# Bluetooth Mesh iOS API 1.0.3

by Silicon Labs

# 1. Table of Contents

# 2. Intro

This is the Bluetooth Mesh iOS API by Silicon Labs.

Besides its name, the API is not dependent to the Bluetooth/BLE iOS API. The user application must be able to deal with all the BLE connections, pass messages from the node to the library and from the library to the node.

The user of this API is able to write commands to the SBMBluetoothMesh mesh object. If the method returns only simple information from the library and doesn't depend on the Bluetooth Mesh network communication, it returns the requested information. If the method depends on the information coming from the Bluetooth Mesh network, the user application will receive a callback in the end of each operation

## a. Setting up iOS Project

Add BluetoothMesh.framework to project. Framework can be used without any additional effort with Swift or Objc applications.

## b. BluetoothMesh

It is the main object of the Bluetooth Mesh iOS API. Exchange of messages between user application and Bluetooth Mesh Framework happens here (e.g. Provisioning, Configuration, Controlling, Deletion, etc).

The API contains objects, that help developers to manage Bluetooth Mesh Network.

These are:

| | |
|---|---|
| SBMBluetoothMesh | |
| SBMBluetoothAdapter | |
| SBMMeshNode | (immutable) |
| SBMMeshGroup | (immutable) |
| SBMMeshNetwork | (immutable) |
| SBMApplicationKey | (immutable) |
| SBMDeviceCompositionData | (immutable) |
| SBMDeviceFunctionality | (immutable) |
| SBMElement | (immutable) |
| SBMModel | (immutable) |
| SBMMeshNodeConfiguration | (immutable) |
| SBMModelConfiguration | (immutable) |
| SBMLogger | |

If object is immutable it is needed to request a change for them by appropriate method in the SBMBluetoothMesh.
Integration process starts with initialization of the BluetoothMesh object.

### - Initialising the BluetoothMesh object

```
var bleMeshManager = SBMBluetoothMesh()
bleMeshManager.initialise()
```

Initialization of BluetoothMesh has to be done before any other mesh stack calls.

### - Callbacks

In the BluetoothMesh framework a lot of communication is done asynchronously. In the SBMBluetoothMesh object responses are putted in the method's completion blocks.

In the SBMBluetoothAdapter you need to implement SBMBluetoothAdapterDelegate to receive responses from the Bluetooth Mesh connection.

```
public func bluetoothAdapter(_ bluetoothAdapter: SBMBluetoothAdapter,
gattRequestToGattHandle gattHandle: Int, _ gattServiceType:
SBMGattServiceType)

public func bluetoothAdapter(_ bluetoothAdapter: SBMBluetoothAdapter,
disconnectionRequestToGattHandle gattHandle: Int)

public func bluetoothAdapter(_ bluetoothAdapter: SBMBluetoothAdapter,
gattWriteToGattHandle gattHandle: Int, data: Data, serviceUUID: String,
characteristic characteristicUUID: String)
```

## c. Support Objects

The support objects as mentioned above are objects that able the user to manage Bluetooth Mesh Network in a simple way.

### - SBMBluetoothMesh

It is the main object of the Bluetooth Mesh iOS API. Exchange of messages between user application and Bluetooth Mesh Framework happens here (e.g. Provisioning, Configuration, Controlling, Deletion, etc).

### - SBMBluetoothAdapter

SBMBluetoothAdapterDelegate is a delegate used by Mesh library to send messages to Bluetooth devices. It is necessary to prepare implementation for BluetoothAdapter delegate.

## - SBMMeshNode

A SBMMeshNode is the object that contains all the information about a Bluetooth Mesh device in a Network. It is created right after provision a device. After add node to group(Configure available models from DCD), it will be possible to get all configured Elements and Models from the configuration object.

## - SBMMeshGroup

A SBMMeshGroup is the object that contains an Application Key and a publish and subscribe addresses that will be used to configure the models inside a SBMMeshNode. It is abstract part of Bluetooth Mesh not included in official Bluetooth Mesh Specification.

## - SBMMeshNetwork

A SBMMeshNetwork object contains informations about the Bluetooth Mesh Network.

## - SBMApplicationKey

It is a container for application key index. Application key is used to bind model from the device with specific group. Each SBMMeshGroup has own application key index.

## - SBMDeviceCompositionData

SBMDeviceCompositionData contains information about a node, the elements it includes, and the supported models.

## - SBMDeviceFunctionality

Device functionality. This is mainly used for the setting the device's control option in the UI.

## - SBMElement

Each node contains one or more Elements. Element is a container for a Models contained by Node.

## - SBMModel

Model is an object created after receiving the device composition data (DCD) from the Node.

## - SBMMeshNodeConfiguration

Configuration of the Node.

## - SBMModelConfiguration

Model configuration.

## - SBMLogger

Configurable logger for Mesh framework.

# 3. Provisioning

## a. Setting up BLE GATT Connections

To provision device it is needed to discover CBPeripherals by iOS Core Bluetooth. It must be done on the application side.

How to do that:
https://developer.apple.com/bluetooth/

### - Scanning Bluetooth Mesh Devices

To discover **not provisioned** devices it is needed to scan for peripherals that are advertising service:
- service with meshProvisioningServiceUUID = „1827"

### - Enabling Notification to the Characteristic

After receive message gattRequestToGattHandle from the SBMBluetoothAdapterDelegate it is required to subscribe to selected characteristic. If subscription is ended with success it is needed to:

- update MTU Bluetooth parameter by:

  SBMBluetoothAdapter:
  ```
  open func updateMTU(_ mtu: Int)
  ```

- inform BluetoothMesh about successfully prepared Bluetooth connection by:

  SBMBluetoothAdapterDelegate:
  ```
  open func connectGatt(_ gattHandle: Int)
  ```

## b. Creating a Bluetooth Mesh Network

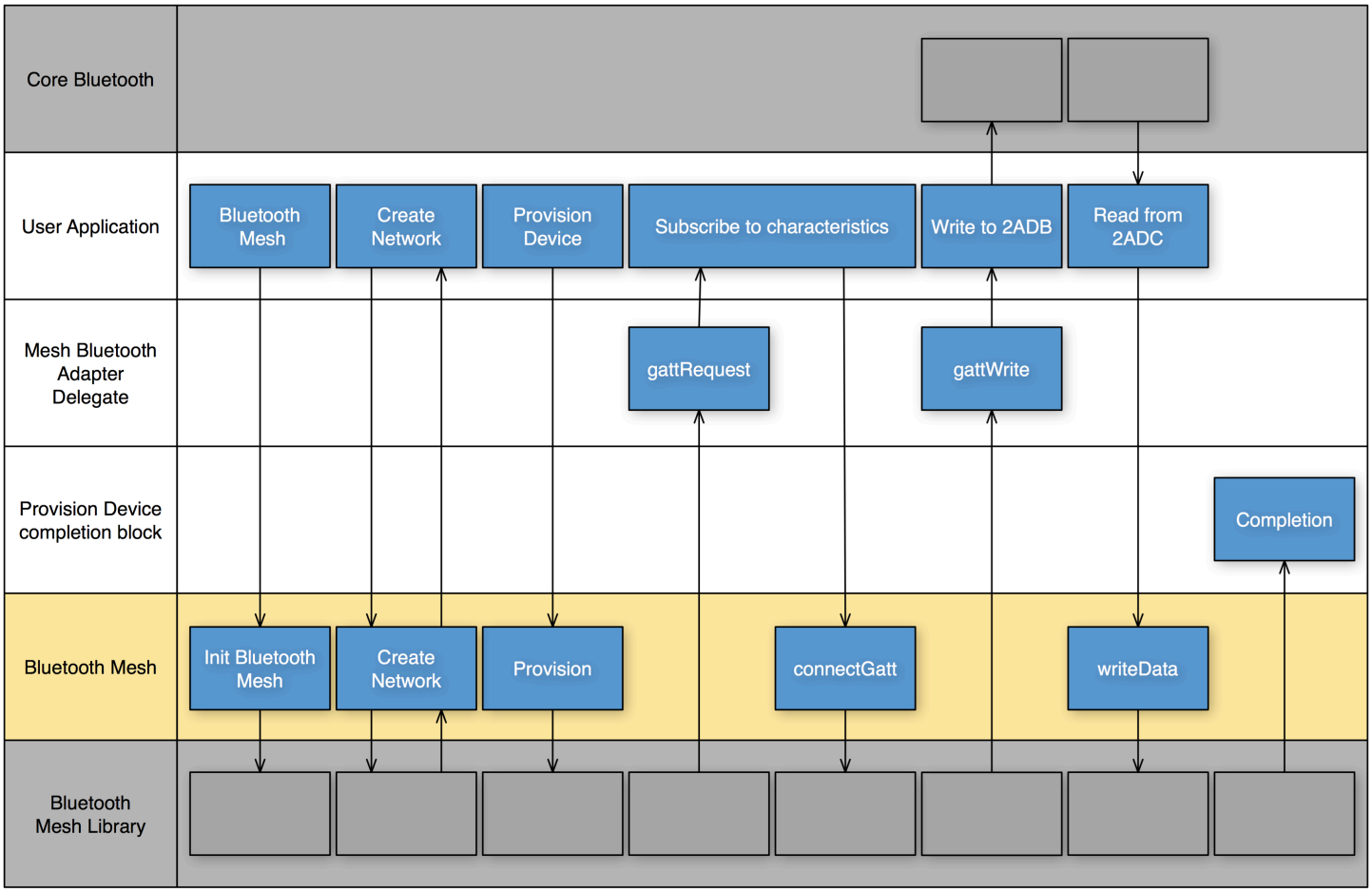
Create a mesh network. From the Bluetooth Mesh specification it is known as create network key.

SBMBluetoothMesh:
```
open func createNetwork(_ name: String) throws -> SBMMeshNetwork
```

## c. Provisioning a mesh device to a Bluetooth mesh network and add them to the Network

SBMBluetoothMesh:

```
open func provisionDevice(_ network: SBMMeshNetwork, deviceName: String,
deviceUUID: Data, gattHandle: Int, completion: @escaping (SBMMeshNode?,
Error?) -> Swift.Void)
```




After successful provisioning, the GATT connection must be dropped. After disconnection the node will start to advertise as a proxy node per 60 seconds.

When the user receives the completion callback in his application, it means that the provisioning session ended. If the error is nil, means that the provisioning session was successful. SBMMeshNode from callback is a provisioned node. This node is automatically saved in the Bluetooth Mesh and added to the network.

# 4. Configuration

## a. Setting up BLE GATT Connections

To provision device it is needed to discover CBPeripherals by iOS Core Bluetooth. It must be done on the application side.

How to do that:
https://developer.apple.com/bluetooth/

## b. Scanning Bluetooth Mesh Devices

To discover **provisioned** devices it is needed to scan for peripherals that are advertising service:

- service with meshProxyServiceUUID = „1828”

Please keep in mind, that provisioned devices with no Proxy feature activated, are not advertising over BLE.

If a device was just provisioned, it has a 60 seconds window of GATT advertising waiting to be configured. To find this device the user can use:

```
SBMBluetoothMesh:
open func deviceIdentityMatches(_ meshDeviceUUID: Data!, advertisingData
data: Data!) -> Bool
```

If the user wants to only connect with the closest proxy:

```
SBMBluetoothMesh:
open func networkHashMatches(_ networkIndex: Int, advertisingData data:
Data!) -> Bool
```

In iOS it is not possible to get raw data from Bluetooth peripheral advertisement message. To do it, we need to reconstruct a following structure:

```
let serviceData = dictionary[CBUUID(string: "1828")]
serviceDataLength = ServiceData.dataLength(for: serviceData)
serviceDataType = ServiceData.dataType
meshProxyService = ServiceData.meshProxy

data = Data()
data.append(Data(bytes: &serviceDataLength, count:
MemoryLayout.size(ofValue: serviceDataLength)))
data.append(Data(bytes: &serviceDataType, count:
MemoryLayout.size(ofValue: serviceDataType)))
data.append(Data(bytes: &meshProxyService, count:
MemoryLayout.size(ofValue: meshProxyService)))
data.append(Data(referencing: serviceData))
```

## - Enabling Notification to the Characteristic

After receive message gattRequestToGattHandle from the SBMBluetoothAdapterDelegate it is required to subscribe to a selected characteristic. If subscription is ended with success it is needed to:

- update MTU Bluetooth parameter by:

  SBMBluetoothAdapter:
  ```
  open func updateMTU(_ mtu: Int)
  ```

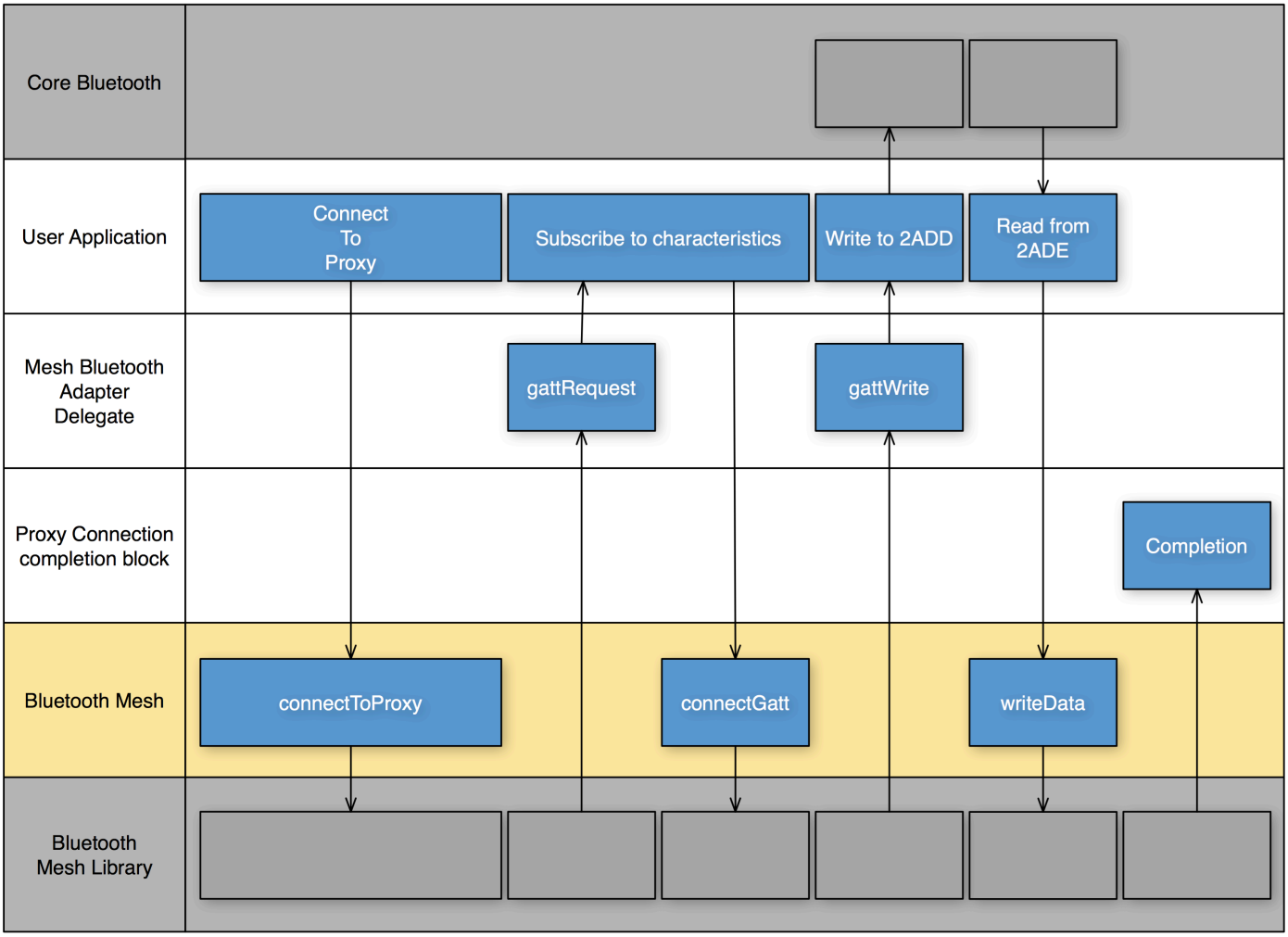
- inform BluetoothMesh about successfully prepared Bluetooth connection by:

  SBMBluetoothAdapterDelegate:
  ```
  open func connectGatt(_ gattHandle: Int)
  ```

## c. Initialising Proxy

SBMBluetoothAdapter:

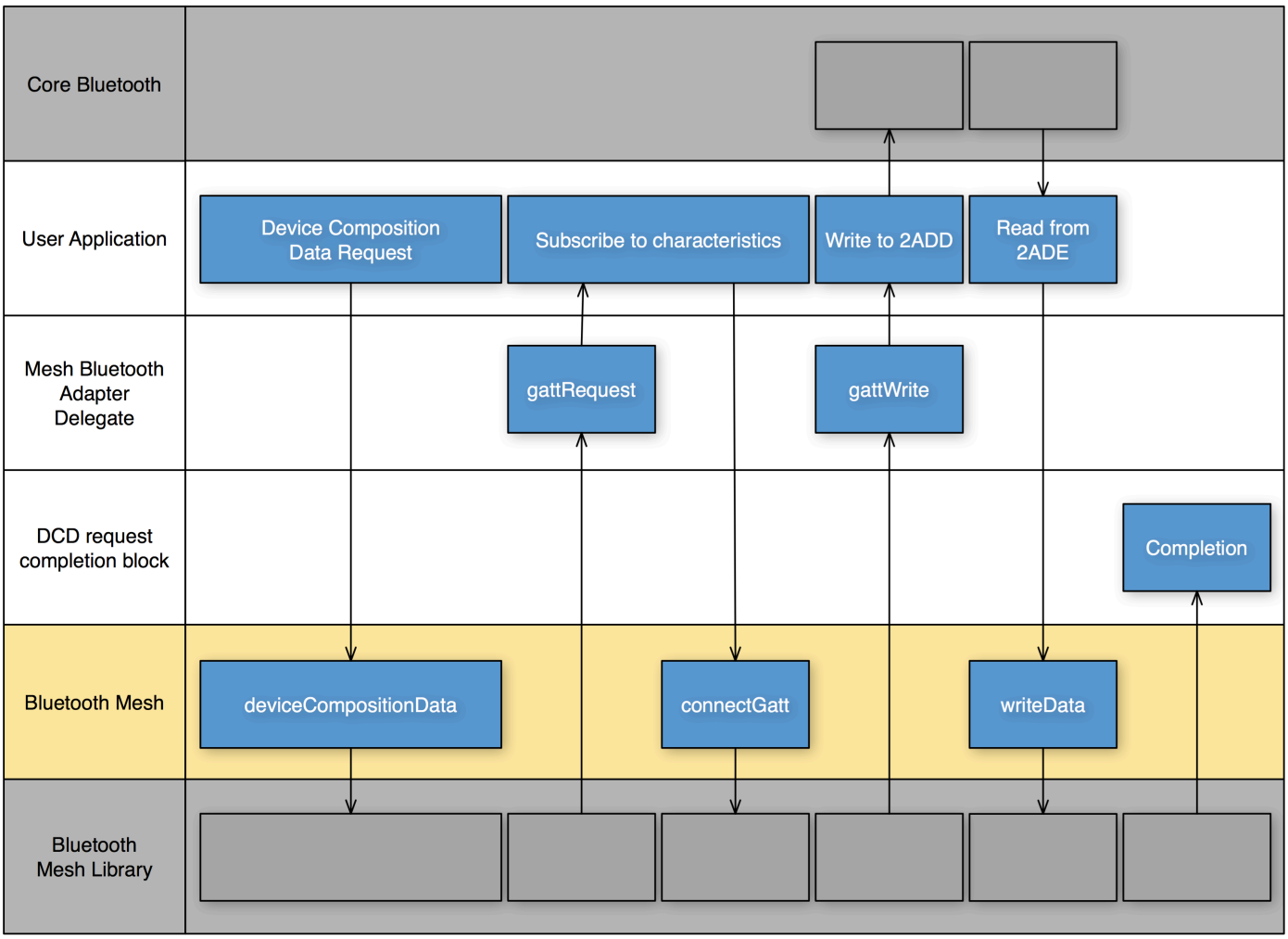
```
open func connectToProxy(withGattHandle gattHandle: Int, completion:
((Int, Bool) -> Swift.Void)!)
```



## d. Request DCD

Assuming the user application has a GATT connection with the node and proxy is initialised. The application is able to requestDCD.

```
SBMBluetoothMesh:
open func deviceCompositionData(for node: SBMMeshNode, completion:
@escaping (SBMDeviceCompositionData?, Error?) -> Swift.Void)
```




After receiving DCD, the SBMMeshNode object will contain SBMDeviceCompositionData object in the configuration. This object will contain the information of what features (proxy, relay) the device supports and also the Elements. The Elements contain the Models, that will be used to configure a device.

## e. Create a Group

To be able to configure a device, the user application must add device to group. Before that it is needed to create group:

```
SBMBluetoothMesh:
open func createGroup(withName name: String, in network: SBMMeshNetwork)
throws -> SBMMeshGroup
```

## f. Configuring a mesh node

### - Add to Network

Only way to add Node to Network is provision this Device to the specific Network.

```
SBMBluetoothMesh:
open func provisionDevice(_ network: SBMMeshNetwork, deviceName: String,
deviceUUID: Data, gattHandle: Int, completion: @escaping (SBMMeshNode?,
Error?) -> Swift.Void)
```

### - Remove from Network

To remove Node form the Network it is needed to call factory reset method from the BluetoothMesh.

### - Factory Resetting a Bluetooth Mesh Node

If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func factoryReset(_ node: SBMMeshNode, completion: @escaping (Bool)
-> Swift.Void)
```

### - Configuring a mesh node (Add to group)

If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func add(_ node: SBMMeshNode!, to group: SBMMeshGroup!, completion:
@escaping (Bool) -> Swift.Void)
```

## - Removing a configuration from the mesh node (Remove from group)

If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func remove(_ node: SBMMeshNode!, from group: SBMMeshGroup!,
completion: @escaping (Bool) -> Swift.Void)
```

## - Change name

To change device name it is needed to call method from the BluetoothMesh.

```
SBMBluetoothMesh:
open func change(_ node: SBMMeshNode!, name: String!)
```

## - Change proxy

To change proxy state for device it is needed to call method from the BluetoothMesh. If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func setProxy(_ node: SBMMeshNode, enable: Bool, completion:
@escaping (Bool) -> Swift.Void)
```

## - Change relay

To change relay state for device it is needed to call method from the BluetoothMesh. If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func setRelay(_ node: SBMMeshNode, enable: Bool, completion:
@escaping (Bool) -> Swift.Void)
```

## - Change friend

To change friend state for device it is needed to call method from the BluetoothMesh. If request will be processed with success then completion with True value will be called.

```
SBMBluetoothMesh:
open func setFriend(_ node: SBMMeshNode, enable: Bool, completion:
@escaping (Bool) -> Swift.Void)
```

# 5. Controlling

This API supports, so far, three server model that we are going to call controllers in this document

## a. Generic On Off Server

Considering that the user have the "Generic On Off Server" model configured (through addToGroup), it is possible to control the device using the following method

```
SBMBluetoothMesh:
open func setOnOff(_ node: SBMMeshNode?,
                   in group: SBMMeshGroup!,
                     status: UInt16,
              transitionMS: UInt32,
                   delayMS: UInt16,
              requestReply: Bool,
                   isFinal: Bool)
```

It is possible to get current state of "Generic On Off Server" model. UInt16 value from completion will have this status. 1 if status is ON, 0 if status is OFF.

```
SBMBluetoothMesh:
open func getOnOff(_ node: SBMMeshNode?,
                 from group: SBMMeshGroup!,
                 completion: ((UInt16) -> Swift.Void)!)
```

## b. Generic Level Server

Considering that the user have the "Generic Level Server" model configured (through addToGroup), it is possible to control the device using the following method

```
SBMBluetoothMesh:
open func setLevel(_ node: SBMMeshNode?,
                   in group: SBMMeshGroup!,
                     status: Int16,
              transitionMS: UInt32,
                   delayMS: UInt16,
              requestReply: Bool,
                   isFinal: Bool)
```

It is possible to get current state of "Generic Level Server" model. Int16 value from completion will have this status. Status is value with range from -32 767 to 32 767.

```
SBMBluetoothMesh:
open func getLevel(_ node: SBMMeshNode?,
                 from group: SBMMeshGroup!,
                 completion: ((Int16) -> Swift.Void)!)
```

## c. Lightning Lightness Server

Considering that the user have the "Lightning Lightness Server" model configured (through addToGroup), it is possible to control the device using the following method.

```
SBMBluetoothMesh:
open func setDimmable(_ node: SBMMeshNode?,
                      in group: SBMMeshGroup!,
                        status: UInt16,
                  transitionMS: UInt32,
                       delayMS: UInt16,
                  requestReply: Bool,
                       isFinal: Bool)
```

It is possible to get current state of "Lightning Lightness Server" model. UInt16 value from completion will have this status. Status is value with range from 0 to 65535.

```
SBMBluetoothMesh:
open func getDimmable(_ node: SBMMeshNode?, from group: SBMMeshGroup!,
completion: ((UInt16) -> Swift.Void)!)
```

# 6. Key refresh

## a. Starting key refresh procedure

To start network key refresh procedure the following method needs to be called.

```
SBMBluetoothMesh:
open func startKeyRefresh(_ network: SBMMeshNetwork,
                withTimeout timeout: UInt32,
                          completion: @escaping (Bool) -> Void)
```

Timeout specifies time in milliseconds after which a node is considered to have failed the procedure.

If key refresh procedure is finished completion is invoked with boolean parameter. This parameter is true when procedure finished successfully, otherwise it is false.

## b. Suspending key refresh procedure

To suspend key refresh procedure (eg. when mobile device leaves network) following method needs to be called.

```
SBMBluetoothMesh:
open func suspendKeyRefresh(_ network: SBMMeshNetwork)
```

Suspended procedure should be resumed when device reenters network.

## c. Resuming key refresh procedure

Resuming previously suspended key refresh procedure is done by calling following method.

```
SBMBluetoothMesh:
open func resumeKeyRefresh(_ network: SBMMeshNetwork,
                           completion: @escaping (Bool) -> Void)
```

If resumed key refresh procedure is finished completion is invoked with boolean parameter. This parameter is true when procedure finished successfully, otherwise it is false.

## d. Blacklisting node from key refresh procedure

To exclude specific node from key refresh procedure following method needs to be called.

```
SBMBluetoothMesh:
open func blacklist(_ node: SBMMeshNode)
```

Blacklisted node is effectively removed from taking part in procedure.