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

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

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

KEY POINTS

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

Table of Contents

1 Introduction 4
2 Prerequisites for Development 6
2.1 iOS 6
2.2 Android 6
3 Contents of Bluetooth mesh for iOS and Android ADK 8
3.1 The Bluetooth Mesh Stack library 8
3.1.1 iOS 8
3.1.2 Android 8
3.2 Bluetooth mesh Network and Device Database 8
3.3 Reference Application Source Code 8
3.4 Documentation 8
4 Getting Started with Development 9
4.1 iOS 9
4.2 Android 11
5 Bluetooth Mesh Structure Overview 15
6 Bluetooth mesh API Reference for iOS 16
6.1 Initializing the BluetoothMesh 16
6.1.1 BluetoothMesh 16
6.1.2 Set Up Supported Vendor Models 16
6.1.3 Set Up Mesh Limits 16
6.2 Set Up Bluetooth Layer (ConnectableDevice) 17
6.2.1 Device Advertisement Data 18
6.2.2 Device UUID 18
6.2.3 Device Name 18
6.2.4 Device Connection State 18
6.2.5 Connect to the Device 18
6.2.6 Disconnect from the Device 19
6.2.7 Check if a Device Contains a Service 19
6.2.8 Maximum Transmission Unit for Given Device Service 19
6.2.9 Write Data to a Given Service and Characteristic 20
6.2.10 Subscribe to a Given Service and Characteristic 20
6.3 Provision a Device to a Subnet 21
6.3.1 Create Network 21
6.3.2 Create Subnet 21
6.3.3 Find Non-Provisioned Bluetooth Devices 21

6.3.4 Provision the Devices 22
6.4 Add a Node to a Subnet 22
6.5 Remove a Node from a Subnet 22
6.6 Connect with a Subnet 23
6.6.1 Find All Proxies in the Device Range 23
6.6.2 Get Node Representing a Given Device 23
6.6.3 Get Node Representing a Given Device in a Specific Subnet 23
6.7 Create a Group in a Given Subnet 23
6.8 Remove Group 24
6.9 Add a Node to a Group 24
6.10 Remove a Node from a Group 24
6.11 Bind a Model with A Group 25
6.12 Unbind a Model from a Group 25
6.13 Add Subscription Settings to a Model 25
6.13.1 SIG Model 25
6.13.2 Vendor Model 26
6.14 Add Publication Settings to a Model 26
6.14.1 SIG Model 26
6.14.2 Vendor Model 27
6.15 Control Node Functionality 28
6.15.1 Get Value for a Single SIG Model from the Node 28
6.15.2 Get Value for All Specific SIG Models Bound with a Group 28
6.15.3 Set Value for a Single SIG Model from the Node 28
6.15.4 Set Value for All Specific SIG Models Bound with the Group 29
6.15.5 Subscribe for Notifications Sent by SIG Model 29
6.15.6 Register a Local Vendor Model (SBMLocalVendorModel) 29
6.15.7 Local Vendor Model Binding with Application Key 30
6.15.8 Manage Notifications from the SBMVendorModel 31
6.15.9 Send Value to SBMVendorModel 32
7 Bluetooth mesh API Reference for Android 35
7.1 Initializing the BluetoothMesh 35
7.1.1 BluetoothMesh 35
7.1.2 Set Up Supported Vendor Models 35
7.1.3 Set Up Mesh Limits 35
7.2 Set Up Bluetooth Layer (ConnectableDevice) 36
7.2.1 Device Advertisement Data 37

7.2.2 Device UUID 37
7.2.3 Device Name 37
7.2.4 Device Connection State 37
7.2.5 Connect to the Device 38
7.2.6 Disconnect from the Device 38
7.2.7 Check if a Device Contains a Service 38
7.2.8 Maximum Transmission Unit for Given Device Service 38
7.2.9 Write Data to a Given Service and Characteristic 39
7.2.10 Subscribe to a Given Service and Characteristic 39
7.3 Provision a Device to a Subnet 40
7.3.1 Create Network 40
7.3.2 Create Subnet 40
7.3.3 Find Non-Provisioned Bluetooth Devices 40
7.3.4 Provision the Devices 41
7.4 Add a Node to a Subnet 41
7.5 Remove a Node from a Subnet 42
7.6 Connect with a Subnet 42
7.6.1 Find All Proxies in the Device Range 42
7.6.2 Get Node Representing a Given Device 43
7.7 Create a Group in a Given Subnet 43
7.8 Remove Group 43
7.9 Add a Node to a Group 43
7.10 Remove a Node from a Group 44
7.11 Bind a Model with A Group 44
7.12 Unbind a Model from a Group 44
7.13 Add Subscription Settings to a Model 45
7.14 Add Publication Settings to a Model 45
7.15 Control Node Functionality 46
7.15.1 Get Value for a Single Model from the Node 46
7.15.2 Get Value for All Specific Models Bound with a Group 46
7.15.3 Set Value for a Single Model from the Node 46
7.15.4 Set Value for All Specific Models Bound with the Group 47

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	1.4.1	
Simultaneous GATT connections	1	
Mesh bearers	GATT	
Supported node types	Proxy Relay Low Power	Proxy Relay Low Power
Maximum number of network keys (Subnets)	255	255
Maximum number of application keys (Groups)	255	255
Maximum number of mesh nodes	255	

Feature	Android Value	iOS Value
Supported mesh models	Generic OnOff Server 0x1000	Generic OnOff Server 0x1000
	Generic OnOff Client 0x1001	Generic OnOff Client 0x1001
	Generic Level Server 0x1002	Generic Level Server 0x1002
	Generic Level Client 0x1003	Generic Level Client 0x1003
	Generic Default Transition Time Server 0x1004	Generic Default Transition Time Server 0x1004
	Generic Default Transition Time Client 0x1005	Generic Default Transition Time Client 0x1005
	Generic Power OnOff Server 0x1006	Generic Power OnOff Server 0x1006
	Generic Power OnOff Client 0x1008	Generic Power OnOff Client 0x1008
		Generic Power Level Server 0x1009
		Generic Power Level Client 0x100B
	Generic Battery Server 0x100C	Generic Battery Server 0x100C
	Generic Battery Client 0x100D	Generic Battery Client 0x100D
		Generic Location Server 0x100E
		Generic Location Client 0x1010
		Generic Property Client 0x1015
	Light Lightness Server 0x1300	Light Lightness Server 0x1300
	Light Lightness Client 0x1302	Light Lightness Client 0x1302
	Light CTL Server 0x1303	Light CTL Server 0x1303
	Light CTL Client 0x1305	Light CTL Client 0x1305
	Light CTL Temperature Server 0x1306	Light CTL Temperature Server 0x1306
Supported GATT services	Provisioning, Proxy	Provisioning, Proxy

Table 1-2. Open Source Licenses Used

Feature	License	Comment
OpenSSL	See here	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.

1. Basic understanding of Bluetooth LE and Bluetooth mesh 1.0 specifications and technology.
    1. Silicon Labs Bluetooth mesh learning center
    2. Silicon Labs Bluetooth mesh technology white paper
    3. Bluetooth mesh profile 1.0 specification
    4. Bluetooth mesh model 1.0 specification (application layer)
    5. Bluetooth 5.0 core specification
2. Simplicity Studio and Bluetooth mesh SDK 1.4.1 or newer are installed.
    1. Download
    2. Getting started
3. You have a Blue Gecko SoC (SLWSTK6020B) or module WSTKs to act as Bluetooth mesh nodes.
    3. Recommended SoCs and Modules are: EFR32BG13 or EFR32BG12 SoCs or BGM13P or BGM13S module because they have 512kB to 1024kB of flash available.
    4. SLWSTK6020B User Guide

2.1 iOS

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

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/ios/BluetoothMesh.framework
       ```

    2. Bluetooth Mesh iOS API Documentation:

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/ios/docs/index.html
       ```

    3. Source code of the iOS reference application:

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/ios_application
       ```

       The application is available in iOS AppStore: Bluetooth mesh by Silicon Labs
2. Xcode 10 or newer is installed.
3. This ADK supports iOS versions 11 and 12.
4. You understand Bluetooth LE operation on iOS.
    1. Bluetooth LE documentation for iOS
    2. Bluetooth LE API documentation

2.2 Android

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

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/android/BluetoothMesh
       ```

    2. Bluetooth Mesh Android API Documentation:

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/android/docs/index.html
       ```

    3. Source code of the Android reference application:

       ```
       /Applications/Simplicity\ Studio.app/Contents/Eclipse/
       developer/sdks/blemesh/v1.4/app/bluetooth/android_application
       ```

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

2. Android Studio is installed.
    1. Download
    2. Instruction
3. You have an Android phone or tablet preferably running Android 6.0 or newer.
4. You understand Bluetooth LE operation on Android as well as the JNI interface.
    1. Bluetooth LE documentation for Android
    2. Bluetooth LE API documentation
    3. Android JNI documentation

3 Contents of Bluetooth mesh for iOS and Android ADK

3.1 The Bluetooth Mesh Stack library

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

3.1.1 iOS

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

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.

3.2 Bluetooth mesh Network and Device Database

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

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

3.3 Reference Application Source Code

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

3.4 Documentation

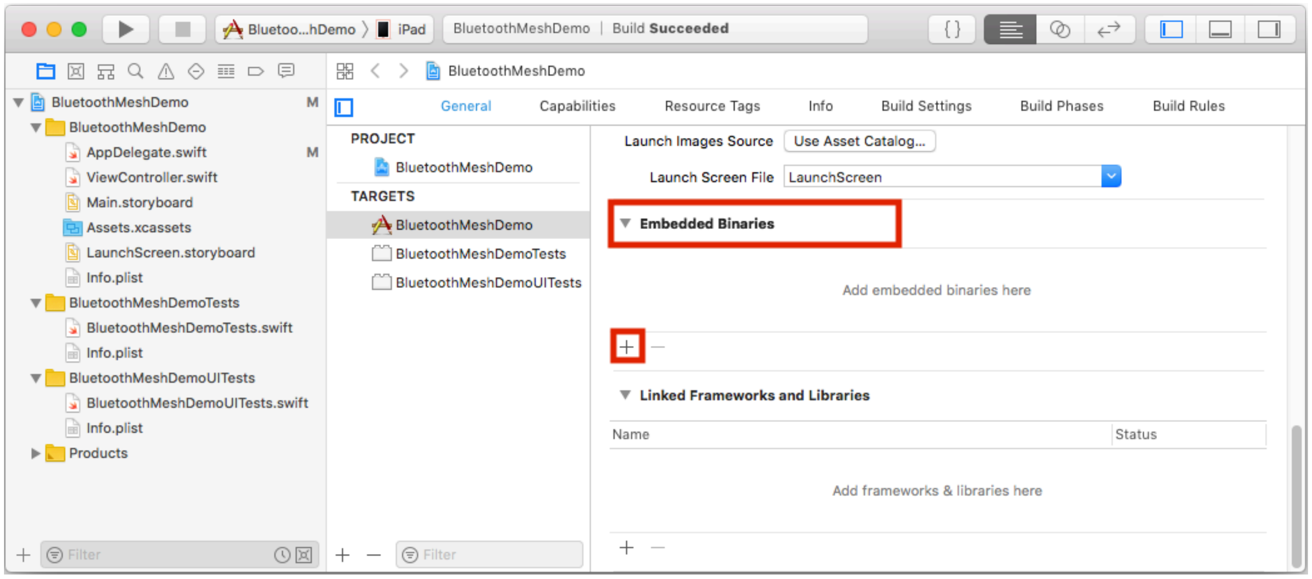
The delivery also contains API documentation for the Bluetooth mesh stack.

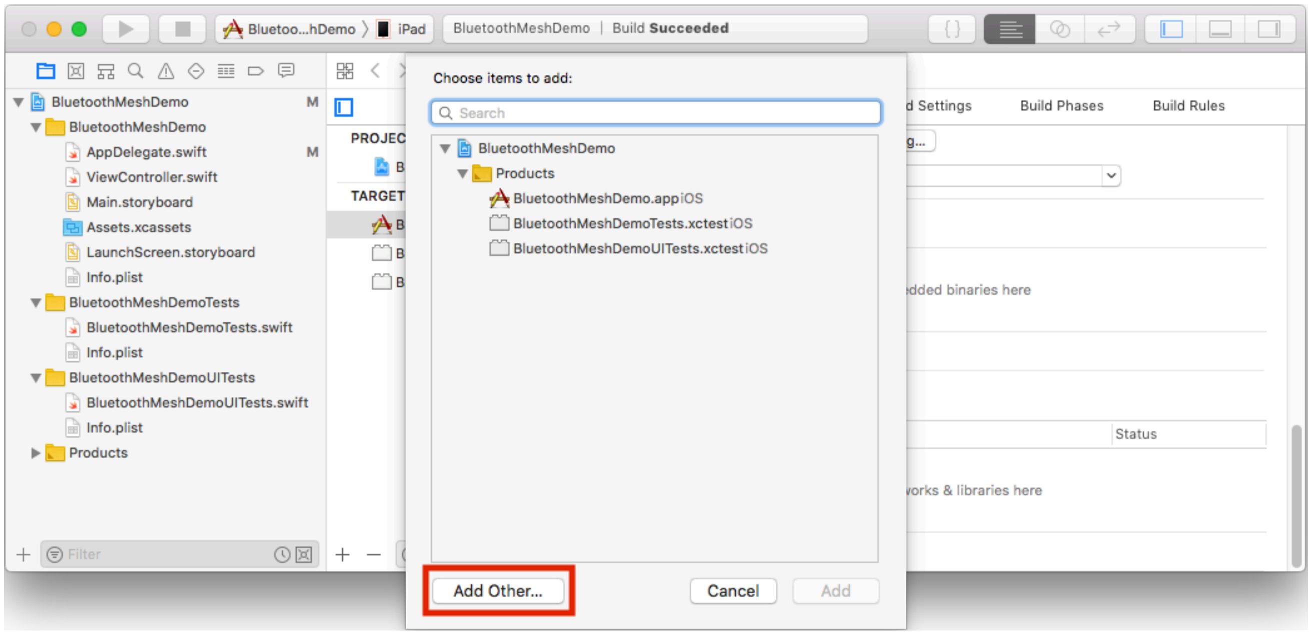
4 Getting Started with Development

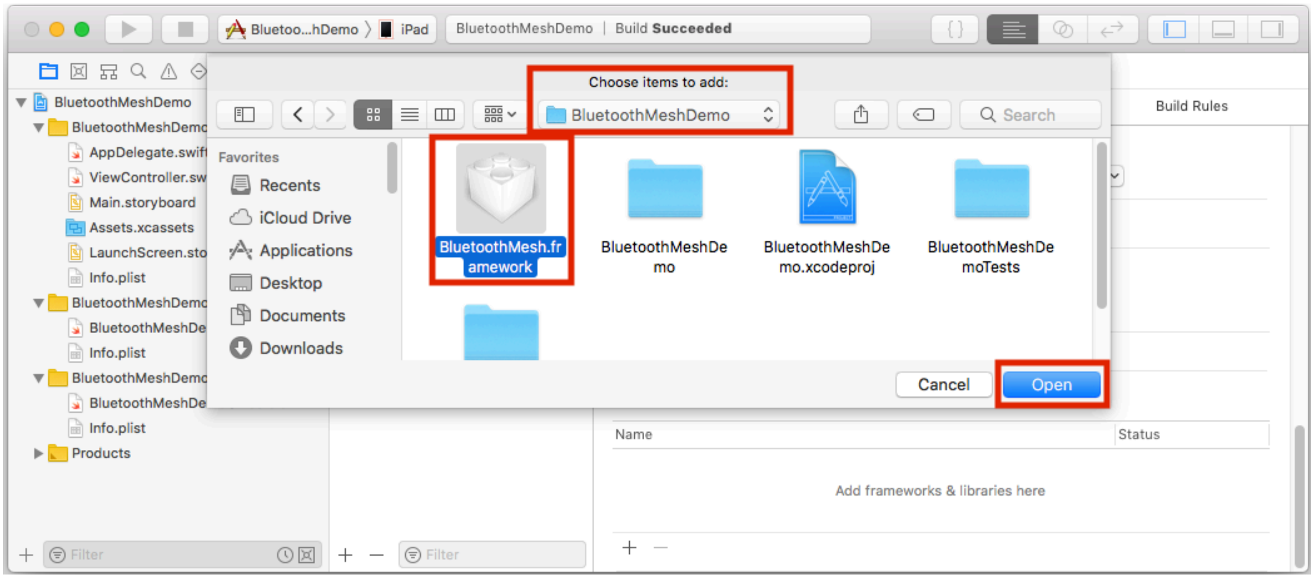
The first step is to set up a project.

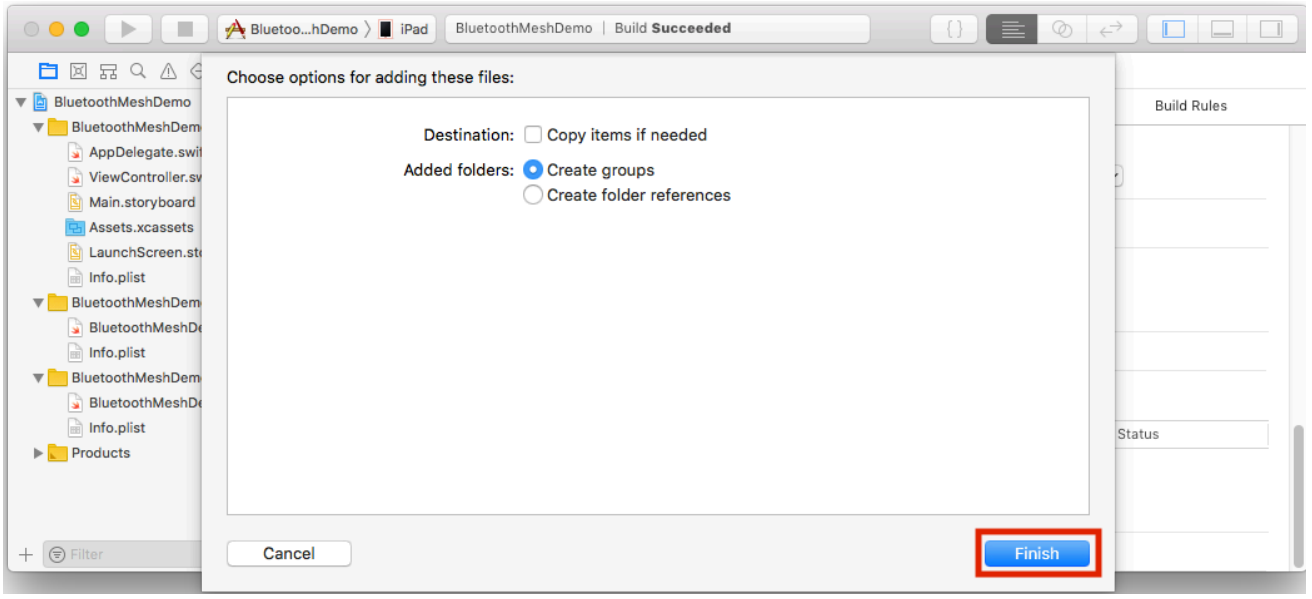
4.1 iOS

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




Choose items to add:
Search
BluetoothMeshDemo
Products
BluetoothMeshDemo.app iOS
BluetoothMeshDemoTests.xctest iOS
BluetoothMeshDemoUITests.xctest iOS
Add Other...
Cancel
Add

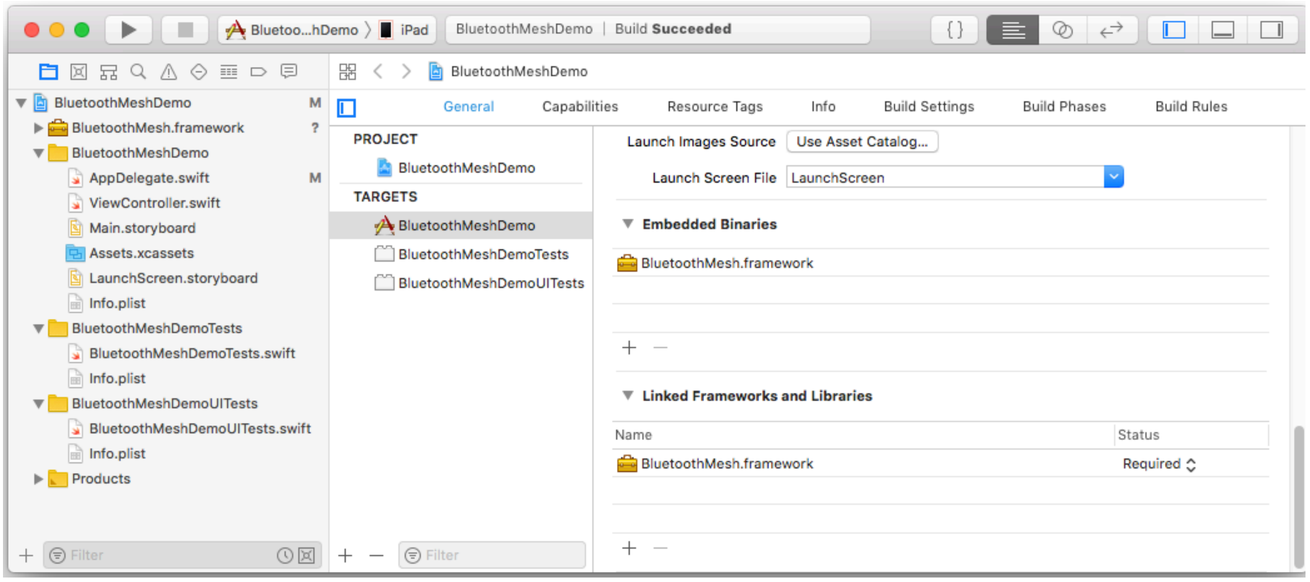



Choose options for adding these files:

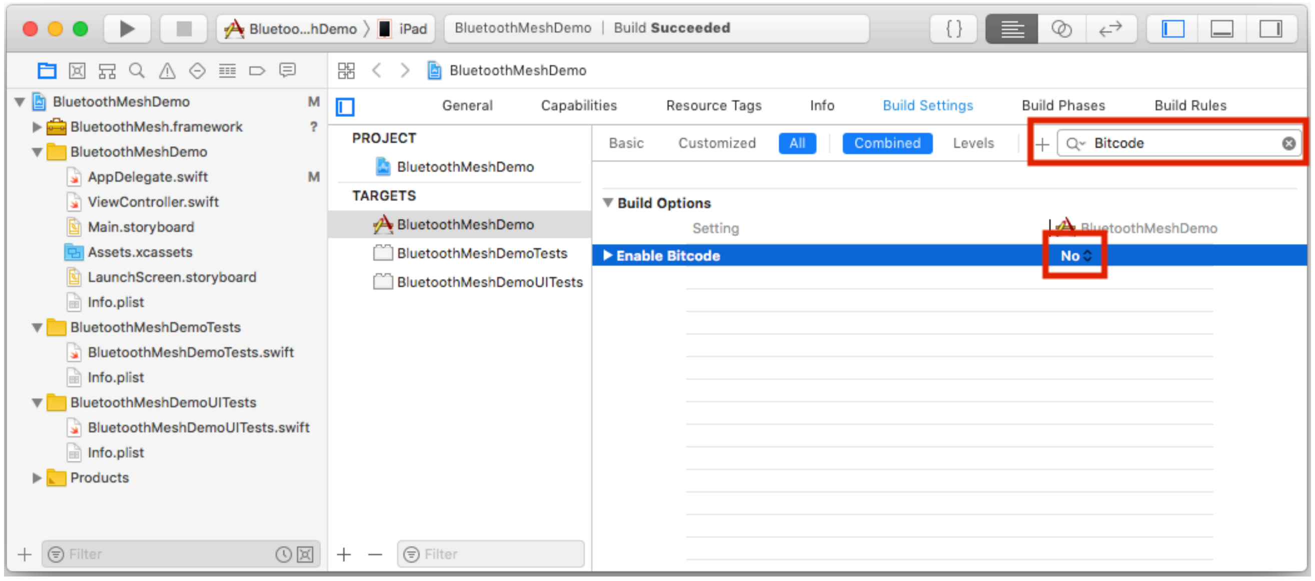
Destination: ☐ Copy items if needed

Added folders: ● Create groups
○ Create folder references

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



4. Disable Bitcode in project. BluetoothMesh.framework does not use Bitcode.
    1. Select main target in project.
    2. Go to Build Settings view.
    3. Search for Bitcode.
    4. Set Enable Bitcode to 'No'.



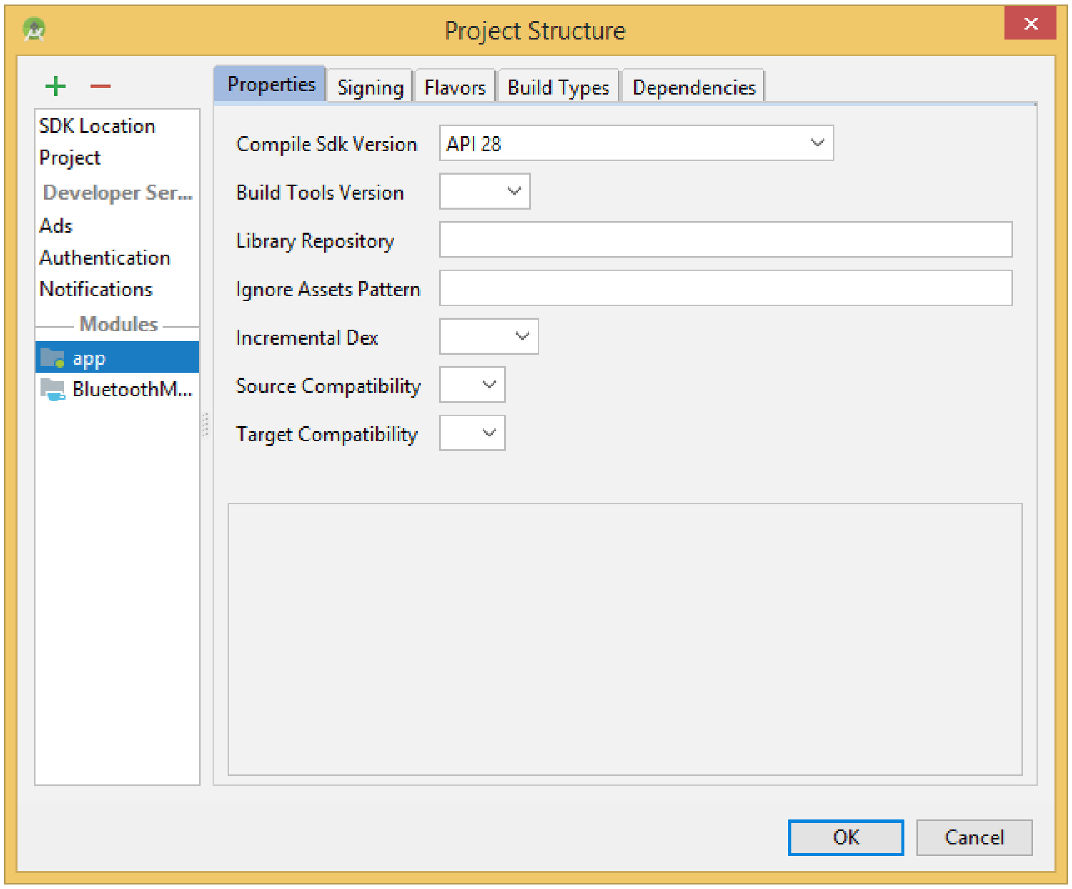
Enable Bitcode: No

4.2 Android

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

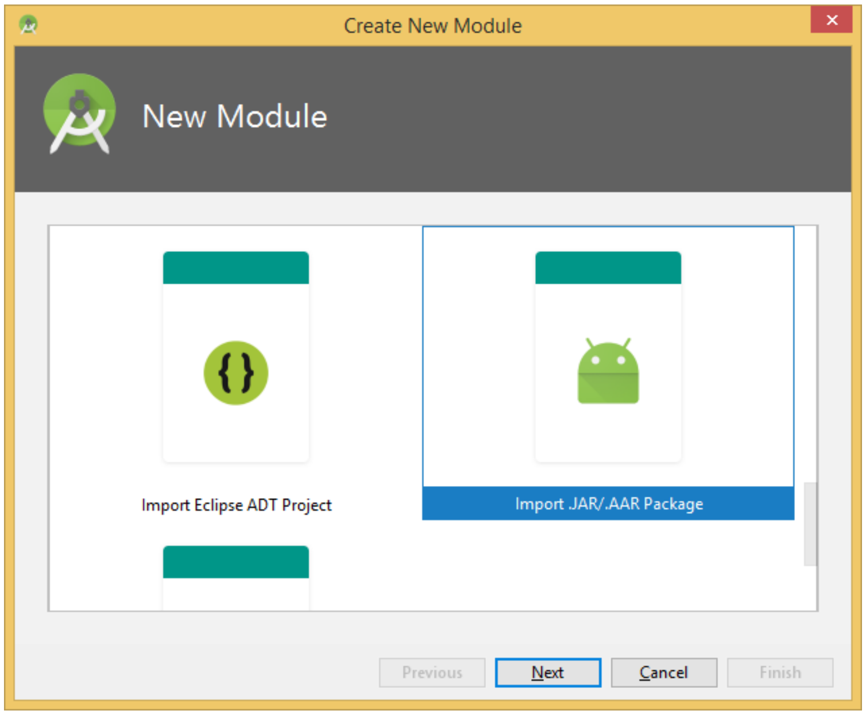
3. Create a project such as Empty Activity project.

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

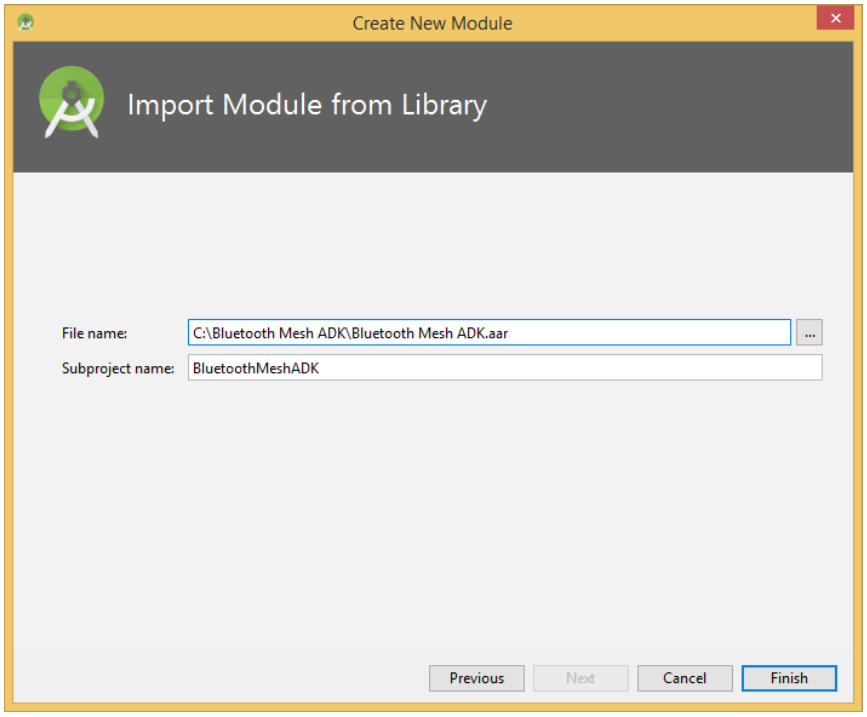


Compile Sdk Version: API 28
Build Tools Version:
Library Repository:
Ignore Assets Pattern:
Incremental Dex:
Source Compatibility:
Target Compatibility:

6. Select "Import .JAR or .AAR Package".

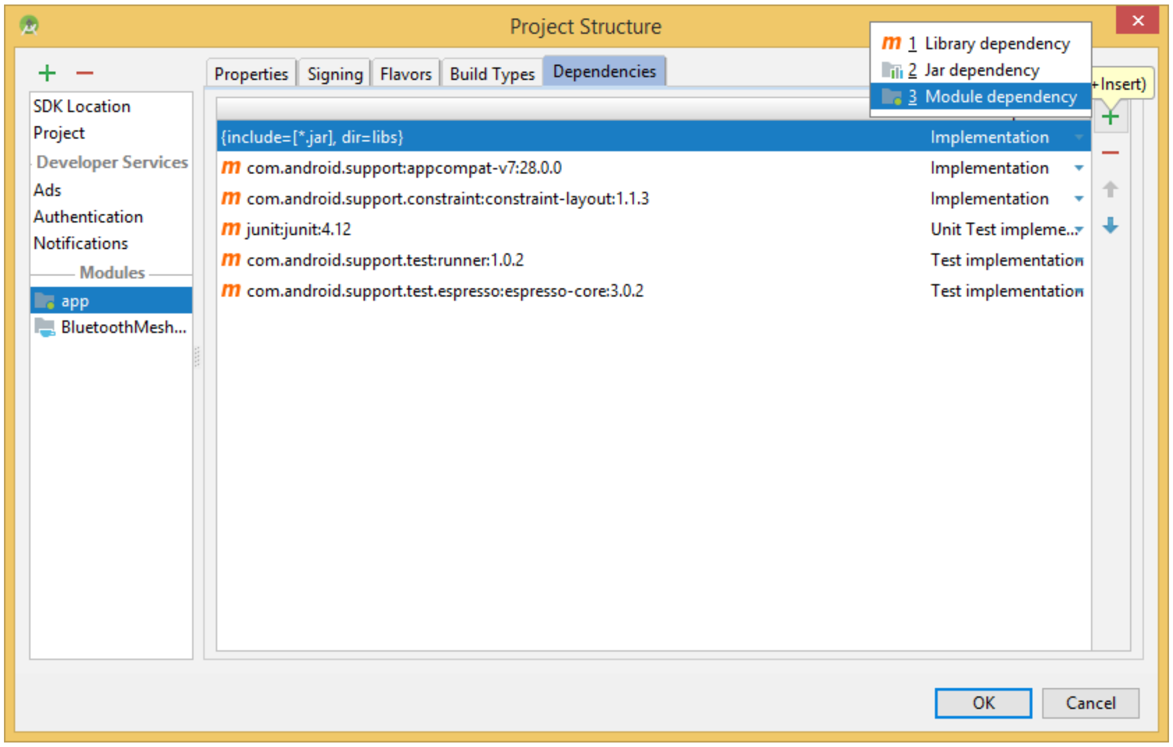


7. Specify path to .aar file and proceed.



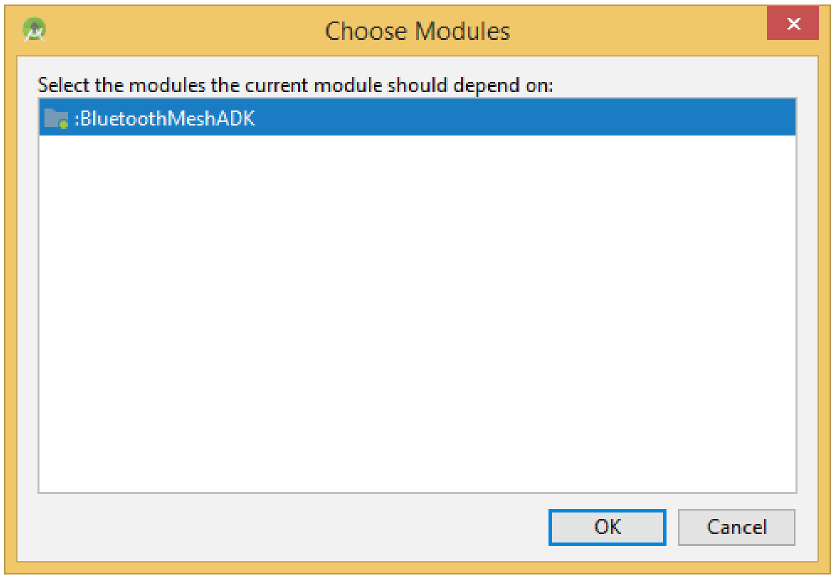
File name: C:\Bluetooth Mesh ADK\Bluetooth Mesh ADK.aar
Subproject name: BluetoothMeshADK

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



{include=[*.jar], dir=libs}	Implementation
com.android.support:appcompat-v7:28.0.0	Implementation
com.android.support.constraint:constraint-layout:1.1.3	Implementation
junit:junit:4.12	Unit Test impleme...
com.android.support.test:runner:1.0.2	Test implementation
com.android.support.test.espresso:espresso-core:3.0.2	Test implementation

9. Select module and proceed.


Choose Modules

Select the modules the current module should depend on:

:BluetoothMeshADK

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

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

public class MainActivity extends AppCompatActivity {

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

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

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

12. Compile the project and make sure it compiles without errors.

5 Bluetooth Mesh Structure Overview

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

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

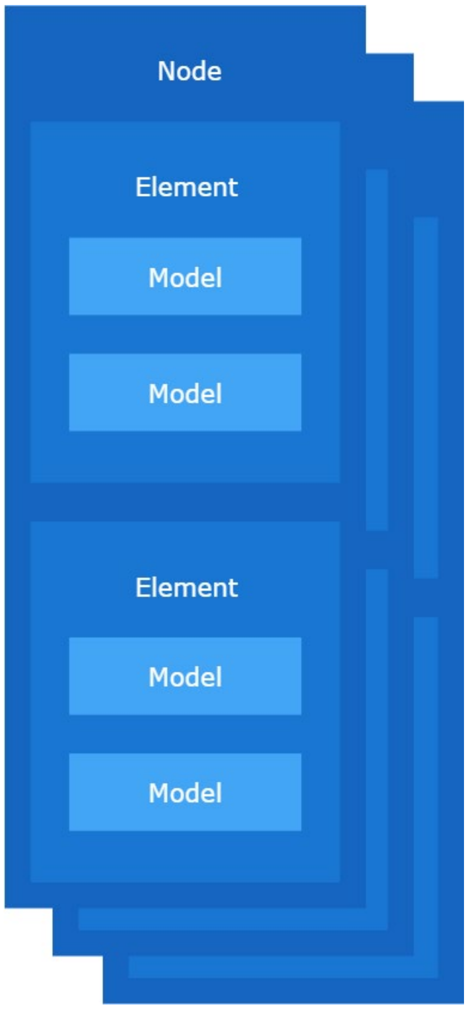


Figure 5-1. Node Structure

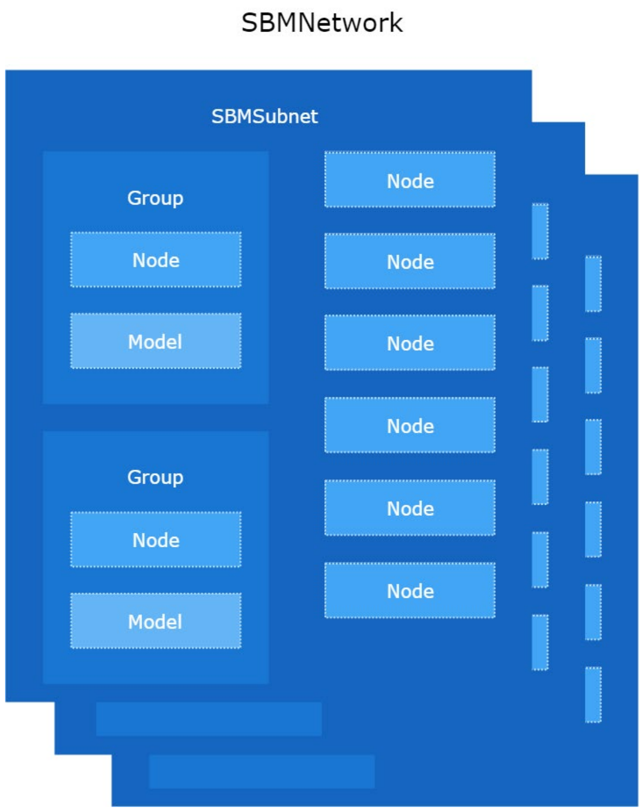


Figure 5-2. Network Structure

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 API Reference for iOS

To control base of Bluetooth mesh structure you must be familiar with a few of the most important layers of the Bluetooth mesh API:

1. Bluetooth connection
2. Provision session
3. Proxy connection
4. Node configuration
5. Model configuration (subscription and publication settings)
6. Control model and group

All are described in the next part of this document.

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

6.1 Initializing the BluetoothMesh

6.1.1 BluetoothMesh

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

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

In the base configuration supported vendor models are not needed.

To get the instance call the following for each platform noted

```
SBMBluetoothMesh.sharedInstance()

let configuration = SBMBluetoothMeshConfiguration(localVendorModels: [], andLogger: SBMLog-
ger.sharedInstance())
```

6.1.2 Set Up Supported Vendor Models

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

```
let vendorModel = SBMLocalVendorModel.init(vendorCompanyIdentifier: A, vendorAssignedModelIdenti-
fier: B)

//A – vendor company identifier
//B – vendor assigned model identifier

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

6.1.3 Set Up Mesh Limits

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

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

```
let limits = SBMBluetoothMeshConfigurationLimits()

limits.networks // maximum number of networks that can be created - MAX is 255
limits.groups // maximum number of groups that can be created - MAX is 255 but in one network the
max is 8 groups
limits.nodes // maximum number of nodes that can be provisioned - MAX is 255
limits.nodeNetworks // maximum number of networks that a single node can be added to - MAX is 7
limits.nodeGroups // maximum number of groups that a single node can be added to - MAX is 8
limits.rplSize // maximum number of nodes that can be communicated with - MAX is 255
limits.segmentedMessagesReceived // maximum number of concurrent segmented messages being received
      - MAX is 255
limits.segmentedMessagesSent // maximum number of concurrent segmented messages being sent - MAX is
       255
limits.provisionSession // maximum number of parallel provisioning sessions - MAX is 1

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

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

```
networks = 4;
groups = 10;
nodes = 255;
nodeNetworks = 4;
nodeGroups = 4;
rplSize = 32;
segmentedMessagesReceived = 4;
segmentedMessagesSent = 4;
provisionSessions = 1;
```

6.2 Set Up Bluetooth Layer (ConnectableDevice)

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

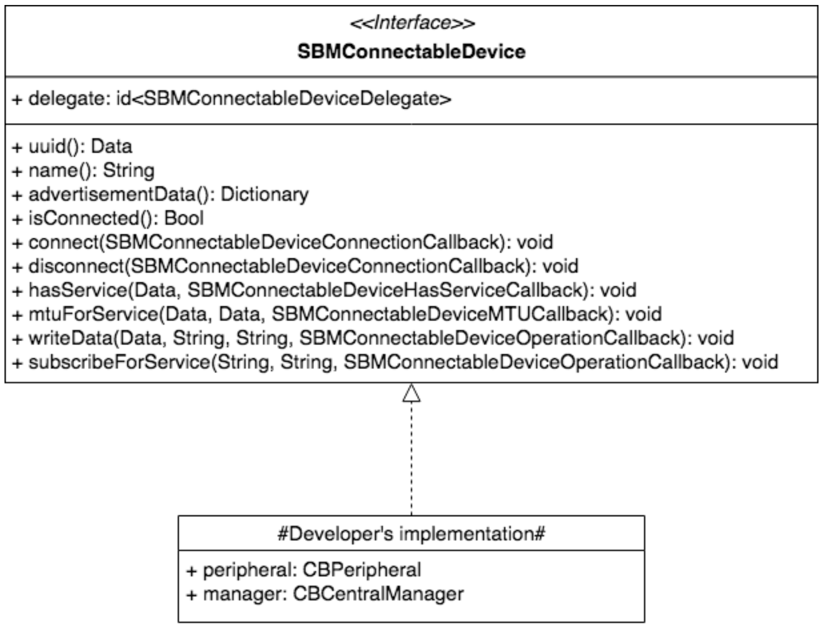

<<Interface>>
SBMConnectableDevice
+ delegate: id<SBMConnectableDeviceDelegate>
+ uuid(): Data
+ name(): String
+ advertisementData(): Dictionary
+ isConnected(): Bool
+ connect(SBMConnectableDeviceConnectionCallback): void
+ disconnect(SBMConnectableDeviceConnectionCallback): void
+ hasService(Data, SBMConnectableDeviceHasServiceCallback): void
+ mtuForService(Data, Data, SBMConnectableDeviceMTUCallback): void
+ writeData(Data, String, String, SBMConnectableDeviceOperationCallback): void
+ subscribeForService(String, String, SBMConnectableDeviceOperationCallback): void
#Developer's implementation#
+ peripheral: CBPeripheral
+ manager: CBCentralManager

Figure 6-1. SBMConnectableDevice

6.2.1 Device Advertisement Data

Advertisement data can be obtained from `CBCentralManagerDelegate`.

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

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

6.2.2 Device UUID

The device UUID comes from advertisement data.

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

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

6.2.3 Device Name

The device name comes from `CBPeripheral`.

```
let peripheral: CBPeripheral

// (…)

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

6.2.4 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
}
```

6.2.5 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)
}
```

6.2.6 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)
}
```

6.2.7 Check if a Device Contains a Service

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

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

6.2.8 Maximum Transmission Unit for Given Device Service

```
let peripheral: CBPeripheral

// (…)

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

6.2.9 Write Data to a Given Service and Characteristic

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

```
let peripheral: CBPeripheral

// (…)

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

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

6.2.10 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 SBMCon-
nectableDeviceOperationCallback) {
       //Need to provide logic to discover given characteristic for the device
       let characteristic = peripheral.services?.first {
                                  $0.uuid.uuidString == service
                           }?.characteristics?.first {
                                  $0.uuid.uuidString ==  characteristic
                           }

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

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

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

6.3 Provision a Device to a Subnet

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

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

6.3.1 Create Network

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

6.3.2 Create Subnet

```
let subnet = network.createSubnet(withName: "Subnet name")
```

6.3.3 Find Non-Provisioned Bluetooth Devices

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

```
let centralManager: CBCentralManager

// (…)

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

startDiscovering(services: [CBUUID(string: SBMProxyConnection.meshProvisioningServiceUUID())])

// CBCentralManagerDelegate
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral, advertisement-
Data: [String : Any], rssi RSSI: NSNumber) {
        // BluetoothDevice class implementation of SBMConnectableDevice interface.
       let connectableDevice = BluetoothDevice(withManager: self,
                                              peripheral: peripheral,
                                              advertisement: advertisementData)
        // provide logic to store all Mesh capable devices
}
```

6.3.4 Provision the Devices

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

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

```
let connectableDevice: SBMConnectableDevice
let subnet: SBMSubnet

// (…)

let provisionerConnection = SBMProvisionerConnection(for: connectableDevice, subnet: subnet)

// (…)

provisionerConnection.provision(asProxy: { (connection, node, error) in
       // handle result of provisioning
})
```

6.4 Add a Node to a Subnet

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

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

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

// (…)

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

6.5 Remove a Node from a Subnet

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

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

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

//(…)

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

6.6 Connect with a Subnet

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

6.6.1 Find All Proxies in the Device Range

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

```
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, advertisement-
Data: [String : Any], rssi RSSI: NSNumber) {
       // BluetoothDevice class implementation of SBMConnectableDevice interface.
       let connectableDevice = BluetoothDevice(withManager: self,
                                              peripheral: peripheral,
                                              advertisement: advertisementData)
        //provide logic to store all proxy nodes
}
```

6.6.2 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)
```

6.6.3 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)
```

6.7 Create a Group in a Given Subnet

```
let subnet: SBMSubnet

// (…)

let group = subnet.createGroup(withName: "Group name")
```

6.8 Remove Group

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

```
let group: SBMGroup

// (…)

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

6.9 Add a Node to a Group

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

```
let node: SBMNode
let group: SBMGroup

//(…)

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

6.10 Remove a Node from a Group

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

```
let node: SBMNode
let group: SBMGroup

// (…)

let control = SBMNodeControl.init(node: node)

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

6.11 Bind a Model with A Group

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

```
let model: SBMModel
let group: SBMGroup

//  (…)

let binder = SBMFunctionalityBinder(group: group)

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

6.12 Unbind a Model from a Group

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

```
let model: SBMModel
let group: SBMGroup

//  (…)

let binder = SBMFunctionalityBinder(group: group)

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

6.13 Add Subscription Settings to a Model

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

6.13.1 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
})
```

6.13.2 Vendor Model

```
let vendorModel: SBMVendorModel
let group: SBMGroup

//(…)

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

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

6.14 Add Publication Settings to a Model

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

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

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

6.14.1 SIG Model

6.14.1.1 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: { subscriptionCon-
trol, settings in
       //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
       //handle error callback
})
```

6.14.1.2 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: { subscriptionCon-
trol, settings in
       //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
       //handle error callback
})
```

6.14.2 Vendor Model

6.14.2.1 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: { subscriptionCon-
trol, settings in
       //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
       //handle error callback
})
```

6.14.2.2 Publish directly to the Provisioner

Known issue. There is a problem with starting capturing notifications from the Vendor Model if it publishes messages directly to the Provisioner. Currently there is a workaround for it. The Provisioner will start capturing notifications after it sends one SET message to the Vendor Model (see section 6.15.9 Send Value to SBMVendorModel).

```
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: { subscriptionCon-
trol, settings in
       //handle success callback
}, errorCallback: { subscriptionControl, settings, error in
       //handle error callback
})
```

6.15 Control Node Functionality

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

6.15.1 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
       })
}
```

6.15.2 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: { (control, response) in
              let response = response as! SBMGenericLevel
              //handle success callback
       }, errorCallback: { (control, response, error) in
              //handle error callback
       })
}
```

6.15.3 Set Value for a Single SIG Model from the Node

```
func set(level: Int, forElement element: SBMElement, inGroup group: SBMGroup {
       let controlElement = SBMControlElement(element: element, in: group)
       let status = SBMGenericLevel(level: Int16(level))
       let parameters = SBMControlRequestParameters(transitionTime: 1,
                                                     delayTime: 1,
                                                     requestReplay: false)

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

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

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

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

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

6.15.5 Subscribe for Notifications Sent by SIG Model

To capture notifications from the SIG Model publication settings must be configured on the model side (see section 6.14 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.

```
func subscribeLightness(forElement element: SBMElement, inGroup group: SBMGroup) {
       let controlElement = SBMControlElement(element: element, in: group)

       controlElement.subscribeStatus(SBMLightningLightnessActual.self) { controlElement,
                                                                           valueGetSigModel in
            //handle notifications
        }
}
```

6.15.6 Register a Local Vendor Model (SBMLocalVendorModel)

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

6.15.6.1 Create SBMLocalVendorSettings

Represents vendor registration settings.

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 // Supported Operation codes 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: message-
Handler)
Create SBMLocalVendorRegistrator
```

6.15.6.2 Create SBMLocalVendorRegistrator

Used to set registration settings of a local vendor model.

```
let localVendorModel: SBMLocalVendorModel

//(…)

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

6.15.6.2.1 Register a Local Vendor Model

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

```
let vendorRegistrator: SBMLocalVendorRegistrator
let registerSettings: SBMLocalVendorSettings

//(…)

vendorRegistrator.register(registerSettings)
```

6.15.6.2.2 Unregister a Local Vendor Model

```
let vendorRegistrator: SBMLocalVendorRegistrator

//(…)

vendorRegistrator.unregister
```

6.15.7 Local Vendor Model Binding with Application Key

SBMLocalVendorModel must be bound with SBMApplicationKey so that the Mesh Library can decrypt incoming messages from the mesh network. First, SBMVendorModel should be bound with SBMGroup (see section 6.11 Bind a Model with A Group), because SBMGroup is the source of the SBMApplicationKey. Messages that come from the SBMVendorModel from the SBMNode are encrypted with SBMApplicationKey from this SBMGroup.

6.15.7.1 Bind Local Vendor Model with Application Key

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

```
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

//(…)

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

6.15.7.2 Unbind Local Vendor Model from Application Key

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

```
let applicationKey: SBMApplicationKey
let localVendorModel: SBMLocalVendorModel

//(…)

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

6.15.8 Manage Notifications from the SBMVendorModel

It is possible to configure SBMVendorModel to publish notifications by SBMGroup address or directly by SBMNode address. It can be done by adding publication settings to the model (see section 6.14 Add Publication Settings to a Model).

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

6.15.8.1 Sign Up for The Notifications

Method used to subscribe to the vendor model notifications.

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

6.15.8.1.1 Notifications Come by SBMGroup Address

```
let group: SBMGroup
let vendorModel: SBMVendorModel

//(…)

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

6.15.8.1.2 Notifications Come by SBMNode Address

```
let node: SBMNode
let vendorModel: SBMVendorModel

//(…)

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

6.15.8.2 Sign Out from the Notifications

Method used to unsubscribe from the vendor model notifications.

6.15.8.2.1 Sign Out from Notifications from the SBMGroup Address

```
let group: SBMGroup
let vendorModel: SBMVendorModel

//(…)

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

6.15.8.2.2 Sign Out from Nnotificationsfrom the SBMNode Address

```
let node: SBMNode
let vendorModel: SBMVendorModel

//(…)

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

6.15.9 Send Value to SBMVendorModel

6.15.9.1 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? //it is deprecated, do NOT use it
    var vendorModel: SBMVendorModel
    var data: Data
    var flags: SBMControlValueSetVendorModelFlag

    init(with vendorModel: SBMVendorModel, messageToSend: Data, flags: SBMControlValueSetVendorMod-
elFlag) {
        self.vendorModel = vendorModel
        self.data = messageToSend
        self.flags = flags
    }
}
```

6.15.9.2 Prepare Message to Send

It is very important to prepare the message with the correct structure. The first part of the message structure must contain the Opcode (Operation code) 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 example my opcode is 0. 0xC0 must be HERE, reason can be found in the Bluetooth SIG Documenta-
tion(Mesh Profile 3.7.3.1 Operation codes)

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

data.append(messageToSend)
```

6.15.9.3 Send Prepared Message to the Single SBMVendorModel on the SBMNode

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

```
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, mes-
sageToSend: 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 ,error-
Callback: controlElementSetVendorError)
```

6.15.9.4 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, mes-
sageToSend: 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 Bluetooth mesh API Reference for Android

To control base of Bluetooth mesh structure you must be familiar with a few of the most important layers of the Bluetooth mesh API:

1. Bluetooth connection
2. Provision session
3. Proxy connection
4. Node configuration
5. Model configuration (subscription and publication settings)
6. Control model and group

All are described in the next part of this document.

7.1 Initializing the BluetoothMesh

7.1.1 BluetoothMesh

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

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

In the base configuration supported vendor models are not needed.

To get the instance call the following for each platform noted

```
BluetoothMesh.getInstance();

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

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

7.1.2 Set Up Supported Vendor Models

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

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

// companyIdentifier – vendor company identifier
// assignedModelIdentifier – vendor assigned model identifier

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

7.1.3 Set Up Mesh Limits

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

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

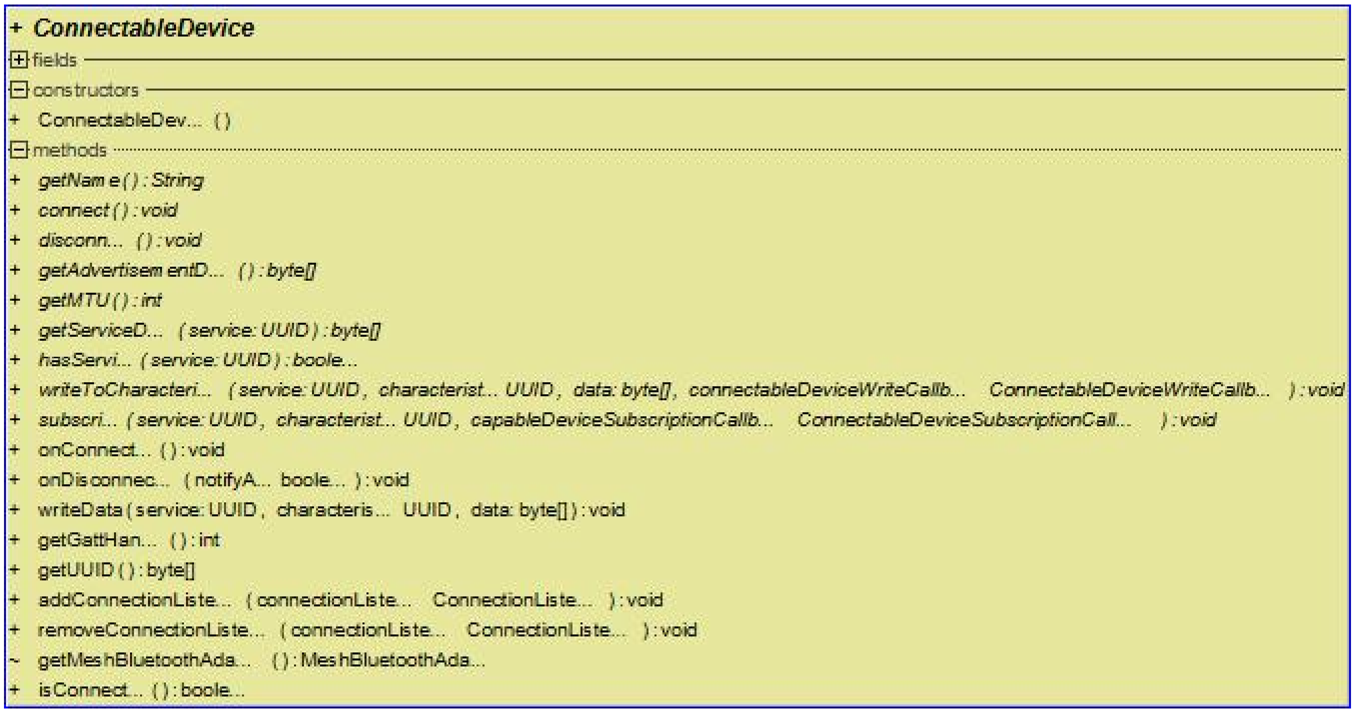
```
BluetoothMeshConfigurationLimits limits = new BluetoothMeshConfigurationLimits();

int networks; //maximum number of networks that can be created - MAX is 255
int groups; //maximum number of groups that can be created - MAX is 255 but in one network MAX is 8
int nodes; //maximum number of nodes that can be provisioned - MAX is 255
int nodeNetworks; //maximum number of networks that a single node can be added to - MAX is 7
int nodeGroups; //maximum number of groups that a single node can be added to - MAX is 8
int rplSize; //maximum number of nodes that can be communicated with - MAX is 255
int segmentedMessagesReceived; //maximum number of concurrent segmented messages being received -
      MAX is 255
int segmentedMessagesSent; //maximum number of concurrent segmented messages being sent - MAX is 255
int provisionSessions; //maximum number of parallel provisioning sessions - MAX is 1

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

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

```
networks = 16;
groups = 255;
nodes = 255;
nodeNetworks = 7;
nodeGroups = 8;
rplSize = 255;
segmentedMessagesReceived = 4;
segmentedMessagesSent = 4;
provisionSessions = 1;

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

7.2 Set Up Bluetooth Layer (ConnectableDevice)

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



```
+ ConnectableDevice
⊞ fields
⊟ constructors
+  ConnectableDev... ()
⊟ methods
+  getName() : String
+  connect() : void
+  disconn... () : void
+  getAdvertisementD... () : byte[]
+  getMTU() : int
+  getServiceD... (service: UUID) : byte[]
+  hasServi... (service: UUID) : boole...
+  writeToCharacteri... (service: UUID, characterist... UUID, data: byte[], connectableDeviceWriteCallb... ConnectableDeviceWriteCallb... ) : void
+  subscri... (service: UUID, characterist... UUID, capableDeviceSubscriptionCallb... ConnectableDeviceSubscriptionCall... ) : void
+  onConnect... () : void
+  onDisconnec... (notifyA... boole... ) : void
+  writeData(service: UUID, characteris... UUID, data: byte[]) : void
+  getGattHan... () : int
+  getUUID() : byte[]
+  addConnectionListe... (connectionListe... ConnectionListe... ) : void
+  removeConnectionListe... (connectionListe... ConnectionListe... ) : void
~  getMeshBluetoothAda... () : MeshBluetoothAda...
+  isConnect... () : boole...
```

Figure 4-4: ConnectableDevice

7.2.1 Device Advertisement Data

Advertisement data can be obtained from `ScanResult`.

```
ScanResult: scanResult

// (…)

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

// (…)

public byte[] getAdvertisementData() {
    return advertisementData;
}
```

7.2.2 Device UUID

The device UUID comes from advertisement data.

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

```
// ConnectableDevice
public byte[] getUUID() {
        byte[] uuid = new byte[16];

        byte[] data = getServiceData(MESH_UNPROVISIONED_SERVICE);
        System.arraycopy(data, 0, uuid, 0, uuid.length);

        return uuid;
}
```

7.2.3 Device Name

The device name comes from `BluetoothDevice`.

```
BluetoothDevice bluetoothDevice;

// (…)

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

7.2.4 Device Connection State

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

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

7.2.5 Connect to the Device

```
BluetoothGatt bluetoothGatt;
BluetoothGattCallback bluetoothGattCallback;

// (…)

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

7.2.6 Disconnect from the Device

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

// (…)

public void disconnect() {
    if (bluetoothGatt != null) {
        bluetoothGatt.disconnect();
    }
}
```

7.2.7 Check if a Device Contains a Service

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

```
ScanResult scanResult;

// (…)

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

    return serviceUuids.contains(new ParcelUuid(service));
}
```

7.2.8 Maximum Transmission Unit for Given Device Service

```
int mtuSize;

// (…)

private BluetoothGattCallback bluetoothGattCallback = new BluetoothGattCallback() {

// (…)

    @Override
    public void onMtuChanged(BluetoothGatt bluetoothGatt, int mtu, int status) {
        super.onMtuChanged(bluetoothGatt, mtu, status);
        Log.d(TAG, "onMtuChanged : status: " + status + ", mtu: " + mtu);
        if (status == BluetoothGatt.GATT_SUCCESS) {
            mtuSize = mtu;
            bluetoothGatt.discoverServices();
        }
    }

// (…)

}

// (…)

public int getMTU() {
    return mtuSize;
}
```

7.2.9 Write Data to a Given Service and Characteristic

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

```
BluetoothGatt bluetoothGatt;

// (…)

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

7.2.10 Subscribe to a Given Service and Characteristic

```
BluetoothGatt bluetoothGatt;

// (…)

public void subscribe(UUID service, UUID characteristic, ConnectableDeviceSubscriptionCallback
callback) {
    if (bluetoothGatt == null) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattService bluetoothGattService = bluetoothGatt.getService(service);
    if (bluetoothGattService == null) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattCharacteristic gattCharacteristic = bluetoothGattService.getCharacteristic(charac-
teristic);
```

```
    if (!bluetoothGatt.setCharacteristicNotification(gattCharacteristic, true)) {
        callback.onFail(service, characteristic);
        return;
    }
    if (gattCharacteristic.getDescriptors().size() != 1) {
        callback.onFail(service, characteristic);
        return;
    }
    BluetoothGattDescriptor gattDescriptor = gattCharacteristic.getDescriptors().get(0);
    gattDescriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
    if (!bluetoothGatt.writeDescriptor(gattDescriptor)) {
        callback.onFail(service, characteristic);
        return;
    }
    callback.onSuccess(service, characteristic);
}
```

7.3 Provision a Device to a Subnet

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

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

7.3.1 Create Network

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

7.3.2 Create Subnet

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

7.3.3 Find Non-Provisioned Bluetooth Devices

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

```
BluetoothMesh bluetoothMesh;
Context context;

// (…)

ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        if (result == null || result.getScanRecord() == null ||
                result.getScanRecord().getServiceUuids() == null ||
                result.getScanRecord().getServiceUuids().isEmpty()) {
            return;
        }
        // BTConnectableDevice extends ConnectableDevice abstract class.
        BTConnectableDevice device = new BTConnectableDevice(context, result);
    }
};

ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
        .build();

UUID uuid = ProvisionerConnection.MESH_UNPROVISIONED_SERVICE;
```

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

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

7.3.4 Provision the Devices

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

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

```
ConnectableDevice connectableDevice;
Subnet subnet;

// (…)

ProvisionerConnection provisionerConnection = new ProvisionerConnection(connectableDevice, subnet);


// (…)

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

    }

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

    }
});
```

7.4 Add a Node to a Subnet

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

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

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

// (…)

control.addTo(subnet, new NodeControlCallback() {
    @Override
    public void succeed() {

    }

    @Override
    public void error(ErrorType errorType) {

    }
});
```

7.5 Remove a Node from a Subnet

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

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

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

// (…)

control.addTo(subnet, new NodeControlCallback() {
    @Override
    public void succeed() {

    }

    @Override
    public void error(ErrorType errorType) {

    }
});
```

7.6 Connect with a Subnet

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

7.6.1 Find All Proxies in the Device Range

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

```
BluetoothMesh bluetoothMesh;
Context context;

// (…)

ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        super.onScanResult(callbackType, result);
        if (result == null || result.getScanRecord() == null ||
                result.getScanRecord().getServiceUuids() == null ||
                result.getScanRecord().getServiceUuids().isEmpty()) {
            return;
        }
        // BTConnectableDevice extends ConnectableDevice abstract class.
        BTConnectableDevice device = new BTConnectableDevice(context, result);
    }
};

ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
        .build();

UUID uuid = ProxyConnection.MESH_PROXY_SERVICE;

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

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

7.6.2 Get Node Representing a Given Device

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

```
ConnectableDevice connectableDevice:
ConnectableDeviceHelper connectableDeviceHelper;

//(…)

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

7.7 Create a Group in a Given Subnet

```
Subnet subnet;

// (…)

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

7.8 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, ErrorType errorType) {

        }
    });
```

7.9 Add a Node to a Group

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

```
Node node;
Group group;

//(…)

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

    }

    @Override
    public void error(ErrorType errorType) {

    }
});
```

7.10 Remove a Node from a Group

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

```
Node node;
Group group;

//(…)

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

    }

    @Override
    public void error(ErrorType errorType) {

    }
});
}
```

7.11 Bind a Model with A Group

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

```
Model model;
Group group;

// (…)

FunctionalityBinder binder = new FunctionalityBinder(group);

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

    }

    @Override
    public void error(List<Model> list, Group group, ErrorType errorType) {

    }
});
```

7.12 Unbind a Model from a Group

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

```
Model model;
Group group;

// (…)

FunctionalityBinder binder = new FunctionalityBinder(group);

binder.unbindModel(model, new FunctionalityBinderCallback() {
    @Override
    public void succeed(List<Model> list, Group group) {

    }
    @Override
    public void error(List<Model> list, Group group, ErrorType errorType) {

    }
});
       //handle error callback
})
```

7.13 Add Subscription Settings to a Model

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

```
Model model;
Group group;

//(…)

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

subscriptionControl.addSubscriptionSettings(subscriptionSettings, new SubscriptionSettingsCallback()
{
    @Override
    public void success(SigModel sigModel, SubscriptionSettings subscriptionSettings) {

    }

    @Override
    public void error(SigModel sigModel, ErrorType errorType) {

    }
});
```

7.14 Add Publication Settings to a Model

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

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

```
Model model;
Group group;

//(…)

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

subscriptionControl.setPublicationSettings(publicationSettings, new PublicationSettingsCallback() {
    @Override
    public void success(SigModel sigModel, PublicationSettings publicationSettings) {

    }

    @Override
    public void error(SigModel sigModel, ErrorType errorType) {

    }
});
```

7.15 Control Node Functionality

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

7.15.1 Get Value for a Single Model from the Node

```
GenericLevel genericLevel;

//(…)

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

    controlElement.getStatus(genericLevel, new GetElementStatusCallback<ControlValueGetSigModel>() {
        @Override
        public void success(Element element, Group group, ControlValueGetSigModel controlValueGetSig-
Model) {

        }

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

        }
    });
}
```

7.15.2 Get Value for All Specific Models Bound with a Group

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

```
GenericLevel genericLevel;

//(…)

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

    controlGroup.getStatus(genericLevel, new GetGroupStatusCallback<GenericLevel>() {
        @Override
        public void success(Group group, GenericLevel genericLevel) {

        }

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

        }
    });
}
```

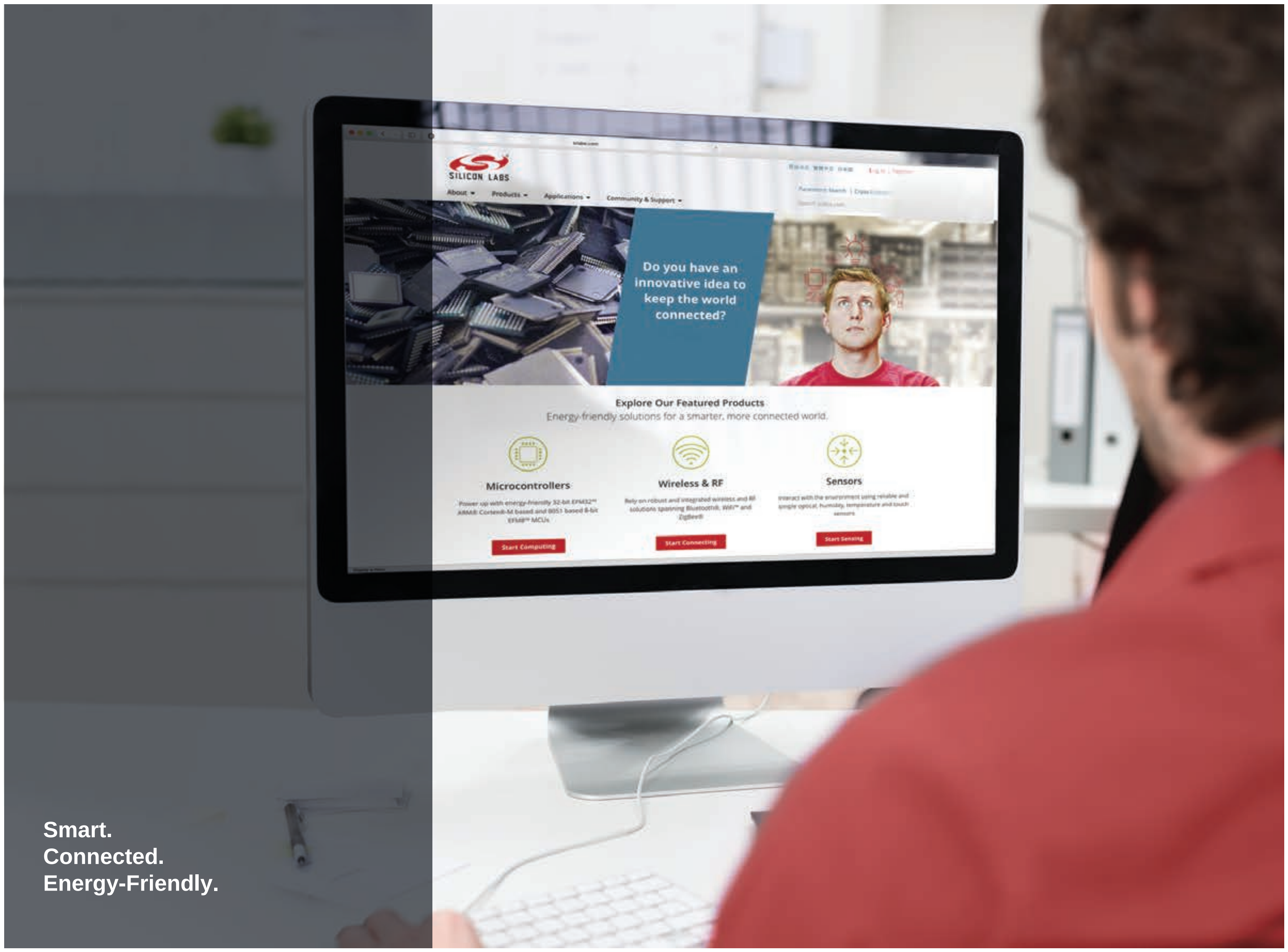
7.15.3 Set Value for a Single Model from the Node

```
void set(int level, Element element, Group group) {
    ControlElement controlElement = new ControlElement(element, group);
    GenericLevel status = new GenericLevel();
    status.setLevel(16);

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

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

            }

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

            }
        }
    );
}
```

7.15.4 Set Value for All Specific Models Bound with the Group

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

```
void set(int level, Group group) {
    ControlGroup controlGroup = new ControlGroup(group);
    GenericLevel status = new GenericLevel();
    status.setLevel(16);

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

    controlGroup.setStatus(status, parameters, new SetGroupStatusCallback<GenericLevel>() {
            @Override
            public void success(Group group, GenericLevel genericLevel) {

            }

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

            }
        }
    );
}
```


Smart.
Connected.
Energy-Friendly.
Do you have an innovative idea to keep the world connected?
Explore Our Featured Products
Energy-friendly solutions for a smarter, more connected world.
Microcontrollers
Wireless & RF
Sensors
Start Computing
Start Connecting
Start Sensing

Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com