the provisioner |
| Input parameter | `MOBLEUINT16 na`: node address of the provisioned node<br>`MOBLEUINT32 *dev`: device key address |
| Output parameter | `MOBLEUINT8*`: pointer of the node device key |

### 7.1.5 GetNewProvNodeAppKey

This API is called by the configuration client application to get the application key to be shared with the node provisioned by the provisioner. The app key is shared by the configuration client to the node to help it get or send the mesh application messages. This is called after the successful provisioning and during the configuration of the node. The table below details the prototype of this API: `(void) MOBLEUINT8* GetNewProvNodeAppKey(void)`.

**Table 8. GetNewProvNodeAppKey prototype**

| Function | Description |
|---|---|
| Prototype | `GetNewProvNodeAppKey` |
| Behavior description | Gets the application key to be shared with the node provisioned by the provisioner |
| Input parameter | Void |
| Output parameter | `MOBLEUINT8*`: pointer of the application key |

### 7.1.6 GetAddressToConfigure

This API is called by the configuration client application to get the node address of the node provisioned by the provisioner. The node address is used by the configuration client to start the configuration process of the just provisioned node. This is called after the successful provisioning of the node and during the configuration of the node.

**Table 9. GetAddressToConfigure prototype**

| Function | Description |
|---|---|
| Prototype | `GetAddressToConfigure` |
| Behavior description | Gets the node address of the node provisioned by the provisioner |
| Input parameter | Void |
| Output parameter | `MOBLE_ADDRESS`: address of the just provisioned node |

### 7.1.7 GetNewAddressToProvision

This API is called by the mesh library during the provisioning procedure through `BluenrgMesh_PvnrDataInputCallback` callback function. The node addresses in the mesh network are exclusive and unique. So, the next available address, which has not been previously used by the provisioner, is provided by this API. This node address is shared and assigned to the provisioned node.

Table 10. **GetNewAddressToProvision prototype**

| Function | Description |
|---|---|
| Prototype | `GetNewAddressToProvision` |
| Behavior description | Gets the next available address, which has not been previously used by the provisioner |
| Input parameter | Void |
| Output parameter | `MOBLE_ADDRESS`: address to be assigned to the provisioned node |

### 7.1.8 BluenrgMesh_PvnrDataInputCallback

This callback function is called by the mesh library during the provisioning procedure to get the information of the node address. Then it provides this information, such as the device key, the app key, and the number of elements of the device, to the application. The node addresses in the mesh network are exclusive and unique. So, the next available address, which has not been previously used, is provided by this API. This node address is shared and assigned to the provisioned node.

Table 11. **BluenrgMesh_PvnrDataInputCallback prototype**

| Function | Description |
|---|---|
| Prototype | `BluenrgMesh_PvnrDataInputCallback` |
| Behavior description | Gets the information of the node address and provides the information, such as the device key, the app key, and the number of elements of the device to the application |
| Input parameter | `MOBLEUINT8* devKey`: pointer of the device key<br>`MOBLEUINT8* appKey`: pointer of the app key<br>`MOBLEUINT8 numOfElements`: number of elements |
| Output parameter | `MOBLEUINT16`: address to be assigned to the provisioned node |

### 7.1.9 Appli_StartProvisionerMode

This API is called once at the beginning to initialize the node as a provisioner and create the mesh network parameters. This function also performs the provisioner node self-configuration. It is called after the application initialization in the main function.

Table 12. **Appli_StartProvisionerMode**

| Function | Description |
|---|---|
| Prototype | `Appli_StartProvisionerMode` |
| Behavior description | Initializes the node as provisioner and creates the mesh network parameters |
| Input parameter | `MOBLEUINT8 mode`: flag to save the credentials |
| Output parameter | Void |

### 7.1.10 Start_SelfConfiguration

This API is called once at the beginning to initialize the node as a provisioner and create the mesh network parameters. This function performs the provisioner node self-configuration. It is called by the `Appli_StartProvisionerMode` function.

Table 13. **Start_SelfConfiguration prototype**

| Function | Description |
|---|---|
| Prototype | `Start_SelfConfiguration` |
| Behavior description | Provisioner node self-configuration |
| Input parameter | Void |

| Function | Description |
|---|---|
| Output parameter | Void |

### 7.1.11 Appli_SelfConfigurationProcess

This function performs the provisioner node self-configuration. It is periodically called by the application process function to complete the configuration process of the provisioner node.

**Table 14. Appli_SelfConfigurationProcess prototype**

| Function | Description |
|---|---|
| Prototype | `Appli_SelfConfigurationProcess` |
| Behavior description | Provisioner node self-configuration |
| Input parameter | Void |
| Output parameter | Void |

### 7.1.12 BluenrgMesh_ProvisionDevice

This function can be used to start the provisioning of one of the unprovisioned devices available in the radio range of the embedded provisioner. After calling this function, the mesh library initializes the link creation and the provisioning procedure. It is called by the serial command available in the embedded provisioner application.

**Table 15. BluenrgMesh_ProvisionDevice prototype**

| Function | Description |
|---|---|
| Prototype | `BluenrgMesh_ProvisionDevice` |
| Behavior description | Starts the provisioning of one of the unprovisioned devices |
| Input parameter | `neighbor_params_t *unprovDeviceArray`: pointer of an array that has an unprovisioned device UUID<br>`MOBLEUINT16 index`: index of the device to be provisioned |
| Output parameter | `MOBLE_RESULT`: result status |

### 7.1.13 BluenrgMesh_PbAdvLinkOpenCb

This callback function is called by the mesh library when the PB-ADV link is established with the unprovisioned device. This function indicates the successful link creation between the embedded provisioner and the device. If the link is not created after sending the link open message, the embedded provisioner tries to establish the link with the device multiple times. After attempting multiple times, if the link is not established, the embedded provisioner stops the link establishment procedure with the device and calls the `PbAdvLinkCloseCb` callback. Then, the embedded provisioner is ready to establish the link with any device.

**Table 16. BluenrgMesh_PbAdvLinkOpenCb prototype**

| Function | Description |
|---|---|
| Prototype | `BluenrgMesh_PbAdvLinkOpenCb` |
| Behavior description | Indicates the successful link creation between the embedded provisioner and the device |
| Input parameter | Void |
| Output parameter | Void |

### 7.1.14 BluenrgMesh_PbAdvLinkCloseCb

This callback function is called by the mesh library when the PB-ADV link is either not established or closed after packet sharing with the unprovisioned device. This function indicates that the successful link is closed between the embedded provisioner and the device. After the link open callback, and before the link closed callback, `BluenrgMesh_ProvisionCallback` indicates that the provisioning procedure is finished.

**Table 17. BluenrgMesh_PbAdvLinkCloseCb prototype**

| Function | Description |
|---|---|
| Prototype | `BluenrgMesh_PbAdvLinkCloseCb` |
| Behavior description | Callback function to indicate that the successful link is closed between the embedded provisioner and the device |
| Input parameter | Void |
| Output parameter | Void |

### 7.1.15 BluenrgMesh_ProvisionCallback

This callback function is called by the mesh library after completion of the provisioning procedure on both the provisioner and the device side. This function indicates the successful provisioning of the device. Information on the assigned node address is also printed on the node side.

**Table 18. BluenrgMesh_ProvisionCallback prototype**

| Function | Description |
|---|---|
| Prototype | `BluenrgMesh_ProvisionCallback` |
| Behavior description | Callback function to indicate the successful completion of the provisioning procedure |
| Input parameter | Void |
| Output parameter | Void |

## 7.2 APIs used for the configuration process

### 7.2.1 ConfigClient_CompositionDataGet

The following APIs are available in config_client.c.

**Table 19. ConfigClient_CompositionDataGet prototype**

| Function | Description |
|---|---|
| Prototype | `ConfigClient_CompositionDataGet` |
| Behavior description | This function is called to read the composition data of the node |
| Input parameter | `MOBLE_ADDRESS dst_peer`: address of the node for which the configuration client needs to read the composition page data |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.2 ConfigClient_AppKeyAdd

**Table 20. ConfigClient_AppKeyAdd prototype**

| Function | Description |
|---|---|
| Prototype | `ConfigClient_AppKeyAdd` |
| Behavior description | This function is called to add the default app keys and the net keys to a node under configuration |

| Function | Description |
|---|---|
| Input parameter | MOBLE_ADDRESS dst_peer: address of the node for which the configuration client needs to read the composition page data |
| Input parameter | MOBLEUINT16 netKeyIndex: parameter for the network key index that should be updated for the app key |
| Input parameter | MOBLEUINT16 appKeyIndex: parameter for the app key index that should be updated for the app key |
| Input parameter | MOBLEUINT8* appkey: pointer to the application key |
| Output parameter | MOBLE_RESULT: result |

### 7.2.3 ConfigClient_PublicationSet

**Table 21. ConfigClient_PublicationSet prototype**

| Function | Description |
|---|---|
| Prototype | MOBLE_RESULT ConfigClient_PublicationSet (MOBLEUINT16 elementAddress,<br>MOBLEUINT16 publishAddress,<br>MOBLEUINT16 appKeyIndex,<br>MOBLEUINT8 credentialFlag,<br>MOBLEUINT8 publishTTL,<br>MOBLEUINT8 publishPeriod,<br>MOBLEUINT8 publishRetransmitCount,<br>MOBLEUINT8 publishRetransmitIntervalSteps,<br>MOBLEUINT32 modelIdentifier) |
| Behavior description | This function is called to add the default app keys and the net keys to a node under configuration |
| Input parameter | MOBLEUINT16 publishAddress: contains the publication address to be configured |
| Input parameter | MOBLEUINT16 appKeyIndex: parameter for the app key index that should be updated for the app key |
| Input parameter | MOBLEUINT8 credentialFlag: value of the friendship credential flag |
| Input parameter | MOBLEUINT8 publishTTL: default TTL value for the outgoing messages |
| Input parameter | MOBLEUINT8 publishPeriod: period for the periodic status publishing |
| Input parameter | MOBLEUINT8 publishRetransmitCount: number of retransmissions for each published message |
| Input parameter | MOBLEUINT8 publishRetransmitIntervalSteps: number of 50-millisecond steps between retransmissions |
| Input parameter | MOBLEUINT32 modelIdentifier: SIG model ID or vendor model ID |
| Output parameter | MOBLE_RESULT: result |

### 7.2.4 ConfigClient_SubscriptionAdd

**Table 22. ConfigClient_SubscriptionAdd prototype**

| Function | Description |
|---|---|
| Prototype | MOBLE_RESULT ConfigClient_SubscriptionAdd (MOBLEUINT16 elementAddress,<br>MOBLEUINT16 address,<br>MOBLEUINT32 modelIdentifier) |

| Function | Description |
|---|---|
| Input parameter | `MOBLEUINT16 elementAddress`: contains the element address to be configured for the subscription |
| Input parameter | `MOBLEUINT16 address`: contains the subscription address for the element address |
| Input parameter | `MOBLEUINT32 modelIdentifier`: SIG model ID or vendor model ID |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.5 ConfigClient_NodeReset

**Table 23. ConfigClient_NodeReset prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_NodeReset (MOBLEUINT16 elementAddress)` |
| Behavior description | This function is called to reset the nNode |
| Input parameter | `MOBLEUINT16 elementAddress`: contains the element address to be reset |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.6 ConfigClient_ModelAppBind

**Table 24. ConfigClient_ModelAppBind prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_ModelAppBind (MOBLEUINT16 elementAddress, MOBLEUINT16 appKeyIndex, MOBLEUINT32 modelIdentifier)` |
| Behavior description | This function is called to bind an app key to a model |
| Input parameter | `MOBLEUINT16 elementAddress`: contains the element address to be reset |
| Input parameter | `MOBLEUINT16 appKeyIndex`: parameter for the app key index that should be updated for the model binding |
| Input parameter | `MOBLEUINT32 modelIdentifier`: SIG model ID or vendor model ID |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.7 ConfigClient_CompositionDataStatusResponse

**Table 25. ConfigClient_CompositionDataStatusResponse prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_CompositionDataStatusResponse (MOBLEUINT8 const *pSrcComposition, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the composition |
| Input parameter | `MOBLEUINT8 const *pSrcComposition`: pointer to the data string read from the node |
| Input parameter | `MOBLEUINT32 length`: contains the length of the composition data |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.8 ConfigClient_AppKeyStatus

**Table 26. ConfigClient_AppKeyStatus prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_AppKeyStatus (MOBLEUINT8 const *pSrcAppKeyStatus, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the app key add |
| Input parameter | `MOBLEUINT8 const *pSrcAppKeyStatus`: pointer to the data string read from the node as response |
| Input parameter | `MOBLEUINT32 length`: contains the length of the status data |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.9 MOBLE_RESULT ConfigClient_PublicationStatus(MOBLEUINT8 const *pPublicationStatus, MOBLEUINT32 length)

**Table 27. MOBLE_RESULT ConfigClient_PublicationStatus(MOBLEUINT8 const *pPublicationStatus, MOBLEUINT32 length) prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_PublicationStatus(MOBLEUINT8 const *pPublicationStatus, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the model publication state of an outgoing message that is published by the model |
| Input parameter | `MOBLEUINT8 const * pPublicationStatus`: pointer to the data string read from the node as response |
| Input parameter | `MOBLEUINT32 length`: contains the length of the status data |
| Output parameter | `MOBLE_RESULT` result |

### 7.2.10 ConfigClient_SubscriptionStatus

**Table 28. ConfigClient_SubscriptionStatus prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_SubscriptionStatus(MOBLEUINT8 const *pSrcSubscriptionStatus, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the model subscription message |
| Input parameter | `MOBLEUINT8 const * pSrcSubscriptionStatus`: pointer to the data string read from the node as response |
| Input parameter | `MOBLEUINT32 length`: contains the length of the status data |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.11 ConfigClient_NodeResetStatus

**Table 29. ConfigClient_NodeResetStatus prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_NodeResetStatus(MOBLEUINT8 const *pStatus, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the node reset |
| Input parameter | `MOBLEUINT8 const * pStatus`: pointer to the data string read from the node as response |
| Input parameter | `MOBLEUINT32 length`: contains the length of the status data |
| Output parameter | `MOBLE_RESULT`: result |

### 7.2.12 ConfigClient_ModelAppStatus

**Table 30. ConfigClient_ModelAppStatus prototype**

| Function | Description |
|---|---|
| Prototype | `MOBLE_RESULT ConfigClient_ModelAppStatus (MOBLEUINT8 const *pSrcModelAppStatus, MOBLEUINT32 length)` |
| Behavior description | This function is a callback when the response is received for the app key add |
| Input parameter | `MOBLEUINT8 const * pSrcModelAppStatus`: pointer to the data string read from the node as response |
| Input parameter | `MOBLEUINT32 length`: contains the length of the status data |
| Output parameter | `MOBLE_RESULT`: result |

## 7.3 APIs used for the self-configuration process

### 7.3.1 ApplicationSetNodeSigModelList

This function is called by the provisioner to transmit the composition data of the application to the mesh library. This helps to configure the mesh library with the right set of the parameters.

**Table 31. ApplicationSetNodeSigModelList prototype**

| Function | Description |
|---|---|
| Prototype | `ApplicationSetNodeSigModelList` |
| Behavior description | This function is called by the provisioner to pass the composition data of the application to the mesh library.<br>It sets the list of the SIG models from the application to the library.<br>It also internally calls the following functions:<br>• `ApplicationGetSigModelList`<br>• `ApplicationGetVendorModelList`<br>The purpose of this API is to initialize the array for the models so that the binding, subscription, and publication can be performed. |

| Function | Description |
|---|---|
| | When called, the following functions search the available models in the array and add the binding, subscription, and publication information:<br>• `Appli_ConfigClient_SelfDefaultAppKeyBind()`<br>• `AppliConfigClient_SelfSubscriptionSetDefault();`<br>• `AppliConfigClient_SelfPublicationSetDefault();`<br>So, `ApplicationSetNodeSigModelList()` must be called previously than all the other functions.<br>On the node side, the `GetComposition` message updates the model array. |
| Input parameter | Void |
| Output parameter | `MOBLE_RESULT`: result status |

### 7.3.2 Appli_ConfigClient_SelfDefaultAppKeyBind

This function is called by the provisioner to set the app key from the application to the library.

**Table 32. Appli_ConfigClient_SelfDefaultAppKeyBind prototype**

| Function | Description |
|---|---|
| Prototype | `Appli_ConfigClient_SelfDefaultAppKeyBind` |
| Behavior description | This function binds the element (node) with the app key index and models.<br>It takes the `aSigModelsToBind` as an input. You have to fill or enable the models to be added or bound to the application key. The model IDs in this array are a subset of the models declared in `Appli_SIG_Models[]`.<br>`elementAddress = BluenrgMesh_GetAddress(); /* Take the node address to bind */`<br>`modelIdentifier -> This is taken from the array one by one and the binding is done`<br>`appKeyIndex = DEFAULT_APPKEY_INDEX; -> This value is taken a Zero for AppKey index`<br>This function then calls the following one:<br>`ConfigClient_ModelAppBind (elementAddress, appKeyIndex, modelIdentifier);` |
| Input parameter | Void |
| Output parameter | `MOBLE_RESULT`: result status |

### 7.3.3 AppliConfigClient_SelfSubscriptionSetDefault

This function is called by the provisioner to set the subscription information from the application to the mesh library.

**Table 33. AppliConfigClient_SelfSubscriptionSetDefault prototype**

| Function | Description |
|---|---|
| Prototype | `AppliConfigClient_SelfSubscriptionSetDefault` |
| Behavior description | This function adds the subscription addresses to the element models for the default settings.<br>`MOBLEUINT16 address = DEFAULT_GROUP_ADDR;` → the default settings are defined in the macro<br>`MOBLE_RESULT result = MOBLE_RESULT_SUCCESS;`<br>`numSIGmodels = sizeof(aSubscribeModels)/sizeof(MOBLEUINT16);` → this array is taken as an input to read the models to subscribe |

| Function | Description |
|---|---|
| | `elementAddress = BluenrgMesh_GetAddress();` → self-address is taken<br>`modelIdentifier = 0;`<br>`modelIdentifier = (MOBLEUINT16)`<br>`aSubscribeModels[index];`<br>To add the subscription, this function then calls the following one:<br>`ConfigClient_SubscriptionAdd (elementAddress, address, modelIdentifier);` |
| Input parameter | Void |
| Output parameter | `MOBLE_RESULT`: result status |

### 7.3.4 AppliConfigClient_SelfPublicationSetDefault

**Table 34. AppliConfigClient_SelfPublicationSetDefault prototype**

| Function | Description |
|---|---|
| Prototype | `AppliConfigClient_SelfPublicationSetDefault` |
| Behavior description | This function adds the publication addresses to the element models for the default settings. The default settings are taken from the macros and put in the message. You can change these settings as per the application requirements.<br>`MOBLEUINT16 publishAddress =DEFAULT_PUBLISH_ADDR;`<br>`MOBLEUINT16 appKeyIndex = DEFAULT_APPKEY_INDEX;`<br>`MOBLEUINT8 credentialFlag = DEFAULT_CREDENTIAL_FLAG;`<br>`MOBLEUINT8 publishTTL = DEFAULT_PUBLISH_TTL;`<br>`MOBLEUINT8 publishPeriod = DEFAULT_PUBLISH_PERIOD;`<br>`MOBLEUINT8 publishRetransmitCount = DEFAULT_PUBLISH_RETRANSMIT_COUNT;`<br>`MOBLEUINT8 publishRetransmitIntervalSteps= DEFAULT_PUBLISH_RETRANSMIT_INTERVAL_STEPS;`<br>`numSIGmodels = sizeof(aPublishModels)/sizeof(MOBLEUINT16);` → this array is taken as an input for the publication models<br>`/* Start the Publication Add message */`<br>`elementAddress = BluenrgMesh_GetAddress();` → self-address is taken<br>`modelIdentifier = (MOBLEUINT16) aPublishModels[index];` → this array is taken as an input for the publication models and the models are read one-by-one from the array |
| Input parameter | Void |
| Output parameter | `MOBLE_RESULT`: result status |

# Revision history

Table 35. **Document revision history**

| Date | Revision | Changes |
|---|---|---|
| 17-Mar-2022 | 1 | Initial release. |

# Contents

# List of tables

# List of figures