AN1200.1: iOS and Android ADK for *Bluetooth*® Mesh SDK 2.x and Higher

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

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

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

KEY POINTS

- Introduction to the Silicon Labs' Bluetooth mesh for iOS and Android ADK
- Getting started with development
- ADK usage

Table of Contents

1 Introduction

The iOS and Android (API version 27 or older) Bluetooth LE stacks do not have native support for Bluetooth mesh and therefore devices with these operating systems cannot directly interact with Bluetooth mesh nodes using the Bluetooth mesh advertisement bearer. However, the Bluetooth mesh specification 1.0 also defines a GATT bearer, which enables any Bluetooth LE-capable device to interact with Bluetooth mesh nodes over 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 for Gecko SoCs and Modules. The Silicon Labs Bluetooth Mesh ADK embeds the same stack and enable development of Bluetooth mesh applications for the iOS and Android systems.

The basic concept is that the native Bluetooth APIs of iOS and Android are used to discover and connect Bluetooth LE devices, while the ADK is used to manage the Bluetooth mesh-specific operations such as Bluetooth mesh security, device and node management, network, transport, and application layer operations.

2 Installation

2.1 Download

You have downloaded the ADK zip file from SiliconLabs/gecko_sdk GitHub site.

2.2 ADK Structure

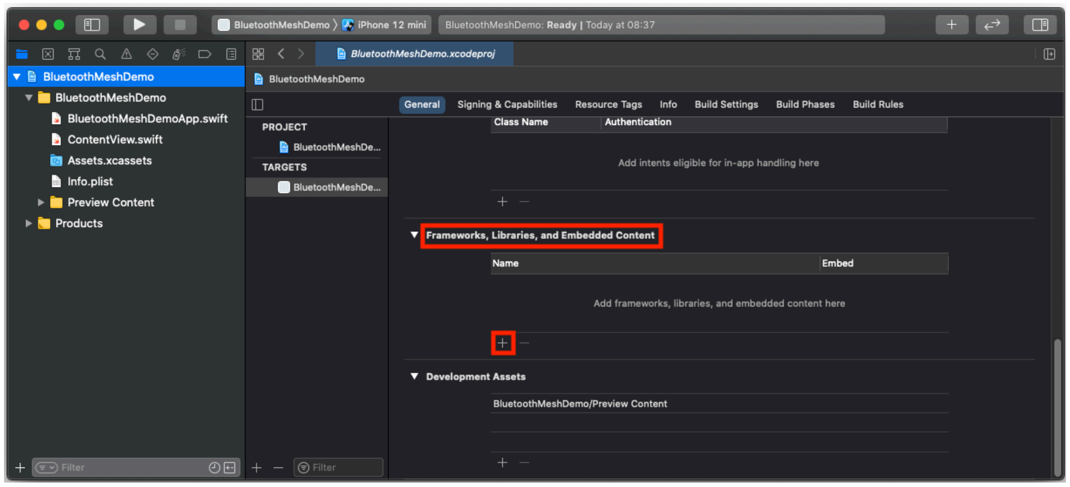
The ADK zip contains:

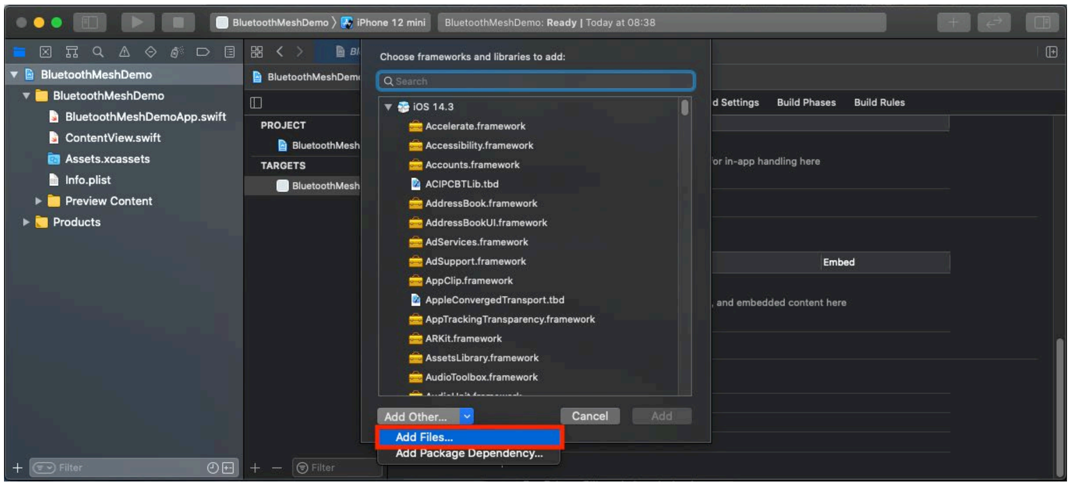
`./app/bluetooth/`	
`android/`	– Android ADK frameworks (debug and release)
`android/dokka/`	– Android ADK reference documentation
`android_application/`	– Android reference application (source code)
`ios/`	– iOS ADK Frameworks (debug and release)
`ios/docs/`	– iOS ADK reference documentation
`ios_application/`	– iOS reference application (source code)

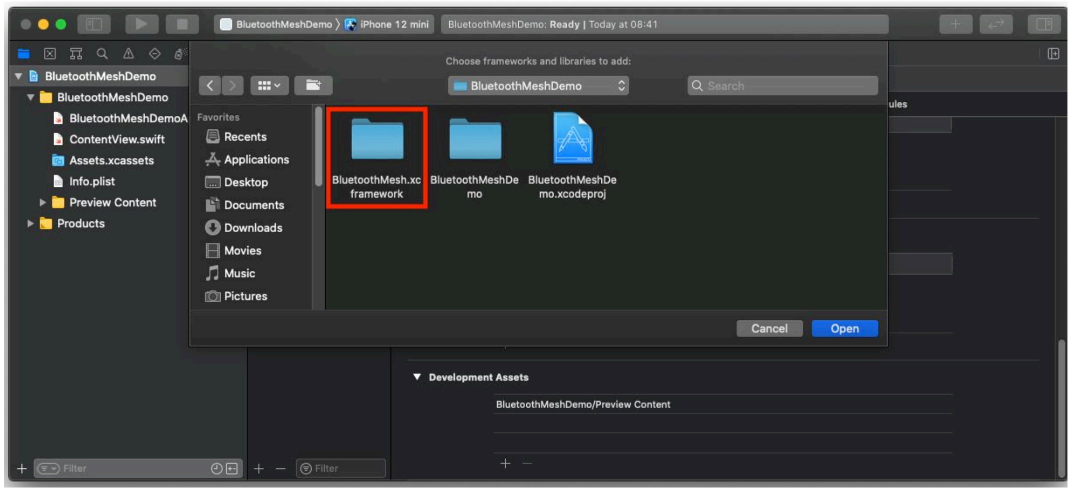
Disclaimer: The mobile apps are a reference apps for the Bluetooth Mesh mobile ADK, but it should not be taken as a starting point for customers to create their own mobile apps.

2.3 Set Up iOS Project

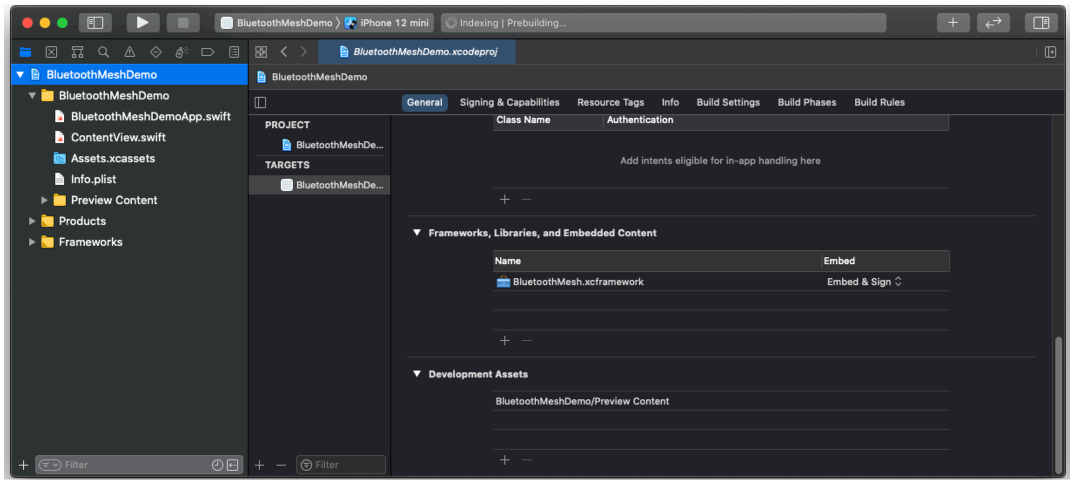
- Copy the BluetoothMesh.xcframework to a folder with the new iOS project (see section 2.2 ADK Structure for frameworks location).
- Open the main target in your project.
- Go to the General view.
- Add BluetoothMesh.xcframework to Frameworks, Libraries and Embedded Content.





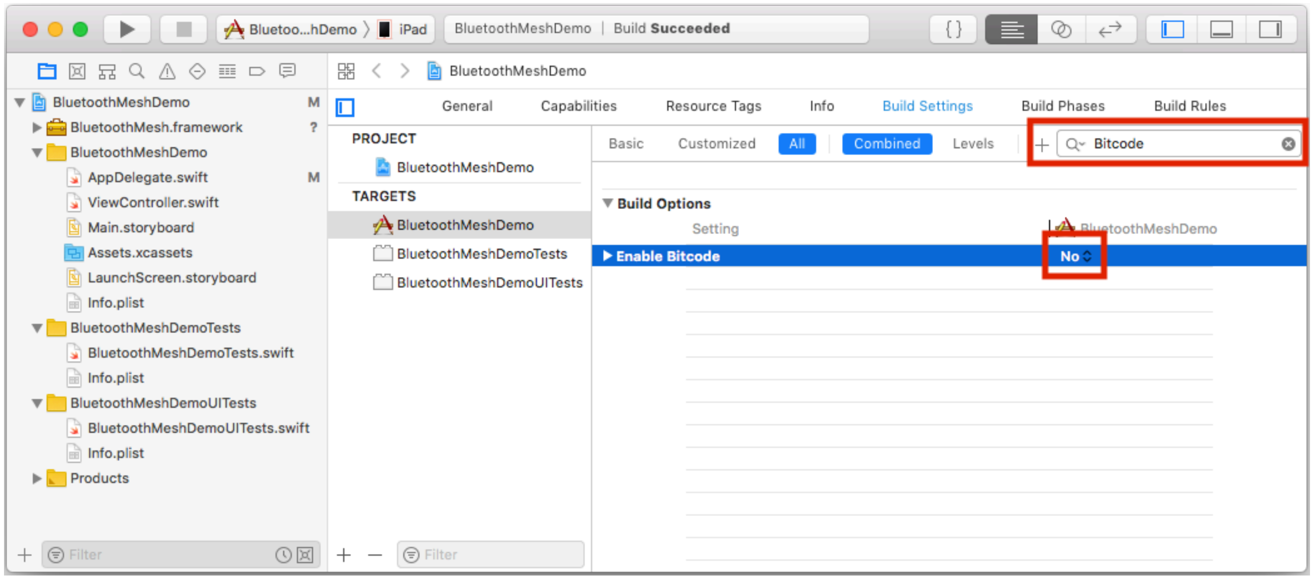

BluetoothMeshDemo 〉 iPhone 12 mini
BluetoothMeshDemo: Ready | Today at 08:41
Choose frameworks and libraries to add:
BluetoothMeshDemo
Search
Favorites
Recents
Applications
Desktop
Documents
Downloads
Movies
Music
Pictures
BluetoothMesh.xcframework
BluetoothMeshDemo
BluetoothMeshDemo.xcodeproj
Cancel
Open
Development Assets
BluetoothMeshDemo/Preview Content
Filter

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


BluetoothMeshDemo 〉 iPhone 12 mini
Indexing | Prebuilding...
BluetoothMeshDemo.xcodeproj
BluetoothMeshDemo
Frameworks
General
Signing & Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
Class Name
Authentication
Add intents eligible for in-app handling here
Frameworks, Libraries, and Embedded Content
Name
Embed
BluetoothMesh.xcframework
Embed & Sign
Development Assets
BluetoothMeshDemo/Preview Content
Filter

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

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



Enable Bitcode: No

2.4 Set Up Android Project

1. Create a project in Android Studio.
2. Copy the `*.aar` file to `<root path>/app/libs/` from the ADK package.
3. Add to `dependencies {...}` section of `<root path>/app/build.gradle`:

```
dependencies {
  ...
  implementation files('libs/ble_mesh-android_api-release.aar')
  implementation 'com.google.code.gson:gson:2.10.1'
}
```

4. Synchronize project after changes.
5. Initialize a `BluetoothMesh` object as shown in the following example.

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

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

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

6. Compile and run the project.

3 Usage: Basic Use Cases

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

- Network – the main container in the mesh structure. Network is the owner of subnets and nodes.
- Subnet – a specific subnet belongs to a network.
- Node – a node can be added to many subnets. A node contains elements.
- Element – an addressable entity within a device containing models.
- Model – defines a set of States, State Transitions, State Bindings, Messages, and other associated behaviors.

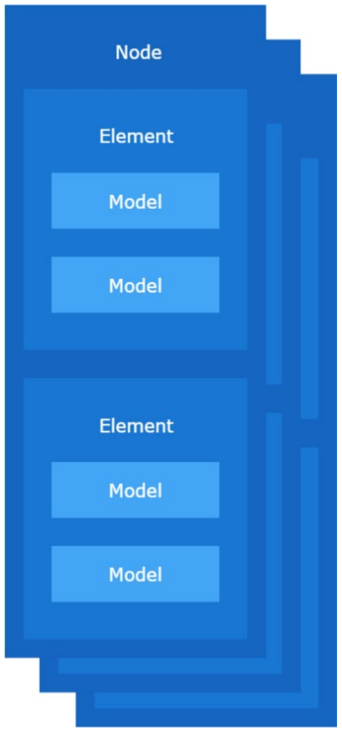


Figure 3-1. Node Structure

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

Full API documentation and the reference applications' source code are provided in the ADK package - see section 2.2 ADK Structure for its location.

3.1 Provisioning a Device (Over GATT Bearer)

Bluetooth device discovery must be done on the application side. Devices that can be provisioned advertise themselves with the Unprovisioned Device beacon and Bluetooth Mesh Provisioning Service shall be present in the GATT database.

To provision a new node, the ADK expects that object implementing:

iOS: `SBMConnectableDevice`
Android: `ConnectableDevice`
Represents the device you want to provision, which can be connected to before provisioning.

and

iOS: `SBMSubnet`
Android: `Subnet`
Represents the target subnet to which to provision node.

are provided.

The subnet can be created using:

iOS: `SBMNetwork.createSubnet`
Android: `Network.createSubnet`

To initiate the provisioning process, create

iOS: `SBMProvisionerConnection`
Android: `ProvisionerConnection`

Initiate the provisioning session using:

iOS: `SBMProvisionerConnection.provision`
Android: `ProvisionerConnection.provision`

3.2 Proxy Connection and Configuration

To configure provisioned nodes, the application must connect to subnet via proxy node.

The newly provisioned device will accept incoming proxy connection only for 60 seconds. If you intend to keep it as a proxy node you have to confirm its proxy role in this time window. Once you have one or more proxy nodes in your subnet, you may connect to that subnet and change settings of any node (including its proxy role) at any time.

To establish a proxy connection, initialize:

iOS: `SBMProxyConnection`
Android: `ProxyConnection`

object using:

iOS: `SBMConnectableDevice`
Android: `ConnectableDevice`

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

After initializing ProxyConnection object, function:

iOS: `SBMProxyConnection.connect`
Android: `ProxyConnection.connectToProxy`

must be invoked to establish the connection. There is an overloaded `ProxyConnection.connectToProxy` method in Android with an additional boolean `refreshBluetoothDevice` argument, which supports deciding if `BluetoothDevice` should be refreshed before connecting to the proxy node. This is needed to obtain current GATT services after they have been changed during the provisioning process, because subsequent `discoverServices` method calls on the `BluetoothGatt` object can result in cached services from the first call (even if device was disconnected). Refreshing `BluetoothDevice` before connecting to the proxy node is required only during the first connection and configuration after provisioning.

After establishing the connection, class:

iOS: `SBMConfigurationControl`
Android: `ConfigurationControl`

can be used to set proxy service and get device composition data using:

iOS: `SBMConfigurationControl.setProxy`
Android: `ConfigurationControl.setProxy`

and

iOS: `SBMConfiruationControl.getDeviceCompositionData`
Android: `ConfigurationControl. getDeviceCompositionData`

After getting raw Device Composition Data it should be passed to node using:

iOS:	`SBMNode.overrideDeviceCompositionData`
Android:	`Node.overrideDeviceCompositionData`

It will parse raw data into elements and models.

After establishing a connection with a proxy node, using the proxy server creates a filter list, which can be used to reduce the number of packets exchanged with the proxy node. This can be done using:

iOS:	`SMBProxyControl.accept, SBMProxyControl.reject`
Android:	`ProxyControl.accept, ProxyControl.reject`

which adds addresses to or removes them from the proxy filter list, depending on the list type.

The filter list can be either an accept list or a reject list. An Accept list is a list of destination addresses of the packets which the Proxy Server passes through to the Client. Other packets will not be received. Reject list is a list of destination addresses of the packets which the Proxy Server will not pass through to the Client. All other packets will be received.

By default, the Proxy Server always initiates the filter as an empty accept list. The type of list can be changed using:

iOS:	`SBMProxyControl.setFilterType`
Android:	`ProxyControl.setFilterType`

Whenever the Proxy Client sends a message to the network, the source address is added to its accept list or removed from its reject list, to let the Client receive a response. The following method can be used to get the current filter list type and number of entries in list on the Proxy Server:

iOS:	`SBMProxyControl.getFilterStatus`
Android:	`ProxyControl.getFilterStatus`

3.3 Binding Models

Supported mesh models are listed in `ModelIdentifier` class.

To bind application key to a model within the node, the following steps should be done:

- An application key exists within the subnet to which the node belongs. An application key can be created using:

iOS:	`SBMSubnet.createAppKey`
Android:	`Subnet.createAppKey`

- An application key must be bound to node using:

iOS:	`SBMNodeControl.bind`
Android:	`NodeControl.bind`

- An application key must be bound to model using:

iOS:	`SBMFunctionalityBinder.bindModel`
Android:	`FunctionalityBinder.bindModel`

Models can be found within elements of the node. The property

iOS:	`SBMNode.elements`
Android:	`Node.elements`

contains elements of a node. Each element contains an array of `sigModels` and `vendorModels`.

For example, to send Generic OnOff get message to the node, first we need to find:

iOS:	`SBMSigModel`
Android:	`SigModel`

in that Node's `sigModels` array, where the `modelIdentifier` is equal to:

iOS:	`SBMModelIdentifier.genericOnOffServer`
Android:	`ModelIdentifier.GenericOnOffServer`

3.4 Sending the Message

To send the Generic message use:

iOS:	`SBMControlElement`
Android:	`GenericClient`

Initialize it with:

iOS: `SBMElement` and `SBMApplicationKey`

which contains the previously found `GenericOnOffServer` model.

To send get message use:

iOS:	`SBMControlElement.getStatus`
Android:	`ControlElement.getStatus`

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

iOS:	`SBMGenericOnOff.self.`
Android:	`ModelIdentifier.GenericOnOffClient`

The response will be received in the success callback on iOS side and related flow on Android side.

4 Resources

4.1 Silicon Labs Resources

- Silicon Labs: Bluetooth LE
- Silicon Labs: Bluetooth Mesh Android and iOS Mobile Applications

4.2 Bluetooth SIG Resources

- Bluetooth Mesh specifications
- Bluetooth Mesh glossary
- Bluetooth Mesh articles

4.3 iOS: Complying with Encryption Export Regulations

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

More detailed explanation can be found here: Encryption Export Regulations

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

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

4.4 Android: Known Bluetooth Issues

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

Working with Bluetooth LE on Android is difficult, because:

- Device manufacturers make changes to the Android Bluetooth LE stack. Your application can work well on one device but could have problems on another.
- Documentation on Bluetooth LE describes only basic concepts, but does not provide enough information about managing connections, need for queuing operations, or dealing with bugs.

4.4.1 Scanning

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

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

Some applications may scan continuously, which would consume the phone's battery power. In order to limit this Android has implemented changes related to scanning. In Android 7.0 and newer versions there is protection against Bluetooth LE scanning abuse. If your app starts and stops Bluetooth LE scans more than 5 times within 30 seconds, scan results will not be received temporarily. Moreover, starting with Android 7.0, you can perform one scan with a maximum time of 30 minutes. After this time Android will change the scan mode to

SCAN_MODE_OPPORTUNISTIC. As of Android 8.1, if you do not set any ScanFilters scanning will be paused when the user turns off the screen, and will resume after the screen is turned on again.

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

4.4.2 Connecting

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

Auto connect

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

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

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

4.4.3 Managing a Connection

Connection State

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

Changing MTU

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

Discovering Services

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

Reading/Writing Characteristics

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

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

Disconnecting

In Android you might have problems with performing some operations, as reconnecting, after improperly disconnecting from a device. There is a timeout while the phone continues opening connection events and the device is not fully disconnected, so you could have trouble connecting to it again. In Android this timeout was hardcoded to 20 seconds and has been changed to 5 seconds in Android 10, so it can take a lot of time until you are notified about the closed connection. In iOS this usually takes less than 1 second. If you want to reconnect immediately after disconnecting you could get status code 22, so it would be better to wait about 500 milliseconds before the connection attempt.

4.4.4 Errors

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

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

5 Open-Source Licenses Used

Table 5-1. Open-Source Licenses Used

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

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com