AN1200: *Bluetooth*® Mesh for iOS and Android ADK

This document describes how to get started with Bluetooth Mesh application development for iOS and Android smart phones and tablets using the Silicon Labs Bluetooth Mesh for iOS and Android Application Development Kit (ADK).

The document also provides a high-level architecture overview of the Silicon Labs Bluetooth Mesh library, how it relates to the Bluetooth LE stack provided by the iOS and Android operating systems and what APIs are available. It also contains code snippets and explanations for the most common Bluetooth Mesh use cases.

**KEY POINTS**

- Introduction to the Silicon Labs’ Bluetooth mesh for iOS and Android ADK.
- Prerequisites for development.
- Contents of the Bluetooth mesh iOS and Android ADK.
- Requirements to get started with development.
- Bluetooth mesh structure overview.
- API references for iOS and Android.
## Table of Contents

1 Introduction ................................................................................................................................................................... 7

2 Prerequisites for Development ........................................................................................................................................ 9

2.1 iOS ........................................................................................................................................................................................................... 9

2.2 Android .................................................................................................................................................................................................... 9

3 Contents of Bluetooth Mesh for iOS and Android ADK .................................................................................................. 11

3.1 The Bluetooth Mesh Stack library ........................................................................................................................................ 11

3.1.1 iOS ................................................................................................................................................................................................... 11

3.1.2 Android ................................................................................................................................................................................................ 11

3.2 Bluetooth Mesh Network and Device Database ......................................................................................................... 11

3.3 Reference Application Source Code ................................................................................................................................ 11

3.4 Documentation .................................................................................................................................................................. 11

4 Getting Started with Development .................................................................................................................................... 12

4.1 iOS ................................................................................................................................................................................................... 12

4.2 Android ................................................................................................................................................................................................ 14

5 Bluetooth Mesh Structure Overview .................................................................................................................................... 18

6 Bluetooth Mesh API Reference for iOS .................................................................................................................................... 19

6.1 Initializing the BluetoothMesh .................................................................................................................................................. 19

6.1.1 BluetoothMesh .................................................................................................................................................................. 19

6.1.2 Set Up Supported Vendor Models ...................................................................................................................................... 19

6.1.3 Set Up Mesh Limits ............................................................................................................................................................ 19

6.2 Request IV Index Update ...................................................................................................................................................... 20

6.3 Set Up Bluetooth Layer (ConnectableDevice) ................................................................................................................... 20

6.3.1 Device Advertisement Data .................................................................................................................................................. 21

6.3.2 Device UUID ....................................................................................................................................................................... 21

6.3.3 Device Name ....................................................................................................................................................................... 21

6.3.4 Device Connection State ....................................................................................................................................................... 22

6.3.5 Connect to the Device ........................................................................................................................................................... 22

6.3.6 Disconnect from the Device .................................................................................................................................................. 22

6.3.7 Check if a Device Contains a Service ................................................................................................................................ 23

6.3.8 Maximum Transmission Unit for Given Device Service .................................................................................................. 23

6.3.9 Write Data to a Given Service and Characteristic .......................................................................................................... 23

6.3.10 Subscribe to a Given Service and Characteristic ............................................................................................................ 24

6.4 Provision a Device to a Subnet ........................................................................................................................................ 24

6.4.1 Create Network ................................................................................................................................................................. 25

6.4.2 Create Subnet ................................................................................................................................................................. 25
6.4.3 Find Non-Provisioned Bluetooth Devices ................................................................. 25
6.4.4 Provision the Device ................................................................................................. 25
6.4.4.1 In-Band Provisioning............................................................................................ 26
6.4.4.2 OOB Provisioning ............................................................................................... 26
6.4.4.2.1 SBMProvisionerOOB Protocol ....................................................................... 26
6.5 Add a Node to Another Subnet ................................................................................... 29
6.6 Remove a Node from a Subnet .................................................................................... 29
6.7 Connect with a Subnet ............................................................................................... 30
6.7.1 Find All Proxies in the Device Range ..................................................................... 30
6.7.2 Get Node Representing a Given Device ................................................................. 30
6.7.3 Get Node Representing a Given Device in a Specific Subnet ............................... 30
6.8 Create a Group in a Given Subnet .............................................................................. 31
6.9 Remove Group .......................................................................................................... 31
6.10 Add a Node to a Group ............................................................................................ 31
6.11 Remove a Node from a Group .................................................................................. 31
6.12 Bind a Model with A Group ..................................................................................... 32
6.13 Unbind a Model from a Group ................................................................................ 32
6.14 Add Subscription Settings to a Model .................................................................... 32
6.14.1 SIG Model .......................................................................................................... 32
6.14.2 Vendor Model ..................................................................................................... 33
6.15 Add Publication Settings to a Model ....................................................................... 33
6.15.1 SIG Model .......................................................................................................... 33
6.15.1.1 Publish via SBMGroup Address ..................................................................... 33
6.15.1.2 Publish Directly to the Provisioner ................................................................. 34
6.15.2 Vendor Model ..................................................................................................... 34
6.15.2.1 Publish via SBMGroup Address ..................................................................... 34
6.15.2.2 Publish Directly to the Provisioner ................................................................. 34
6.16 Control Node Functionality ...................................................................................... 35
6.16.1 Get Value for a Single SIG Model from the Node ............................................... 35
6.16.2 Get Value for All Specific SIG Models Bound with a Group ............................. 35
6.16.3 Set Value for a Single SIG Model from the Node ............................................... 35
6.16.4 Set Value for All Specific SIG Models Bound with the Group .......................... 36
6.16.5 Control Sensor Models ....................................................................................... 36
6.16.5.1 Get Sensor Model Values ............................................................................... 36
6.16.5.1.1 Get Sensor Descriptors from the Node ....................................................... 36
6.16.5.1.2 Get Sensor State from the Node .................................................................. 37
6.16.5.1.3 Get Sensor Cadence from the Node ............................................................ 37
6.16.5.1.4 Get Sensor Setting from the Node ............................................................... 38
6.16.5.1.5 Get Sensor Settings from the Node .............................................................. 38
6.16.5.1.6 Get Sensor Descriptor from All Nodes in the Group .................................. 38
6.16.5.1.7 Get Sensor State from All Nodes in the Group .......................................... 39
## Bluetooth Mesh API Reference for Android

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Initializing the BluetoothMesh</td>
<td>66</td>
</tr>
<tr>
<td>7.1.1 BluetoothMesh</td>
<td>66</td>
</tr>
<tr>
<td>7.1.2 Set Up Supported Vendor Models</td>
<td>66</td>
</tr>
<tr>
<td>7.1.3 Set Up Mesh Limits</td>
<td>66</td>
</tr>
<tr>
<td>7.2 Set Up Bluetooth Layer (ConnectableDevice)</td>
<td>68</td>
</tr>
<tr>
<td>7.2.1 Device Advertisement Data</td>
<td>68</td>
</tr>
<tr>
<td>7.2.2 Device UUID</td>
<td>69</td>
</tr>
<tr>
<td>7.2.3 Device Name</td>
<td>69</td>
</tr>
<tr>
<td>7.2.4 Device Connection State</td>
<td>69</td>
</tr>
<tr>
<td>7.2.5 Connect to the Device</td>
<td>69</td>
</tr>
<tr>
<td>7.2.6 Disconnect from the Device</td>
<td>69</td>
</tr>
<tr>
<td>7.2.7 Check if a Device Contains a Service</td>
<td>70</td>
</tr>
<tr>
<td>7.2.8 Maximum Transmission Unit for Given Device Service</td>
<td>70</td>
</tr>
<tr>
<td>7.2.9 Write Data to a Given Service and Characteristic</td>
<td>70</td>
</tr>
<tr>
<td>7.2.10 Subscribe to a Given Service and Characteristic</td>
<td>71</td>
</tr>
<tr>
<td>7.3 Provision a Device to a Subnet</td>
<td>72</td>
</tr>
<tr>
<td>7.3.1 Create Network</td>
<td>72</td>
</tr>
<tr>
<td>7.3.2 Create Subnet</td>
<td>72</td>
</tr>
<tr>
<td>7.3.3 Find Non-Provisioned Bluetooth Devices</td>
<td>72</td>
</tr>
<tr>
<td>7.3.4 Provision the Devices</td>
<td>73</td>
</tr>
<tr>
<td>7.3.4.1 In-Band Provisioning</td>
<td>73</td>
</tr>
<tr>
<td>7.3.4.2 OOB Provisioning</td>
<td>73</td>
</tr>
<tr>
<td>7.3.4.2.1 ProvisionerOOBControl</td>
<td>73</td>
</tr>
<tr>
<td>7.4 Add a Node to a Subnet</td>
<td>76</td>
</tr>
<tr>
<td>7.5 Remove a Node from a Subnet</td>
<td>76</td>
</tr>
<tr>
<td>7.6 Connect with a Subnet</td>
<td>77</td>
</tr>
<tr>
<td>7.6.1 Find All Proxies in the Device Range</td>
<td>77</td>
</tr>
<tr>
<td>7.6.2 Get Node Representing a Given Device</td>
<td>77</td>
</tr>
<tr>
<td>7.7 Create a Group in a Given Subnet</td>
<td>77</td>
</tr>
<tr>
<td>7.7.1 Create a Group with a Name</td>
<td>77</td>
</tr>
<tr>
<td>7.7.2 Create a Group with a Name and an AppKey</td>
<td>78</td>
</tr>
<tr>
<td>7.8 Remove Group</td>
<td>78</td>
</tr>
<tr>
<td>7.9 Add a Node to a Group</td>
<td>78</td>
</tr>
<tr>
<td>7.10 Remove a Node from a Group</td>
<td>78</td>
</tr>
</tbody>
</table>
## Contents

7.11 Bind a Model with A Group .................................................................................................................. 79
7.12 Unbind a Model from a Group ............................................................................................................... 79
7.13 Add Subscription Settings to a Model ................................................................................................. 80
  7.13.1 SIG model ......................................................................................................................................... 80
  7.13.2 Vendor Model .................................................................................................................................... 80
7.14 Add Publication Settings to a Model ...................................................................................................... 81
  7.14.1 SIG Model .......................................................................................................................................... 81
  7.14.1.1 Publish via Group Address ........................................................................................................ 81
  7.14.1.2 Publish Directly to the Provisioner .......................................................................................... 81
  7.14.2 Vendor Model ................................................................................................................................... 82
  7.14.2.1 Publish via Group Address ........................................................................................................ 82
  7.14.2.2 Publish Directly to the Provisioner .......................................................................................... 82
7.15 Control Node Functionality .................................................................................................................. 82
  7.15.1 Get Value for a Single Model from the Node .................................................................................... 82
  7.15.2 Get Value for All Specific Models Bound with a Group .............................................................. 83
  7.15.3 Set Value for a Single Model from the Node .................................................................................... 83
  7.15.4 Set Value for All Specific Models Bound with the Group .............................................................. 84
  7.15.5 Control Sensor Models .................................................................................................................... 84
  7.15.5.1 Get Sensor Model Values ............................................................................................................ 84
    7.15.5.1.1 Get Sensor Descriptors from the Node .................................................................................. 84
    7.15.5.1.2 Get Sensor State from the Node .......................................................................................... 85
    7.15.5.1.3 Get Sensor Column from the Node ...................................................................................... 85
    7.15.5.1.4 Get Sensor Series from the Node .......................................................................................... 86
    7.15.5.1.5 Get Sensor Cadence from the Node ...................................................................................... 86
    7.15.5.1.6 Get Sensor Settings from the Node ........................................................................................ 87
    7.15.5.1.7 Get Sensor Setting from the Node .......................................................................................... 87
    7.15.5.1.8 Get Sensor Descriptors from All Nodes in the Group ............................................................ 88
    7.15.5.1.9 Get Sensor State from All Nodes in the Group ....................................................................... 88
    7.15.5.1.10 Get Sensor Series from All Nodes in the Group ................................................................... 89
    7.15.5.1.11 Get Sensor Column from All Nodes in the Group .............................................................. 89
    7.15.5.1.12 Get Sensor Cadence from All Nodes in the Group ............................................................. 90
    7.15.5.1.13 Get Sensor Setting from All Nodes in the Group ................................................................. 90
    7.15.5.1.14 Get Sensor Settings from All Nodes in the Group ............................................................... 91
  7.15.5.2 Set Sensor Setup Model Values .................................................................................................... 92
    7.15.5.2.1 Set Sensor Cadence within the Node ...................................................................................... 92
    7.15.5.2.2 Set Sensor Setting within the Node ........................................................................................ 92
    7.15.5.2.3 Set Sensor Cadence within All Nodes in the Group ............................................................ 93
    7.15.5.2.4 Set Sensor Setting within All Nodes in the Group ............................................................... 94
7.15.6 Register a Local Vendor Model (LocalVendorModel) ...................................................................... 94
  7.15.6.1 Create LocalVendorSettings ....................................................................................................... 95
  7.15.6.2 Create LocalVendorRegistrar ..................................................................................................... 95
  7.15.6.2.1 Register a Local Vendor Model ............................................................................................... 95
  7.15.6.2.2 Unregister a Local Vendor Model .......................................................................................... 95
7.15.7 Local Vendor Model Binding with Application Key ......................................................................... 95
  7.15.7.1 Bind Local Vendor Model with Application Key ......................................................................... 96
  7.15.7.2 Unbind Local Vendor Model from Application Key .................................................................... 96
7.15.8 Manage Notifications from the VendorModel.......................................................................................................................... 96
  7.15.8.1 Sign Up for The Notifications .................................................................................................................................................. 96
    7.15.8.1.1 Notifications Come by Group Address .......................................................................................................................... 96
    7.15.8.1.2 Notifications Come by Node Address .......................................................................................................................... 96
  7.15.8.2 Sign Out from the Notifications ........................................................................................................................................ 97
    7.15.8.2.1 Sign Out from Notifications from the Group Address ................................................................................................. 97
    7.15.8.2.2 Sign Out from Notifications from the Node Address ................................................................................................. 97

7.15.9 Send Value to VendorModel......................................................................................................................................................... 97
  7.15.9.1 Create Implementation of the ControlValueSetVendorModel Protocol .................................................................................. 97
  7.15.9.2 Prepare Message to Send ...................................................................................................................................................... 98
  7.15.9.3 Send Prepared Message to the Single VendorModel on the Node ....................................................................................... 98
  7.15.9.4 Send Prepared Message to the Group .................................................................................................................................. 99

7.16 Sequence Diagrams for Vendor Model Functionality ...................................................................................................................... 100
  7.16.1 LocalVendorModel Initialization ............................................................................................................................................... 100
    7.16.1.1 Create LocalVendorModel .............................................................................................................................................. 100
    7.16.1.2 Register the LocalVendorModel in the Mesh Stack ........................................................................................................ 101
  7.16.2 Configure the VendorModel ...................................................................................................................................................... 102
    7.16.2.1 Bind the VendorModel with the Group ............................................................................................................................ 102
    7.16.2.2 Send Publication Settings to the VendorModel ............................................................................................................. 103
      7.16.2.2.1 Set the Provisioner Address as the Publication Address ............................................................................................. 103
      7.16.2.2.2 Set the Group Address as the Publication Address .................................................................................................. 104
    7.16.2.3 Send Subscription Settings to the VendorModel ......................................................................................................... 105
  7.16.3 Send Message to the Vendor Model on the Node ..................................................................................................................... 106
    7.16.3.1 Send Message Directly to the Node .................................................................................................................................. 106
      7.16.3.1.1 Message without Response ....................................................................................................................................... 106
      7.16.3.1.2 Message with Response ........................................................................................................................................ 108
    7.16.3.2 Send Message via Group Address ................................................................................................................................... 110
      7.16.3.2.1 Message without Response ....................................................................................................................................... 110
      7.16.3.2.2 Message with Response ....................................................................................................................................... 112
  7.16.4 Subscribe to Notifications Published by the Vendor Model From the Node .................................................................................. 114
    7.16.4.1 Messages are Sent Directly to the Provisioner ................................................................................................................... 114
    7.16.4.2 Messages are Sent via Group Address to the Provisioner ............................................................................................. 116
1 Introduction

The iOS and Android (API version 27 or older) Bluetooth LE stacks do not have native support for Bluetooth mesh and therefore devices with these operating systems cannot directly interact with Bluetooth mesh nodes using the Bluetooth mesh advertisement bearer. However, the Bluetooth mesh specification 1.0 also defines a GATT bearer, which enables any Bluetooth LE-capable device to interact with Bluetooth mesh nodes of GATT. iOS and Android (since API version 18) have included support for the Bluetooth GATT layer, and therefore it is possible to implement an iOS or Android application to provision, configure, and interact with Bluetooth mesh networks and nodes. Silicon Labs provides a Bluetooth mesh stack not only for Gecko SoCs and Modules but also for the iOS and Android operating systems, to enable development of Bluetooth mesh applications.

The basic concept is that the iOS and Android stack APIs are used to discover and connect Bluetooth LE devices and exchange data with them over GATT. The Silicon Labs Bluetooth mesh stack is used to manage the Bluetooth mesh-specific operations such as Bluetooth mesh security, device and node management, network, transport, and application layer operations.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Android Value</th>
<th>iOS Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth mesh stack version</td>
<td>1.5.2</td>
<td></td>
</tr>
<tr>
<td>Simultaneous GATT connections</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Mesh bearers</td>
<td>GATT</td>
<td></td>
</tr>
<tr>
<td>Supported node types</td>
<td>Proxy</td>
<td>Proxy</td>
</tr>
<tr>
<td></td>
<td>Relay</td>
<td>Relay</td>
</tr>
<tr>
<td></td>
<td>Low Power</td>
<td>Low Power</td>
</tr>
<tr>
<td>Mesh limits</td>
<td>(see section 6.1.3 Set Up Mesh Limits)</td>
<td>(see section 7.1.3 Set Up Mesh Limits)</td>
</tr>
<tr>
<td>Feature</td>
<td>Android Value</td>
<td>iOS Value</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------------------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>Supported mesh models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic OnOff Server 0x1000</td>
<td></td>
<td>Generic OnOff Server 0x1000</td>
</tr>
<tr>
<td>Generic OnOff Client 0x1001</td>
<td></td>
<td>Generic OnOff Client 0x1001</td>
</tr>
<tr>
<td>Generic Level Server 0x1002</td>
<td></td>
<td>Generic Level Server 0x1002</td>
</tr>
<tr>
<td>Generic Level Client 0x1003</td>
<td></td>
<td>Generic Level Client 0x1003</td>
</tr>
<tr>
<td>Generic Default Transition Time Server 0x1004</td>
<td></td>
<td>Generic Default Transition Time Server 0x1004</td>
</tr>
<tr>
<td>Generic Default Transition Time Client 0x1005</td>
<td></td>
<td>Generic Default Transition Time Client 0x1005</td>
</tr>
<tr>
<td>Generic Power OnOff Server 0x1006</td>
<td></td>
<td>Generic Power OnOff Server 0x1006</td>
</tr>
<tr>
<td>Generic Power OnOff Client 0x1008</td>
<td></td>
<td>Generic Power OnOff Client 0x1008</td>
</tr>
<tr>
<td>Generic Battery Server 0x100C</td>
<td></td>
<td>Generic Battery Server 0x100C</td>
</tr>
<tr>
<td>Generic Battery Client 0x100D</td>
<td></td>
<td>Generic Battery Client 0x100D</td>
</tr>
<tr>
<td>Light Lightness Server 0x1300</td>
<td></td>
<td>Light Lightness Server 0x1300</td>
</tr>
<tr>
<td>Light Lightness Client 0x1302</td>
<td></td>
<td>Light Lightness Client 0x1302</td>
</tr>
<tr>
<td>Light CTL Server 0x1303</td>
<td></td>
<td>Light CTL Server 0x1303</td>
</tr>
<tr>
<td>Light CTL Client 0x1305</td>
<td></td>
<td>Light CTL Client 0x1305</td>
</tr>
<tr>
<td>Light CTL Temperature Server 0x1306</td>
<td></td>
<td>Light CTL Temperature Server 0x1306</td>
</tr>
<tr>
<td>Supported GATT services</td>
<td>Provisioning, Proxy</td>
<td>Provisioning, Proxy</td>
</tr>
</tbody>
</table>

**Table 1-2. Open Source Licenses Used**

<table>
<thead>
<tr>
<th>Feature</th>
<th>License</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL</td>
<td>See <a href="#">here</a></td>
<td>Used for AES and ECDH and other cryptographic algorithms.</td>
</tr>
<tr>
<td>GSON (Android only)</td>
<td><a href="#">Apache License 2.0</a></td>
<td>Used to store and load the Bluetooth mesh and device database to the Android secure storage.</td>
</tr>
</tbody>
</table>
2 Prerequisites for Development

Before you can start developing Bluetooth mesh iOS or Android applications the following prerequisites must be met.

1. Basic understanding of Bluetooth LE and Bluetooth mesh 1.0 specifications and technology.
   1. Silicon Labs Bluetooth mesh learning center
   2. Silicon Labs Bluetooth mesh technology white paper
   3. Bluetooth mesh profile 1.0 specification
   4. Bluetooth mesh model 1.0 specification (application layer)
   5. Bluetooth 5.0 core specification
2. Simplicity Studio and Bluetooth mesh SDK 1.5.2 or newer are installed.
   1. Download
   2. Getting started
3. You have a Blue Gecko SoC (SLWSTK6020B) or module WSTKs to act as Bluetooth mesh nodes.
   3. Recommended SoCs and Modules are: EFR32BG13 or EFR32BG12 SoCs or BGM13P or BGM13S module because they have 512kB to 1024kB of flash available.
   4. SLWSTK6020B User Guide

2.1 iOS
1. You have installed the Bluetooth Mesh ADK. It is available through the Simplicity Studio Package Manager.
   1. Bluetooth Mesh iOS Framework:
      ```
      /Applications/Simplicity\ Studio.app/Contents/Eclipse/
      developer/sdks/blemesh/v1.5/app/bluetooth/ios/BluetoothMesh.framework
      ```
   2. Bluetooth Mesh iOS API Documentation:
      ```
      /Applications/Simplicity\ Studio.app/Contents/Eclipse/
      developer/sdks/blemesh/v1.5/app/bluetooth/ios/docs/index.html
      ```
   3. Source code of the iOS reference application:
      ```
      /Applications/Simplicity\ Studio.app/Contents/Eclipse/
      developer/sdks/blemesh/v1.5/app/bluetooth/ios_application
      ```
      The application is available in iOS AppStore: Bluetooth Mesh by Silicon Labs
   2. Xcode 10 or newer is installed.
   3. This ADK supports iOS versions 11, 12, and 13.
   4. The following devices have been tested with this ADK:
      - iPad mini 4
      - iPad (5th generation)
      - iPad 9.7
      - iPad Pro 10.5
      - iPhone X
   5. You understand Bluetooth LE operation on iOS.
      1. Bluetooth LE documentation for iOS
      2. Bluetooth LE API documentation

2.2 Android
1. You have installed the Bluetooth Mesh ADK. It is available through the Simplicity Studio Package Manager.
   1. Bluetooth Mesh Android Framework:
      ```
      /Applications/Simplicity\ Studio.app/Contents/Eclipse/
      developer/sdks/blemesh/v1.5/app/bluetooth/android/BluetoothMesh
      ```
2. Bluetooth Mesh Android API Documentation:
   /Applications/Simplicity\ Studio.app/Contents/Eclipse/developer/sdks/blemesh/v1.5/app/bluetooth/android/docs/index.html

3. Source code of the Android reference application:
   /Applications/Simplicity\ Studio.app/Contents/Eclipse/developer/sdks/blemesh/v1.5/app/bluetooth/android_application

   The application is available in Google Play: Bluetooth Mesh by Silicon Labs

2. Android Studio is installed.
   1. Download
   2. Instruction

3. You have an Android phone or tablet preferably running Android 6.0 or newer.

4. The following devices have been tested with the ADK:
   - Lenovo TB-8704X
   - Samsung SM-T710
   - Samsung SM-T580
   - One Plus 6
   - Xiaomi Mi6

5. You understand Bluetooth LE operation on Android as well as the JNI interface.
   1. Bluetooth LE documentation for Android
   2. Bluetooth LE API documentation
   3. Android JNI documentation
3 Contents of Bluetooth Mesh for iOS and Android ADK

3.1 The Bluetooth Mesh Stack library

The Bluetooth mesh protocol stack is provided as a pre-compiled library.

3.1.1 iOS

Bluetooth Mesh Swift/Objective-C API: iOS applications are developed with Swift programming language. The Silicon Labs Bluetooth Mesh Objective-C API provides an API for the application to interface with the Bluetooth Mesh stack library. Silicon Labs Bluetooth Mesh Objective-C API can be used with applications written in Objective-C language and Swift language.

The library is available as Release and Debug builds, found in the “Release” and “Debug” directories located in “app\bluetooth\ios” directory. The debug build of the library provides more logs that can be helpful when resolving issues with the Bluetooth mesh stack library, so it should be used only in special cases. Other than during debugging, the Release build should always be used.

3.1.2 Android

JNI Interface: The JNI interface provides the necessary abstraction between the Bluetooth mesh stack library and the application.

Bluetooth Mesh Java API: Android applications are developed with Java programming language. The Silicon Labs Bluetooth Mesh Java API provides an API for the application to interface with the Bluetooth mesh stack library. As well as providing access to the Bluetooth mesh stack library, the Java API also contains the necessary helper classes for Bluetooth mesh devices, networks, groups, models, and so on.

The library is available as Release and Debug builds, found in the “app\bluetooth\android” directory. The debug build of the library provides more logs that can be helpful when resolving issues with the Bluetooth mesh stack library, so it should be used only in special cases. Other than during debugging, the Release build should always be used.

3.2 Bluetooth Mesh Network and Device Database

The Bluetooth mesh device and network database contains all the information the provisioner stores about devices and the network, including security keys, device addresses, supported elements, models and so on.

The device database is stored in the secure content of the application in AES-encrypted JSON file format.

3.3 Reference Application Source Code

The Bluetooth mesh by Silicon Labs reference application is provided in source code as a part of the delivery and can be used as a reference implementation for application development.

3.4 Documentation

The delivery also contains API documentation for the Bluetooth mesh stack.
4 Getting Started with Development

The first step is to set up a project.

4.1 iOS
1. Copy BluetoothMesh.framework to the folder with the iOS project.
2. Add BluetoothMesh.framework reference to the project.
   1. Open main target in your project.
   2. Go to General view.
   3. Add BluetoothMesh.framework to Embedded Binaries.
Getting Started with Development

Choose items to add:
- BluetoothMeshDemo
- BluetoothMeshDemo.xcodeproj
- BluetoothMeshDemoTests

Choose options for adding these files:
- Destination: Copy items if needed
- Added folders: Create groups

Finish
3. BluetoothMesh.framework should be visible in Embedded Binaries and Linked Frameworks and Libraries.

   1. Select main target in project.
   2. Go to Build Settings view.
   4. Set Enable Bitcode to ‘No’.

4.2 Android
   1. Install the Android Studio.
   2. Unzip the Bluetooth Mesh for Android ADK.
   3. Open the Android studio.
   4. Create a new project.
      1. Create the project for Phone and Tablet.
      2. Select at least API 23: Android 6.0 (Marshmallow).
3. Create a project such as Empty Activity project.

5. In “Project Structure” add new Module (execute steps 5 to 9 for “ble_mesh-android_api-v2_high-release.aar” and “ble_mesh-android_api-v2_low-release.aar”).

6. Select “Import .JAR or .AAR Package”.

7. Specify path to .aar file and proceed.

8. In “Dependencies” tab click **Add** and select “Module dependency”.

9. Select module and proceed.

10. In build gradle add gson dependency: implementation 'com.google.code.gson:gson:2.8.5'
11. Create a new BluetoothMesh object as shown below.

```java
import com.silabs.bluetooth_mesh.BluetoothMesh;
import com.silabs.bluetooth_mesh.configuration.BluetoothMeshConfiguration;

public class MainActivity extends AppCompatActivity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        BluetoothMeshConfiguration configuration = new BluetoothMeshConfiguration();
        BluetoothMesh.initialize(getApplicationContext(), configuration);

        BluetoothMesh bluetoothMesh = BluetoothMesh.getInstance();
    }
}
```

12. Compile the project and make sure it compiles without errors.
5 Bluetooth Mesh Structure Overview

The API is provided with support objects that help the user manage the Bluetooth mesh network. These are:

1. Network – The main container in the mesh structure. Network is the owner of nodes and subnets. See Figure 5-2, Network Structure.
2. Subnet – A specific subnet belongs only to one network. A subnet is the owner of groups.
3. Node – A node can be added to many subnets from the network. A node is owner of elements and models. One node can exist only in one network. See Figure 5-1, Node Structure.
4. Element – A part of a node.
5. Model – A part of an element.
6. Group – A group can be added to only one subnet.
   • Many nodes can be bound to a group.
   • Many models can be bound to a group.
   • Subscription/publication settings can be added for many models.

Application developers are responsible for keeping track of any changes in the Bluetooth mesh structure. Objects are mutable and will change over time.
6 Bluetooth Mesh API Reference for iOS

To control base of Bluetooth mesh structure you must be familiar with a few of the most important layers of the Bluetooth mesh API:
1. Bluetooth connection
2. Provision session
3. Proxy connection
4. Node configuration
5. Model configuration (subscription and publication settings)
6. Control model and group

All are described in the next part of this document.

All iOS classes and interfaces are named the same as the Android classes and interfaces plus the prefix ‘SBM,’ for example Node -> SBMNode, Network -> SBMNetwork.

6.1 Initializing the BluetoothMesh

6.1.1 BluetoothMesh

The Bluetooth mesh structure is represented by singleton object of the BluetoothMesh class. This is a main entry point to the library and it gives access to all other objects within the library.

BluetoothMesh must be configured before using the library. Configuration can be provided using the BluetoothMeshConfiguration class.

In the base configuration supported vendor models are not needed.

To get the instance call the following for each platform noted

SBMBluetoothMesh.sharedInstance()

let configuration = SBMBluetoothMeshConfiguration(localVendorModels: [], andLogger: SBMLogger.sharedInstance())

6.1.2 Set Up Supported Vendor Models

Supported vendor models can be set using BluetoothMeshConfiguration during its initialization. To do that you must know a specification for each vendor model. It should be delivered by the external provider.

let vendorModel = SBMLocalVendorModel.init(vendorCompanyIdentifier: A, vendorAssignedModelIdentifier: B)

// A - vendor company identifier. Need to be the same as vendor company identifier from the SBMVendorModel from the SBMNode which will be controlled.
// B - vendor assigned model identifier. Need to be the same as vendor assigned model identifier from the SBMVendorModel from the SBMNode which will be controlled.

let configuration = SBMBluetoothMeshConfiguration(localVendorModels: [vendorModel])
SBMBluetoothMesh.sharedInstance().initialize(configuration)

6.1.3 Set Up Mesh Limits

Limits for the Bluetooth Mesh database can be set using BluetoothMeshConfiguration during its initialization.

Warning: Changing configuration limits between each application launch may corrupt the database.

let limits = SBMBluetoothMeshConfigurationLimits()

limits.networks // maximum number of network keys that can be created - MAX is 7
limits.groups // maximum number of application keys that can be created - MAX is 8. Be aware that it is possible to create maximum 16,128 groups when groups share application keys between each other.
limits.nodes // maximum number of nodes that can be provisioned - MAX is 255
limits.nodeNetworks // maximum number of network keys that a single node can be added to - MAX is 7
limits.nodeGroups // maximum number of application keys that a single node can be added to - MAX is 8
limits.rplSize // maximum number of nodes that can be communicated with - MAX is 255
limits.segmentedMessagesReceived // maximum number of concurrent segmented messages being received - MAX is 255
limits.segmentedMessagesSent // maximum number of concurrent segmented messages being sent - MAX is 255
limits.provisionSession // maximum number of parallel provisioning sessions - MAX is 1

let configuration = SBMBluetoothMeshConfiguration(localVendorModels: [], limits: limits, andLogger: SBMLogger.sharedInstance())
SBMBluetoothMesh.sharedInstance().initialize(configuration)

When a specific limit is not set, a default value is assigned:

networks = 4;
groups = 8;
nodes = 255;
nodeNetworks = 4;
nodeGroups = 4;
rplSize = 32;
segmentedMessagesReceived = 4;
segmentedMessagesSent = 4;
provisionSessions = 1;

### 6.2 Request IV Index Update

The IV Index of the network can be retrieved using the following method.

```swift
let network: SBMNetwork
let currentIvIndex = network.currentIvIndex()
```

An IV Index update can be requested by calling:

```swift
let network: SBMNetwork
// result - error if request failed, nil otherwise
let result = network.requestIvIndexUpdate() { index, state in
    // called when IV Index Update procedure has transitioned to a new state
    // index - current IV Index
    // state - procedure state: can be either normal or in progress
}
```

Note that this request may fail for the following reasons:
- An IV Index update is already ongoing
- Not enough time has passed since the previous update
- Node is not a member of the primary subnet
- Node is not sending secure network beacons

### 6.3 Set Up Bluetooth Layer (ConnectableDevice)

The BluetoothMesh iOS API provides a layer that helps manage iOS Bluetooth LE. The developer must provide an implementation of the SBMConnectableDevice class, which is a connection between CBPeripheral from the CoreBluetooth and a layer responsible for
communication with the Bluetooth mesh structure. SBMConnectableDeviceDelegate from the SBMConnectableDevice is set by BluetoothMesh framework.

6.3.1 Device Advertisement Data

Advertisement data can be obtained from CBCentralManagerDelegate.

```swift
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral, advertisementData: [String : Any], rssi RSSI: NSNumber) {
    // provide logic to handle advertisement data updates
}

func advertisementData() -> [AnyHashable : Any] {
    return advertisementData
}
```

6.3.2 Device UUID

The device UUID comes from advertisement data.

Note that Device UUID is available only for non-provisioned Bluetooth mesh-capable devices.

```swift
func uuid() -> Data {
    let serviceData = advertisementData [CBAdvertisementDataServiceDataKey]
    let data = serviceData[CBUUID(string: "1827")]
    return data.subdata(with: NSMakeRange(0, 16))
}
```

6.3.3 Device Name

The device name comes from CBPeripheral.

```swift
let peripheral: CBPeripheral
// (…)
// SBMConnectableDevice
func name() -> String {
    return peripheral.name
}

### 6.3.4 Device Connection State

The Device Connection State is a Boolean value that determines whether a device is connected.

```swift
let peripheral: CBPeripheral
// (…)

// SBMConnectableDevice
func isConnected() -> Bool {
    return peripheral.state == .connected
}
```

### 6.3.5 Connect to the Device

```swift
let centralManager: CBCentralManager
let peripheral: CBPeripheral
let callback: SBMConnectableDeviceConnectionCallback
// (…)

// SBMConnectableDevice
func connect(_ completion: @escaping SBMConnectableDeviceConnectionCallback) {
    callback = completion
    // call completion callback after establish Bluetooth connection by application
    centralManager.connect(peripheral)
}

// CBCentralManagerDelegate
func centralManager(_ central: CBCentralManager, didConnect peripheral: CBPeripheral) {
    callback(self, true)
}
```

### 6.3.6 Disconnect from the Device

```swift
let centralManager: CBCentralManager
let peripheral: CBPeripheral
let callback: SBMConnectableDeviceConnectionCallback
// (…)

// SBMConnectableDevice
func disconnect(_ completion: @escaping SBMConnectableDeviceConnectionCallback) {
    callback = completion
    // call completion callback after break Bluetooth connection by application
    centralManager.cancelPeripheralConnection(peripheral)
}

// CBCentralManagerDelegate
func centralManager(_ central: CBCentralManager, didDisconnectPeripheral peripheral: CBPeripheral, error: Error?) {
    callback(self, true)
}
6.3.7 Check if a Device Contains a Service

The Bluetooth mesh framework sometimes needs to check if a Bluetooth device contains a needed service for its operation.

```swift
// SBMConnectableDevice
func hasService(_ service: Data, completion: @escaping SBMConnectableDeviceHasServiceCallback) {
    // service argument contains Service UUID
    // Check if CBPeripheral contains a given service
}
```

6.3.8 Maximum Transmission Unit for Given Device Service

Maximum Transmission Unit is the size of both payload and header. iOS reports only the payload size, therefore it is required to add header size.

```swift
let peripheral: CBPeripheral
// (…)

// SBMConnectableDevice
func mtu(forService serviceUuid: Data, characteristic: Data, completion: @escaping SBMConnectableDeviceMTUCallback) {
    let headerSize = 3
    let mtu = UInt(peripheral.maximumWriteValueLength(for: .withoutResponse) + headerSize)
    completion(self, serviceUuid, characteristic, mtu)
}
```

6.3.9 Write Data to a Given Service and Characteristic

Write method is a function where the Bluetooth mesh framework sends bytes to the ConnectableDevice.

```swift
let peripheral: CBPeripheral
// (…)

// SBMConnectableDevice
func write(_ data: Data, service: String, characteristic: String, completion: @escaping SBMConnectableDeviceOperationCallback) {
    let characteristic = peripheral.services?.first {
        $0.uuid.uuidString == service
    }?.characteristics?.first {
        $0.uuid.uuidString == characteristic
    }

    if characteristic != nil {
        peripheral.writeValue(data, for: characteristic, type: .withoutResponse)
        // It is very important to write data without response
        completion(self,
            characteristic.service.uuid.data,
            characteristic.uuid.data,
            true)
    } else {
        completion(connectableDevice,
            CBUUID(string: service).data,
            CBUUID(string: characteristic).data,
```
6.3.10 Subscribe to a Given Service and Characteristic

```swift
let peripheral: CBPeripheral
let delegate: SBMConnectableDeviceDelegate // delegate is set by BluetoothMesh framework, do NOT override it
let callback: SBMConnectableDeviceOperationCallback

// SBMConnectableDevice
func subscribe(forService service: String, characteristic: String, completion: @escaping SBMConnectableDeviceOperationCallback) {
    // Need to provide logic to discover given characteristic for the device
    let characteristic = peripheral.services?.first {
        $0.uuid.uuidString == service
    }?.characteristics?.first {
        $0.uuid.uuidString == characteristic
    }

    if characteristic != nil {
        callback = completion
        peripheral.setNotifyValue(true, for: characteristic)
    } else {
        completion(self,
                   CBUUID(string: service).data,
                   CBUUID(string: characteristic).data,
                   false)
    }
}

// CBCentralManagerDelegate
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic: CBCharacteristic, error: Error?) {
    self.delegate?.didUpdate(characteristic.value,
                               forDevice: self,
                               service: characteristic.service.uuid.data,
                               characteristic: characteristic.uuid.data)
}

// CBCentralManagerDelegate
func peripheral(_ peripheral: CBPeripheral, didUpdateNotificationStateFor characteristic: CBCharacteristic, error: Error?) {
    callback(self,
             characteristic.service.uuid.data,
             characteristic.uuid.data,
             error == nil && characteristic.isNotifying)
}
```

6.4 Provision a Device to a Subnet

First you must discover a Bluetooth device that is compatible with Bluetooth mesh networking. This device will be in a non-provisioned state. It is not possible to provision a device a second time without a factory reset.
Before you can provision a device, you must prepare a Network with a Subnet. To start a provisioning session, choose a subnet to which to provision the device. The steps are described below.

### 6.4.1 Create Network

You can create a network by specifying only its name (address and IV Index are assigned default values).

```swift
let network = SBMBluetoothMesh.sharedInstance().createNetwork("Network name")
```

You can also create a network by specifying its address and IV Index. Address 0 means that default value (0x2001) will be assigned.

```swift
let network = SBMBluetoothMesh.sharedInstance().createNetwork(withName: "Network name", address: address, ivIndex: ivIndex)
```

### 6.4.2 Create Subnet

To create a subnet with a randomly generated network key, pass nil as the netKey parameter.

```swift
let subnet = try? network.createSubnet(withName: "Subnet name", netKey: nil)
```

To create a subnet with a specified network key, create SBMNetworkKey and then pass it as the netKey parameter.

```swift
let key = SBMNetworkKey(key: data, index: index)
// data – 16 randomly generated bytes
// index – network key index, value of type UInt16
// key can be nil if specified data or index is incorrect
let subnet = try? network.createSubnet(withName: "Subnet name", netKey: key!)
```

### 6.4.3 Find Non-Provisioned Bluetooth Devices

Find a device that is compatible with Bluetooth Mesh networking and is not provisioned. A non-provisioned device will advertise its provisioning service. To represent a non-provisioned device, you should use a class that implements ConnectableDevice.

```swift
let centralManager: CBCentralManager
// (...)

func startDiscovering(services: [CBUUID], centralManager: CBCentralManager) {
    centralManager.scanForPeripherals(withServices: services,
            options: [CBCentralManagerScanOptionAllowDuplicatesKey : true])
}

startDiscovering(services: [CBUUID(string: SBMProxyConnection.meshProvisioningServiceUUID())])
// CBCentralManagerDelegate
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
    // BluetoothDevice class implementation of SBMConnectableDevice interface.
    let connectableDevice = BluetoothDevice(withManager: self,
            peripheral: peripheral,
            advertisement: advertisementData)

    // provide logic to store all Mesh capable devices
}
```

### 6.4.4 Provision the Device

During device provisioning, ensure that at least one device is provisioned as a proxy. The Bluetooth mesh network can only connect to devices with active proxy.

The node returned from the provision connection callback is a Bluetooth mesh counterpart of the ConnectableDevice.
The following examples show in-band provisioning and out-of-band (OOB) provisioning.

### 6.4.4.1 In-Band Provisioning

```swift
let connectableDevice: SBMConnectableDevice
let subnet: SBMSubnet

// (...) let provisionerConnection = SBMProvisionerConnection(for: connectableDevice, subnet: subnet) // configuration: specifies whether proxy and nodeIdentity are enabled // configuration can be set to nil if provisioning should not do any setup let configuration = SBMProvisionerConfiguration(proxyEnabled: true, nodeIdentityEnabled: true) // parameters: specifies device parameters, including attention timer to use // parameters can be nil to provision without attention timer and specific address let parameters = SBMProvisionerParameters(address: 5, elements: 2, attentionTimer: 1) // retry count: specifies how many times connection should be retried in case of failure let retryCount = 1 // (...) provisionerConnection.provision(withConfiguration: configuration, parameters: parameters, retryCount: retryCount) { (connection, node, error) in // handle result of provisioning }
```

### 6.4.4.2 OOB Provisioning

```swift
let connectableDevice: SBMConnectableDevice
let subnet: SBMSubnet
let provisionerOOB: ProvisionerOOB // ProvisionerOOB is an implementation of SBMProvisionerOOB protocol which have to be provided by developer.

// (...) let provisionerConnection = SBMProvisionerConnection(for: connectableDevice, subnet: subnet) provisionerConnection.provisionerOOB = provisionerOOB // (...) provisionerConnection.provision(withConfiguration: nil, parameters: nil, retryCount: 1) { (connection, node, error) in // handle result of provisioning }
```

### 6.4.4.2.1 SBMProvisionerOOB Protocol

This protocol must be implemented in order to use OOB provisioning for a Bluetooth device.

The following is an example of implementing SBMProvisionerOOB protocol. Detailed explanations are provided after the example.

```swift
class ProvisionerOOB: NSObject, SBMProvisionerOOB {
    var delegate: SBMProvisionerOOBDelegate?

    func ecPublicKeyAllowed() -> SBMProvisionerOOBECPublicKeyAllowed {
        return .yes
    }

    func allowedAuthMethods() -> SBMProvisionerOOBAuthMethods {
        return SBMProvisionerOOBAuthMethods(inputOOB.rawValue: SBMProvisionerOOBAuthMethods.inputOOB.rawValue | rawValue: SBMProvisionerOOBAuthMethods.inputOOB.rawValue) |
```
SBMProvisionerOOBAAllowedAuthMethods.outputOOB.rawValue | SBMProvisionerOOBAAllowedAuthMethods.staticOOB.rawValue)!}

    func allowedOutputActions() -> SBMProvisionerOOBAAllowedOutputActions {
        return SBMProvisionerOOBAAllowedOutputActions(rawValue: SBMProvisionerOOBAAllowedOutputActions.alpha.rawValue | SBMProvisionerOOBAAllowedOutputActions.beep.rawValue | SBMProvisionerOOBAAllowedOutputActions.blink.rawValue | SBMProvisionerOOBAAllowedOutputActions.numeric.rawValue | SBMProvisionerOOBAAllowedOutputActions.vibrate.rawValue)!
    }

    func allowedInputActions() -> SBMProvisionerOOBAAllowedInputActions {
        return SBMProvisionerOOBAAllowedInputActions(rawValue: SBMProvisionerOOBAAllowedInputActions.alpha.rawValue | SBMProvisionerOOBAAllowedInputActions.numeric.rawValue | SBMProvisionerOOBAAllowedInputActions.push.rawValue | SBMProvisionerOOBAAllowedInputActions.twist.rawValue)!
    }

    func minLengthOfOOBData() -> UInt8 {
        return 0
    }

    func maxLengthOfOOBData() -> UInt8 {
        return 0
    }

    func oobPkeyRequest(_ uuid: UUID, algorithm: UInt8, oobPkeyType: UInt8) -> SBMProvisionerOOBResult {
        var oobData: Data
        //{}
        // Developer has to provide implementation to get needed oobData.
        //{}
        self.delegate?.providePublicKey(oobData!, for: uuid)
        return .success //if success
    }

    func outputRequest(_ uuid: UUID, outputAction: SBMProvisionerOOBOutputAction, outputSize: UInt8) -> SBMProvisionerOOBResult {
        var oobData: Data
        //{}
        // Developer has to provide implementation to get needed oobData.
        //{}
        self.delegate?.provideAuthData(oobData!, for: uuid)
        return .success //if success
    }

    func authRequest(_ uuid: UUID) -> SBMProvisionerOOBResult {
        var oobData: Data
        //{}
        // Developer has to provide implementation to get needed oobData.
func inputDisplay(_ uuid: UUID, inputAction: SBMProvisionerOOBInputAction, authData: Data) -> SBMProvisionerOOBResult {
    //(...)
    // Developer has to provide implementation to display authData.
    //(...)
    return .success //if success
}

Explanation:

var delegate: SBMProvisionerOOBDelegate?

Delegate to provide an out-of-band public key or out-of-band authentication data to the Bluetooth device. This is set by the SBMBluetoothMesh library; do NOT override it.

func ecPublicKeyAllowed() -> SBMProvisionerOOBECPublicKeyAllowed

Used to determine whether OOB EC public key is allowed or not during OOB provisioning.

func allowedAuthMethods() -> SBMProvisionerOOBAAllowedAuthMethods

Determines the allowed authentication methods during OOB provisioning.

dfunc allowedOutputActions() -> SBMProvisionerOOBAllowedOutputActions

Determines the allowed output actions during OOB provisioning.

func allowedInputActions() -> SBMProvisionerOOBAllowedInputActions

Determines the allowed input actions during OOB provisioning.

func minLengthOfOOBData() -> UInt8

Returns a value of type UInt8 that is then used to determine the minimum allowed input/output OOB data length during OOB provisioning. A value of 0 is interpreted as default value of 1 when setting OOB authentication requirements.

func maxLengthOfOOBData() -> UInt8

Returns a value of type UInt8 that is then used to determine the maximum allowed input/output OOB data length during OOB provisioning. A value of 0 is interpreted as default value of 8 when setting OOB authentication requirements.
func oobPkeyRequest(_ uuid: UUID, algorithm: UInt8, oobPkeyType: UInt8) -> SBMProvisionerOOBResult

Optional method called when the platform requests an out-of-band public key.

func outputRequest(_ uuid: UUID, outputAction: SBMProvisionerOOBOutputAction, outputSize: UInt8) -> SBMProvisionerOOBResult

Optional method called when the platform requests an out-of-band output authentication data.

func authRequest(_ uuid: UUID) -> SBMProvisionerOOBResult

Optional method called when the platform requests an out-of-band static authentication data.

func inputDisplay(_ uuid: UUID, inputAction: SBMProvisionerOOBInputAction, authData: Data) -> SBMProvisionerOOBResult

Optional method called when the platform requests an out-of-band input authentication data.

**SBMProvisionerOOBDelegate**

Use this delegate from the SBMProvisionerOOB protocol if you need to provide any authentication data to the Bluetooth device during provisioning. Do NOT override this delegate in the SBMProvisionerOOB. It is set by the SBMBluetoothMesh library.

func providePublicKey(…)

Provides an out-of-band device public key of a device to the Bluetooth device. Call this method after the platform has requested an out-of-band public key for a device.

func provideAuthData(…)

Provides out-of-band authentication data of a device to the Bluetooth device. Call this method after the platform has requested out-of-band static authentication data or output authentication data.

### 6.5 Add a Node to Another Subnet

Note: To perform this action you must have established a connection with a subnet that already has access to this node.

During the device provisioning session, the node is added to a subnet. It is possible to add a node to multiple subnets from the same network. Below is an example of how to add a node to another subnet.

```
let node: SBMNode
let subnet: SBMSubnet
let control = SBMNodeControl.init(node: node)

// (…)
control.add(to: subnet, successCallback: {
  //handle success callback
}, errorCallback: {
  //handle error callback
})
```

### 6.6 Remove a Node from a Subnet

Note: To perform this action you must have established a connection with a subnet that already has access to this node.

It is possible to remove a node from a subnet. Be aware that you can lose access to the node irreversibly if you remove it from its last subnet.
let node: SBMNode
let subnet: SBMSubnet
let control = SBMNodeControl.init(node: node)

// (...)

control.remove(from: subnet, successCallback: {
    // handle success callback
}, errorCallback: {
    // handle error callback
})

6.7 Connect with a Subnet

You can only be connected with one subnet at a time.

6.7.1 Find All Proxies in the Device Range

Find devices that are compatible with Bluetooth mesh networking and were provisioned as proxies. A provisioned proxy device will advertise its proxy service. To represent a provisioned proxy device, you should use a class that extends ConnectableDevice.

```swift
func startDiscovering(services: [CBUUID], centralManager: CBCentralManager) {
    centralManager.scanForPeripherals(withServices: services,
        options: [CBCentralManagerScanOptionAllowDuplicatesKey : true])
}
```

```swift
startDiscovering(services: [CBUUID(string: SBMProxyConnection.meshProxyServiceUUID ())])
```

```swift
//CBCentralManagerDelegate
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
    // BluetoothDevice class implementation of SBMConnectableDevice interface.
    let connectableDevice = BluetoothDevice(withManager: self,
        peripheral: peripheral,
        advertisement: advertisementData)

    //provide logic to store all proxy nodes
}
```

6.7.2 Get Node Representing a Given Device

To retrieve a Bluetooth mesh node counterpart of the ConnectableDevice use ConnectableDeviceHelper.

```swift
let connectableDevice: SBMConnectableDevice

// (...)

let node: SBMNode = SBMConnectableDeviceHelper.node(connectableDevice)
```

6.7.3 Get Node Representing a Given Device in a Specific Subnet

To retrieve a Bluetooth mesh node counterpart of the SBMConnectableDevice that is part of the subnet use SBMConnectableDeviceHelper.

```swift
let subnet: SBMSubnet
let connectableDevice: SBMConnectableDevice

// (...)
```
let node: SBMNode = SBMConnectableDeviceHelper.node(connectableDevice, in: subnet)

### 6.8 Create a Group in a Given Subnet

let subnet: SBMSubnet

// (...)

// application key that group should be assigned
// data - 16 randomly generated bytes
// index - application key index, value of type UInt16
// subnet - subnet which should be bound to application key
// key can be nil if specified data or index is incorrect
let applicationKey = SBMApplicationKey(key: data, index: index, for: subnet)
// application key can be nil if group should be assigned random application key
// address that group should be assigned
// address can be nil if group should be assigned first available address
let address = 0xC005

let group = subnet.createGroup(withName: "Group name", appKey: applicationKey, address: address)

### 6.9 Remove Group

Note: To perform this action you must have an established a connection with a subnet that already has access to this group.

let group: SBMGroup

// (...)
group.remove(callback: { group in
    //handle success callback
}, errorCallback: { group, result, error in
    //handle error callback
})

### 6.10 Add a Node to a Group

Note: To perform this action you must have an established connection with a subnet that already has access to this group. The node must already be added to the same subnet.

let node: SBMNode
let group: SBMGroup

//(...)  

let control = SBMNodeControl.init(node: node)
control.bind(to: group, successCallback: {
    //handle success callback
}, errorCallback: {
    //handle error callback
})

### 6.11 Remove a Node from a Group

Note: To perform this action you must have an established connection with a subnet that already has access to this group. The node must already be added to the same subnet.

let node: SBMNode
let group: SBMGroup
let control = SBMNodeControl(node: node)

control.unbind(from: group, successCallback: {
    //handle success callback
}, errorCallback: {
    //handle error callback
})

### 6.12 Bind a Model with A Group

Note: To perform this action you must have established a connection with a subnet that already has access to this group. Before this step, a node containing the model must already be added to the corresponding subnet and group.

```swift
let model = SBMModel
let group = SBMGroup

// (...) let binder = SBMFunctionalityBinder(group: group)

binder.bindModel(model, successCallback: { bindedModel, bindedGroup in
    //handle success callback
}, errorCallback: { modelToBind, groupToBind, error in
    //handle error callback
})
```

### 6.13 Unbind a Model from a Group

Note: To perform this action you must have established a connection with a subnet that already has access to this group. Before this step, a node containing the model must already be added to the corresponding subnet.

```swift
let model = SBMModel
let group = SBMGroup

// (...) let binder = SBMFunctionalityBinder(group: group)

binder.unbindModel(model, successCallback: { unbindedModel, unbindedGroup in
    //handle success callback
}, errorCallback: { modelToUnbind, groupToUnbind, error in
    //handle error callback
})
```

### 6.14 Add Subscription Settings to a Model

Note: To perform this action you must have established a connection with a subnet that already has access to this model. Before this step, a node containing the model must already be added to the corresponding subnet and group.

#### 6.14.1 SIG Model

```swift
let sigModel = SBM SigModel
let group = SBMGroup

// (...) let subscriptionSettings = SBMSubscriptionSettings(group: group)
```
let subscriptionControl = SBMSubscriptionControl(model: sigModel)

subscriptionControl.add(subscriptionSettings, successCallback: { subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})

6.14.2 Vendor Model

let vendorModel: SBMVendorModel
let group: SBMGroup

// (...)  

let subscriptionSettings = SBMSubscriptionSettings(group: group)
let subscriptionControl = SBMSubscriptionControl(vendorModel: vendorModel)

subscriptionControl.add(subscriptionSettings, successCallback: { subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})

6.15 Add Publication Settings to a Model

Note: To perform this action you must have established a connection with a subnet that already has access to this model. Before this step, a node containing the model must already be added to the corresponding subnet and group.

To allow a switch node to operate properly on the group publication settings must be set up on this model with a given group. To activate notifications from the model publication settings must be configured. The model has to know the address to which it should send notifications. Without this step messages can only be received from the model with GET/SET calls; automatic notifications cannot be received. This step is important for both SIG Models and Vendor Models.

6.15.1 SIG Model

6.15.1.1 Publish via SBMGroup Address

let sigModel: SBMSigModel
let group: SBMGroup

// (...)  

let publicationSettings = SBMPublicationSettings(group: group)
let subscriptionControl = SBMSubscriptionControl(model: sigModel)
publicationSettings.ttl = 5 // For example 5. If needed, set this value as bigger.

subscriptionControl.setPublicationSettings(publicationSettings, successCallback: { subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})
6.15.1.2 Publish Directly to the Provisioner

```swift
let sigModel: SBMSigModel

//(...) 

let publicationSettings = SBMPublicationSettings(kind: .localAddress)
let subscriptionControl = SBMSubscriptionControl(model: sigModel)
publicationSettings.ttl = 5 // For example 5. If need, set this value as bigger.

subscriptionControl.setPublicationSettings(publicationSettings, successCallback: {
    subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})
```

6.15.2 Vendor Model

6.15.2.1 Publish via SBMGroup Address

```swift
let vendorModel: SBMVendorModel
let group: SBMGroup

//(...) 

let publicationSettings = SBMPublicationSettings(group: group)
let subscriptionControl = SBMSubscriptionControl(vendorModel: vendorModel)
publicationSettings.ttl = 5 // For example 5. If need, set this value as bigger.

subscriptionControl.setPublicationSettings(publicationSettings, successCallback: {
    subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})
```

6.15.2.2 Publish Directly to the Provisioner

```swift
let vendorModel: SBMVendorModel

//(...) 

let publicationSettings = SBMPublicationSettings(kind: .localAddress)
let subscriptionControl = SBMSubscriptionControl(vendorModel: vendorModel)
publicationSettings.ttl = 5 // For example 5. If need, set this value as bigger.

subscriptionControl.setPublicationSettings(publicationSettings, successCallback: {
    subscriptionControl, settings in
    //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
    //handle error callback
})
```
6.16 Control Node Functionality

Note: To perform the actions listed below you must have established a connection with a subnet that already has access to the node with the model that will be controlled. Before this step, the node containing the model must already be added to this group.

6.16.1 Get Value for a Single SIG Model from the Node

```swift
func getLevel(forElement element: SBMElement, inGroup group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.getStatus(SBMGenericLevel.self, successCallback: { (control, response) in
        let response = response as! SBMGenericLevel
        //handle success callback
    }, errorCallback: { (control, response, error) in
        //handle error callback
    })
}
```

6.16.2 Get Value for All Specific SIG Models Bound with a Group

Note: Models have to be subscribed to a given group.

```swift
func getLevel(forGroup group: SBMGroup) {
    let controlGroup = SBMControlGroup(group: group)
    controlGroup.getStatus(SBMGenericLevel.self, successCallback: { (control, response) in
        let response = response as! SBMGenericLevel
        //handle success callback
    }, errorCallback: { (control, response, error) in
        //handle error callback
    })
}
```

6.16.3 Set Value for a Single SIG Model from the Node

```swift
func set(level: Int, forElement element: SBMElement, inGroup group: SBMGroup {
    let controlElement = SBMControlElement(element: element, in: group)
    let status = SBMGenericLevel(level: Int16(level))
    let parameters = SBMControlRequestParameters(transitionTime: 1,
                                               delayTime: 1,
                                               requestReplay: false)
    controlElement.setStatus(status,
                     parameters: parameters,
                     successCallback: { control, response in
                                 //handle success callback
                }, errorCallback: { control, response, error in
                                 //handle error callback
                })
}
```
6.16.4 Set Value for All Specific SIG Models Bound with the Group

Note: Models have to be subscribed to a given group.

```swift
func set(lightness: Int, for group: SBMGroup) {
    let controlGroup = SBMControlGroup(group: group)
    let status = SBMLightningLightnessActual(lightness: UInt16(lightness))
    let parameters = SBMControlRequestParameters(transitionTime: 1,
                                               delayTime: 1,
                                               requestReplay: false)

    controlGroup.setStatus(status,
                           parameters: parameters,
                           successCallback: { control, response in
                                              //handle success callback
                           },
                           errorCallback: { control, response, error in
                                              //handle error callback
                           })
}
```

6.16.5 Control Sensor Models

Despite the fact that the SIG models already have logic for set/get states (see the previous four sections), a separate logic controls Sensor models. The Sensor model is available as a SIG model with a Sensor model identifier in the Device Composition Data.

Currently the Sensor API allows:
- Get sensor descriptors from the Sensor Server model
- Get measurement data as a single Sensor Data state from the given Sensor from the Sensor Server model
- Set/Get Sensor cadence
- Set/Get Sensor settings

Currently the Sensor API does NOT allow:
- Handle publications that are automatically sent by the Sensor model as: base data, columns, series and so on.
- Fetch measurement data as columns
- Fetch measurement data as series

To set up publication/subscription settings use the API prepared for SIG models (see sections 6.14.1 SIG Model and 6.15.1 SIG Model, respectively).

To set up Sensor model bindings use the API prepared for SIG models (see sections 6.12 Bind a Model with A Group and 6.13 Unbind a Model from a Group)

6.16.5.1 Get Sensor Model Values

6.16.5.1.1 Get Sensor Descriptors from the Node

```swift
propertyID – Insert a Sensor Property ID to receive the Sensor Descriptor for this ID. Put 0 to receive all Sensor Descriptors from the Sensor Server model.

func getSensorDescriptors(forElement element: SBMElement, inGroup group: SBMGroup, propertyID:Int) {
    let controlElement = SBMControlElement(element: element, in: group)
    let properties = SBMSensorPropertiesGet.descriptorStatus(propertyID)
    controlElement.getSensorStatus(SBMSensorDescriptors.self, properties: properties
                                    successCallback: { (control, response) in
                                           //handle success callback
                                    },
                                    errorCallback: { control, response, error in
                                           //handle error callback
                                    })
}
```
After Sensor Descriptors are successfully downloaded, they will be available in the SBMElement. They will be locally available because descriptors do not change over the time. Sensor Property ID is kept by SBMDescriptor as in the Mesh Model Bluetooth® Specification.

Structure:

```swift
let element: SBMElement
(...)
let sensors:[SBMSensor] = element.sensorsFromSensorServerModel()
(...)
let descriptor: SBMDescriptor = sensor.descriptor
```

### 6.16.5.1.2 Get Sensor State from the Node

**propertyID** – Insert a Sensor Property ID to receive the Sensor Status from the Sensor with this ID. Put 0 to receive states of all Sensors from the Sensor Server model.

Data received in the SBMSensorStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.4 Sensor Data from the Mesh Model Bluetooth® Specification).

```swift
func getSensorStatus(forElement element: SBMElement, inGroup group: SBMGroup, propertyID: Int) {
    let controlElement = SBMControlElement(element: element, in: group)
    let properties = SBMSensorPropertiesGet.sensorStatus(propertyID)
    controlElement.getSensorStatus(SBMSensorStatus.self, properties: properties, successCallback: { (control, response) in
        let response = response as! SBMSensorStatus
        //handle success callback
    }, errorCallback: { (control, response, error) in
        //handle error callback
    })
}
```

### 6.16.5.1.3 Get Sensor Cadence from the Node

**propertyID** - Insert a Sensor Property ID to receive the Sensor Cadence Status from the Sensor with this ID.

Data received in the SBMSensorCadenceStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

```swift
func getCadenceState(for element: SBMElement, in group: SBMGroup, propertyID: Int) {
    let controlElement = SBMControlElement(element: element, in: group)
    let properties = SBMSensorPropertiesGet.cadenceStatus(propertyID)
    controlElement.getSensorStatus(SBM SensorCadenceStatus.self, properties: properties, successCallback: {control, response in
        let response = response as! SBMSensorCadenceStatus
        //handle success callback
    }, errorCallback: {control, response, error in
        //handle error callback
    })
}
```
6.16.5.1.4 Get Sensor Setting from the Node

propertyID – Insert a Sensor Property ID to receive the Sensor Setting Status from the Sensor with this ID.

settingID – Insert a Sensor Setting Property ID to determine a specific setting within a sensor that will be received.

Data received in the SBMSensorSettingStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

```swift
func getSettingState(for element: SBMElement, in group: SBMGroup, propertyID: Int, settingID: Int) {
    let controlElement = SBMControlElement(element: element, in: group)
    let properties = SBMSensorPropertiesGet.settingStatus(settingID, sensor: propertyID)
    controlElement.getSensorStatus(SBMSensorSettingStatus.self, properties: properties,
                                   successCallback: {control, response in
                                      let response = response as! SBMSensorSettingStatus
                                      //handle success callback
                                   }, errorCallback: {control, response, error in
                                      //handle error callback
                                   })
}
```

6.16.5.1.5 Get Sensor Settings from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Settings Status from the Sensor with this ID.

Data received in the SBMSensorSettingsStatus is an array of 16-bit integer values representing IDs of Sensor Setting Properties.

```swift
func getSettingsState(for element: SBMElement, in group: SBMGroup, propertyID: Int) {
    let controlElement = SBMControlElement(element: element, in: group)
    let properties = SBMSensorPropertiesGet.settingsStatus(propertyID)
    controlElement.getSensorStatus(SBMSensorSettingsStatus.self, properties: properties,
                                    successCallback: {control, response in
                                       let response = response as! SBMSensorSettingsStatus
                                       //handle success callback
                                    }, errorCallback: {control, response, error in
                                       //handle error callback
                                    })
}
```

6.16.5.1.6 Get Sensor Descriptor from All Nodes in the Group

propertyID – Insert a Sensor Property ID to receive the Sensor Descriptor for this ID. Put 0 to receive all Sensor Descriptors from the Sensor Server model.

successHandler - This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call `task.cancel()`.

element – The response will be received from this SBMElement.

```swift
func getSensorDescriptors(inGroup group: SBMGroup, propertyID: Int) {
    let controlGroup = SBMControlGroup(group: group)
    let properties = SBMSensorPropertiesGet.descriptorStatus(propertyID)
```
After Sensor Descriptors are successfully downloaded, they will be available in the SBMElement. They will be locally available because descriptors do not change over the time. Sensor Property ID is kept by SBMDescriptor as in the Mesh Model Bluetooth® Specification.

Structure:

```swift
let element: SBMElement
(…)
let sensors:[SBMSensor] = element.sensorsFromSensorServerModel()
(…)
let descriptor: SBMDescriptor = sensor.descriptor
```

6.16.5.1.7 Get Sensor State from All Nodes in the Group

propertyID – Insert a Sensor Property ID to receive the Sensor Status from the Sensor with this ID. Put 0 to receive states of all Sensors from the Sensor Server model.

Measurement data received in the SBMSensorStatus has a structure defined by Bluetooth SIG Mesh Specification (see section 4.1.4 Sensor Data from the Mesh Model Bluetooth® Specification).

successHandler - This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call `task.cancel()`.

```swift
element – The response will be received from this SBMElement.

func getSensorStatuses(inGroup group: SBMGroup, propertyID: Int) {
    let controlGroup = SBMControlGroup(group: group)
    let properties = SBMSensorPropertiesGet.sensorStatus(propertyID)
    let task: SBMTask = controlGroup.getSensorStatus(SBMSensorStatus.self, properties: properties
        successHandler: { (control, response, element) in
            let response = response as! SBMSensorStatus
            //handle success
        },
        errorHandler: { (control, response, error) in
            //handle error
        }
    }
}
```

6.16.5.1.8 Get Sensor Cadence from All Nodes in the Group

propertyID - Insert a Sensor Property ID to receive the Sensor Cadence Status from the Sensor with this ID.

Data received in the SBMSensorCadenceStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call `task.cancel()`.

```swift
element – The response will be received from this SBMElement.

func getSensorCadenceStatuses(inGroup group: SBMGroup, propertyID: Int) {
    let controlGroup = SBMControlGroup(group: group)
    let cadenceProperties = SBMSensorPropertiesGet.sensorCadenceStatus(propertyID)
    let task: SBMTask = controlGroup.getSensorStatus(SBMSensorCadenceStatus.self, properties: cadenceProperties
        successHandler: { (control, response, element) in
            let response = response as! SBMSensorCadenceStatus
            //handle success
        },
        errorHandler: { (control, response, error) in
            //handle error
        }
    }
}
```
func getCadenceState(inGroup group: SBMGroup, propertyID: Int) {
    let controlGroup = SBMControlGroup(group: group)

    let properties = SBMSensorPropertiesGet.cadenceStatus(propertyID)
    let task = controlGroup.getSensorStatus(SBMSensorCadenceStatus.self, properties: properties,
                                               successHandler: {(control, response, element) in
                                                    let response = response as! SBMSensorCadenceStatus
                                                    //handle success
                                               }, errorHandler: { (control, response, error) in
                                                    //handle error
                                               })
}

6.16.5.9 Get Sensor Setting from All Nodes in the Group

propertyID - Insert a Sensor Property ID to receive the Sensor Setting Status from the Sensor with this ID.

settingID - Insert a Sensor Setting Property ID to determine a specific setting within a sensor that will be received.

Data received in the SBMSensorSettingStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call task.cancel().

element – The response will be received from this SBMElement.

func getSettingsState(inGroup group: SBMGroup, propertyID: Int, settingID: Int) {
    let controlGroup = SBMControlGroup(group: group)

    let properties = SBMSensorPropertiesGet.settingStatus(settingID, sensor: propertyID)
    let task = controlGroup.getSensorStatus(SBMSensorSettingStatus.self, properties: properties,
                                               successHandler: { (control, response, element) in
                                                    let response = response as! SBMSensorSettingStatus
                                                    //handle success
                                               }, errorHandler: { (control, response, error) in
                                                    //handle error
                                               })
}

6.16.5.10 Get Sensor Settings from All Nodes in the Group

propertyID - Insert a Sensor Property ID to receive the Sensor Settings Status from the Sensor with this ID.

Measurement data received in the SBMSensorSettingStatus is an array of 16-bit integer values representing IDs of Sensor Setting Properties.

successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call task.cancel().

element – The response will be received from this SBMElement.

func getCadenceState(inGroup group: SBMGroup, propertyID: Int) {
    let controlGroup = SBMControlGroup(group: group)

    let properties = SBMSensorPropertiesGet.setssettingsStatus(propertyID)
    let task = controlGroup.getSensorStatus(SBMSensorSettingsStatus.self, properties: properties,
                                               successHandler: { (control, response, element) in
                                                    //handle success
                                               })
}
let response = response as! SBMSensorSettingsStatus
//handle success
}, errorHandler: { (control, response, error) in
//handle error
})

6.16.5.2 Set Sensor Server Setup Model Values

6.16.5.2.1 Set Sensor Cadence within the Node

SBMSensorCadenceSet message contains 9 parameters:

SensorMessageFlags – determines whether acknowledge of sending message is required.

SensorPropertyID – determines ID of sensor to receive the message.

FastCadenceHigh – defines the upper boundary of a range of measured quantities when the publishing cadence is increased.

FastCadenceLow – defines the lower boundary of a range of measured quantities when the publishing cadence is increased.

FastCadencePeriodDivisor – controls the increased cadence of publishing Sensor Status messages. Valid values are 0 – 15 only.

StatusMinInterval – controls the minimum interval between publishing two consecutive Sensor Status messages. Valid values are 0 – 26 only.

StatusTriggerDeltaDown – controls the negative change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.

StatusTriggerDeltaUp - controls the positive change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.

StatusTriggerType – defines the unit and format of the Status Trigger Delta Down and Status Trigger Delta Up. Valid values are: SBMStatusTriggerTypeDefined (format is based on Sensor Property ID) and SBMStatusTriggerTypeUnitless (value is represented as a percentage change).

For full details about the parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

func setCadenceState(for element: SBMElement, in group: SBMGroup, propertyID: Int, cadenceSet: SBMSensorCadenceSet) {
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.setSensorSetupStatus(cadenceSet, successCallback: { control, response in
        let cadenceStatus = response as! SBMSensorCadenceStatus
        //handle success
    }, errorCallback: { control, response, error in
        //handle error
    })
}

6.16.5.2.2 Set Sensor Setting within the Node

SBMSensorSettingSet message contains 4 parameters:
SensorMessageFlags – determines whether acknowledgment of sending message is required.

SensorPropertyID – determines ID of sensor to receive the message.

SettingPropertyID – determines ID of device property for setting, including the size, format, and representation of the SettingRaw parameter.

SettingRaw – raw data representing setting of sensor as defined by the SettingPropertyID.

For full details about the parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

```swift
func setSettingState(for element: SBMElement, in group: SBMGroup, settingSet: SBMSensorSettingSet) {
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.setSensorSetupStatus(cadenceSet, successCallback: { control, response in
        let settingStatus = response as! SBMSensorSettingStatus
        //handle success
    }, errorCallback: { control, response, error in
        //handle error
    })
}
```

### 6.16.5.2.3 Set Sensor Cadence within All Nodes in the Group

SBMSensorCadenceSet message contains 9 parameters:

SensorMessageFlags – determines whether acknowledgment of sending message is required.

SensorPropertyID – determines ID of sensor to receive the message.

FastCadenceHigh – defines the upper boundary of a range of measured quantities when the publishing cadence is increased.

FastCadenceLow – defines the lower boundary of a range of measured quantities when the publishing cadence is increased.

FastCadencePeriodDivisor – controls the increased cadence of publishing Sensor Status messages. Valid values are 0 – 15 only.

StatusMinInterval – controls the minimum interval between publishing two consecutive Sensor Status messages. Valid values are 0 – 26 only.

StatusTriggerDeltaDown – controls the negative change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.

StatusTriggerDeltaUp - controls the positive change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.

StatusTriggerType – defines the unit and format of the Status Trigger Delta Down and Status Trigger Delta Up. Valid values are: SBMStatusTriggerTypeDefined (format is based on Sensor Property ID) and SBMStatusTriggerTypeUnitless (value is represented as a percentage change).

For full details about parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

```swift
func setCadenceState(for group: SBMGroup, cadenceSet: SBMSensorCadenceSet) {
    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setSensorSetupStatus(cadenceSet, successCallback: { control, response, element in
        let cadenceStatus = response as! SBMSensorCadenceStatus
        //handle success
    }, errorCallback: { control, request, error in
```
6.16.5.2.4 Set Sensor Setting within All Nodes in the Group

SBMSensorSettingSet message contains 4 parameters:

SensorMessageFlags – determines whether acknowledge of sending message is required.

SensorPropertyID – determines ID of sensor to receive the message.

SettingPropertyID – determines ID of device property for setting, including the size, format, and representation of the SettingRaw parameter.

SettingRaw – raw data representing setting of sensor as defined by the SettingPropertyID.

For full details about parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

```swift
func setSettingState(for group: SBMGroup, settingSet: SBMSensorSettingSet) {
    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setSensorSetupStatus(cadenceSet, successCallback: { control,
        response, element in
        let settingStatus = response as! SBMSensorSettingStatus
        //handle success
    }, errorCallback: { control, request, error in
        //handle error
    })
}
```

6.16.6 Subscribe for Notifications Sent by SIG Model

To capture notifications from the SIG Model publication settings must be configured on the model side (see section 6.15 Add Publication Settings to a Model). After setting up publication settings it is possible to start capturing notifications from the model by subscribe on change.

```swift
func subscribeLightness(forElement element: SBMElement, inGroup group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.subscribeStatus(SBMLightningLightnessActual.self) { controlElement,
        valueGetSigModel in
        //handle notifications
    }
}
```

6.16.7 Register a Local Vendor Model (SBMLocalVendorModel)

Registering a local vendor model hooks it to the mesh stack and enables the stack to pass incoming messages to the local vendor model. Typically registration should be done one time for the local vendor model after initializing SBMBluetoothMesh (see section 6.1 Initializing the BluetoothMesh).
6.16.7.1 Create SBMLocalVendorSettings

 Represents vendor registration settings.

 The opcodes used in this operation are manufacturer-specific opcode numbers. They can be calculated by subtracting 0xC0 from the first byte of the 3-octet opcode.

 Note: More information about Operation codes can be found in the Bluetooth SIG Documentation (Mesh Profile 3.7.3.1 Operation codes).

 let opCodes: Data // Manufacturer-specific operation codes supported by Vendor Model

 // (...)

 let messageHandler: SBMLocalVendorSettingsMessageHandler = { localVendorModel, applicationKeyIndex, sourceAddress, destinationAddress, virtualAddress, message, messageFlags in
    //Callback for handling incoming vendor messages
 }

 // (...)

 let registerSettings = SBMLocalVendorSettings(opCodes: Data(bytes: opCodes, messageHandler: messageHandler)

 Create SBMLocalVendorRegistrator

 6.16.7.2 Create SBMLocalVendorRegistrator

 Used to set registration settings of a local vendor model.

 let localVendorModel: SBMLocalVendorModel

 // (...)

 let vendorRegistrator = SBMLocalVendorRegistrator(model: localVendorModel)

 6.16.7.2.1 Register a Local Vendor Model

 Note: Typically registration should be done one time for the local vendor model after initializing SBMBluetoothMesh (see section 6.1 Initializing the BluetoothMesh).

 let vendorRegistrator: SBMLocalVendorRegistrator
 let registerSettings: SBMLocalVendorSettings

 // (...)

 vendorRegistrator.register(registerSettings)

 6.16.7.2.2 Unregister a Local Vendor Model

 let vendorRegistrator: SBMLocalVendorRegistrator

 // (...)

 vendorRegistrator.unregister

 6.16.8 Local Vendor Model Binding with Application Key

 SBMLocalVendorModel must be bound with SBMApplicationKey so that the Mesh Library can decrypt incoming messages from the mesh network. First, SBMVendorModel should be bound with SBMGroup (see section 6.12 Bind a Model with A Group), because SBMGroup is the source of the SBMApplicationKey. Messages that come from the SBMVendorModel from the SBMNode are encrypted with SBMApplicationKey from this SBMGroup.
6.16.8.1 Bind Local Vendor Model with Application Key

This function creates binding between a local vendor model and an application key required to decrypt incoming messages.

```swift
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

// (...) let cryptoBinder = SBMLocalVendorCryptoBinder(applicationKey: applicationKey) cryptoBinder.bindApplicationKey(to: localVendorModel)
```

6.16.8.2 Unbind Local Vendor Model from Application Key

Remove an existing binding between a local vendor model and an application key, meaning that the local vendor model will no longer process messages encrypted using that key.

```swift
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

// (...) let cryptoBinder = SBMLocalVendorCryptoBinder(applicationKey: applicationKey) cryptoBinder.unbindApplicationKey(from: localVendorModel)
```

6.16.9 Manage Notifications from the SBMVendorModel

It is possible to configure SBMVendorModel to publish notifications by SBMGroup address or directly by SBMNode address. It can be done by adding publication settings to the model (see section 6.15 Add Publication Settings to a Model).

To receive those notifications from the SBMVendorModel from the SBMNode the local stack must be able to collect those messages. The library must know what addresses it should follow. It is possible to set up the SBMGroup address or the direct SBMNode address for it. In this case the SBMVendorModel must have set publication by the address of the SBMGroup to which it is bound or the address of the SBMNode to which it belongs.

6.16.9.1 Sign Up for The Notifications

Method used to subscribe to the vendor model notifications.

**Note:** Notifications will come from the SBMLocalVendorSettingsMessageHandler from the SBMLocalVendorSettings, which should previously have been configured by SBMLocalVendorRegistrar.

6.16.9.1.1 Notifications Come by SBMGroup Address

```swift
let group: SBMGroup
let vendorModel: SBMVendorModel

// (...) let vendorNotifications = SBMVendorModelNotifications(group: group) vendorNotifications.signUp(forNotifications: vendorModel)
```

6.16.9.1.2 Notifications Come by SBMNode Address

```swift
let node: SBMNode
let vendorModel: SBMVendorModel

// (...) let vendorNotifications = SBMVendorModelNotifications(node: node)
```
vendorNotifications.signUp(forNotifications: vendorModel)

### 6.16.9.2 Sign Out from the Notifications

Method used to unsubscribe from the vendor model notifications.

#### 6.16.9.2.1 Sign Out from Notifications from the SBMGroup Address

```swift
let group: SBMGroup
let vendorModel: SBMVendorModel

// (...) 

let vendorNotifications = SBMVendorModelNotifications(group: group)
vendorNotifications.signOut(fromNotifications: vendorModel)
```

#### 6.16.9.2.2 Sign Out from Notifications from the SBMNode Address

```swift
let node: SBMNode
let vendorModel: SBMVendorModel

// (...) 

let vendorNotifications = SBMVendorModelNotifications(node: node)
vendorNotifications.signOut(fromNotifications: vendorModel)
```

### 6.16.10 Send Value to SBMVendorModel

#### 6.16.10.1 Create Implementation of the SBMControlValueSetVendorModel Protocol

Developers who use our library need to create their own implementation of the SBMControlValueSetVendorModel protocol. It will be used to send messages to the vendor model from the network.

Example:

```swift
class CompanyControlSetVendorModel: NSObject, SBMControlValueSetVendorModel {
    var localVendorModelClient: SBMLocalVendorModel? // it is deprecated, do NOT use it
    var vendorModel: SBMVendorModel
    var data: Data
    var flags: SBMControlValueSetVendorModelFlag

    init(with vendorModel: SBMVendorModel, messageToSend: Data, flags: SBMControlValueSetVendorModelFlag) {
        self.vendorModel = vendorModel
        self.data = messageToSend
        self.flags = flags
    }
}
```
6.16.10.2 Prepare Message to Send

It is very important to prepare the message with the correct structure. The first part of the message structure must contain the Opcode (Operation code) which will be used to send this message. This Opcode must be supported by the SBMVendorModel. The message structure must also contain the vendor company identifier that comes from the SBMVendorModel to which the message will be sent.

Note: More information about Operation codes can be found in the Bluetooth SIG Documentation (Mesh Profile 3.7.3.1 Operation codes).

Example:

```swift
let vendorModel: SBMVendorModel
let messageToSend: Data

// (...)

let companyID = vendorModel.vendorCompanyIdentifier()

// (...)

var data = Data(bytes: [UInt8(0) | 0xC0])
// In example my opcode is 0. 0xC0 must be HERE, reason can be found in the Bluetooth SIG Documentation (Mesh Profile 3.7.3.1 Operation codes)

data.append(Data(bytes: [UInt8(companyID & 0x00ff), UInt8(companyID >> 8 & 0x00ff)]))
// Add vendor company identifier

data.append(messageToSend)
```

6.16.10.3 Send Prepared Message to the Single SBMVendorModel on the SBMNode

Note: Replay messages will be sent from the SBMVendorModel if it supports that. Messages will be received in the SBMLocalVendorSettingsMessageHandler from the SBMLocalVendorSettings, which should previously have been configured by SBMLocalVendorRegistrar.

```swift
let element: SBMElement
let group: SBMGroup

// (...)

let controlElement = SBMControlElement(element: element, in: group)

// (...)

let messageToSend: Data
let vendorModel: SBMVendorModel
let flags: SBMControlValueSetVendorModelFlag

// (...)

let setVendorModel: CompanyControlSetVendorModel = CompanyControlSetVendorModel(with: vendor,
messageToSend: messageToSend, flags: flags)

// (...)

let controlElementSetVendorSuccess: SBMControlElementSetVendorSuccess = { controlElement, request in
    // Action invoked when message is successfully sent.
}

let controlElementSetVendorError: SBMControlElementSetVendorError = { controlElement, request, error in
    // Action invoked when message could not be sent.
}
controlElement.setStatus(setVendorModel, successCallback: controlElementSetVendorSuccess, errorCallback: controlElementSetVendorError)

6.16.10.4 Send Prepared Message to the SBMGroup

Note: Models have to be subscribed to a given group.

let group: SBMGroup

// ...

let controlGroup = SBMControlGroup(group: group)

// ...

let messageToSend: Data
let vendorModel: SBMVendorModel
let flags: SBMControlValueSetVendorModelFlag

// ...

let setVendorModel: CompanyControlSetVendorModel = CompanyControlSetVendorModel(with: vendor, messageToSend: messageToSend, flags: flags)

// ...

let controlGroupSetVendorSuccess: SBMControlGroupSetVendorSuccess = { controlGroup, request in
  //Action invoked when message is successfully sent.
}

let controlGroupSetVendorError: SBMControlGroupSetVendorError = { controlGroup, request, error in
  //Action invoked when message could not be sent.
}

controlGroup.setStatus(setVendorModel, successCallback: controlGroupSetVendorSuccess, errorCallback: controlGroupSetVendorError)
6.17 Sequence Diagrams for Vendor Model Functionality

6.17.1 SBMLocalVendorModel Initialization

6.17.1.1 Create SBMLocalVendorModel

Create SBMLocalVendorModel with vendor company identifier and vendor assigned model identifier which are the same as in the vendor model which will be controlled.

Create mesh configuration object with supported local vendor models.

Initialization should be done only once.

Initialize SBMBluetoothMesh object with given configuration.
6.17.1.2 Register the SBMLocalVendorModel in the Mesh Stack

Each SBMLocalVendorModel should have created its own message handler. It is a place where messages will come from the SBMLocalVendorModel with the same vendor company identifier and vendor assigned model.

For example network has two nodes. Each one have SBMLocalVendorModel with the same id. If all needed configuration will be done, messages from those two vendor models will be passed by the same message handler.

In the initializer of the SBMLocalVendorSettings there is place for two parameters:

1) opCodes - supported operation codes by vendor model. Developers who are implementing mobile application should contact developers who are responsible for the application for the embedded mesh device to get know what opcodes they should use. It is also important to get know how those opcodes work because it defines how they should use vendor model on the application side.

2) messageHandler - it is handler which was created a step before

Create SBMLocalVendorRegistrar with local vendor model which will be operated by registrar.

Register local vendor model with created SBMLocalVendorSettings. Registering a local vendor model hooks it to the mesh stack and enables the stack to pass incoming messages to the local vendor model.
6.17.2 Configure the SBMVendorModel

6.17.2.1 Bind the SBMVendorModel with the SBMGroup

The vendor model must always be bound on the node with the group.
6.17.2.2 Send Publication Settings to the SBMVendorModel

To configure the publication address for the notifications sent by the vendor model on the node, you must send publication settings to this node.

6.17.2.2.1 Set the Provisioner Address as the Publication Address

![Diagram of SBMVendorModel setup process]

- Pick SBMVendorModel which will be configured.
- Pass the SBMVendorModel via parameter to the initializer. The SBMSubscriptionControl will operate with this model.
- In this case it is needed to setup the SBMVendorModel to publish data directly to provisioner. To do it, call: SBMSubscriptionSettings(Kind: .localAddress)
- Pass SBMPublicationSettings as a parameter. It will send message to the node with configuration to which address the vendor model should publish notifications. SBMVendorModel which is needed for this configuration was passed during SBMSubscriptionControl initialization.
- Asynchronous callback with the result of configuration.
6.17.2.2.2 Set the SBMGroup address as the Publication Address

- **Pass the SBMVendorModel** via parameter to the initializer. The SBMSubscriptionControl will operate with this model.

- **Create SBMSubscriptionControl object**

- **Create SBMPublicationSettings object**

- **setPublicationSettings**: Set the SBMGroup address as the Publication Address. To do it, call:
  ```
  SBMSubscriptionControl(sbMVendorModel: SBMVendorModel, sbmPublicationSettings: SBMPublicationSettings)
  ```

- **Asynchronous callback with the result of configuration.**

- **In this case it is needed to setup SBMVendorModel to publish data via SBMGroup address. To do it, call: SBMPublicationSettings(group: SBMGroup)**

- **Pass SBMVendorModel as a parameter. It will send message to the node with configuration to which address the vendor model should publish notifications. SBMVendorModel which is needed for this configuration was passed during SBMSubscriptionControl initialization.**
6.17.2.3 Send Subscription Settings to the SBMVendorModel

When the vendor model on the node expects a message from the group address, you must send the subscription settings containing the address of this group to the vendor model on the node. Without that the vendor model will not receive a message sent by the group address, because it is not observing this group address.
6.17.3 Send Message to the Vendor Model on the Node

6.17.3.1 Send Message Directly to the Node

6.17.3.1.1 Message Without Response

```
6.17.3.1.1 Message Without Response

![Diagram showing the process of sending a message to the Vendor Model]
```

- Pick SBMVendorModel which will be controlled.
- Pick SBMGroup. SBMAplicationKey from this group will be used to encrypt message which will be sent.
- Create SBMControlElement object
- Parameters to pass to initializer:
  1) vendorModel - it is a SBMVendorModel which will be controlled
  2) data - data which will be sent to model
  3) flags - message flags
- Parameters to pass:
  1) status - Message object
  2) successCallback - will be called after send message with success
  3) errorCallback - will be called if any error occur
- Asynchronous callback with the result of send message.
6.17.3.1.2 Message With Response

- Pick SBMVendorModel which will be controlled.
- Create SBMLocalVendorCryptoBinder object
- Bind SBMApplcationKey with SBMLocalVendorModel. For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and a needed application key to decrypt this message.

In one moment it is possible to create only one binding between the SBMApplcationKey and the SBMLocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

- Pass SBMApplcationKey from the given SBMGroup as parameter to initializer.
- Array of vendor models
- localVendorModels
- Array of bound groups
- bindedGroups
- vendorModels

Pick SBMVendorModel which will be controlled.
Create SBMControlElement

- Pass picked SBMGroup and SBMElement via parameters to initializer.

Message object

- Parameters to pass to initializer:
  1) vendorModel - it is a SBMVendorModel which will be controled
  2) data - data which will be sent to model
  3) flags - message flags

setStatus:

- Parameters to pass:
  1) status - Message object
  2) successCallback - will be called after send message with success
  3) errorCallback - will be called if any error occur

callback

- Asynchronous callback with the result of send message.

RX message which is a result for TX message
which was sent to the SBMVendorModel.
SBMLocalVendorSettingsMessageHandler was
created during SBMLocalVendorModel
configuration by developer.
6.17.3.2 Send Message Via Group Address

6.17.3.2.1 Message without Response

![Diagram showing the process of sending a message via group address with SBMVendorModel and SBMElement.]

- Pick SBMVendorModel which will be controlled.
- Array of binded groups.
- Pick SBMGroup. SBMAplicationKey from this group will be used to encrypt message. All SBMVendorModels with the same ids will be controlled from this group if they are configured for it.
6.17.3.2.2 Message with Response

- Pick SBMVendorModel which will be controlled.
  - bindedGroups
  - Array of binded groups
  - SBVVendorModel
    - vendorModels
      - Array of vendor models

- Pick SBMGroup with SBMApplicationKey from this group to be used to encrypt TX message and decrypt RX message. All SBMVendorModels with the same ids will be controlled from this group if they are configured for it.

- Create SBMLocalVendorCryptoBinder object

- Bind SBMApplicationKey with SBMLocalVendorModel. For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and a needed application key to decrypt this message.

- Pass SBMApplicationKey from the given SBMGroup as parameter to initializer.

- In one moment it is possible to create only one binding between the SBMApplicationKey and the SBMLocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone wants to duplicate binding.
SDMControlGroup

Create

Message: SBMControlValueSetVendorModel

Pass picked SBMGroup via parameter in the initializer.

SBMControlGroup object

Create

Message object

setStatus:

Parameters to pass:
- status - Message object
- successCallback - will be called after send message with success
- errorCallback - will be called if any error occur

Message will be sent to all vendor models from the group with the same ids as the vendor model that is given in the Message object.

RX messages which are a result for TX messages which was sent.
SBMVendorSetVendorModelMessageHandler was created during SBMVendorModel configuration by developer.
Each vendor model from the group will reply in the separate message.

callback

Asynchronous callback with the result of send message.
6.17.4 Subscribe to Notifications Published by the Vendor Model from the Node

6.17.4.1 Messages are Sent Directly to the Provisioner

In one moment it is possible to create only one binding between the SBMApplcationKey and the SBMLocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.
Pick SBMNode. This SBMNode is needed to define node address which will be observed by provisioner. It is needed to capture notifications from the vendor model.

Pass picked SBMNode via parameter to initializer.

Pass SBMVendorModel via parameter. From this SBMVendorModel notifications will be accepted.

Incoming messages which are published by the vendor model to the provisioner as a notifications. SBMLocalVendorSettingsMessageHandler was created during SBMLocalVendorModel configuration by developer.
6.17.4.2 Messages are Sent via Group Address to the Provisioner

Pick SBMVendorModel which will be controlled.

This SBMGroup is also needed to define group address which will be observed by provisioner. It is needed to capture notifications from the vendor model which publishes notifications via this group address.

In one moment it is possible to create only one binding between the SBMApplicationKey and the SBMLocalVendorCryptoBinder. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

Pass SBMApplicationKey from the given SBMGroup as parameter to initializer.

Bind SBMApplicationKey with SBMLocalVendorCryptoBinder. For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and a needed application key to decrypt this message.
AN1200: Bluetooth® Mesh for iOS and Android ADK
Bluetooth Mesh API Reference for iOS

![Diagram of Bluetooth Mesh API Reference for iOS]

- **Developer**
  - Create
    - Pass picked SBMGroup via parameter to initializer
    - SBMVendorModelNotifications object
  - signUpForNotifications:vendorModel
    - Pass SBMVendorModel via parameter. From this SBMVendorModel notifications will be accepted.
  - Incoming messages which are published by the vendor model to the provisioner via group address as a notifications. SBMLocalVendorSettingsMessageHandler was created during SBMLocalVendorModel configuration by developer.

*Silabs.com | Building a more connected world.*
7 Bluetooth Mesh API Reference for Android

To control base of Bluetooth mesh structure you must be familiar with a few of the most important layers of the Bluetooth mesh API:
1. Bluetooth connection
2. Provision session
3. Proxy connection
4. Node configuration
5. Model configuration (subscription and publication settings)
6. Control model and group

All are described in the next part of this document.

7.1 Initializing the BluetoothMesh

7.1.1 BluetoothMesh

The Bluetooth mesh structure is represented by singleton object of the BluetoothMesh class. This is a main entry point to the library and it gives access to all other objects within the library.

BluetoothMesh must be configured before using the library. Configuration can be provided using the BluetoothMeshConfiguration class.

In the base configuration supported vendor models are not needed.

To get the instance call the following for each platform noted

```java
BluetoothMesh.getInstance();

BluetoothMeshConfiguration configuration = new BluetoothMeshConfiguration();
BluetoothMesh.initialize(context, configuration);
```

The `context` parameter is the application context the user can access through `getApplicationContext()`.

7.1.2 Set Up Supported Vendor Models

Supported vendor models can be set using `BluetoothMeshConfiguration` during its initialization. To do that you must know a specification for each vendor model. It should be delivered by the external provider.

```java
LocalVendorModel vendorModel = new LocalVendorModel(companyIdentifier, assignedModelIdentifier);

// companyIdentifier – vendor company identifier. Need to be the same as vendor company identifier from the VendorModel from the Node which will be controlled.
// assignedModelIdentifier – vendor assigned model identifier. Need to be the same as vendor assigned model identifier from the VendorModel from the Node which will be controlled.

BluetoothMeshConfiguration configuration = new BluetoothMeshConfiguration(Collections.singletonList(vendorModel));
BluetoothMesh.getInstance().initialize(context, configuration);
```

7.1.3 Set Up Mesh Limits

Limits for Bluetooth Mesh database can be set using `BluetoothMeshConfigurationLimits`.

Warning: Changing limits between each application launch may corrupt the database.

```java
BluetoothMeshConfigurationLimits limits = new BluetoothMeshConfigurationLimits();

int networks; // maximum number of network keys that can be created - MAX is 7
int groups; // maximum number of application keys that can be created - MAX is 8. Be aware that it is possible to create maximum 16,128 groups when groups share application keys between each other.
```
int nodes; //maximum number of nodes that can be provisioned - MAX is 255
int nodeNetworks; //maximum number of network keys that a single node can be added to - MAX is 7
int nodeGroups; //maximum number of application keys that a single node can be added to - MAX is 8
int rplSize; //maximum number of nodes that can be communicated with - MAX is 255
int segmentedMessagesReceived; //maximum number of concurrent segmented messages being received - MAX is 255
int segmentedMessagesSent; //maximum number of concurrent segmented messages being sent - MAX is 255
int provisionSessions; //maximum number of parallel provisioning sessions - MAX is 1

limits.setNetworks(networks);
limits.setGroups(groups);
limits.setNodes(nodes);
limits.setNodeNetworks(nodeNetworks);
limits.setNodeGroups(nodeGroups);
limits.setRplSize(rplSize);
limits.setSegmentedMessagesReceived(segmentedMessagesReceived);
limits.setSegmentedMessagesSent(segmentedMessagesSent);
limits.setProvisionSessions(provisionSessions);

When a specific limit is not set, a default value is assigned:

networks = 7;
groups = 8;
nodes = 255;
nodeNetworks = 7;
nodeGroups = 8;
rplSize = 255;
segmentedMessagesReceived = 4;
segmentedMessagesSent = 4;
provisionSessions = 1;

BluetoothMeshConfiguration configuration = new BluetoothMeshConfiguration(localVendorModels, limits);
BluetoothMesh.initialize(context, configuration);
### 7.2 Set Up Bluetooth Layer (ConnectableDevice)

The Bluetooth Mesh Android API provides a layer that helps manage Android Bluetooth LE. The developer must provide an implementation of the ConnectableDevice abstract class, which is a connection between BluetoothDevice from the BluetoothGatt and a layer responsible for communication with the Bluetooth mesh structure.

#### 7.2.1 Device Advertisement Data

Advertisement data can be obtained from `ScanResult`.

```java
ScanResult: scanResult

// (...) advertisementData = Objects.requireNonNull(scanResult.getScanRecord()).getBytes();

// (...) public byte[] getAdvertisementData() {
    return advertisementData;
}
```
7.2.2 Device UUID

The device UUID comes from advertisement data. Note that Device UUID is available only for non-provisioned Bluetooth mesh-capable devices.

```java
// ConnectableDevice
public byte[] getUUID() {
    byte[] uuid = new byte[16];
    byte[] data = getServiceData(MESH_UNPROVISIONED_SERVICE);
    System.arraycopy(data, 0, uuid, 0, uuid.length);
    return uuid;
}
```

7.2.3 Device Name

The device name comes from BluetoothDevice.

```java
BluetoothDevice bluetoothDevice; // (…)

public String getName() {
    return bluetoothDevice.getName();
};
```

7.2.4 Device Connection State

The Device Connection State is a Boolean value that determines whether a device is connected.

```java
// ConnectableDevice
public boolean isConnected() {
    return connected;
}
```

7.2.5 Connect to the Device

```java
BluetoothGatt bluetoothGatt;
BluetoothGattCallback bluetoothGattCallback;

// (…)

public void connect() {
    bluetoothGatt = bluetoothDevice.connectGatt(context, false, bluetoothGattCallback, BluetoothDevice.TRANSPORT_LE);
}
```

7.2.6 Disconnect from the Device

```java
BluetoothGatt bluetoothGatt;
let peripheral: CBPeripheral
let callback: SBMConnectableDeviceConnectionCallback

// (…)

public void disconnect() {
    if (bluetoothGatt != null) {
        bluetoothGatt.disconnect();
    }
```
7.2.7 Check if a Device Contains a Service

The Bluetooth mesh framework sometimes needs to check if a Bluetooth device contains a needed service for its operation.

```java
ScanResult scanResult;
// (...) public boolean hasService(UUID service) {
    ScanRecord scanRecord = scanResult.getScanRecord();
    if (scanRecord == null) {
        return false;
    }
    List<ParcelUuid> serviceUuids = scanRecord.getServiceUuids();
    if (serviceUuids == null) {
        return false;
    }
    return serviceUuids.contains(new ParcelUuid(service));
}
```

7.2.8 Maximum Transmission Unit for Given Device Service

```java
int mtuSize;
// (...) private BluetoothGattCallback bluetoothGattCallback = new BluetoothGattCallback() {
    // (...) @Override
    public void onMtuChanged(BluetoothGatt bluetoothGatt, int mtu, int status) {
        super.onMtuChanged(bluetoothGatt, mtu, status);
        Log.d(TAG, "onMtuChanged : status: ", status + ", mtu: " + mtu);
        if (status == BluetoothGatt.GATT_SUCCESS) {
            mtuSize = mtu;
            bluetoothGatt.discoverServices();
        }
    }
    // (...) public int getMTU() {
        return mtuSize;
    }
```

7.2.9 Write Data to a Given Service and Characteristic

Write method is a function where the Bluetooth mesh framework sends bytes to the ConnectableDevice.

```java
BluetoothGatt bluetoothGatt;
// (...)
public void writeData(UUID service, UUID characteristic, byte[] data,
ConnectableDeviceWriteCallback callback) {
    if (bluetoothGatt == null) {
        callback.onFailed(service, characteristic);
        return;
    }
    BluetoothGattService bluetoothGattService = bluetoothGatt.getService(service);
    if (bluetoothGattService == null) {
        callback.onFailed(service, characteristic);
        return;
    }
    BluetoothGattCharacteristic gattCharacteristic =
    bluetoothGattService.getCharacteristic(characteristic);
    gattCharacteristic.setValue(data);
    gattCharacteristic.setWriteType(BluetoothGattCharacteristic.WRITE_TYPE_NO_RESPONSE);
    if (bluetoothGatt.writeCharacteristic(gattCharacteristic)) {
        callback.onWrite(service, characteristic);
    } else {
        callback.onFailed(service, characteristic);
    }
}

7.2.10 Subscribe to a Given Service and Characteristic

BluetoothGatt bluetoothGatt;

// (...)

public void subscribe(UUID service, UUID characteristic, ConnectableDeviceSubscriptionCallback
callback) {
    if (bluetoothGatt == null) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattService bluetoothGattService = bluetoothGatt.getService(service);
    if (bluetoothGattService == null) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattCharacteristic gattCharacteristic =
    bluetoothGattService.getCharacteristic(characteristic);
    if (!bluetoothGatt.setCharacteristicNotification(gattCharacteristic, true)) {
        callback.onFail(service, characteristic);
        return;
    }
    if (gattCharacteristic.getDescriptors().size() != 1) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattDescriptor gattDescriptor = gattCharacteristic.getDescriptors().get(0);
    gattDescriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
    if (!bluetoothGatt.writeDescriptor(gattDescriptor)) {
        callback.onFail(service, characteristic);
        return;
    }
    callback.onSuccess(service, characteristic);
7.3 Provision a Device to a Subnet

First you must discover a Bluetooth device that is compatible with Bluetooth mesh networking. This device will be in a non-provisioned state. It is not possible to provision a device a second time without a factory reset.

Before you can provision a device, you must prepare a Network with a Subnet. To start a provisioning session, choose a subnet to which to provision the device. The steps are as follows.

7.3.1 Create Network

Network network = BluetoothMesh.getInstance().createNetwork("Network name")

7.3.2 Create Subnet

Subnet subnet = network.createSubnet("Subnet name")

7.3.3 Find Non-Provisioned Bluetooth Devices

Find a device that is compatible with Bluetooth mesh networking and is not provisioned. A non-provisioned device will advertise its provisioning service. To represent a non-provisioned device, you should use a class that implements ConnectableDevice,

BluetoothMesh bluetoothMesh;
Context context;

// (…)
ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        if (result == null || result.getScanRecord() == null ||
            result.getScanRecord().getServiceUuids() == null ||
            result.getScanRecord().getServiceUuids().isEmpty()) {
            return;
        } // BTConnectableDevice extends ConnectableDevice abstract class.
        BTConnectableDevice device = new BTConnectableDevice(context, result);
    }
};
ScanSettings settings = new ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
    .build();
UUID uuid = ProvisionerConnection.MESH_UNPROVISIONED_SERVICE;
ScanFilter filter = new ScanFilter.Builder().setServiceUuid(new ParcelUuid(uuid)).build();
BluetoothLeScanner bluetoothLeScanner;
bluetoothLeScanner.startScan(Collections.singletonList(filter), settings, scanCallback);
7.3.4 Provision the Devices

During device provisioning, ensure that at least one device is provisioned as a proxy. The Bluetooth mesh network can only connect to devices with active proxy.

The node returned from the provision connection callback is a Bluetooth mesh counterpart of the ConnectableDevice.

The following examples show in-band provisioning and out-of-band (OOB) provisioning.

7.3.4.1 In-Band Provisioning

ConnectableDevice connectableDevice;
Subnet subnet;
// (...)
ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);
// (...)
provisionerConnection.provisionAsProxy(new ProvisioningCallback() {
    @Override
    public void success(ConnectableDevice device, Subnet subnet, Node node) {
    }
    @Override
    public void error(ConnectableDevice device, Subnet subnet, ErrorType error) {
    }
});

7.3.4.2 OOB Provisioning

ConnectableDevice connectableDevice;
Subnet subnet;
ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);
ProvisionerOOBControl provisionerOOBControl;
provisionerConnection.setProvisionerOOB(provisionerOOBControl);
provisionerConnection.provisionAsProxy(this);

7.3.4.2.1 ProvisionerOOBControl

This abstract class must be implemented in order to use OOB provisioning for a Bluetooth device.

The following is an example of implementing ProvisionerOOBControl. Detailed explanations are provided after the example.

import com.siliconlab.bluetoothmesh.adk.provisioning.ProvisionerOOB;
import com.siliconlab.bluetoothmesh.adk.provisioning.ProvisionerOOBControl;
import java.util.HashSet;
import java.util.Set;

public class ProvisionerOOBImpl2 extends ProvisionerOOBControl {
    @Override
    public PUBLIC_KEY_ALLOWED publicKeyAllowed() {
        return ProvisionerOOB.PUBLIC_KEY_ALLOWED.YES;
    }

    @Override
    public Set<AUTH_METHODS_ALLOWED> setOfAuthMethodsAllowed() {
        Set<AUTH_METHODS_ALLOWED> authMethodsAllowed = new HashSet<>();
    }
authMethodsAllowed.add(AUTH_METHODS_ALLOWED.INPUT_OOB);
authMethodsAllowed.add(AUTH_METHODS_ALLOWED.OUTPUT_OOB);
authMethodsAllowed.add(AUTH_METHODS_ALLOWED.STATIC_OOB);
return authMethodsAllowed;
}

@Override
public Set<OUTPUT_ACTIONS_ALLOWED> setOfOutputActionsAllowed() {
    Set<OUTPUT_ACTIONS_ALLOWED> outputActionsAllowed = new HashSet<>();
    outputActionsAllowed.add(ProvisionerOOB.OUTPUT_ACTIONS_ALLOWED.ALPHA);
    outputActionsAllowed.add(ProvisionerOOB.OUTPUT_ACTIONS_ALLOWED.BEEP);
    outputActionsAllowed.add(ProvisionerOOB.OUTPUT_ACTIONS_ALLOWED.BLINK);
    outputActionsAllowed.add(ProvisionerOOB.OUTPUT_ACTIONS_ALLOWED.NUMERIC);
    outputActionsAllowed.add(ProvisionerOOB.OUTPUT_ACTIONS_ALLOWED.VIBRATE);
    return outputActionsAllowed;
}

@Override
public Set<INPUT_ACTIONS_ALLOWED> setOfInputActionsAllowed() {
    Set<INPUT_ACTIONS_ALLOWED> inputActionsAllowed = new HashSet<>();
    inputActionsAllowed.add(INPUT_ACTIONS_ALLOWED.ALPHA);
    inputActionsAllowed.add(INPUT_ACTIONS_ALLOWED.NUMERIC);
    inputActionsAllowed.add(INPUT_ACTIONS_ALLOWED.PUSH);
    inputActionsAllowed.add(INPUT_ACTIONS_ALLOWED.TWIST);
    return inputActionsAllowed;
}

@Override
public int minLengthOfOOBData() {
    return 0;
}

@Override
public int maxLengthOfOOBData() {
    return 0;
}

@Override
public RESULT oobPublicKeyRequest(byte[] uuid, int algorithm, int publicKeyType) {
    byte[] data;  //(...)
    //Developer has to provide implementation to get needed data.
    //(...)
    providePublicKey(uuid, data);
    return RESULT.SUCCESS;
}

@Override
public RESULT outputRequest(byte[] uuid, OUTPUT_ACTIONS outputActions, int outputSize) {
    byte[] data;  //(...)
    //Developer has to provide implementation to get needed data.
    //(...)
    provideAuthData(uuid, data);
    return RESULT.SUCCESS;
}

@Override
public RESULT authRequest(byte[] uuid) {
    return RESULT.SUCCESS;
}

@Override
public void inputOobDisplay(byte[] uuid, INPUT_ACTIONS inputAction, int inputSize, byte[] authData) {
    // (...)
    // Developer has to provide implementation to display authData.
    // (...)
}

Explanation:

PUBLIC_KEY_ALLOWED publicKeyAllowed();

Returns whether OOB EC public key is allowed or not during OOB provisioning.

Set<AUTH_METHODS_ALLOWED> setOfAuthMethodsAllowed();

Returns allowed authentication methods during OOB provisioning.

Set<OUTPUT_ACTIONS_ALLOWED> setOfOutputActionsAllowed();

Returns allowed output actions during OOB provisioning.

Set<input_ACTIONS_ALLOWED> setOfInputActionsAllowed();

Returns allowed input actions during OOB provisioning.

int minLengthOfOOBData();

Returns minimum allowed input/output OOB data length during OOB provisioning. Zero value is interpreted as default value of 1.

int maxLengthOfOOBData();

Returns maximum allowed input/output OOB data length during OOB provisioning. Zero value is interpreted as default value of 8.

RESULT oobPublicKeyRequest(byte[] uuid, int algorithm, int publicKeyType);

Optional method called when platform requests an out-of-band public key.

RESULT outputRequest(byte[] uuid, OUTPUT_ACTIONS outputActions, int outputSize);

Optional method called when platform requests an out-of-band output authentication data.

RESULT authRequest(byte[] uuid);

Optional method called when platform requests an out-of-band static authentication data.

void inputOobDisplay(byte[] uuid, INPUT_ACTIONS inputAction, int inputSize, byte[] authData);

Optional method called when platform requests an out-of-band input authentication data.
public void providePublicKey(final byte[] uuid, final byte[] publicKey)

Provides out-of-band device public key of a device to the Bluetooth device. Call this method after the platform has requested an out-of-band public key for a device.

public void provideAuthData(final byte[] uuid, final byte[] data)

Provides out-of-band authentication data of a device to the Bluetooth device. Call this method after the platform has requested out-of-band static authentication data or output authentication data.

7.4 Add a Node to a Subnet

Note: To perform this action you have to have established a connection with a subnet that already has access to this node.

During the device provisioning session, the node is added to a subnet. It is possible to add a node to multiple subnets from the same network. Below is an example of how to add a node to another subnet.

Node node;
Subnet subnet;
NodeControl control = new NodeControl(node);

// (…)
control.addTo(subnet, new NodeControlCallback() {
    @Override
    public void succeed() {
    }
    @Override
    public void error(ErrorType errorType) {
    }
});

7.5 Remove a Node from a Subnet

Note: To perform this action you must have established a connection with a subnet that already has access to this node.

It is possible to remove a node from a subnet. Be aware that you can lose access to the node irreversibly if you remove it from its last subnet.

Node node;
Subnet subnet;
NodeControl control = new NodeControl(node);

// (…)
control.addTo(subnet, new NodeControlCallback() {
    @Override
    public void succeed() {
    }
    @Override
    public void error(ErrorType errorType) {
    }
});
7.6 Connect with a Subnet

You can only be connected with one subnet at a time.

7.6.1 Find All Proxies in the Device Range

Find devices that are compatible with Bluetooth mesh networking and were provisioned as proxies. A provisioned proxy device will advertise its proxy service. To represent a provisioned proxy device, you should use a class that extends ConnectableDevice.

```java
BluetoothMesh bluetoothMesh;
Context context;

// (…)
ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        if (result == null || result.getScanRecord() == null ||
            result.getScanRecord().getServiceUuids() == null ||
            result.getScanRecord().getServiceUuids().isEmpty()) {
            return;
        }
        // BTConnectableDevice extends ConnectableDevice abstract class.
        BTConnectableDevice device = new BTConnectableDevice(context, result);
    }
};
ScanSettings settings = new ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
    .build();
UUID uuid = ProxyConnection.MESH_PROXY_SERVICE;
ScanFilter filter = new ScanFilter.Builder().setServiceUuid(new ParcelUuid(uuid)).build();
BluetoothLeScanner bluetoothLeScanner;
bluetoothLeScanner.startScan(Collections.singletonList(filter), settings, scanCallback);
```

7.6.2 Get Node Representing a Given Device

To retrieve a Bluetooth mesh node counterpart of the ConnectableDevice use ConnectableDeviceHelper.

```java
ConnectableDevice connectableDevice:
ConnectableDeviceHelper connectableDeviceHelper;

// (…)
Node node = connectableDeviceHelper.findNode(connectableDevice)
```

7.7 Create a Group in a Given Subnet

7.7.1 Create a Group with a Name

```java
Subnet subnet;

// (…)
Group group = subnet.createGroup("Group name");
```
7.7.2 Create a Group with a Name and an AppKey

Subnet subnet;

// (...)

Group group = subnet.createGroup("Group name", appKey);

7.8 Remove Group

Note: To perform this action you must have an established connection with a subnet that already has access to this group.

Group group;

// (...)

group.removeGroup(new GroupRemovalCallback() {
    @Override
    public void success(Group group) {
    }

    @Override
    public void error(Group group, ErrorType errorType) {
    }
});

7.9 Add a Node to a Group

Note: To perform this action you must have an established connection with a subnet that already has access to this group. The node must already be added to the same subnet.

Node node;
Group group;

// (...)

NodeControl control = new NodeControl(node);
control.bind(group, new NodeControlCallback() {
    @Override
    public void succeed() {
    }

    @Override
    public void error(ErrorType errorType) {
    }
});

7.10 Remove a Node from a Group

Note: To perform this action you must have an established connection with a subnet that already has access to this group. The node must already be added to the same subnet.

Node node;
Group group;

// (...)

NodeControl control = new NodeControl(node);
control.unbind(group, new NodeControlCallback() {
    @Override
    public void succeed() {
    }
    @Override
    public void error(ErrorType errorType) {
    }
});

7.11 Bind a Model with A Group

Note: To perform this action you must have established a connection with a subnet that already has access to this group. Before this step, a node containing the model must already be added to the corresponding subnet and group.

    Model model;
    Group group;
    // (...)
    FunctionalityBinder binder = new FunctionalityBinder(group);

    binder.bindModel(model, new FunctionalityBinderCallback() {
        @Override
        public void succeed(List<Model> list, Group group) {
        }
        @Override
        public void error(List<Model> list, Group group, ErrorType errorType) {
        }
    });

7.12 Unbind a Model from a Group

Note: To perform this action you must have established a connection with a subnet that already has access to this group. Before this step, a node containing the model must already be added to the corresponding subnet.

    Model model;
    Group group;
    // (...)
    FunctionalityBinder binder = new FunctionalityBinder(group);

    binder.unbindModel(model, new FunctionalityBinderCallback() {
        @Override
        public void succeed(List<Model> list, Group group) {
        }
        @Override
        public void error(List<Model> list, Group group, ErrorType errorType) {
        }
    });
7.13 Add Subscription Settings to a Model

Note: To perform this action you must have established a connection with a subnet that already has access to this model. Before this step, a node containing the model must already be added to the corresponding subnet and group.

7.13.1 SIG model

Model model;
Group group;

// (...

SubscriptionSettings subscriptionSettings = new SubscriptionSettings(group);
SubscriptionControl subscriptionControl = new SubscriptionControl(model);

subscriptionControl.addSubscriptionSettings(subscriptionSettings, new
SubscriptionSettingsCallback() {
    @Override
    public void success(SigModel sigModel, SubscriptionSettings subscriptionSettings) {
    }
    @Override
    public void error(SigModel sigModel, ErrorType errorType) {
    }
});

7.13.2 Vendor Model

VendorModel vendorModel;
Group group;
// (...

SubscriptionSettings subscriptionSettings = new SubscriptionSettings(group);
SubscriptionControl subscriptionControl = new SubscriptionControl(vendorModel);

subscriptionControl.addSubscriptionSettings(subscriptionSettings, new
SubscriptionSettingsCallback() {
    @Override
    public void success(SigModel sigModel, SubscriptionSettings subscriptionSettings) {
        }
    @Override
    public void error(SigModel sigModel, ErrorType errorType) {
        }
    });
7.14 Add Publication Settings to a Model

Note: To perform this action you must have established a connection with a subnet that already has access to this model. Before this step, a node containing the model must already be added to the corresponding subnet and group.

To allow a switch node to operate properly on the group publication settings must be set up on this model with a given group.

To activate notifications from the model publication settings must be configured. The model has to know the address to which it should send notifications. Without this step messages can only be received from the model with GET/SET calls; automatic notifications cannot be received. This step is important for both SIG Models and Vendor Models.

7.14.1 SIG Model

7.14.1.1 Publish via Group Address

Model model;
Group group;

//(...)
PublicationSettings publicationSettings = new PublicationSettings(group);
SubscriptionControl subscriptionControl = new SubscriptionControl(model);

subscriptionControl.setPublicationSettings(publicationSettings, new PublicationSettingsCallback() {
    @Override
    public void success(SigModel sigModel, PublicationSettings publicationSettings) {
    }
    @Override
    public void error(SigModel sigModel, ErrorType errorType) {
    }
});

7.14.1.2 Publish Directly to the Provisioner

SigModel sigModel;
//(...)
PublicationSettings publicationSettings = new PublicationSettings(NotificationSettings.Kind.LOCAL_ADDRESS);
SubscriptionControl subscriptionControl = new SubscriptionControl(sigModel);
publicationSettings.setTtl(5); //5 is an example value. Set higher if needed.

subscriptionControl.setPublicationSettings(publicationSettings, new PublicationSettingsGenericCallback() {
    @Override
    public void success(Model meshModel, PublicationSettings publicationSettings) {
    }
    @Override
    public void error(Model meshModel, ErrorType errorType) {
    }
});
7.14.2 Vendor Model

7.14.2.1 Publish via Group Address

VendorModel vendorModel;
Group group;
//(...)
PublicationSettings publicationSettings = new PublicationSettings(group);
SubscriptionControl subscriptionControl = new SubscriptionControl(vendorModel);
publicationSettings.setTtl(5); //5 is an example value. Set higher if needed.

subscriptionControl.setPublicationSettings(publicationSettings, new
PublicationSettingsGenericCallback() {
    @Override
    public void success(Model meshModel, PublicationSettings publicationSettings) {
    }

    @Override
    public void error(Model meshModel, ErrorType errorType) {
    }
});

7.14.2.2 Publish Directly to the Provisioner

Known issue. There is a problem with starting capturing notifications from the Vendor Model if it publishes messages directly to the Provisioner. Currently there is a workaround for it. The Provisioner will start capturing notifications after it sends one SET message to the Vendor Model (see section 7.15.9 Send Value to VendorModel).

VendorModel vendorModel;
//(...)
PublicationSettings publicationSettings = new
PublicationSettings(NotificationSettings.Kind.LOCAL_ADDRESS);
SubscriptionControl subscriptionControl = new SubscriptionControl(vendorModel);
publicationSettings.setTtl(5); //5 is an example value. Set higher if needed.

subscriptionControl.setPublicationSettings(publicationSettings, new
PublicationSettingsGenericCallback() {
    @Override
    public void success(Model meshModel, PublicationSettings publicationSettings) {
    }

    @Override
    public void error(Model meshModel, ErrorType errorType) {
    }
});

7.15 Control Node Functionality

Note: To perform the actions listed below you must have established a connection with a subnet that already has access to the node with the model that will be controlled. Before this step, the node containing the model must already be added to this group.

7.15.1 Get Value for a Single Model from the Node

GenericLevel genericLevel;
//(...)

void getLevel(Element element, Group group) {
    ControlElement controlElement = new ControlElement(element, group);
controlElement.getStatus(genericLevel, new GetElementStatusCallback<ControlValueGetSigModel>() {
    @Override
    public void success(Element element, Group group, ControlValueGetSigModel controlValueGetSigModel) {
    }
    @Override
    public void error(Element element, Group group, ErrorType errorType) {
    }
});

### 7.15.2 Get Value for All Specific Models Bound with a Group

**Note:** Models have to be subscribed to a given group.

```
GenericLevel genericLevel;
//(...)
void getLevel(Group group) {
    ControlGroup controlGroup = new ControlGroup(group);
    controlGroup.getStatus(genericLevel, new GetGroupStatusCallback<GenericLevel>() {
        @Override
        public void success(Group group, GenericLevel genericLevel) {
        }
        @Override
        public void error(Group group, ErrorType errorType) {
        }
    });
}
```

### 7.15.3 Set Value for a Single Model from the Node

```
void set(int level, Element element, Group group) {
    ControlElement controlElement = new ControlElement(element, group);
    GenericLevel status = new GenericLevel();
    status.setLevel(16);
    ControlRequestParameters parameters = new ControlRequestParameters(1, 1, false);
    controlElement.setStatus(status, parameters, new SetElementStatusCallback<GenericLevel>() {
        @Override
        public void success(Element element, Group group, GenericLevel genericLevel) {
        }
        @Override
        public void error(Element element, Group group, ErrorType errorType) {
        }
    });
}
7.15.4 Set Value for All Specific Models Bound with the Group

Note: Models have to be subscribed to a given group.

```java
void set(int level, Group group) {
    ControlGroup controlGroup = new ControlGroup(group);
    GenericLevel status = new GenericLevel();
    status.setLevel(level);

    ControlRequestParameters parameters = new ControlRequestParameters(1, 1, false);

    controlGroup.setStatus(status, parameters, new SetGroupStatusCallback<GenericLevel>() {
        @Override
        public void success(Group group, GenericLevel genericLevel) {
        }
        @Override
        public void error(Group group, ErrorType errorType) {
        }
    });
}
```

7.15.5 Control Sensor Models

Despite the fact that the SIG models already have logic for set/get states (see the previous four sections), a separate logic controls Sensor models. The Sensor model is available as a SIG model with a Sensor model identifier in the Device Composition Data.

Currently the Sensor API allows:
- Get sensor descriptors from the Sensor Server model
- Get measurement data as single Sensor Data state from the given Sensor from the Sensor Server model
- Get measurement data as a Sensor Column or series of columns
- Set/Get Sensor cadence
- Set/Get Sensor settings

Currently the Sensor API does NOT allow:
- Handle publications that are automatically sent by the Sensor model as: base data, columns, series and so on.

To set up publication/subscription settings use the API prepared for SIG models (see sections 7.13.1 SIG model and 7.14.1 SIG Model, respectively).

To set up Sensor model bindings use the API prepared for SIG models (see sections 7.11 Bind a Model with A Group and 7.12 Unbind a Model from a Group).

7.15.5.1 Get Sensor Model Values

7.15.5.1.1 Get Sensor Descriptors from the Node

```java
propertyID – Insert a Sensor Property ID to receive the Sensor Descriptor for this ID. Put 0 to receive all Sensor Descriptors from the Sensor Server model.

void getSensorDescriptors(Element element, Group group, int propertyID) {
    ControlElement controlElement = new ControlElement(element, group);

    SensorPropertiesGetForDescriptor(propertyID);
```
controlElement.getSensorStatus(new SensorDescriptors(), properties, new GetElementSensorStatusCallback<SensorDescriptors>() {
    @Override
    public void success(Element element, Group group, SensorDescriptors value) {
        //handle success
    }

    @Override
    public void error(Element element, Group group, ErrorType error) {
        //handle error
    }
}));

After Sensor Descriptors are successfully downloaded, they will be available in the Element. They will be locally available because descriptors do not change over the time. Sensor Property ID is kept by Descriptor as in the Mesh Model Bluetooth® Specification.

Structure:
Element element;
(...)
Set<Sensor> sensors = element.sensorsFromSensorServerModel();
(...)
Descriptor descriptor = sensor.getDescriptor();

7.15.5.1.2 Get Sensor State from the Node

propertyID – Insert a Sensor Property ID to receive the Sensor Status from the Sensor with this ID. Put 0 to receive states of all Sensors from the Sensor Server model.

Measurement data received in the SensorStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.4 Sensor Data from the Mesh Model Bluetooth® Specification).

```java
void getSensorStatus(Element element, Group group, int propertyID) {
    ControlElement controlElement = new ControlElement(element, group);
        SensorPropertiesGetForSensorStatus(propertyID);
    controlElement.getSensorStatus(new SensorStatus(), properties,
        new GetElementSensorStatusCallback<SensorStatus>() {
            @Override
            public void success(Element element, Group group, SensorStatus value) {
                //handle success
            }

            @Override
            public void error(Element element, Group group, ErrorType error) {
                //handle error
            }
        });
}
```

7.15.5.1.3 Get Sensor Column from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Series Status from the Sensor with this ID.

rawValueX – Insert raw value identifying a column you want to get the value from.

Measurement data received in the SensorColumnStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.5 Sensor Series Column from the Mesh Model Bluetooth® Specification).

```java
void getSensorColumn(Element element, Group group, int propertyId, byte[] rawValueX) {
    ...
}
ControlGroup controlGroup = new ControlGroup(group);

SensorPropertiesGetForSensorColumnStatus(propertyId, rawValueX);
MeshTask meshTask = controlGroup.getSensorStatus(new SensorColumnStatus(), properties, 
new GetGroupSensorStatusHandler<SensorColumnStatus>() { 
    @Override
    public void success(Element element, Group group, SensorColumnStatus value) {
        // handle success
    }
    @Override
    public void error(Element element, Group group, ErrorType error) {
        // handle error
    }
});

7.15.5.1.4 Get Sensor Series from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Series Status from the Sensor with this ID.

rawValueX1 – Insert raw value identifying a starting column of series.

rawValueX2 – Insert raw value identifying an ending column of series.

Measurement data received in the SensorSeriesStatus is a sequence of columns with a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.5 Sensor Series Column from the Mesh Model Bluetooth® Specification).

void getSensorStatus(Element element, Group group, int propertyID, byte[] rawValueX1, byte[] rawValueX2) {
    ControlElement controlElement = new ControlElement(element, group);

    SensorPropertiesGetForSensorSeriesStatus(propertyId, rawValueX1, rawValueX2);

    controlElement.getSensorStatus(new SensorSeriesStatus(), properties, 
    new GetElementSensorStatusCallback<SensorSeriesStatus>() { 
        @Override
        public void success(Element element, Group group, SensorSeriesStatus value) {
            // handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            // handle error
        }
    });
}

7.15.5.1.5 Get Sensor Cadence from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Cadence Status from the Sensor with this ID.

Measurement data received in the SensorCadenceStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

void getSensorCadence(Element element, Group group, int propertyID) {
    ControlElement controlElement = new ControlElement(element, group);

    SensorPropertiesGetForSensorCadenceStatus(propertyId);
controlElement.getSensorStatus(new SensorCadenceStatus(), properties, new GetElementSensorStatusCallback<SensorCadenceStatus>() {
    @Override
    public void success(Element element, Group group, SensorCadenceStatus value) {
        //handle success
    }
    @Override
    public void error(Element element, Group group, ErrorType error) {
        //handle error
    }
});

7.15.5.1.6 Get Sensor Settings from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Settings Status from the Sensor with this ID.

Measurement data received in the SensorSettingStatus is an array of 16-bit integer values representing IDs of Sensor Setting Properties.

void getSensorSettings(Element element, Group group, int propertyID) {
    ControlElement controlElement = new ControlElement(element, group);
    SensorPropertiesGetForSensorSettingsStatus(propertyId);
    controlElement.getSensorStatus(new SensorSettingsStatus(), properties, new GetElementSensorStatusCallback<SensorSettingsStatus>() {
        @Override
        public void success(Element element, Group group, SensorSettingsStatus value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}

7.15.5.1.7 Get Sensor Setting from the Node

propertyID - Insert a Sensor Property ID to receive the Sensor Setting Status from the Sensor with this ID.

settingID - Insert a Sensor Setting Property ID to determine a specific setting within a sensor that will be received.

Measurement data received in the SensorSettingStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

void getSensorSetting(Element element, Group group, int propertyID, int settingID) {
    ControlElement controlElement = new ControlElement(element, group);
    SensorPropertiesGetForSensorSettingStatus(propertyID, settingID);
    controlElement.getSensorStatus(new SensorSettingsStatus(), properties, new GetElementSensorStatusCallback<SensorSettingsStatus>() {
        @Override
        public void success(Element element, Group group, SensorSettingsStatus value) {
            //handle success
        }
    });
}
@Override
public void error(Element element, Group group, ErrorType error) {
    //handle error
}
}

7.15.5.1.8 Get Sensor Descriptors from All Nodes in the Group

propertyID – Insert a Sensor Property ID to receive the Sensor Descriptor for this ID. Put 0 to receive all Sensor Descriptors from the Sensor Server model.

successHandler - This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call meshTask.cancel().

element – The response will be received in this SBMElement.

void getSensorDescriptors(Group group, int propertyID) {
    ControlGroup controlGroup = new ControlGroup (group);
    SensorPropertiesGetForDescriptor(propertyID);
    MeshTask meshTask = controlGroup.getSensorStatus(new SensorDescriptors(), properties,
    new GetGroupSensorStatusHandler<SensorDescriptors>() {
        @Override
        public void success(Element element, Group group, SensorDescriptors value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}

After Sensor Descriptors are successfully downloaded, they will be available in the Element. They will be locally available because descriptors do not change over the time. Sensor Property ID is kept by Descriptor as in the Mesh Model Bluetooth® Specification.

Structure:
Element element;
(...)
Set<Sensor> sensors = element.sensorsFromSensorServerModel();
(...)
Descriptor descriptor = sensor.getDescriptor();

7.15.5.1.9 Get Sensor State from All Nodes in the Group

propertyID – Insert a Sensor Property ID to receive the Sensor Status from the Sensor with this ID. Put 0 to receive states of all Sensors from the Sensor Server model.

Measurement data received in the SensorStatus has a structure defined by Bluetooth SIG Mesh Specification (see section 4.1.4 Sensor Data from the Mesh Model Bluetooth® Specification).

successHandler - This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call meshTask.cancel().

element – The response will be received in this Element.

void getSensorStatus(Group group, int propertyID) {
    ControlGroup controlGroup = new ControlGroup(group);
    ...
    SensorPropertiesGetForSensorStatus(propertyID);

MeshTask meshTask = controlGroup.getSensorStatus(new SensorStatus(), properties,
    new GetElementSensorStatusCallback<SensorStatus>()
    { 
    @Override
    public void success(Element element, Group group, SensorStatus value) {
      //handle success
    }
    
    @Override
    public void error(Element element, Group group, ErrorType error) {
      //handle error
    }
  });
}

7.15.5.1.10 Get Sensor Series from All Nodes in the Group

propertyID - Insert a Sensor Property ID to receive the Sensor Series Status from the Sensor with this ID.

rawValueX1 – Insert raw value identifying a starting column of series.

rawValueX2 – Insert raw value identifying an ending column of series.

Measurement data received in the SensorSeriesStatus is a sequence of columns with a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.5 Sensor Series Column from the Mesh Model Bluetooth® Specification).

successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call meshTask.cancel().

element – The response will be received in this Element.

    void getSensorSeries(Group group, int propertyId, byte[] rawValueX1, byte[] rawValueX2) {
      ControlGroup controlGroup = new ControlGroup(group);

          SensorPropertiesGetForSensorSeriesStatus(propertyId, rawValueX1, rawValueX2);

      MeshTask meshTask = controlGroup.getSensorStatus(new SensorSeriesStatus(), properties,
          new GetGroupSensorStatusHandler<SensorSeriesStatus>()
          { 
          @Override
          public void success(Element element, Group group, SensorSeriesStatus value) {
            //handle success
          }
          
          @Override
          public void error(Element element, Group group, ErrorType error) {
            //handle error
          }
        });
    }
}

7.15.5.1.11 Get Sensor Column from All Nodes in the Group

propertyID - Insert a Sensor Property ID to receive the Sensor Series Status from the Sensor with this ID.

rawValueX – Insert raw value identifying a column you want to get the value from.

Measurement data received in the SensorColumnStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.5 Sensor Series Column from the Mesh Model Bluetooth® Specification).
successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call meshTask.cancel().

element – The response will be received in this Element.

```java
void getSensorColumn(Group group, int propertyId, byte[] rawValueX1) {
    ControlGroup controlGroup = new ControlGroup(group);
            SensorPropertiesGetForSensorColumnStatus(propertyId, rawValueX1);
    MeshTask meshTask = controlGroup.getSensorStatus(new SensorColumnStatus(), properties,
            new GetGroupSensorStatusHandler<SensorColumnStatus>() {
                @Override
                public void success(Element element, Group group, SensorColumnStatus value) {
                    //handle success
                }
                @Override
                public void error(Element element, Group group, ErrorType error) {
                    //handle error
                }
});
}
```

### 7.15.5.1.12 Get Sensor Cadence from All Nodes in the Group

**propertyId** - Insert a Sensor Property ID to receive the Sensor Cadence Status from the Sensor with this ID.

Measurement data received in the SensorCadenceStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

successHandler – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call meshTask.cancel().

element – The response will be received in this Element.

```java
void getSensorCadence(Group group, int propertyId) {
    ControlGroup controlGroup = new ControlGroup(group);
            SensorPropertiesGetForSensorCadenceStatus(propertyId);
    MeshTask meshTask = controlGroup.getSensorStatus(new SensorCadenceStatus(), properties,
            new GetGroupSensorStatusHandler<SensorCadenceStatus>() {
                @Override
                public void success(Element element, Group group, SensorCadenceStatus value) {
                    //handle success
                }
                @Override
                public void error(Element element, Group group, ErrorType error) {
                    //handle error
                }
});
}
```

### 7.15.5.1.13 Get Sensor Setting from All Nodes in the Group

**propertyId** - Insert a Sensor Property ID to receive the Sensor Setting Status from the Sensor with this ID.

**settingID** - Insert a Sensor Setting Property ID to determine a specific setting within a sensor that will be received.
Measurement data received in the SensorSettingStatus has a structure defined by the Bluetooth SIG Mesh Specification (see section 4.1.2 Sensor Setting from the Mesh Model Bluetooth® Specification).

**successHandler** – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call `meshTask.cancel()`.

**element** – The response will be received in this Element.

```java
void getSensorSetting(Group group, int propertyId, int settingId) {
    ControlGroup controlGroup = new ControlGroup(group);
    SensorPropertiesGetForSensorSettingStatus(propertyId, settingId);
    MeshTask meshTask = controlGroup.getSensorStatus(new SensorSettingStatus(), properties,
    new GetGroupSensorStatusHandler<SensorSettingStatus>() {
        @Override
        public void success(Element element, Group group, SensorSettingStatus value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}
```

### 7.15.5.1.14 Get Sensor Settings from All Nodes in the Group

**propertyID** - Insert a Sensor Property ID to receive the Sensor Settings Status from the Sensor with this ID.

Measurement data received in the SensorSettingStatus is an array of 16-bit integer values representing IDs of Sensor Setting Properties.

**successHandler** – This handler is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving responses call `meshTask.cancel()`.

**element** – The response will be received in this Element.

```java
void getSensorSettings(Group group, int propertyId) {
    ControlGroup controlGroup = new ControlGroup(group);
    SensorPropertiesGetForSensorSettingsStatus(propertyId);
    MeshTask meshTask = controlGroup.getSensorStatus(new SensorSettingsStatus(), properties,
    new GetGroupSensorStatusHandler<SensorSettingsStatus>() {
        @Override
        public void success(Element element, Group group, SensorSettingsStatus value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}
```
7.15.5.2 Set Sensor Setup Model Values

7.15.5.2.1 Set Sensor Cadence within the Node

SensorCadenceSet message requires filling 7 parameters describing cadence:

- **FastCadenceHigh** – defines the upper boundary of a range of measured quantities when the publishing cadence is increased.
- **FastCadenceLow** – defines the lower boundary of a range of measured quantities when the publishing cadence is increased.
- **FastCadencePeriodDivisor** – controls the increased cadence of publishing Sensor Status messages. Valid values are 0 – 15 only.
- **StatusMinInterval** – controls the minimum interval between publishing two consecutive Sensor Status messages. Valid values are 0 – 26 only.
- **StatusTriggerDeltaDown** – controls the negative change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.
- **StatusTriggerDeltaUp** – controls the positive change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.
- **StatusTriggerType** – defines the unit and format of the Status Trigger Delta Down and Status Trigger Delta Up. Valid values are: FORMAT_TYPE_DEFINED (format is based on Sensor Property ID) and UNITLESS (value is represented as a percentage change).

SensorCadenceSet message also requires filling 2 parameters concerning message:

- **SensorPropertyID** – determines ID of sensor to receive the message.
- **SensorMessageFlags** – determines whether acknowledge of sending message is required.

By default the message is unacknowledged. In order to set **SensorMessageFlags** it is required to pass a **SensorMessageFlags** object to the setter. Use the **setAcknowledgeRequired()** method of the **SensorMessageFlags** object to determine the required value of message flag.

For full details about parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.3 Sensor Cadence from the **Mesh Model Bluetooth® Specification**).

```java
void setSensorCadence(Element element, Group group) {
    ControlElement controlElement = new ControlElement(element, group);
    SensorCadenceSet cadenceSet = new SensorCadenceSet();
    cadenceSet.set[Property Name]([value]);
    SensorMessageFlags flags = new SensorMessageFlags();
    flags.setAcknowledgeRequired([true/false]);
    cadenceSet.setSensorMessageFlags(flags);
    controlElement.setSensorSetupValue(cadenceSet, new SetElementSensorStatusCallback() {
        @Override
        public void success(Element element, Group group, ControlValueGetSensorModel value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}
```

7.15.5.2.2 Set Sensor Setting within the Node

To set a specific setting within a sensor it is required to specify the ID of that sensor, the ID of the selected setting and data for that setting. It is also possible to determine flags for messages to the sensor.

Note that in order to set **SensorMessageFlags** it is required to pass a **SensorMessageFlags** object into the setter. Use the **setAcknowledgeRequired()** method of the **SensorMessageFlags** object to determine the required value of message flag.
void setSensorSetting(Element element, Group group) {
    ControlElement controlElement = new ControlElement(element, group);
    SensorSettingSet settingSet = new SensorSettingSet();
    settingSet.setSensorPropertyID([sensorID]);
    settingSet.setSettingPropertyID([settingID]);
    settingSet.setSettingRaw([Raw data for setting]);
    SensorMessageFlags flags = new SensorMessageFlags();
    flags.setAcknowledgeRequired([true/false]);
    settingSet.setSensorMessageFlags(flags);
    controlElement.setSensorSetupValue(settingSet, new SetElementSensorStatusCallback() {
        @Override
        public void success(Element element, Group group, ControlValueGetSensorModel value) {
            //handle success
        }
        @Override
        public void error(Element element, Group group, ErrorType error) {
            //handle error
        }
    });
}

7.15.5.2.3 Set Sensor Cadence within All Nodes in the Group

SensorCadenceSet message requires filling 7 parameters describing cadence:
- **FastCadenceHigh** – defines the upper boundary of a range of measured quantities when the publishing cadence is increased.
- **FastCadenceLow** – defines the lower boundary of a range of measured quantities when the publishing cadence is increased.
- **FastCadencePeriodDivisor** – controls the increased cadence of publishing Sensor Status messages. Valid values are 0 – 15 only.
- **StatusMinInterval** – controls the minimum interval between publishing two consecutive Sensor Status messages. Valid values are 0 – 26 only.
- **StatusTriggerDeltaDown** – controls the negative change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.
- **StatusTriggerDeltaUp** - controls the positive change of quantity measured by sensor. Calculation of this setting is based on StatusTriggerType.
- **StatusTriggerType** – defines the unit and format of the Status Trigger Delta Down and Status Trigger Delta Up. Valid values are: FORMAT_TYPE_DEFINED (format is based on Sensor Property ID) and UNITLESS (value is represented as a percentage change).

SensorCadenceSet message also requires filling 2 parameters concerning message:
- **SensorPropertyID** – determines ID of sensor to receive the message.
- **SensorMessageFlags** – determines whether acknowledge of sending message is required.

By default message is unacknowledged. In order to set SensorMessageFlags it is required to pass a SensorMessageFlags object to the setter. Use the setAcknowledgeRequired() method of the SensorMessageFlags object to determine required value of message flag.

For full details about parameters listed above see Bluetooth SIG Mesh Specification (section 4.1.3 Sensor Cadence from the Mesh Model Bluetooth® Specification).

void setSensorCadence(Group group) {
    ControlGroup controlGroup = new ControlGroup(group);
    SensorCadenceSet cadenceSet = new SensorCadenceSet();
    cadenceSet.set[Property Name]([value])
    SensorMessageFlags flags = new SensorMessageFlags();
flags.setAcknowledgeRequired([true/false]);
cadenceSet.setSensorMessageFlags(flags);

controlGroup.setSensorSetupValue(cadenceSet,
new SetGroupSensorStatusHandler<ControlValueGetSensorModel>() {
    @Override
    public void success(Element element, Group group, ControlValueGetSensorModel value) {
        //handle success
    }
    @Override
    public void error(Element element, Group group, ErrorType error) {
        //handle error
    }
});
}

7.15.5.2.4 Set Sensor Setting within All Nodes in the Group

To set a specific setting within a sensor it is required to specify the ID of that sensor, the ID of the selected setting and data for that setting. It is also possible to determine flags for messages to the sensor.

Note that in order to set SensorMessageFlags it is required to pass a SensorMessageFlags object into the setter. Use the setAcknowledgeRequired() method of SensorMessageFlags object to determine the required value of the message flag.

void setSensorSettingGroup(Group group) {
    ControlGroup controlGroup = new ControlGroup(group);

    SensorSettingSet settingSet = new SensorSettingSet();
    settingSet.setSensorPropertyID([sensorID]);
    settingSet.setSettingPropertyID([settingID]);
    settingSet.setSettingRaw([Raw data for setting]);

    SensorMessageFlags flags = new SensorMessageFlags();
    flags.setAcknowledgeRequired([true/false]);
    settingSet.setSensorMessageFlags(flags);

    controlGroup.setSensorSetupValue(settingSet,
new SetGroupSensorStatusHandler<ControlValueGetSensorModel>() {
    @Override
    public void success(Element element, Group group, ControlValueGetSensorModel value) {
        //handle success
    }
    @Override
    public void error(Element element, Group group, ErrorType error) {
        //handle error
    }
});
}

For more information see Bluetooth SIG Mesh Specification (section 4.2.10 Sensor Setting Set from the Mesh Model Bluetooth® Specification).

7.15.6 Register a Local Vendor Model (LocalVendorModel)

Registering a local vendor model hooks it to the mesh stack and enables the stack to pass incoming messages to the local vendor model. Typically, registration should be done one time for the local vendor model after initializing BluetoothMesh (see section 6.1 Initializing the BluetoothMesh).
7.15.6.1 Create LocalVendorSettings

Represents vendor registration settings.

The opcodes used in this operation are manufacturer-specific opcode numbers. They can be calculated by subtracting 0xC0 from the first byte of the 3-octet opcode.

Note: More information about Operation codes can be found in the Bluetooth SIG Documentation (Mesh Profile 3.7.3.1 Operation codes).

```java
byte[] opCodes; // Manufacturer-specific operation codes supported by Vendor Model

// (...)
LocalVendorSettingsMessageHandler messageHandler = new LocalVendorSettingsMessageHandler() {
    @Override
    public void message(LocalVendorModel model, int appKeyIndex, int sourceAddress, int destinationAddress, byte[] virtualAddress, byte[] message, int messageFlags) {
        //Callback for handling incoming vendor messages
    }
};

// (...)
LocalVendorSettings registerSettings = LocalVendorSettings(opCodes, localVendorSettingsMessage);
```

7.15.6.2 Create LocalVendorRegistrator

Used to set the registration settings of a local vendor model.

```java
LocalVendorModel localVendorModel;

// (...)
LocalVendorRegistrator vendorRegistrator = new LocalVendorRegistrator(localVendorModel);
```

7.15.6.2.1 Register a Local Vendor Model

Note: Typically, registration should be done one time for the local vendor model after initializing BluetoothMesh (see section 6.1 Initializing the BluetoothMesh).

```java
LocalVendorRegistrator vendorRegistrator;
LocalVendorSettings registerSettings;

// (...)
vendorRegistrator.register(registerSettings)
```

7.15.6.2.2 Unregister a Local Vendor Model

```java
LocalVendorRegistrator vendorRegistrator;

// (...)
vendorRegistrator.unregister()
```

7.15.7 Local Vendor Model Binding with Application Key

LocalVendorModel must be bound with ApplicationKey so that the Mesh Library can decrypt incoming messages from the mesh network. First, VendorModel should be bound with Group (see section 6.12 Bind a Model with A Group), because Group is the source of the ApplicationKey. Messages that come from the VendorModel from the Node are encrypted with ApplicationKey from this Group.
7.15.7.1 Bind Local Vendor Model with Application Key

This function creates binding between a local vendor model and an application key required to decrypt incoming messages.

```java
AppKey appKey;
LocalVendorModel localVendorModel;

// (...)
LocalVendorCryptoBinder cryptoBinder = new LocalVendorCryptoBinder(appKey);
cryptoBinder.bindAppKey(localVendorModel);
```

7.15.7.2 Unbind Local Vendor Model from Application Key

Remove an existing binding between a local vendor model and an application key, meaning that the local vendor model will no longer process messages encrypted using that key.

```java
AppKey appKey;
LocalVendorModel localVendorModel;

// (...)
LocalVendorCryptoBinder cryptoBinder = new LocalVendorCryptoBinder(appKey);
cryptoBinder.unbindAppKey(localVendorModel);
```

7.15.8 Manage Notifications from the VendorModel

VendorModel can be configured to publish notifications by Group address or directly by Node address. Configuration is done by adding publication settings to the model (see section 6.15 Add Publication Settings to a Model).

To receive notifications sent by the VendorModel from the Node the local stack must be able to collect those messages. The library must know what addresses it should follow. It is possible to set up the Group address or the direct Node address. The VendorModel must have set publication by the address of the Group to which it is bound or the address of the Node to which it belongs.

7.15.8.1 Sign Up for The Notifications

Method used to subscribe to the vendor model notifications.

Note: Notifications will come from the LocalVendorSettingsMessageHandler from the LocalVendorSettings, which should previously have been configured by LocalVendorRegister.

7.15.8.1.1 Notifications Come by Group Address

```java
Group group;
VendorModel vendorModel;

// (...)
VendorModelNotifications vendorNotifications = new VendorModelNotifications(group);
vendorNotifications.signUpForNotifications(vendorModel);
```

7.15.8.1.2 Notifications Come by Node Address

```java
Node node;
VendorModel vendorModel;

// (...)
VendorModelNotifications vendorNotifications = new VendorModelNotifications(node);
vendorNotifications.signUpForNotifications(vendorModel);
```
7.15.8.2 Sign Out from the Notifications

Method used to unsubscribe from the vendor model notifications.

7.15.8.2.1 Sign Out from Notifications from the Group Address

```java
Group group;
VendorModel vendorModel;

// (…)
VendorModelNotifications vendorNotifications = new VendorModelNotifications(group);
vendorNotifications.signOutFromNotifications(fromNotifications: vendorModel);
```

7.15.8.2.2 Sign Out from Notifications from the Node Address

```java
Node node;
VendorModel vendorModel;

// (…)
VendorModelNotifications vendorNotifications = new VendorModelNotifications(node);
vendorNotifications.signOutFromNotifications(fromNotifications: vendorModel);
```

7.15.9 Send Value to VendorModel

7.15.9.1 Create Implementation of the ControlValueSetVendorModel Protocol

Developers who use our library needs to create their own implementation of the ControlValueSetVendorModel interface. It will be used to send messages to the vendor model from the network.

Example:

```java
class CompanyControlSetVendorModel implements ControlValueSetVendorModel {

    CompanyControlSetVendorModel(LocalVendorModel vendor, byte[] messageToSend, FLAGS flags) {
        // (…)
    }

    @Override
    public LocalVendorModel getLocalVendorModel() {
        // (…)
    }

    @Override
    public byte[] getData() {
        // (…)
    }

    @Override
    public FLAGS getFlags() {
        // (…)
    }

    }
```
7.15.9.2 Prepare Message to Send

It is very important to prepare the message with the correct structure. The first part of the message structure must contain the Opcode (Operation code) that will be used to send this message. This Opcode must be supported by the VendorModel. The message structure must also contain the vendor company identifier that comes from the VendorModel to which the message will be sent.

Note: More information about Operation codes can be found in the Bluetooth SIG Documentation (Mesh Profile 3.7.3.1 Operation codes).

Example:

```java
VendorModel vendorModel;
byte[] messageToSend;

// (...)
int companyID = vendorModel.vendorCompanyIdentifier();

// (...)
ByteArrayOutputStream stream = new ByteArrayOutputStream();
stream.write(new byte[(0) | 0xC0]);
// In example my opcode is 0. 0xC0 must be HERE, reason can be found in the Bluetooth SIG
// Documentation(Mesh Profile 3.7.3.1 Operation codes)
stream.write(new byte[(companyID & 0x00ff)]);
stream.write(new byte[(companyID >> 8 & 0x00ff)]); //Add vendor company identifier
stream.write(messageToSend);
byte data[] = stream.toByteArray()
```

7.15.9.3 Send Prepared Message to the Single VendorModel on the Node

Note: Replay messages will be sent from the VendorModel if it supports that. Messages will be received in the LocalVendorSettingsMessageHandler from the LocalVendorSettings, which should previously have been configured by LocalVendorRegisterControl.

```java
Element element;
Group group;

// (...)
ControlElement controlElement = new ControlElement(element, group);

// (...)
byte[] messageToSend;
LocalVendorModel vendorModel;
FLAGS flags;

// (...)
CompanyControlSetVendorModel setVendorModel = new CompanyControlSetVendorModel(vendorModel,
messageToSend, flags);

// (...)
```
controlElement.setStatus(setVendorModel, new SetVendorElementStatusCallback() {
    @Override
    public void success(Element element, Group group) {
        //Action invoked when message is successfully sent.
    }

    @Override
    public void error(Element element, Group group, ErrorType error) {
        //Action invoked when message could not be sent.
    }
});

7.15.9.4 Send Prepared Message to the Group

Note: Models have to be subscribed to a given group.

Group group;

    // (...)

ControlGroup controlGroup = new ControlGroup(group);

    // (...)

byte[] messageToSend;
LocalVendorModel vendorModel;
FLAGS flags;

    // (...)

CompanyControlSetVendorModel setVendorModel = new CompanyControlSetVendorModel(vendorModel,
    messageToSend, flags);

    // (...)

    controlGroup.setStatus(setVendorModel, new SetVendorGroupStatusCallback() {
        @Override
        public void success(Group group) {
            //Action invoked when message is successfully sent.
        }

        @Override
        public void error(Group group, ErrorType error) {
            //Action invoked when message could not be sent.
        }
    });
7.16 Sequence Diagrams for Vendor Model Functionality

7.16.1 LocalVendorModel Initialization

7.16.1.1 Create LocalVendorModel
### 7.16.1.2 Register the LocalVendorModel in the Mesh Stack

Each LocalVendorModel should have created their own message handler. It is a place where messages will come from the VendorModel with the same vendor company identifier and vendor assigned model.

For example network has two nodes. Each one have VendorModel with the same ids. If all needed configuration will be done, messages from those two vendor models will be passed by the same message handler.

In the initializer of the LocalVendorSettings there is place for two parameters:

1) opCodes - supported operation codes by vendor model. Developers who are implementing mobile application should contact developers who are responsible for the application for the embedded mesh device to get know what opcodes they should use. It is also important to get know how those opcodes work because it defines how they should use vendor model on the application side.

2) messageHandler - it is handler which was created a step before.

Create LocalVendorRegistrar with local vendor model which will be operated by registrar.

Register local vendor model with created LocalVendorSettings.

Registering a local vendor model hooks it to the mesh stack and enables the stack to pass incoming messages to the local vendor model.
7.16.2 Configure the VendorModel

7.16.2.1 Bind the VendorModel with the Group

It is always needed to bind vendor model on the node with the group.
7.16.2.2 Send Publication Settings to the VendorModel

To configure the publication address for the notifications sent by the vendor model on the node you must send publication settings to this node.

7.16.2.2.1 Set the Provisioner Address as the Publication Address

![Diagram showing the process of setting publication settings to the vendor model](image-url)
7.16.2.2.2 Set the Group Address as the Publication Address

- **Pass the VendorModel via parameter to the initializer. The SubscriptionControl will operate with this model.**
- **Create SubscriptionControl object**
- **PublicationSettings object**
- **setPublicationSettings**
- **Asynchronous callback with the result of configuration.**

**In this case it is needed to setup VendorModel to publish data via Group address. To do it, call:**
```
PublicationSettings(group: group)
```
**Group is passed as group.**

**Pass PublicationSettings as a parameter. It will send message to the node with configuration to which address the vendor model should publish notifications. VendorModel which is needed for this configuration was passed during SubscriptionControl initialization.**
7.16.2.3 Send Subscription Settings to the VendorModel

When the vendor model on the node expects a message from the group address, you must send the subscription settings containing the address of this group to the vendor model on the node. Without that the vendor model will not receive a message sent by the group address,
7.16.3 Send Message to the Vendor Model on the Node

7.16.3.1 Send Message Directly to the Node

7.16.3.1.1 Message without Response

Diagram:
- Developer
- VendorModel
- Element
- `getVendorModels`
- `Set of vendor models`
- `getBindedGroups`
- `Array of binded groups`
- `Pick VendorModel which will be controlled`
- `Pick Group. ApplicationKey from this group will be used to encrypt message. All VendorModels with the same ids will be controlled from this group if they are configured for it.`
Create

ControlElement object

Pass picked Group and Element via parameter to initializer.

Parameters to pass to initializer:
1) vendorModel - it is a VendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags

Create

Message object

setMessage: status

Parameters to pass:
1) status - Message object
2) successCallback - will be called after send message with success
3) errorCallback - will be called if any error occur

callback

Asynchronous callback with the result of send message.
7.16.3.1.2 Message with Response

- **VendorModel**
  - Set of vendor models
  - Pick VendorModel which will be controlled
  - bindedGroups
    - Array of binded groups

- **Element**

- **BluetoothMeshConfiguration**
  - In one moment it is possible to create only one binding between the ApplicationKey and the LocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

- **LocalVendorCryptoBinder**
  - Bind ApplicationKey with LocalVendorModel
    - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and an needed application key to decrypt this message.

- **Developer**
  - getVendorModels
    - Set of vendor models
  - Pick LocalVendorModel which have the same ids as picked VendorModel
  - getLocalVendorModels
    - LocalVendorCryptoBinder object
  - bindAppKeyLocalVendorModel
7.16.3.2 Send Message via Group Address

7.16.3.2.1 Message without Response

Pick Group. ApplicationKey from this group will be used to encrypt message. All VendorModels with the same ids will be controlled from this group if they are configured for it.
Pass picked Group via parameter in the initializer.

**ControlGroup**

Create

ControlGroup object

Create

Message object

**Message: ControlValueSetVendorModel**

Parameters to pass to initializer:
1) vendorModel - it is a VendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags

setStatus: status

callback

Asynchronous callback with the result of Sending Message

Parameters to pass:
1) status - Message object
2) successCallback - will be called after sending message with success
3) errorCallback - will be called if any error occurs
7.16.3.2.2 Message with Response

- **VendorModel**
  - getVendorModels
  - Set of vendor models

- **Element**
  - getBinderGroups
  - Set of binder groups

- **BluetoothMeshConfiguration**
  - Pick VendorModel which will be controlled
  - Pass ApplicationKey from the given Group as parameter to initializer

- **LocalVendorCryptoBinder**
  - Pick Group. ApplicationKey from this group will be used to encrypt TX message and decrypt RX message.
  - Bind ApplicationKey with LocalVendorModel. For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and a needed application key to decrypt this message.

- **Developer**
  - In one moment it is possible to create only one binding between the ApplicationKey and the LocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

- **LocalVendorCryptoBinder object**
  - getLocalVendorModels
  - Set of vendor models

- **bindAppKey localVendorModel**
7.16.4 Subscribe to Notifications Published by the Vendor Model From the Node

7.16.4.1 Messages are Sent Directly to the Provisioner

In one moment it is possible to create only one binding between the ApplicationKey and the LocalVendorModel. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

Pass ApplicationKey from the given Group as parameter to initializer.

Bind ApplicationKey with LocalVendorModel. For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side. This function creates binding between a local vendor model and an needed application key to decrypt this message.
**AN1200: Bluetooth® Mesh for iOS and Android ADK**

**Bluetooth Mesh API Reference for Android**

---

Diagram:

- **Developer**
  - `getNodes`
  - `Set of nodes`
  - **Pick Node.**
    - This Node is needed to define node address which will be observed by provisioner. It is needed to capture notifications from the vendor model.
  - **Create**
    - `VendorModelNotifications object`
    - `Pass VendorModel via parameter. From this VendorModel notifications will be accepted`
  - `signUpForNotifications vendorModel`
  - `callback`
  - `callback`
  - `callback`
  - `LocalVendorSettingsMessageHandler`
    - `Incoming messages which are published by the vendor model to the provisioner as notifications. LocalVendorSettingsMessageHandler was created during LocalVendorModel configuration by developer.`

---

silabs.com | Building a more connected world.
7.16.4.2 Messages are Sent via Group Address to the Provisioner

- Pick the Vendor Model which will be controlled.
  - Get a set of vendor models.

- Pick a Group, Application Key from this group will be used to decrypt incoming messages.
  - Array of banded groups.

- Bind Application Key with Local Vendor Model.
  - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side.
  - This function creates binding between a local vendor model and an needed application key to decrypt this message.

- In one moment it is possible to create only one binding between the Application Key and the Local Vendor Model. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

- Pass Application Key from the given Group as parameter to initializer.

- Create a Local Vendor Crypto Binder object.
  - Array of vendor models.

- Bind Application Key with Local Vendor Model.
  - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side.
  - This function creates binding between a local vendor model and an needed application key to decrypt this message.

- Bind Application Key with Local Vendor Model.
  - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side.
  - This function creates binding between a local vendor model and an needed application key to decrypt this message.

- In one moment it is possible to create only one binding between the Application Key and the Local Vendor Model. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.

- Pass Application Key from the given Group as parameter to initializer.

- Create a Local Vendor Crypto Binder object.
  - Array of vendor models.

- Bind Application Key with Local Vendor Model.
  - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side.
  - This function creates binding between a local vendor model and an needed application key to decrypt this message.

- Bind Application Key with Local Vendor Model.
  - For a local vendor model to receive an incoming message, the message must be decrypted using an application key which is bound to the vendor model on the node side.
  - This function creates binding between a local vendor model and an needed application key to decrypt this message.

- In one moment it is possible to create only one binding between the Application Key and the Local Vendor Model. In case when this binding is needed in the next request, it is not needed to create next binding. Bluetooth mesh library returns error with proper information if someone want to duplicate binding.
VendorModelNotifications object

Pass VendorModel via parameter. From this VendorModel notifications will be accepted.

signUpForNotifications:vendorModel

Pass picked Group via parameter to intializer

Incoming messages which are published by the vendor model to the provisioner via group address as a notifications.
LocalVendorSettingsMessageHandler was created during LocalVendorModel configuration by developer.