AN1200.0: *Bluetooth*® Mesh 1.x 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.

The Bluetooth Mesh mobile app is intended to demonstrate the Silicon Labs Bluetooth Mesh technology together with the Bluetooth Mesh SDK sample apps. The mobile app is a reference app for the Bluetooth Mesh mobile ADK but it should not be taken as a starting point for customers to create their own mobile apps.

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
- Getting started with development
- Bluetooth mesh structure overview
- ADK use cases
- API references for iOS and Android
- Model descriptions
- Import and Export instructions
- Using multiple networks
- Code examples

Table of Contents

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 1-1. Features of the Bluetooth Mesh Android and iOS Stacks

Feature	Android Value	iOS Value
Bluetooth mesh stack version	1.7.5	
Simultaneous GATT connections	1	
Mesh bearers	GATT	
Supported node types	Proxy Relay Low Power	Proxy Relay Low Power
Mesh limits	(see section 8.2.3 Set Up Mesh Limits)	(see section 7.2.3 Set Up Mesh Limits)

Feature	Android Value	iOS Value
Supported mesh models	Generic OnOff Server 0x1000 Generic OnOff Client 0x1001 Generic Level Server 0x1002 Generic Level Client 0x1003 Generic Default Transition Time Server 0x1004 Generic Default Transition Time Client 0x1005 Generic Power OnOff Server 0x1006 Generic Power OnOff Setup Server 0x1007 Generic Power OnOff Client 0x1008 Generic Power Level Server 0x1009 Generic Power Level Setup Server 0x100A Generic Power Level Client 0x100B Generic Battery Server 0x100C Generic Battery Client 0x100D Generic Location Server 0x100E Generic Location Setup Server 0x100F Generic Location Client 0x1010 Generic Admin Property Server 0x1011 Generic Manufacturer Property Server 0x1012 Generic User Property Server 0x1013 Generic Client Property Server 0x1014 Generic Property Client 0x1015 Light Lightness Server 0x1300 Light Lightness Setup Server 0x1301 Light Lightness Client 0x1302 Light CTL Server 0x1303 Light CTL Setup Server 0x1304 Light CTL Client 0x1305 Light CTL Temperature Server 0x1306 Light LC Server 0x130F Light LC Setup Server 0x1310 Light LC Client 0x1311 Scene Server 0x1203 Scene Setup Server 0x1204 Scene Client 0x1205 Sensor Server 0x1100 Sensor Setup Server 0x1101 Sensor Client 0x1102 Scheduler Server 0x1206 Scheduler Setup Server 0x1207 Scheduler Client 0x1208 Time Server 0x1200 Time Setup Server 0x1201 Time Client 0x1202	Generic OnOff Server 0x1000 Generic OnOff Client 0x1001 Generic Level Server 0x1002 Generic Level Client 0x1003 Generic Default Transition Time Server 0x1004 Generic Default Transition Time Client 0x1005 Generic Power OnOff Server 0x1006 Generic Power OnOff Setup Server 0x1007 Generic Power OnOff Client 0x1008 Generic Power Level Server 0x1009 Generic Power Level Setup Server 0x100A Generic Power Level Client 0x100B Generic Battery Server 0x100C Generic Battery Client 0x100D Generic Location Server 0x100E Generic Location Setup Server 0x100F Generic Location Client 0x1010 Generic Admin Property Server 0x1011 Generic Manufacturer Property Server 0x1012 Generic User Property Server 0x1013 Generic Client Property Server 0x1014 Generic Property Client 0x1015 Light Lightness Server 0x1300 Light Lightness Setup Server 0x1301 Light Lightness Client 0x1302 Light CTL Server 0x1303 Light CTL Setup Server 0x1304 Light CTL Client 0x1305 Light CTL Temperature Server 0x1306 Light LC Server 0x130F Light LC Setup Server 0x1310 Light LC Client 0x1311 Scene Server 0x1203 Scene Setup Server 0x1204 Scene Client 0x1205 Sensor Server 0x1100 Sensor Setup Server 0x1101 Sensor Client 0x1102 Scheduler Server 0x1206 Scheduler Setup Server 0x1207 Scheduler Client 0x1208 Time Server 0x1200 Time Setup Server 0x1201 Time Client 0x1202
Supported GATT services	Provisioning, Proxy	Provisioning, Proxy

Table 1-2. Open Source Licenses Used

Feature	License	Comment
Mbed TLS	Apache License 2.0	Used for AES and ECDH and other cryptographic algorithms.
GSON (Android only)	Apache License 2.0	Used to store and load the Bluetooth mesh and device database to the Android secure storage.

2 Prerequisites for Development

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

Basic understanding of Bluetooth LE and Bluetooth mesh 1.0 specifications and technology.

Silicon Labs Bluetooth mesh learning center

Silicon Labs Bluetooth mesh technology white paper

Bluetooth mesh profile 1.0 specification

Bluetooth mesh model 1.0 specification (application layer)

Bluetooth 5.0 core specification

Simplicity Studio 4 and Bluetooth mesh SDK 1.7.5 or newer are installed.

- Download
- Getting started

You have a Blue Gecko SoC (SLWSTK6020B) or module WSTKs to act as Bluetooth mesh nodes.

- Recommended SoCs and Modules are: EFR32BG13 or EFR32BG12 SoCs or BGM13P or BGM13S module because they have 512 kB to 1024 kB of flash available.
- SLWSTK6020B User Guide

2.1 iOS

You have installed the Bluetooth Mesh ADK. It is available through the Simplicity Studio 4 Package Manager. Release and Debug versions of the Framework are provided. Use the Debug version to get more comprehensive and detailed logging.

- Bluetooth Mesh iOS Release Framework:

```
/Applications/Simplicity\ Studio.app/Contents/Eclipse/
developer/sdks/blemesh/v1.7/app/bluetooth/ios/Release/BluetoothMesh.xcframework
```

- Bluetooth Mesh iOS Debug Framework:

```
/Applications/Simplicity\ Studio.app/Contents/Eclipse/
developer/sdks/blemesh/v1.7/app/bluetooth/ios/Debug/BluetoothMesh.xcframework
```

Bluetooth Mesh iOS API Documentation:

```
/Applications/Simplicity\ Studio.app/Contents/Eclipse/
developer/sdks/blemesh/v1.7/app/bluetooth/ios/docs/index.html
```

Source code of the iOS reference application:

```
/Applications/Simplicity\ Studio.app/Contents/Eclipse/
developer/sdks/blemesh/v1.7/app/bluetooth/ios_application
```

The application is available in iOS AppStore: Bluetooth Mesh by Silicon Labs

Xcode 12 or newer is installed.

This ADK supports iOS versions 11, 12, 13 and 14.

The following devices have been tested with this ADK:

- iPhone 6 Plus; model MGA92PK/A; iOS 12.4.4
- iPad Pro; model ML0F2FD/A; iOS 13.4.1
- iPad Mini 4; model MK9P2FD/A; iOS 13.3.1
- iPhone 11; model MWLT2PM/A; iOS 13.3
- iPhone SE; model MHGQ3PM/A; iOS 14.3

You understand Bluetooth LE operation on iOS.

- Bluetooth LE documentation for iOS
- Bluetooth LE API documentation

2.1.1 Complying with Encryption Export Regulations

Every app submitted to TestFlight or the App Store is uploaded to a server in the United States. It is a developer responsibility to make sure that the uploaded app is fully legal and contains all necessary information. For that reason, each developer should become familiar with Encryption Export Regulations. If the app uses, accesses, contains, implements, or incorporates encryption, this is considered an export of encryption software, which means that the app is subject to U.S. export compliance requirements, as well as the import compliance requirements of the countries where the app is distributed.

More detailed explanation can be found here: Encryption Export Regulations

The following authentication, encryption and hash algorithms are used by the Bluetooth Mesh ADK:

- AES, 256 bit
- AES CCM, 128 bit
- AES ECB, 128 bit
- Elliptic Curve Diffie-Hellman

2.2 Android

You have installed the Bluetooth Mesh ADK. It is available through the Simplicity Studio Package Manager.

- macOS
  - Bluetooth Mesh Android Framework:

    ```
    /Applications/Simplicity\ Studio.app/Contents/Eclipse/
    developer/sdks/blemesh/v1.7/app/bluetooth/android/
    ```

  - Bluetooth Mesh Android API Documentation:

    ```
    /Applications/Simplicity\ Studio.app/Contents/Eclipse/
    developer/sdks/blemesh/v1.7/app/bluetooth/android/javadoc/index.html
    ```

  - Source code of the Android reference application:

    ```
    /Applications/Simplicity\ Studio.app/Contents/Eclipse/
    developer/sdks/blemesh/v1.7/app/bluetooth/android_application
    ```

- Windows
  - Bluetooth Mesh Android Framework:

    ```
    C:\SiliconLabs\SimplicityStudio\v4\
    developer\sdks\blemesh\v1.7\app\bluetooth\android
    ```

  - Bluetooth Mesh Android API Documentation

    ```
    C:\SiliconLabs\SimplicityStudio\v4\
    developer\sdks\blemesh\v1.7\app\bluetooth\android\javadoc
    ```

  - Source code of the Android reference application:

    ```
    C:\SiliconLabs\SimplicityStudio\v4\
    developer\sdks\blemesh\v1.7\app\bluetooth\android_application
    ```

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

Android Studio is installed.

- Download
- Instruction

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

The following devices have been tested with the ADK:

- Pixel 4 XL, Android 11
- Redmi Note 8T, Android 9
- One Plus 6, Android 10
- Huawei P20 lite, Android 9
- Nokia 8.3 5G, Android 10

You understand Bluetooth LE operation on Android as well as the JNI interface.

- Bluetooth LE documentation for Android
- Bluetooth LE API documentation
- Android JNI documentation

2.2.1 Known Bluetooth Issues

While developing applications using Bluetooth LE for Android devices many problems can occur. Unfortunately, troubleshooting is not as straightforward as for the iOS. This section describes collected information how Bluetooth LE on Android works, common issues, and advice on how to solve them. It can help you to develop your application faster.

Working with Bluetooth LE on Android is difficult, because:

- Device manufacturers make changes to the Android Bluetooth LE stack. Your application can work well on one device, but could have problems on another.
- Documentation on Bluetooth LE describes only basic concepts, but does not provide enough information about managing connections, need for queuing operations, or dealing with bugs.
- Older versions of Android as 4.X, 5.X and 6.X have some known bugs, which you have to handle if you want to support them. Newer versions are much better, but unfortunately still have some problems.

2.2.1.1 Scanning

Scanning for Bluetooth LE devices is power consuming. Four scan modes are available:

- SCAN_MODE_BALANCED – good trade-off between scan frequency and power consumption
- SCAN_MODE_LOW_LATENCY – highest scanning frequency
- SCAN_MODE_LOW_POWER – default scan mode consuming the least power
- SCAN_MODE_OPPORTUNISTIC – application that is using this mode will get scan results if another application is scanning (it does not start its own scanning)

Some applications may scan continuously, which would consume the phone's battery power. In order to limit this Android has implemented changes related to scanning. In Android 7.0 and newer versions there is protection against Bluetooth LE scanning abuse. If your app starts and stops Bluetooth LE scans more than 5 times within 30 seconds, scan results will not be received temporarily. Moreover, starting with Android 7.0, you can perform one scan with a maximum time of 30 minutes. After this time Android will change the scan mode to SCAN_MODE_OPPORTUNISTIC. As of Android 8.1, if you do not set any ScanFilters scanning will be paused when the user turns off the screen, and will resume after the screen is turned on again.

Remember that the scanning process has to be stopped in your application. If you know the devices the user is looking for, stop the process when all devices are found. If you do not know which devices the user is looking for, stop scanning after a fixed period. Also consider stopping the scanning process if the user goes to another Activity or your application goes background.

2.2.1.2 Connecting

Some phones have problems with connecting during scanning, so it would be better to stop scanning if you do not need to find another device. It is also recommended to wait about 500 milliseconds after stopping the scan before trying to connect to a device in order to avoid GATT_ERROR.

2.2.1.2.1 Auto connect

When you get a proper *BluetoothDevice* object from *ScanResult* you can connect to it by calling one of *connectGatt()* method on *BluetoothDevice*. All versions of this method contain a parameter named *autoConnect*. Official documentation describes it only as “Boolean: It determines whether to directly connect to the remote device (false) or to automatically connect as soon as the remote device becomes available (true).”

When you connect to a device with *autoConnect* set to false (direct connect) Android will try to connect to the device with a 30 second timeout. After that (if there was no other callback) you will receive an update with status GATT_ERROR (code 133). If there are pending connection attempts with *autoConnect* set to true they will be suspended for this time. This direct connect attempt will not be executed until another pending direct connect is finished. A direct connect attempt usually takes less time to succeed than an auto connect one.

Android waits until it sees this device and connects when it is available. Using auto connect allows you to have more than one pending connection at the same time. These connections have no timeout, but they will be canceled when Bluetooth is turned off. If you are using *autoConnect* set to true, you could be able to reconnect to the device as well. But the device must be in Bluetooth cache or be bonded before. Remember that turning Bluetooth off, rebooting your phone or manually clearing cache in settings menu will clear device information, so check the cache before attempting to reconnect.

2.2.1.3 Managing a Connection

2.2.1.3.1 Connection State

After trying to connect to a device with the *connectGatt()* method you should be informed about the result with the *onConnectionStateChange* callback. It provides information about *status* and *newState*, which you will use to perform appropriate steps.Remember to use the *close()* method on the *BluetoothGatt* object if the status is different than GATT_SUCCESS, or it is GATT_SUCCESS and the state is equal to STATE_DISCONNECTED, which means that device was successfully disconnected. If you do not call *close()* the client registered for this connection will not be removed. Once 30 clients are reached (usually 5 are used by default after rebooting the phone) the user will not be able to connect to another device (until they clear the cache).

2.2.1.3.2 Changing MTU

Maximum Transmission Unit (MTU) determines the maximum length of the data packet sent between phone and Bluetooth LE device. You can request changing MTU size after successfully connecting with the device, before exchanging data with it. The default value of MTU is 23 (GATT_DEF_BLE_MTU_SIZE), but usually 3 bytes contain ATT headers, so only 20 bytes can be sent. Change MTU by calling *requestMtu(size)* on the *BluetoothGatt* object, where *size* parameter is the new MTU length. Remember that the maximum available value is 517 (GATT_MAX_MTU_SIZE). If calling *requestMtu(size)* returns true, wait for the *onMtuChanged* callback with the result.

2.2.1.3.3 Discovering Services

Remember also that many Bluetooth LE operations are asynchronous and you need to wait for a callback to perform next operation. For example, after getting *onConnectionStateChange* with *status* GATT_SUCCESS and *newState* STATE_CONNECTED you need to call *discoverServices()*. It returns true if service discovery started and you have to wait for the *onServicesDiscovered* callback containing the status of this process. If you receive GATT_SUCCESS you can, for example, read/write characteristics, but if the result was not successful you need to disconnect from the device, because without services discovered you cannot perform those operations.

2.2.1.3.4 Reading/Writing Characteristics

These operations are also asynchronous and you can perform only one operation at a time. To solve this, use a queue for your operations. Add the next operation to it and, when the first is completed, it is removed from queue and the next command is executed.

When writing data to a characteristic you can specify the write type. There are two available types: WRITE_TYPE_DEFAULT and WRITE_TYPE_NO_RESPONSE. Bluetooth Mesh supports only WRITE_TYPE_NO_RESPONSE.

2.2.1.3.5 Disconnecting

In Android you might have problems with performing some operations, as reconnecting, after improperly disconnecting from a device. There is a timeout while the phone continues opening connection events and the device is not fully disconnected, so you could have trouble connecting to it again. In Android this timeout was hardcoded to 20 seconds and has been changed to 5 seconds in Android 10,

so it can take a lot of time until you are notified about the closed connection. In iOS this usually takes less than 1 second. If you want to reconnect immediately after disconnect you could get status code 22, so it would be better to wait about 500 milliseconds before the connection attempt.

2.2.1.4 Errors

Many errors can be received on some callback when working with a Bluetooth LE device. Unfortunately, not all of them have descriptions to help you to determine the problem. A common error is status 133 named GATT_ERROR. Unfortunately, no information about it is in *BluetoothGatt* class documentation. If you got this error, the problem could be one of the following:

- You try to connect with *autoConnect* set to false and receive the error after the 30 second timeout.
- After disconnecting from the device you do not invoke *close()* so you get the error when next trying to connect.
- The Bluetooth cache contains some invalid data, so restart your phone.
- You use a device that has problems with Bluetooth LE. Some models, for example older Huawei phones, are known to have low Bluetooth LE quality. Try using another phone.
- There was a problem on the Bluetooth side. After calling *close()* and waiting a little time, try connecting again.

2.2.1.5 Devices with Low Bluetooth LE Quality

The following devices are known as ones commonly to have problems with Bluetooth LE:

- Huawei P8 Lite
- Huawei P9 Lite
- Motorola G5s Plus (XT1805)

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

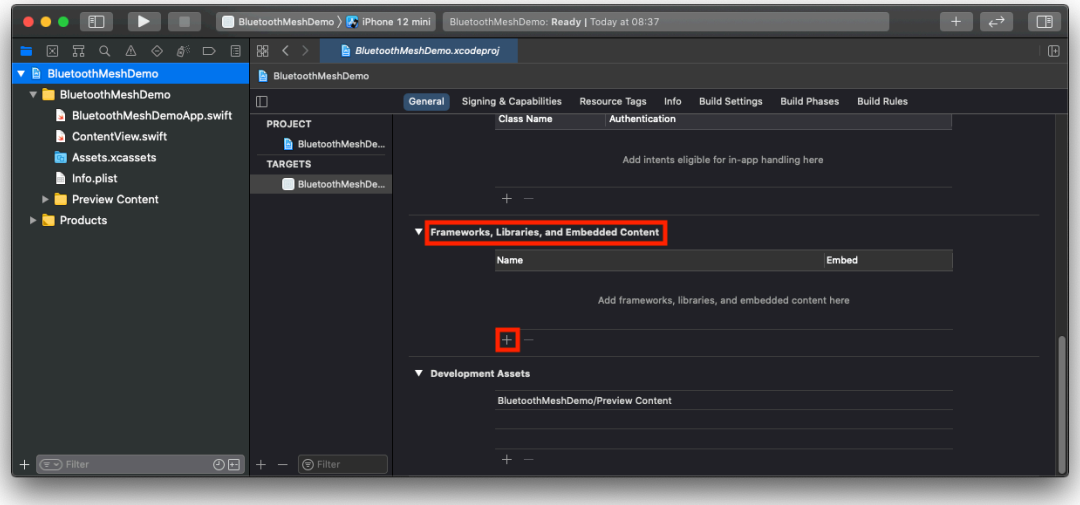
The delivery also contains API documentation for the Bluetooth mesh stack.

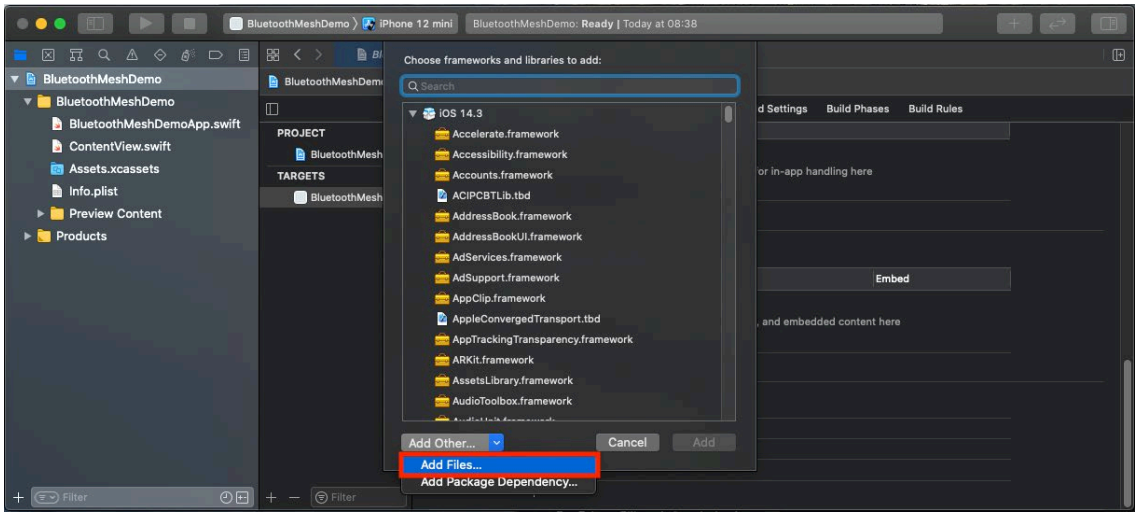
4 Getting Started with Development

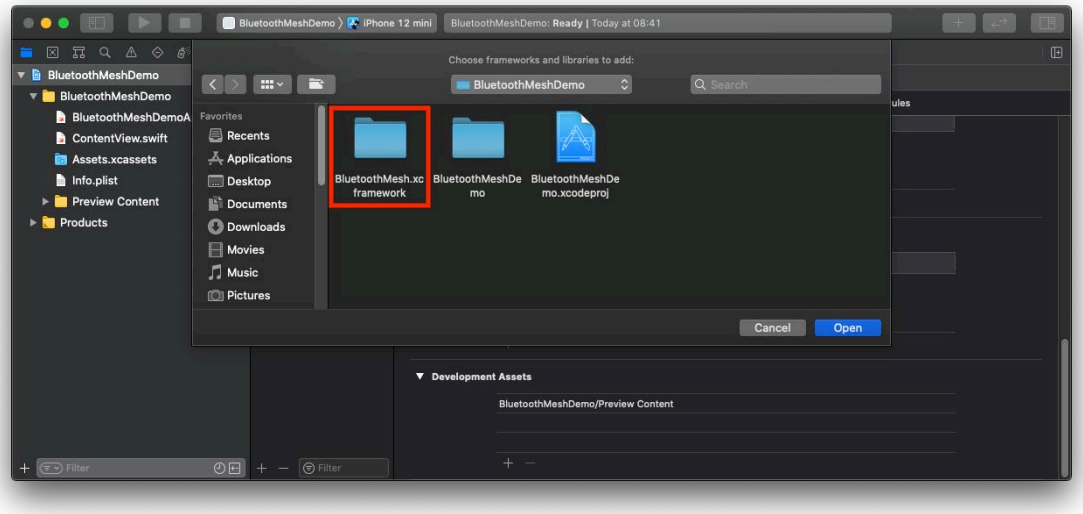
Once you have installed all the necessary tools listed in chapter 2, Prerequisites for Development, you can get started with developing a Bluetooth Mesh application based on the Silicon Labs' Mobile ADK. The first step is to set up a new project.

4.1 iOS

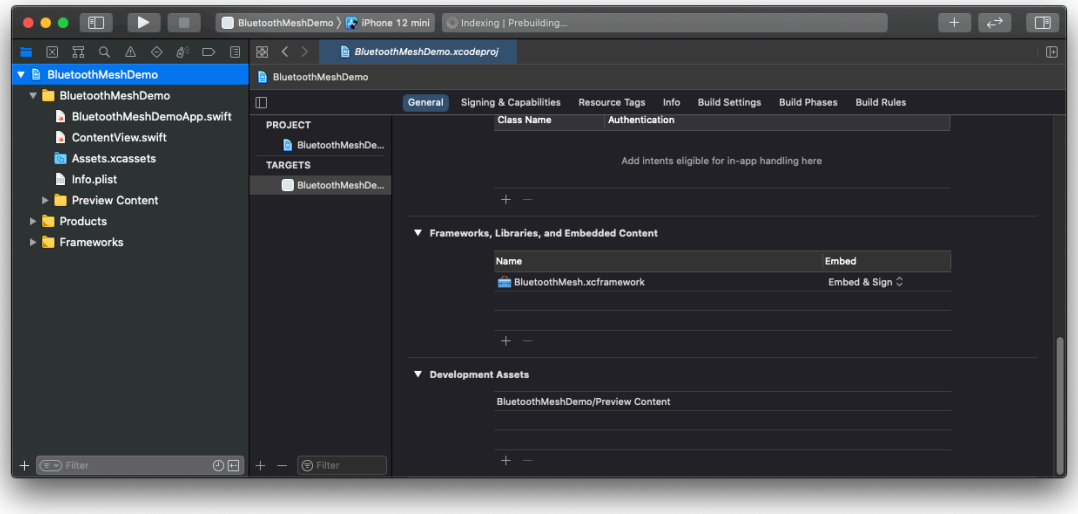
- Copy the BluetoothMesh.xcframework to a folder with the new iOS project. (The framework is located under the Simplicity Studio directory mentioned in section 2.1 iOS of this document)
- Open the main target in your project.
- Go to the General view.
- Add BluetoothMesh.xcframework to Frameworks, Libraries and Embedded Content.





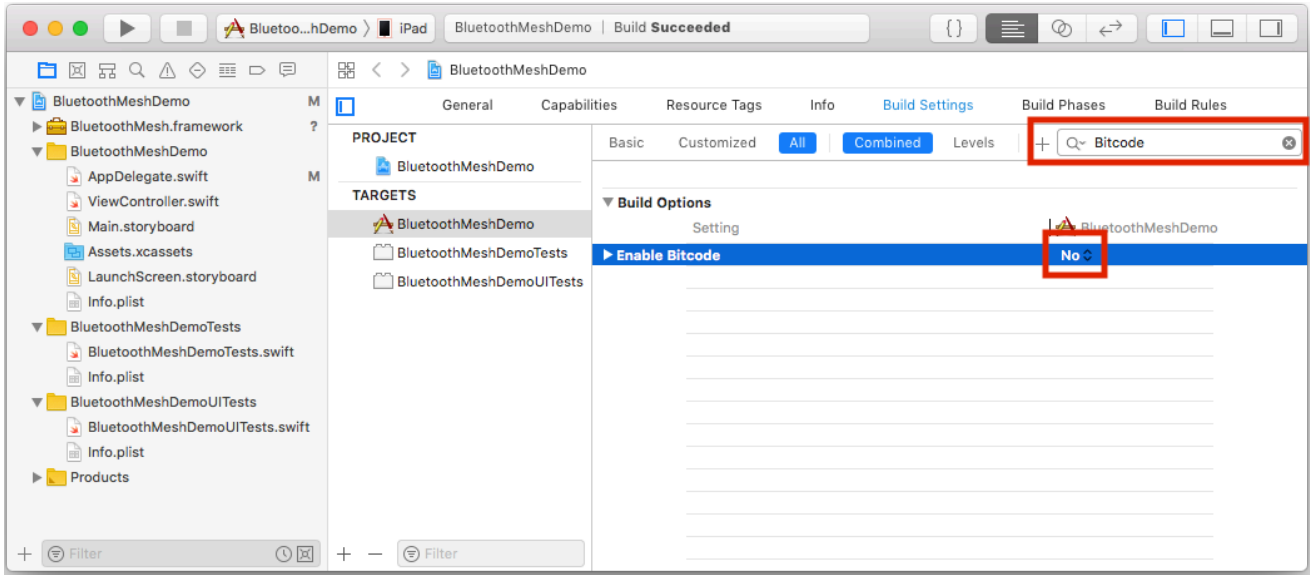


BluetoothMesh.xcframework should be visible in Frameworks, Libraries and Embedded Content section with “Embed & Sign” chosen.



Disable Bitcode in the project. BluetoothMesh.xcframework does not use Bitcode.

- Select the main target in the project.
- Go to the Build Settings view.
- Search for Bitcode.
- Set Enable Bitcode to 'No'.



4.2 Android

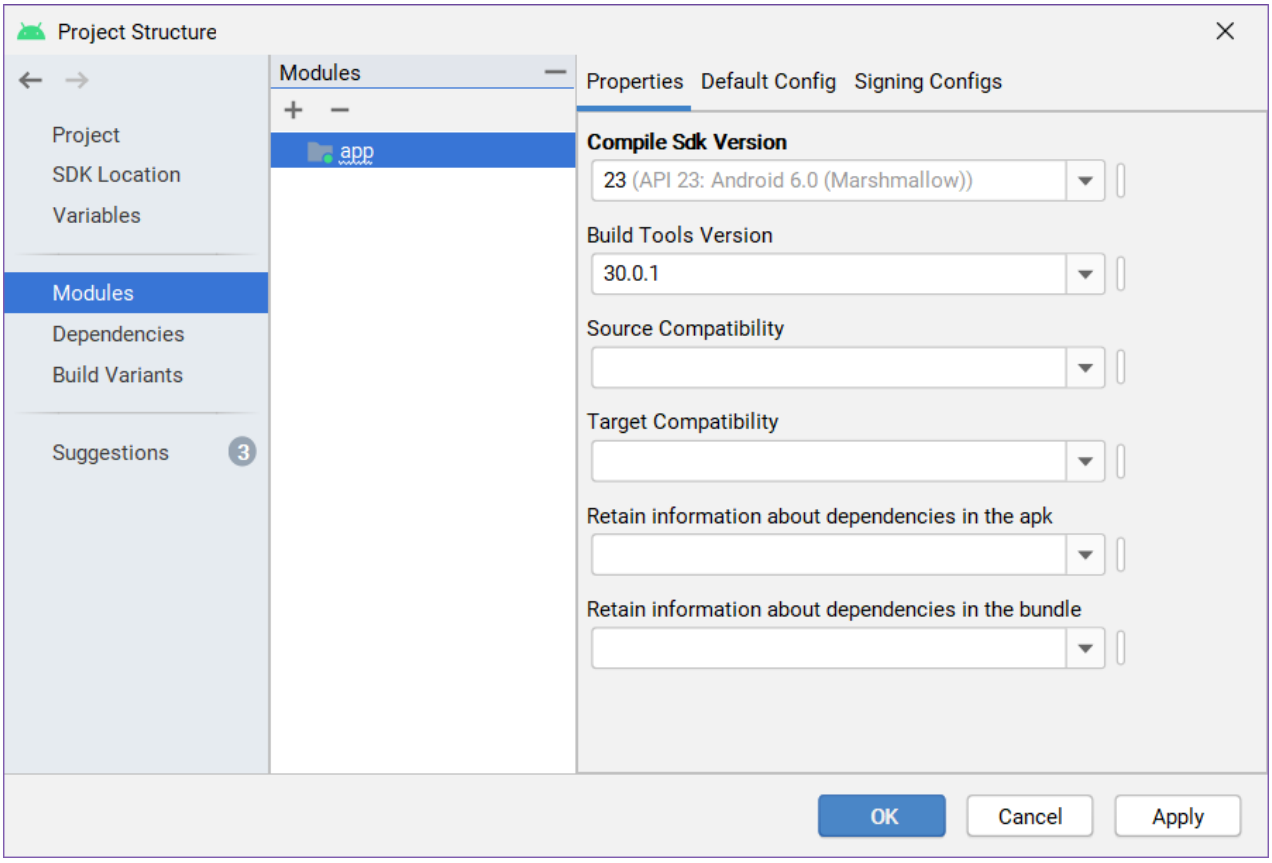
Open the Android Studio.

Create a new project.

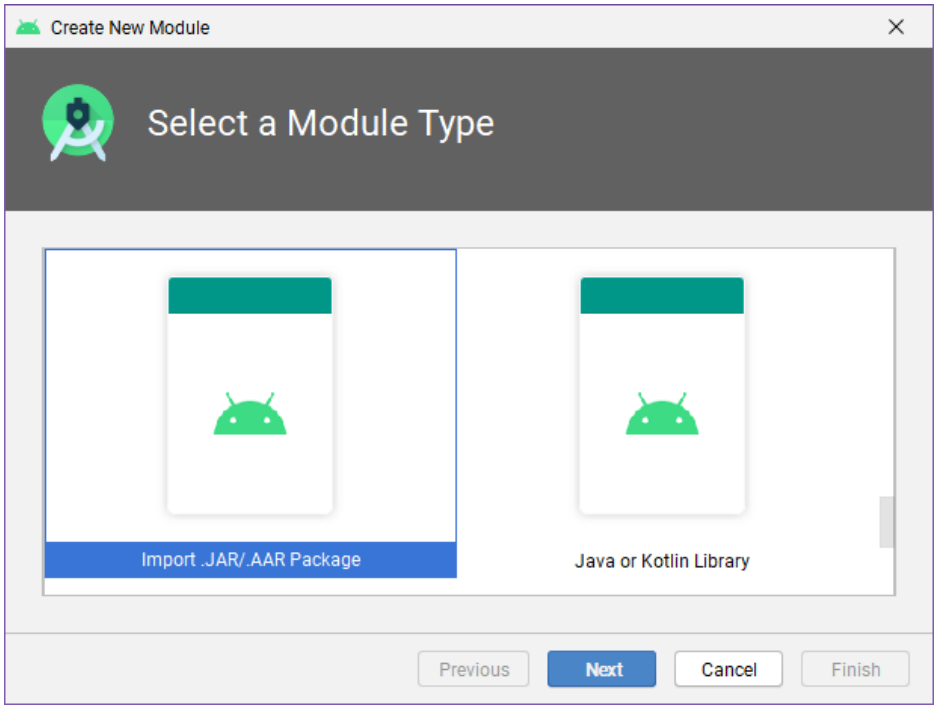
- Create the project for Phone and Tablet.
- Select at least API 23: Android 6.0 (Marshmallow).
- Create a project such as Empty Activity project.

In the "Project Structure" add new Modules by executing the following steps (for Android Studio 4.0) for both "ble_mesh-android_api-v2_high-release.aar" and "ble_mesh-android_api-v2_low-release.aar". (The files can be found under the Simplicity Studio directory mentioned in "Prerequisites for Development" section 2.2 Android.

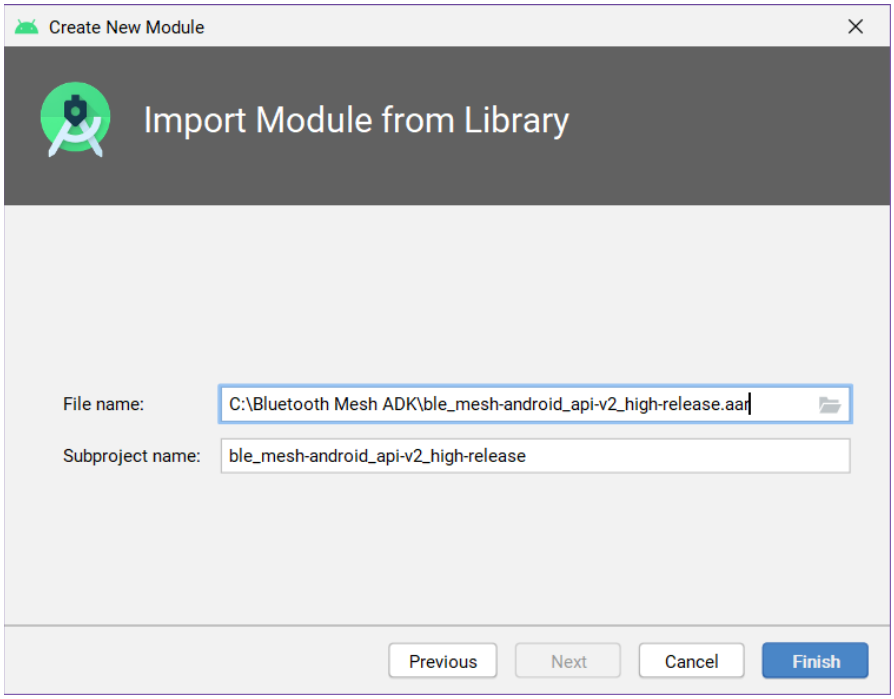
1. Open the “Modules” section and click “+” to add a new module.



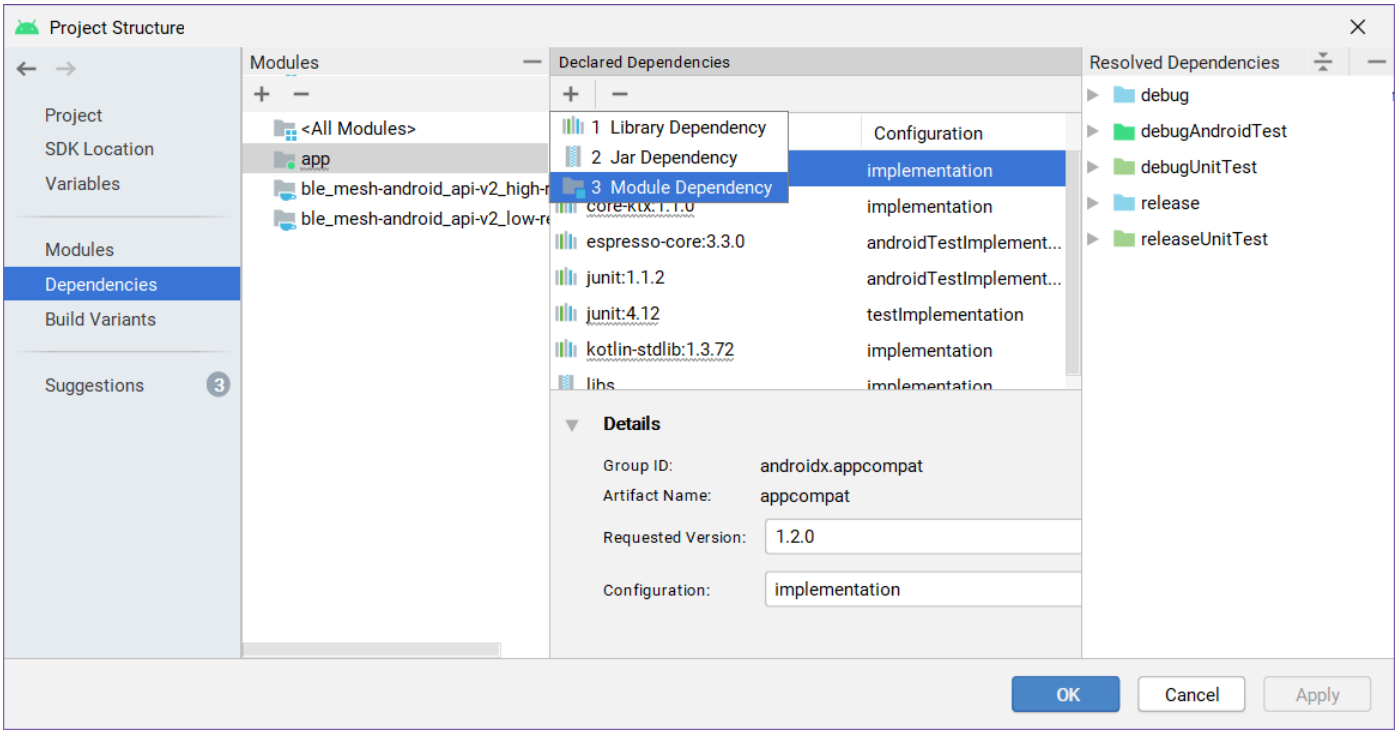
2. Select “Import .JAR or .AAR Package”.



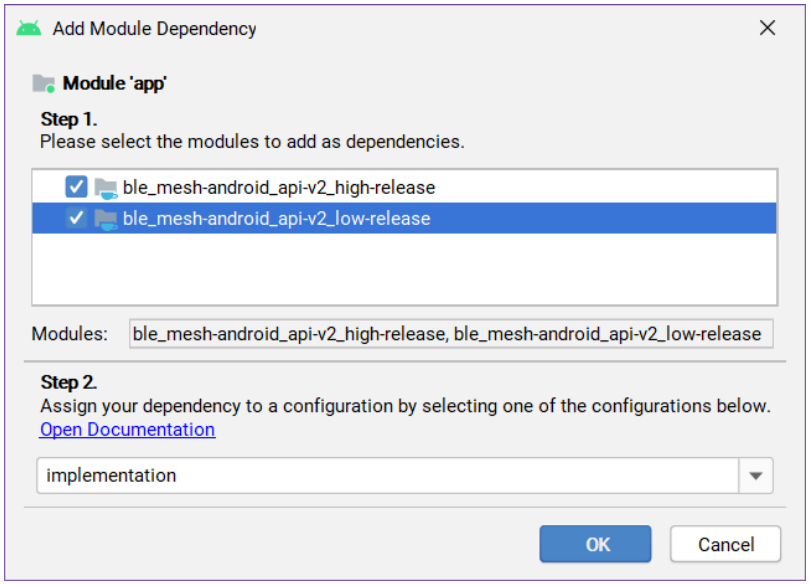
3. Specify the path to the .AAR file and proceed.



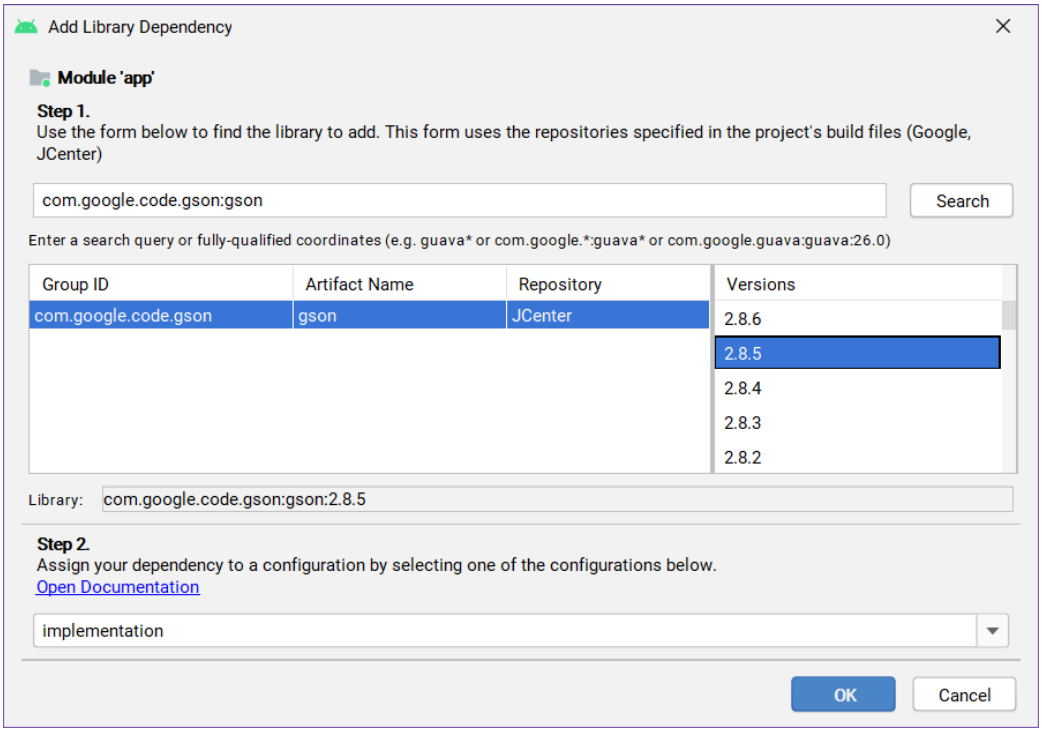
4. Repeat steps 1–3 to add a second .AAR module.
5. In the “Dependencies” section select the app module, click “+” in the “Declared Dependencies” section, and select “Module dependency”.



6. Select the modules and proceed.



7. Add Gson dependency to the project. Add a new dependency as in step 5, but select “Library Dependency”. Search for **com.google.code.gson:gson**, select version 2.x.x, and proceed.



8. Create a new BluetoothMesh object as shown in the following example.

```
import com.siliconlab.bluetoothmesh.adk.BluetoothMesh
import com.siliconlab.bluetoothmesh.adk.configuration.BluetoothMeshConfiguration

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        BluetoothMesh.initialize(applicationContext, BluetoothMeshConfiguration())
```

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

9. 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:

- Network – The main container in the mesh structure. Network is the owner of nodes and subnets. See Figure 5-2. Network Structure.
- Subnet – A specific subnet belongs only to one network. A subnet is the owner of groups.
- 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.
- Element – A part of a node.
- Model – A part of an element.
- 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.

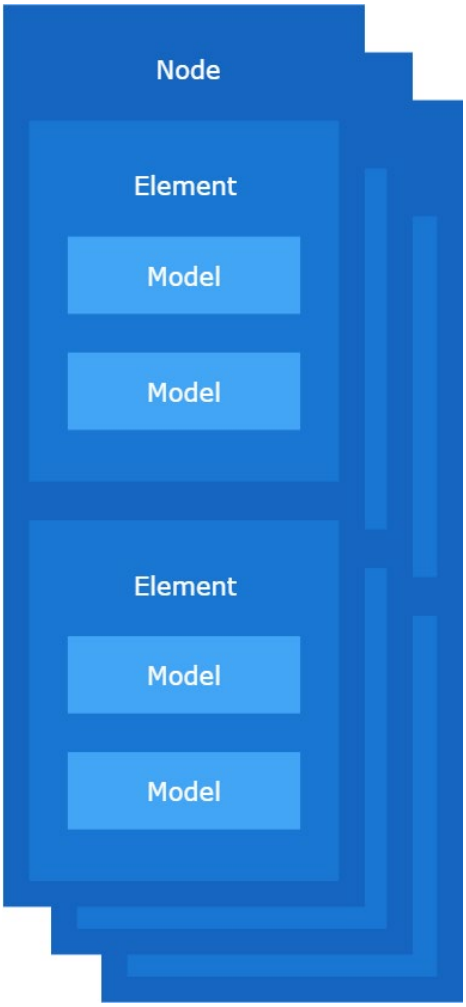


Figure 5-1. Node Structure

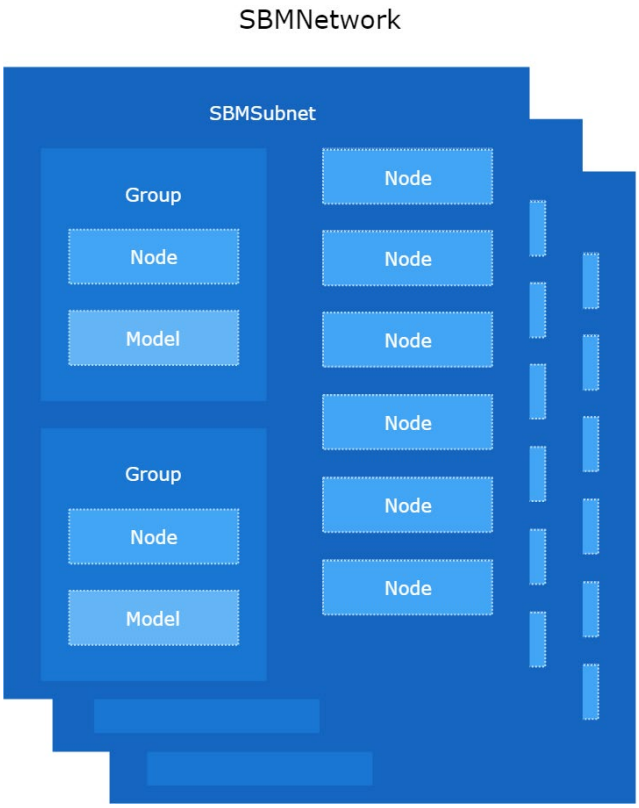


Figure 5-2. Network Structure

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: Using the ADK with a Simple Use Case

6.1 Example 1: GenericOnOff Get for element

Disclaimer: The example may not contain some necessary code or concepts, for example error handling or synchronization mechanisms, which are not specific to the Bluetooth mesh library. This code should not be used as is in a production environment.

6.1.1 Provisioning a Device

Bluetooth device discovery has to be done on the application side. The library expects the user to provide an object implementing:

iOS: SBMConnectableDevice protocol

Android: ConnectableDevice

This object represents the Bluetooth device and will be used to provision a new node. Devices that can be provisioned advertise themselves with the Bluetooth Mesh Provisioning Service.

To provision a new node, you need:

iOS: SBMConnectableDevice

Android: ConnectableDevice

Represents the device you want to provision that can be connected to before provisioning.

And:

iOS: SBMSubnet

Android: Subnet

Represents the target subnet to which to provision node.

The subnet can be created using:

iOS: SBMNetwork.createSubnet(withName: netKey:).

Android: Network.createSubnet

To initiate the provisioning process, create:

iOS: SBMProvisionerConnection

Android: ProvisionerConnection

The object using the subnet you want to provision the node to.

And:

iOS: SBMConnectableDevice

Android: ConnectableDevice

Represents the device.

Initiate the provisioning session using:

iOS: SBMProvisionerConnection.provision(withConfiguration:parameters:retryCount:callback:)

Android: ProvisionerConnection.provision

The configuration parameter should in this case be nil, since configuration will be done by hand.

6.1.2 Proxy Connection and Configuration

Once a node exists, you must establish a proxy connection to it and configure it.

A newly provisioned node allows an incoming proxy connection for some time after provisioning to enable configuration. This configuration may be also done after establishing a connection with the subnet to which a node was provisioned. It is important that, if done through a subnet, at least one node with has proxy enabled.

To establish a proxy connection, initialize:

iOS: SBMProxyConnection

Android: ProxyConnection

object using:

iOS: SBMConnectableDevice

Android: ConnectableDevice

object, which represents a Bluetooth device advertising with a Bluetooth Mesh Proxy Service. This device will be used to send messages to the network. After initializing this object:

iOS: SBMProxyConnection.connect(toProxy:errorCallback:)

Android: ProxyConnection.connectToProxy

must be invoked to actually establish the connection. There is a new overloaded *ProxyConnection.connectToProxy* method in Android with an additional boolean *refreshBluetoothDevice* argument, which supports deciding if *BluetoothDevice* should be refreshed before connecting to the proxy node. This is needed to obtain current GATT services after they have been changed during the provisioning process, because subsequent *discoverServices* method calls on the *BluetoothGatt* object can result in cached services from the first call (even if device was disconnected). Refreshing *BluetoothDevice* before connecting to the proxy node is required only during the first connection and configuration after provisioning. There is no need to use it later.

After establishing the connection:

iOS: SBMConfigurationControl

Android: ConfigurationControl

class can be used to set proxy service and node identity using:

iOS: SBMConfigurationControl.setProxy(_:with:errorCallback:)

Android: ConfigurationControl.setProxy

and

iOS: SBMConfiruationControl.setNodeIdentity(_:enabled:with:errorCallback:)

Android: ConfigurationControl.setNodeIdentity

Setting proxy is only required if the node is the only node within the network, since you cannot connect to network without a proxy node. Setting node identity enables associating:

iOS: SBMNode

Android: Node

object with:

iOS: SBMConnectableDevice,

Android: ConnectableDevice

object. Without an enabled node identity it is still possible to associate a Bluetooth device with a Subnet.

6.1.3 Binding Models

To bind models to AppKey a group must exist within the subnet to which the node belongs. A group can be created using:

iOS: SBMSubnet.createGroup(withName:appKey:address:)

Android: Subnet.createGroup

method. AppKey and the address argument can be nil, as they then will be autogenerated. After creating a group, bind the node to AppKey from the group using:

iOS: SBMNodeControl.bind(to:successCallback:errorCallback:)

Android:NodeControl.bind

method.

Models can be found within elements of the node.

iOS: SBMNode.elements

Android: Node.getElements

property contains elements of a node. Each element contains an array of sigModels and vendorModels. Since we will be sending Generic OnOff get message, we need to find:

iOS: SBMSigModel

Android: SigModel

in sigModels array, where the modelIdentifier is equal to:

iOS:SBMModelIdentifier.genericOnOffServer.

Android: ModelIdentifier.GenericOnOffServer

To bind this model to appKey use:

iOS: SBMFunctionalityBinder.bindModel(_:successCallback:errorCallback:)

Android: FunctionalityBinder.bindModel

method.

Initialize:

iOS: SBMFunctionalityBinder

Android: FunctionalityBinder

with the group containing the node with the model to which we will be binding.

6.1.4 Sending the Message

To send the message use:

iOS: SBMControlElement

Android: ControlElement

object. Initialize it with:

iOS: SBMElement and SBMGroup

Android: Element and Group

which contains the previously located generic on off server model.

To send get message use:

iOS: SBMControlElement.getStatus(_:successCallback:errorCallback:)

Android: ControlElement.getStatus

method.

Since we want to receive Generic OnOff status, the first argument needs to be:

iOS: SBMGenericOnOff.self.

Android: GenericOnOff

The response will be received in the success callback.

6.2 Example 2: GenericOnOff Get for group

Disclaimer: The example may not contain some necessary code or concepts, for example error handling or synchronization mechanisms, which are not specific to the Bluetooth mesh library. This code should not be used as is in a production environment.

6.2.1 Provisioning Devices

Bluetooth device discovery has to be done on the application side. The library expects the user to provide an object implementing:

iOS: SBMConnectableDevice protocol

Android: ConnectableDevice

This object represents the Bluetooth device and will be used to provision a new node. Devices that can be provisioned advertise themselves with the Bluetooth Mesh Provisioning Service.

To provision a new node, you need:

iOS: SBMConnectableDevice

Android: ConnectableDevice

Represents the device you want to provision that can be connected to before provisioning.

And:

iOS: SBMSubnet

Android: Subnet

Represents the target subnet to which to provision node.

Subnet can be created using:

iOS: SBMNetwork.createSubnet(withName: netKey:)

Android: Network.createSubnet

To initiate the provisioning process, create:

iOS: SBMProvisionerConnection

Android: ProvisionerConnection

The object using the subnet you want to provision the node to.

And:

iOS: SBMConnectableDevice

Android: ConnectableDevice

Represents the device.

Initiate the provisioning session using:

iOS: SBMProvisionerConnection.provision(withConfiguration:parameters:retryCount:callback:)

Android: ProvisionerConnection.provision

The configuration parameter should in this case be nil, since configuration will be done by hand.

The provisioning process needs to be performed for each device to be added to the group. Each device must be provisioned to the same subnet, so they can be added to same group afterwards. That is because the application key used by the group can only be bound to a single network key (used by the subnet).

6.2.2 Proxy Connection and Configuration

Once a node exists, you must establish a proxy connection to it and configure it.

A newly provisioned node allows an incoming proxy connection for some time after provisioning to enable configuration. This configuration may be also done after establishing a connection with the subnet to which a node was provisioned. It is important that, if done through a subnet, at least one node with has proxy enabled.

To establish a proxy connection, initialize:

iOS: SBMProxyConnection

Android: ProxyConnection

object using:

iOS: SBMConnectableDevice

Android: ConnectableDevice

object, which represents a Bluetooth device advertising with a Bluetooth Mesh Proxy Service. This device will be used to send messages to the network. After initializing this object:

iOS: SBMProxyConnection.connect(toProxy:errorCallback:)

Android: ProxyConnection.connectToProxy

must be invoked to actually establish the connection. There is new overloaded *ProxyConnection.connectToProxy* method in Android with an additional boolean *refreshBluetoothDevice* argument, which supports deciding if the *BluetoothDevice* should be refreshed before connecting to the proxy node. This is needed to obtain current GATT services after they have been changed during the provisioning process, because subsequent *discoverServices* method calls on a *BluetoothGatt* object can result in cached services from the first call (even if the device was disconnected). Refreshing the *BluetoothDevice* before connecting to a proxy node is required only during the first connection and configuration after provisioning. There is no need to use it later.

After establishing the connection:

iOS: SBMConfigurationControl

Android: ConfigurationControl

class can be used to set proxy service and node identity using:

iOS: SBMConfigurationControl.setProxy(_:with:errorCallback:)

Android: ConfigurationControl.setProxy

and

iOS: SBMConfigurationControl.setNodeIdentity(_:enabled:with:errorCallback:).

Android: ConfigurationControl.setNodeIdentity

At least one node in the network needs to act as a proxy node to enable sending messages to that network. Setting proxy is only required if the node is the only node within the network, since you cannot connect to network without a proxy node. Setting node identity enables associating:

iOS: SBMNode

Android: Node

object with:

iOS: SBMConnectableDevice,

Android: ConnectableDevice

object. Without an enabled node identity it is still possible to associate a Bluetooth device with a Subnet.

6.2.3 Binding models

To bind models to AppKey a group must exist within the subnet to which the node belongs. A group can be created using:

iOS: SBMSubnet.createGroup(withName:appKey:address:)

Android: Subnet.createGroup

method. AppKey and the address argument can be nil, as they then will be autogenerated. After creating a group, bind the node to AppKey from the group using:

iOS: SBMNodeControl.bind(to:successCallback:errorCallback:)

Android: NodeControl.bind

method.

Models can be found within elements of the node.

iOS: SBMNode.elements

Android: Node.getElements

property contains elements of a node. Each node contains an array of sigModels and vendorModels. Since we will be sending Generic OnOff get message, we need to find:

iOS: SBMSigModel

Android: SigModel

in sigModels array, where the modelIdentifier is equal to:

iOS: SBMModelIdentifier.genericOnOffServer.

Android: ModelIdentifier.GenericOnOffServer

To bind this model to appKey use:

iOS: SBMFunctionalityBinder.bindModel(_:successCallback:errorCallback:)

Android: FunctionalityBinder.bindModel

method.

Initialize:

iOS: SBMFunctionalityBinder

Android: FunctionalityBinder

with the group containing the node with the model to which we will be binding.

To perform a group request, add subscription settings to the model. Those settings tell the model to listen for messages sent to a group address, which will allow the model to receive and respond to the group request. This can be done by creating:

iOS: SBMSubscriptionSettings instance using init(group:) initializer.

Android: SubscriptionSettings

To add subscription settings create:

iOS: SBMSubscriptionControl

Android: SubscriptionControl

object for the Generic OnOff server model and use:

iOS: SBMSubscriptionControl.add(_:successCallback:errorCallback)

Android: SubscriptionControl.addSubscriptionSettings

method to add previously created subscription settings to the model.

The previous operations must be performed for each node in the group.

6.2.4 Sending the Message

To send the message use:

iOS: SBMControlGroup

Android: ControlGroup

object. Initialize it with:

iOS: SBMGroup

Android: Group

which contains nodes with Generic OnOff server models.

To send get message use:

iOS: SBMControlGroup.getStatus(_:successHandler:errorCallback:)

Android: ControlGroup.getStatus

method, which returns:

iOS: SBMTask

Android: MeshTask

object.

Since we want to receive Generic OnOff status, the first argument needs to be:

iOS: SBMGenericOnOff.self

Android: GenericOnOff

Responses will be received in the success handler for each element that responds to the message until we call:

iOS: SBMTask.cancel()

Android: MeshTask.cancel

method on the object returned from *getStatus* method.

7 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:

- Bluetooth connection
- Provision session
- Proxy connection
- Node configuration
- Model configuration (subscription and publication settings)
- 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.

7.1 Errors

In case of operation failure callbacks usually provide an NSError object with the appropriate domain and code. A list of all possible domains and codes can be found in SBMError.h header. Error codes have a short description of the situation in which the error can be encountered.

7.2 Initializing the BluetoothMesh

7.2.1 BluetoothMesh

The Bluetooth mesh structure is represented by a 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())
```

7.2.2 Set Up Supported Vendor Models

Supported vendor models can be set using *SBMBluetoothMeshConfiguration* 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(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], andLogger:
SBMLogger.sharedInstance())
SBMBluetoothMesh.sharedInstance().initialize(configuration)
```

7.2.3 Set Up Mesh Limits

Limits for the Bluetooth Mesh database can be set using *SBMBluetoothMeshConfiguration* 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 32766
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
32
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;
```

7.3 Request IV Index Update

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

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

An IV Index update can be requested by calling:

```
let ivIndexControl = SBMIvIndexControl()
try ivIndexControl.requestUpdate()
```

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

To receive information about the IV Update procedure status changes, subscribe to the event with:

```
ivIndexControl.setUpdateHandler() { 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
}
```

7.3.1 Text IV Update

Since the 96-hour limit for changing the IV Update procedure state might be a problem when testing the IV Update, there is a test mode to remove this limit. It can be enabled with:

```
try ivIndexControl.setUpdateTestModeEnabled(true)
```

7.3.2 IV Index Recovery

When a device has not been connected to the network for some time, it may have missed IV Index updates. The IV Index recovery procedure can be used to fix this issue. It is triggered with:

```
try ivIndexControl.setRecoveryModeEnabled(true)
```

The ADK will initiate recovery procedure if it detects that recovery is needed. Recovery will be performed after a new Secure Network Beacon is received (for example, after reconnecting with a proxy node).

7.4 Get Secure Network Beacon Information

Secure Network Beacon information can be obtained after connecting to the proxy service of any node from a known subnet. To observe this for a secure network beacon, call method *observeForSecureNetworkBeacon* before connecting to the proxy. Assume that you have any object that conforms to *ConnectableDevice*. For example:

```
let connectableDevice: SBMConnectableDevice!
let proxyConnection = SBMProxyConnection(connectableDevice: connectableDevice)
proxyConnection.observe(forSecureNetworkBeacon: { (netKeyIndex, keyRefresh, ivUpdate, ivIndex) in
    // Handle update here
})
proxyConnection.connect(toProxy: { (device) in
    // Handle success
}) { (device, error) in
    // Handle failure
}
```

For more information about Secure Network Beacon see section 3.9.3 Secure Network Beacon from the *Mesh Profile Bluetooth® Specification*.

7.5 Server Configuration

7.5.1 Time to Live

The TTL value is the maximum number of hops the packet takes along the path to its destination. The default value is 5.

Please note that the value 0 can only be used when communicating directly with a proxy node, otherwise the message will be lost.

Lowering this value may improve network performance during the node configuration process, since the number of messages in the network will be decreased. If such a change has been made, it is important to restore TTL to the higher value after configuration is done to ensure proper network operation.

```
let serverConfigurationControl = SBMServerConfigurationControl()
try serverConfigurationControl.setTTL(ttl)
```

7.6 Set Up Bluetooth Layer (SBMConnectableDevice)

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.

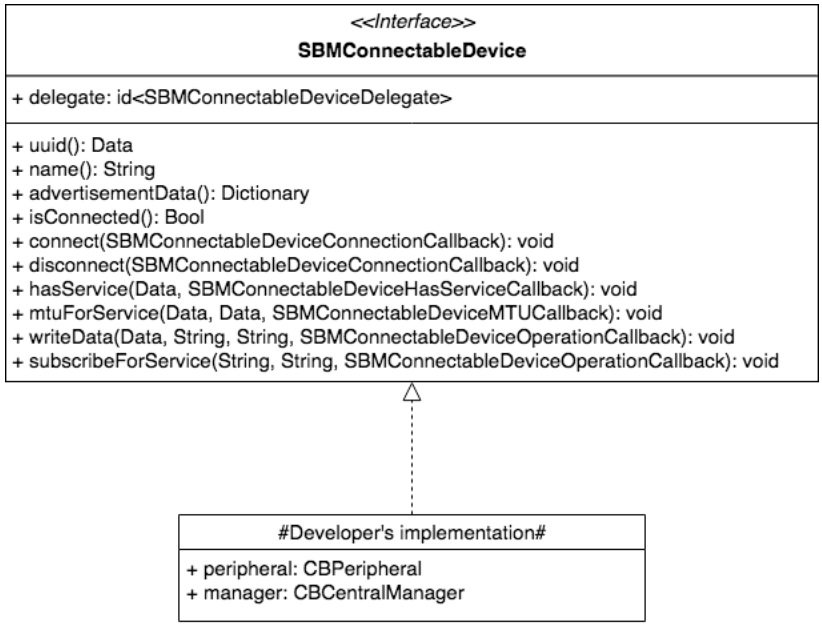


Figure 7-1. SBMConnectableDevice

7.6.1 Device Advertisement Data

Advertisement data can be obtained from `CBCentralManagerDelegate`.

```
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
}
```

7.6.2 Device UUID

The device UUID comes from advertisement data.

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

```
func uuid() -> Data? {
guard let serviceData = advertisementData [CBAdvertisementDataServiceDataKey] as? NSDictionary,
let data = serviceData[CBUUID(string: "1827")] as? Data else {
           return nil
```

```
        }
        return data.subdata(in: 0..<16)
}
```

7.6.3 Device Name

The device name comes from `CBPeripheral`.

```
let peripheral: CBPeripheral

// (…)

// SBMConnectableDevice
func name() -> String {
        return peripheral.name
}
```

7.6.4 Device Connection State

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

```
let peripheral: CBPeripheral

// (…)

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

7.6.5 Connect to the Device

```
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)
}
```

7.6.6 Disconnect from the Device

```
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)
}
```

7.6.7 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.

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

7.6.8 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.

```
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)
}
```

7.6.9 Write Data to a Given Service and Characteristic

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

```
let peripheral: CBPeripheral

// (…)

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

        If cbCharacteristic != nil {
              peripheral.writeValue(data,
                                    for: cbCharacteristic!,
                                    type: .withoutResponse)
                                    // It is very important to write data without response
              completion(self,
                             cbCharacteristic!.service.uuid.data,
                             cbCharacteristic!.uuid.data,
                             true)
       } else {
              completion(self,
                             CBUUID(string: service).data,
                             CBUUID(string: characteristic).data,
                             false)
       }
}
```

7.6.10 Subscribe to a Given Service and Characteristic

```
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 cbCharacteristic = peripheral.services?.first {
                                    $0.uuid.uuidString == service
                             }?.characteristics?.first {
                                    $0.uuid.uuidString ==  characteristic
                             }

       if cbCharacteristic != nil {
              callback = completion
              peripheral.setNotifyValue(true, for: cbCharacteristic!)
       } 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)
}
```

7.6.11 Unsubscribe from a Given Service and Characteristic

This method is mandatory only when Provisioning with Configuration using one GATT connection. See section 7.7.4.3.1 One GATT Connection for Provisioning and Proxy Session.

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

// (…)

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

	if cbCharacteristic != nil {
		callback = completion
		peripheral.setNotifyValue(false, for: cbCharacteristic!)
	} 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)
}
```

7.7 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.

7.7.1 Create Network

You can create a network by specifying its name

```
let network = try? SBMBluetoothMesh.sharedInstance().createNetwork(with: "Network name")
```

After creating a new network initialize it with address and IV Index. Address 0 means that default value (0x2001) will be assigned.

```
SBMBluetoothMesh.sharedInstance().initializeNetwork(network, address: address, ivIndex: ivIndex)
```

Currently it is possible to create more than one Network instance, but it is not recommended because of the known issue described in section 13.1 Concept.

7.7.2 Create Subnet

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

```
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.

```
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!)
```

7.7.3 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, use a class that implements ConnectableDevice.

```
let centralManager: CBCentralManager

// (…)

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

startDiscovering(services: [CBUUID(string:
SBMProvisionerConnection.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
}
```

7.7.4 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. You can connect to a device before provisioning, for example to identify the provisioning target.

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

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

7.7.4.1 In-Band Provisioning

```
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
}
```

7.7.4.2 OOB Provisioning

```
let connectableDevice: SBMConnectableDevice
let subnet: SBMSubnet
let provisionerOOB: ProvisionerOOB // ProvisionerOOB is an implementation of SBMProvisionerOOB
protocol which has 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
}
```

7.7.4.2.1 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.

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

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

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

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

    func allowedInputActions() -> SBMProvisionerOOBAllowedInputActions {
        return SBMProvisionerOOBAllowedInputActions(rawValue:
SBMProvisionerOOBAllowedInputActions.alpha.rawValue |
SBMProvisionerOOBAllowedInputActions.numeric.rawValue |
SBMProvisionerOOBAllowedInputActions.push.rawValue |
SBMProvisionerOOBAllowedInputActions.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 oobNumber: NSNumber
       var oobString: String

       //(…)

       // Developer has to provide implementation to get needed oobData.
       // Depending on outputAction it can be string or number.

       //(…)

       if outputAction == .alpha {
           self.delegate?.provideAlphanumericAuthData(oobString, for: uuid)
       } else {
           self.delegate?.provideNumericAuthData(oobNumber, for: uuid)
}
        return .success //if success
    }

    func authRequest(_ uuid: UUID) -> SBMProvisionerOOBResult {
       var oobData: Data
```

```
        //(…)

        // Developer has to provide implementation to get needed oobData.

        //(…)

          self.delegate?.provideAuthData(oobData!, for: uuid)
          return .success //if success
    }

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

        // Developer has to provide implementation to display authData.

        //(…)

          return .success //if success
    }

    func numericInputDisplay(_ uuid: UUID, authNumber: NSNumber) -> SBMProvisionerOOBResult {
        //(…)

        // Developer has to provide implementation to display authNumber.

        //(…)

        return .success //if success
    }

    func alphanumericInputDisplay(_ uuid: UUID, authString: String) -> SBMProvisionerOOBResult {
        //(…)

        // Developer has to provide implementation to display authString.

        //(…)

        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() -> SBMProvisionerOOBAllowedAuthMethods
```

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. It is deprecated and either `numericInputDisplay:authNumber` or `alphanumericInputDisplay:authString` method should be used instead. This method will not be called if any of the above mentioned method is implemented.

```
func alphanumericInputDisplay(_ uuid: UUID, authString: String) -> SBMProvisionerOOBResult
```

Optional method called when platform requests an out-of-band input authentication data of alphanumeric type. Output authentication data will have alphanumeric type when selected Output Action is Alpha.

```
func numericInputDisplay(_ uuid: UUID, authNumber: NSNumber) -> SBMProvisionerOOBResult
```

Optional method called when platform requests an out-of-band input authentication data of numeric type. Authentication data will have numeric type when selected Input Action is one of Numeric, Push, Twist.

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. In cases when:

- output authentication method is used
- static authentication method is used and the authentication data is a number or an alphanumeric string

provideNumericAuthData(...) or provideAlphanumericAuthData(...) methods can be used to provide authentication data without conversion to bytes.

```
func provideNumericAuthData(…)
```

Convenience method that provides out-of-band numeric 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. Output authentication data will have numeric type when selected Output Action is one of Numeric, Blink, Beep, Vibrate.

```
func provideAlphanumericAuthData(…)
```

Convenience method that provides out-of-band alphanumeric 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. Output authentication data will have alphanumeric type when selected Output Action is Alpha.

7.7.4.3 Provisioning with Proxy Session

A node can be configured immediately after it has been provisioned. The behavior depends on the first argument of SBMProvisionerConnection.provision(..) method. It is expected to be an object of type SBMProvisionerConfiguration, which determines what parts of the configuration process should be executed.

Null may be passed as an argument, meaning that a device should only be provisioned. Otherwise a Proxy session with node configuration will be performed as well. If all the configuration tasks are chosen, the following operations will occur in the order given:

1. Connecting to proxy
2. Getting device composition data
3. Setting up proxy
4. Setting up node identity
5. Disconnecting from proxy

Tasks 1 and 2 are performed by default when SBMProvisionerConfiguration is not null while tasks 3-5 can be added to the process by the appropriate SBMProvisionerConfiguration init method. The following example shows how to provision a node and configure it within one user operation.

```
let connectableDevice: SBMConnectableDevice
let subnet: SBMSubnet
...
let provisionerConnection = SBMProvisionerConnection(for: connectableDevice, subnet: subnet)

let provisionerConfiguration = SBMProvisionerConfiguration(proxyEnabled: true, nodeIdentityEnabled:
true)

...
provisionerConnection.provision(withConfiguration: provisionerConfiguration, parameters: nil,
retryCount: 3) { (connection, node, error) in
       //Handle result of provisioning
       }
```

7.7.4.3.1 One GATT Connection for Provisioning and Proxy Session

After the device is successfully provisioned, the Bluetooth GATT connection is closed and reopened to start the configuration process during the Proxy session. The same Bluetooth GATT connection can be used for this Proxy session. To use this feature toggle on the useOneGattConnection field in the SBMProvisionerConfiguration object.

A few changes are needed to use this feature. The SBMConnectableDevice implementation should implement following behaviour:

1. Implement SBMConnectableDevice.unsubscribe(fromService:, characteristic:, completion:) method. See section 7.6.11 Unsubscribe from a Given Service and Characteristic.
2. Call SBMConnectableDeviceDelegate whenever the connectable device's services are invalidated:

```
//SBMConnectableDevice property
var delegate: SBMConnectableDeviceDelegate?

//CBPeripheralDelegate
func peripheral(_ peripheral: CBPeripheral, didModifyServices invalidatedServices: [CBService]) {
        delegate?.didModifyServices(invalidatedServices.map(\.uuid.uuidString), for: self)
}
```

7.8 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(node: node)

// (…)

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

7.9 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(node: node)

//(…)

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

7.10 Removing Subnet

To remove a subnet use the *SBMSubnet.remove(callback:, errorCallback:)* method. This method sends a factory reset message to every node in the subnet that is only in this subnet, and with the currently connected proxy node being the last one to which the message is sent. However, this method is very simplistic and does not allow dealing with some problems that may occur during subnet removal. Since some messages can be lost, the subnet may end up in a state where some nodes were not reset and can no longer be reached, or all nodes have successfully reset but the message confirming it was not received.

If more reliability is required, it is better to deal with each node individually. If a node is only in the subnet that is to be removed, a factory reset message can be sent to remove it from subnet. If the node is also in another subnet, it can be removed from subnet that is to be deleted by using the *SBMNodeControl.remove(from:, successCallback:, errorCallback:)* method.

After removing all nodes, calling *SBMSubnet.remove(callback:, errorCallback:)* will only remove the subnet from database, since it is empty already.

7.11 Factory reset node

To factory reset a node, use the *SBMConfigurationControl.factoryReset(callback:, errorCallback:)* method. A success callback should be received on successful factory reset, but it is not 100% reliable. If the factory reset is done on the proxy node through which the provisioner is connected, then no success callback will be called. In this case, the connection will be dropped and errorCallback will be called with a timeout error.

If the message has been successfully sent, it still may be lost in network. Also, the response message from the node may be lost or the node may not send it at all. In such cases *errorCallback* will be called with a timeout error.

When *errorCallback* is called, we do not know if the node has successfully performed the factory reset. It is possible that the request was lost and not delivered to the node. To confirm, send any acknowledged message that the node would normally handle (apart from the factory reset message). If the node has performed the factory reset, the message will result in a timeout error. Otherwise the node will respond. If the factory reset has not been performed, simply send the factory reset message again and verify again. It is also possible that the request was delivered successfully, but the node's response was lost. It can be verified by scanning to see if the reset device advertises as unprovisioned. If the node has performed the factory reset, use the *SBMNode.removeOnlyFromLocalStructure()* method to remove the node from the local database. It is also possible that the device neither responds to requests, nor advertises as unprovisioned, for example when resetting a low power node. In this case, sending the factory reset request multiple times can improve reliability, but does not guarantee success.

It is also possible that the node has performed a factory reset due to other reasons, such as someone manually performing it on the device, without using Mesh messages. In that case, the node will not respond to any messages, which will then with timeout errors. To remove such a node from the local database, use *SBMNode.removeOnlyFromLocalStructure()*.

7.12 Configure Node Default TTL

To get the node default TTL value, use the *SBMConfigurationControl.checkDefaultTTL(_ successCallback:, errorCallback:)* method. A success callback should be received on successfully getting node default TTL value. Otherwise errorCallback will be called.

To set the node default TTL value, use the *SBMConfigurationControl.setDefaultTTL(_ defaultTTL:, with successCallback:, errorCallback:)* method. A success callback should be received on successfully setting node default TTL value. Otherwise errorCallback will be called.

7.13 Connect with a Subnet

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

7.13.1 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, use a class that extends ConnectableDevice

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

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

//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
}
```

7.13.2 Get Node Representing a Given Device

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

```
let connectableDevice: SBMConnectableDevice

//(…)

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

7.13.3 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 *SBMConnectableDevice-Helper*.

```
let subnet: SBMSubnet
let connectableDevice: SBMConnectableDevice

// (…)

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

7.14 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)
```

7.15 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
})
```

7.16 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(node: node)
control.bind(to: group, successCallback: {
       //handle success callback
}, errorCallback: {
       //handle error callback
})
```

7.17 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
})
```

7.18 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.

```
let model: SBMModel
let group: SBMGroup

// (…)

let binder = SBMFunctionalityBinder(group: group)

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

7.19 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.

```
let model: SBMModel
let group: SBMGroup

// (…)

let binder = SBMFunctionalityBinder(group: group)

binder.unbindModel(sd, successCallback: { unboundModel, unboundGroup in
       //handle success callback
}, errorCallback: { modelToUnbind, groupToUnbind, error in
       //handle error callback
})
```

7.20 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.

7.20.1 SIG Model

```
let sigModel: SBMSigModel
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
})
```

7.20.2 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
})
```

7.21 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.

7.21.1 SIG Model

7.21.1.1 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
})
```

7.21.1.2 Publish Directly to the Provisioner

```
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
})
```

7.21.2 Vendor Model

7.21.2.1 Publish via SBMGroup Address

```
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
})
```

7.21.2.2 Publish Directly to the Provisioner

```
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
})
```

7.22 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.

7.22.1 Get Value for a Single SIG Model from the Node

```
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
        })
}
```

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

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

```
func getLevel(forGroup group: SBMGroup) {
        let controlGroup = SBMControlGroup(group: group)

        controlGroup.getStatus(SBMGenericLevel.self, successCallback: {
                (controlGroup, response, element) in
                let response = response as! SBMGenericLevel
                //handle success callback
        }, errorCallback: { (control, response, error) in
                //handle error callback
        })
}
```

7.22.3 Set Value for a Single SIG Model from the Node

Note: A previous method for this function has been deprecated. Do NOT use the deprecated method when the new one is also used. The library is not adapted to use these two constructors alternately during the lifetime of an application.

Transaction time, delay time and transaction ID are not available for each SET request. Refer to the following table to see which type of *SBMControlRequestParamters* should be used for a specific SET message.

init(transitionTime: UInt16, delayTime: UInt16, requestReply: Bool, transactionId: UInt8)	init(requestReply: Bool)
Generic OnOff, Generic Level, Generic Delta, Generic Move, Generic Power Level, Light Lightness, Light Lightness Linear, Light CTL, Light CTL Temperature	Generic Default Transition Time, Generic OnPowerUp, Generic Power Default, Generic Power Range, Generic Location Global, Generic Location Local, Generic User Property, Generic Admin Property, Generic Manufacturer Property, Light Lightness Default, Light Lightness Range, Light CTL Temperature Range, Light CTL Default

TRANSITION_TIME - Transition time in milliseconds, or zero for an immediate change request.

DELAY_TIME - Delay in milliseconds before the server acts on the request, or zero for immediate action.

REQUEST_REPLY - A boolean value that determines whether an explicit response (status message) is required.

TRANSACTION_ID - A transaction identifier indicating whether the message is a new message or a retransmission of a previously sent message. The Bluetooth Mesh library user is responsible for properly handling the transaction id.

To retransmit the message, use the same value for the transactionId field as in the previously sent message, within 6 seconds from sending that message.

```
var parameters: SBMControlRequestParameters
parameters = SBMControlRequestParameters(transitionTime: TRANSITION_TIME,
                                         delayTime: DELAY_TIME,
                                      requestReply: REQUEST_REPLY,
                                     transactionId: TRANSACTION_ID)
```

OR

```
parameters = SBMControlRequestParameters(requestReply: REQUEST_REPLY)
```

```
(…)

func set(level: Int, forElement element: SBMElement, inGroup group: SBMGroup, with parameters:
SBMControlRequestParameters) {
       let controlElement = SBMControlElement(element: element, in: group)
       let status = SBMGenericLevel(level: Int16(level))
       controlElement.setStatus(status,
                                parameters: parameters,
                                successCallback: { control, response in
              //handle success callback
       }, errorCallback: { control, response, error in
              //handle error callback
       })
}
```

7.22.4 Set Value for All Specific SIG Models Bound with the Group

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

Note: A previous method for this function has been deprecated. Do NOT use the deprecated method when the new one is also used. The library is not adapted to use these two constructors alternately during the lifetime of an application.

Note: See section 7.22.3 Set Value for a Single SIG Model from the Node for a description of SBMControlRequestParameters.

```
var parameters: SBMControlRequestParameters
parameters = SBMControlRequestParameters(transitionTime: TRANSITION_TIME,
                                                delayTime: DELAY_TIME,
                                             requestReply: REQUEST_REPLY,
                                            transactionId: TRANSACTION_ID)

OR

parameters = SBMControlRequestParameters(requestReply: REQUEST_REPLY)

(…)

func set(lightness: Int, for group: SBMGroup, with parameters: SBMControlRequestParameters) {
       let controlGroup = SBMControlGroup(group: group)
       let status = SBMLightningLightnessActual(lightness: UInt16(lightness))
       controlGroup.setStatus(status,
                                 parameters: parameters,
                                 successHandler: { controlGroup, result, element in
              //handle success callback
       }, errorCallback: { controlGroup, request, error in
              //handle error callback
       })
}
```

7.22.5 Control Sensor Models

Despite the fact that the SIG models already have logic for set/get states (see the previous 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 7.20.1 SIG Model and 7.21.1 SIG Model, respectively).

To set up Sensor model bindings use the API prepared for SIG models (see sections 7.18 Bind a Model with a Group and 7.19 Unbind a Model from a Group).

7.22.5.1 Get Sensor Model Values

7.22.5.1.1 Get Sensor Descriptors from the Node

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
		let response = response as! SBMSensorDescriptors
		//handle success callback
	}, errorCallback: { (control, request, 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:

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

7.22.5.1.2 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*).

```
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, request, error) in
		//handle error callback
	})
}
```

7.22.5.1.3 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*).

```
func getCadenceState(for element: SBMElement, in group: SBMGroup, propertyID: Int) {
	let controlElement = SBMControlElement(element: element, in: group)

	let properties = SBMSensorPropertiesGet.cadenceStatus(propertyID)
	controlElement.getSensorStatus(SBMSensorCadenceStatus.self, properties: properties,
	successCallback: {control, response in
		let response = response as! SBMSensorCadenceStatus
		//handle success callback
	}, errorCallback: {control, request, error in
		//handle error callback
	})
}
```

7.22.5.1.4 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*).

```
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, request, error in
		//handle error callback
	})
}
```

7.22.5.1.5 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.

```
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, request, error in
		//handle error callback
	})

}
```

7.22.5.1.6 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.

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

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

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:

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

7.22.5.1.7 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()`.

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, request, error) in
		//handle error
	})
}
```

7.22.5.1.8 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()`.

element – The response will be received from this SBMElement.

```
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, request, error) in
		//handle error
     })
}
```

7.22.5.1.9 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 getSettingState(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, request, error) in
		//handle error
	})
}
```

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

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

7.22.5.2 Set Sensor Server Setup Model Values

7.22.5.2.1 Set Sensor Cadence within the Node

SBMSensorCadenceSet message contains 9 parameters:

- SensorMessageFlags – determines whether the sent message must be acknowledged.
- SensorPropertyID – determines the ID of the 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 percent-age 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, 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, request, error in
		//handle error
	})
}
```

7.22.5.2.2 Set Sensor Setting within the Node

SBMSensorSettingSet message contains 4 parameters:

- SensorMessageFlags – determines whether the message sent must be acknowledged.
- SensorPropertyID – determines the ID of the sensor to receive the message.
- SettingPropertyID – determines the ID of the sensor setting to be set, including the size, format, and representation of the SettingRaw parameter.
- SettingRaw – raw data representing the sensor setting 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*).

```
func setSettingState(for element: SBMElement, in group: SBMGroup, settingSet: SBMSensorSettingSet)
{
	let controlElement = SBMControlElement(element: element, in: group)

	controlElement.setSensorSetupStatus(settingSet, successCallback: { control, response in
		let settingStatus = response as! SBMSensorSettingStatus
		//handle success
	}, errorCallback: { control, request, error in
		//handle error
	})
}
```

7.22.5.2.3 Set Sensor Cadence within All Nodes in the Group

SBMSensorCadenceSet message contains 9 parameters:

- SensorMessageFlags – determines whether the sent message must be acknowledged.
- SensorPropertyID – determines the ID of the 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*).

```
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
		//handle error
	})
}
```

7.22.5.2.4 Set Sensor Setting within All Nodes in the Group

SBMSensorSettingSet message contains 4 parameters:

- SensorMessageFlags – determines whether the sent message must be acknowledged.
- SensorPropertyID – determines the ID of the sensor to receive the message.
- SettingPropertyID – determines the ID of the sensor 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*).

```
func setSettingState(for group: SBMGroup, settingSet: SBMSensorSettingSet) {
       let controlGroup = SBMControlGroup(group: group)

       let task = controlGroup.setSensorSetupStatus(settingSet, successCallback: { control,
response, element in
              let settingStatus = response as! SBMSensorSettingStatus
              //handle success
       }, errorCallback: { control, request, error in
              //handle error
       })
}
```

7.22.6 Subscribe to Publications sent by SIG Model

To capture notifications from the SIG Model publication settings must be configured on the model side (see section 7.21 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.

Beginning with API 2.6.1, use the SBMControlElementPublications.subcribe(toStatus:successHandler:errorHandler:) method.

```
func subscribeLightness(forElement element: SBMElement, inGroup group: SBMGroup) {
    let controlElementPublications = SBMControlElementPublications(element: element, in: group)
    controlElementPublications.subscribe(toStatus: SBMLightningLightnessActual.self,
successHandler: { (element, group, result) in
        // handle notifications
    }, errorHandler: { (element, group, error) in
        // handle error
    })
}
```

7.22.7 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 7.2 Initializing the BluetoothMesh).

7.22.7.1 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)
```

7.22.7.2 Create SBMLocalVendorRegistrator

Used to set registration settings of a local vendor model.

```
let localVendorModel: SBMLocalVendorModel

//(…)

let vendorRegistrator = SBMLocalVendorRegistrator(model: localVendorModel)
```

7.22.7.2.1 Register a Local Vendor Model

Note: Typically, registration should be done one time for the local vendor model after initializing SBMBluetoothMesh (see section 7.2 Initializing the BluetoothMesh).

```
let vendorRegistrator: SBMLocalVendorRegistrator
let registerSettings: SBMLocalVendorSettings

//(…)

vendorRegistrator.register(registerSettings)
```

7.22.7.2.2 Unregister a Local Vendor Model

```
let vendorRegistrator: SBMLocalVendorRegistrator

//(…)

vendorRegistrator.unregister()
```

7.22.8 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 7.18 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.

7.22.8.1 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.

```
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

//(…)

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

7.22.8.2 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.

```
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

//(…)

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

7.22.9 Manage Subscriptions to the SBMVendorModel

The SBMVendorModel can be configured to publish status by SBMGroup address or directly by SBMNode address by adding publication settings to the model (see section 7.21 Add Publication Settings to a Model).

To receive those publications from the SBMVendorModel in SBMNode, the local stack must be able to collect those messages. The library must know what address it should follow, either the SBMGroup address or the direct SBMNode address. 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.

7.22.9.1 Sign Up for Publications

This section describes the method used to subscribe to the vendor model publications.

Note: Publications will come from the SBMLocalVendorSettingsMessageHandler in SBMLocalVendorSettings, which should previously have been configured by SBMLocalVendorRegistrator.

7.22.9.1.1 Publications sent to SBMGroup Address

```
let group: SBMGroup
let vendorModel: SBMVendorModel

//(…)

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

7.22.9.1.2 Notifications Sent to SBMNode Address

```
let node: SBMNode
let vendorModel: SBMVendorModel

//(…)

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

7.22.9.2 Sign Out from Publications

This section describes the method used to unsubscribe from the vendor model publications.

7.22.9.2.1 Sign Out from Publications Sent to the SBMGroup Address

```
let group: SBMGroup
let vendorModel: SBMVendorModel

//(…)

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

7.22.9.2.2 Sign Out from Publications Sent to the SBMNode Address

```
let node: SBMNode
let vendorModel: SBMVendorModel

//(…)

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

7.22.10 Send Value to SBMVendorModel

7.22.10.1 Create Implementation of the SBMControlValueSetVendorModel Protocol

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

Example:

```
class CompanyControlSetVendorModel: NSObject, SBMControlValueSetVendorModel {
    var localVendorModelClient: SBMLocalVendorModel? // this property is deprecated, use
vendorModel instead
    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
    }
}
```

7.22.10.2 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:

```
let vendorModel: SBMVendorModel
let messageToSend: Data

//(…)

let companyID = vendorModel.vendorCompanyIdentifier()

//(…)

var data = Data(bytes: [UInt8(0) | 0xC0])
// In this example, an 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)
```

7.22.10.3 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 in SBMLocalVendorSettings, which should previously have been configured by SBMLocalVendorRegistrator.

```
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)
```

7.22.10.4 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)
```

7.22.11 Control Light Control (LC) Models

Light Control (LC) models are used to handle automated lighting. These models are designed to automatically control a dimmable light on a device. LC Models have multiple configurable parameters and they are also able to receive external inputs from sensors.

The Light Control API allows for:

- Get or set LC Mode state
- Get or set LC Occupancy Mode state
- Get or set LC Light On-Off state
- Get or set LC Property state

To set up publication/subscription settings use the API prepared for SIG models (see sections 7.20.1 SIG Model and 7.21.1 SIG Model, respectively).

To set up LC model bindings use the API prepared for SIG models (see sections 7.18 Bind a Model with a Group and 7.19 Unbind a Model from a Group).

7.22.11.1 Get LC Model Values

7.22.11.1.1 Get LC Mode from the Node

The request to get LC mode state has no parameters.

The value received in SBMLightControlModeStatus object (mode) is an enumerated type and has two values:

- SBMLightControlModeOff - the controller is turned off. The binding with the Light Lightness state is disabled.
- SBMLightControlModeOn – the controller is turned on. The binding with the Light Lightness state is enabled.

Response data received in the SBMLightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status in the *Mesh Model Bluetooth® Specification*).

```
func getLightControlMode(for element: SBMElement, in group: SBMGroup) {
    let message = SBMLightControlModeGet()
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.getLightControlResponse(message, successCallback: { controlElement, response in
        let response = response as! SBMLightControlModeStatus
        // Handle result
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.1.2 Get LC Occupancy Mode from the Node

The request to get LC Occupancy Mode state has no parameters.

The value received in SBMLightControlOccupancyModeStatus object is an enumerated type and has one of two values:

- SBMLightControlOccupancyModeTypeStandbyTransitionDisabled – the controller does not transition from a standby state when occupancy is reported.

- SBMLightContrlOccupancyModeTypeStandbyTransitionEnabled – the controller may transition from a standby state when occupancy is reported.

Response data received in the SBMLightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

```
func getLightControlOccupancyMode(for element: SBMElement, in group: SBMGroup) {
    let message = SBMLightControlOccupancyModeGet()
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.getLightControlResponse(message, successCallback: { controlElement, response in
        let response = response as! SBMLightControlOccupancyModeStatus
        // Handle result
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.1.3 Get LC Light On-Off from the Node

The request to get LC Light On-Off state has no parameters.

The values received in the object are as follows:

- presentLightOnOff – is an enumerated type SBMLightControlLightOnOffType and represents the state of a Light Lightness controller. This variable can have the following values:
  - SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  - SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an optional NSNumber. If this value is present, its value should be decoded as SBMLightControlLightOnOffType (see above). This value is optional, meaning it does not have to be in the received status response. If it is not, this value will be set to nil.
- remainingTime – is of type NSNumber and specifies the time remaining to complete the transition. This value will be nil unless the Target Light On-Off value is specified. The format of this value is specified in 3.1.3 Generic Default Transition Time from the *Mesh Model Bluetooth Specification*.

Response data received in the SBMLightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

```
func getLightControlOnOff(for element: SBMElement, in group: SBMGroup) {
    let message = SBMLightControlLightOnOffGet()
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.getLightControlResponse(message, successCallback: { controlElement, response in
        let response = (response as! SBMLightControlLightOnOffStatus)
        // Handle result
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.1.4 Get LC Property from the Node

The request to get LC Property state has one parameter:

- propertyId – UInt16 identifying the ID of the property whose value is requested.

The values received in the SBMLightControlPropertyStatus object are as follows:

- propertyId – specifies the ID of the property whose value is received.
- value – NSData value for the received Property.

Response data received in the SBMLightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

```
func getLightControlProperty(for element: SBMElement, in group: SBMGroup, propertyId: UInt16) {
```

```
    let message = SBMLightControlPropertyGet(propertyID: propertyId)
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.getLightControlResponse(message, successCallback: { controlElement, response in
        let response = (response as! SBMLightControlPropertyStatus)
        // Handle result
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.1.5 Get LC Mode from All Nodes in the Group

The request to get LC mode state has no parameters.

The value received in the SBMLightControlModeStatus object (mode) is an enumerated type and has one of two values:

- SBMLightControlModeOff - the controller is turned off. The binding with the Light Lightness state is disabled.
- SBMLightControlModeOn – the controller is turned on. The binding with the Light Lightness state is enabled.

Response data received in the SBMLightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status 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 getMode(from group: SBMGroup) {

    let message = SBMLightControlModeGet()

    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.getLightControlResponse(message, successHandler: { controlGroup,
response, element in
        let response = response as! SBMLightControlModeStatus
        // Handle response
    }, errorHandler: { controlGroup, error in
        // Handle error
    })
}
```

7.22.11.1.6 Get LC Occupancy Mode from All Nodes in the Group

The request to get LC Occupancy Mode state has no parameters.

The value received in SBMLightControlOccupancyModeStatus object (mode) is an enumerated type and has one of two values:

- SBMLightControlOccupancyModeTypeStandbyTransitionDisabled – the controller does not transition from a standby state when occupancy is reported.
- SBMLightContrlOccupancyModeTypeStandbyTransitionEnabled – the controller may transition from a standby state when occupancy is reported.

Response data received in the SBMLightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status 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 getOccupancyMode(from group: SBMGroup) {
    let message = SBMLightControlOccupancyModeGet()

    let controlGroup = SBMControlGroup(group: group)
```

```
    let task = controlGroup.getLightControlResponse(message, successHandler: { controlGroup,
response, element in
        let response = response as! SBMLightControlOccupancyModeStatus
        // Handle response
    }, errorHandler: { group, error in
        // Handle error
    })
}
```

7.22.11.1.7 Get LC Light On-Off from All Nodes in the Group

The request to get LC Light On-Off state has no parameters.

The values received in object are as followed:

- presentLightOnOff – is an enumerated type SBMLightControlLightOnOffType and represents the state of a Light Lightness controller. This variable can have the following values:
  - SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  - SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an optional NSNumber. If this value is present, it should be decoded as SBMLightControlLightOnOffType (see above). This value is optional, meaning it does not have to be in the received status response. If it is not, this value will be set to nil.
- remainingTime – is of type NSNumber and specifies the time remaining to complete the transition. This value will be nil unless the Target Light On-Off value is specified. The format of this value is specified in 3.1.3 Generic Default Transition Time from the *Mesh Model Bluetooth Specification*.

Response data received in the SBMLightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status 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 getLightOnOff(from group: SBMGroup) {
    let message = SBMLightControlLightOnOffGet()

    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.getLightControlResponse(message, successHandler: { controlGroup,
response, element in
        let response = response as! SBMLightControlLightOnOffStatus
        // Handle response
    }, errorHandler: { group, error in
        // Handle error
    })
}
```

7.22.11.1.8 Get LC Property from All Nodes in the Group

The request to get LC Property state has one parameter:

- propertyId – UInt16 identifying ID of the property whose value is requested.

The values received in the SBMLightControlPropertyStatus object are as followed:

- propertyId – specifies the ID of the property whose value is received.
- value – NSData value for the received Property.

Response data received in the SBMLightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status 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 getProperty(propertyId: UInt16, from group: SBMGroup) {
    let message = SBMLightControlPropertyGet(propertyID: propertyId)
    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.getLightControlResponse(message, successHandler: { controlGroup,
response, element in
        let response = response as! SBMLightControlPropertyStatus
        // Handle response
    }, errorHandler: { controlGroup, error in
        // Handle error
    })
}
```

7.22.11.2 Set LC Setup Model Values

7.22.11.2.1 Set LC Mode within the Node

The request to set LC Mode state has two parameters:

- flags – object of type SBMLightControlMessageFlags which determines if the message requires acknowledgment from the node.
- mode – an enumerated type SBMLightControlMode that has one of two values:
  - SBMLightControlModeOff - the controller is turned off. The binding with the Light Lightness state is disabled.
  - SBMLightControlModeOn – the controller is turned on. The binding with the Light Lightness state is enabled.

This variable determines the desired value for LC Mode state of target element.

The value received in SBMLightControlModeStatus object is an enumerated type SBMLightControlMode (for details see above).

Response data received in the SBMLightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set request as acknowledged, acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

```
func setMode(for element: SBMElement, in group: SBMGroup, mode: SBMLightControlMode, acknowledged:
Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlModeSet(flags: flags, mode: mode)

    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.setLightControlValue(message, successCallback: { controlElement, response in
        let response = response as! SBMLightControlModeStatus
        // Handle result
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.2.2 Set LC Occupancy Mode within the Node

The request to set LC Occupancy Mode state has two parameters:

flags – object of type SBMLightControlMessageFlags which determines if message requires acknowledgment from the node.

mode – an enumerated type SBMLightControlOccupancyModeType which has two values:

- SBMLightControlOccupancyModeTypeStandbyTransitionDisabled – the controller does not transition from a standby state when occupancy is reported.
- SBMLightContrlOccupancyModeTypeStandbyTransitionEnabled – the controller may transition from a standby state when occupancy is reported.

This variable determines desired value for LC Occupancy Mode state of target element.

The value received in SBMLightControlOccupancyModeStatus object (status) is an enumerated type SBMLightControlOccupancyModeType (for details see above).

Response data received in the SBMLightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set request as acknowledged, set the acknowledgeRequired property in the SBMLightControlMessageFlags object to true.

```
func setOccupancyMode(for element: SBMElement, in group: SBMGroup, mode:
SBMLightControlOccupancyModeType, acknowledged: Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlOccupancyModeSet(flags: flags, mode: mode)

    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.setLightControlValue(message, successCallback: { element, response in
        let response = response as! SBMLightControlOccupancyModeStatus
        // Handle response
    }, errorCallback: { element, error in
        // Handle error
    })
}
```

7.22.11.2.3 Set LC Light On-Off within the Node

The request to set LC Light On-Off state has five parameters:

- flags – object of type SBMLightControlMessageFlags which determines if message requires acknowledge from the node.
- lightOnOff – determines desired value for LC Light On-Off state of target element. This variable is an enumerated type SBMLightControlLightOnOffType and has two values:
  - SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  - SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- transactionId – 8-bit unsigned number identifying Transaction Identifier (TID). This value indicates whether the message is a new message or a retransmission of a previously sent message.
- transitionTime – Is of type UInt32 and specifies the time an element will take to transition to the target state in milliseconds. This value is optional. If this value will not be set (is null or has a value of 0xFFFFFFFF) the delay parameter will not be consider in the request.
- delay – Is of type UInt32 and determines request execution delay in milliseconds. This value will not be considered in the request if transitionTime is not set (has default value 0 or 0xFFFFFFFF).

For full characteristics of LC Light OnOff Set Message see 6.3.5.3.2 Light LC Light OnOff Set from the *Mesh Model Bluetooth Specification*.

The values received in SBMLightControlLightOnOffStatus object (status) are as follows:

- presentLightOnOff – is an enumerated type SBMLightControlLightOnOffType and represents the state of a Light Lightness controller. This variable can have the following values:
  a) SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  b) SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an optional NSNumber. If this value is present, its value should be decoded as SBMLightControlLightOnOffType (see above). This value is optional which means it doesn't necessarily have to be in the received status response. If it is not this value will be set to nil.
- remainingTime – is of type NSNumber and specifies the time remaining to complete the transition. This value will be nil unless the Target Light On-Off value is specified. Format of this value is specified in 3.1.3 Generic Default Transition Time from the Mesh Model Mluetooth Specification.

Response data received in the SBMLightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

```
func setLightOnOff(for element: SBMElement, in group: SBMGroup, onOff:
SBMLightControlLightOnOffType, transactionId: UInt8, transitionTime: UInt32, delay: UInt32,
acknowledged: Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlLightOnOffSet(flags: flags,
                                               lightOnOff: onOff,
                                               transactionId: transactionId,
                                               transitionTime: transitionTime,
                                               delay: delay)

    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.setLightControlValue(message, successCallback: { element, response in
        let response = response as! SBMLightControlLightOnOffStatus
        // Handle response
    }, errorCallback: { element, error in
        // Handle error
    })
}
```

7.22.11.2.4 Set LC Property within the Node

The request to set LC Property state has three parameters:

- flags – object of type SBMLightControlMessageFlags which determines if the message requires acknowledge from the node.
- propertyId – 16-bit unsigned number identifying the ID of the property whose value will be set.
- propertyValue – NSData value to set for the Property.

The values received in SBMLightControlPropertyStatus object are as follows:

- propertyId – specifies the ID of property which value is received.
- value – NSData value for the received Property.

Response data received in the SBMLightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

```
func setProperty(for element: SBMElement, in group: SBMGroup, propertyId: UInt16, propertyValue:
Data, acknowledged: Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlPropertySet(flags: flags, propertyId: propertyId, propertyValue:
propertyValue)
    let controlElement = SBMControlElement(element: element, in: group)
    controlElement.setLightControlValue(message, successCallback: { controlElement, response in
        let response = response as! SBMLightControlPropertyStatus
        // Handle response
    }, errorCallback: { controlElement, error in
        // Handle error
    })
}
```

7.22.11.2.5 Set LC Mode within All Nodes in the Group

The request to set LC Mode state has two parameters:

- flags – object of type SBMLightControlMessageFlags which determines if the message requires acknowledgement from the node.
- mode – an enumerated type SBMLightControlMode which has two values:
  - SBMLightControlModeOff - the controller is turned off. The binding with the Light Lightness state is disabled.
  - SBMLightControlModeOn – the controller is turned on. The binding with the Light Lightness state is enabled.

This variable determines the desired value for LC Mode state of target elements.

The value received in SBMLightControlModeStatus object is an enumerated type SBMLightControlMode (for details see above).

Response data received in the SBMLightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

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 setMode(for group: SBMGroup, mode: SBMLightControlMode, acknowledged: Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlModeSet(flags: flags, mode: mode)

    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setLightControlValue(message, successHandler: { controlGroup, response,
element in
        let response = response as! SBMLightControlModeStatus
        // Handle response
    }, errorHandler: { controlGroup, error in
        // Handle error
    })
}
```

7.22.11.2.6 Set LC Occupancy Mode within All Nodes in the Group

The request to set LC Occupancy Mode state has two parameters:

- flags – object of type SBMLightControlMessageFlags which determines if message requires acknowledge from the node.
- mode – an enumerated type SBMLightControlOccupancyModeType which has two values:
  - SBMLightControlOccupancyModeTypeStandbyTransitionDisabled – the controller does not transition from a standby state when occupancy is reported.
  - SBMLightContrlOccupancyModeTypeStandbyTransitionEnabled – the controller may transition from a standby state when occupancy is reported.

This variable determines the desired value for LC Occupancy Mode state of target elements.

The value received in SBMLightControlOccupancyModeStatus object is an enumerated type SBMLightControlOccupancyModeType (for details see above).

Response data received in the SBMLightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

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 setOccupancyMode(for group: SBMGroup, mode: SBMLightControlOccupancyModeType, acknowledged:
Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlOccupancyModeSet(flags: flags, mode: mode)

    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setLightControlValue(message, successHandler: { controlGroup, response,
element in
        let response = response as! SBMLightControlOccupancyModeStatus
        // Handle response
    }, errorHandler: { group, error in
        // Handle error
    })
}
```

7.22.11.2.7 Set LC Light On-Off within All Nodes in the Group

The request to set the LC Light On-Off state has five parameters:

- flags – object of type SBMLightControlMessageFlags which determines if the message requires acknowledgment from the node.
- lightOnOff – determines the desired value for LC Light On-Off state of target elements. This variable is an enumerated type SBMLightControlLightOnOffType and has two values:
  - SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  - SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- transactionId – 8-bit unsigned number identifying Transaction Identifier (TID). This value indicates whether the message is a new message or a retransmission of a previously sent message.
- transitionTime – Is of type UInt32 and specifies the time an element will take to transition to the target state in milliseconds. This value is optional. If this value will not be set (is null or has a value of 0xFFFFFFFF) the delay parameter will not be considered in the request.
- delay – Is of type UInt32 and determines the request execution delay in milliseconds. This value will not be considered in the request if transitionTime is not set (has a default value of 0 or 0xFFFFFFFF).

For full characteristic of LC Light OnOff Set Message see 6.3.5.3.2 Light LC Light OnOff Set from the *Mesh Model Bluetooth Specification*.

The values received in SBMLightControlLightOnOffStatus object (status) are as follows:

- presentLightOnOff – is an enumerated type SBMLightControlLightOnOffType and represents the state of a Light Lightness controller. This variable can have following values:
  - SBMLightControlLightOnOffTypeOffOrStandby – State is equal to Off or equal to Standby.
  - SBMLightControlLightOnOffTypeNotOffAndNotStandby - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an optional NSNumber. If this value is present, its value should be decoded as SBMLightControlLightOnOffType (see above). This value is optional, meaning it does not have to be in the received status response. If it is not this value will be set to nil.
- remainingTime – is of type NSNumber and specifies the time remaining to complete the transition. This value will be nil unless the Target Light On-Off value is specified. Format of this value is specified in 3.1.3 Generic Default Transition Time from the *Mesh Model Mluetooth Specification*.

Response data received in the SBMLightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

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 setLightOnOff(for group: SBMGroup, onOff: SBMLightControlLightOnOffType, transactionId: UInt8,
transitionTime: UInt32, delay: UInt32, acknowledged: Bool) {
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlLightOnOffSet(flags: flags,
                                               lightOnOff: onOff,
                                               transactionId: transactionId,
                                               transitionTime: transitionTime,
                                               delay: delay)

    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setLightControlValue(message, successHandler: { controlGroup, response,
element in
        let response = response as! SBMLightControlLightOnOffStatus
        // Handle response
    }, errorHandler: { element, error in
        // Handle error
    })
}
```

7.22.11.2.8 Set LC Property within All Nodes in the Group

The request to set LC Property state has three parameters:

- flags – object of type SBMLightControlMessageFlags which determines if the message requires acknowledgment from the node.
- propertyId – 16-bit unsigned number identifying the ID of the property whose value will be set.
- propertyValue – NSData value to be set for the Property.

The values received in the SBMLightControlPropertyStatus object are as follows:

- propertyId – specifies the ID of the Property that received the value.
- value – NSData value for the received Property.

Response data received in the SBMLightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set the request as acknowledged, the acknowledgeRequired property in SBMLightControlMessageFlags object must be set to true.

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 setProperty(for group: SBMGroup, propertyId: UInt16, propertyValue: Data, acknowledged: Bool)
{
    let flags = SBMLightControlMessageFlags()
    flags.acknowledgeRequired = acknowledged

    let message = SBMLightControlPropertySet(flags: flags, propertyId: propertyId, propertyValue:
propertyValue)
    let controlGroup = SBMControlGroup(group: group)
    let task = controlGroup.setLightControlValue(message, successHandler: { controlGroup, response,
element in
        let response = response as! SBMLightControlPropertyStatus
        // Handle response
    }, errorHandler: { controlGroup, error in
        // Handle error
    })
}
```

7.22.11.3 Subscriptions

Publications with statuses can be received from LC models. It means that a node can publish its current state to all subscribers.

Subscription can be done to a specific element or to a group and to one or more status types.

To receive group publications after subscription, the node and specific model must be bound to the group. Also, the node must be configured to publish its status to the specific address, either group or device, that matches the subscription.

7.22.11.3.1 Subscription to Status Publications from a Single Node

The subscription takes one parameter that describes the status of which to be notified. This object is the same as when manually requesting a single status report.

The success handler is executed every time the subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object that contains its current status.

The error handler is executed either if the subscription failed or a node has reported any error.

After subscription, the callback will be called for every received publication, until manually unsubscribed from this notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

```
func subscribeToModeChange() {
    let message = SBMLightControlModeGet()

    let publications = SBMControlElementPublications(element: element, in: group)
    let task = publications.subscribe(toLightControlStatus: message, successHandler: { element,
group, response in
        let response = response as! SBMLightControlModeStatus
        // Handle response
    }, errorHandler: { element, group, error in
        // Handle error
    })
}
```

7.22.11.3.2 Subscription to Status Publications from a Group of Nodes

Subscription to group publications is similar to subscription to a specific node. The only difference is that the subscribe method from SBMControlGroupPublications should be used instead of SBMControlElementPublications and that this object's constructor takes only a group identifier.

```
func subscribeToGroupModeChange() {
    let message = SBMLightControlModeGet()

    let publications = SBMControlGroupPublications(group: group)
    let task = publications.subscribe(toLightControlStatus: message, successHandler: { element,
group, response in
        let response = response as! SBMLightControlModeStatus
        // Handle response
    }, errorHandler: { group, error in
        // Handle error
    })
}
```

7.22.12 Control Scene Models

Scenes serve as memory banks for storage of states (such as a power level or a light level/color). Values of states of an element can be stored as a scene and can be recalled later from the scene memory. Ultimately scenes support adjusting multiple states of different nodes in one action.

The Scene API allows for:

- Get the current status of a currently active scene of an element
- Get the current status of the Scene Register of an element. The Scene Register state is a 16-element array of 16-bit values representing a scene number. These values are associated with a storage container that stores information about the scene state.
- Store the current state of an element as a Scene
- Recall the current state of an element from a previously stored scene.
- Delete a scene from the Scene Register state of an element.

To set up publication/subscription settings use the API developed for SIG models (see sections 7.20.1 SIG Model and 7.21.1 SIG Model, respectively).

To set up Scene model bindings use the API developed for SIG models (see sections 7.18 Bind a Model with a Group and 7.19 Unbind a Model from a Group).

7.22.12.1 Get Scene Model Values

7.22.12.1.1 Get Scene Status from a Node

The request to get Scene Status has no parameters.

The response to a Scene Status Get request is the SBMSceneStatus object. This object has four properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is nil when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is nil when the scene changing process does not occur.

Response data received in the SBMSceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

```
    func getScene(for element: SBMElement, group: SBMGroup) {
        let message = SBMSceneGet();
        let controlElement = SBMControlElement(element: element, in: group)

        controlElement.getSceneValue(message, successCallback: { (element, response) in
            // Handle response
        }) { (element, error) in
            // Handle error
        }
    }
```

7.22.12.1.2 Get Scene Register Status from a Node

The request to get Scene Register Status has no parameters.

The response to a Scene Register Status Get request is the SBMSceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

```
func getSceneRegister(for element: SBMElement, group: SBMGroup) {
    let message = SBMSceneRegisterGet();
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.getSceneValue(message, successCallback: { (element, response) in
        // Handle response
    }) { (element, error) in
        // Handle error
    }
}
```

7.22.12.1.3 Get Scene Status from All Nodes in a Group

The request to get Scene Status has no parameters.

The response to a Scene Status Get request is the SBMSceneStatus object. This object has four properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is nil when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is nil when the scene changing process does not occur.

Response data received in the SBMSceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

Note that the success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
func getScene(for group: SBMGroup) {
    let message = SBMSceneGet();
    let controlGroup = SBMControlGroup(group: group)

    let task = controlGroup.getSceneValue(message, successHandler: { (group, response, element) in
        // Handle response
    }) { (group, error) in
        // Handle error
    }
}
```

7.22.12.1.4 Get Scene Register Status from All Nodes in a Group

The request to get Scene Register Status has no parameters.

The response to a Scene Register Status Get request is the SBMSceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

Note that the success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
func getSceneRegister(for group: SBMGroup) {
    let message = SBMSceneRegisterGet();
    let controlGroup = SBMControlGroup(group: group)

    let task = controlGroup.getSceneValue(message, successHandler: { (group, response, element) in
        // Handle response
    }) { (group, error) in
        // Handle error
    }
}
```

7.22.12.2 Control Scene Model Values

7.22.12.2.1 Store Scene within a Node

The request to store scene has two parameters:

- flags – object of type SBMSceneMessageFlags which determines if the message requires acknowledgment from the node.
- scene – an object that represents a SBMScene to be stored. Use the createScene(withName, number) method from the SBMNetwork object to get new instance of a scene object.

The response to a Scene Store request is the SBMSceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` property of the SBMSceneMessageFlags object to determine the required value of the message flag.

```
func storeScene(for element: SBMElement, group: SBMGroup, scene: SBMScene, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    let message = SBMSceneStore(flags: flags, scene: scene);
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.changeSceneRegister(message, successCallback: { (element, response) in
        // Handle response
    }) { (element, error) in
        // Handle error
    }
}
```

7.22.12.2.2 Recall Scene Within a Node

The request to recall a scene has three or five parameters:

- flags – object of type SBMSceneMessageFlags which determines if the message requires acknowledgment from the node.
- scene – an object that represents a SBMScene to be recalled.
- transactionId – 8-bit unsigned number identifying the Transaction Identifier (TID). This value indicates whether the message is a new message or a re-transmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transit to the state defined by the scene being recalled. This value is optional. If this value is not initialized, the delay parameter will not be considered in the request.
- delay – Is of type Integer and determines the request execution delay in milliseconds. This value will not be considered in the request if the transition-time is not initialized or has maximum value of 0xFFFFFFFF.

The response to a Scene Recall request is the SBMSceneStatus object. This object has four properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is nil when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is nil when the scene changing process does not occur.

Response data received in the SBMSceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` property of the SBMSceneMessageFlags object to determine the required value of the message flag.

```
func recallScene(for element: SBMElement, group: SBMGroup, scene: SBMScene, transactionId: UInt8,
transitionTime: UInt32?, delay: UInt32?, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    var message : SBMSceneRecall
    if transitionTime != nil {
        message = SBMSceneRecall(flags: flags, scene: scene, transactionId: transactionId,
transitionTime: transitionTime!, delay: delay!);
```

```
    } else {
        message = SBMSceneRecall(flags: flags, scene: scene, transactionId: transactionId);
    }

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

    controlElement.changeSceneRegister(message, successCallback: { (element, response) in
        // Handle response
    }) { (element, error) in
        // Handle error
    }
}
```

7.22.12.2.3 Delete Scene Within a Node

The request to delete a scene has two parameters:

- flags – object of type SBMSceneMessageFlags that determines if the message requires acknowledgment from the node.
- scene – an object that represents a Scene to be deleted.

The response to a Scene Delete request is the SBMSceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` property of the SBMSceneMessageFlags object to determine the required value of message flag.

```
func deleteScene(for element: SBMElement, group: SBMGroup, scene: SBMScene, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    let message = SBMSceneDelete(flags: flags, scene: scene);
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.changeSceneRegister(message, successCallback: { (element, response) in
        // Handle response
    }) { (element, error) in
        // Handle error
    }
}
```

7.22.12.2.4 Store Scene Within All Nodes in a Group

The request to store a scene has two parameters:

- flags – object of type SBMSceneMessageFlags which determines if the message requires acknowledgment from the node.
- scene – an object that represents a SBMScene to be stored. Use the createScene(withName, number) method from the SBMNetwork object to get new instance of a scene object.

The response to a Scene Store request is the SBMSceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull

  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` property of the SBMSceneMessageFlags object to determine the required value of the message flag.

Note that the success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses, call the `cancel()` method on task object: `task.cancel()`.

```
func storeScene(for group: SBMGroup, scene: SBMScene, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    let message = SBMSceneStore(flags: flags, scene: scene);
    let controlGroup = SBMControlGroup(group: group)

    let task = controlGroup.changeSceneRegister(message, successHandler: { (SBMControlGroup,
SBMSceneResponse, SBMElement) in
            // Handle responses
    }) { (SBMControlGroup, Error) in
        // Handle error
    }
}
```

7.22.12.2.5 Recall Scene within All Nodes in a Group

The request to recall a scene has three or five parameters:

- flags – object of type SBMSceneMessageFlags which determines if the message requires acknowledgment from the node.
- scene – an object that represents a SBMScene to be recalled.
- transactionId – 8-bit unsigned number identifying the Transaction Identifier (TID). This value indicates whether the message is a new message or a re-transmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transit to the state defined by the scene being recalled. This value is optional. If this value is not initialized, the delay parameter will not be considered in the request.
- delay – Is of type Integer and determines the request execution delay in milliseconds. This value will not be considered in the request if the transition-Time is not initialized or has maximum value of 0xFFFFFFFF.

The response to a Scene Recall request is the SBMSceneStatus object. This object has four properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is nil when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is nil when the scene changing process does not occur.

Response data received in the SBMSceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` property of the SBMSceneMessageFlags object to determine the required value of the message flag.

Note that the success method is called every time the provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
func recallScene(for group: SBMGroup, scene: SBMScene, transactionId: UInt8, transitionTime:
UInt32?, delay: UInt32?, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    var message : SBMSceneRecall
    if transitionTime != nil {
        message = SBMSceneRecall(flags: flags, scene: scene, transactionId: transactionId,
transitionTime: transitionTime!, delay: delay!);
    } else {
        message = SBMSceneRecall(flags: flags, scene: scene, transactionId: transactionId);
    }
    let controlGroup = SBMControlGroup(group: group)

    let task = controlGroup.changeSceneRegister(message, successHandler: { (SBMControlGroup,
SBMSceneResponse, SBMElement) in
            // Handle responses
    }) { (SBMControlGroup, Error) in
        // Handle error
    }
}
```

7.22.12.2.6 Delete Scene Within All Nodes in a Group

The request to delete a scene has two parameters:

- flags – object of type SBMSceneMessageFlags which determines if the message requires acknowledgment from the node.
- scene – an object that represents a Scene to be deleted.

The response to a Scene Delete request is SceneRegisterStatus object. This object has three properties:

- status – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SBMSceneStatusCodeTypeSuccess
  - SBMSceneStatusCodeTypeSceneRegisterFull
  - SBMSceneStatusCodeTypeSceneNotFound
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SBMSceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SBMSceneMessageFlags object to the request constructor. Use the `isAcknowledgeRequired` method of the SBMSceneMessageFlags object to determine the required value of the message flag.

Note that the success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
func deleteScene(for group: SBMGroup, scene: SBMScene, acknowledged: Bool) {
    let flags = SBMSceneMessageFlags()
    flags.isAcknowledgedRequired = acknowledged

    let message = SBMSceneDelete(flags: flags, scene: scene);
    let controlGroup = SBMControlGroup(group: group)

    let task = controlGroup.changeSceneRegister(message, successHandler: { (SBMControlGroup,
SBMSceneResponse, SBMElement) in
            // Handle responses
    }) { (SBMControlGroup, Error) in
        // Handle error
```

```
    }
}
```

7.22.12.3 Subscriptions

7.22.12.3.1 Subscription to Status Publications from a Single Node

The subscription takes one parameter that describes the status to be notified about. This object is the same as when manually requesting a single status report.

The success handler is executed every time that a subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object contains its current status.

The error handler is executed either if the subscription request fails or the node reports any error.

After subscription, a callback will be called for every received notification until manually unsubscribed from the notification. To unsubscribe, the `cancel()` method should be called on the `task` object returned after subscription.

```
func subscribeToSceneRegisterGet(element: SBMElement, group: SBMGroup) {
    let message = SBMSceneRegisterGet()

    let publications = SBMControlElementPublications(element: element, in: group)
    let task = publications.subscribe(toSceneStatus: message, successHandler: { (element, group,
response) in
        // Handle response
    }) { (element, group, error) in
        // Handle error
    }
}
```

7.22.12.3.2 Subscription to Status Publications from a Group of Nodes

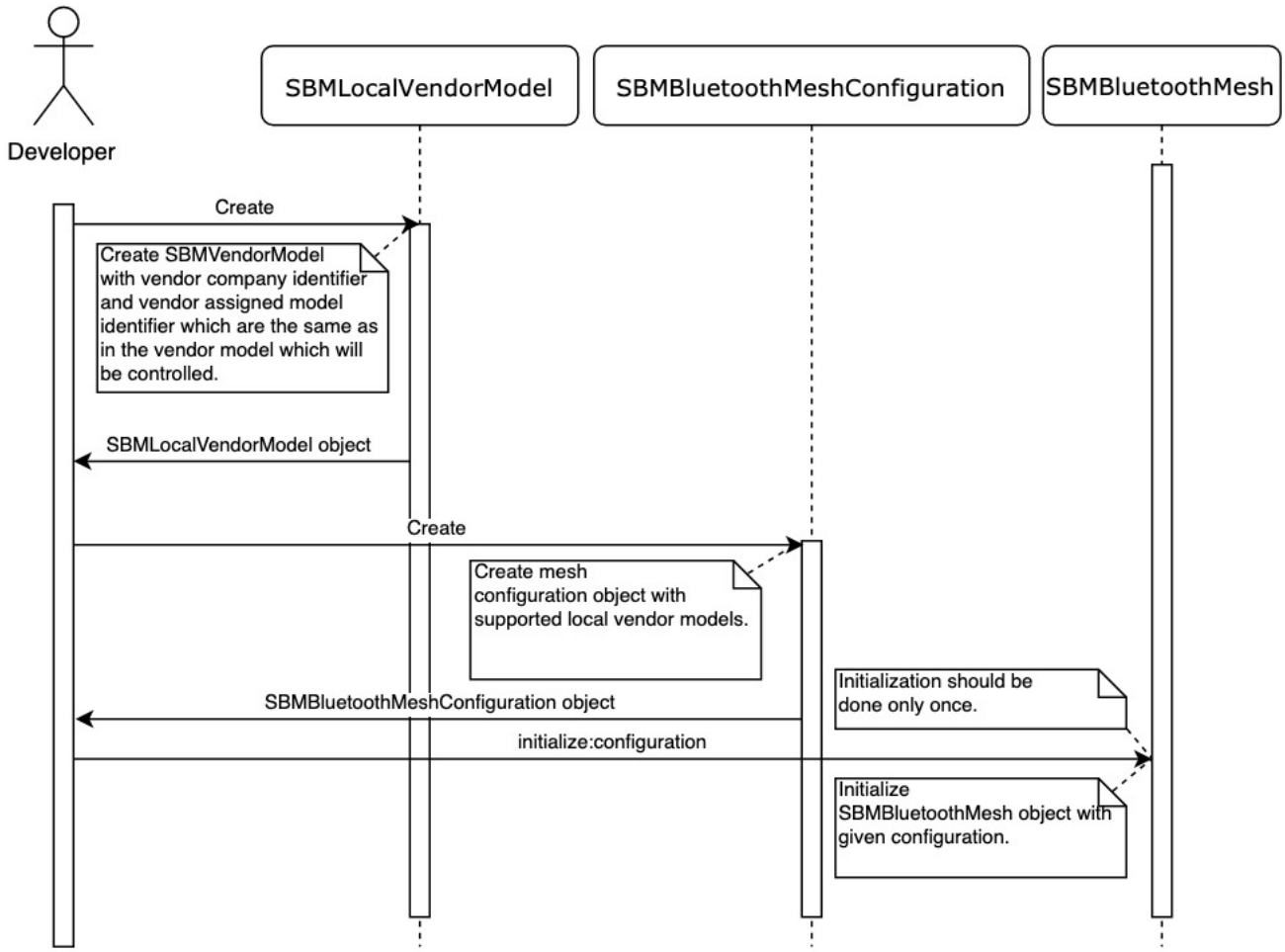
Subscription to group notifications is similar to subscription to a specific node. The only difference is that the subscribe method from SBMControlGroupPublications should be used instead of that from SBMControlElementPublications and that this object's constructor takes only a group identifier.

```
func subscribeToSceneRegisterGet( group: SBMGroup) {
    let message = SBMSceneRegisterGet()

    let publications = SBMControlGroupPublications(group: group)
    let task = publications.subscribe(toSceneStatus: message, successHandler: { (element, group,
response) in
        // Handle response
    }) { (group, error) in
        // Handle error
    }
}
```

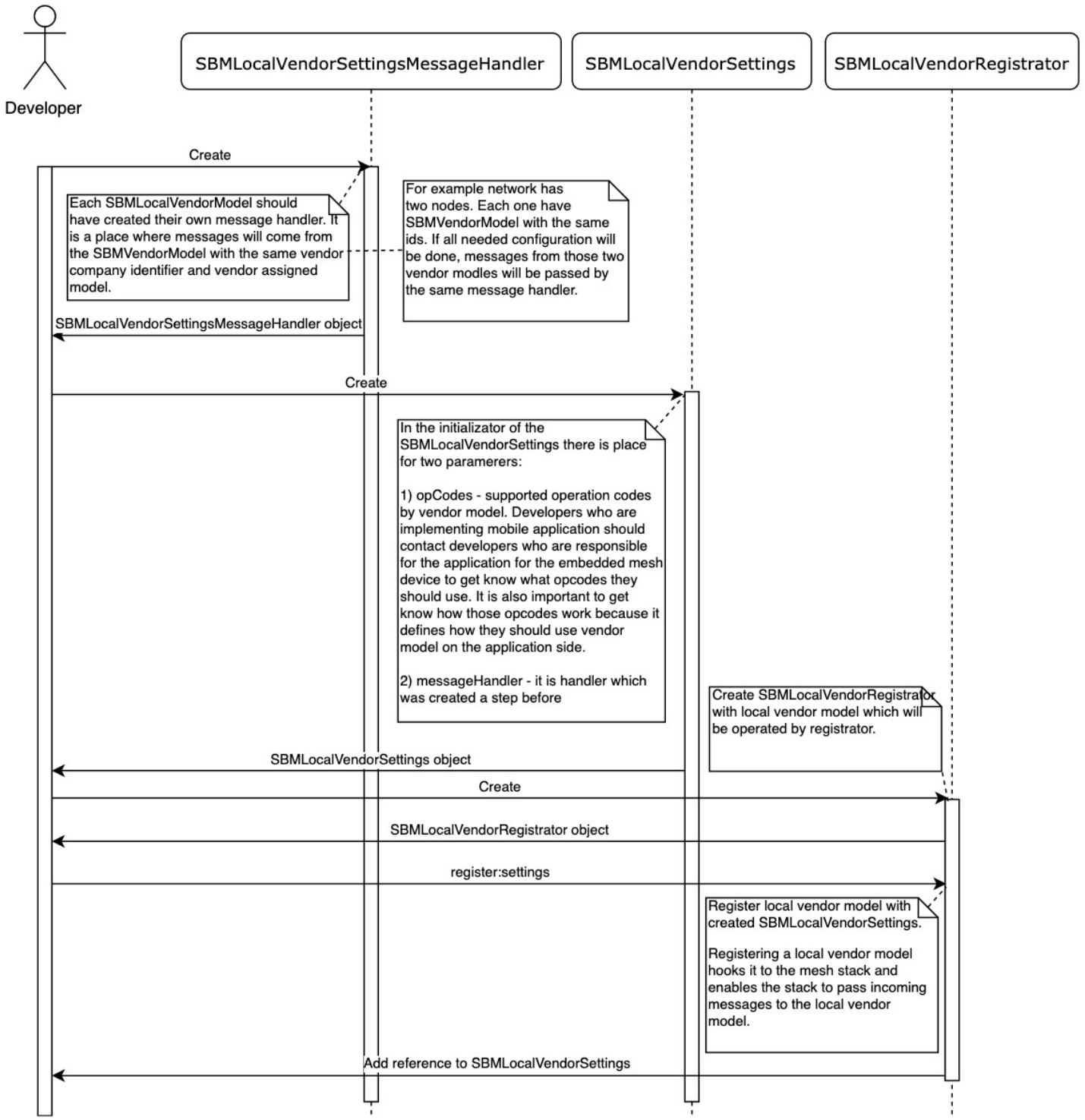
7.23 Sequence Diagrams for Vendor Model Functionality

7.23.1 SBMLocalVendorModel Initialization

7.23.1.1 Create SBMLocalVendorModel



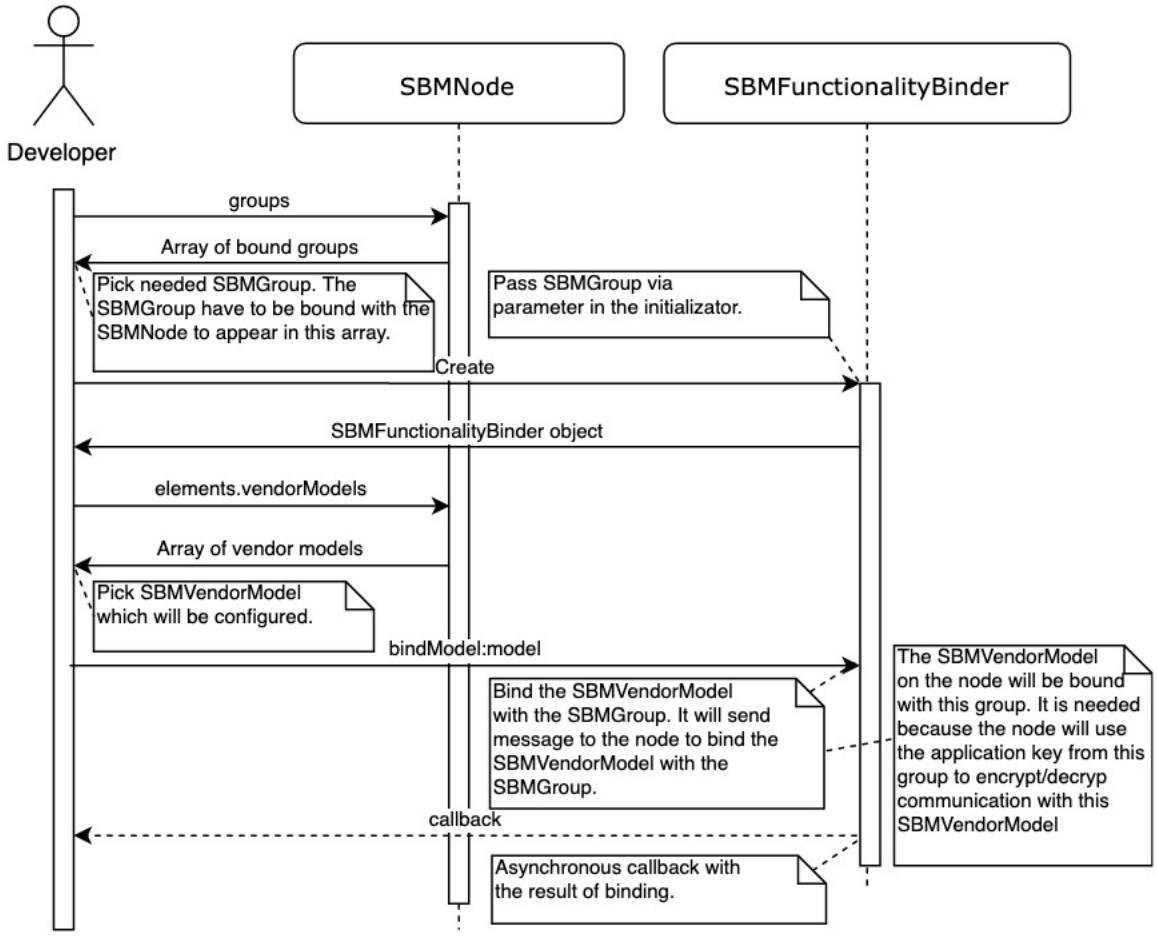
7.23.1.2 Register the SBMLocalVendorModel in the Mesh Stack



7.23.2 Configure the SBMVendorModel

7.23.2.1 Bind the SBMVendorModel with the SBMGroup

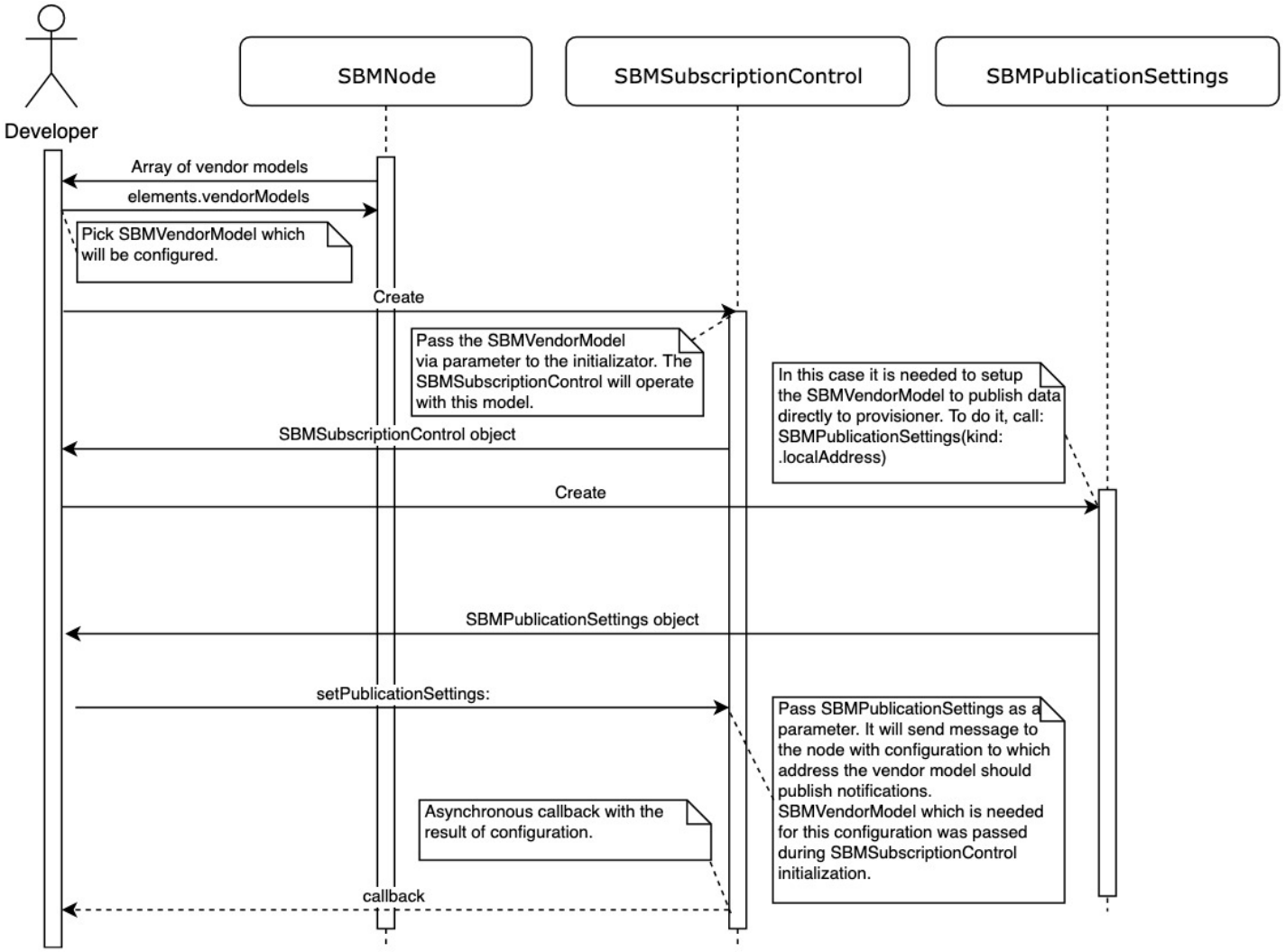
The vendor model must always be bound on the node with the group.



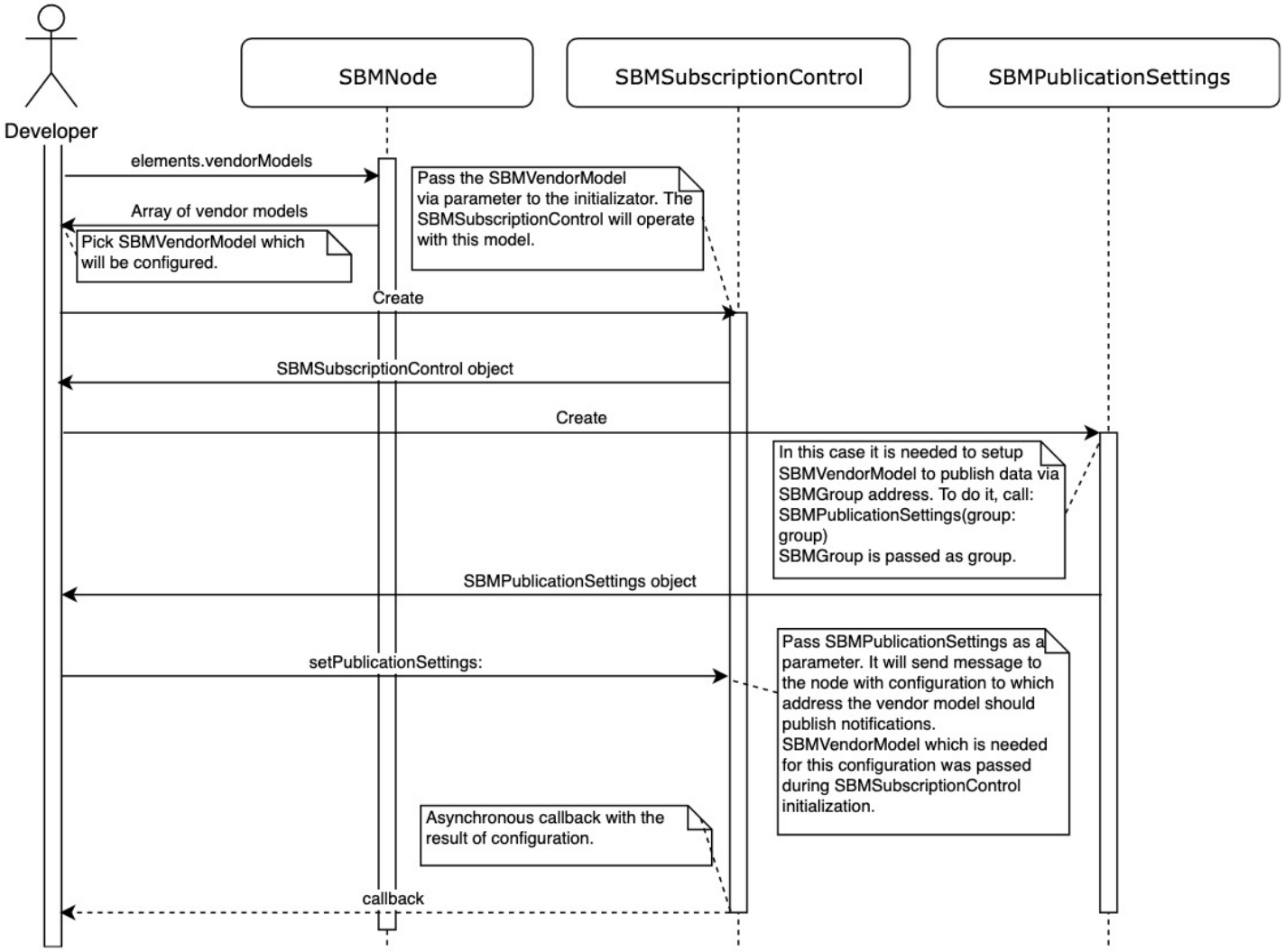
7.23.2.2 Send Publication Settings to the SBMVendorModel

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

7.23.2.2.1 Set the Provisioner Address as the Publication Address

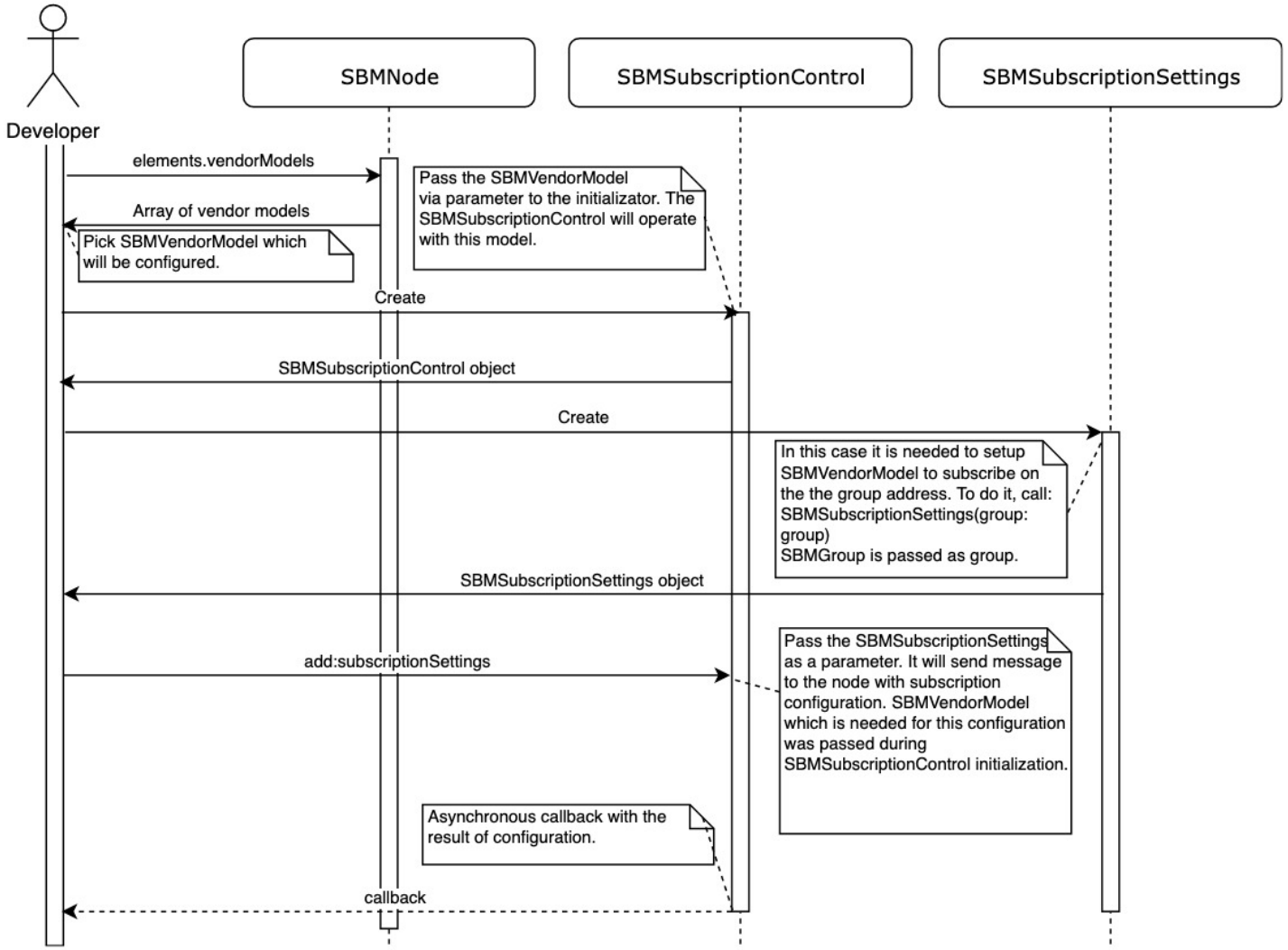


7.23.2.2.2 Set the SBMGroup address as the Publication Address



7.23.2.3 Send Subscription Settings to the SBMVendorModel

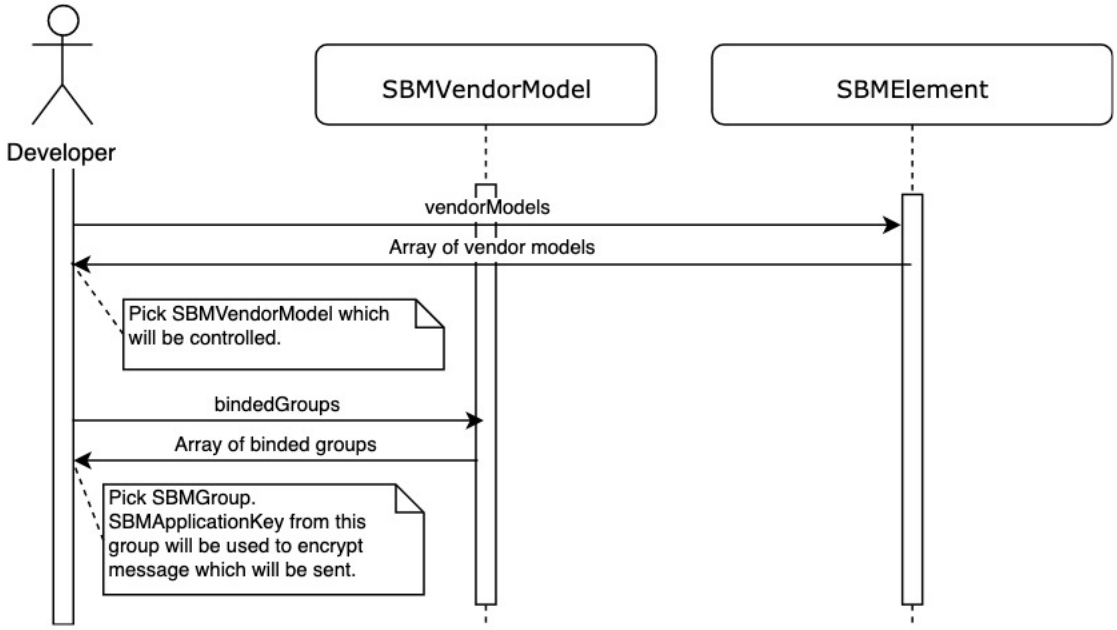
When the vendor model on the node expects a message from the group address, you must send the subscription settings containing the address of this group to the vendor model on the node. Without that the vendor model will not receive a message sent by the group address, because it is not observing this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

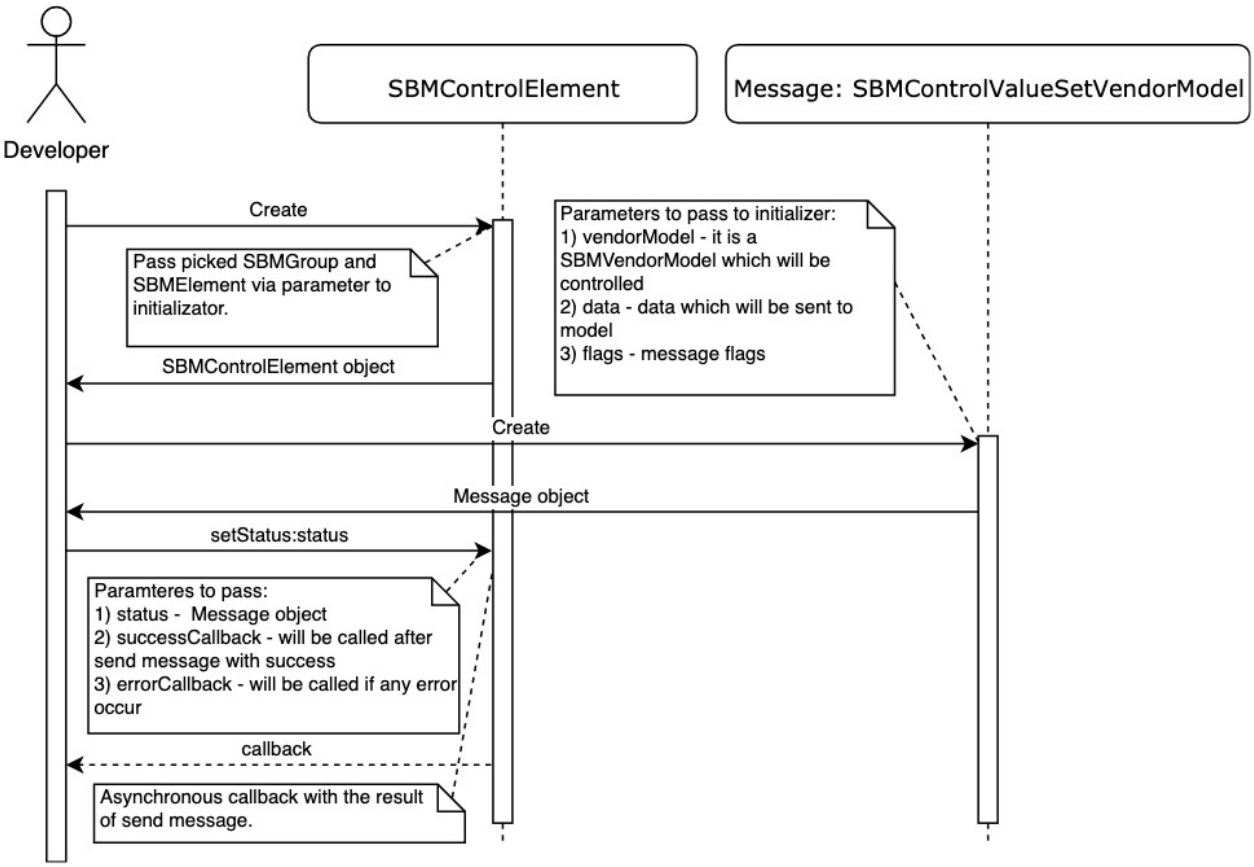


7.23.3 Send Message to the Vendor Model on the Node

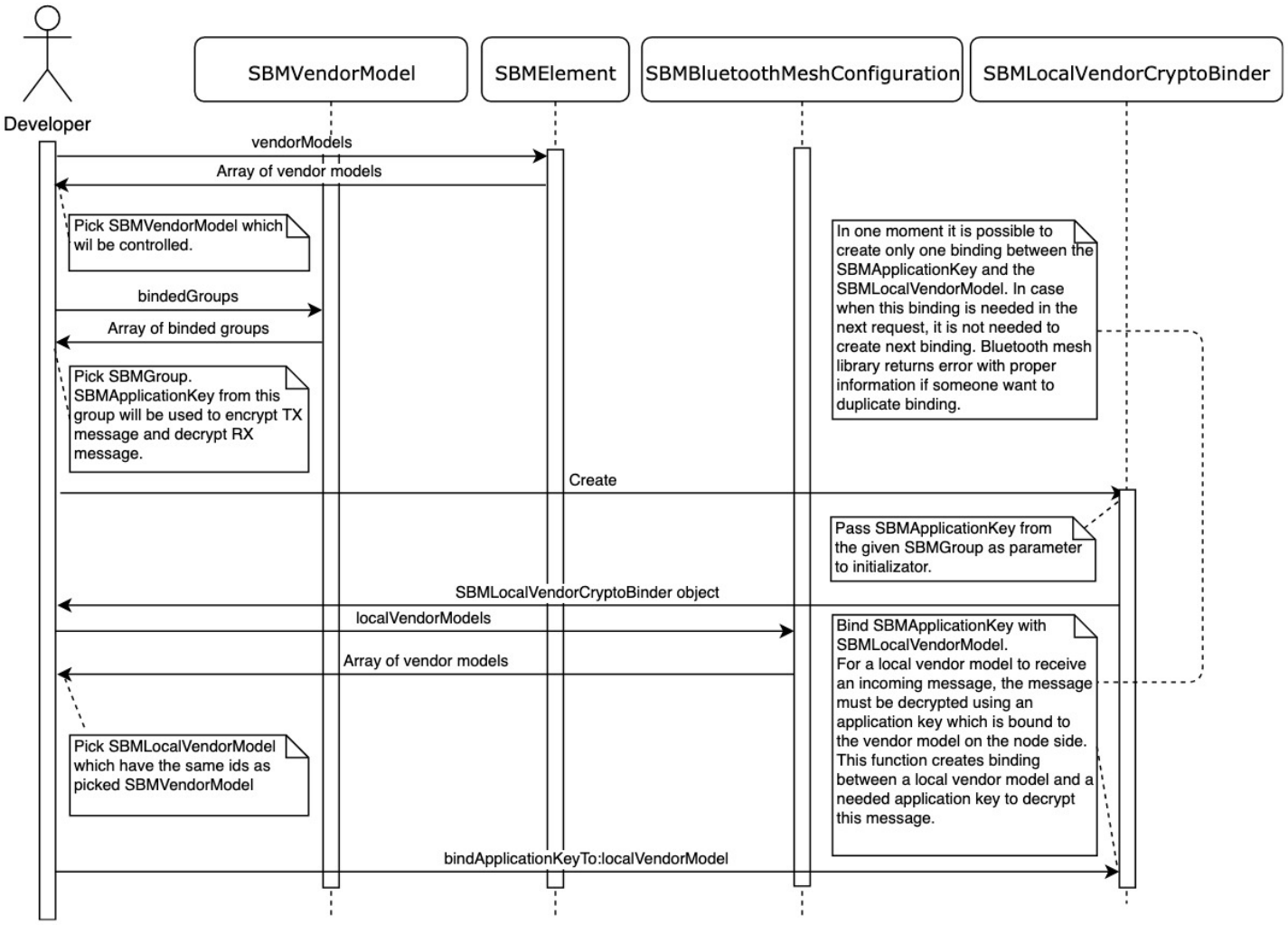
7.23.3.1 Send Message Directly to the Node

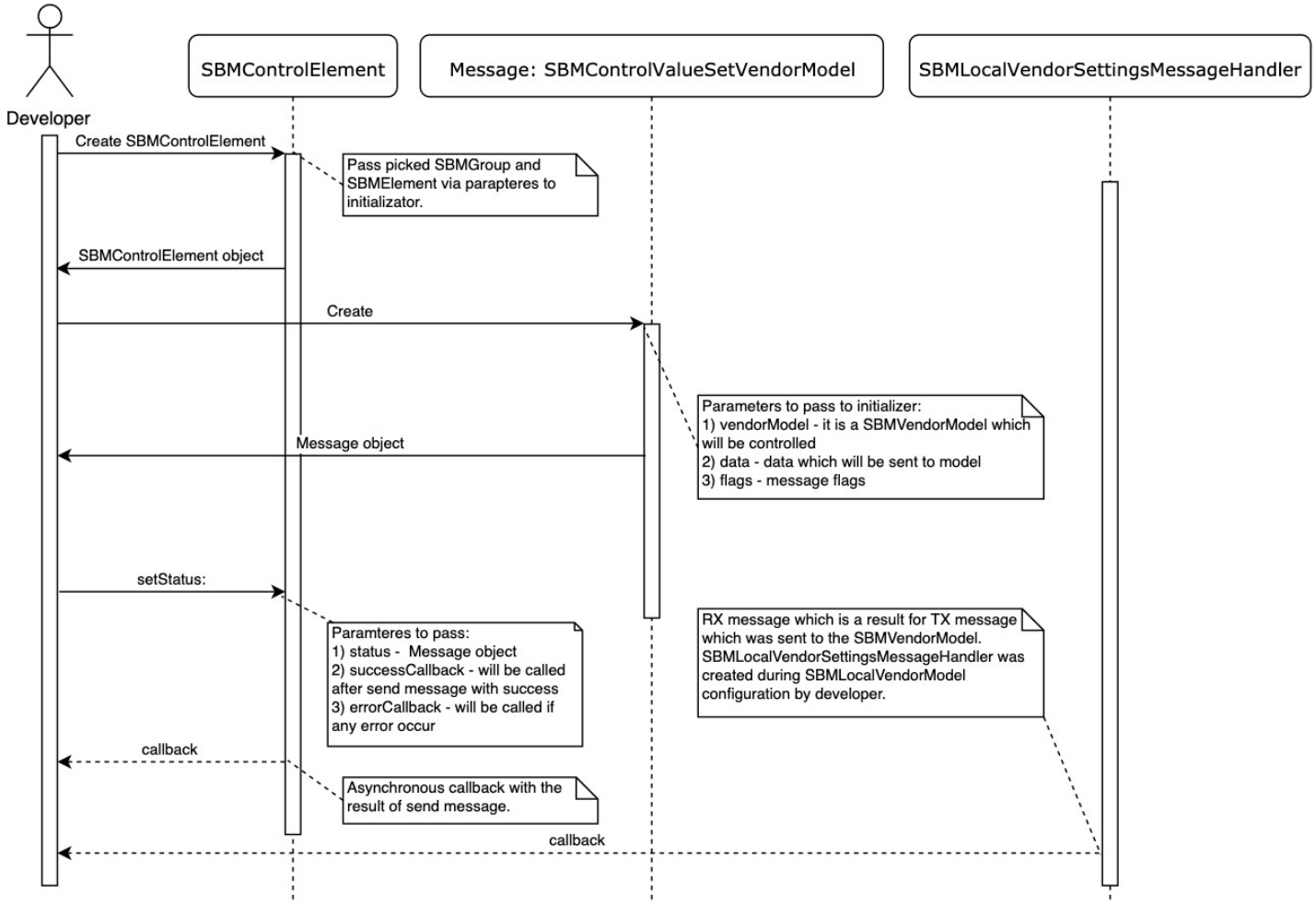
7.23.3.1.1 Message Without Response





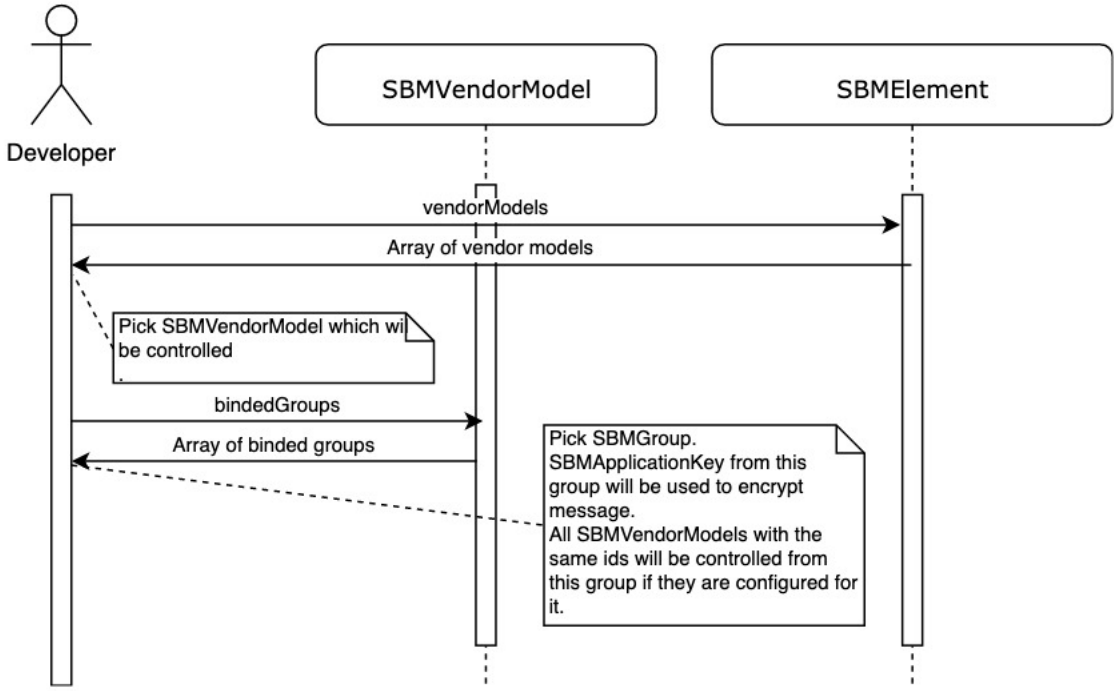
7.23.3.1.2 Message With Response

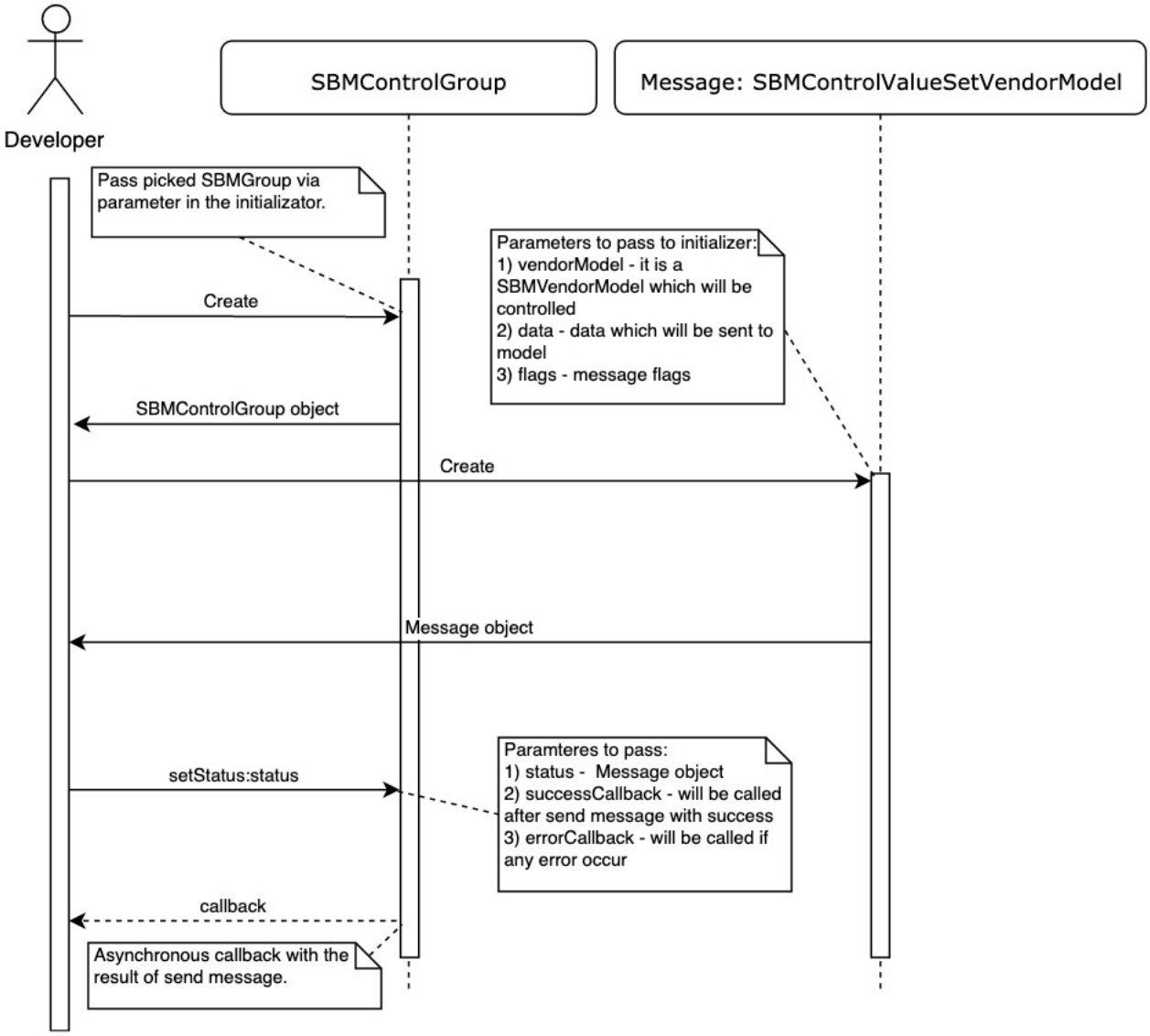



Developer
SBMControlElement
Message: SBMControlValueSetVendorModel
SBMLocalVendorSettingsMessageHandler
Create SBMControlElement
Pass picked SBMGroup and SBMElement via parapteres to initializator.
SBMControlElement object
Create
Parameters to pass to initializer:
1) vendorModel - it is a SBMVendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags
Message object
setStatus:
Paramteres 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
RX message which is a result for TX message which was sent to the SBMVendorModel. SBMLocalVendorSettingsMessageHandler was created during SBMLocalVendorModel configuration by developer.
callback
Asynchronous callback with the result of send message.
callback

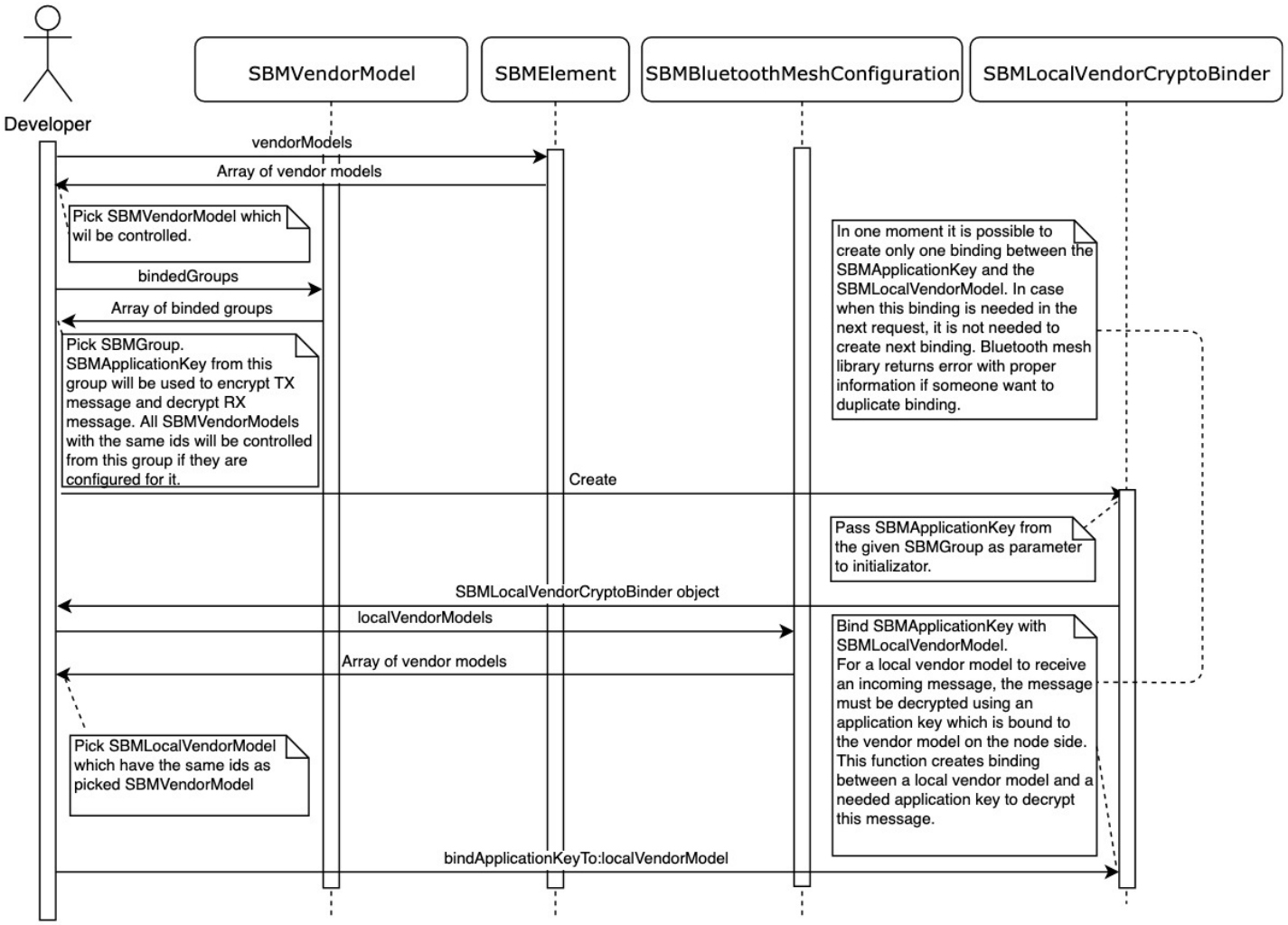
7.23.3.2 Send Message Via Group Address

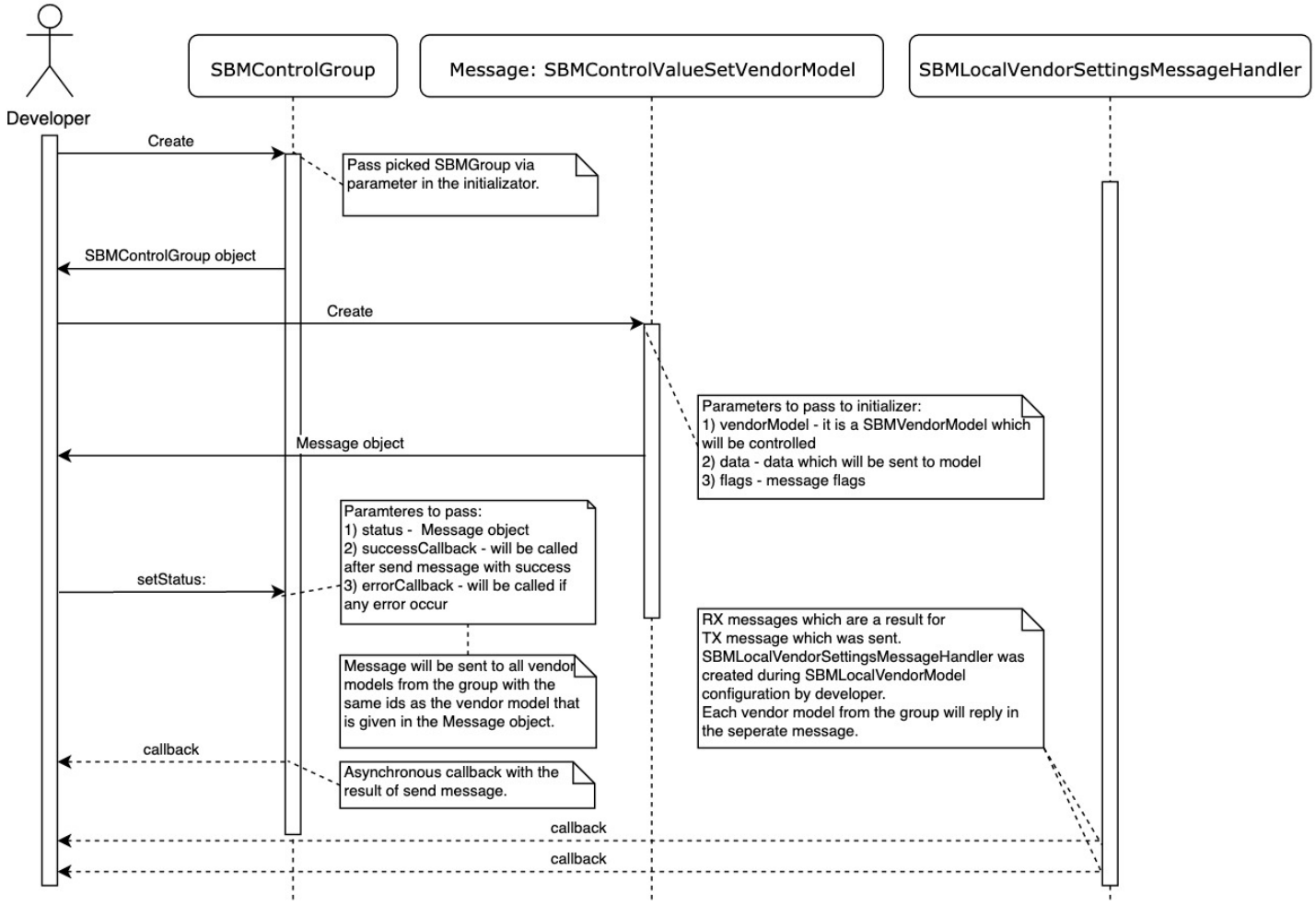
7.23.3.2.1 Message without Response




Developer
SBMControlGroup
Message: SBMControlValueSetVendorModel
Pass picked SBMGroup via parameter in the initializator.
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
Create
SBMControlGroup object
Create
Message object
Paramteres 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
setStatus:status
callback
Asynchronous callback with the result of send message.

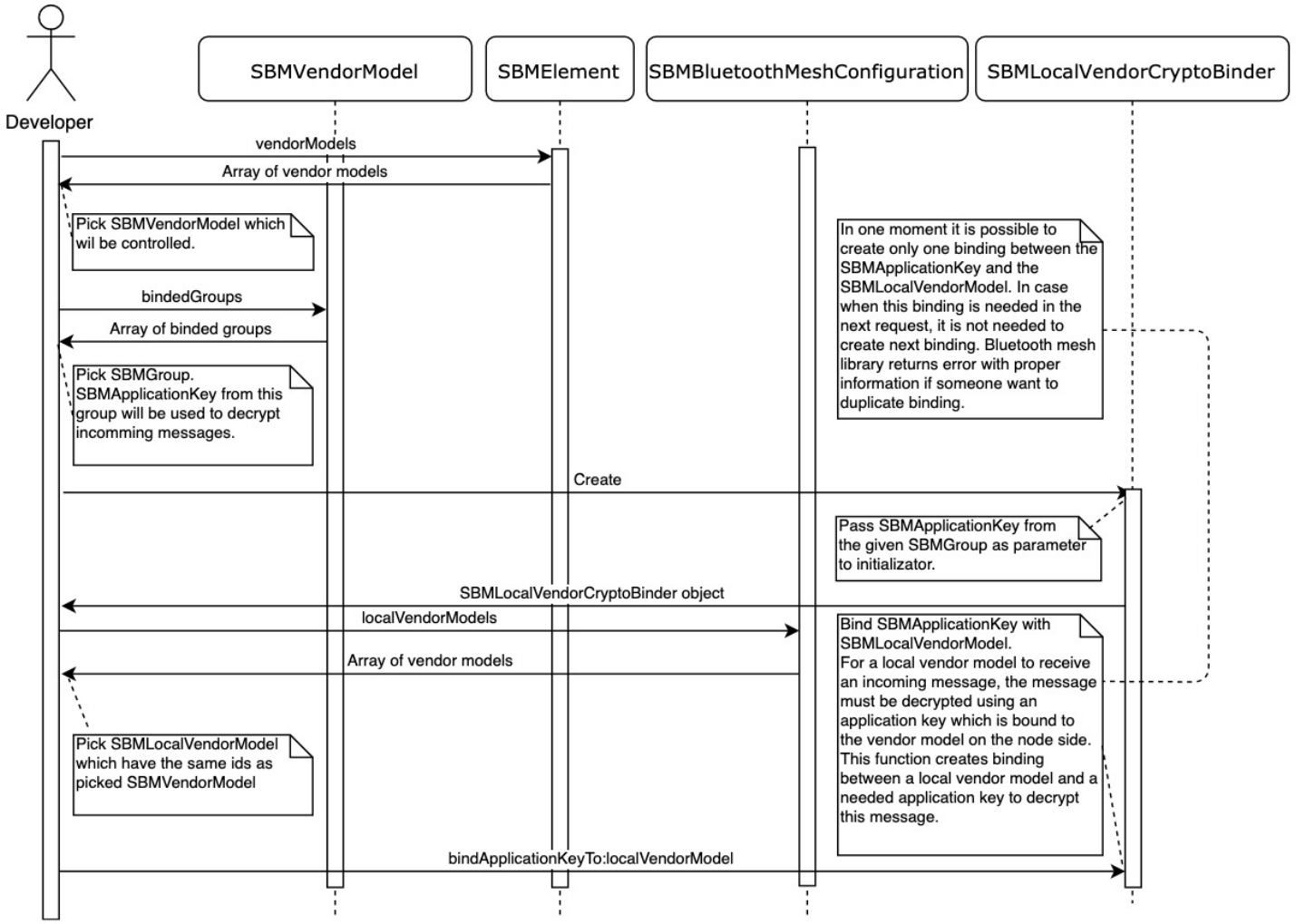
7.23.3.2.2 Message with Response

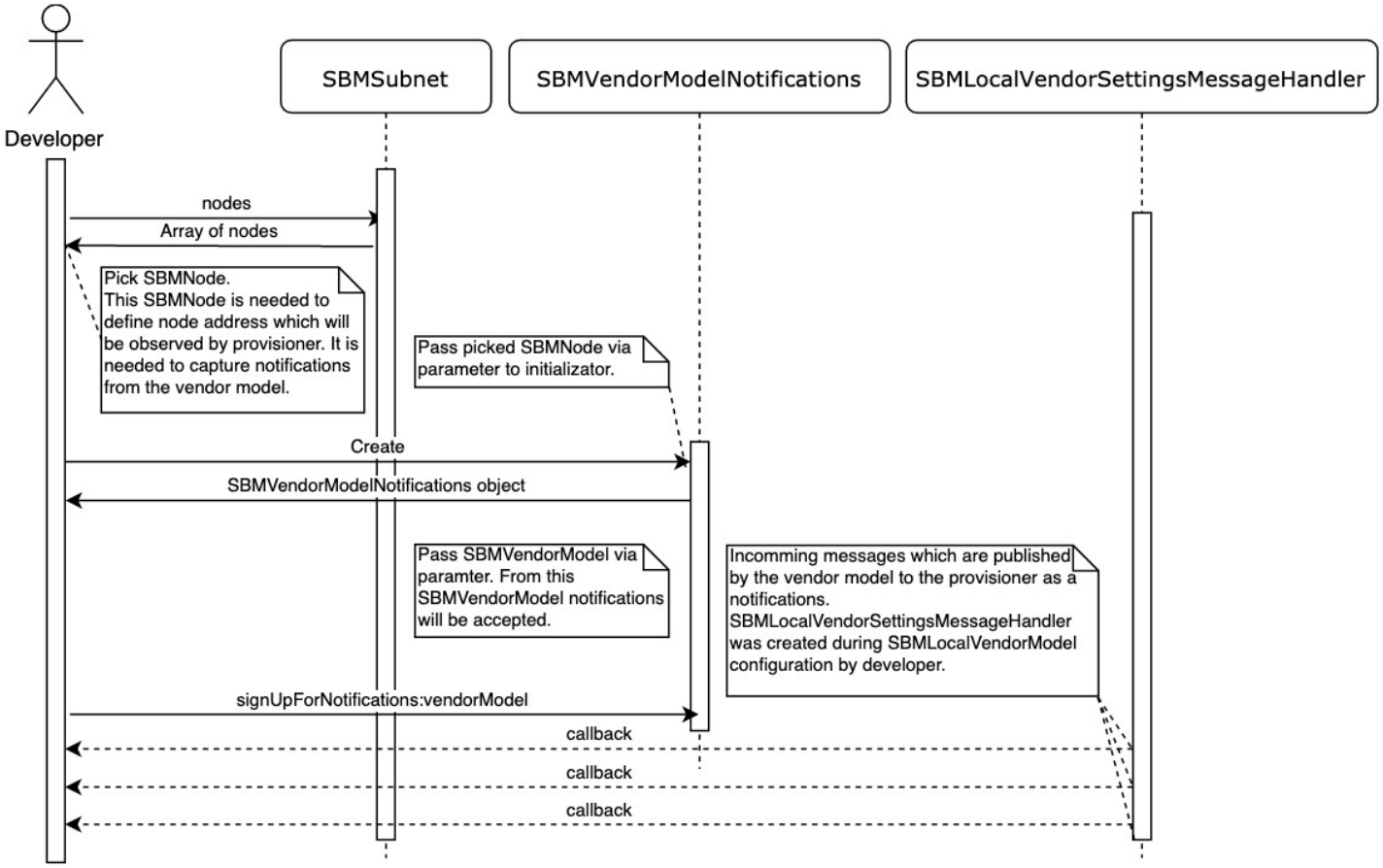



Developer
SBMControlGroup
Message: SBMControlValueSetVendorModel
SBMLocalVendorSettingsMessageHandler
Create
Pass picked SBMGroup via parameter in the initializator.
SBMControlGroup object
Create
Parameters to pass to initializer:
1) vendorModel - it is a SBMVendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags
Message object
Paramteres 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
setStatus:
RX messages which are a result for TX message which was sent. SBMLocalVendorSettingsMessageHandler was created during SBMLocalVendorModel configuration by developer. Each vendor model from the group will reply in the seperate message.
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.
callback
Asynchronous callback with the result of send message.
callback
callback

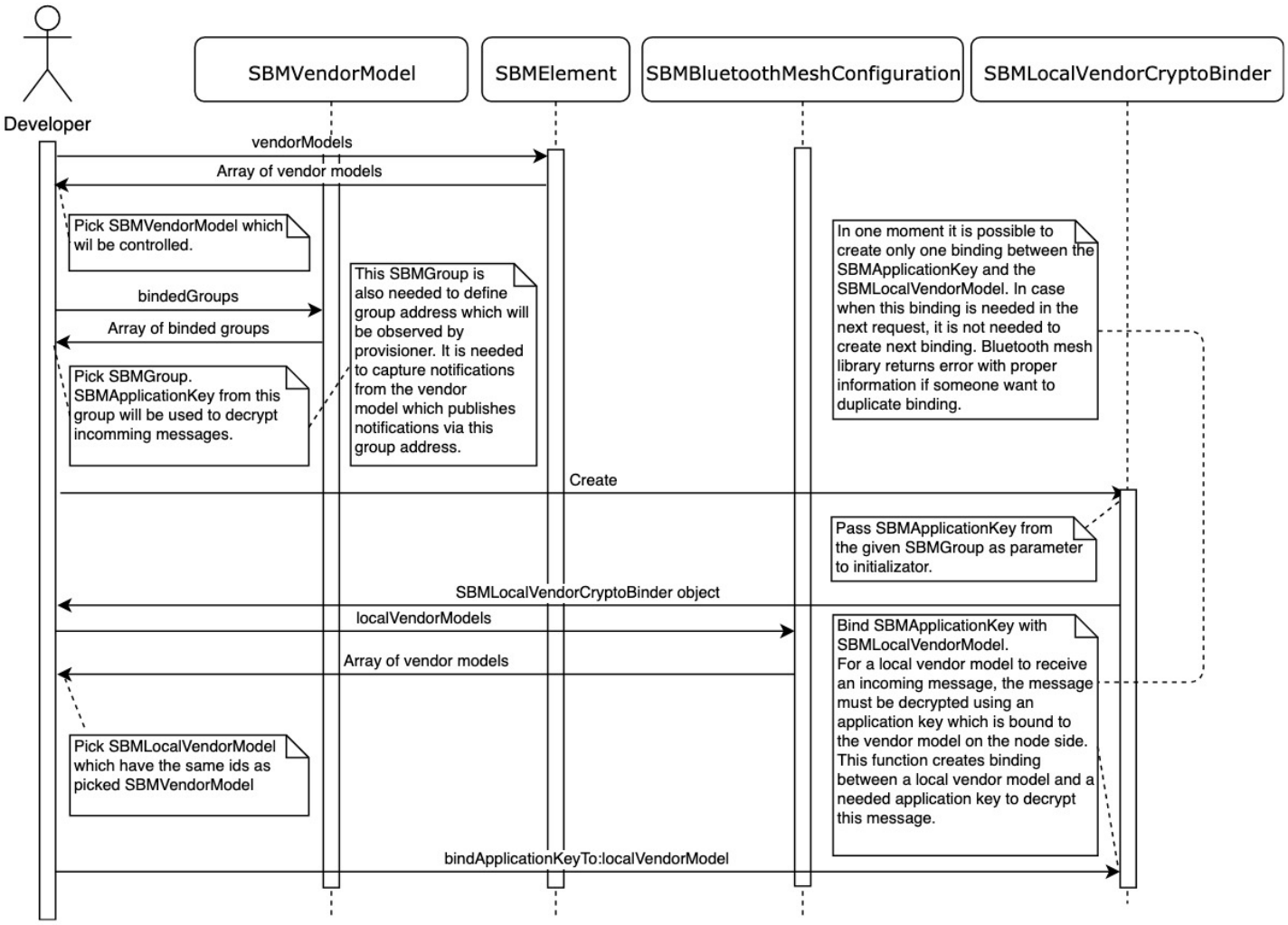
7.23.4 Subscribe to Publications Sent by Vendor Model from the Node

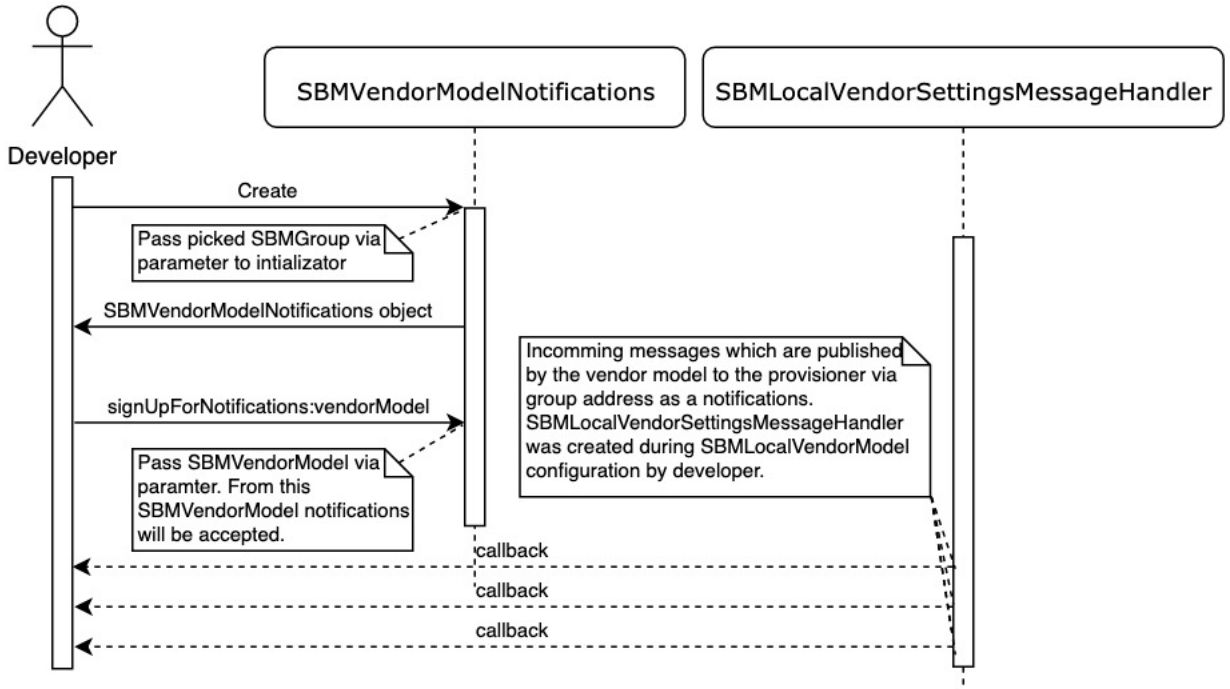
7.23.4.1 Messages Sent Directly to the Provisioner




Developer
SBMSubnet
SBMVendorModelNotifications
SBMLocalVendorSettingsMessageHandler
nodes
Array of nodes
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 initializator.
Create
SBMVendorModelNotifications object
Pass SBMVendorModel via paramter. From this SBMVendorModel notifications will be accepted.
Incomming messages which are published by the vendor model to the provisioner as a notifications.
SBMLocalVendorSettingsMessageHandler was created during SBMLocalVendorModel configuration by developer.
signUpForNotifications:vendorModel
callback
callback
callback

7.23.4.2 Messages Sent via Group Address to the Provisioner





7.24 Helpers Method

7.24.1 Check if Advertisement Data Match netKey

Class SBMConnectableDeviceHelper provides static method `doesNetKey:matchAdvertisingData:error:` which checks if received advertisement data match provided network key. It returns true only if Identification Type of Advertising is Network ID type (see section 7.2.2.2.1 Advertising from the *Mesh Profile Bluetooth® Specification*) and NetworkKey match Network ID in advertising data.

8 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:

- Bluetooth connection
- Provision session
- Proxy connection
- Node configuration
- Model configuration (subscription and publication settings)
- Control model and group

All are described in the next part of this document.

8.1 Errors

In case of operation failure, callbacks usually provide an ErrorType object with defined type and sometimes code and/or a message. All possible error types are defined by the ErrorType.TYPE enum. ErrorType.TYPE documentation has short descriptions of the situation in which the error can be encountered and sometimes possible solutions.

When the ErrorType object has the API_ERROR type, the `getErrorCode` method returns codes described in Bluetooth Mesh Software API Reference Manual section 2.41 Error codes.

8.2 Initializing the BluetoothMesh

8.2.1 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

```
BluetoothMesh.getInstance();

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

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

8.2.2 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.

```
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);
```

8.2.3 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.

```
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 32766
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 32
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 = 4;
groups = 8;
nodes = 255;
nodeNetworks = 4;
nodeGroups = 4;
rplSize = 32;
segmentedMessagesReceived = 4;
segmentedMessagesSent = 4;
provisionSessions = 1;

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

8.3 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.

8.3.1 Device Advertisement Data

Advertisement data can be obtained from `ScanResult`.

```
private var scanResult: ScanResult

// (…)

advertisementData = Objects.requireNonNull(scanResult.scanRecord).bytes

// (…)

override fun getAdvertisementData() = advertisementData
```

8.3.2 Device UUID

The device UUID comes from advertisement data.

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

```
// ConnectableDevice
public byte[] getUUID() {

    byte[] data = getServiceData(MESH_UNPROVISIONED_SERVICE);

    if (data != null && data.length >= 16) {
        byte[] uuid = new byte[16];
        System.arraycopy(data, 0, uuid, 0, uuid.length);

        return uuid;
    }

    return null;

}
```

8.3.3 Device Name

The device name comes from `BluetoothDevice`.

```
lateinit var bluetoothDevice: BluetoothDevice

// (…)

override fun getName(): String? {
    return bluetoothDevice.name
}
```

8.3.4 Device Connection State

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

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

8.3.5 Refresh BluetoothDevice

The Refresh BluetoothDevice method performs scanning to update a BluetoothDevice object.

Subsequent `discoverServices` method calls to the BluetoothGatt object result in cached services from the first call (even if the device was disconnected). This could cause problems because, after provisioning, device services are being changed. Refreshing BluetoothDevice object is a solution to obtain the current GATT services. This is required during the first connection and configuration after provisioning, before connecting to proxy node.

```
lateinit var bluetoothDevice: BluetoothDevice
private var scanResult: ScanResult
lateinit var address: String
private var bluetoothLeScanner: BluetoothLeScanner
private var refreshBluetoothDeviceCallback: RefreshBluetoothDeviceCallback? = null

// (…)

init {
    this.bluetoothDevice = scanResult.device
    this.address = bluetoothDevice.address
    this.scanResult = scanResult
}

// (…)

val scanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult?) {

        result?.let {
            if (it.device.address == address) {
                bluetoothLeScanner.stopScan(this)
                bluetoothDevice = result.device
                scanResult = result

                refreshBluetoothDeviceCallback?.success()
            }
        }
    }
}

override fun refreshBluetoothDevice(callback: RefreshBluetoothDeviceCallback) {
        refreshBluetoothDeviceCallback = callback
        val settings = ScanSettings.Builder()
                .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                .build()
        bluetoothLeScanner.startScan(null, settings, scanCallback)

}
```

8.3.6 Refresh Gatt Services

The Refresh Gatt Services method is used to refresh gatt services during a connection.

This method is required by one gatt connection feature which can be set using `ProvisionerConfiguration`. For more information about this feature see section 8.7.1 Create Network.

Please note that on some Android devices discovering services again may not be enough, because Android returns cached data. The solution to this issue is clearing the cache. This can be done by calling the refresh() method on the BluetoothGatt object using Java reflection.

```
lateinit var bluetoothGatt: BluetoothGatt
lateinit var bluetoothGattCallback: BluetoothGattCallback
var refreshGattServicesCallback: RefreshGattServicesCallback?

// (…)

bluetoothGattCallback = object : BluetoothGattCallback() {

// (…)

    override fun onServicesDiscovered (gatt: BluetoothGatt, status: Int) {
        super.onServicesDiscovered(gatt, status)
        if (status == BluetoothGatt.GATT_SUCCESS) {
           refreshGattServicesCallback?.onSuccess()
        } else {
           refreshGattServicesCallback?.onFail()
       }

    }

// (…)

}

fun refreshDeviceCache() {
    try {
        val refreshMethod: Method = bluetoothGatt.javaClass.getMethod("refresh")
        val result = refreshMethod.invoke(bluetoothGatt, *arrayOfNulls(0)) as? Boolean
        Log.d(TAG, "refreshDeviceCache $result")
    } catch (localException: Exception) {
        Log.e(TAG, "An exception occured while refreshing device")
    }
}

override fun refreshGattServices(refreshGattServicesCallback: RefreshGattServicesCallback) {
    refreshGattServicesCallback = callback

    refreshDeviceCache()
    bluetoothGatt.discoverServices()
}
```

8.3.7 Connect to the Device

```
lateinit var bluetoothDevice: BluetoothDevice
lateinit var bluetoothGatt: BluetoothGatt
private lateinit var bluetoothGattCallback: BluetoothGattCallback

// (…)

override fun connect() {
    bluetoothGatt = bluetoothDevice.connectGatt(context, false, bluetoothGattCallback,
BluetoothDevice.TRANSPORT_LE)
}
```

8.3.8 Disconnect from the Device

```
lateinit var bluetoothGatt: BluetoothGatt

// (…)

override fun disconnect() {
    if (this::bluetoothGatt.isInitialized) {
        bluetoothGatt.let {
            bluetoothGatt.close()
        }
    }
}
```

8.3.9 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.

Note that this can be done using two methods. `ScanRecord.getServiceUuids()` or `BluetoothGatt.getServices()`. Both methods should be implemented to return current state all the time. ScanRecord is used before connection is established, BluetoothGatt when services are discovered.

```
private var scanResult: ScanResult
private var bluetoothGatt: BluetoothGatt

// (…)

override fun hasService(service: UUID?): Boolean {
        if (bluetoothGatt.services.isNotEmpty()) {
            return bluetoothGatt.getService(service) != null
        } else {
            return scanResult.scanRecord?.serviceUuids?.contains(ParcelUuid(service))
                    ?: return false
        }
}
```

8.3.10 Maximum Transmission Unit for Given Device Service

```
private var mtuSize = 0
private lateinit var bluetoothGattCallback: BluetoothGattCallback

// (…)

bluetoothGattCallback = object : BluetoothGattCallback() {

// (…)

    override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
        super.onMtuChanged(gatt, mtu, status)
        if (status == BluetoothGatt.GATT_SUCCESS) {
            mtuSize = mtu
            gatt.discoverServices()
        }
    }

// (…)

}

// (…)

override fun getMTU(): Int {
    return mtuSize
}
```

8.3.11 Write Data to a Given Service and Characteristic

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

```
lateinit var bluetoothGatt: BluetoothGatt

// (…)

override fun writeData(service: UUID?, characteristic: UUID?, data: ByteArray?,
connectableDeviceWriteCallback: ConnectableDeviceWriteCallback) {
    try {
        val bluetoothGattCharacteristic =
bluetoothGatt.getService(service)!!.getCharacteristic(characteristic)
        bluetoothGattCharacteristic.value = data
        bluetoothGattCharacteristic.writeType = BluetoothGattCharacteristic.WRITE_TYPE_NO_RESPONSE
        if (!bluetoothGatt.writeCharacteristic(bluetoothGattCharacteristic)) {
            throw Exception("Writing to characteristic failed")
        }
        connectableDeviceWriteCallback.onWrite(service, characteristic)
    } catch (e: Exception) {
        Log.e(TAG, "writeData error: ${e.message}")
        connectableDeviceWriteCallback.onFailed(service, characteristic)
    }
}
```

8.3.12 Subscribe to a Given Service and Characteristic

```
lateinit var bluetoothGatt: BluetoothGatt

// (…)

override fun subscribe(service: UUID?, characteristic: UUID?,
connectableDeviceSubscriptionCallback: ConnectableDeviceSubscriptionCallback) {
    try {
        val bluetoothGattCharacteristic =
bluetoothGatt.getService(service)!!.getCharacteristic(characteristic)
        if (!bluetoothGatt.setCharacteristicNotification(bluetoothGattCharacteristic, true)) {
            throw Exception("Enabling characteristic notification failed")
        }
        val bluetoothGattDescriptor = bluetoothGattCharacteristic.descriptors.takeIf { it.size == 1
}?.first() ?: throw Exception("Descriptors size (${bluetoothGattCharacteristic.descriptors.size})
different than expected: 1")
        bluetoothGattDescriptor.apply { value = BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE }
        if (!bluetoothGatt.writeDescriptor(bluetoothGattDescriptor)) {
            throw Exception("Writing to descriptor failed")
        }
        connectableDeviceSubscriptionCallback.onSuccess(service, characteristic)
    } catch (e: Exception) {
        Log.e(TAG, "subscribe error: ${e.message}")
        connectableDeviceSubscriptionCallback.onFail(service, characteristic)
    }
}
```

8.3.13 Unsubscribe from a Given Service and Characteristic

This method is required by one gatt connection feature which can be set using `ProvisionerConfiguration`. For more information about this feature see section 8.7.1 Create Network.

```
lateinit var bluetoothGatt: BluetoothGatt

// (…)

override fun unsubscribe(service: UUID?, characteristic: UUID?,
connectableDeviceUnsubscriptionCallback: ConnectableDeviceUnsubscriptionCallback) {
    try {
        val bluetoothGattCharacteristic =
bluetoothGatt.getService(service)!!.getCharacteristic(characteristic)
        if (!bluetoothGatt.setCharacteristicNotification(bluetoothGattCharacteristic, false)) {
            throw Exception("Disabling characteristic notification failed")
        }
        val bluetoothGattDescriptor = bluetoothGattCharacteristic.descriptors.takeIf { it.size == 1
}?.first() ?: throw Exception("Descriptors size (${bluetoothGattCharacteristic.descriptors.size})
different than expected: 1")
        bluetoothGattDescriptor.apply { value = BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE
}
        if (!bluetoothGatt.writeDescriptor(bluetoothGattDescriptor)) {
            throw Exception("Writing to descriptor failed")
        }
        connectableDeviceUnsubscriptionCallback.onSuccess(service, characteristic)
    } catch (e: Exception) {
       Log.e(TAG, "unsubscribe error: ${e.message}")
        connectableDeviceUnsubscriptionCallback.onFail(service, characteristic)
    }
}
```

8.4 IV Update

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

```
IvIndexControl ivIndexControl = new IvIndexControl();
int currentIvIndex = ivIndexControl.getValue();
```

An IV Index update can be requested by calling:

```
ivIndexControl.requestUpdate();
```

Note that the request may fail for the following reasons:

- The IV Update is already ongoing
- Not enough time has passed since the previous update
- A node is not a member of the primary subnet
- A node is not sending Secure Network beacons

To receive information about the IV Update procedure status changes, subscribe to the event with:

```
ivIndexControl.setUpdateHandler(new IvUpdateHandler() {
    @Override
    public void handle(int ivIndex, IvUpdateState transition) {
        // here define event handler
   }
});
```

8.4.1 Test IV Update

Since the 96-hour limit for changing the IV Update procedure state might be a problem when testing the IV Update, there is a test mode to remove this limit. It can be enabled with:

```
ivIndexControl.setUpdateTestModeEnabled(true);
```

8.4.2 IV Index Recovery

When device has not been connected to the network for some time, it may have missed IV Index updates. The IV Index recovery procedure can be used to fix this issue. It can be triggered with:

```
ivIndexControl.setRecoveryModeEnabled(true);
```

By default, the ADK will initiate recovery procedure if it detects that recovery is needed. Recovery will be performed after a new Secure Network Beacon is received (for example after reconnecting with a proxy node). Behavior on a recovery-needed event can be changed using the IvIndexControl method.

8.5 Get Secure Network Beacon Information

Secure Network Beacon information can be obtained after connecting to the proxy service of any node from a known subnet. To observe this for a secure network beacon, call method `observeSecureNetworkBeacon` before connecting to the proxy. The object is assumed to conform to ConnectableDevice. For example:

```
ConnectableDevice connectableDevice;
ConnectionCallback connectionCallback;

ProxyConnection proxyConnection = new ProxyConnection(connectableDevice)
SecureNetworkBeaconCallback callback = new SecureNetworkBeaconCallback(){
       @Override
       void success(int netKeyIndex, boolean keyRefresh, boolean ivUpdate, int ivIndex){
```

```
            //Handle callback here
        }
};
proxyConnection.observeSecureNetworkBeacon(callback);
proxyConnection.connectToProxy(connectionCallback);
```

For more information about Secure Network Beacon see section 3.9.3 Secure Network Beacon from the *Mesh Profile Bluetooth® Specification*.

8.6 Server Configuration

8.6.1 Time to Live

The TTL value is the maximum number of hops the packet takes along the path to its destination. The default value is 5.

Please note that the value 0 can only be used when communicating directly with a proxy node, otherwise the message will be lost.

Lowering this value may improve network performance during the node configuration process, since the number of messages in the network will be decreased. If such a change has been made, it is important to restore TTL to the higher value after configuration is done to ensure proper network operation.

```
ServerConfigurationControl serverConfigurationControl = new ServerConfigurationControl();
serverConfigurationControl.setTTL(ttl);
```

8.7 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.

8.7.1 Create Network

```
Network network = BluetoothMesh.getInstance().createNetworkWithoutSubnet("Network name")
```

Currently more than one Network instance can be created, but it is not recommended because of the known issue described in section 13.1 Concept.

8.7.2 Create Subnet

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

8.7.3 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, 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);
```

8.7.4 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.

8.7.4.1 In-Band Provisioning

```
ConnectableDevice connectableDevice;
Subnet subnet;

// (…)

ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);

// (…)

// provisionerConfiguration: specifies whether proxy and nodeIdentity are enabled
// provisionerConfiguration can be set to null if provisioning should not do any setup
ProvisionerConfiguration provisionerConfiguration = new ProvisionerConfiguration();
provisionerConfiguration.setGettingDeviceCompositionData(true);
provisionerConfiguration.setEnablingProxy(true);
provisionerConfiguration.setEnablingNodeIdentity(true);
// nodeProperties: specifies node properties, including attention timer to use
// nodeProperties can be null to provision without attention timer and specific address
int address = 5;
int elements = 2;
int attentionTimer = 1;
NodeProperties nodeProperties = new NodeProperties(address, elements, attentionTimer);
provisionerConnection.provision(provisionerConfiguration , nodeProperties,
new ProvisioningCallback() {
    @Override
    public void success(ConnectableDevice device, Subnet subnet, Node node) {

    }

    @Override
    public void error(ConnectableDevice device, Subnet subnet, ErrorType error) {

    }
});
```

8.7.4.2 OOB Provisioning

```
ConnectableDevice connectableDevice;
Subnet subnet;
ProvisionerOOBImpl provisionerOOB; // ProvisionerOOBImpl extends ProvisionerOOBControl abstract
class.

// (…)

ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);
provisionerConnection.setProvisionerOOB(provisionerOOB);

// (…)

provisionerConnection.provision(null, null, new ProvisioningCallback() {
    @Override
    public void success(ConnectableDevice device, Subnet subnet, Node node) {

    }

    @Override
    public void error(ConnectableDevice device, Subnet subnet, ErrorType error) {

    }
});
```

8.7.4.2.1 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_OBB);
        authMethodsAllowed.add(AUTH_METHODS_ALLOWED.OUTPUT_OOB);
        authMethodsAllowed.add(AUTH_METHODS_ALLOWED.STATIC_OBB);
        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.

8.7.5 Possible Provisioning Errors

Provisioning may be unsuccessful at different stages of the process, so it would be helpful to determine if it ended with a device provisioned or not (for example, due to configuration errors after provisioning). The ProvisionerConnection class contains a field of type Node that is initialized as null. Its state changes after a device has become provisioned. A simple null check will provide information on what happened during the process. If Node object is null, it means that the device is still unprovisioned, otherwise its state has changed and it is possible to establish a connection again.

8.8 Node configuration

A node can be configured immediately after it has been provisioned. The behavior depends on the first argument of `ProvisionerConnection.provision(..)` method. It is expected to be an object of type ProvisionerConfiguration, which decides what parts of the configuration process should be executed.

Null may be passed as an argument, meaning that a device should only be provisioned. Otherwise a node configuration will be performed as well. If all the configuration tasks will be chosen, the following operations will happen in the order given:

1. Connecting to proxy
2. Getting device composition data
3. Setting up proxy
4. Setting up node identity
5. Disconnecting from proxy

Tasks 1 and 5 are performed by the default when ProvisionerConfiguration is not null while tasks 2-4 can be added to the process by toggling ProvisionerConfiguration fields on. In addition, task 5 can be removed if the user wants to go on with the node. The following example shows how to provision a node and configure it within one user operation.

```
ConnectableDevice connectableDevice;
Subnet subnet;
...
ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);
```

```
ProvisionerConfiguration provisionerConfiguration = new ProvisionerConfiguration();
provisionerConfiguration.setGettingDeviceCompositionData(true);
provisionerConfiguration.setEnablingProxy(true);
provisionerConfiguration.setEnablingNodeIdentity(true);
...
provisionerConnection.provision(provisionerConfiguration, null, new ProvisioningCallback() {
       @Override
       public void success(ConnectableDevice device, Subnet subnet, Node node) {
              //success
       }

       @Override
       public void error(ConnectableDevice device, Subnet subnet, ErrorType error) {
              //error
       }
});
```

8.8.1 One GATT Connection for Provisioning and Configuration

After the device is successfully provisioned, the Bluetooth GATT connection is closed and reopened again to start the configuration process. It is possible to use the same Bluetooth GATT connection for the configuration process. To use this feature toggle on the useOneGattConnection field in the ProvisionerConfiguration object.

A few changes are needed to use this feature. The ConnectableDevice implementation should have implemented methods:

3. Unsubscribe (see section 8.3.13 Unsubscribe from a Given Service and Characteristic for more details)
4. refreshGattServices (see section 8.3.6 Refresh Gatt Services for more details)

A change in the method `hasService(...)` in the ConnectableDevice implementation is also needed. It must return the current status of the GATT Services.This means that when services are discovered use `BluetoothGatt.getServices()` instead of `ScanRecord.getServiceUuids()`.

8.9 Add a Node to a 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. A node can be added 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) {

    }
});
```

8.10 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.removeFrom(subnet, new NodeControlCallback() {
    @Override
    public void succeed() {

    }

    @Override
    public void error(ErrorType errorType) {

    }
});
```

8.11 Removing a Subnet

To remove a subnet the Subnet#removeSubnet method can be used. This method sends a factory reset message to every node in the subnet that is only in this subnet, with the currently connected proxy node being the last to which the message is sent. However, this method is very simplistic and does not allow dealing with some problems that may occur during subnet removal. Since some messages can be lost, the subnet may end up in a state where some nodes were not reset and can no longer be reached, or all nodes have successfully been reset but the confirmation message was not received.

If more reliability is required, it is better to deal with each node individually. If a node is only in the subnet that is to be removed, a factory reset message can be sent to remove it from subnet. If the node is also in another subnet, it can be removed from the subnet that is to be deleted by using the NodeControl#removeFrom method.

After removing all nodes, call Subnet#removeSubnet to remove only the subnet from the database, since it is empty already.

8.12 Factory Reset a Node

To factory reset a node, use ConfigurationControl#factoryReset. FactoryResetCallback#success should be received on a successful factory reset, but it is not 100% reliable.

If the message has been successfully sent, it may still be lost in a network. Also, a response message from the node may be lost or the node may not send it at all. In such cases, FactoryResetCallback#error will be called with a timeout error.

When FactoryResetCallback#error is called, we do not know if the node has successfully performed a factory reset. To confirm it, send an acknowledged message which the node would normally handle (other than a factory reset message). If the node has performed a factory reset, the message will result in a timeout error, otherwise the node will respond. If a factory reset has not been performed, send a factory reset message again. If the node has already performed a factory reset, use the Node#removeOnlyFromLocalStructure method to remove the node from the local database.

Furthermore, The node may have performed a factory reset due to other reasons, for example someone manually performed it on a device, without using the Mesh messages. In this case, the node will not respond to any messages and they will end with timeout errors., Use Node#removeOnlyFromLocalStructure To remove such a node from the local database.

8.13 Configure Node Default TTL

To get the node default TTL value, use the *ConfigurationControl#getDefaultTtl* method. *DefaultTtlCallback#success* should be received on successfully getting the node default TTL value. Otherwise *DefaultTtlCallback#error* will be called.

To set the node default TTL value, use the *ConfigurationControl#setDefaultTtl* method with the parameter *defaultTtl*. *DefaultTtl-Callback#success* should be received on successfully setting the node default TTL value. Otherwise *DefaultTtlCallback#error* will be called.

8.14 Connect with a Subnet

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

8.14.1 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, use a class that extends 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 = ProxyConnection.MESH_PROXY_SERVICE;

ScanFilter filter = new ScanFilter.Builder().setServiceUuid(new ParcelUuid(uuid)).build();

BluetoothLeScanner bluetoothLeScanner;
bluetoothLeScanner.startScan(Collections.singletonList(filter), settings, scanCallback);
```

8.14.2 Get Node Representing a Given Device

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

```
ConnectableDevice connectableDevice:
ConnectableDeviceHelper connectableDeviceHelper;

//(…)

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

8.15 Create a Group in a Given Subnet

```
Subnet subnet;

// (…)

// appKey - Application key of the group. Null generates a random key.
// key - A 16-bytes array used for Key generation.
// index - Application Key index.
// subnet - Subnet which should be bound to Application Key.
```

```
AppKey appKey = AppKey.createKey(key, index, subnet);
// address - Address of the group. It must be an integer between 0xC000 and 0xFEFF. Null assigns
next available address.
int address = 0xFF00

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

8.16 Remove Group

Note: To perform this action you must have an established a 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, GroupRemovalErrorResult result, ErrorType errorType) {

        }
    });
```

8.17 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) {

    }
});
```

8.18 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) {

    }
});
}
```

8.19 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) {

    }
});
```

8.20 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) {

    }
});
```

8.21 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.

8.21.1 SIG model

```
SigModel sigModel;
Group group;

//(…)

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

subscriptionControl.addSubscriptionSettings(subscriptionSettings, new
SubscriptionSettingsGenericCallback() {
    @Override
    public void success(Model meshModel, SubscriptionSettings subscriptionSettings) {

    }

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

    }
});
```

8.21.2 Vendor Model

```
VendorModel vendorModel;
Group group;

//(…)

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

subscriptionControl.addSubscriptionSettings(subscriptionSettings, new
SubscriptionSettingsGenericCallback() {
    @Override
    public void success(Model meshModel, SubscriptionSettings subscriptionSettings) {

    }

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

    }
});
```

8.22 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.

8.22.1 SIG Model

8.22.1.1 Publish via Group Address

```
SigModel sigModel;
Group group;

//(…)

PublicationSettings publicationSettings = new PublicationSettings(group);
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) {

    }
});
```

8.22.1.2 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) {
    }
});
```

8.22.2 Vendor Model

8.22.2.1 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) {
    }
});
```

8.22.2.2 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 8.23.9 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) {
    }
});
```

8.23 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.

8.23.1 Get Value for a Model

8.23.1.1 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 value) {

        }

        @Override
        public void error(Element element, Group group, ErrorType errorType) {

        }
    });
}
```

8.23.1.2 Get Value for All Specific Models Bound with a Group

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

The Success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
GenericLevel genericLevel;

//(…)

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

    MeshTask task  = controlGroup.
        getStatus(genericLevel, new GenericGroupHandler<GenericLevel>() {
               @Override
               public void success(Element element, Group group, GenericLevel value) {

               }

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

               }
    });
}
```

8.23.2 Set Value for a Model

8.23.2.1 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(level);

    ControlRequestParameters parameters = new ControlRequestParameters(1, 1, false, 0);

    controlElement.setStatus(status, parameters, new SetElementStatusCallback<GenericLevel>() {
            @Override
            public void success(Element element, Group group, GenericLevel value) {

            }

            @Override
            public void error(Element element, Group group, ErrorType errorType) {

            }
        }
    );
}
```

8.23.2.2 Set Value for All Specific Models Bound with the Group

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

The Success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
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, 0);

    MeshTask task = controlGroup.
              setStatus(status, parameters, new GenericGroupHandler<GenericLevel>() {
                   @Override
                   public void success(Element element, Group group, GenericLevel value) {

                   }

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

                   }
         }
    );
}
```

8.23.2.3 Set Request Parameters

Two constructors for Set request parameters replace the old one. Deprecated constructor:

```
ControlRequestParameters(int transitionTime, int delayTime, boolean requestReply)
```

does not allow setting a custom transaction ID, which makes it impossible to use retransmission.

Note: Do NOT use the deprecated constructor when the new one is also used. The library is not adapted to using these two constructors alternately during the lifetime of an application.

Transaction time, delay time, and transaction ID are not available for each Set request. Refer to the following table to see which constructor of `ControlRequestParameters` should be used for a specific Set message.

`ControlRequestParameters(int transitionTime, int delayTime, boolean requestReply, int transactionId)`	`ControlRequestParameters(boolean requestReply)`
Generic OnOff, Generic Level, Generic Delta, Generic Move, Generic Power Level, Light Lightness, Light Lightness Linear, Light CTL, Light CTL Temperature	Generic Default Transition Time, Generic OnPowerUp, Generic Power Default, Generic Power Range, Generic Location Global, Generic Location Local, Generic User Property, Generic Admin Property, Generic Manufacturer Property, Light Lightness Default, Light Lightness Range, Light CTL Temperature Range, Light CTL Default

transitionTime - Transition time in milliseconds, or zero for an immediate change request.

delayTime - Delay in milliseconds before the server acts on the request, or zero for immediate action.

requestReply - A Boolean value that determines whether an explicit response (status message) is required.

transactionId - A transaction identifier indicating whether the message is a new message or a retransmission of a previously sent message. The Bluetooth Mesh library user is responsible for properly handling the transaction id.

To retransmit the message, use the same value for the transactionId field as in the previously sent message, within 6 seconds from sending that message.

8.23.3 Control Sensor Models

Even though 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 8.21.1 SIG model and 8.22.1 SIG Model, respectively).

To set up Sensor model bindings use the API prepared for SIG models (see sections 8.19 Bind a Model with a Group and 8.20 Unbind a Model from a Group).

8.23.3.1 Get Sensor Model Values

8.23.3.1.1 Get Sensor Descriptors from the Node

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);

        SensorPropertiesGet properties = SensorPropertiesGet.
              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();
```

8.23.3.1.2 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*).

```
void getSensorStatus(Element element, Group group, int propertyId) {
       ControlElement controlElement = new ControlElement(element, group);

       SensorPropertiesGet properties = SensorPropertiesGet.
              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
            }
        });
    }
```

8.23.3.1.3 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*).

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

        SensorPropertiesGet properties = SensorPropertiesGet.
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
            }
        });
}
```

8.23.3.1.4 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);

       SensorPropertiesGet properties = SensorPropertiesGet.
              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
            }
       });
}
```

8.23.3.1.5 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);

        SensorPropertiesGet properties = SensorPropertiesGet.
              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
            }
        });
    }
```

8.23.3.1.6 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);

        SensorPropertiesGet properties = SensorPropertiesGet.
             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
            }
        });
    }
```

8.23.3.1.7 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);

        SensorPropertiesGet properties = SensorPropertiesGet.
             SensorPropertiesGetForSensorSettingStatus(propertyId, settingId);

        controlElement.getSensorStatus(new SensorSettingStatus(), properties,
         new GetElementSensorStatusCallback<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
            }
        });
    }
```

8.23.3.1.8 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.

GetGroupSensorStatusHandler - 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 getSensorDescriptors(Group group, int propertyId) {
        ControlGroup controlGroup = new ControlGroup (group);

        SensorPropertiesGet properties = SensorPropertiesGet.
             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();
```

8.23.3.1.9 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*).

GetGroupSensorStatusHandler - 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);

        SensorPropertiesGet properties = SensorPropertiesGet.
             SensorPropertiesGetForSensorStatus(propertyId);

        MeshTask meshTask = controlGroup.getSensorStatus(new SensorStatus(), properties,
        new GetGroupSensorStatusHandler<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
            }
        });
    }
```

8.23.3.1.10 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*).

GetGroupSensorStatusHandler – 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);

      SensorPropertiesGet properties = SensorPropertiesGet.
            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
            }
      });
}
```

8.23.3.1.11 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*).

GetGroupSensorStatusHandler – 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 getSensorColumn(Group group, int propertyId, byte[] rawValueX) {
        ControlGroup controlGroup = new ControlGroup(group);

        SensorPropertiesGet properties = SensorPropertiesGet.
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
            }
        });
}
```

8.23.3.1.12 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*).

GetGroupSensorStatusHandler – 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 getSensorCadence(Group group, int propertyId) {
        ControlGroup controlGroup = new ControlGroup(group);

        SensorPropertiesGet properties = SensorPropertiesGet.
             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
            }
        });
    }
```

8.23.3.1.13 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*).

`GetGroupSensorStatusHandler` – 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 getSensorSetting(Group group, int propertyId, int settingId) {
        ControlGroup controlGroup = new ControlGroup(group);

        SensorPropertiesGet properties = SensorPropertiesGet.
         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
            }
        });
    }
```

8.23.3.1.14 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.

`GetGroupSensorStatusHandler` – 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 getSensorSettings(Group group, int propertyId) {
       ControlGroup controlGroup = new ControlGroup(group);

        SensorPropertiesGet properties = SensorPropertiesGet.
              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
            }
        });
    }
```

8.23.3.2 Set Sensor Setup Model Values

8.23.3.2.1 Set Sensor Cadence within the Node

SensorCadenceSet message requires filling 7 parameters describing the 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` you must pass a `SensorMessageFlags` object to the setter. Use the `setAcknowledgeRequired()` method of the `SensorMessageFlags` object to determine the required value of the 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(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
            }
        });
    }
```

8.23.3.2.2 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.

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

```
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
            }
        });
    }
```

8.23.3.2.3 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
            }
        });
    }
```

8.23.3.2.4 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*).

8.23.4 Control Light Control (LC) Models

Light Control (LC) models are used to handle automated lighting. These models are designed to automatically control a dimmable light on a device. LC Models have multiple configurable parameters and they are also able to receive external inputs from sensors.

The Light Control API allows for:

- Get or set LC Mode state
- Get or set LC Occupancy Mode state
- Get or set LC Light On-Off state
- Get or set LC Property state

To set up publication/subscription settings use the API prepared for SIG models (see sections 8.21.1 SIG model and 8.22.1 SIG Model, respectively).

To set up LC model bindings use the API prepared for SIG models (see sections 8.19 Bind a Model with a Group and 8.20 Unbind a Model from a Group).

8.23.4.1 Get LC Model Values

8.23.4.1.1 Get LC Mode from the Node

The request to get LC Mode state has no parameters.

The value received in LightControlModeStatus object (status) is an enumerated type and has two values:

- OFF – the controller is turned off. The binding with the Light Lightness state is disabled.
- ON – the controller is turned on. The binding with the Light Lightness state is enabled.

Response data received in the LightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

```
void getLightControlModeState(Element element, Group group) {
        ControlElement controlElement = new ControlElement(element, group);

        controlElement.getLightControlValue(
new LightControlModeGet(),
new LightControlElementCallback<LightControlModeStatus>() {
            @Override
            public void success(Element element, Group group, LightControlModeStatus status) {
                //handle result
            }

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

8.23.4.1.2 Get LC Occupancy Mode from the Node

The request to get LC Occupancy Mode state has no parameters.

The value received in LightControlOccupancyModeStatus object (status) is an enumerated type and has two values:

- STANDBY_TRANSITION_DISABLED – the controller does not transition from a standby state when occupancy is reported.
- STANDBY_TRANSITION_ENABLED – the controller may transition from a standby state when occupancy is reported.

Response data received in the LightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification)*.

```
void getLightControlOccupancyModeState(Element element, Group group) {
        ControlElement controlElement = new ControlElement(element, group);

        controlElement.getLightControlValue(
             new LightControlOccupancyModeGet(),
             new LightControlElementCallback<LightControlOccupancyModeStatus>() {
         @Override
         public void success(Element element, Group group, LightControlOccupancyModeStatus status) {
             //handle result
         }

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

8.23.4.1.3 Get LC Light On-Off from the Node

The request to get LC Light On-Off state has no parameters.

The values received in LightControlLightOnOffStatus object (status) are as follows:

- presentLightOnOff – is an enumerated type LightControlLightOnOff and represents the state of a Light Lightness controller. This variable can have the following values:
  - OFF_OR_STANDBY – State is equal to Off or equal to Standby.
  - NOT_OFF_AND_NOT_STANDBY - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an enumerated type LightControlLightOnOff (see above). This value is optional, meaning it does not have to be in the received status response. If it is not this value will be set to null.
- remainingTime – is of type Integer and specifies the time remaining to complete the transition in milliseconds. This value will be null unless the Target Light On-Off value is specified.

Response data received in the LightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

```
void getLightControlLightOnOffState(Element element, Group group) {
        ControlElement controlElement = new ControlElement(element, group);

        controlElement.getLightControlValue(
             new LightControlLightOnOffGet(),
             new LightControlElementCallback<LightControlLightOnOffStatus>() {
            @Override
            public void success(Element element, Group group, LightControlLightOnOffStatus status) {
                //handle result
            }
            @Override
            public void error(Element element, Group group, ErrorType error) {
                //handle error
            }
        });

    }
```

8.23.4.1.4 Get LC Property from the Node

The request to get LC Property state has one parameter:

- propertyId – 16-bit unsigned number identifying the ID of the property whose value will be received.

The values received in LightControlPropertyStatus object (status) are as follows:

- propertyId – specifies the ID of property whose value is received.
- propertyValue – Raw value for the received Property.

Response data received in the LightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

```
void getLightControlPropertyState(Element element, Group group, int propertyId) {
        ControlElement controlElement = new ControlElement(element, group);

        LightControlPropertyGet request = new LightControlPropertyGet(propertyId);

        controlElement.getLightControlValue(
             request,
             new LightControlElementCallback<LightControlPropertyStatus>() {
            @Override
            public void success(Element element, Group group, LightControlPropertyStatus status) {
                //handle result
            }

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

8.23.4.1.5 Get LC Mode from All Nodes in the Group

The request to get LC Mode state has no parameters.

The value received in LightControlModeStatus object (status) is an enumerated type and has two values:

- OFF – the controller is turned off. The binding with the Light Lightness state is disabled.
- ON – the controller is turned on. The binding with the Light Lightness state is enabled.

Response data received in the LightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getLightControlModeState(Group group) {
        ControlGroup controlGroup = new ControlGroup(group);

        MeshTask task = controlGroup.getLightControlValue(
            new LightControlModeGet(),
            new LightControlGroupHandler<LightControlModeStatus>() {
            @Override
            public void success(Element element, Group group, LightControlModeStatus status) {
                //handle result
            }

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

8.23.4.1.6 Get LC Occupancy Mode from All Nodes in the Group

The request to get LC Occupancy Mode state has no parameters.

The value received in LightControlOccupancyModeStatus object (status) is an enumerated type and has two values:

- STANDBY_TRANSITION_DISABLED – the controller does not transition from a standby state when occupancy is reported.
- STANDBY_TRANSITION_ENABLED – the controller may transition from a standby state when occupancy is reported.

Response data received in the LightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getLightControlOccupancyModeState(Group group) {
        ControlGroup controlGroup = new ControlGroup(group);


        MeshTask task = controlGroup.getLightControlValue(
              new LightControlOccupancyModeGet(),
              new LightControlGroupHandler<LightControlOccupancyModeStatus>() {
         @Override
         public void success(Element element, Group group, LightControlOccupancyModeStatus status) {
                //handle result
         }

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

 }
```

8.23.4.1.7 Get LC Light On-Off from All Nodes in the Group

The request to get LC Light On-Off state has no parameters.

The values received in LightControlLightOnOffStatus object (status) are as followed:

- presentLightOnOff – is an enumerated type LightControlLightOnOff and represents the state of a Light Lightness controller. This variable can have following values:
  - OFF_OR_STANDBY – State is equal to Off or equal to Standby.
  - NOT_OFF_AND_NOT_STANDBY - State is not equal to Off and not equal to Standby.
- targetLightOnOff – is an enumerated type LightCntrolLightOnOff (see above). This value is optional, meaning it does not have to be in the received status response. If it is not this value will be set to null.
- remainingTime – is of type Integer and specifies the time remaining to complete the transition in milliseconds. This value will be null unless the Target Light On-Off value is specified.

Response data received in the LightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getLightControlLightOnOffState(Group group) {
        ControlGroup controlGroup = new ControlGroup(group);

        MeshTask task = controlGroup.getLightControlValue(
              new LightControlLightOnOffGet(),
              new LightControlGroupHandler<LightControlLightOnOffStatus>() {
            @Override
            public void success(Element element, Group group, LightControlLightOnOffStatus status) {
                //handle result
            }

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

    }
```

8.23.4.1.8 Get LC Property from All Nodes in the Group

The request to get LC Property state has one parameter:

- propertyId – 16-bit unsigned number identifying ID of property whose value will be received.

The values received in LightControlPropertyStatus object (status) are as follows:

- propertyId – specifies the ID of the property whose value is received.
- propertyValue – Raw value for the received Property.

Response data received in the LightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getLightControlPropertyState(Group group, int propertyId) {
        ControlGroup controlGroup = new ControlGroup(group);

        LightControlPropertyGet request = new LightControlPropertyGet(propertyId);

        MeshTask task = controlGroup.getLightControlValue(
             request,
             new LightControlGroupHandler<LightControlPropertyStatus>() {
            @Override
            public void success(Element element, Group group, LightControlPropertyStatus status) {
                //handle result
            }

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

8.23.4.2 Set LC Setup Model Values

8.23.4.2.1 Set LC Mode within the Node

The request to set LC Mode state has one parameter:

- mode – an enumerated type LightControlMode which has two values:
  - OFF – the controller is turned off. The binding with the Light Lightness state is disabled.
  - ON – the controller is turned on. The binding with the Light Lightness state is enabled.

This variable determines desired value for LC Mode state of target element.

The value received in LightControlModeStatus object (status) is an enumerated type LightControlMode (for details see above).

Response data received in the LightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set LightControlMessageFlags it is required to pass a LightControlMessageFlags object to the request setter. Use the `setAcknowledged()` method of LightControlMessageFlags object to determine the required value of message flag.

```
void setLightControlModeState(Element element, Group group, LightControlMode mode) {
        ControlElement controlElement = new ControlElement(element, group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlModeSet request = new LightControlModeSet(mode);
        request.setFlags(flags);

        controlElement.setLightControlValue(
             request,
             new LightControlElementCallback<LightControlModeStatus>() {
            @Override
            public void success(Element element, Group group, LightControlModeStatus status) {
                //handle result
            }

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

    }
```

8.23.4.2.2 Set LC Occupancy Mode within the Node

The request to set LC Occupancy Mode state has one parameter:

- mode – an enumerated type LightControlOccupancyMode which has two values:
  - STANDBY_TRANSITION_DISABLED – the controller does not transition from a standby state when occupancy is reported.
  - STANDBY_TRANSITION_ENABLED – the controller may transition from a standby state when occupancy is reported.

This variable determines desired value for LC Occupancy Mode state of target element.

The value received in LightControlOccupancyModeStatus object (status) is an enumerated type LightControlOccupancyMode (for details see above).

Response data received in the LightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

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

```
void setLightControlOccupancyModeState(Element element, Group group, LightControlOccupancyMode mode)
{
        ControlElement controlElement = new ControlElement(element, group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlOccupancyModeSet request = new LightControlOccupancyModeSet(mode);
        request.setFlags(flags);

        controlElement.setLightControlValue(
             request,
             new LightControlElementCallback<LightControlOccupancyModeStatus>() {
         @Override
         public void success(Element element, Group group, LightControlOccupancyModeStatus status) {
             //handle result
         }

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

8.23.4.2.3 Set LC Light On-Off within the Node

The request to set LC Light On-Off state has four parameters:

- lightOnOff – determines desired value for LC Light On-Off state of target element. This variable is an enumerated type LightControl-LightOnOff and has two values:
  - OFF_OR_STANDBY – State is equal to Off or equal to Standby.
  - NOT_OFF_AND_NOT_STANDBY - State is not equal to Off and not equal to Standby.
- transactionId – 8-bit unsigned number identifying Transaction Identifier (TID). This value indicates whether the message is a new message or a retransmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transition to the target state in milliseconds. This value is optional. If this value will not be set (is null or has a value of 0xFFFFFFFF) the delay parameter will not be consider in the request.
- delay – Is of type Integer and determines the request execution delay in milliseconds. This value will not be considered in request if transitionTime is not set (has a default value null or 0xFFFFFFFF).

For full characteristic of LC Light OnOff Set Message see 6.3.5.3.2 Light LC Light OnOff Set from the *Mesh Model Bluetooth Specification*.

The values received in LightControlLightOnOffStatus object (status) are as follows:

- presentLightOnOff – is an enumerated type LightControlLightOnOff and represents the state of a Light Lightness controller (see above).
- targetLightOnOff – is an enumerated type LightCntrolLightOnOff (see above). This value is optional, meaning it does not have to be in the received status response. If it is not, this value will be set to null.
- remainingTime – is of type Integer and specifies the time remaining to complete the transition in milliseconds. This value will be null unless the Target Light On-Off value is specified.

Response data received in the LightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set LightControlMessageFlags, pass a LightControlMessageFlags object to the request setter. Use the `setAcknowledged()` method of the LightControlMessageFlags object to determine the required value of the message flag.

```
void setLightControlLightOnOffState(Element element, Group group, LightControlLightOnOff
lightOnOff, int transactionId, int transitionTime, int delay) {
        ControlElement controlElement = new ControlElement(element, group);


        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlLightOnOffSet request = new LightControlLightOnOffSet(lightOnOff, transactionId);
        request.setTransitionTime(transitionTime);
        request.setDelay(delay);
        request.setFlags(flags);

        controlElement.setLightControlValue(
             request,
             new LightControlElementCallback<LightControlLightOnOffStatus>() {
            @Override
            public void success(Element element, Group group, LightControlLightOnOffStatus status) {
                //handle result
            }

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

    }
```

8.23.4.2.4 Set LC Property within the Node

The request to set LC Property state has two parameters:

- propertyId – 16-bit unsigned number identifying the ID of the property whose value will be set.
- propertyValue – Raw value to set for the Property.

The values received in LightControlPropertyStatus object (status) are as follows:

- propertyId – specifies the ID of the property whose value is received.
- propertyValue – Raw value for the received Property.

Response data received in the LightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set LightControlMessageFlags it is required to pass a LightControlMessageFlags object to the request setter. Use `setAcknowledged()` method of LightControlMessageFlags object to determine required value of message flag.

```
void setLightControlPropertyState(Element element, Group group, int propertyId, byte[] propertyValue)
{
        ControlElement controlElement = new ControlElement(element, group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlPropertySet request = new LightControlPropertySet(propertyId, propertyValue);
        request.setFlags(flags);

        controlElement.setLightControlValue(
             request,
             new LightControlElementCallback<LightControlPropertyStatus>() {
            @Override
            public void success(Element element, Group group, LightControlPropertyStatus status) {
                //handle result
            }

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

    }
```

8.23.4.2.5 Set LC Mode within All Nodes in the Group

The request to set LC Mode state has one parameter:

- mode – an enumerated type LightControlMode which has two values:
  - OFF – the controller is turned off. The binding with the Light Lightness state is disabled.
  - ON – the controller is turned on. The binding with the Light Lightness state is enabled.

This variable determines desired value for LC Mode state of target elements.

The value received in LightControlModeStatus object (status) is an enumerated type LightControlMode (for details see above).

Response data received in the LightControlModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.1.4 Light LC Mode Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged.To set LightControlMessageFlags, pass a LightControlMessageFlags object to the request setter. Use the `setAcknowledged()` method of the LightControlMessageFlags object to determine the required value of message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void setLightControlModeState(Group group, LightControlMode mode) {
        ControlGroup controlGroup = new ControlGroup(group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlModeSet request = new LightControlModeSet(mode);
        request.setFlags(flags);

        MeshTask task = controlGroup.setLightControlValue(
             request,
             new LightControlGroupHandler<LightControlModeStatus>() {
            @Override
            public void success(Element element, Group group, LightControlModeStatus status) {
                //handle result
            }

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

    }
```

8.23.4.2.6 Set LC Occupancy Mode within All Nodes in the Group

The request to set LC Occupancy Mode state has one parameter:

- mode – an enumerated type LightControlOccupancyMode which has two values:
  - STANDBY_TRANSITION_DISABLED – the controller does not transition from a standby state when occupancy is reported.
  - STANDBY_TRANSITION_ENABLED – the controller may transition from a standby state when occupancy is reported.

This variable determines desired value for LC Occupancy Mode state of target elements.

The value received in LightControlOccupancyModeStatus object (status) is an enumerated type LightControlOccupancyMode (for details see above).

Response data received in the LightControlOccupancyModeStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.2.4 Light LC OM Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. In order to set LightControlMessageFlags it is required to pass a LightControlMessageFlags object to the request setter. Use the `setAcknowledged()` method of the LightControlMessageFlags object to determine the required value of the message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void setLightControlOccupancyModeState(Group group, LightControlOccupancyMode mode) {
        ControlGroup controlGroup = new ControlGroup(group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlOccupancyModeSet request = new LightControlOccupancyModeSet(mode);
        request.setFlags(flags);

        MeshTask task = controlGroup.setLightControlValue(
             request,
             new LightControlGroupHandler<LightControlOccupancyModeStatus>() {
         @Override
         public void success(Element element, Group group, LightControlOccupancyModeStatus status) {
             //handle result
         }

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

 }
```

8.23.4.2.7 Set LC Light On-Off within All Nodes in the Group

The request to set LC Light On-Off state has four parameters:

- lightOnOff – determines desired value for LC Light On-Off state of target elements. This variable is an enumerated type LightControlLightOnOff and has two values:
  - OFF_OR_STANDBY – State is equal to Off or equal to Standby.
  - NOT_OFF_AND_NOT_STANDBY - State is not equal to Off and not equal to Standby.
- transactionId – 8-bit unsigned number identifying Transaction Identifier (TID). This value indicates whether the message is a new message or a retransmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transition to the target state in milliseconds. This value is optional. If this value will not be set (is null or has a value of 0xFFFFFFFF) the delay parameter will not be consider in request.
- delay – Is of type Integer and determines request execution delay in milliseconds. This value will not be considered in the request if transitionTime is not set (has a default value of null or 0xFFFFFFFF).

For full characteristic of LC Light OnOff Set Message see 6.3.5.3.2 Light LC Light OnOff Set from the *Mesh Model Bluetooth Specification*.

The values received in LightControlLightOnOffStatus object (status) are as follows:

- presentLightOnOff – is an enumerated type LightControlLightOnOff and represents the current state of a Light Lightness controller (see above).
- targetLightOnOff – is an enumerated type LightCntrolLightOnOff (see above). This value is optional, meaning it does not have to be in the received status response. If it is not, this value will be set to null.
- remainingTime – is of type Integer and specifies the time remaining to complete the transition in milliseconds. This value will be null unless the Target Light On-Off value is specified.

Response data received in the LightControlLightOnOffStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.5.3.4 Light LC Light OnOff Status from the Mesh Model Bluetooth® Specification).

By default, the request is unacknowledged. In order to set LightControlMessageFlags it is required to pass a LightControlMessageFlags object to the request setter. Use `setAcknowledged()` method of LightControlMessageFlags object to determine required value of message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void setLightControlLightOnOffState(
       Group group,
       LightControlLightOnOff lightOnOff,
       int transactionId,
       int transitionTime,
       int delay) {
        ControlGroup controlGroup = new ControlGroup(group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlLightOnOffSet request = new LightControlLightOnOffSet(lightOnOff, transactionId);
        request.setFlags(flags);
        request.setTransitionTime(transitionTime);
        request.setDelay(delay);

        MeshTask task = controlGroup.setLightControlValue(
             request,
             new LightControlGroupHandler<LightControlLightOnOffStatus>() {
            @Override
            public void success(Element element, Group group, LightControlLightOnOffStatus status) {
                //handle result
            }

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

8.23.4.2.8 Set LC Property within All Nodes in the Group

The request to set LC Property state has two parameters:

- propertyId – 16-bit unsigned number identifying the ID of the property whose value will be set.
- propertyValue – Raw value to set for the Property.

The values received in LightControlPropertyStatus object (status) are as follows:

- propertyId – specifies the ID of the Property whose value is received.
- propertyValue – Raw value for the received Property.

Response data received in the LightControlPropertyStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 6.3.6.4 Light LC Property Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set LightControlMessageFlags, pass a LightControlMessageFlags object to the request setter. Use the `setAcknowledged()` method of the LightControlMessageFlags object to determine the required value of the message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void setLightControlPropertyState(Group group, int propertyId, byte[] propertyValue) {
        ControlGroup controlGroup = new ControlGroup(group);

        LightControlMessageFlags flags = new LightControlMessageFlags();
        flags.setAcknowledged(true);

        LightControlPropertySet request = new LightControlPropertySet(propertyId, propertyValue);
        request.setFlags(flags);

        MeshTask    task    =    controlGroup.setLightControlValue(request,    new    LightControl-
GroupHandler<LightControlPropertyStatus>() {
            @Override
            public void success(Element element, Group group, LightControlPropertyStatus status) {
                //handle result
            }

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

    }
```

8.23.4.3 Notifications

Notifications with statuses can be received from LC models. It means that a node can publish its current state to all subscribers.

Subscription can be done to a specific element or to a group and to one or more status types.

To receive group notifications after subscription, the node and specific model must be bound to the group. Also the node must be configured to publish its status to the specific address, either group or device, that matches the subscription.

8.23.4.3.1 Subscribe to Status Notifications from a Single Node

The subscription takes one parameter that describes the status of which to be notified. This object is the same as when manually requesting a single status report.

The success handler is executed every time the subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object that contains its current status.

The error handler is executed either if the subscription failed or the node has reported any error.

After subscription, the callback will be called for every received notification until manually unsubscribed from this notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

```
void subscribeToLightControlPropertyChange(Element element, Group group, int propertyId) {
        ControlElementPublications elementPublications = new ControlElementPublications(
element, group);

        MeshTask meshTask = elementPublications.subscribe(
             new LightControlPropertyGet(propertyId),
             new LightControlElementPublicationHandler<LightControlPropertyStatus>() {
           @Override
           public void success(Element element, Group group, LightControlPropertyStatus status) {
               //handle result...
           }

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

}
```

8.23.4.3.2 Subscribe to Status Notifications from a Group of Nodes

Subscription to group notifications is similar to subscription to a specific node. The only difference is that the subscribe method from ControlGroupPublications should be used instead of ControlElementPublications and that this object's constructor takes only a group identifier.

```
void subscribeToLightControlPropertyChange(Group group, int propertyId) {
        ControlGroupPublications groupPublications = new ControlGroupPublications(group);

       MeshTask meshTask = groupPublications.subscribe(
             new LightControlPropertyGet(propertyId),
             new LightControlGroupPublicationHandler<LightControlPropertyStatus>() {
           @Override
           public void success(Element element, Group group, LightControlPropertyStatus status) {
               //handle result...
           }

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

After subscription, the callback will be called for every received notification until manually unsubscribed from this notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

8.23.5 Control Scene Models

Scenes serve as memory banks for storage of states (such as a power level or a light level/color). Values of states of an element can be stored as a scene and can be recalled later from the scene memory. Ultimately scenes support adjusting multiple states of different nodes in one action.

The Scene API allows for:

- Get the current status of a currently active scene of an element
- Get the current status of the Scene Register of an element. The Scene Register state is a 16-element array of 16-bit values representing a scene number. These values are associated with a storage container that stores information about the scene state.
- Store the current state of an element as a Scene
- Recall the current state of an element from a previously stored scene.
- Delete a scene from the Scene Register state of an element.

To set up publication/subscription settings use the API developed for SIG models (see sections 8.21.1 SIG model and 8.22.1 SIG Model, respectively).

To set up Scene model bindings use the API developed for SIG models (see sections 8.19 Bind a Model with a Group and 8.20 Unbind a Model from a Group).

8.23.5.1 Get Scene Model Values

8.23.5.1.1 Get Scene Status from a Node

The request to get Scene Status has no parameters.

The response to a Scene Status Get request is the SceneStatus object. This object has four properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is null when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is null when the scene changing process does not occur.

Response data received in the SceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

```
void getSceneStatus(Element element, Group group) {
        ControlElement controlElement = new ControlElement(element, group);

        controlElement.getSceneValue(new SceneGet(), new SceneElementCallback<SceneStatus>() {
            @Override
            public void success(Element element, Group group, SceneStatus status) {
                // handle result...
            }

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

8.23.5.1.2 Get Scene Register Status from a Node

The request to get Scene Register Status has no parameters.

The response to a Scene Register Status Get request is the SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

```
void getSceneRegisterStatus(Element element, Group group) {
        ControlElement controlElement = new ControlElement(element, group);

        controlElement.getSceneValue(new SceneRegisterGet(), new
SceneElementCallback<SceneRegisterStatus>() {
            @Override
            public void success(Element element, Group group, SceneRegisterStatus status) {
               // handle result...
            }

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

8.23.5.1.3 Get Scene Status from All Nodes in a Group

The request to get Scene Status has no parameters.

The response to a Scene Status Get request is the SceneStatus object. This object has four properties:

- statusCode – an enumerated type that specifies the outcome of last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is null when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is null when the scene changing process does not occur.

Response data received in the SceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

Note that the success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getSceneStatus(Group group) {
        ControlGroup controlGroup = new ControlGroup(group);

        MeshTask task = controlGroup.getSceneValue(
              new SceneGet(),
              new SceneGroupHandler<SceneStatus>() {
           @Override
           public void success(Element element, Group group, SceneStatus status) {
               // handle result...
           }

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

8.23.5.1.4 Get Scene Register Status from All Nodes in a Group

The request to get Scene Register Status has no parameters.

The response to a Scene Register Status Get request is the SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void getSceneRegisterStatus(Group group) {
        ControlGroup controlGroup = new ControlGroup(group);

        MeshTask task = controlGroup.getSceneValue(
              new SceneRegisterGet(),
              new SceneGroupHandler<SceneRegisterStatus>() {
            @Override
            public void success(Element element, Group group, SceneRegisterStatus status) {
                // handle result...
            }

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

8.23.5.2 Control Scene Model Values

8.23.5.2.1 Store Scene within a Node

The request to store scene has one parameter:

- scene – an object that represents a Scene to be stored. Use the createScene method from the Network object to get new instance of a scene object.

The response to a Scene Store request is the SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of the message flag.

```
void storeScene(Element element, Group group, Scene scene, boolean acknowledge) {
        ControlElement controlElement = new ControlElement(element, group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        controlElement.changeSceneRegister(
              new SceneStore(scene, flags),
              new SceneElementCallback<SceneRegisterStatus>() {
           @Override
           public void success(Element element, Group group, SceneRegisterStatus status) {
               // handle result...
           }

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

8.23.5.2.2 Recall Scene Within a Node

The request to recall a scene has four parameters:

- scene – an object that represents a Scene to be recalled.
- transactionId – 8-bit unsigned number identifying the Transaction Identifier (TID). This value indicates whether the message is a new message or a re-transmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transit to the state defined by the scene being recalled. This value is optional. If this value is null, the delay parameter will not be considered in the request.
- delay – Is of type Integer and determines the request execution delay in milliseconds. This value will not be considered in the request if the transition-Time is null or has maximum value of 0xFFFFFFFF.

The response to a Scene Recall request is the SceneStatus object. This object has four properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is null when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is null when the scene changing process does not occur.

Response data received in the SceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of the message flag.

```
void recallScene(Element element, Group group, Scene scene, int transactionId, boolean acknowledge)
{
        ControlElement controlElement = new ControlElement(element, group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        SceneRecall recallRequest = new SceneRecall(scene, transactionId, flags);
        Integer customTransitionTime = ...
        Integer customDelay = ...
        recallRequest.setTransitionTime(customTransitionTime);
        recallRequest.setDelay(customDelay);

        controlElement.changeSceneRegister(recallRequest, new SceneElementCallback<SceneStatus>() {
            @Override
            public void success(Element element, Group group, SceneStatus status) {
                //handle result...
            }

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

8.23.5.2.3 Delete Scene Within a Node

The request to delete a scene has one parameter:

- scene – an object that represents a Scene to be deleted.

The response to a Scene Delete request is the SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of message flag.

```
void deleteScene(Element element, Group group, Scene scene, boolean acknowledge) {
        ControlElement controlElement = new ControlElement(element, group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        controlElement.changeSceneRegister(
              new SceneDelete(scene, flags),
              new SceneElementCallback<SceneRegisterStatus>() {
           @Override
           public void success(Element element, Group group, SceneRegisterStatus status) {
               // handle result...
           }

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

8.23.5.2.4 Store Scene Within All Nodes in a Group

The request to store a scene has one parameter:

- scene – an object which represents a Scene to be stored. Use the createScene method from the Network object to get new instance of the scene object.

The response to a Scene Store request is the SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses, call the `cancel()` method on task object: `task.cancel()`.

```
void storeScene(Group group, Scene scene, boolean acknowledge) {
        ControlGroup controlGroup = new ControlGroup(group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        MeshTask task = controlGroup.changeSceneRegister(
              new SceneStore(scene, flags),
              new SceneGroupHandler<SceneRegisterStatus>() {
           @Override
           public void success(Element element, Group group, SceneRegisterStatus status) {
               // handle result...
           }

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

8.23.5.2.5 Recall Scene within All Nodes in a Group

The request to recall a scene has four parameters:

- scene – an object that represents a Scene to be recalled.
- transactionId – 8-bit unsigned number identifying the Transaction Identifier (TID). This value indicates whether the message is a new message or a re-transmission of a previously sent message.
- transitionTime – Is of type Integer and specifies the time an element will take to transit to the state defined by the scene being recalled. This value is optional. If this value is null, the delay parameter will not be considered in the request.
- delay – Is of type Integer and determines the request execution delay in milliseconds. This value will not be considered in the request if transitionTime is null or has a maximim value of 0xFFFFFFFF.

The response to a Scene Recall request is the SceneStatus object. This object has four properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- targetScene – a 16-bit value identifying the number of the Scene to be reached by an element at the end of the scene changing process. This value is null when the scene changing process does not occur.
- remainingTime – number of milliseconds to finish the scene changing process. This value is null when the scene changing process does not occur.

Response data received in the SceneStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.6 Scene Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of the message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void recallScene(Group group, Scene scene, int transactionId, boolean acknowledge) {
        ControlGroup controlGroup = new ControlGroup(group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        SceneRecall recallRequest = new SceneRecall(scene, transactionId, flags);
        Integer customTransitionTime = ...
        Integer customDelay = ...
        recallRequest.setTransitionTime(customTransitionTime);
        recallRequest.setDelay(customDelay);

        MeshTask task = controlGroup.changeSceneRegister(
               recallRequest,
               new SceneGroupHandler<SceneStatus>() {
           @Override
           public void success(Element element, Group group, SceneStatus status) {
               //handle result...
           }

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

8.23.5.2.6 Delete Scene Within All Nodes in a Group

The request to delete a scene has one parameter:

- scene – an object that represents a Scene to be deleted.

The response to a Scene Delete request is SceneRegisterStatus object. This object has three properties:

- statusCode – an enumerated type that specifies the outcome of the last operation. This type has three values:
  - SUCCESS
  - SCENE_REGISTER_FULL
  - SCENE_NOT_FOUND
- currentScene – a 16-bit value identifying the scene number of the current Scene. If no scene is active, this value will be zero.
- scenes – an array of 16-bit values (scenes numbers) stored within an element.

Response data received in the SceneRegisterStatus object has a structure defined by the Bluetooth SIG Mesh Specification (see section 5.2.2.8 Scene Register Status from the *Mesh Model Bluetooth® Specification*).

By default, the request is unacknowledged. To set SceneMessageFlags, pass a SceneMessageFlags object to the request constructor. Use the `setAcknowledged()` method of the SceneMessageFlags object to determine the required value of the message flag.

Note that success method is called every time the Provisioner receives a success response from the request sent through a group multicast address. To stop receiving these responses call the `cancel()` method on task object: `task.cancel()`.

```
void deleteScene(Group group, Scene scene, boolean acknowledge) {
        ControlGroup controlGroup = new ControlGroup(group);

        SceneMessageFlags flags = new SceneMessageFlags();
        flags.setAcknowledged(acknowledge);

        MeshTask task = controlGroup.changeSceneRegister(
              new SceneDelete(scene, flags),
              new SceneGroupHandler<SceneRegisterStatus>() {
           @Override
           public void success(Element element, Group group, SceneRegisterStatus status) {
               // handle result...
           }

           @Override
           public void error(Group group, ErrorType error) {
               // handler error...
           }
       });
    }
```

8.23.5.3 Notifications

8.23.5.3.1 Subscribe to Status Notifications from a Single Node

The subscription takes one parameter that describes the status to be notified about. This object is the same as when manually requesting a single status report.

The success handler is executed every time that a subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object contains its current status.

The error handler is executed either if the subscription request fails or the node reports any error.

After subscription, a callback will be called for every received notification until manually unsubscribed from the notification. To unsubscribe, the `cancel()` method should be called on the `task` object returned after subscription.

```
void subscribeToSceneRegisterGet(Element element, Group group) {
        ControlElementPublications elementPublications = new ControlElementPublications(element,
group);

        MeshTask meshTask = elementPublications.subscribe(
                new SceneRegisterGet(),
                new SceneElementPublicationHandler<SceneRegisterStatus>() {
                    @Override
                    public void success(Element element, Group group, SceneRegisterStatus status) {
                        //handle result...
                    }

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

8.23.5.3.2 Subscribe to Status Notifications from a Group of Nodes

Subscription to group notifications is similar to subscription to a specific node. The only difference is that the subscribe method from ControlGroupPublications should be used instead of that from ControlElementPublications and that this object's constructor takes only a group identifier.

```
void subscribeToSceneRegisterGet(Group group) {
        ControlGroupPublications groupPublications = new ControlGroupPublications(group);

        MeshTask meshTask = groupPublications.subscribe(
                new SceneRegisterGet(),
                new SceneGroupPublicationHandler<SceneRegisterStatus>() {
                    @Override
                    public void success(Element element, Group group, SceneRegisterStatus status) {
                        //handle result...
                    }

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

8.23.6 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 7.2 Initializing the BluetoothMesh).

8.23.6.1 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).

```
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 = new LocalVendorSettings(opCodes, messageHandler);
```

8.23.6.2 Create LocalVendorRegistrator

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

```
LocalVendorModel localVendorModel;

//(…)

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

8.23.6.2.1 Register a Local Vendor Model

Note: Typically, registration should be done one time for the local vendor model after initializing BluetoothMesh (see section 7.2 Initializing the BluetoothMesh).

```
LocalVendorRegistrator vendorRegistrator;
LocalVendorSettings registerSettings;

//(…)

vendorRegistrator.registerSettings(registerSettings);
```

8.23.6.2.2 Unregister a Local Vendor Model

```
LocalVendorRegistrator vendorRegistrator;

//(…)

vendorRegistrator.unregister();
```

8.23.7 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 7.18 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.

8.23.7.1 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.

```
AppKey appKey;
LocalVendorModel localVendorModel;

//(…)

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

8.23.7.2 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.

```
AppKey appKey;
LocalVendorModel localVendorModel;

//(…)

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

8.23.8 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 7.21 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, either 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.

8.23.8.1 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.

8.23.8.1.1 Notifications Come by Group Address

```
Group group;
VendorModel vendorModel;

//(…)

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

8.23.8.1.2 Notifications Come by Node Address

```
Node node;
VendorModel vendorModel;

//(…)

VendorModelNotifications vendorNotifications = new VendorModelNotifications(node);
vendorNotifications.signUpForNotifications(vendorModel);
```

8.23.8.2 Sign Out from the Notifications

Method used to unsubscribe from the vendor model notifications.

8.23.8.2.1 Sign Out from Notifications from the Group Address

```
Group group;
VendorModel vendorModel;

//(…)

VendorModelNotifications vendorNotifications = new VendorModelNotifications(group);
vendorNotifications.signOutFromNotifications(vendorModel);
```

8.23.8.2.2 Sign Out from Notifications from the Node Address

```
Node node;
VendorModel vendorModel;

//(…)

VendorModelNotifications vendorNotifications = new VendorModelNotifications(node);
vendorNotifications.signOutFromNotifications(vendorModel);
```

8.23.9 Send Value to VendorModel

8.23.9.1 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:

```
class CompanyControlSetVendorModel implements ControlValueSetVendorModel {

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

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

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

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

8.23.9.2 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:

```
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();
```

8.23.9.3 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.

```
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.
            }
        });
```

8.23.9.4 Send Prepared Message to the Group

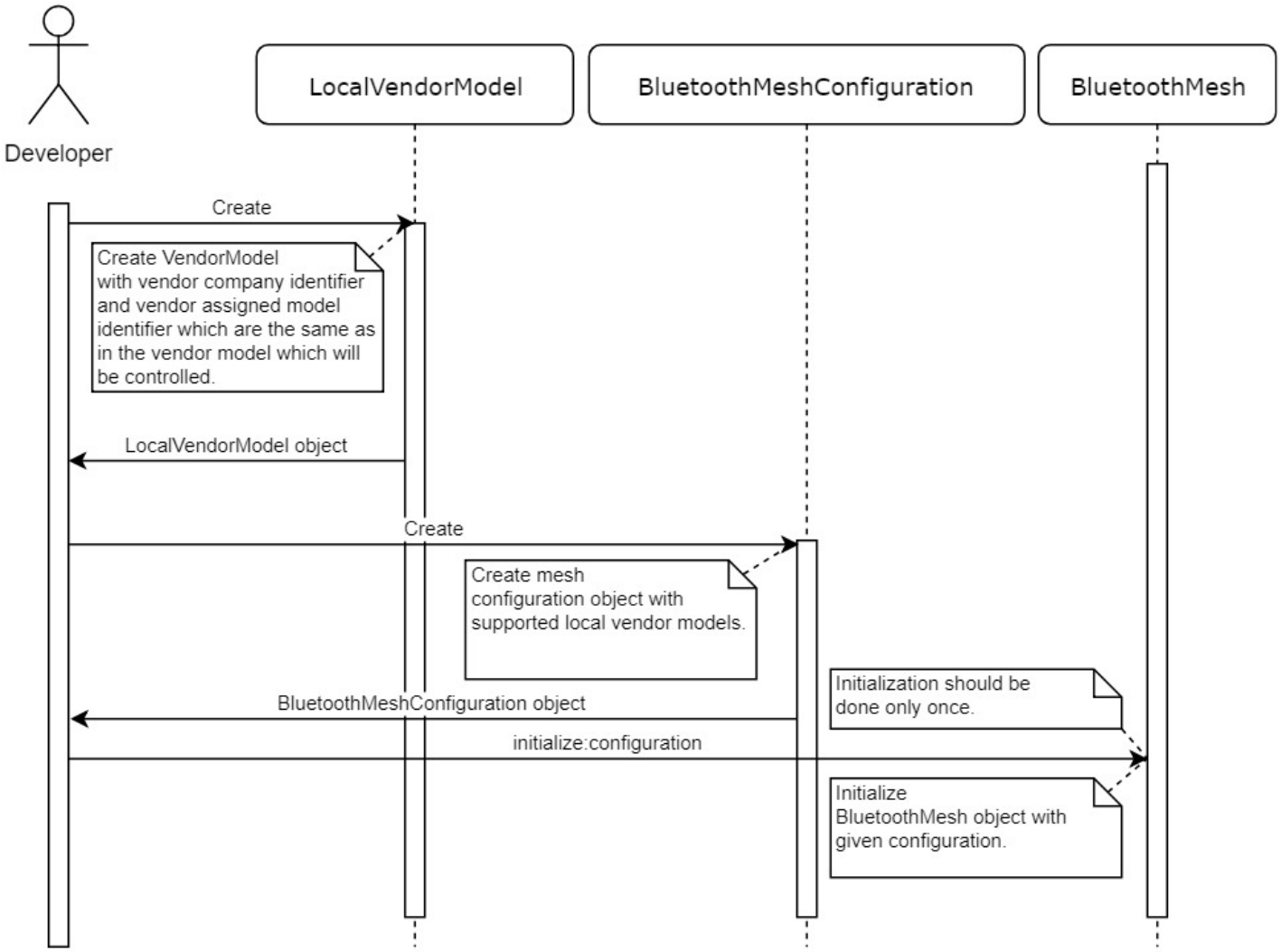
Note: Models have to be subscribed to a given group.

```
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.
            }
        });
```

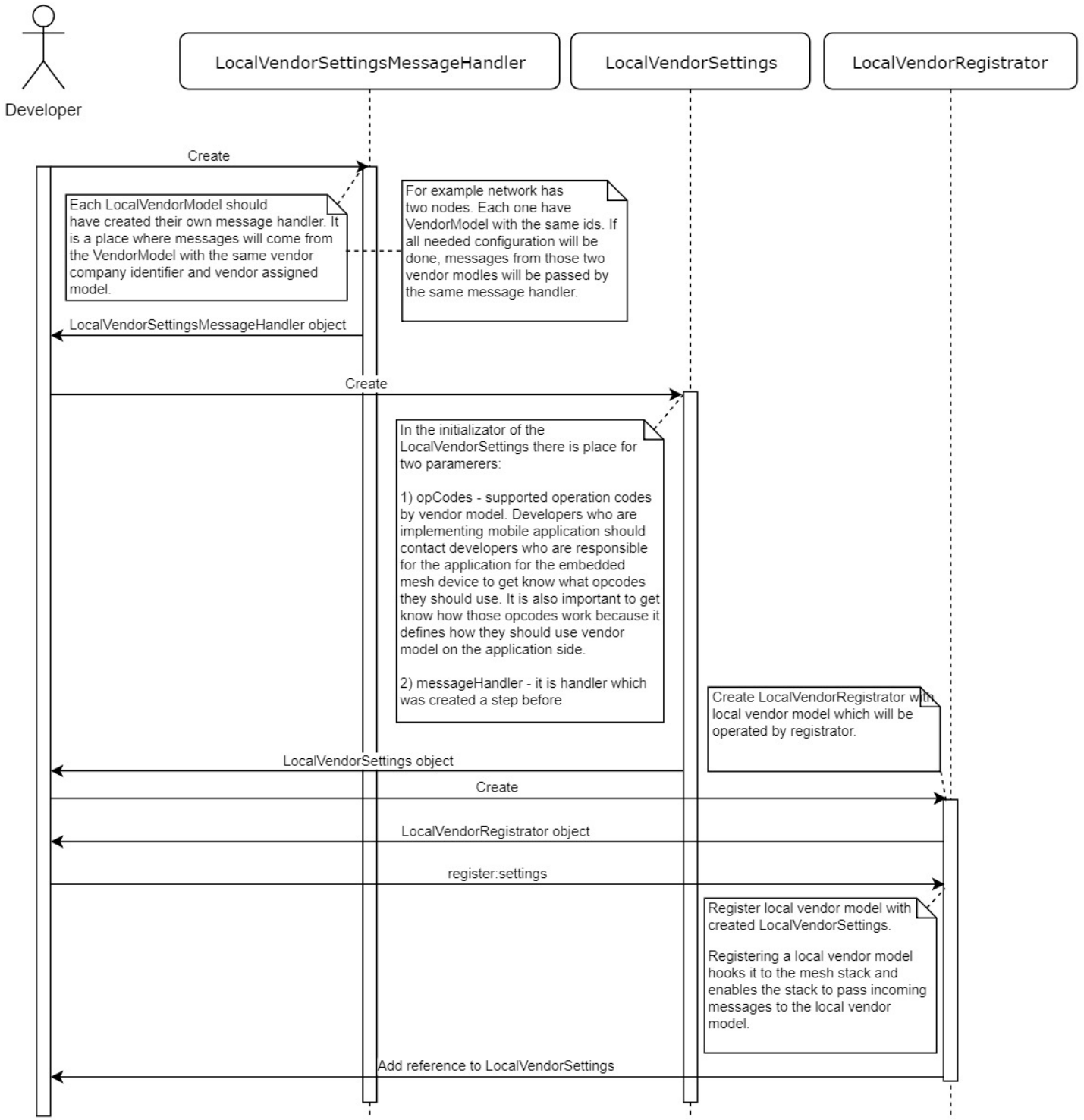
8.24 Sequence Diagrams for Vendor Model Functionality

8.24.1 LocalVendorModel Initialization

8.24.1.1 Create LocalVendorModel



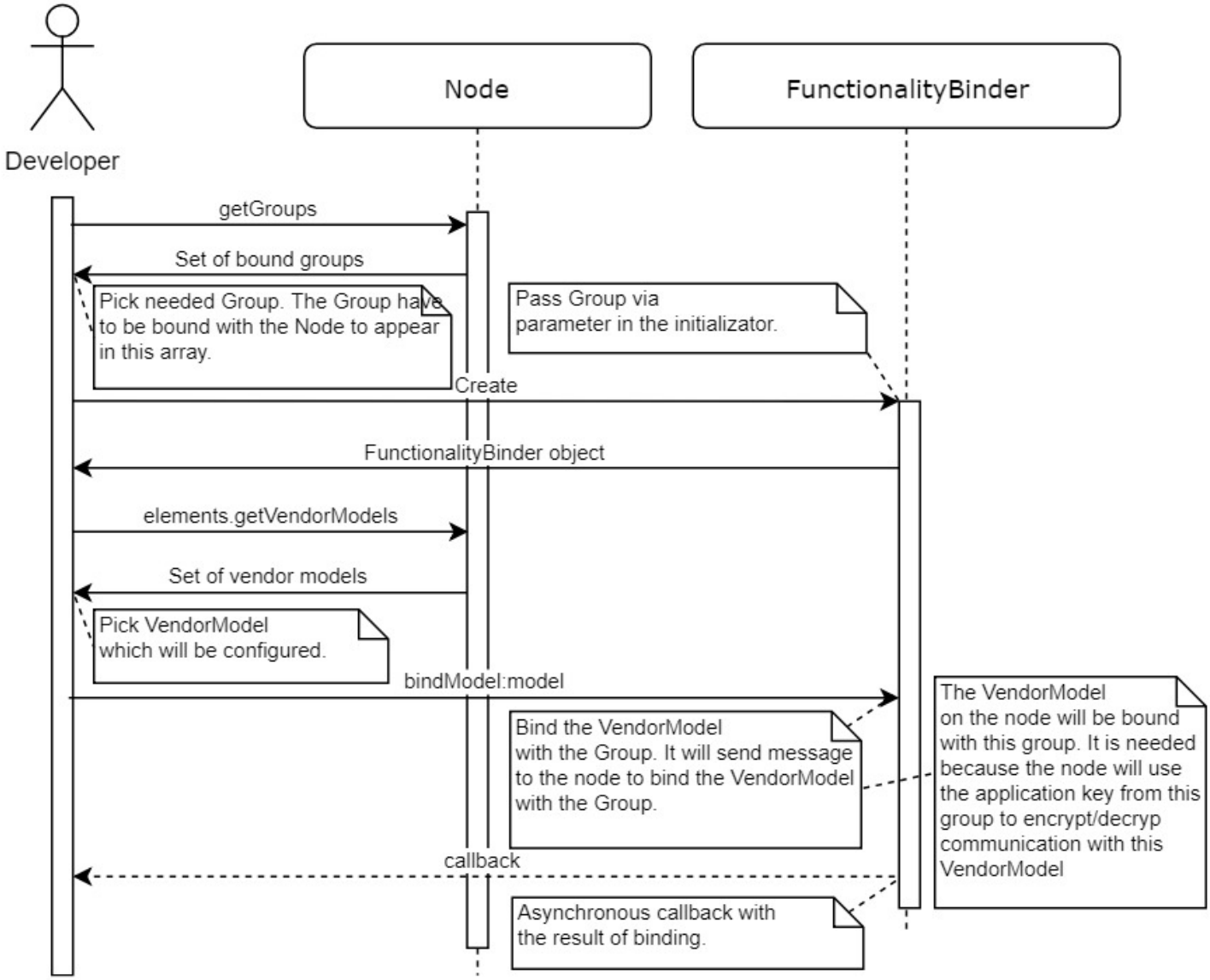
8.24.1.2 Register the LocalVendorModel in the Mesh Stack



8.24.2 Configure the VendorModel

8.24.2.1 Bind the VendorModel with the Group

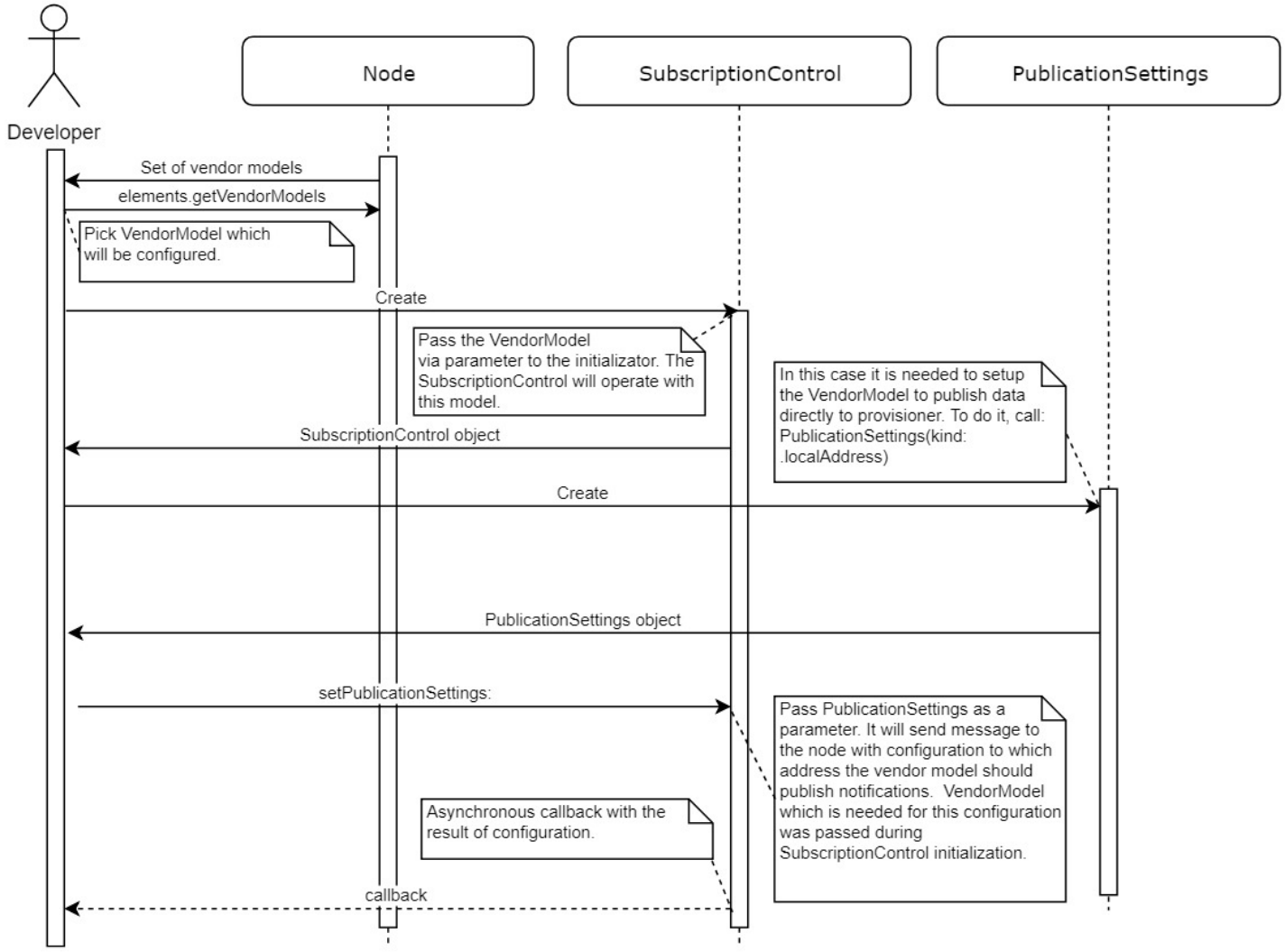
It is always needed to bind vendor model on the node with the group.



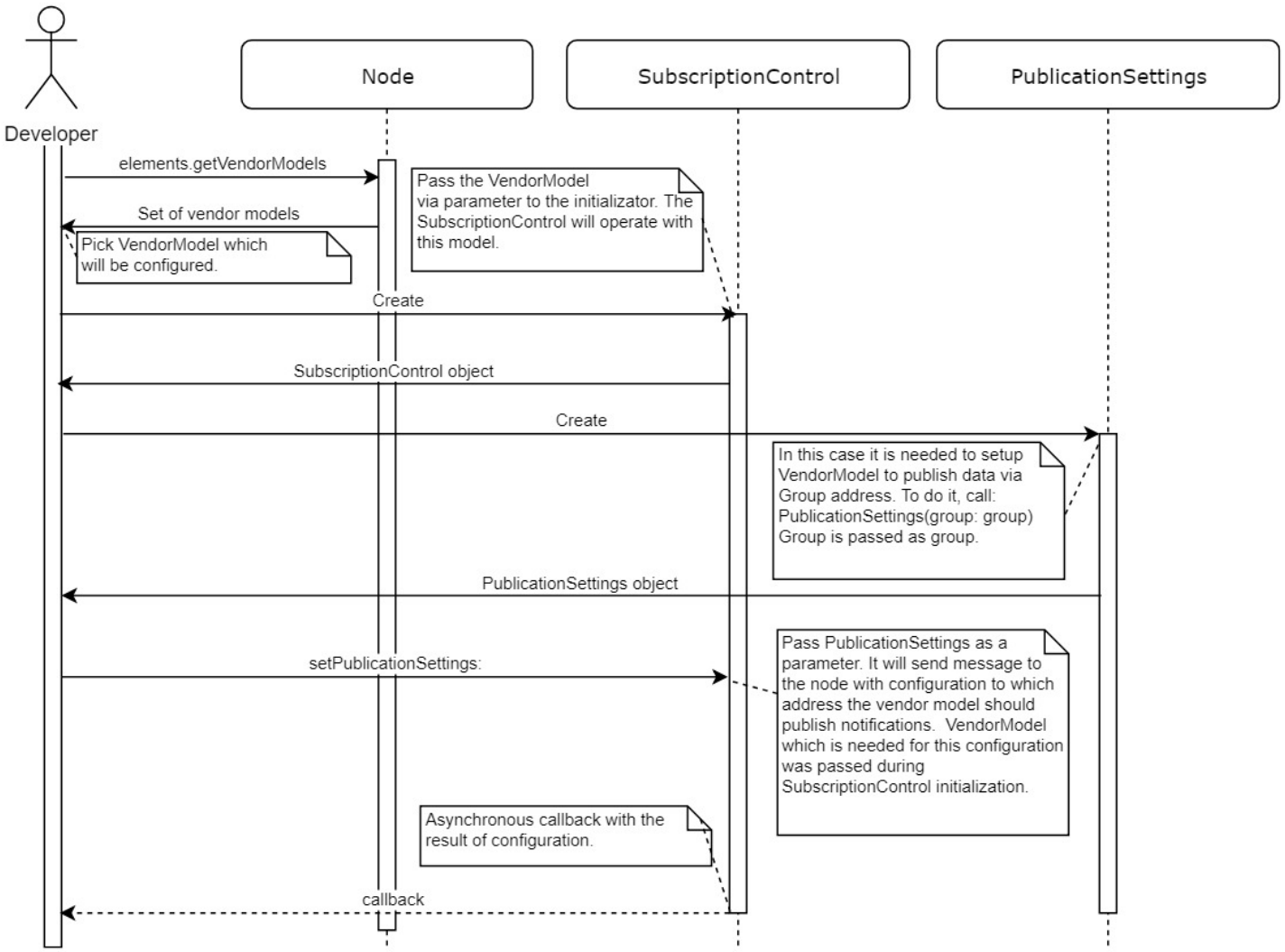
8.24.2.2 Send Publication Settings to the VendorModel

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

8.24.2.2.1 Set the Provisioner Address as the Publication Address

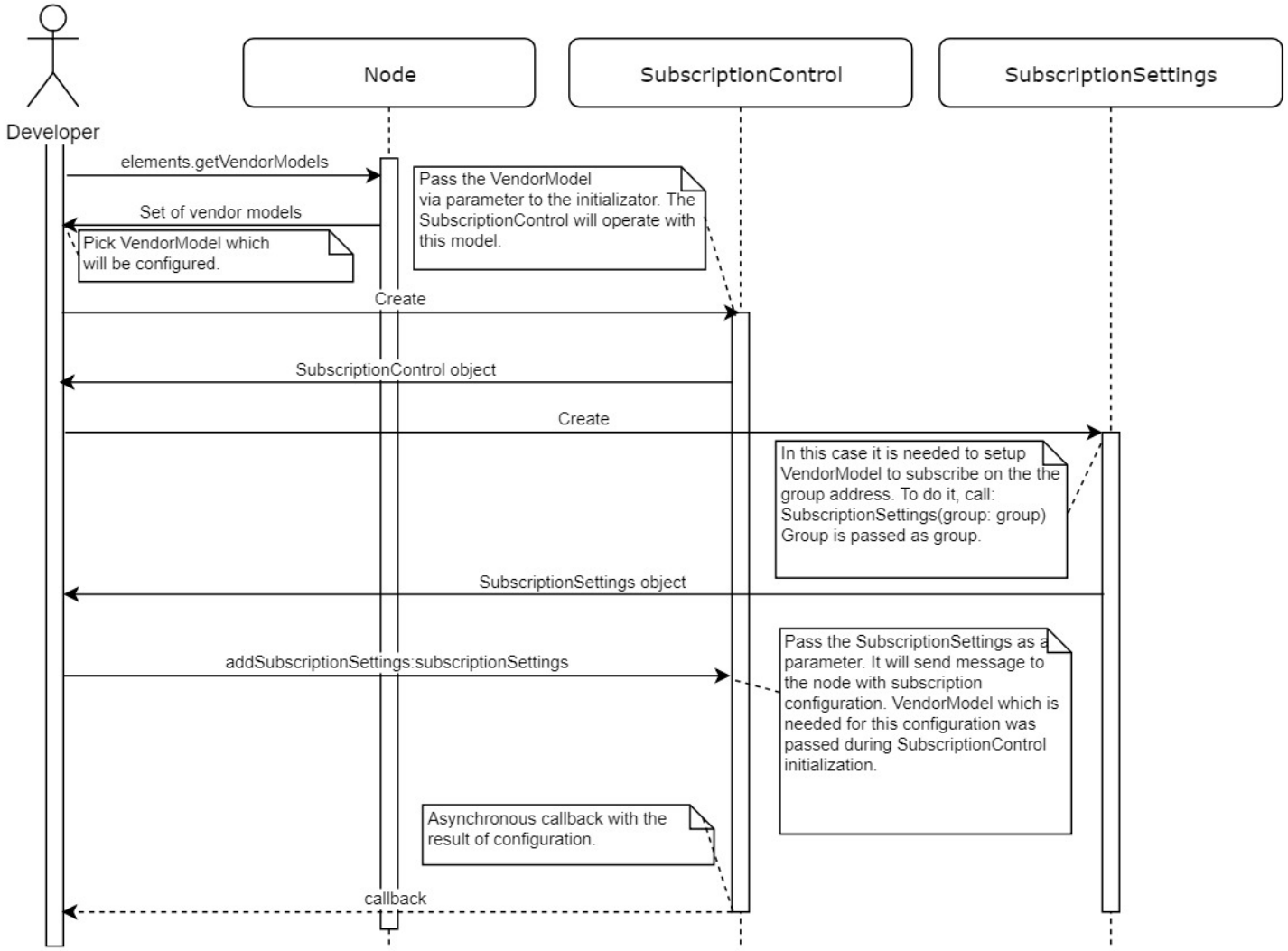


8.24.2.2.2 Set the Group Address as the Publication Address



8.24.2.3 Send Subscription Settings to the VendorModel

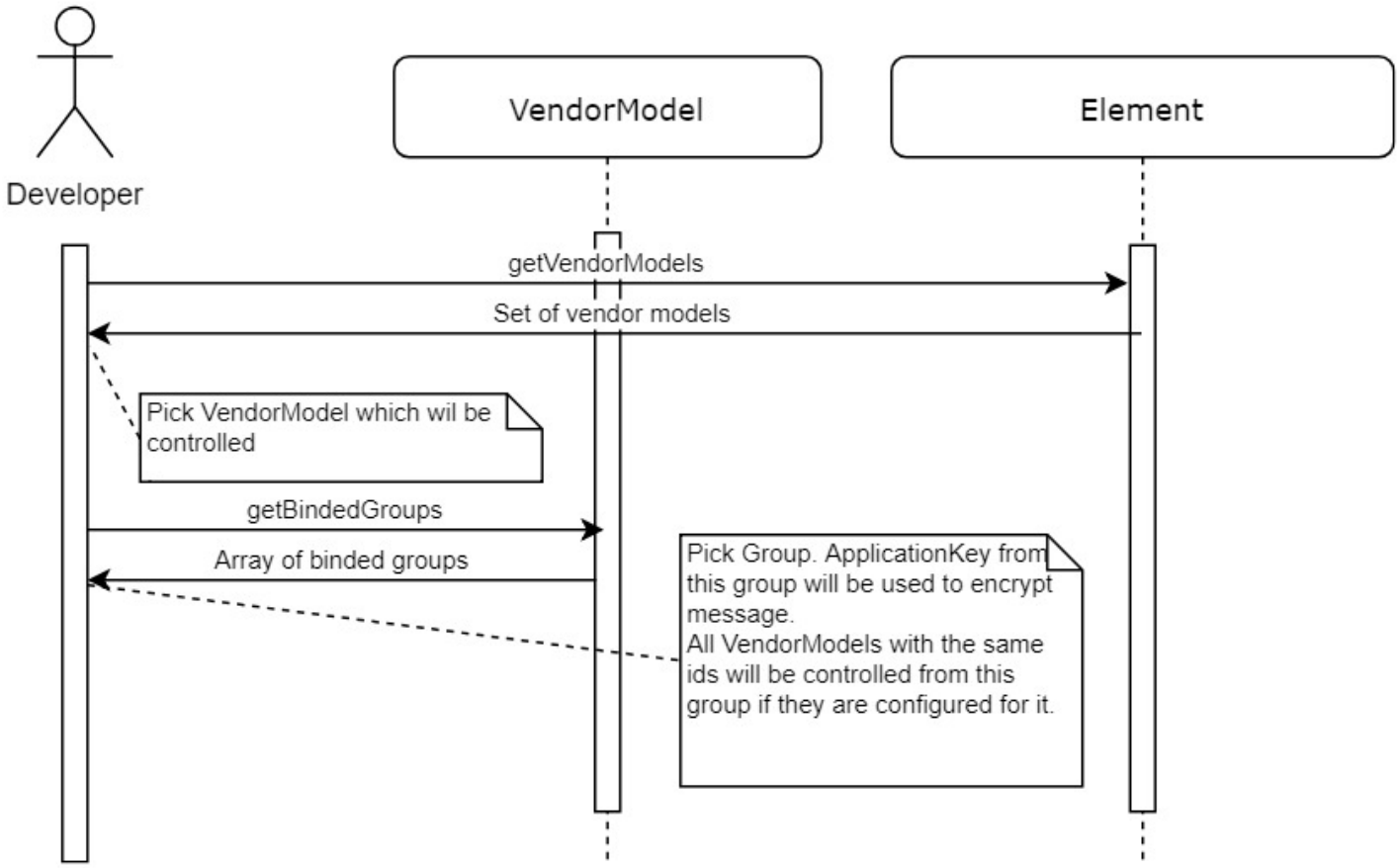
When the vendor model on the node expects a message from the group address, you must send the subscription settings containing the address of this group to the vendor model on the node. Without that the vendor model will not receive a message sent by the group address,

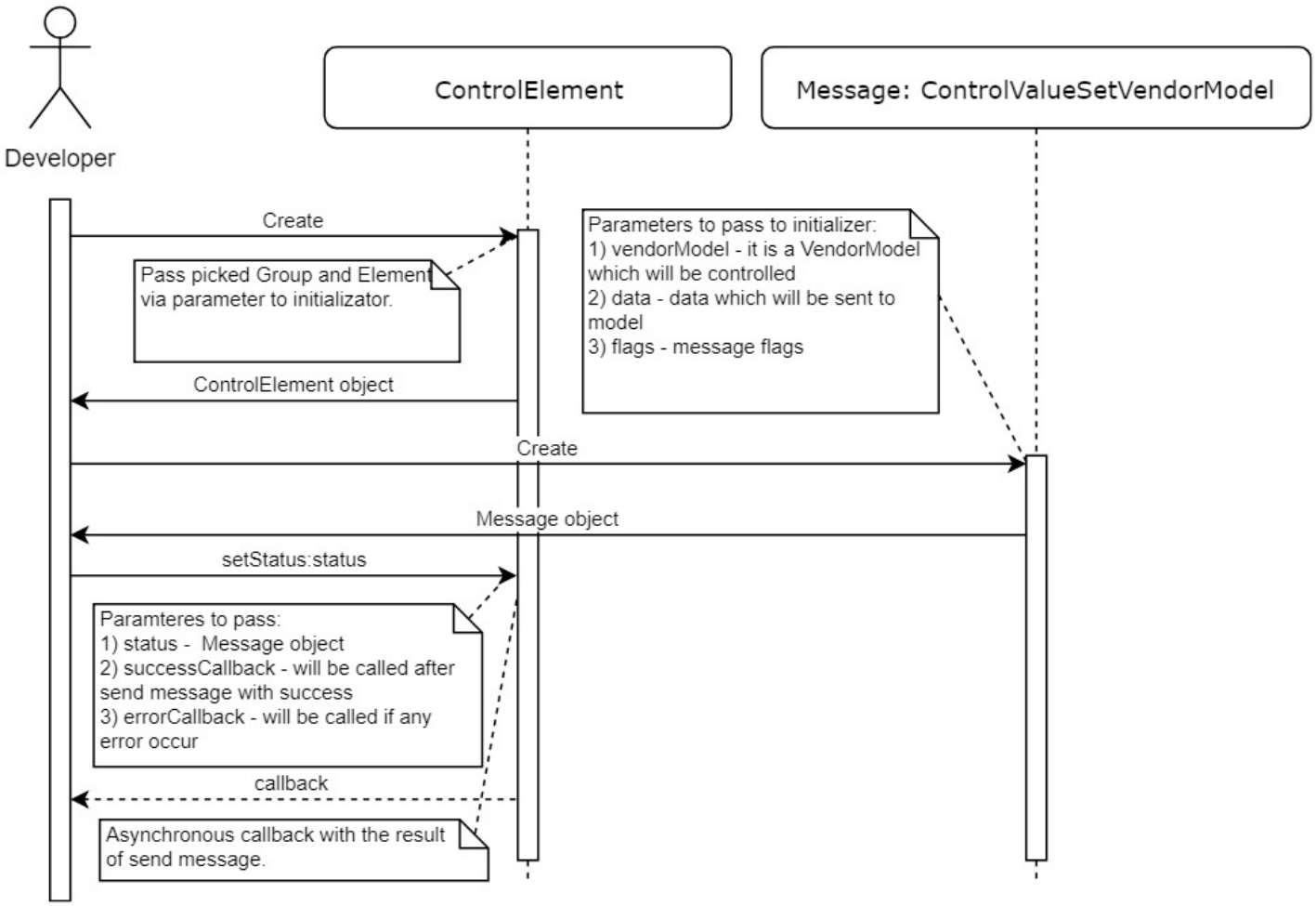


8.24.3 Send Message to the Vendor Model on the Node

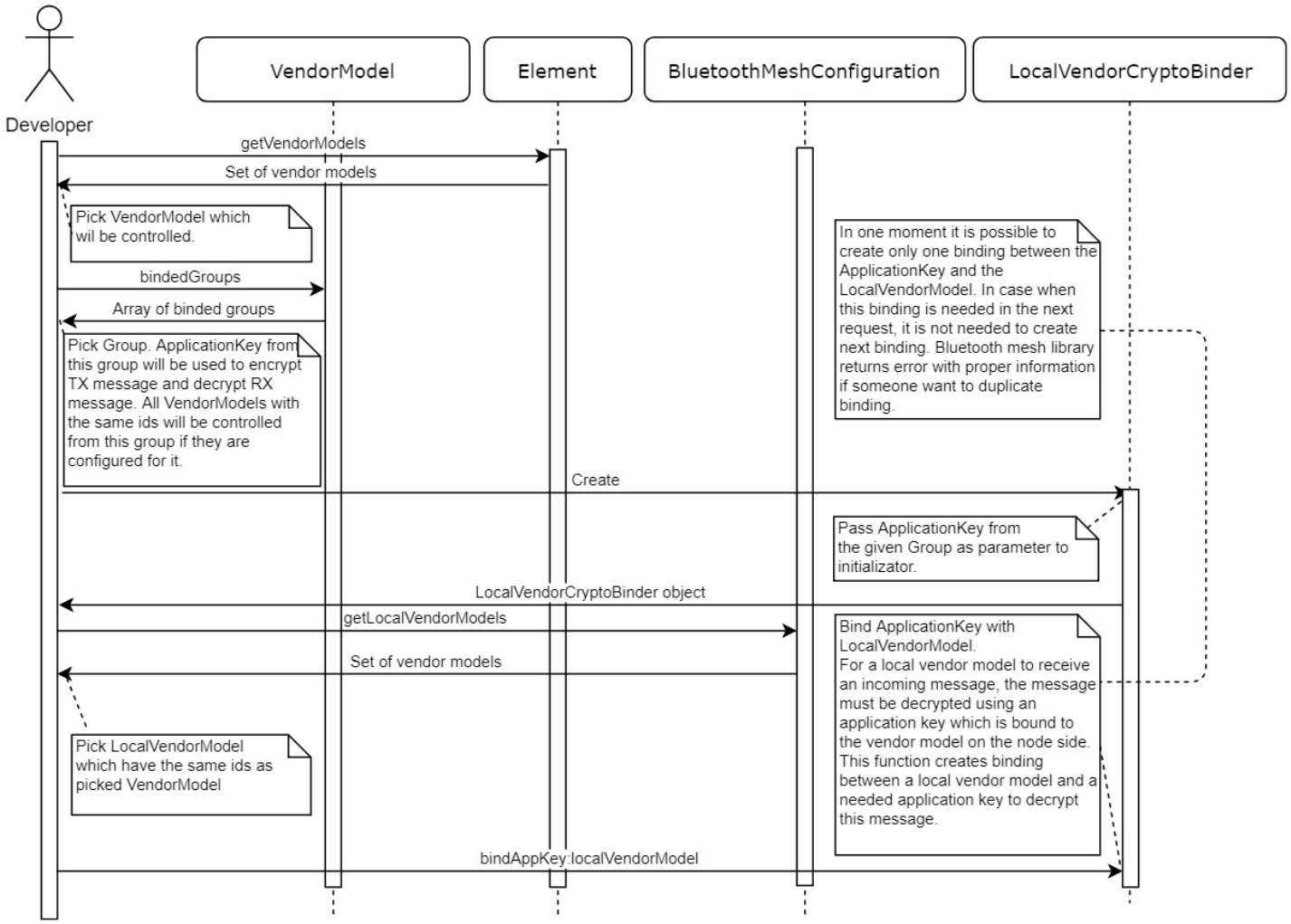
8.24.3.1 Send Message Directly to the Node

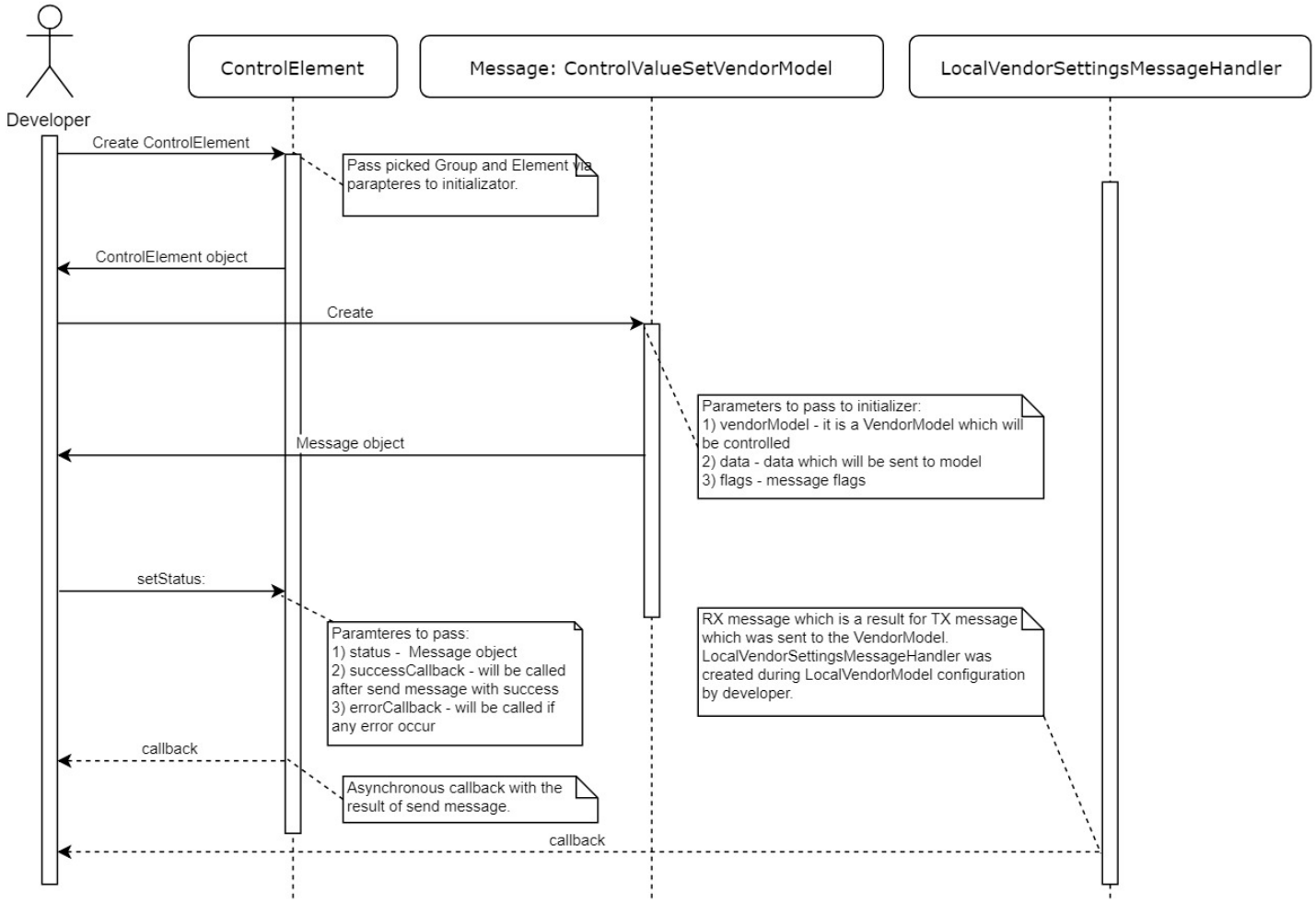
8.24.3.1.1 Message without Response




Developer
ControlElement
Message: ControlValueSetVendorModel
Create
Pass picked Group and Element via parameter to initializator.
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
ControlElement object
Create
Message object
setStatus:status
Paramteres 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.

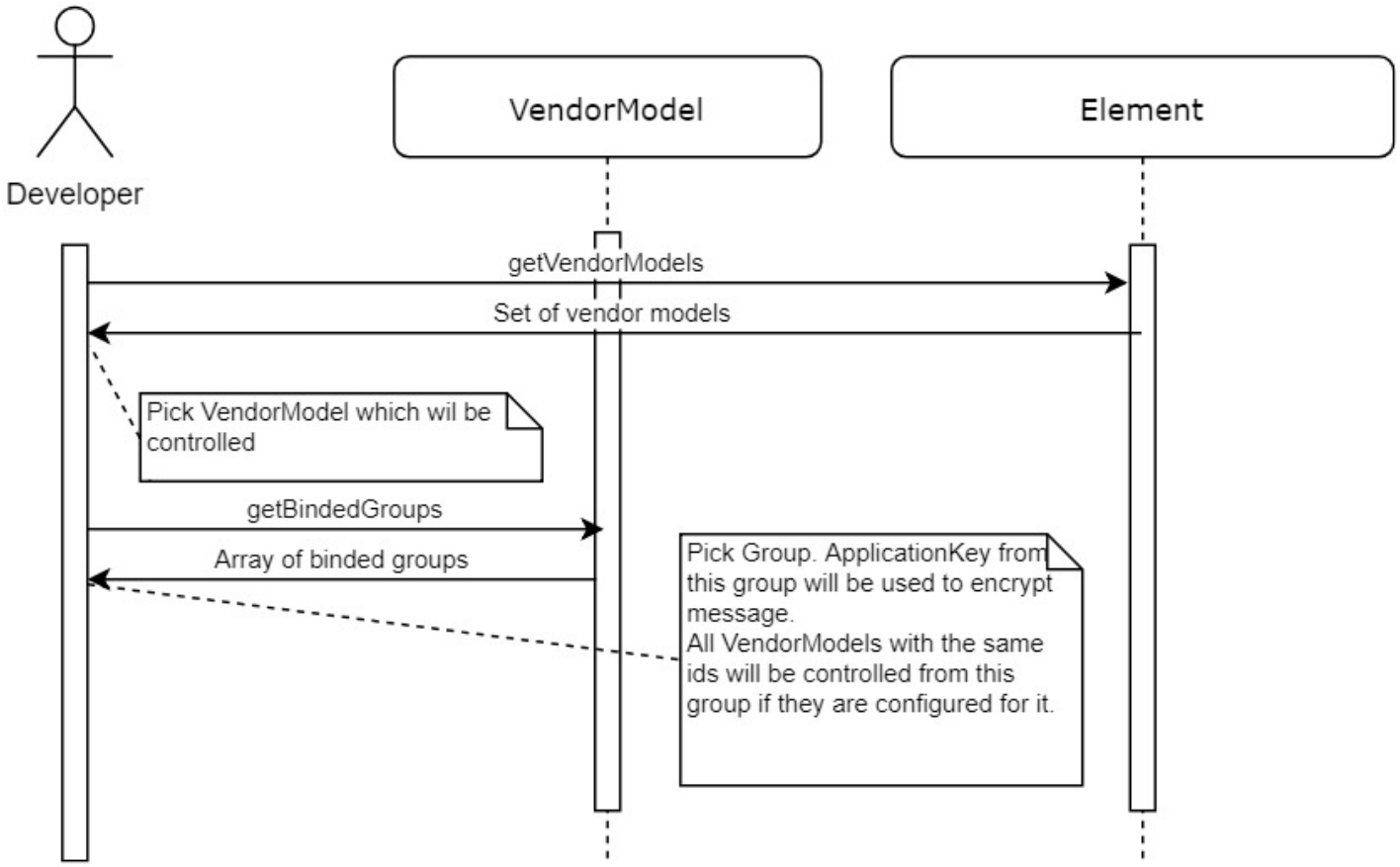
8.24.3.1.2 Message with Response

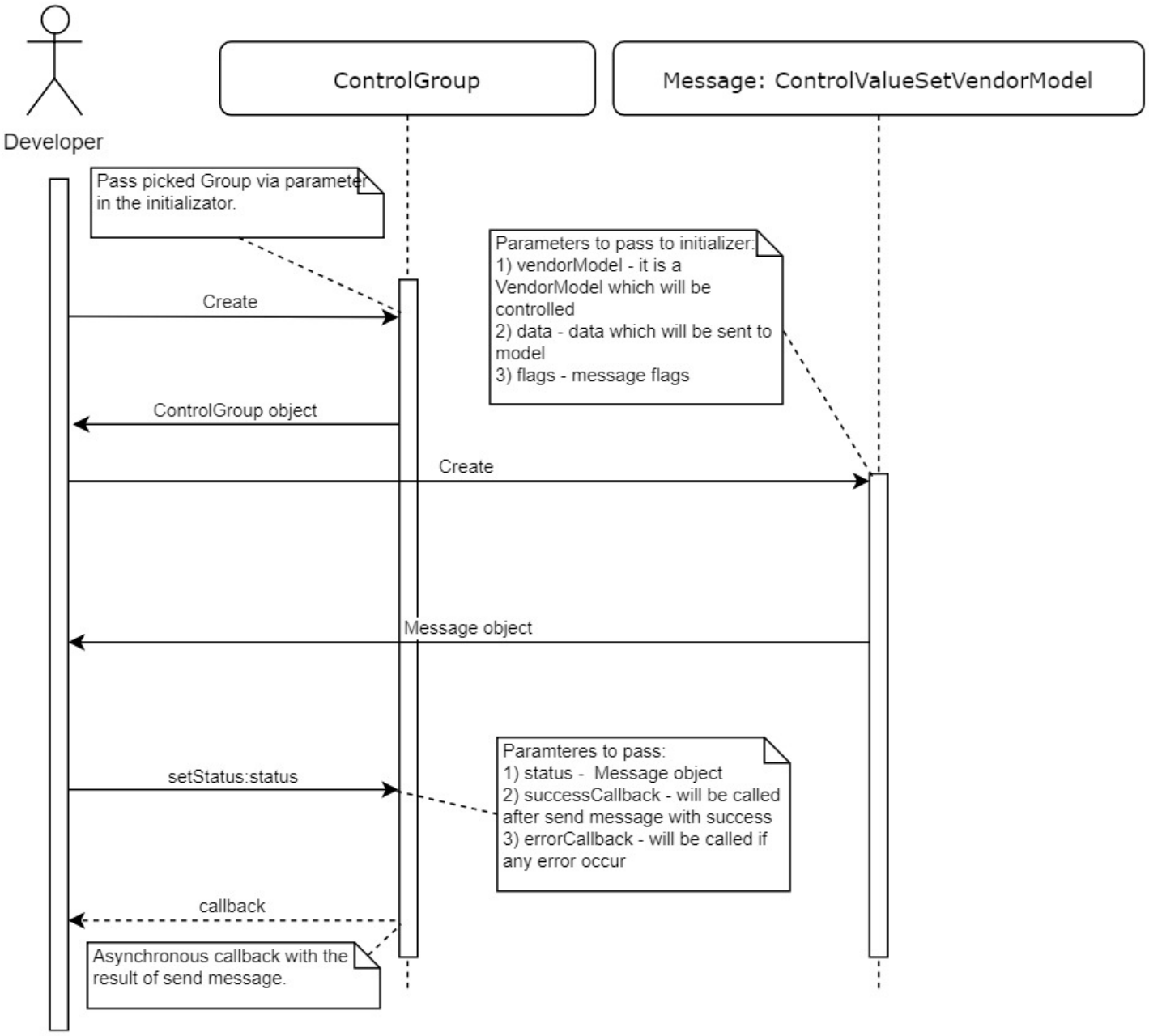



Developer
ControlElement
Message: ControlValueSetVendorModel
LocalVendorSettingsMessageHandler
Create ControlElement
Pass picked Group and Element via parapteres to initializator.
ControlElement object
Create
Parameters to pass to initializer:
1) vendorModel - it is a VendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags
Message object
setStatus:
Paramteres 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
RX message which is a result for TX message which was sent to the VendorModel. LocalVendorSettingsMessageHandler was created during LocalVendorModel configuration by developer.
callback
Asynchronous callback with the result of send message.
callback

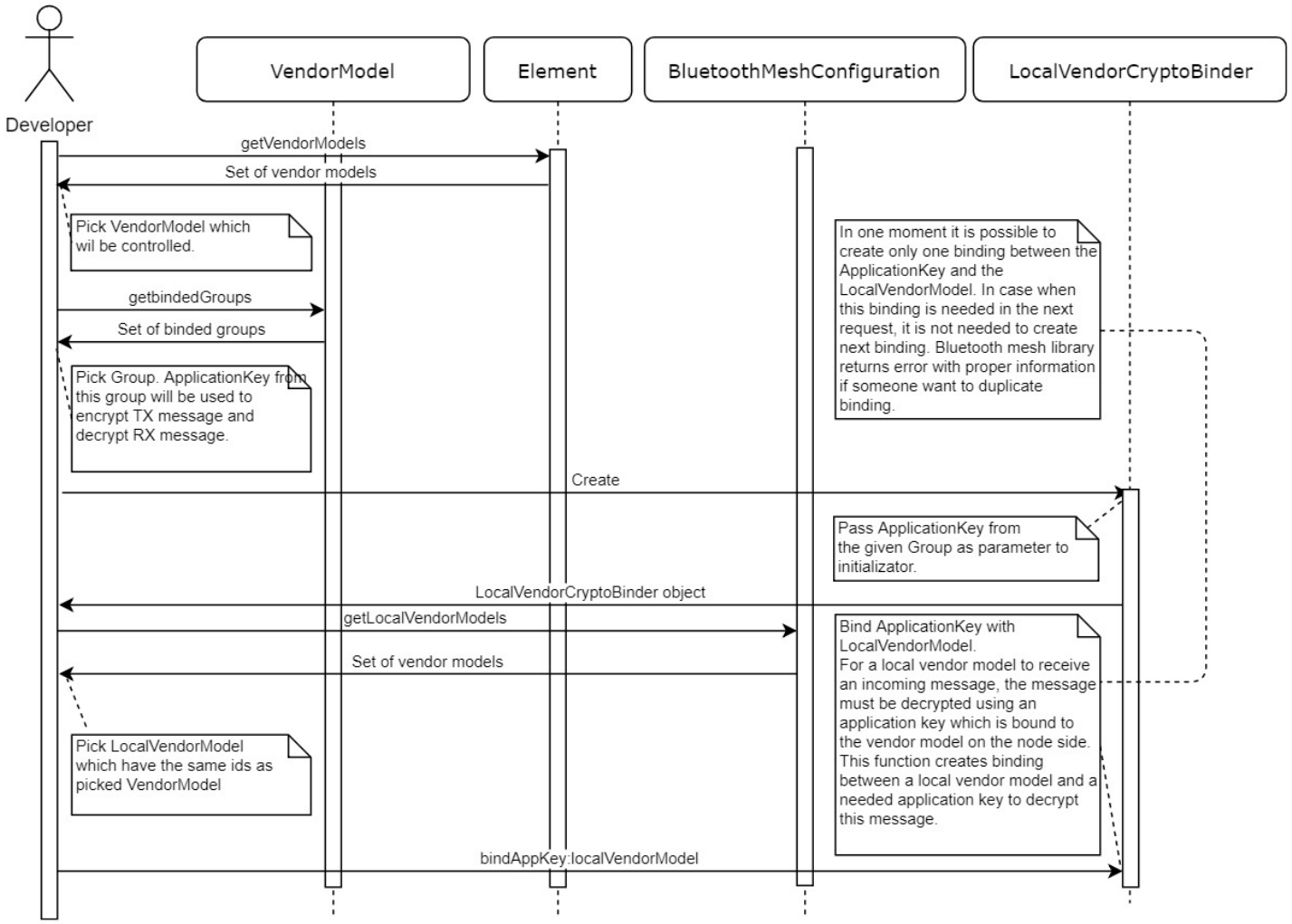
8.24.3.2 Send Message via Group Address

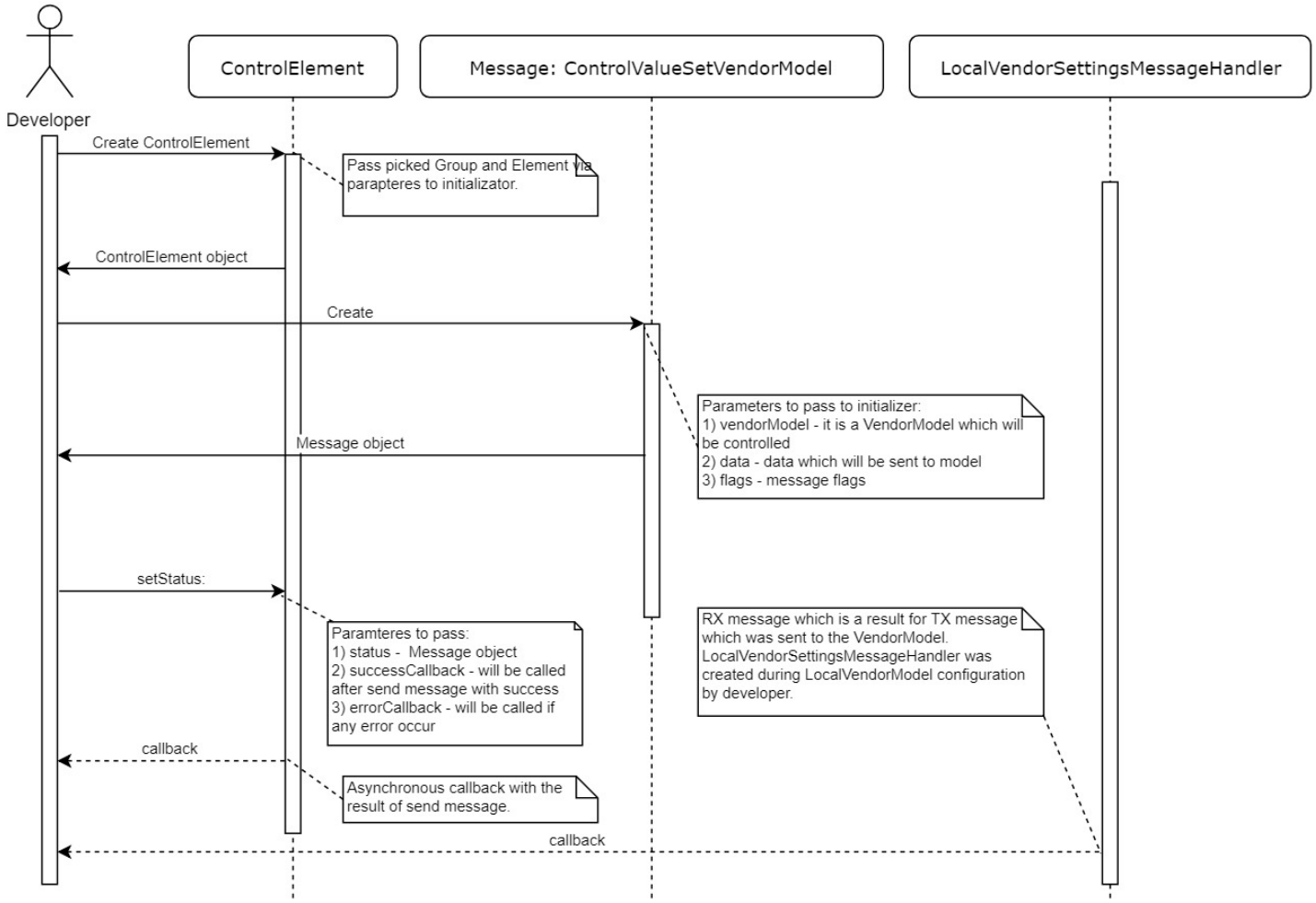
8.24.3.2.1 Message without Response




Developer
ControlGroup
Message: ControlValueSetVendorModel
Pass picked Group via parameter in the initializator.
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
ControlGroup object
Create
Message object
Paramteres 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
setStatus:status
callback
Asynchronous callback with the result of send message.

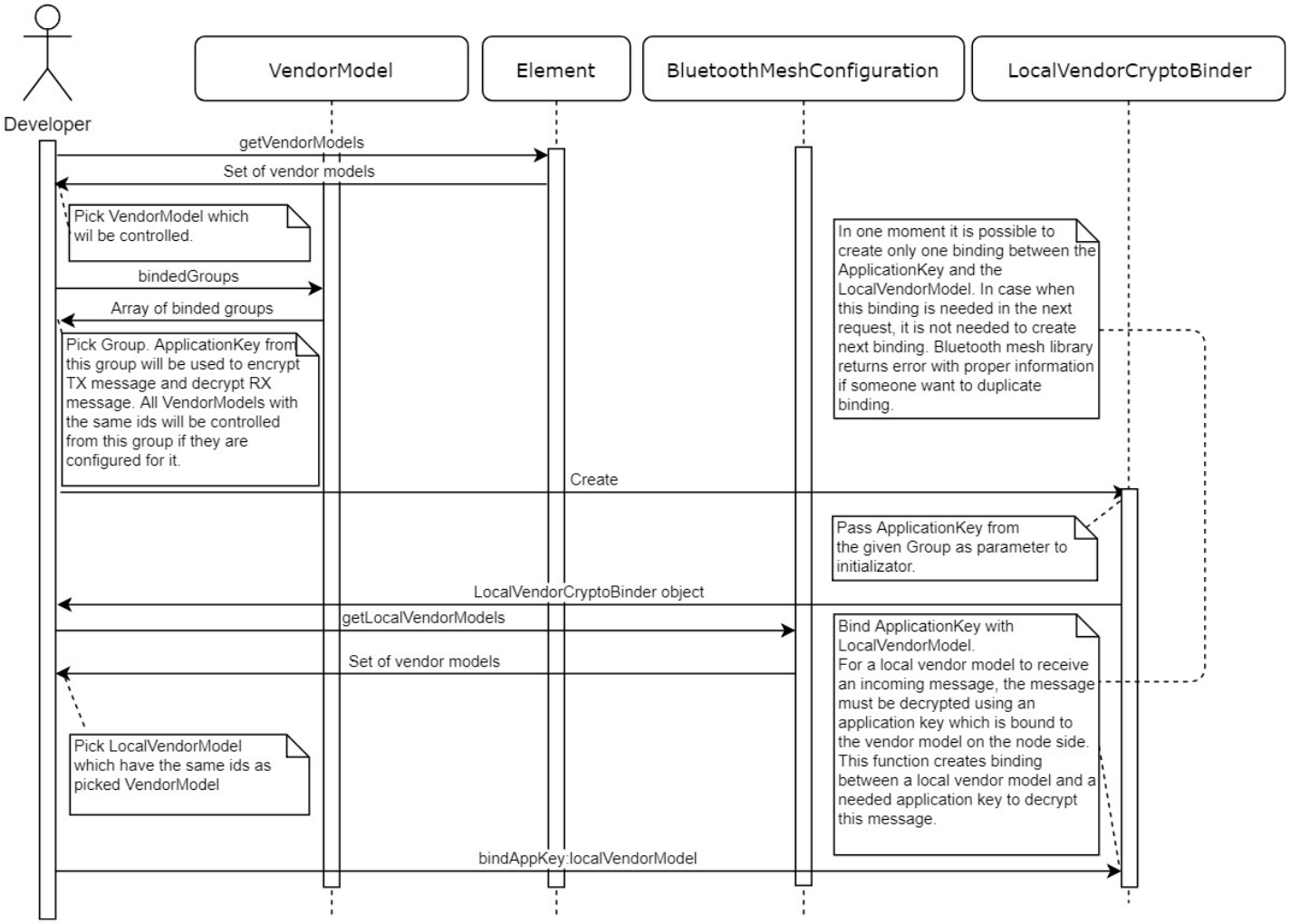
8.24.3.2.2 Message with Response

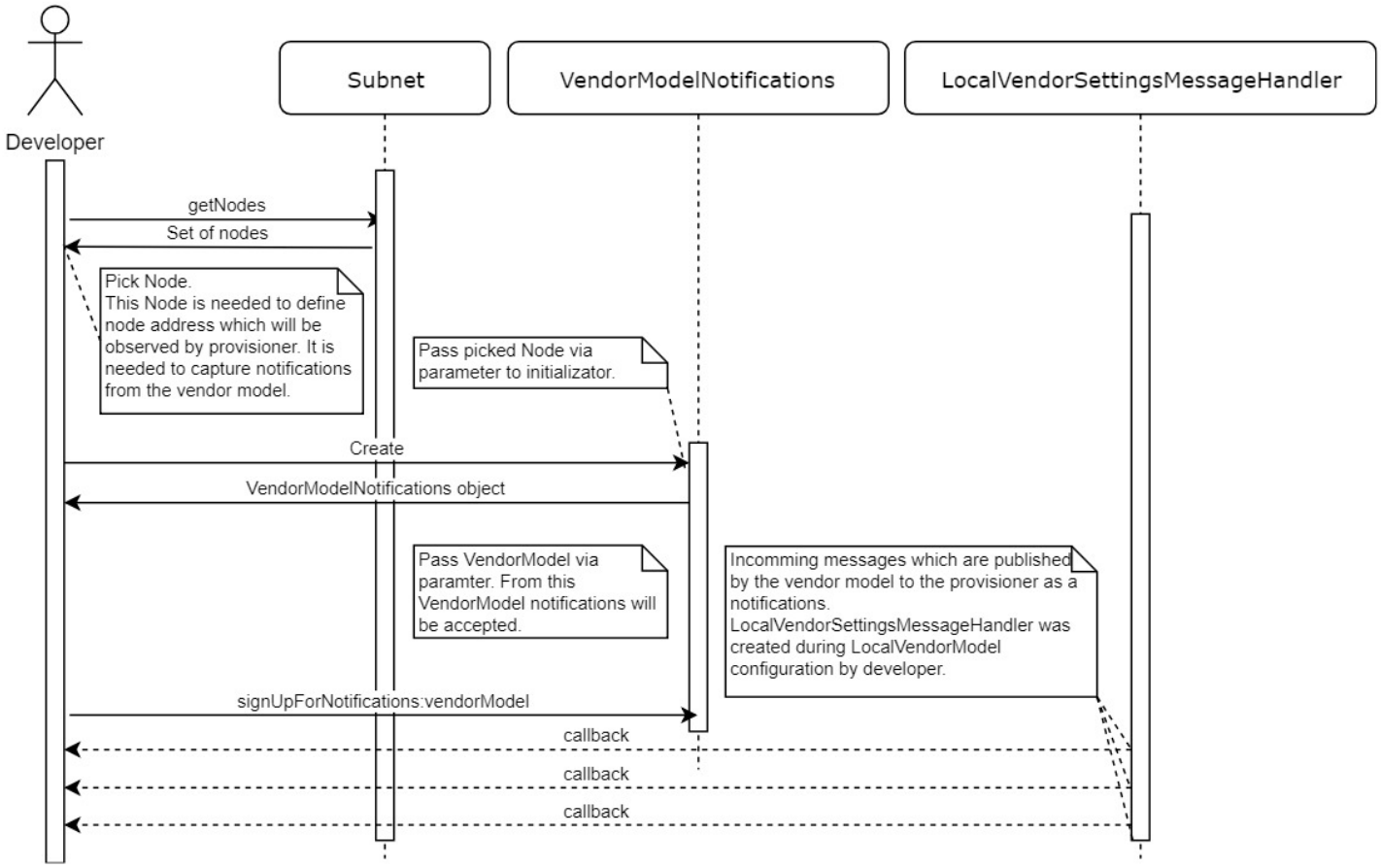



Developer
ControlElement
Message: ControlValueSetVendorModel
LocalVendorSettingsMessageHandler
Create ControlElement
Pass picked Group and Element via parapteres to initializator.
ControlElement object
Create
Parameters to pass to initializer:
1) vendorModel - it is a VendorModel which will be controlled
2) data - data which will be sent to model
3) flags - message flags
Message object
setStatus:
Paramteres 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
RX message which is a result for TX message which was sent to the VendorModel. LocalVendorSettingsMessageHandler was created during LocalVendorModel configuration by developer.
callback
Asynchronous callback with the result of send message.
callback

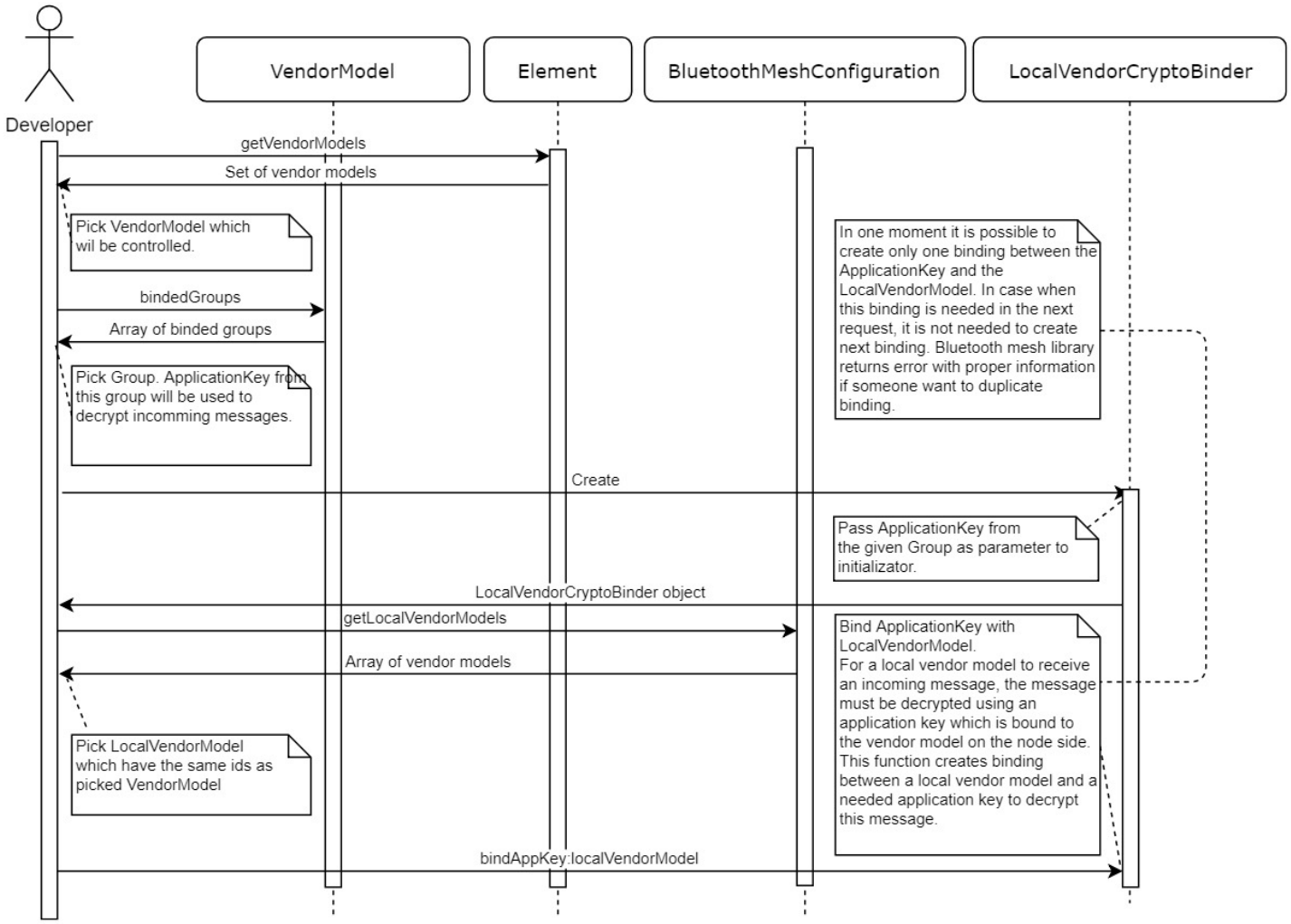
8.24.4 Subscribe to Notifications Published by the Vendor Model From the Node

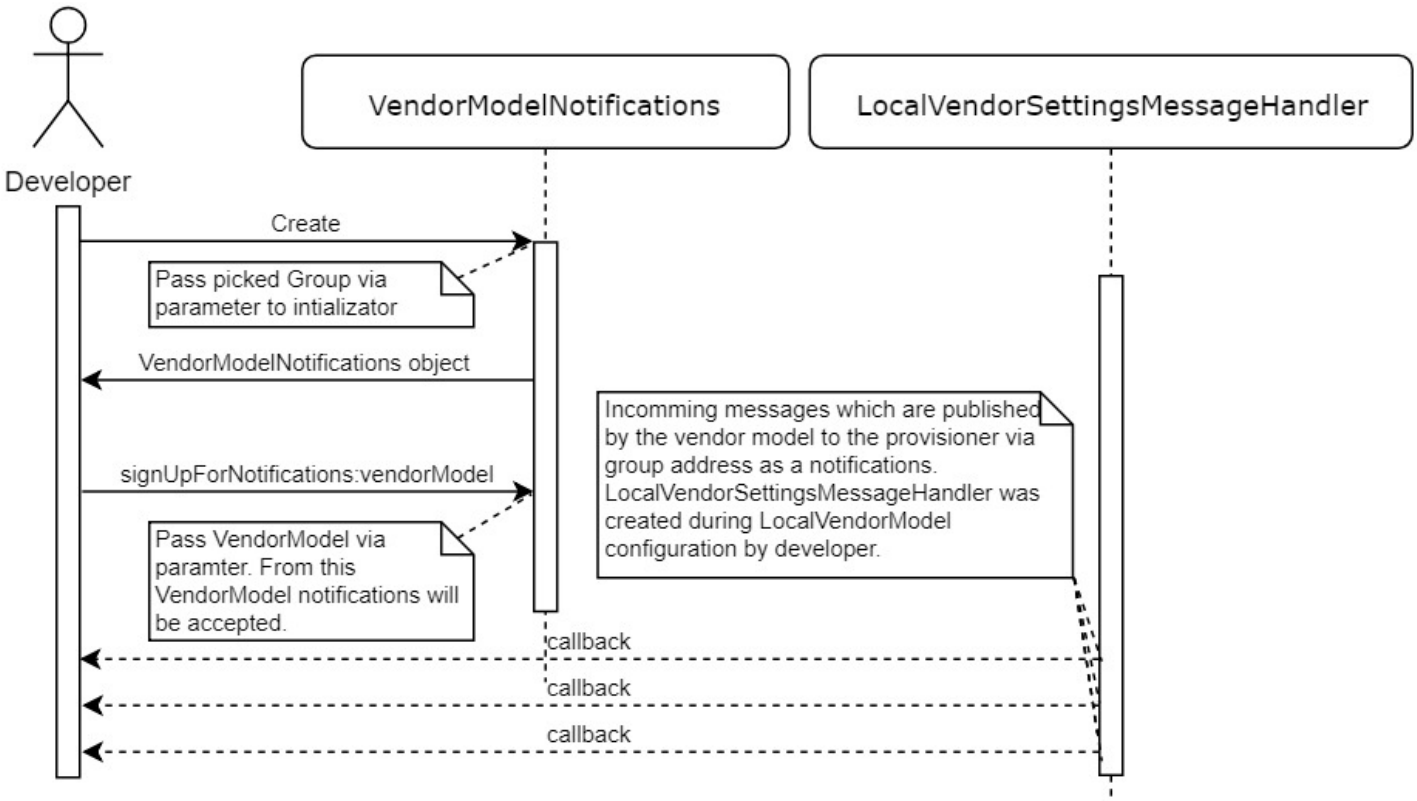
8.24.4.1 Messages are Sent Directly to the Provisioner




Developer
Subnet
VendorModelNotifications
LocalVendorSettingsMessageHandler
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.
Pass picked Node via parameter to initializator.
Create
VendorModelNotifications object
Pass VendorModel via paramter. From this VendorModel notifications will be accepted.
Incomming messages which are published by the vendor model to the provisioner as a notifications.
LocalVendorSettingsMessageHandler was created during LocalVendorModel configuration by developer.
signUpForNotifications:vendorModel
callback
callback
callback

8.24.4.2 Messages are Sent via Group Address to the Provisioner





8.25 Helper Method

8.25.1 Check if Advertisement Data Matches a Net Key

A class ConnectableDeviceHelper provides a method doesNetKeyMatch which checks if received advertisement data matches a provided net key. The method returns true if it matches and Identification Type of advertising is Network ID type (see section 7.2.2.2.1 Advertising from *Mesh Profile Bluetooth® Specification*).

8.26 Heartbeat Messages

Heartbeat messages are sent periodically by nodes that have it enabled in the configuration. These messages can be used to monitor nodes on a network and discover how far nodes are apart from each other. For more information see section 3.6.7 Heartbeat of the Mesh Profile Specification.

Note that it is important not to set too many publications, because it can cause a congestion of the mesh.

In order to make a node send Heartbeat messages to a given `destinationAddress` (unicast or group address), the Heartbeat publication must be configured.

```
HeartbeatPublication configuration = new HeartbeatPublication(destinationAddress, countLogarithm,
periodLogarithm, ttl, features, netKeyIndex);
configurationControl.setHeartbeatPublication(configuration, new HeartbeatPublicationCallback() {
    @Override
    public void success(@NonNull Node node, @NonNull HeartbeatPublication state) {
        Log.d(TAG, "Heartbeat Publication set for " + node + " to " + state);
    }

    @Override
    public void error(@NonNull Node node, @NonNull ErrorType error) {
        Log.d(TAG, "Error setting Heartbeat Publication for " + node + ": " + error);    }
});
```

To receive Heartbeat messages sent from `sourceAddress` to `destinationAddress` for a time given by `periodLogarithm`, the Heartbeat Subscription must be configured.

```
configurationControl.setHeartbeatSubscription(sourceAddress, destinationAddress, periodLogarithm,
        new HeartbeatSubscriptionCallback() {
            @Override
            public void success(@NonNull Node node, @NonNull HeartbeatSubscription state) {
                Log.d(TAG, "Heartbeat Subscription set for " + node + " to " + state);
            }

            @Override
            public void error(@NonNull Node node, @NonNull ErrorType error) {
                Log.d(TAG, "Error setting Heartbeat Subscription for " + node + ": " + error);
            }
        });
```

Heartbeat messages can be received on a mobile device by changing server configuration.

```
serverConfigurationControl.setHeartbeatSubscription(sourceAddress, destinationAddress,
        periodLogarithm, new HeartbeatHandler() {
            @Override
            public void receive(int sourceAddress, int destinationAddress, int hops) {
                Log.d(TAG, "Received a Heartbeat from " + sourceAddress +
                        " to " + destinationAddress + " that travelled " + hops + " hops");
            }
        });
```

Local subscription to the Heartbeat can be stopped by calling:

```
serverConfigurationControl.clearHeartbeatSubscription();
```

9 Models

9.1 Time Models

Time models are used to make mesh devices time-aware and to propagate time information to neighbor nodes. These models enable setting or getting current time, time zone, the difference between TAI and UTC time, and the role of a node in time propagation.

Time models are based on International Atomic Time (TAI), which is further described in Mesh Model Specification paragraph 5.1 and appendix A.1.

The Time API allows for:

- Get or set current time
- Get or set time zone
- Get or set TAI-UTC delta
- Get or set time role

Note: Time models do not support set requests and getTimeRole requests sent to group addresses. Those must be sent to an element address. Other requests can be sent to either element or group addresses.

All set requests to time setup models are acknowledged. That means every set request should result in receiving a corresponding status object if the operation was successful.

9.1.1 Get Time Model Values

9.1.1.1 Get Time From the Node

The request has no parameters. In response, a Time Status object is received. Its structure is presented in section 9.1.3.1 Time Status.

Android:

```
void getTime(Element element, Group group) {
       TimeGet request = new TimeGet();
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getTimeValue(request, new TimeElementCallback<TimeStatus>() {
              @Override
              public void success(Element element, Group group, TimeStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
func getTime(from element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    let request = SBMTimeGet()

    controlElement.getTimeControlResponse(request, successCallback: { _, response in
        let response = response as! SBMTimeStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.1.1.2 Get Time Zone from the Node

This request has no parameters. In response, a Time Zone Status object is received. Its structure is presented in section 9.1.3.2 Time Zone Status.

Android:

```
void getTimeZone(Element element, Group group) {
       TimeZoneGet request = new TimeZoneGet();
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getTimeValue(request, new TimeElementCallback<TimeZoneStatus>() {
              @Override
              public void success(Element element, Group group, TimeZoneStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
func getTimeZone(from element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    let request = SBMTimeZoneGet()

    controlElement.getTimeControlResponse(request, successCallback: { _, response in
        let response = response as! SBMTimeZoneStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.1.1.3 Get TAI-UTC Delta from the Node

This request has no parameters. In response, a Time TAI-UTC Delta Status object is received. Its structure is presented in section 9.1.3.3 Time TAI-UTC Delta Status.

Android:

```
void getTimeTaiUtcDelta(Element element, Group group) {
       TimeTaiUtcDeltaGet request = new TimeTaiUtcDeltaGet();
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getTimeValue(request, new TimeElementCallback<TimeTaiUtcDeltaStatus>() {
              @Override
              public void success(Element element, Group group, TimeTaiUtcDeltaStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
func getTimeTaiUtcDelta(from element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    let request = SBMTimeTaiUtcDeltaGet()

    controlElement.getTimeControlResponse(request, successCallback: { _, response in
        let response = response as! SBMTimeTaiUtcDeltaStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.1.1.4 Get Time Role from the Node

This request has no parameters. In response, a Time Role Status object is received. Its structure is presented in section 9.1.3.4 Time Role Status.

Android:

```
void getTimeRole(Element element, Group group) {
       TimeRoleGet request = new TimeRoleGet();
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getTimeValue(request, new TimeElementCallback<TimeRoleStatus>() {
              @Override
              public void success(Element element, Group group, TimeRoleStatus status) {
                            //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                            //implementation
              }
       });
}
```

iOS:

```
func getTimeRole(from element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    let request = SBMTimeRoleGet()

    controlElement.getTimeControlResponse(request, successCallback: { _, response in
        let response = response as! SBMTimeRoleStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.1.2 Set Time Setup Model Values

9.1.2.1 Set Time for the Node

This request has six parameters:

- taiSeconds - The current TAI time in seconds
- subsecond - The sub-second time in units of 1/256th second
- uncertainty - The estimated uncertainty in 10-milisecond steps
- timeAuthority - The element has or does not have a reliable source of TAI
- taiUtcDelta - Current difference between TAI and UTC in seconds
- timeZoneOffset - The local time zone offset in 15-minute increments

If this operation is successful, a Time Status object is received from the node. Its structure is presented in section 9.1.3.1 Time Status.

Android:

```
TimeSet request = new TimeSet(taiSeconds, subseconds, uncertainty, timeAuthority, taiUtcDelta,
timeZoneOffset);
… _
void setTime(Element element, Group group, TimeSet request) {
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.setTimeValue(request, new TimeElementCallback<TimeStatus>() {
              @Override
              public void success(Element element, Group group, TimeStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
let request = SBMTimeSet(taiSeconds: seconds, subseconds: subseconds, uncertainty: uncertainty,
timeAuthority: isTimeAuthority, taiUtcDelta: utcDelta, timeZoneOffset: timeZoneOffset)
...
func setTime(request: SBMTimeSet, for element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.setTimeControlValue(request, successCallback: { (element, response) in
        let response = response as! SBMTimeStatus
        // Handle response
    }) { (element, error) in
        // Handle failure
    }
}
```

9.1.2.2 Set Time Zone for the Node

This request has two parameters:

- timeZoneOffsetNew - The new (i.e. upcoming Time Zone change) zone offset in 15-minute increments. This variable can take values from -64 to 191, which converts to a range from -16 to 47.75 hours
- taiOfZoneChange - TAI seconds time of the upcoming Time Zone Offset change

If this operation is successful, a Time Zone Status object is received from the node. Its structure is presented in section 9.1.3.2 Time Zone Status.

Android:

```
TimeZoneSet request = new TimeZoneSet(timeZoneOffsetNew, taiOfZoneChange);
… _
void setTimeZone(Element element, Group group, TimeZoneSet request) {
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.setTimeValue(request, new TimeElementCallback<TimeZoneStatus>() {
              @Override
              public void success(Element element, Group group, TimeZoneStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
let request = SBMTimeZoneSet(timeZoneOffSetNew: offsetNew, taiOfZoneChange: taiChange)
...
func setTimeZone(request: SBMTimeZoneSet, for element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.setTimeControlValue(request, successCallback: { (element, response) in
        let response = response as! SBMTimeZoneStatus
        // Handle response
    }) { (element, error) in
        // Handle failure
    }
}
```

9.1.2.3 Set TAI-UTC delta for the node

This request has two parameters:

- taiUtcDeltaNew - Upcoming difference between TAI and UTC in seconds
- taiOfDeltaChange - TAI seconds time of the upcoming TAI-UTC Delta change

If this operation is successful, a Time TAI-UTC Delta Status object is received from the node. Its structure is presented in section 9.1.3.3 Time TAI-UTC Delta Status.

Android:

```
TimeTaiUtcDeltaSet request = new TimeTaiUtcDeltaSet(taiUtcDeltaNew, taiOfDeltaChange);
… _
void setTimeTaiUtcDelta(Element element, Group group, TimeTaiUtcDeltaSet request) {
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.setTimeValue(request, new TimeElementCallback<TimeTaiUtcDeltaStatus>() {
              @Override
              public void success(Element element, Group group, TimeTaiUtcDeltaStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
let request = SBMTimeTaiUtcDeltaSet(taiUtcDeltaNew: deltaNew, taiOfDeltaChange: deltaChange)
...
func setTimeTaiUtcDelta(request: SBMTimeTaiUtcDeltaSet, for element: SBMElement, in group:
SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.setTimeControlValue(request, successCallback: { (element, response) in
        let response = response as! SBMTimeTaiUtcDeltaStatus
        // Handle response
    }) { (element, error) in
        // Handle failure
    }
}
```

9.1.2.4 Set Time Role for the Node

This request has one parameter:

- timeRole - Time role for the element. This parameter can have one of four defined values:
  - SBMTimeRoleNone - Element does not participate in propagation of time information
  - SBMTimeRoleTimeAuthority - Element publishes Time Status messages, but does not process received Time Status messages
  - SBMTimeRoleTimeRelay - Element processes received and publishes Time Status messages
  - SBMTimeRoleTimeClient - Element does not publish but processes received Time Status messages

If this operation is successful, a Time Role Status object is received from the node. Its structure is presented in section 9.1.3.4 Time Role Status.

Android:

```
TimeRoleSet request = new TimeRole(timeRole);
void setTimeRole(Element element, Group group, TimeRoleSet request) {
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.setTimeValue(request, new TimeElementCallback<TimeRoleStatus>() {
              @Override
              public void success(Element element, Group group, TimeRoleStatus status) {
                           //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                           //implementation
              }
       });
}
```

iOS:

```
let request = SBMTimeRoleSet(timeRole: .timeAuthority)
...
func setTimeRole(request: SBMTimeRoleSet, for element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)

    controlElement.setTimeControlValue(request, successCallback: { (element, response) in
        let response = response as! SBMTimeRoleStatus
        // Handle response
    }) { (element, error) in
        // Handle failure
    }
}
```

9.1.3 Time Status Messages

9.1.3.1 Time Status

This object represents the current Time state of an element. It has six properties:

- taiSeconds - The current TAI time in seconds. If this value is equal to 0, other properties do not represent valid values and should be ignored.
- subsecond - The sub-second time in units of 1/256th second
- uncertainty - The estimated uncertainty in 10-milisecond steps
- timeAuthority - The element has or does not have a reliable source of TAI
- taiUtcDelta - Current difference between TAI and UTC in seconds
- timeZoneOffset - The local time zone offset in 15-minute increments

9.1.3.2 Time Zone Status

This object represents the current Time Zone state of an element. It has three properties:

- timeZoneOffsetCurrent - The current zone offset in 15-minute increments. Positive numbers are eastwards from UTC. This variable takes values from -64 to 191, which converts to a range from -16 to 47.75 hours.
- timeZoneOffsetNew - The new (i.e. upcoming Time Zone change) zone offset in 15-minute increments. Its format is identical to the format of property timeZoneOffsetCurrent.
- taiOfZoneChange - The time (in TAI Seconds format) which indicates when the Time Zone offset will be applied. Value 0 means that the Time Server does not contain this information.

9.1.3.3 Time TAI-UTC Delta Status

This object represents the current TAI-UTC delta state of an element. It has three properties:

- taiUtcDeltaCurrent - Current difference between TAI and UTC in seconds
- taiUtcDeltaNew - Upcoming difference between TAI and UTC in seconds
- taiOfDeltaChange - TAI Seconds time of the upcoming TAI-UTC Delta change

9.1.3.4 Time Role Status

This object represents the current Time Role of an element. It has one property:

- timeRole - Time role for the element. This parameter can have one of four defined values:
  - SBMTimeRoleNone - Element does not participate in propagation of time information
  - SBMTimeRoleTimeAuthority - Element publishes Time Status messages, but does not process received Time Status messages
  - SBMTimeRoleTimeRelay - Element processes received and publishes Time Status messages
  - SBMTimeRoleTimeClient - Element does not publish but processes received Time Status messages

9.1.4 Group Messages

Requests can be sent to group addresses, however only the Time Server model is able to process them, since it allows publication and subscription. This means that all requests received by the Time Setup Server cannot be sent to group addresses. Based on chapter 5.3.2.1 of Mesh Model Documentation these messages are:

- Time Set
- Time Zone Set
- Time TAI-UTC Delta Set
- Time Role Set
- Time Role Get

Statuses from these messages can be retrieved for a subgroup of devices instead of the whole group, as shown below.

Android:

```
void getTimeForGroup(Group group) {
        TimeGet request = new TimeGet();
        ControlGroup controlGroup = new ControlGroup(group);

        controlGroup.getTimeValue(request, new TimeGroupHandler<TimeStatus>() {
                @Override
                public void success(Element element, Group group, TimeStatus status) {
                        //implementation
                }
                @Override
                public void error(Group group, ErrorType error) {
                        //implementation
                }
        });
}
```

iOS:

```
func getGroupTime(from group: SBMGroup) {
    let controlGroup = SBMControlGroup(group: group)
    let request = SBMTimeGet()

    controlGroup.getTimeControlResponse(request, successHandler: { (group, response, element) in
        let response = response as! SBMTimeStatus
        // Handle response
    }) { (group, error) in
        // Handle failure
    }
}
```

9.1.5 Subscribing to Publications

9.1.5.1 Publications from a Single Node

The subscription takes one parameter that describes the status to be notified about. This object is the same as when manually requesting a single status report.

The success handler is executed every time that a subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object contains its current status.

The error handler is executed either if the subscription request fails or the node reports any error.

After subscription, a callback is called for every received notification until manually unsubscribed from the notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

Android:

```
void subscribeToTimeZoneGet(Element element, Group group) {
       TimeZoneGet request = new TimeZoneGet();
       ControlElementPublications elementPublications = new ControlElementPublications(element,
group);

        MeshTask meshTask = elementPublications.subscribe(
                new TimeZoneGet(),
                new TimeElementPublicationHandler<TimeZoneStatus>() {
                    @Override
                    public void success(Element element, Group group, TimeZoneStatus status) {
                        //implementation
                    }

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

iOS:

```
func subscribeToTime(from element: SBMElement, in group: SBMGroup) {
    let controlElementPublications = SBMControlElementPublications(element: element, in: group)
    let request = SBMTimeGet()

    controlElementPublications.subscribe(toTimeStatus: request, successHandler: { (element, group,
response) in
        let response = response as! SBMTimeStatus
        // Handle response
    }) { (element, group, error) in
        // Handle failure
    }
}
```

9.1.5.2 Group Publications

Since Time Setup Server model does not support subscribing or publishing, some messages cannot be subscribed to. They are listed in section 9.1.4 Group Messages.

For the rest of the messages, group subscription is similar to subscription to a specific node. The only difference is that the subscribe method from `ControlGroupPublications` should be used instead of `ControlElementPublications` and that this object's constructor takes only a group identifier.

After subscription, the callback is called for every received notification until manually unsubscribed from this notification. To unsubscribe, call the `cancel()` method on the task object returned after subscription.

Android:

```
void subscribeToGroupTimeGet(Group group) {
       TimeGet request = new TimeGet();
       ControlGroupPublications groupPublications = new ControlGroupPublications(group);

       MeshTask meshTask = groupPublications.subscribe(request,
       new TimeGroupPublicationHandler<TimeStatus>() {
           @Override
           public void success(Element element, Group group, TimeStatus status) {
               //implementation
           }

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

iOS:

```
func subscribeToGroupTime(from group: SBMGroup) {
    let controlGroupPublications = SBMControlGroupPublications(group: group)
    let request = SBMTimeGet()

    controlGroupPublications.subscribe(toTimeStatus: request, successHandler: { (element, group,
response) in
        let response = response as! SBMTimeStatus
        // Handle response
    }) { (element, group, error) in
        // Handle failure
    }
}
```

9.2 Scheduler Models

Scheduler models are used to join Time and Scene model functions. It provides a method for autonomous change of states on a node.

The Scheduler API allows for:

- Get current state of Schedule Register
- Get or set entry in Schedule Register

9.2.1 Get Scheduler Model Values

9.2.1.1 Get Schedule Register from the Node

This request has no parameters. In response, a Scheduler Status object is received. Its structure is presented in section 9.2.3.1 Scheduler Status.

Android:

```
void getScheduleRegister(Element element, Group group) {
       SchedulerGet request = new SchedulerGet();
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getScheduleRegister(request, new SchedulerElementCallback<SchedulerStatus>() {
              @Override
              public void success(Element element, Group group, SchedulerStatus status) {
                     //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                     //implementation
              }
       });
}
```

iOS:

```
func getScheduler(for element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)
    let request = SBMSchedulerGet()

    controlElement.getScheduleRegister(request, successCallback: { _, response in
        let response = response as! SBMSchedulerStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.2.1.2 Get Scheduler Action from the Node

This request to get the action defined by the entry of the schedule register has one parameter:

- index – Specifies which entry of the schedule register should be received. The Schedule Register consist of 16 entries, starting from 0, so this value must be within the range of 0 to 15 inclusive.

In response, a Scheduler Action Status object should be received. Its structure is presented in section 9.2.3.2 Scheduler Action Status.

Android:

```
void getSchedulerAction(Element element, Group group, int index) {
       SchedulerActionGet request = new SchedulerActionGet(index);
       ControlElement controlElement = new ControlElement(element, group);

       controlElement.getScheduleRegister(request, new
SchedulerElementCallback<SchedulerActionStatus>() {
              @Override
              public void success(Element element, Group group, SchedulerActionStatus status) {
                     //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                     //implementation
              }
       });
}
```

iOS:

```
func getSchedulerAction(index: Int, for element: SBMElement, in group: SBMGroup) {
    let controlElement = SBMControlElement(element: element, in: group)

    let request = SBMSchedulerActionGet(index: UInt8(index))

    controlElement.getScheduleRegister(request, successCallback: { _, response in
        let response = response as! SBMSchedulerActionStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.2.2 Set Scheduler Setup Model Values

9.2.2.1 Set Scheduler Action for the Node

This request to set the action defined in the entry of the schedule register has three parameters:

- (Android and iOS) flags – The object whose property defines if this request should be acknowledged
- (Android and iOS) index – Value defining which entry in the schedule register should be set with the values in this request
- (iOS only) scheduleRegister – the object containing properties that should be assigned to the entry specified by the index property, in the schedule register of the element. Its structure is defined in section 9.2.4 Schedule Register Object.

If this request is acknowledged and it was successful, a Scheduler Action Status object is received from the node. Its structure is defined in section 9.2.3.2 Scheduler Action Status.

Android:

```
SchedulerActionSet prepareSchedulerActionSet(boolean isAcknowledged, Integer index, Integer year,
Set<Month> months, Integer day, Integer hour, Integer minute, Integer second, Set<DayOfWeek>
daysOfWeek, Action action, Integer transitionTime, Integer scene) {
       SchedulerActionSet request = new SchedulerActionSet();
       SchedulerMessageFlags flags = new SchedulerMessageFlags();
       flags.setAcknowledged(isAcknowledged);
       request.setFlags(flags);
       request.setIndex(index);
       request.setYear(year);
       request.setMonths(months);
       request.setDay(day);
       request.setHour(hour);
       request.setMinute(minute);
       request.setSecond(second);
       request.setDaysOfWeek(daysOfWeek);
       request.setAction(action);
       request.setTransitionTime(transitionTime);
       request.setScene(scene);
       return request;
}
… _
void setSchedulerAction(Element element, Group group, SchedulerActionSet request()) {
       ControlElement controlElement = new ControlElement(element, group);
       controlElement.setScheduleRegister(request, new SchedulerElementCallback<SchedulerActionSta-
       tus>() {
              @Override
              public void success(Element element, Group group, SchedulerActionStatus status) {
                     //implementation
              }
              @Override
              public void error(Element element, Group group, ErrorType errorType) {
                     //implementation
              }
       });

}
```

iOS:

```
func setSchedulerAction(index: Int, register: SBMScheduleRegister, for element: SBMElement, in
group: SBMGroup, acknowledged: Bool) {
    let controlElement = SBMControlElement(element: element, in: group)

    let flags = SBMSchedulerMessageFlags()
    flags.isAcknowledgedRequired = acknowledged
    let request = SBMSchedulerActionSet(flags: flags, index: UInt8(index), scheduleRegister:
register)

    controlElement.setScheduleRegister(request, successCallback: { _, response in
        let response = response as! SBMSchedulerActionStatus
        // Handle response
    }, errorCallback: { _, error in
        // Handle failure
    })
}
```

9.2.3 Scheduler Status Messages

9.2.3.1 Scheduler Status

This object represents the current Schedule Register state of an element. It has one property:

schedules – an array containing 16 objects. Each object is NSNumber with bool value that determines the state of an entry in the schedule register. True means that an action is set in the specified entry. False means no action is set. The index in this array corresponds to the index of entries in the schedule register.

9.2.3.2 Scheduler Action Status

This object represents the state of an entry in the Schedule Register within an element. It has two properties:

- index – Identifies which entry in the schedule register this object represents
- scheduleRegister – The object that represents fields defined by the entry in the schedule register. Its structure is defined in section 9.2.4 Schedule Register Object.

9.2.4 Schedule Register Object

A schedule register entry is represented by a dedicated object in iOS. In Android it is represented by fields placed directly in status and request messages.

This object is used by Scheduler Action Set and Scheduler Action Status messages. It represents an entry in Schedule Register. It has 10 properties, which are described differently depending on the platform.

9.2.4.1 iOS - Schedule Register as its Own Object:

- year – (type SBMScheduleRegisterYear) A specific year or a predefined value such as every year. If an action should happen in a specific year, the two least significant digits of the year should be set. Otherwise one of the other predefined value should be used.
- months – (an array of SBMScheduleRegisterMonth objects) Each object represents one month when the action should happen. Each object can be created using predefined enum values, or integer values, where 1 means January, and 12 means December.
- day – (type SBMScheduleRegisterDay) A specific day of month or a predefined value such as every day.
- hour – (type SBMScheduleRegisterHour) A specific hour or a predefined value such as every hour or random hour.
- minute – (type SBMScheduleRegisterMinute) A specific minute of an hour or a predefined value such as every minute, every 15 minutes, every 20 minutes, or a random minute.
- second – (type SBMScheduleRegisterSecond) A specific second of a minute or a predefined value such as every second, every 15 seconds, every 20 seconds, or a random second.

- daysOfWeek – (an array of SBMScheduleRegisterDayOfWeek) Each object represents one day of the week in which the action should take place.
- action –(type SBMScheduleRegisterAction) The action that should take place on the date and time specified in this entry. Currently supported actions are turnOff, turnOn, sceneRecall or noAction. No action means that this entry in the schedule register is disabled. Turning on/off is equal to receiving a Generic OnOff message by all elements in the node.
- transitionTime – How long the transition should take, in milliseconds.
- scene –(type SBMScheduleRegisterScene) The scene number that should be recalled if scene recall was set as the action. If another action is set, this value should be set as the predefined value of none.

9.2.4.2 Android – Fields of Schedule Register as a Part of Request and Status Object

- year – this property can mean:
  - a specific year by passing an Integer from range [0; 99]
  - every year by passing enum value Year.EVERY_YEAR
- months – this property is a set of enumerations of type Month. Each one represents one month when the action should happen. If an event should occur during a specific month, add its enum to set
- day – this property can mean:
  - a specific day by passing an Integer from range [1; 31]; If the number of the day is larger than the number of days in the month, the event occurs on the last day of the month
  - every day by passing enum value Day.EVERY_DAY;
- hour – this property can mean:
  - a specific hour by passing an Integer from range [0; 23]
  - every hour by passing enum value Hour.EVERY_HOUR
  - random hour of the day by passing enum value Hour.ANY_HOUR
- minute – this property can mean:
  - a specific minute by passing an Integer from range [0;59]
  - every minute by passing enum value Minute.EVERY_MINUTE
  - every 15 minutes by passing enum value Minute.EVERY_15_MINUTES
  - every 20 minutes by passing enum value Minute.EVERY_20_MINUTES
  - random minute of the hour by passing enum value Minute.ANY_MINUTE
- second – this property can mean:
  - a specific second by passing an Integer from range [0;59]
  - every second by passing enum value Second.EVERY_MINUTE
  - every 15 seconds by passing enum value Second.EVERY_15_SECONDS
  - every 20 seconds by passing enum value Second.EVERY_20_SECONDS
  - random second of the minute by passing enum value Second.ANY_SECOND
- daysOfWeek – this property in a set of enumerations of type Month. Each one of them represents one day of the week in which the action should occur. If an event should occur during a specific month, add its enum to set
- action – this property can be set to one of the values of Action enum. It defines which action should occur when date and time specified in this entry take place. Currently supported actions are turnOff, turnOn, sceneRecall or noAction. No action means that this entry in the schedule register is disabled. Turning on/off is equal to receiving Generic OnOff message by all elements in node.
- transitionTime – this property defines how long the transition should take. This property is in milliseconds stored as Integer
- scene – this property describes the scene number that should be recalled if scene recall was set as action. It can mean:
  - a specific scene by passing an Integer
  - no scene by passing enum value Scene.NO_SCENE

9.2.5 Group messages

Receiving group messages is similar to getting one message from a specific node. The only difference is that the get/set methods from `ControlGroup` should be used instead of `ControlElement` and that this object's constructor takes only a group identifier, as shown below.

Android:

```
void getSchedulerActionForGroup(Group group) {
       SchedulerActionGet request = new SchedulerActionGet(7);
       ControlGroup controlGroup = new ControlGroup(group);

       controlGroup.getScheduleRegister(request, new SchedulerGroupHandler<SchedulerActionStatus>()
{
              @Override
              public void success(Element element, Group group, SchedulerActionStatus status) {
                     //implementation
              }
              @Override
              public void error(Group group, ErrorType errorType) {
                                  //implementation
              }
       });
}
```

iOS:

```
func getScheduler(from group: SBMGroup) {
    let controlGroup = SBMControlGroup(group: group)
    let request = SBMSchedulerGet()

    controlGroup.getScheduleRegister(request, successHandler: { (group, response, element) in
        let response = response as! SBMSchedulerStatus
        // Handle response
    }) { (group, error) in
        // Handle failure
    }
}
```

9.2.6 Subscribing to Publications

9.2.6.1 Subscribing to a Single Node

The subscription takes one parameter that describes the status to be notified about. This object is the same as when manually requesting a single status report.

The success handler is executed every time that a subscribed node publishes its status. Element and group parameters specify which node reported its status, and the response object contains its current status.

The error handler is executed either if the subscription request fails or the node reports any error.

After subscription, a callback will be called for every received notification until manually unsubscribed from the notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

Android:

```
void subscribeToSchedulerGet(Element element, Group group) {
        SchedulerGet request = new SchedulerGet();
         ControlElementPublications elementPublications = new ControlElementPublications(element,
group);

        MeshTask meshTask = elementPublications.subscribe(request,
                new SchedulerElementPublicationCallback<SchedulerStatus>() {
                    @Override
                    public void success(Element element, Group group, SchedulerStatus status) {
                        //implementation
                    }

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

iOS:

```
func subscribeToScheduler(from element: SBMElement, in group: SBMGroup) {
    let controlElementPublications = SBMControlElementPublications(element: element, in: group)
    let request = SBMSchedulerGet()

    controlElementPublications.subscribe(toScheduleRegister: request, successHandler: { (element,
group, response) in
        let response = response as! SBMSchedulerStatus
        // Handle response
    }) { (element, group, error) in
        // Handle failure
    }
}
```

9.2.6.2 Subscribing to a Group

Group subscription is similar to subscription to a specific node. The only difference is that the subscribe method from ControlGroupPublications should be used instead of ControlElementPublications and that this object's constructor takes only a group identifier.

After subscription, the callback will be called for every received notification until manually unsubscribed from this notification. To unsubscribe, the `cancel()` method should be called on the task object returned after subscription.

Android:

```
void subscribeToGroupSchedulerGet(Group group) {
       SchedulerGet request = new SchedulerGet();
        ControlGroupPublications groupPublications = new ControlGroupPublications(group);

       MeshTask meshTask = groupPublications.subscribe(request,
       new SchedulerGroupPublicationHandler<SchedulerStatus>() {
           @Override
           public void success(Element element, Group group, SchedulerStatus status) {
               //implementation
           }

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

iOS:

```
func subscribeToScheduler(group: SBMGroup) {
    let controlGroupPublications = SBMControlGroupPublications(group: group)
    let request = SBMSchedulerGet()

    controlGroupPublications.subscribe(toScheduleRegister: request, successHandler: { (element,
group, response) in
        let response = response as! SBMSchedulerStatus
        // Handle response
    }) { (group, error) in
        // Handle failure
    }
}
```

10 Foundation Models

10.1 Configuration Model

10.1.1 Low Power Node

The ADK 2.3.0 provides a base support for low power node. It is possible to get LPN Poll Timeout status from a node that is a Friend of a Low Power node. The API provides a way to change mesh configuration timeouts for Low Power node messages.

Note: If a Low Power node has yet to establish a friendship with a Friend node, it listens for incoming traffic only when expecting a response to the Friend Request message from the Friend node. This is intentional, as it saves power, but it also makes communication with such a node very unreliable until it establishes a friendship.

10.1.1.1 Poll Timeout

The Poll Timeout Get is an acknowledged message used to get the current value of the Poll Timeout timer of the Low Power node within a Friend node. The message is sent to a Friend node that has claimed to be handling messages of the Low Power node.

The success callback (SBMLpnPollTimeoutSuccessCallback or LpnPollTimeoutCallback#success) is a response to the PollTimeout Get message. It contains the current value of the PollTimeout timer of the Low Power node, a Friend node, and a Low Power node.

The PollTimeout property contains the current value of the PollTimeout timer or 0 if the node is not the Friend node of the Low Power node. The PollTimeout timer value is in units of 100 milliseconds, in a range from 10x100 milliseconds to 3455999x100 milliseconds.

The error callback (SBMLpnPollTimeoutErrorCallback or LpnPollTimeoutCallback#error) is a response to the PollTimeout Get message if an error occurs. It contains a Friend node, a Low Power node, and the error.

iOS:

```
var friendNode: SBMNode
var lowPowerNode: SBMNode
(…)
let configurationControl = SBMConfigurationControl(node: friendNode)
configurationControl.getLpnPollTimeout(lowPowerNode,
successCallback: { (friendNode, lowPowerNode, pollTimeout) in
       //handle success
}, errorCallback: { (friendNode, lowPowerNode, error) in
       //handle error
})
```

Android:

```
Node friendNode;
Node lowPowerNode;
(…)
ConfigurationControl configurationControl = new ConfigurationControl(friendNode);
configurationControl.getLpnPollTimeout(lowPowerNode, new LpnPollTimeoutCallback(){
       @Override
       public void success(Node friendNode, Node lowPowerNode, int pollTimeout) {
       }

       @Override
       public void error(Node friendNode, Node lowPowerNode, ErrorType errorType) {
       }
});
```

10.1.1.2 Timeout for Low Power Node Configuration Request

The Bluetooth Mesh library enables changing the timeout for the Low Power node configuration request. Low Power node and Friend node can have different poll timeouts configured and, depending on it, it is important to set a correct timeout for the Low Power node configuration request. An error from the configuration request will be thrown if the timeout is reached.

Low Power nodes should be handled in an optimal way by the application.

10.1.1.2.1 Get Timeout

The default timeout for Low Power node configuration control request is set to 120 000 milliseconds

Get the default timeout for LPN configuration requests.

iOS:

```
let configurationControlSettings = SBMConfigurationControlSettings()
let timeout = configurationControlSettings.getLpnLocalTimeout()
```

Android:

```
ConfigurationControlSettings configurationControlSettings = new ConfigurationControlSettings();
configurationControlSettings.getLpnLocalTimeout();
```

10.1.1.2.2 Set Timeout

Set the default timeout for LPN configuration requests.

iOS

```
var timeout
(…)
let configurationControlSettings = SBMConfigurationControlSettings()
configurationControlSettings.setLpnLocalTimeout(timeout)
```

Android:

```
int timeout;
(…)
ConfigurationControlSettings configurationControlSettings = new ConfigurationControlSettings();
configurationControlSettings.setLpnLocalTimeout(timeout);
```

11 Export

The Mesh library provides the following API, allowing extraction of data needed to perform the export operation.

Note that the following diagrams show only the API needed for export, not the full API of the Mesh ADK.

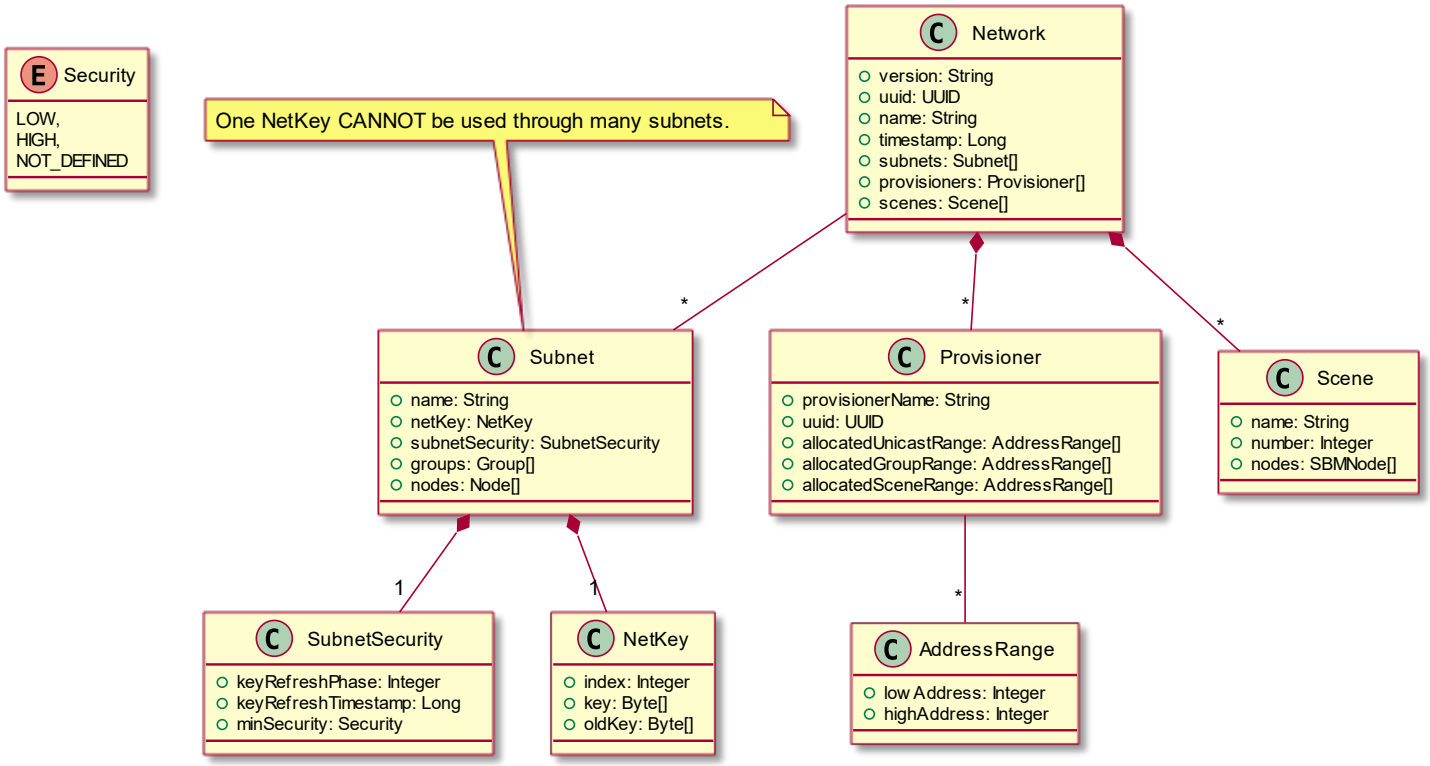
The Mesh ADK does not define any format for exporting data. Developers using the ADK are responsible for making sure that all desired values have been filled by sending necessary requests or performing necessary operations. Developers are also responsible for collecting data needed for export and saving it in a format appropriate for external storage.

Example of export data to json string is available in the section 14 Code Examples. For the iOS see function `performExport` in `SBMMesh.swift` file. For the Android see method `export` of `JsonExporter` class.

11.1 Network

The Network object contains the current state of the mesh network. It contains, among others, provisioner objects and subnet objects.

- Provisioner object represents a mesh node capable of adding a device to a mesh network.
- Subnet object represents a network key utilized for securing communication at the network layer.



New in ADK 2.2.0

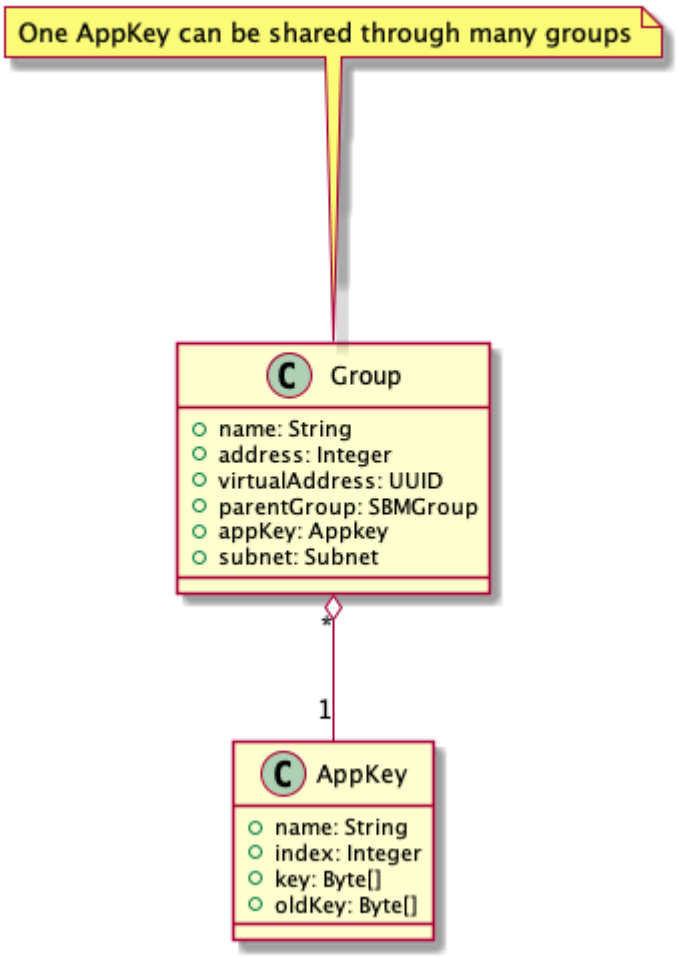
- SubnetSecurity, Provisioner, AddressRange, Scene and Security

Changed in ADK 2.2.0

- Network – added properties: version, provisioners, scenes and timestamp
- Subnet – added property: subnetSecurity

11.2 Group

The Group object represents the group using the application key utilized for securing communication at the access layer.

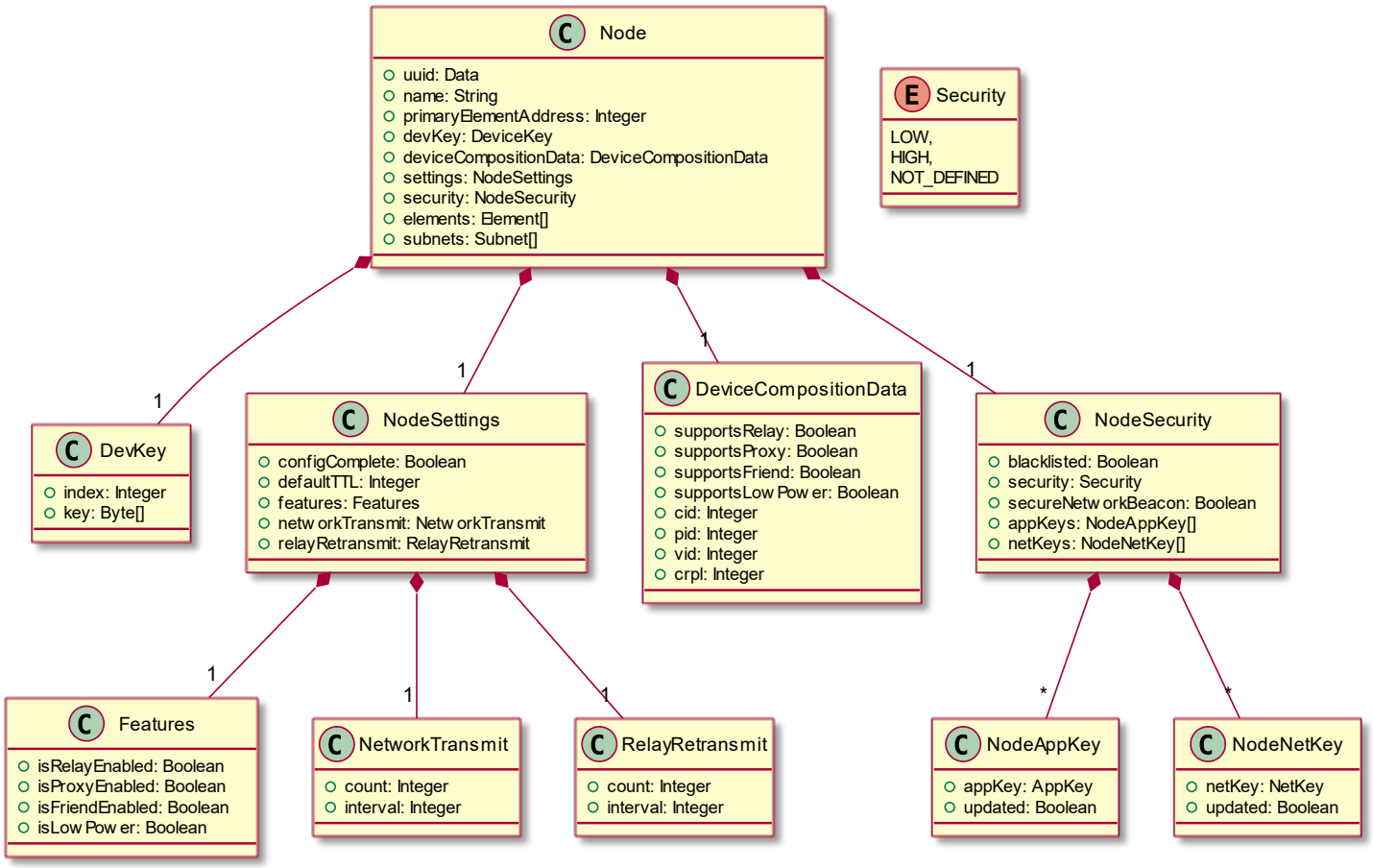


Changed in ADK 2.2.0

- Group – added properties: parentGroup and virtualAddress
- AppKey – added property: name

11.3 Node

The Node object represents the configured state of the mesh node.



New in ADK 2.2.0

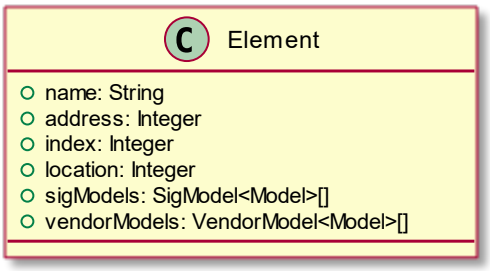
- NodeSettings, Features, NetworkTransmit, RelayRetransmit, NodeSecurity, NodeAppKey, NodeNetKey and Security

Changed in ADK 2.2.0

- Node – added properties: security and settings

11.4 Element

The Element object represents a mesh element – an addressable entity within a mesh node.

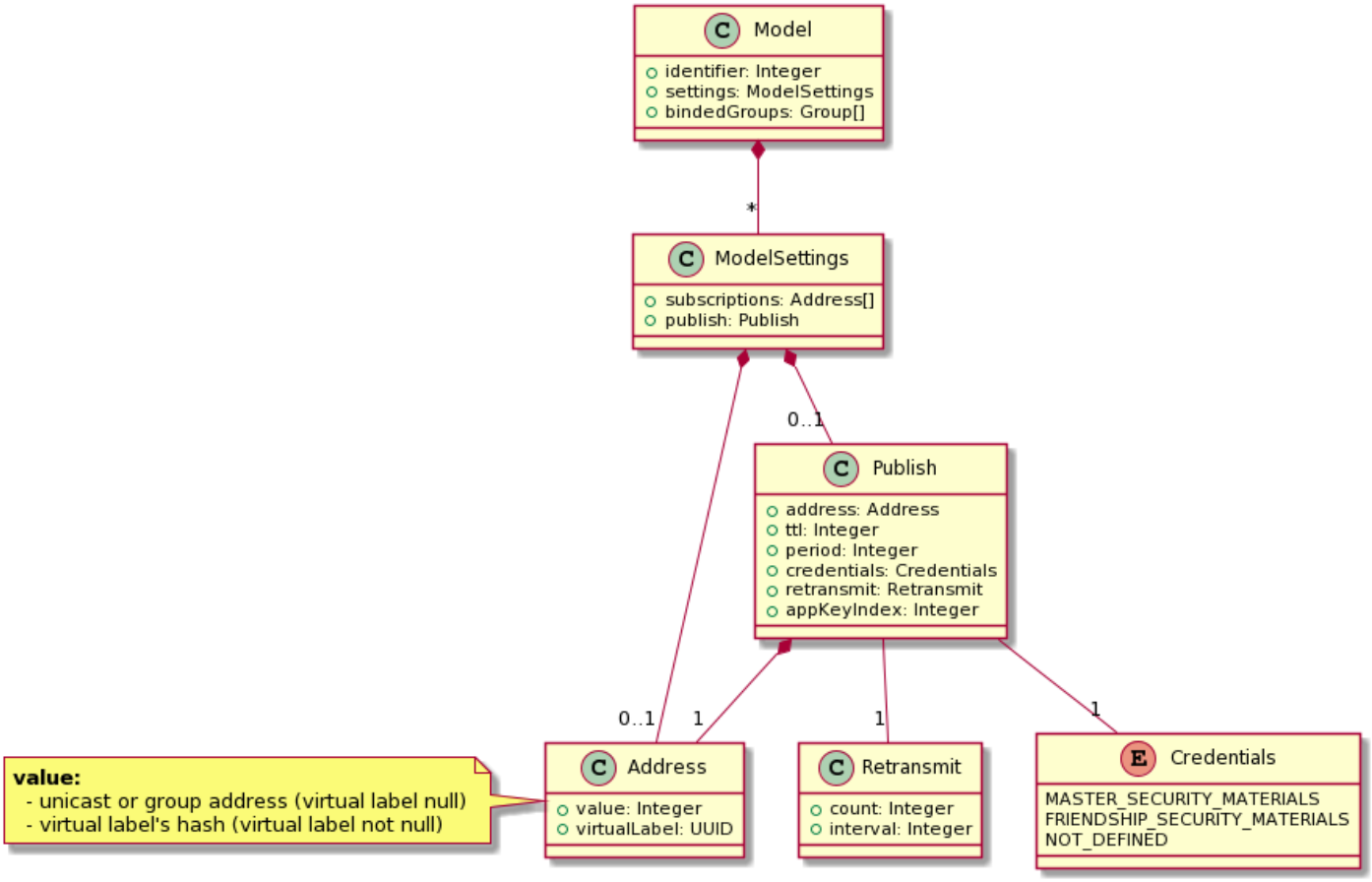


Changed in ADK 2.2.0

- Element – added properties: name, index and location

11.5 Model

The Model object represents a configured state of a mesh model.



New in ADK 2.2.0

- ModelSettings, Publish, Address, Retransmit and Credentials

Changed in ADK 2.2.0

- Model – added property: modelSettings

Changed in ADK 2.2.3

- Publish – added property: appKeyIndex

12 Import

The Mesh library provides an API that supports importing a Mesh Network. In the API the Importer class contains logic used to import data. Currently only one Mesh Network can be imported to the Bluetooth Mesh library. For each Mesh Network import, follow these steps to correctly load data:

- The BluetoothMesh database must be empty before import. Use the `clearDatabase` method from the BluetoothMesh class if the database is not empty.
- Use the Importer class to import Mesh Network to database. The classes that support importing mesh structure can be found below. It is the developer's responsibility to fill the object with mesh data.
- Call the `initializeNetwork` method from the BluetoothMesh class to configure the provisioner's address and iv index. The developer must maintain sequence numbers, iv indexes, and provisioners' addresses in the Mesh Network. More information about it can be found in the Bluetooth Mesh Profile specification.

In a test environment the developer can use a different address and iv index for each network initialization.

An example of import data from json string is available in section 14 Code Examples. For the iOS see function `performImport` in `SBMMesh.swift` file. For the Android see method `import` of `JsonImporter` class.

12.1 Importer Class

Importer - A base class that can be used to import Mesh Network. Data will be loaded to the BluetoothMesh database after use of the `performImport` method from the Importer class. To create an empty Network the developer must use the `createNetwork` method from the Importer class. After that the developer must fill the given Network with Mesh data.

12.2 Network Import Classes

- NetworkImport
- ProvisionerImport
- SceneImport
- AddressRangeImport
- SubnetImport
- SubnetSecurityImport
- NetKeyImport

12.3 Group Import Classes

- GroupImport
- AppKeyImport

12.4 Node Import Classes

- NodeImport
- NodeSettingsImport
- NetworkTransmitImport
- RelayRetransmitImport
- FeaturesImport
- NodeSecurityImport
- NodeAppKeyImport
- NodeNetKeyImport
- SecurityType
- DeviceCompositionDataImport
- DevKeyImport

12.5 Element Import Classes

- ElementImport
- ModelImport
- ModelSettingsImport
- PublishImport
- AddressImport
- RetransmitImport
- CredentialsType

12.6 Sequence Number

The importer does not ensure that imported mesh uses correct sequence number. After performing a mesh import, to ensure that messages are being sent properly, the sequence number should be changed to the one used by the original of this mesh.

Use `BluetoothMesh.getInstance().setSequenceNumber(value);` for the Android and `SBMBluetoothMesh.sharedInstance.setSequenceNumber(value)` for the iOS

You can get original sequence number by calling proper method on the original mesh before export mesh data.

Use `BluetoothMesh.getInstance().getSequenceNumber();` for the Android and `SBMBluetoothMesh.sharedInstance.sequenceNumber()` for the iOS

For more information about Sequence number see the Bluetooth SIG Mesh Specification (see section 3.8.3 Sequence Number from the *Mesh Profile Bluetooth® Specification*).

13 Multiple Networks

13.1 Concept

The Bluetooth Mesh ADK does not support either multiple networks or databases. As a workaround, use the import/export feature.

The main concept is to have the exported networks' structures within the application, for example in json files. Examples of import and export from/to json files is in section 14 Code Examples.

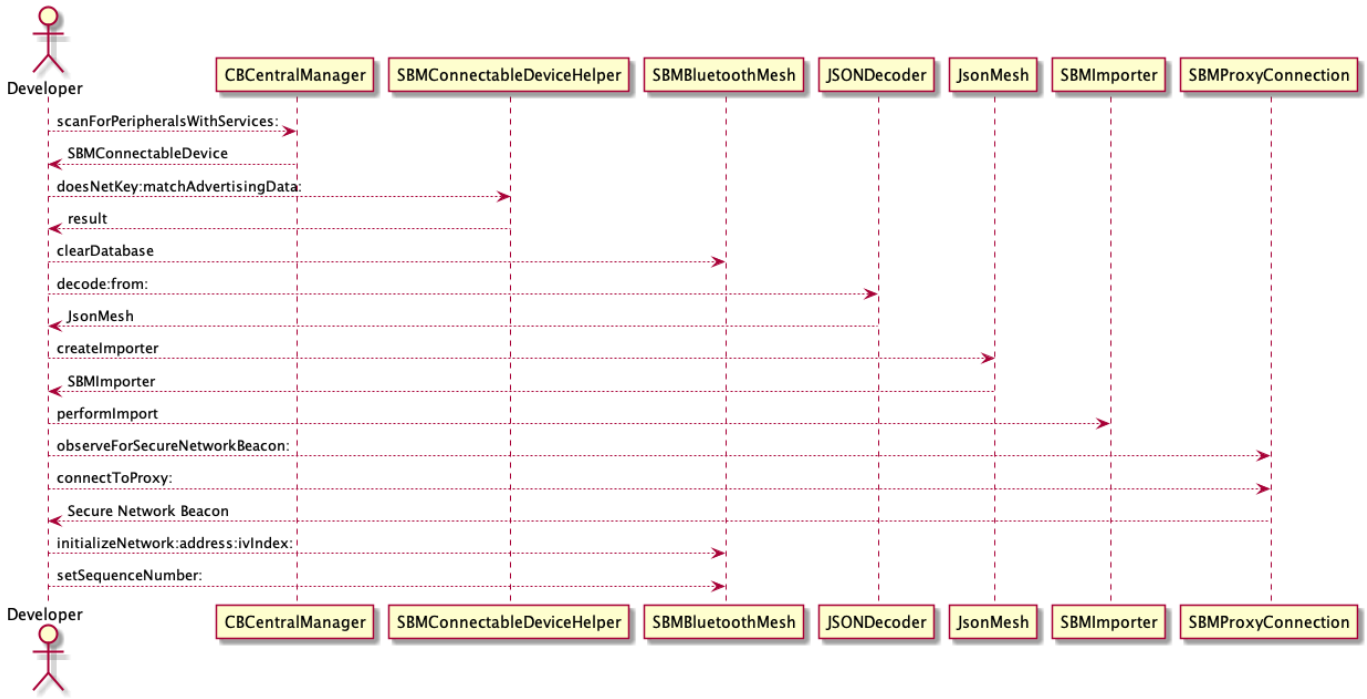
To choose between different networks, the nearest devices can be discovered. It is possible to scan for provisioned devices with a disabled Node Identity feature and compare them to existing Network Keys. After finding the appropriate network, its configuration should be imported. Currently only one Network can be imported at the same time in the Bluetooth Mesh library. It means that multiple instances of the Mesh Network cannot be initialized in the library. If many instances of Mesh Networks must be stored, export the current network from the Bluetooth Mesh library, save it on the Mobile Application side, and then clear the Bluetooth Mesh library database and create the next Mesh Network instance using the createNetwork method from the BluetoothMesh object.

It is a known issue that the createNetwork method from the BluetoothMesh object will not create a separate Mesh Network instance in Bluetooth Mesh storage. Each network created with the createNetwork method uses the same resources pool. Therefore, create only one Mesh Network instance in Bluetooth Mesh library at a time.

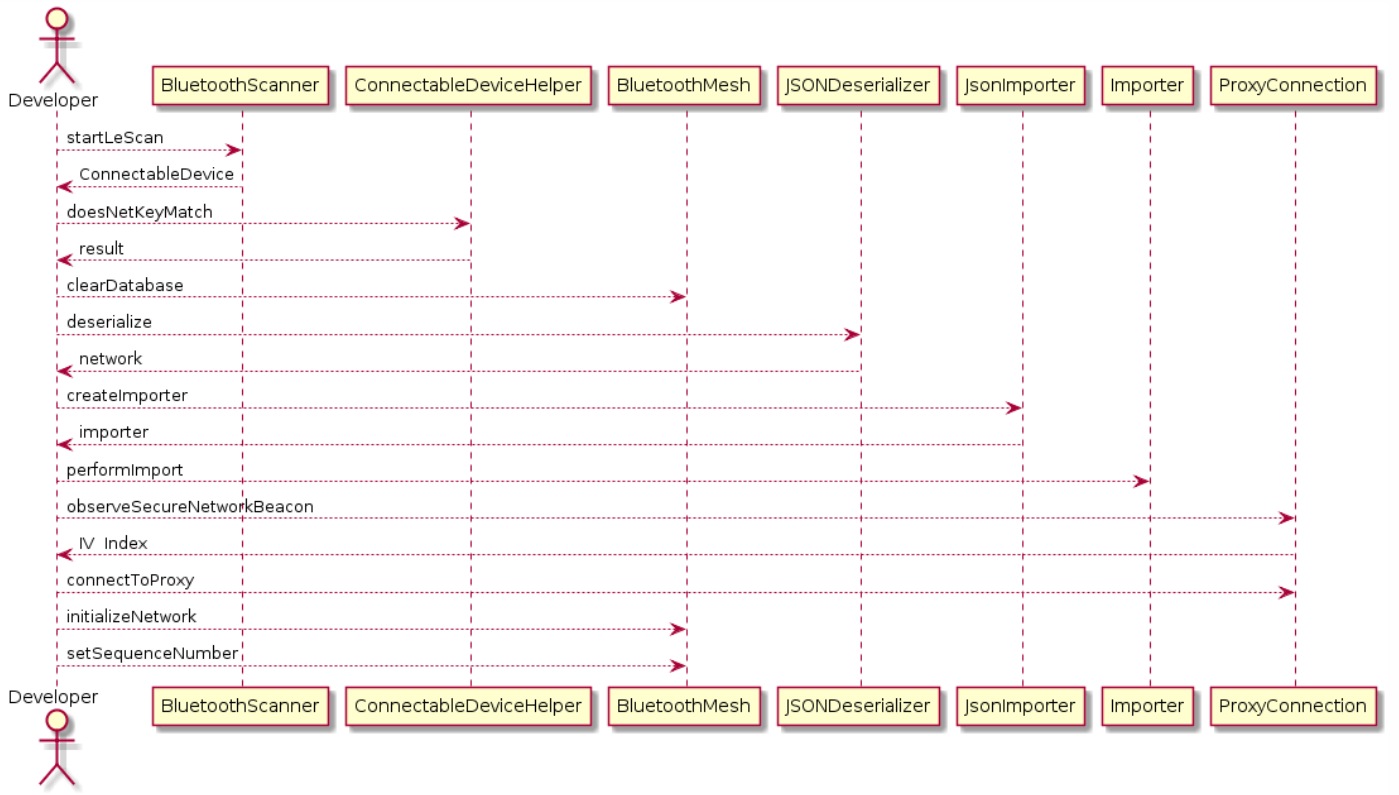
The device address and sequence number used in communication with the mesh network are not stored in the exported configuration file. They should be stored by the application and preserved for later use before clearing the database. The IV index can be retrieved from Secure Network Beacons so it does not need to be saved.

The following sections describe how to handle multiple networks using this workaround. All necessary components and their connections are shown on UML diagrams for clarity. Classes differ between iOS and Android implementations, so versions for both are included.

iOS:



Android:



13.2 Discovering Provisioned Devices

To scan for the nearest devices, use:

- scanForPeripherals from CBCentralManager **in iOS**
- startLeScan from BluetoothScanner **in Android**

These contain a parameter that specifies which services should be discovered. Mesh devices before provisioning have a mesh UUID of value "*1827*" (full: *"00001827-0000-1000-8000-00805f9b34fb"*). As specified in the Bluetooth Mesh Profile specification, section 8.6.1, provisioned devices advertise their UUID with a value of "*1828*" (full: *"00001828-0000-1000-8000-00805f9b34fb"*). So, this value should be used to scan for existing mesh networks. In Android you must provide full, 128-bit value.

iOS:

```
let centralManager = CBCentralManager(delegate: self, queue: nil)
centralManager.scanForPeripherals(withServices: [CBUUID(string: "1828")])
```

Android:

```
val bluetoothScanner = BluetoothScanner(BluetoothStateReceiver())
bluetoothScanner.startLeScan(ParcelUuid(UUID.fromString("00001828-0000-1000-8000-00805f9b34fb")))
```

13.3 Comparing Advertisement Data to Existing Network Keys

Along with storing all networks' structure files, all network keys must be retrieved and stored in an array. This makes it possible to compare every device's advertisement data with all keys.

A short description of how to do a compare is in section:

- 7.24.1 Check if Advertisement Data Match netKey **for iOS**
- 8.25.1 Check if Advertisement Data Matches a Net Key **for Android**

These methods take two arguments:

The first is a raw network key data in binary form, stored as a bytes array.

The second is the device to be checked for being a part of created network. Its type is ConnectableDevice, whose implementation must be done by the developer as stated in section:

- 7.6 Set Up Bluetooth Layer (SBMConnectableDevice) **for iOS**
- 8.3 Set Up Bluetooth Layer (ConnectableDevice) **for Android**

This method returns true if the compared advertisement data belongs to a device which is part of the network. Otherwise it returns False. The node does not need to be in every subnet to be considered part of a network's structure. It can be in only one subnet to enable detecting the network to which it belongs.

Nodes from multiple networks may be in range, so a method of deciding which network will be imported must be defined.

iOS:

```
var device: SBMConnectableDevice
var keyData : Data
…
do {
    try SBMConnectableDeviceHelper.doesNetKey(keyData, matchAdvertisingData: device)
    // Handle successful comparison
} catch {
    // Ignore node
}
```

Android:

```
val connectableDeviceHelper: ConnectableDeviceHelper =
BluetoothMesh.getInstance().connectableDeviceHelper

var device: BluetoothConnectableDevice
var netKey: NetKey
...
val matchFound = connectableDeviceHelper.doesNetKeyMatch(netKey.key, device)
```

For this method to work, the device must be provisioned in "without Node Identity" mode. If not, it will not advertise its network ID in advertisement data, which will make matching impossible. Such changes can be made in:

- *provisionWithConfiguration:parameters:retryCount:* from SBMProvisionerConnection **in iOS**
- *provisionDevice* from *ProvisioningModel* **in Android**

iOS:

```
let provisionerConnection = SBMProvisionerConnection(for: device, subnet: subnet)
let configuration = SBMProvisionerConfiguration(proxyEnabled: true, nodeIdentityEnabled: false)
provisionerConnection.provision(withConfiguration: configuration, parameters: nil, retryCount: 1) {
connection, node, error in
    // Handle provisioning result
}
```

Android:

```
val provisionerConnection = ProvisionerConnection(connectableDevice, network)
val provisionerConfiguration = ProvisionerConfiguration().apply {
    isGettingDeviceCompositionData = true
    isEnablingProxy = true
    isEnablingNodeIdentity = false //false by default
}
provisionerConnection.provision(provisionerConfiguration, nodeProperties,
new ProvisioningCallback() {
    @Override
    public void success(ConnectableDevice device, Subnet subnet, Node node) {

    }

    @Override
    public void error(ConnectableDevice device, Subnet subnet, ErrorType error) {

    }
});
```

Note: Node Identity can be disabled with the setNodeIdentity method from the ConfigurationControl object.

13.4 Importing the Network Structure

Before importing a network, the database must be cleared by calling the *clearDatabase* method on a shared instance of:

- SBMBluetoothMesh **in iOS**
- BluetoothMesh **in Android**

To import a database using example classes from section 14 Code Examples, the Json object should be decoded from JSON file using:

- JSONDecoder class from Swift's standard library **in iOS**
- Gson class from Gson library **in Android**

After successfully decoding the main configuration object, the method *performImport* should be called on it. Only after it finishes successfully is the network configuration imported to the database and ready to use.

After importing the network, the sequence number and IV index must be set.

iOS:

```
let json = "..."
let decoder = JSONDecoder()
do {
    let meshJson = try decoder.decode(JsonMesh.self, from: (json.data(using: .utf8))!)
    SBMBluetoothMesh.sharedInstance().clearDatabase()
    let importer = meshJson.createImporter()
    try? importer.performImport()
} catch {
    // Handle import failure
}
```

Android:

```
val json = Gson().fromJson<JsonMesh>(jsonString, JsonMesh::class.java)
JsonImporter(json).import()
```

(The rest of the operations is contained in JsonImporter.import method).

13.5 Retrieving the IV Index from the Secure Network Beacon

After establishing a connection to a proxy node, the Secure Network Beacon should be received. This supports reading and storing the current IV index for future use. To do this, the callback to be called after receiving the beacon must be set. In this callback, a value must be stored in a variable that exists outside of it.

iOS:

```
let device = ...
let proxyConnection = SBMProxyConnection(connectableDevice: device)
proxyConnection.observe(forSecureNetworkBeacon: { (netKeyIndex, keyRefresh, ivUpdate, ivIndex) in
    // Save ivIndex somewhere
})

proxyConnection.connect(toProxy: { device in
    // Successful connection
}, errorCallback: { device, error in
    // Handle error
})
```

Android:

```
val device: BluetoothConnectableDevice = ...
val proxyConnection = ProxyConnection(device)
proxyConnection.observeSecureNetworkBeacon { _, _, _, ivIndex ->
    ... = ivIndex
}
```

13.6 Initializing the Network

After receiving the current IV index, the network can be initialized. It sets the device as a provisioner in the mesh network and assigns its address. The provisioner address should be unique for each device and should not change over time. It is not included in the export structure, so it must be saved manually.

iOS:

```
let address = ...
let iv = ...
let network = SBMBluetoothMesh.sharedInstance().networks().first!

try? SBMBluetoothMesh.sharedInstance().initializeNetwork(network, address: address, ivIndex: iv)
```

Android:

```
for (net in BluetoothMesh.getInstance().networks) {
    val address = …
    val iv = …

    BluetoothMesh.getInstance().initializeNetwork(net, address, iv)
}
```

13.7 Getting and Setting the Sequence Number

As described in section 12.6 Sequence Number, after importing the mesh network configuration, sequence numbers are not imported. They should be set manually.

If the sequence number is not set to the correct value, nodes will not respond to commands.

The sequence number must be saved before clearing the database so that it can be retrieved during later imports.

iOS:

```
try SBMBluetoothMesh.sharedInstance().sequenceNumber()
...
let value = ...
try SBMBluetoothMesh.sharedInstance().setSequenceNumber(value)
```

Android:

```
val value = BluetoothMesh.getInstance().sequenceNumber
...
BluetoothMesh.getInstance().sequenceNumber = value
```

14 Code Examples

14.1 iOS

14.1.1 JsonMesh

```
import BluetoothMesh

/**
 Sample implementation of import/export classes.
 This class can be used to import or export whole Mesh network. It uses Codable protocol
implemented specifically for JSON scheme to simplify converting in-memory structure to JSON.
 */
class JsonMesh : Codable {
    var schema : String
    var id : String
    var version : String
    var name : String
    var uuid : String
    var timestamp : JsonTimestamp
    var netKeys : [JsonNetKey]
    var appKeys : [JsonAppKey]
    var provisioners : [JsonProvisioner]
    var nodes : [JsonNode]
    var groups : [JsonGroup]
    var scenes : [JsonScene]

    enum CodingKeys : String, CodingKey {
        case schema = "$schema"
        case id
        case version
        case name = "meshName"
        case uuid = "meshUUID"
        case timestamp

        case netKeys
        case appKeys
        case provisioners
        case nodes
        case groups
        case scenes
    }

    /**
     Initializes data structure used to perform export.
     */
    init(network: SBMNetwork) {
        schema = "http://test-schema.org"
        id = "http://test-id.org/id"
        name = network.name ?? ""
        version = network.version
        uuid = network.uuid.hexEncodedString()
        timestamp = JsonTimestamp(timestamp: network.timestamp)
        netKeys = []
        appKeys  = []
        provisioners = []
        nodes = []
        groups = []
        scenes = []

        initSubnets(subnets: network.subnets)
```

```
        initScenes(scenes: network.scenes)
        initProvisioners(provisioners: network.provisioners)
    }

    private func initSubnets(subnets: [SBMSubnet]) {
        for subnet in subnets {
            for group in subnet.groups{
                groups.append(JsonGroup(group: group))



                if !appKeys.contains(where: {$0.index == group.appKey.keyIndex}) {

                    appKeys.append(JsonAppKey(appKey:  group.appKey,  netKeyIndex:group.subnet!.net-
Key.keyIndex))

                }

            }

            for node in subnet.nodes {
                nodes.append(JsonNode(node: node))
            }

            netKeys.append(JsonNetKey(subnet: subnet))
        }
    }

    private func initScenes(scenes: [SBMScene]) {
        for scene in scenes {
            self.scenes.append(JsonScene(scene: scene))
        }
    }

    private func initProvisioners(provisioners: [SBMProvisioner]) {
        for provisioner in provisioners {
            self.provisioners.append(JsonProvisioner(provisioner: provisioner))
        }
    }

    /**
     Creates importer class and fills it data from JsonMesh structure
     */
    func createImporter() -> SBMImporter {
        let importer = SBMImporter()

        let networkImport = importer.createNetwork((uuid.data(using: .hexadecimal))!)
        networkImport.name = name

        for item in netKeys {
            item.performImport(network: networkImport)
        }

        // AppKeys & Groups
        for item in groups {
            item.performImport(network: networkImport, appKeys: appKeys)
        }

        for item in provisioners {
            item.performImport(network: networkImport)
        }

        for node in nodes {
            node.performImport(network: networkImport)
        }
```

```
        for item in scenes {

            item.performImport(network: networkImport)

        }

        return importer
    }
}

/**
 Sample code to show how to use JsonMesh object to perform import
 */
func performImport(from json: String) {
    let decoder = JSONDecoder()

    do {
        let meshData = try decoder.decode(JsonMesh.self, from: (json.data(using: .utf8))!)
        // clear database to ensure it's ready for import
        SBMBluetoothMesh.sharedInstance().clearDatabase()
        let importer = meshData.createImporter()
        try importer.performImport()

        let network = SBMBluetoothMesh.sharedInstance().networks()[0]
        let address = 0
        let ivi = 0
        try SBMBluetoothMesh.sharedInstance().initializeNetwork(item, address: address, ivIndex:
ivi)
    } catch {
    }
}

/**
 Sample code to show how to use JsonMesh object to perform export
 */
func performExport(of network: SBMNetwork) -> String? {
    let meshData = JsonMesh(network)
    let encoder = JSONEncoder()

    do {
        let jsonData = try encoder.encode(meshData)
        return String(data: jsonData, encoding: .utf8)
    } catch {
        return nil
    }
}
```

14.1.2 JsonNetKey

```
import BluetoothMesh

/**
 Represents Mesh Network Key in JSON Mesh structure
 */
class JsonNetKey : Codable {
    var name : String
    var index : Int
    var key : String
    var oldKey : String?
    var minSecurity : JsonSecurityType
    var phase : UInt8?
```

```
    var timestamp : JsonTimestamp?

    init(subnet: SBMSubnet) {
        name = subnet.name
        index = subnet.netKey.keyIndex
        key = subnet.netKey.rawKey.hexEncodedString()
        minSecurity = JsonSecurityType(type: subnet.subnetSecurity.minSecurity)
        phase = subnet.subnetSecurity.keyRefreshPhase
        timestamp = JsonTimestamp(timestamp: subnet.subnetSecurity.keyRefreshTimestamp)
    }

    /**
     Creatres network key within BluetoothMesh importer classes
     */
    func performImport(network : SBMNetworkImport) {
        let keyData = (key.data(using: .hexadecimal))!
        let oldKeyData = oldKey?.data(using: .hexadecimal)

        let netKey = SBMNetKeyImport.initWith(index, key: keyData, oldKey: oldKeyData)
        let subnet = network.createSubnet(withNetKey: netKey)

        subnet.name = name
        if phase != nil {
            subnet.createSubnetSecurity(withRefreshPhase: phase!, andRefreshTimestamp:
timestamp!.timestamp)
        }
    }
}
```

14.1.3 JsonAppKey

```
import BluetoothMesh

/**
 Represents Application Key object in JSON Mesh scheme
 */
class JsonAppKey : Codable {
    var name : String
    var index : Int
    var boundNetKey : Int
    var key : String
    var oldKey : String?

    init(appKey: SBMApplicationKey, netKeyIndex: Int) {
        name = appKey.name ?? ""
        index = appKey.keyIndex
        key = appKey.rawKey.hexEncodedString()
        oldKey = appKey.oldRawKey != nil ? appKey.oldRawKey?.hexEncodedString() : nil
        boundNetKey = netKeyIndex
    }
}
```

14.1.4 JsonProvisioner

```
import BluetoothMesh

/**
 Represents address range in JSON Mesh structure
 */
class JsonAddressRange : Codable {
    var highAddress : JsonHex
```

```
    var lowAddress : JsonHex

    init(low: UInt, high: UInt){
        lowAddress = JsonHex(value: low, width: 4)
        highAddress = JsonHex(value: high, width: 4)
    }

    enum CodingKeys : String, CodingKey {
        case highAddress
        case lowAddress
    }
}

/**
Represents scene range in JSON Mesh structure
*/
class JsonSceneRange : Codable {
    var firstScene : JsonHex
    var lastScene : JsonHex

    init(first: UInt, last: UInt){
        firstScene = JsonHex(value: first, width: 4)
        lastScene = JsonHex(value: last, width: 4)
    }

    enum CodingKeys : String, CodingKey {
        case firstScene
        case lastScene
    }
}

/**
 Represents provisioner within JSON Mesh structure
 */
class JsonProvisioner : Codable {
    var name : String
    var uuid : String
    var allocatedGroupRange : [JsonAddressRange]
    var allocatedUnicastRange : [JsonAddressRange]
    var allocatedSceneRange : [JsonSceneRange]

    enum CodingKeys : String, CodingKey {
        case name = "provisionerName"
        case uuid = "UUID"
        case allocatedGroupRange
        case allocatedUnicastRange
        case allocatedSceneRange
    }

    init(provisioner : SBMProvisioner) {
        name = provisioner.provisionerName
        uuid = provisioner.uuid.replacingOccurrences(of: "-", with: "")
        allocatedGroupRange = provisioner.allocatedGroupRange.map({ (range: SBMAddressRange) ->
JsonAddressRange in
            return JsonAddressRange(low: UInt(range.lowAddress), high: UInt(range.highAddress))
        })

        allocatedUnicastRange = provisioner.allocatedUnicastRange.map({ (range: SBMAddressRange) ->
JsonAddressRange in
            return JsonAddressRange(low: UInt(range.lowAddress), high: UInt(range.highAddress))
        })
```

```
        allocatedSceneRange = provisioner.allocatedSceneRange.map({ (range: SBMAddressRange) ->
JsonSceneRange in
            return JsonSceneRange(first: UInt(range.lowAddress), last: UInt(range.highAddress))
        })
    }

    /**
     Creates provisioner within BluetoothMesh importer classes
     */
    func performImport(network: SBMNetworkImport) {
        let uuidData = uuid.data(using: .hexadecimal)!
        let provisioner = network.createProvisioner(withUUID: uuidData)
        provisioner.name = name

        importGroupRange(to: provisioner)
        importUnicastRange(to: provisioner)
        importSceneRange(to: provisioner)
    }

    private func importGroupRange(to provisioner: SBMProvisionerImport) {
        for item in allocatedGroupRange {
            let lowAddress = UInt16(item.lowAddress.value)
            let highAddress = UInt16(item.highAddress.value)
            provisioner.createGroupRange(withLowAddress: lowAddress, andHighAddress: highAddress)
        }
    }

    private func importUnicastRange(to provisioner: SBMProvisionerImport) {
        for item in allocatedUnicastRange {
            let lowAddress = UInt16(item.lowAddress.value)
            let highAddress = UInt16(item.highAddress.value)
            provisioner.createUnicastRange(withLowAddress: lowAddress, andHighAddress: highAddress)
        }
    }

    private func importSceneRange(to provisioner: SBMProvisionerImport) {
        for item in allocatedSceneRange {
            let lowAddress = UInt16(item.firstScene.value)
            let highAddress = UInt16(item.lastScene.value)
            provisioner.createSceneRange(withLowAddress: lowAddress, andHighAddress: highAddress)
        }
    }
}
```

14.1.5 JsonNode

```
import BluetoothMesh

/**
 Represents Mesh Node in JSON Mesh structure
 */
class JsonNode : Codable {
    var uuid : String
    var name : String?
    var deviceKey : String
    var unicastAddress : JsonHex
    var security : JsonSecurityType
    var cid : JsonHex?
    var pid : JsonHex?
    var vid : JsonHex?
    var crpl : JsonHex?
    var features : JsonFeatures
```

```
    var elements : [JsonElement]
    var configComplete : Bool
    var netKeys : [JsonNodeKey]
    var appKeys : [JsonNodeKey]
    var knownAddresses: [JsonHex]?
    var secureNetworkBeacon : Bool?
    var defaultTTL : UInt?
    var networkTransmit : JsonRetransmit?
    var relayRetransmit : JsonRetransmit?
    var blacklisted : Bool

    enum CodingKeys : String, CodingKey {
        case uuid = "UUID"
        case name
        case deviceKey
        case unicastAddress
        case security
        case cid
        case pid
        case vid
        case crpl
        case features
        case elements
        case configComplete
        case netKeys
        case appKeys
        case knownAddresses
        case secureNetworkBeacon
        case defaultTTL
        case networkTransmit
        case relayRetransmit
        case blacklisted
    }

    init(node : SBMNode) {
        uuid = node.uuid.hexEncodedString()
        name = node.name
        deviceKey = node.devKey.rawKey.hexEncodedString()
        unicastAddress = JsonHex(value: UInt(node.primaryElementAddress), width: 4)
        security = JsonSecurityType(type: node.security.security)

        features = JsonFeatures(feature: node.settings.feature, dcd: node.deviceCompositionData)
        elements = node.elements.map({ (element: SBMElement) -> JsonElement in
            return JsonElement(element: element)
        })
        configComplete = node.settings.isConfigComplete
        netKeys = node.security.netKeys.map({ (key: SBMNodeNetKey) -> JsonNodeKey in
            return JsonNodeKey(netKey: key)
        })
        appKeys = node.security.appKeys.map({ (key: SBMNodeAppKey) -> JsonNodeKey in
            return JsonNodeKey(appKey: key)
        })
        knownAddresses = node.security.appKeys.flatMap({ (key: SBMNodeAppKey) -> [JsonHex] in
            return node.groups.filter { group -> Bool in
                return group.appKey.keyIndex == key.appKey
            }.map { group -> JsonHex in
                return JsonHex(value: UInt(group.address), width: 4)
            }
        })

        secureNetworkBeacon = node.security.isSecureNetworkBeacon
        defaultTTL = UInt(node.settings.defaultTTL)
        blacklisted = node.security.blacklisted
```

```
        initDCD(dcd: node.deviceCompositionData)
        initNetworkTransmit(networkTransmit: node.settings.networkTransmit)
        initRelayRetransmit(networkTransmit: node.settings.relayRetransmit)
    }

    private func initDCD(dcd : SBMDeviceCompositionData?) {
        if(dcd != nil) {
            cid = JsonHex(value: UInt(dcd!.cid), width: 4)
            pid = JsonHex(value: UInt(dcd!.pid), width: 4)
            vid = JsonHex(value: UInt(dcd!.vid), width: 4)
            crpl = JsonHex(value: UInt(dcd!.crpl), width: 4)
        }
    }

    private func initNetworkTransmit(networkTransmit : SBMNetworkTransmit?) {
        if(networkTransmit != nil && networkTransmit!.count > 0 && networkTransmit!.interval > 0){
            self.networkTransmit = JsonRetransmit(transmit: networkTransmit!)
        }
    }

    private func initRelayRetransmit(networkTransmit : SBMRelayRetransmit?) {
        if(networkTransmit != nil && networkTransmit!.count > 0 && networkTransmit!.interval > 0){
            self.relayRetransmit = JsonRetransmit(transmit: networkTransmit!)
        }
    }

    /**
     Creates node within BluetoothMesh importer classes
     */
    func performImport(network: SBMNetworkImport) {
        let uuidData = self.uuid.data(using: .hexadecimal)!
        let primaryElementAddress = Int(unicastAddress.value)
        let deviceKeyData = self.deviceKey.data(using: .hexadecimal)!
        let deviceKeyImport = SBMDevKeyImport.initWithKey(deviceKeyData)

        let node = network.createNode(uuidData, primaryElementAddress: primaryElementAddress,
deviceKey: deviceKeyImport)
        node.name = self.name

        importSettings(node: node)
        importKeys(from: network, to: node)
        importSecurity(node: node)

        if cid != nil, pid != nil, vid != nil, crpl != nil {
            importDCD(node: node)
        }

        for item in elements {
            item.performImport(node: node)
        }
    }

    private func importSecurity(node: SBMNodeImport) {
        let security = node.security
        security.blacklisted = self.blacklisted
        security.security = self.security.type

        for item in netKeys {
            security.createNodeNetKey(with: item.index, isUpdated: item.updated)
        }

        for item in appKeys {
```

```
            security.createNodeAppKey(with: item.index, isUpdated: item.updated)
        }
    }

    private func importSettings(node: SBMNodeImport) {
        let settings = node.settings
        settings.configComplete = self.configComplete
        settings.defaultTTL = UInt32(self.defaultTTL ?? 0)

        if let networkTransmit = networkTransmit {
            settings.createNetworkTransmit(withCount: networkTransmit.count, interval:
networkTransmit.interval)
        }

        if let relayRetransmit = relayRetransmit {
            settings.createRelayRetransmit(withCount: relayRetransmit.count, interval:
relayRetransmit.interval)
        }
    }

    private func importKeys(from network: SBMNetworkImport, to node: SBMNodeImport) {
        for item in netKeys {
            let subnet = self.getSubnetWithNetKeyIndex(index: item.index, from: network)!
            node.addSubnet(subnet)
        }

        for item in appKeys {
            let groups = self.getGroupsWithAppKeyIndex(index: item.index, from: network)

            for group in groups {
                node.addGroup(group)
            }
        }
    }

    private func importDCD(node: SBMNodeImport) {
        if let friend = self.features.friend,
            let proxy = self.features.proxy,
            let relay = self.features.relay,
            let lowPower = self.features.lowPower {

            let dcd = node.createDeviceCompositionData()
            dcd.cid = Int(cid!.value)
            dcd.pid = Int(pid!.value)
            dcd.vid = Int(vid!.value)
            dcd.crpl = Int(crpl!.value)
            dcd.supportsFriend = (friend != 2)
            dcd.supportsProxy = (proxy != 2)
            dcd.supportsRelay = (relay != 2)
            dcd.supportsLowPower = (lowPower != 2)

            let features = node.settings.features
            features.friendEnabled = (friend != 2) ? NSNumber.init(value: friend) : nil
            features.proxyEnabled = (proxy != 2) ? NSNumber.init(value: proxy) : nil
            features.relayEnabled = (relay != 2) ? NSNumber.init(value: relay) : nil
            features.lowPower = (lowPower != 2) ? NSNumber.init(value: lowPower) : nil
        }
    }

    private func getSubnetWithNetKeyIndex(index: Int, from network: SBMNetworkImport) ->
SBMSubnetImport? {
        return network.subnets.first{ $0.netKey.index == index }
    }
```

```
    private func getGroupsWithAppKeyIndex(index: Int, from network: SBMNetworkImport) ->
[SBMGroupImport] {
        let groups = network.subnets.flatMap { subnet -> [SBMGroupImport] in
            return getGroupsWithAppKeyIndex(index: index, from: subnet)
        }

        return groups
    }

    private func getGroupsWithAppKeyIndex(index: Int, from subnet: SBMSubnetImport) ->
[SBMGroupImport] {
        let groups = subnet.groups.filter { group -> Bool in
            guard let groupAddressIsKnown = knownAddresses?.contains(where: { $0.value ==
group.address }) else {
                return group.appKey.index == index
            }

            return (group.appKey.index == index) && groupAddressIsKnown
        }

        return groups
    }
}
```

14.1.6 JsonGroup

```
import BluetoothMesh

/**
 Represents Mesh Group in JSON Mesh structure.
 */
class JsonGroup : Codable {
    var name : String
    var address : JsonHex
    var parentAddress : JsonHex
    var appKeyIndex : Int

    enum CodingKeys : String, CodingKey {
        case name
        case address
        case parentAddress
        case appKeyIndex
    }

    init(group: SBMGroup) {
        name = group.name
        address = JsonHex(value: UInt(group.address), width: 4)
        parentAddress = (group.parent != nil) ? JsonHex(value: UInt(group.parent!.address), width:
4) : JsonHex(value: 0, width: 4)
        appKeyIndex = group.appKey.keyIndex
    }

    /**
     Creates group within BluetoothMesh importer classes using data from Mesh JSON structure
     */
    func performImport(network: SBMNetworkImport, appKeys: [JsonAppKey]) {
        let appKey = getAppKeyWithIndex(index: appKeyIndex, from: appKeys)!
        let subnet = getSubnetWithNetKeyIndex(index: appKey.boundNetKey, from: network)!
        let appKeyData = appKey.key.data(using: .hexadecimal)!
        let appOldKeyData = appKey.oldKey?.data(using: .hexadecimal)
```

```
        var appKeyImport: SBMAppKeyImport? = subnet.groups.map { groupImport -> SBMAppKeyImport in
            return groupImport.appKey
        }.first { appKeyImport -> Bool in
            return appKeyImport.index == appKey.index && appKeyImport.key == appKeyData &&
appKeyImport.oldKey == appOldKeyData
        }

        if appKeyImport == nil {
             appKeyImport = SBMAppKeyImport.initWith(appKey.index, key: appKeyData, oldKey:
appOldKeyData)
             appKeyImport?.name = appKey.name
        }

        let group = subnet.createAndAddGroup(withAddress: UInt16(address.value), appKey:
appKeyImport!)
        group.name = name

        group.parentGroup = importParentGroup(from: subnet, to: group)
    }

    private func importParentGroup(from subnet: SBMSubnetImport, to group: SBMGroupImport) ->
SBMGroupImport? {
        return subnet.groups.first { $0.address == parentAddress.value }
    }

    private func getAppKeyWithIndex(index: Int, from appKeys: [JsonAppKey]) -> JsonAppKey? {
        return appKeys.first { $0.index == index }
    }

    private func getSubnetWithNetKeyIndex(index: Int, from network: SBMNetworkImport) ->
SBMSubnetImport? {
        return network.subnets.first { $0.netKey.index == index }
    }
}
```

14.1.7 JsonScene

```
import BluetoothMesh

/**
 Represents Mesh Scene within JSON Mesh structure
 */
class JsonScene : Codable {
    var name : String
    var addresses : [JsonHex]
    var number : JsonHex

    init(scene : SBMScene) {
        name = scene.name
        addresses = scene.nodes.map({ (node : SBMNode) -> JsonHex in
            return JsonHex(value: UInt(node.primaryElementAddress), width: 4)
        })
        number = JsonHex(value: UInt(scene.number), width: 4)
    }

    /**
     Creates scene within BluetoothMesh importer classes
     */
    func performImport(network : SBMNetworkImport) {
        let sceneNumber = NSNumber.init(value: number.value)
        let scene = network.createScene(with: sceneNumber)
        scene.name = name
```

```
        for item in addresses {
            let node = findNodeWith(address: item.value, from: network)

            if let node = node {
                scene.addNode(node)
            }
        }
    }

    private func findNodeWith(uuid: UInt, from network: SBMNetworkImport) -> SBMNodeImport? {
        return network.nodes.first { (node) in
            let address = UInt(node.uuid.hexEncodedString(), radix: 16)!
            return address == node.primaryElementAddress
        }
    }
}
```

14.1.8 Data+HexString

```
import Foundation

/**
 Simple extension to simplify converting Data to hex encoded String.
 Used in sample classes of JSON Mesh import/export
 */
extension Data {

    struct HexEncodingOptions: OptionSet {
        let rawValue: Int
        static let upperCase = HexEncodingOptions(rawValue: 1 << 0)
    }

    func hexEncodedString(options: HexEncodingOptions = []) -> String {
        let format = options.contains(.upperCase) ? "%02hhX" : "%02hhx"
        return map { String(format: format, $0) }.joined()
    }
}
```

14.1.9 JsonElement

```
import BluetoothMesh

/**
 Represents Mesh Element within JSON Mesh structure
 */
class JsonElement : Codable {
    var index : UInt8
    var location : JsonHex
    var name : String?
    var models : [JsonModel]

    init(element: SBMElement){
        index = element.index
        location = element.location != nil ? JsonHex(value: element.location!.uintValue, width: 4))
: JsonHex(value: 0, width: 4))
        name = element.name
        models = []
        for model in element.sigModels {
            models.append(JsonModel(sigModel: model))
        }
```

```
        for model in element.vendorModels {
            models.append(JsonModel(vendorModel: model))
        }
    }

    /**
     Creates element within BluetoothMesh importer classes
     */
    func performImport(node : SBMNodeImport) {
        let element = node.createElement(with: index, andLocation: UInt32(location.value))
        element.name = name

        for item in models {
            item.performImport(node: node, element: element)
        }
    }
}
```

14.1.10 JsonFeatures

```
import BluetoothMesh

/**
 Represents Mesh Node Features within JSON Mesh structure
 */
class JsonFeatures : Codable {
    var relay : Int?
    var proxy : Int?
    var friend : Int?
    var lowPower : Int?

    init(feature : SBMFeatures, dcd : SBMDeviceCompositionData?) {
        if let dcd = dcd {
            relay = (dcd.supportsRelay()) ? feature.relayEnabled?.intValue : 2
            proxy = (dcd.supportsProxy()) ? feature.proxyEnabled?.intValue : 2
            friend = (dcd.supportsFriend()) ? feature.friendEnabled?.intValue : 2
            lowPower = (dcd.isLowPower()) ? feature.lowPower?.intValue : 2
        } else {
            relay = nil
            proxy = nil
            friend = nil
            lowPower = nil
        }
    }
}
```

14.1.11 JsonHex

```
import BluetoothMesh

/**
 Used to convert fields of JSON Mesh network which specifically require hexadecimal format.
 */
class JsonHex : Codable {
    var value : UInt
    let width : Int

    enum CodingKeys: String, CodingKey {
        case value
    }
```

```
    init(value : UInt, width : Int = 1) {
        self.value = value
        self.width = width
    }

    required init(from decoder: Decoder) throws {
        let strValue = try decoder.singleValueContainer().decode(String.self)
        value = UInt.init(strValue, radix: 16)!
        width = strValue.count
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        try container.encode(String(format: "%0\(width)X", value).uppercased())
    }
}
```

14.1.12 JsonModel

```
import BluetoothMesh

/**
 Reperesents Mesh Model (SIG or Vendor) in JSON Mesh structure
 */
class JsonModel : Codable {
    var modelId : JsonHex
    var subscribe : [JsonHex]
    var publish : JsonPublish?
    var bind : Set<Int>
    var knownAddresses: [JsonHex]?

    init(sigModel : SBMSigModel){
        modelId = JsonHex(value: UInt(sigModel.identifier), width: 4)
        subscribe = [JsonHex]()
        publish = nil
        bind = Set()
        knownAddresses = [JsonHex]()

        initFields(model: sigModel)
    }

    init(vendorModel : SBMVendorModel){
        modelId = JsonHex(value: UInt(vendorModel.identifier), width: 8)
        subscribe = [JsonHex]()
        publish = nil
        bind = Set()
        knownAddresses = [JsonHex]()

        initFields(model: vendorModel)
    }

    private func initFields(model : SBMModel) {
        for item in model.bindedGroups {
            let appKeyIndex = Int(item.appKey.keyIndex)
            bind.insert(appKeyIndex)

            let groupAddress = JsonHex(value:UInt(item.address), width: 4)
            knownAddresses?.append(groupAddress)
        }

        for item in model.modelSettings.subscriptions {
```

```
            let subscription = JsonHex(value: UInt(item.value), width: 4)
            subscribe.append(subscription)
        }

        if let publish = model.modelSettings.publish {
            self.publish = JsonPublish(publish: publish)
        }
    }

    /**
     Creates model within BluetoothMesh importer classes
     */
    func performImport(node: SBMNodeImport, element : SBMElementImport) {
        let model = element.createModel(Int(modelId.value))
        let settings = model.settings


        for item in subscribe {
            settings.createSubscription(withAddress: UInt32(item.value))
        }

        importGroups(from: node, to: model, with: settings)
        publish?.performImport(settings: settings)
    }

    private func importGroups(from node: SBMNodeImport, to model: SBMModelImport, with settings:
SBMModelSettingsImport) {
        for item in bind {
            let groups = findGroupsWith(appKeyIndex: item, from: node)

            for group in groups {
                model.addGroup(group)
            }
        }
    }

    private func findGroupsWith(appKeyIndex: Int, from node: SBMNodeImport) -> [SBMGroupImport] {
        return node.groups.filter { group -> Bool in
            guard let groupAddressIsKnown = knownAddresses?.contains(where: { $0.value ==
group.address }) else {
                return group.appKey.index == appKeyIndex
            }

            return (group.appKey.index == appKeyIndex) && groupAddressIsKnown
        }
    }
}
```

14.1.13 JsonNodeKey

```
import BluetoothMesh

/**
 Represents status data of Node keys (updated during key refresh procedure) in JSON Mesh structure
 */
class JsonNodeKey : Codable {
    var index : Int
    var updated : Bool

    init(appKey: SBMNodeAppKey){
        index = Int(appKey.appKey)
        updated = appKey.isUpdated
```

```
    }

    init(netKey: SBMNodeNetKey){
        index = Int(netKey.netKey)
        updated = netKey.isUpdated
    }
}
```

14.1.14 JsonPublish

```
import BluetoothMesh

/**
 Represents Mesh Model publishing information in JSON Mesh structure
 */
class JsonPublish : Codable {
    var address : JsonHex
    var ttl : UInt
    var period : UInt
    var retransmit : JsonRetransmit
    var credentials : UInt

    init(publish: SBMPublish) {
        address = JsonHex(value: UInt(publish.address.value), width: 4)
        index = 0
        ttl = UInt(publish.ttl)
        period = UInt(publish.period)
        retransmit = JsonRetransmit(transmit: publish.retransmit)
        credentials = publish.credentials.rawValue
    }

    func performImport(settings: SBMModelSettingsImport) {
        let publish = settings.createPublish()

        publish.createAddress(UInt32(address.value))
        publish.ttl = UInt8(ttl)
        publish.period = UInt32(period)
        if retransmit.count > 0 {
            publish.createRetransmit(withCount: retransmit.count, interval: retransmit.interval)
        }

        publish.credentials = SBMCredentialsType.init(rawValue: credentials)!
    }
}
```

14.1.15 JsonRetransmit

```
import BluetoothMesh

/**
 Represents retransmission configuration within JSON Mesh structure
 */
class JsonRetransmit : Codable {
    var count : Int
    var interval : Int

    init(transmit : SBMTransmit){
        count = transmit.count
        interval = transmit.interval
    }
}
```

14.1.16 JsonSecurityType

```
import BluetoothMesh

/**
 Represents Mesh Node or Network security type within JSON Mesh structure
 */
class JsonSecurityType : Codable {
    var type : SBMSecurityType

    enum CodingKeys: String, CodingKey {
        case type
    }

    init(type : SBMSecurityType) {
        self.type = type
    }

    required init(from decoder: Decoder) throws {
        let strValue = try decoder.singleValueContainer().decode(String.self)

        if strValue == "high" {
            type = SBMSecurityType.high
        } else if strValue == "low" {
            type = SBMSecurityType.low
        } else {
            throw NSError()
        }
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        if(type == SBMSecurityType.high) {
            try container.encode("high")
        } else if(type == SBMSecurityType.low) {
            try container.encode("low")
        }
    }
}
```

14.1.17 JsonTimeStamp

```
import BluetoothMesh

/**
 Represents timestamps used in JSON Mesh structure.
 */
class JsonTimestamp : Codable {
    var timestamp : Int

    enum CodingKeys: String, CodingKey {
        case timestamp
    }

    init(timestamp : Int) {
        self.timestamp = timestamp
    }

    required init(from decoder: Decoder) throws {
        let strValue = try decoder.singleValueContainer().decode(String.self)
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
        let date = dateFormatter.date(from: strValue)!
```

```
        timestamp = Int(date.timeIntervalSince1970)

        Logger.log(message: "Date: \(strValue) -> \(timestamp)")
    }

    func encode(to encoder: Encoder) throws {
        let date = Date(timeIntervalSince1970: TimeInterval(timestamp))
        let df = DateFormatter()
        df.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"

        var container = encoder.singleValueContainer()
        try container.encode(df.string(from: date))
    }
}
```

14.1.18 String+Hex

```
import Foundation

/**
 Simple extension used to simplify converting hex encoded string into Data object
 */
extension String {
    enum ExtendedEncoding {
        case hexadecimal
    }

    func data(using encoding: ExtendedEncoding) -> Data? {
        let hexStr = self.dropFirst(self.hasPrefix("0x") ? 2 : 0)

        guard hexStr.count % 2 == 0 else { return nil }

        var newData = Data(capacity: hexStr.count/2)

        var indexIsEven = true
        for i in hexStr.indices {
            if indexIsEven {
                let byteRange = i...hexStr.index(after: i)
                guard let byte = UInt8(hexStr[byteRange], radix: 16) else { return nil }
                newData.append(byte)
            }
            indexIsEven.toggle()
        }
        return newData
    }
}
```

14.2 Android

14.2.1 JsonImporter

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport

import com.siliconlab.bluetoothmesh.adk.BluetoothMesh
import com.siliconlab.bluetoothmesh.adk.data_model.Security
import com.siliconlab.bluetoothmesh.adk.data_model.model.Credentials
import com.siliconlab.bluetoothmesh.adk.importer.*
import com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects.*
import java.util.*
```

```
class JsonImporter(private val jsonMesh: JsonMesh) {

    private val appKeyImports = HashSet<AppKeyImport>()

    fun import() {
        val importer = createImporter()
        BluetoothMesh.getInstance().clearDatabase()
        importer.performImport()
    }

    private fun createImporter(): Importer {
        val importer = Importer()
        importer.createNetwork(Converter.stringToUuid(jsonMesh.meshUUID!!))
                .apply { name = jsonMesh.meshName }
                .also {
                    handleAppKeys()
                    handleSubnets(it)
                    handleNodes(it)
                    handleProvisioners(it)
                    handleScenes(it)
                }

        return importer
    }

    private fun handleAppKeys() {
        jsonMesh.appKeys.map { jsonAppKey ->
            AppKeyImport(
                    jsonAppKey.index,
                    Converter.hexToBytes(jsonAppKey.key),
                    Converter.hexToBytes(jsonAppKey.oldKey))
                    .apply { name = jsonAppKey.name }
        }.forEach { appKeyImports.add(it) }
    }

    private fun handleSubnets(networkImport: NetworkImport) {
        jsonMesh.netKeys.forEach { jsonNetKey ->
            val subnetImport = createSubnetImport(jsonNetKey, networkImport)
            handleGroups(subnetImport)
            subnetImport.createSubnetSecurity(
                    jsonNetKey.phase,
                    Converter.timestampToLong(jsonNetKey.timestamp) ?: 0)
        }
    }

    private fun createSubnetImport(jsonNetKey: JsonNetKey, networkImport: NetworkImport):
SubnetImport {
        val netKeyImport = NetKeyImport(
                jsonNetKey.index,
                Converter.hexToBytes(jsonNetKey.key),
                Converter.hexToBytes(jsonNetKey.oldKey))

        return networkImport.createSubnet(netKeyImport)
                .apply { name = jsonNetKey.name }
    }

    private fun handleGroups(subnetImport: SubnetImport) {
        jsonMesh.groups.filter { jsonGroup ->
            val appKey = jsonMesh.appKeys.find { it.index == jsonGroup.appKeyIndex }
            appKey?.boundNetKey == subnetImport.netKey.index
        }.forEach { createGroupImport(subnetImport, it) }

        handleParentGroups(jsonMesh.groups, subnetImport.groups)
```

```
    }

    private fun createGroupImport(subnetImport: SubnetImport, jsonGroup: JsonGroup) {
        if (Converter.isVirtualAddress(jsonGroup.address!!)) {
//            Creating groups with virtual addresses is not supported currently
        } else {
            val groupImport = subnetImport.createGroup(
                    Converter.hexToInt(jsonGroup.address)!!,
                    findAppKeyImport(jsonGroup.appKeyIndex))
            groupImport.name = jsonGroup.name
        }
    }

    private fun findAppKeyImport(appKeyIndex: Int): AppKeyImport? {
        return appKeyImports.find { it.index == appKeyIndex }
    }

    private fun handleParentGroups(jsonGroups: Array<JsonGroup>, groupImports: Set<GroupImport>) {
        jsonGroups.forEach { jsonGroup ->
            jsonGroup.parentAddress?.let { parentGroupAddress ->
                jsonGroups.find { it.address == parentGroupAddress }?.let {
                    val child = findGroupImport(jsonGroup, groupImports)
                    val parent = findGroupImport(it, groupImports)

                    child?.parentGroup = parent
                }
            }
        }
    }

    private fun findGroupImport(jsonGroup: JsonGroup, groupImports: Set<GroupImport>): GroupImport?
{
        return groupImports.find { Converter.hexToInt(jsonGroup.address) == it.address }
    }

    private fun handleNodes(networkImport: NetworkImport) {
        jsonMesh.nodes.forEach { jsonNode ->
            val nodeImport = createNodeImport(networkImport, jsonNode)

            fillNodeSubnets(networkImport, nodeImport, jsonNode)
            fillNodeGroups(nodeImport, jsonNode)
            handleElements(nodeImport, nodeImport.groups, jsonNode)
            handleDeviceCompositionData(nodeImport, jsonNode)
            handleNodeSettings(nodeImport.settings, jsonNode)
            handleNodeSecurity(nodeImport.security, jsonNode)
        }
    }

    private fun createNodeImport(networkImport: NetworkImport, jsonNode: JsonNode): NodeImport {
        return networkImport.createNode(
                Converter.hexToBytes(jsonNode.UUID),
                Converter.hexToInt(jsonNode.unicastAddress)!!,
                DevKeyImport(Converter.hexToBytes(jsonNode.deviceKey)))
                .apply { name = jsonNode.name }
    }

    private fun fillNodeSubnets(networkImport: NetworkImport, nodeImport: NodeImport, jsonNode:
JsonNode) {
        networkImport.subnets
                .filter { subnet ->
                    jsonNode.netKeys.any { subnet.netKey.index == it.index }
                }.forEach { nodeImport.addSubnet(it) }
    }
```

```
    private fun fillNodeGroups(nodeImport: NodeImport, jsonNode: JsonNode) {
        nodeImport.subnets.flatMap { it.groups }
                .filter { group ->
                    jsonNode.knownAddresses?.any { Converter.hexToInt(it) == group.address } !=
false
                            && jsonNode.appKeys.any { it.index == group.appKey.index }
                }.forEach { nodeImport.addGroup(it) }
    }

    private fun handleElements(nodeImport: NodeImport, nodeImportGroups: Set<GroupImport>,
jsonNode: JsonNode) {
        jsonNode.elements.forEach {
            val elementImport = nodeImport
                    .createElement(it.index, Converter.hexToInt(it.location)!!)
                    .apply { name = it.name }

            handleModels(elementImport, nodeImportGroups, it)
        }
    }

    private fun handleModels(elementImport: ElementImport, nodeImportGroups: Set<GroupImport>,
jsonElement: JsonElement) {
        jsonElement.models.forEach {
            val modelImport = elementImport.createModel(Converter.hexToInt(it.modelId)!!)
            fillModelGroups(modelImport, nodeImportGroups, it)
            handleModelSettings(modelImport.settings, it)
        }
    }

    private fun fillModelGroups(modelImport: ModelImport, nodeImportGroups: Set<GroupImport>,
jsonModel: JsonModel) {
        nodeImportGroups
                .filter { group ->
                    jsonModel.knownAddresses?.any { Converter.hexToInt(it) == group.address } !=
false
                            && jsonModel.bind.contains(group.appKey.index)
                }.forEach { modelImport.addGroup(it) }
    }

    private fun handleModelSettings(modelSettingsImport: ModelSettingsImport, jsonModel: JsonModel)
{
        modelSettingsImport.apply {
            handlePublish(this, jsonModel.publish)

            jsonModel.subscribe.forEach { address ->
                if (Converter.isVirtualAddress(address)) {
                    createSubscription(Converter.hexToBytes(address))
                } else {
                    createSubscription(Converter.hexToInt(address)!!)
                }
            }
        }
    }

    private fun handlePublish(modelSettingsImport: ModelSettingsImport, jsonPublish: JsonPublish?)
{
        jsonPublish?.let {
            modelSettingsImport.createPublish().apply {
                ttl = it.ttl
                period = it.period
                credentials = Credentials.fromValue(it.credentials)
```

```
            if (Converter.isVirtualAddress(it.address!!)) {
                createAddress(Converter.hexToBytes(it.address))
            } else {
                createAddress(Converter.hexToInt(it.address)!!)
            }

            it.retransmit?.let { createRetransmit(it.count, it.interval) }
        }
    }
}

private fun handleDeviceCompositionData(nodeImport: NodeImport, jsonNode: JsonNode) {
    jsonNode.features?.let { jsonFeatures ->
        val isEveryFieldNotNull = jsonFeatures.friend != null && jsonFeatures.relay != null &&
                jsonFeatures.proxy != null && jsonFeatures.lowPower != null &&
                jsonNode.cid != null && jsonNode.pid != null && jsonNode.vid != null &&
                jsonNode.crpl != null

        if (isEveryFieldNotNull) {
            val deviceCompositionData = nodeImport.createDeviceCompositionData()
            deviceCompositionData.supportsRelay = isFeatureSupported(jsonFeatures.relay)
            deviceCompositionData.supportsProxy = isFeatureSupported(jsonFeatures.proxy)
            deviceCompositionData.supportsFriend = isFeatureSupported(jsonFeatures.friend)
            deviceCompositionData.supportsLowPower = isFeatureSupported(jsonFeatures.lowPower)
            deviceCompositionData.cid = Converter.hexToInt(jsonNode.cid)
            deviceCompositionData.pid = Converter.hexToInt(jsonNode.pid)
            deviceCompositionData.vid = Converter.hexToInt(jsonNode.vid)
            deviceCompositionData.crpl = Converter.hexToInt(jsonNode.crpl)
        }
    }
}

private fun isFeatureSupported(featureState: Int?): Boolean? {
    return when (featureState) {
        0, 1 -> true
        2 -> false
        else -> null
    }
}

private fun handleNodeSettings(nodeSettings: NodeSettingsImport, jsonNode: JsonNode) {
    nodeSettings.isConfigComplete = jsonNode.configComplete
    nodeSettings.defaultTTL = jsonNode.defaultTTL

    jsonNode.features?.let { handleFeatures(nodeSettings.createFeatures(), it) }
    jsonNode.networkTransmit?.let { nodeSettings.createNetworkTransmit(it.count, it.interval) }
    jsonNode.relayRetransmit?.let { nodeSettings.createRelayRetransmit(it.count, it.interval) }
}

private fun handleFeatures(featuresImport: FeaturesImport, jsonFeature: JsonFeature) {
    featuresImport.isRelayEnabled = isFeatureEnabled(jsonFeature.relay)
    featuresImport.isProxyEnabled = isFeatureEnabled(jsonFeature.proxy)
    featuresImport.isFriendEnabled = isFeatureEnabled(jsonFeature.friend)
    featuresImport.isLowPower = isFeatureEnabled(jsonFeature.lowPower)
}

private fun isFeatureEnabled(featureState: Int?): Boolean? {
    return when (featureState) {
        0 -> false
        1 -> true
        else -> null
    }
}
```

```
    private fun handleNodeSecurity(nodeSecurity: NodeSecurityImport, jsonNode: JsonNode) {
        jsonNode.appKeys.forEach { nodeSecurity.createNodeAppKey(it.index, it.updated) }
        jsonNode.netKeys.forEach { nodeSecurity.createNodeNetKey(it.index, it.updated) }

        nodeSecurity.isBlacklisted = jsonNode.blacklisted
        nodeSecurity.security = Security.valueOf(jsonNode.security!!.toUpperCase(Locale.ROOT))
        nodeSecurity.isSecureNetworkBeacon = jsonNode.secureNetworkBeacon
    }

    private fun handleProvisioners(networkImport: NetworkImport) {
        jsonMesh.provisioners.forEach { jsonProvisioner ->
            networkImport.createProvisioner(Converter.stringToUuid(jsonProvisioner.UUID!!))
                    .apply { name = jsonProvisioner.provisionerName }
                    .also {
                        handleUnicastRanges(it, jsonProvisioner.allocatedUnicastRange)
                        handleGroupRanges(it, jsonProvisioner.allocatedGroupRange)
                        handleSceneRanges(it, jsonProvisioner.allocatedSceneRange)
                    }
        }
    }

    private fun handleUnicastRanges(provisionerImport: ProvisionerImport, jsonUnicastRanges:
Array<JsonAddressRange>) {
        jsonUnicastRanges.forEach {
            val low = Converter.hexToInt(it.lowAddress)!!
            val high = Converter.hexToInt(it.highAddress)!!

            provisionerImport.createUnicastRange(low, high)
        }
    }

    private fun handleGroupRanges(provisionerImport: ProvisionerImport, jsonGroupRanges:
Array<JsonAddressRange>) {
        jsonGroupRanges.forEach {
            val low = Converter.hexToInt(it.lowAddress)!!
            val high = Converter.hexToInt(it.highAddress)!!

            provisionerImport.createGroupRange(low, high)
        }
    }

    private fun handleSceneRanges(provisionerImport: ProvisionerImport, jsonSceneRanges:
Array<JsonSceneRange>) {
        jsonSceneRanges.forEach {
            val low = Converter.hexToInt(it.firstScene)!!
            val high = Converter.hexToInt(it.lastScene)!!

            provisionerImport.createSceneRange(low, high)
        }
    }

    private fun handleScenes(networkImport: NetworkImport) {
        jsonMesh.scenes.forEach { jsonScene ->
            val sceneImport = networkImport
                    .createScene(Converter.hexToInt(jsonScene.number)!!)
                    .apply { name = jsonScene.name }
            jsonScene.addresses.mapNotNull { address ->
                networkImport.nodes.find {
                    it.primaryElementAddress == Converter.hexToInt(address)
                }
            }.forEach { sceneImport.addNode(it) }
        }
```

```
    }
}
```

14.2.2 JsonExporter

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport

import com.google.gson.GsonBuilder
import com.siliconlab.bluetoothmesh.adk.BluetoothMesh
import com.siliconlab.bluetoothmesh.adk.data_model.dcd.DeviceCompositionData
import com.siliconlab.bluetoothmesh.adk.data_model.element.Element
import com.siliconlab.bluetoothmesh.adk.data_model.model.Model
import com.siliconlab.bluetoothmesh.adk.data_model.model.Retransmit
import com.siliconlab.bluetoothmesh.adk.data_model.node.NetworkTransmit
import com.siliconlab.bluetoothmesh.adk.data_model.node.Node
import com.siliconlab.bluetoothmesh.adk.data_model.node.RelayRetransmit
import com.siliconlab.bluetoothmesh.adk.data_model.provisioner.AddressRange
import com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects.*
import java.util.*

class JsonExporter {

    fun export(): String {
        val jsonMesh = createJsonMesh()
        val gson = GsonBuilder().setPrettyPrinting().create()
        return gson.toJson(jsonMesh)
    }

    private val network = BluetoothMesh.getInstance().networks.iterator().next()

    private fun createJsonMesh(): JsonMesh {
        return JsonMesh().apply {
            schema = "http://json-schema.org/draft-04/schema#"
            id = "https://www.bluetooth.com/specifications/assigned-numbers/mesh-profile/cdb-
schema.json#"
            version = network.version
            meshUUID = Converter.uuidToString(network.uuid)
            meshName = network.name
            timestamp = Converter.longToTimestamp(System.currentTimeMillis())
            provisioners = createJsonProvisioners()
            netKeys = createJsonNetKeys()
            appKeys = createJsonAppKeys()
            nodes = createJsonNodes()
            groups = createJsonGroups()
            scenes = createJsonScenes()
        }
    }

    private fun createJsonScenes(): Array<JsonScene> {
        return network.scenes.map {
            JsonScene().apply {
                name = it.name
                number = Converter.intToHex(it.number, 4)
                addresses = it.nodes
                        .map { Converter.intToHex(it.primaryElementAddress, 4)!! }
                        .toTypedArray()
            }
        }.toTypedArray()
    }

    private fun createJsonNodes(): Array<JsonNode> {
        return network.subnets.flatMap { it.nodes }.map {
```

```
            JsonNode().apply {
                UUID = Converter.bytesToHex(it.uuid)
                unicastAddress = Converter.intToHex(it.primaryElementAddress, 4)
                deviceKey = Converter.bytesToHex(it.devKey.key)
                security = it.nodeSecurity.security.name.toLowerCase(Locale.ROOT)
                netKeys = createJsonNodeNetKeys(it)
                configComplete = it.nodeSettings.isConfigComplete
                name = it.name
                cid = Converter.intToHex(it.deviceCompositionData.cid, 4)
                pid = Converter.intToHex(it.deviceCompositionData.pid, 4)
                vid = Converter.intToHex(it.deviceCompositionData.vid, 4)
                crpl = Converter.intToHex(it.deviceCompositionData.crpl, 4)
                features = createJsonFeatures(it)
                secureNetworkBeacon = it.nodeSecurity.isSecureNetworkBeacon
                defaultTTL = it.nodeSettings.defaultTTL
                networkTransmit = createJsonNetworkTransmit(it.nodeSettings.networkTransmit)
                relayRetransmit = createJsonRelayRetransmit(it.nodeSettings.relayRetransmit)
                appKeys = createJsonNodeAppKeys(it)
                elements = createJsonElements(it)
                blacklisted = it.nodeSecurity.isBlacklisted
                knownAddresses = fillGroupsInJsonNode(it)
            }
        }.toTypedArray()
    }

    private fun fillGroupsInJsonNode(node: Node): Array<String> {
        return node.groups
                .map { Converter.intToHex(it.address)!! }
                .toTypedArray()
    }

    private fun createJsonRelayRetransmit(relayRetransmit: RelayRetransmit?): JsonRelayRetransmit?
{
        return relayRetransmit?.let {
            JsonRelayRetransmit().apply {
                count = relayRetransmit.count
                interval = relayRetransmit.interval
            }
        }
    }

    private fun createJsonNetworkTransmit(networkTransmit: NetworkTransmit?): JsonNetworkTransmit?
{
        return networkTransmit?.let {
            JsonNetworkTransmit().apply {
                count = networkTransmit.count
                interval = networkTransmit.interval
            }
        }
    }

    private fun createJsonFeatures(node: Node): JsonFeature? {
        val features = node.nodeSettings.features
        val dcd = node.deviceCompositionData
        return JsonFeature().apply {
            relay = convertFeatureState(dcd?.supportsRelay(), features.isRelayEnabled)
            friend = convertFeatureState(dcd?.supportsFriend(), features.isFriendEnabled)
            lowPower = convertLowPowerFeatureState(dcd, features.isLowPower)
            proxy = convertFeatureState(dcd?.supportsProxy(), features.isProxyEnabled)
        }
    }

    private fun convertFeatureState(supports: Boolean?, enabled: Boolean?): Int? {
```

```
        return when {
            supports == false -> 2
            enabled == null -> null
            enabled -> 1
            else -> 0
        }
    }

    private fun convertLowPowerFeatureState(dcd: DeviceCompositionData?, feature: Boolean?): Int? {
        return when {
            dcd != null && !dcd.supportsLowPower() || feature != null && !feature -> 2
            dcd != null && dcd.supportsLowPower() || feature != null && feature -> 1
            else -> null
        }
    }

    private fun createJsonElements(node: Node): Array<JsonElement> {
        return node.elements.map {
            JsonElement().apply {
                name = it.name
                index = it.index
                location = Converter.intToHex(it.location, 4)
                models = createJsonModels(it)
            }
        }.toTypedArray()
    }

    private fun createJsonModels(element: Element): Array<JsonModel> {
        return element.sigModels.union(element.vendorModels).map {
            val idWidth = if (it.isSIGModel) 4 else 8
            JsonModel().apply {
                modelId = Converter.intToHex(it.id, idWidth)
                subscribe = createSubscribe(it)
                publish = createJsonPublish(it)
                bind = createBind(it)
                knownAddresses = fillGroupsInJsonModel(it)
            }
        }.toTypedArray()
    }

    private fun fillGroupsInJsonModel(model: Model): Array<String> {
        return model.bindedGroups
                .map { group -> Converter.intToHex(group.address)!! }
                .toTypedArray()
    }

    private fun createBind(model: Model): Array<Int> {
        return model.bindedGroups
                .map { it.appKey.keyIndex }
                .toTypedArray()
    }

    private fun createSubscribe(model: Model): Array<String> {
        return model.modelSettings.subscriptions
                .map { Converter.addressToHex(it)!! }
                .toTypedArray()
    }

    private fun createJsonPublish(model: Model): JsonPublish? {
        return model.modelSettings.publish?.let {
            JsonPublish().apply {
                address = Converter.addressToHex(it.address)
                ttl = it.ttl
```

```
            period = it.period
            credentials = it.credentials.value
            retransmit = createJsonRetransmit(it.retransmit)
        }
    }
}

private fun createJsonRetransmit(retransmit: Retransmit): JsonRetransmit {
    return JsonRetransmit().apply {
        count = retransmit.count
        interval = retransmit.interval
    }
}

private fun createJsonNodeAppKeys(node: Node): Array<JsonNodeKey> {
    return node.nodeSecurity.appKeys.map {
        JsonNodeKey().apply {
            index = it.appKeyIndex
            updated = it.isUpdated
        }
    }.toTypedArray()
}

private fun createJsonNodeNetKeys(node: Node): Array<JsonNodeKey> {
    return node.nodeSecurity.netKeys.map {
        JsonNodeKey().apply {
            index = it.netKeyIndex
            updated = it.isUpdated
        }
    }.toTypedArray()
}

private fun createJsonAppKeys(): Array<JsonAppKey> {
    return network.subnets.flatMap { it.groups }.map {
        JsonAppKey().apply {
            name = it.appKey.name ?: ""
            index = it.appKey.keyIndex
            key = Converter.bytesToHex(it.appKey.key)
            oldKey = Converter.bytesToHex(it.appKey.oldKey)
            boundNetKey = it.subnet.netKey.keyIndex
        }
    }.toTypedArray()
}

private fun createJsonGroups(): Array<JsonGroup> {
    return network.subnets.flatMap { it.groups }.map {
        JsonGroup().apply {
            name = it.name
            address = Converter.intToHex(it.address, 4)
            appKeyIndex = it.appKey.keyIndex
            parentAddress = Converter.intToHex(it.parentGroup?.address ?: 0, 4)
        }
    }.toTypedArray()
}

private fun createJsonNetKeys(): Array<JsonNetKey> {
    return network.subnets.map {
        JsonNetKey().apply {
            name = it.name
            index = it.netKey.keyIndex
            phase = it.subnetSecurity.keyRefreshPhase ?: 0
            timestamp = Converter.longToTimestamp(it.subnetSecurity.keyRefreshTimestamp)
            key = Converter.bytesToHex(it.netKey.key)
```

```
                minSecurity = it.subnetSecurity.minSecurity.name.toLowerCase(Locale.ROOT)
                oldKey = Converter.bytesToHex(it.netKey.oldKey)
            }
        }.toTypedArray()
    }

    private fun createJsonProvisioners(): Array<JsonProvisioner> {
        return network.provisioners.map {
            JsonProvisioner().apply {
                provisionerName = it.name
                UUID = Converter.uuidToString(it.uuid)
                allocatedUnicastRange = createJsonAddressRange(it.allocatedUnicastRange)
                allocatedGroupRange = createJsonAddressRange(it.allocatedGroupRange)
                allocatedSceneRange = createJsonSceneRange(it.allocatedSceneRange)
            }
        }.toTypedArray()
    }

    private fun createJsonAddressRange(ranges: List<AddressRange>): Array<JsonAddressRange> {
        return ranges.map {
            JsonAddressRange().apply {
                lowAddress = Converter.intToHex(it.lowAddress, 4)
                highAddress = Converter.intToHex(it.highAddress, 4)
            }
        }.toTypedArray()
    }

    private fun createJsonSceneRange(ranges: List<AddressRange>): Array<JsonSceneRange> {
        return ranges.map {
            JsonSceneRange().apply {
                firstScene = Converter.intToHex(it.lowAddress, 4)
                lastScene = Converter.intToHex(it.highAddress, 4)
            }
        }.toTypedArray()
    }
}
```

14.2.3 Converter

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport

import android.util.Log
import com.siliconlab.bluetoothmesh.adk.data_model.model.Address
import java.text.ParseException
import java.text.SimpleDateFormat
import java.util.*

object Converter {

    internal fun hexToBytes(hexString: String?): ByteArray? {
        if (hexString == null) return null

        if (hexString.length % 2 != 0) {
            Log.e("Converter", "hexToBytes: invalid hexString")
            return null
        }

        return ByteArray(hexString.length / 2) {
            hexString.substring(it * 2, it * 2 + 2).toInt(16).toByte()
        }
    }
```

```
    internal fun bytesToHex(bytes: ByteArray?): String? {
        if (bytes == null) return null

        val sb = StringBuilder()
        bytes.forEach { sb.append(String.format("%02x", it)) }
        return sb.toString()
    }

    internal fun hexToInt(hexString: String?): Int? {
        return hexString?.let { Integer.parseInt(it, 16) }
    }

    internal fun intToHex(value: Int?, width: Int = 1): String? {
        return value?.let { String.format("%1$0${width}X", it) }
    }

    internal fun isVirtualAddress(hexString: String): Boolean {
        return hexString.length == 32
    }

    internal fun stringToUuid(uuid: String): UUID {
        return UUID.fromString(uuid.substring(0, 8) + "-" + uuid.substring(8, 12) + "-" +
                uuid.substring(12, 16) + "-" + uuid.substring(16, 20) + "-" + uuid.substring(20,
32))
    }

    internal fun timestampToLong(timestamp: String?): Long? {
        try {
            return timestamp?.let { SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").parse(it).time }
        } catch (e: ParseException) {
            e.printStackTrace()
        }

        return null
    }

    fun longToTimestamp(timestamp: Long?): String? {
        return timestamp?.let { SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(Date(it)) }
    }

    fun addressToHex(address: Address): String? {
        return intToHex(address.value, 4) ?: bytesToHex(address.virtualLabel)
    }

    fun uuidToString(uuid: UUID): String {
        return uuid.toString().replace("-", "")
    }
}
```

14.2.4 JsonMesh

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

import com.google.gson.annotations.SerializedName

class JsonMesh {
    @SerializedName("\$schema")
    var schema: String? = null
    var version: String? = null
    var meshUUID: String? = null
    var meshName: String? = null
    var timestamp: String? = null
```

```
    var provisioners: Array<JsonProvisioner> = emptyArray()
    var netKeys: Array<JsonNetKey> = emptyArray()
    var appKeys: Array<JsonAppKey> = emptyArray()
    var nodes: Array<JsonNode> = emptyArray()
    var groups: Array<JsonGroup> = emptyArray()
    var scenes: Array<JsonScene> = emptyArray()
}
```

14.2.5 JsonNetKey

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonNetKey {
    var name: String? = null
    var index = 0
    var phase = 0
    var key: String? = null
    var minSecurity: String? = null
    var oldKey: String? = null
    var timestamp: String? = null
}
```

14.2.6 JsonAppKey

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonAppKey {
    var name: String? = null
    var index: Int = 0
    var boundNetKey: Int = 0
    var key: String? = null
    var oldKey: String? = null
}
```

14.2.7 JsonProvisioner

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonProvisioner {
    var provisionerName: String? = null
    var UUID: String? = null
    var allocatedUnicastRange = arrayOf<JsonAddressRange>()
    var allocatedGroupRange = arrayOf<JsonAddressRange>()
    var allocatedSceneRange = arrayOf<JsonSceneRange>()
}
```

14.2.8 JsonNode

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonNode {
    var UUID: String? = null
    var unicastAddress: String? = null
    var deviceKey: String? = null
    var security: String? = null
    var netKeys: Array<JsonNodeKey> = emptyArray()
    var configComplete: Boolean = false
    var name: String? = null
    var cid: String? = null
```

```
    var pid: String? = null
    var vid: String? = null
    var crpl: String? = null
    var features: JsonFeature? = null
    var secureNetworkBeacon: Boolean? = null
    var defaultTTL: Int? = null
    var networkTransmit: JsonNetworkTransmit? = null
    var relayRetransmit: JsonRelayRetransmit? = null
    var appKeys: Array<JsonNodeKey> = emptyArray()
    var elements: Array<JsonElement> = emptyArray()
    var blacklisted: Boolean = false
    var knownAddresses: Array<String>? = null
}
```

14.2.9 JsonGroup

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonGroup {
    var name: String? = null
    var address: String? = null
    var parentAddress: String? = null
    var appKeyIndex = 0
}
```

14.2.10 JsonScene

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonScene {
    var name: String? = null
    var number: String? = null
    var addresses: Array<String> = emptyArray()
}
```

14.2.11 JsonAddressRange

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonAddressRange {
    var lowAddress: String? = null
    var highAddress: String? = null
}
```

14.2.12 JsonElement

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonElement {
    var name: String? = null
    var index: Int = 0
    var location: String? = null
    var models: Array<JsonModel> = emptyArray()
}
```

14.2.13 JsonFeature

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonFeature {
    var relay: Int? = null
    var proxy: Int? = null
    var lowPower: Int? = null
    var friend: Int? = null
}
```

14.2.14 JsonModel

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonModel {
    var modelId: String? = null
    var subscribe: Array<String> = emptyArray()
    var publish: JsonPublish? = null
    var bind: Array<Int> = emptyArray()
    var knownAddresses: Array<String>? = null
}
```

14.2.15 JsonNetworkTransmit

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonNetworkTransmit {
    var count: Int = 0
    var interval: Int = 0
}
```

14.2.16 JsonNodeKey

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonNodeKey {
    var index: Int = 0
    var updated: Boolean? = null
}
```

14.2.17 JsonPublish

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonPublish {
    var address: String? = null
    var index: Int = 0
    var ttl: Int = 0
    var period: Int = 0
    var credentials: Int = 0
    var retransmit: JsonRetransmit? = null
}
```

14.2.18 JsonRelayTransmit

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonRelayRetransmit {
    var count: Int = 0
    var interval: Int = 0
}
```

14.2.19 JsonRetransmit

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonRetransmit {
    var count: Int = 0
    var interval: Int = 0
}
```

14.2.20 JsonSceneRange

```
package com.siliconlabs.bluetoothmesh.App.Logic.ImportExport.JsonObjects

class JsonSceneRange {
    var firstScene: String? = null
    var lastScene: String? = null
}
```

Smart. Connected. Energy-Friendly.

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com