BLE on Android v1.0.1

December 5, 2016

Liability disclaimer

Life support applications

Contact details

For your nearest dealer, please see www.nordicsemi.com

Main office:

Otto Nielsens veg 12
7004 Trondheim
Phone: +47 72 89 89 00
Fax: +47 72 89 89 89
www.nordicsemi.com

DNV
NORWEGIAN ACCREDITATION
No. S03
NS-EN ISO 9001 CERTIFICATED FIRM

RoHS statement

Revision history

Date	Version	Description
Nov 2016	1.0	First release
Dec 2016	1.0.1	Update based on comments by Emil Lenngren

Contents

1. **Permissions** 3

2. **Scanning** 4

2.1 Android 4.3 and 4.4.x 4
2.2 Android 5 and 5.1 4
2.2.1 Scan mode 4
2.2.2 Report delay 5
2.3 Android 6+ 5
2.3.1 Callback type 5
2.3.2 Match mode 6
2.3.3 Number of matches 6
2.4 Receiving packets from a single device 6

3. **BluetoothDevice object** 7

4. **Connection** 8

4.1 The autoConnect parameter 8
4.2 BLE operations 8
4.3 Connection parameters 10
4.4 MTU 10
4.5 Packets per connection parameter 11
4.6 Protected services 12

5. **GATT server** 13

6. **Advertising** 14

1. Permissions

Android 6 divided all permissions into two protection level groups: normal and dangerous. Like before, each permission that an app requires must be specified in the AndroidManifest file. However, normal permissions are granted on installation, while dangerous permissions must be requested in runtime by the application when such is needed. For more information, see Permissions normal-dangerous.

On Android, an app must have the BLUETOOTH_ADMIN permission to **scan** for BLE devices and to **connect** to them. If an app just needs to **connect** to already **paired devices**, it is enough that the BLUETOOTH permission is requested.

Both BLUETOOTH_ADMIN and BLUETOOTH permissions belong to the normal permissions group. For more information, see List of normal permissions.

On Android 6, an app must also request a permission from the LOCATION group in order to scan for BLE devices. The permissions in this group are ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION. The protection level for this group is dangerous as it involves user's private data in that beacons can be used to track user's location. Android requires user's explicit approval to reveal a possible loss of privacy. This is why nRF Connect asks for Location permission when the SCAN button is clicked for the first time.

2. Scanning

We recommend using Android Scanner Compat Library for scanning on Android. It brings almost all new Android scanning features to the older platforms.

2.1 Android 4.3 and 4.4.x

On Android 4.3 and 4.4.x, there were two methods to start BLE scanning:

- BluetoothAdapter#startLeScan(LeScanCallback)
- BluetoothAdapter#startLeScan(UUID[], LeScanCallback)

When scanning, they were using a lot of battery and returning all advertising packets, though were not recommended for use in background services. Every application with the scanning feature in a background service would have to start and stop scanning every few seconds on its own, but as each app would do it independently from others, this could end up with 100% time of scanning. This has been solved in the new API in Android 5 with scanning modes (see Scan mode below).

A further restriction was that the method with UUID[] service filter could only be used with 16-bit UUIDs. A custom UUID filtering wasn't working so this method was practically useless.

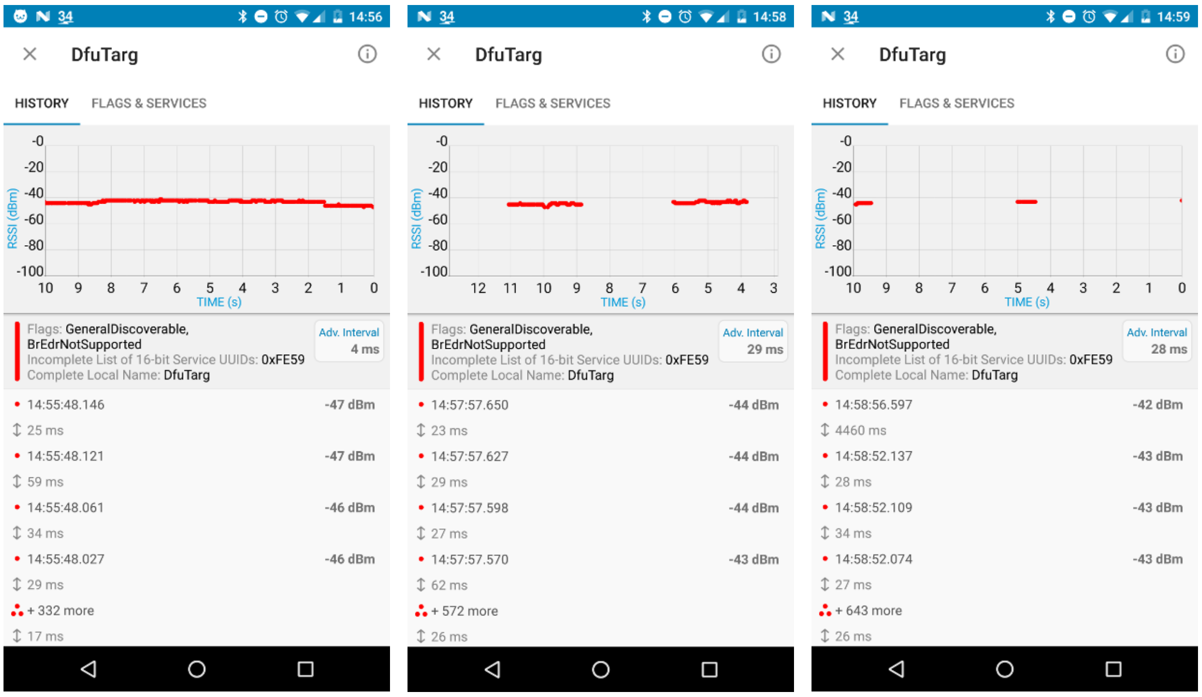
2.2 Android 5 and 5.1

On Android 5+, scanning is performed using BluetoothLeScanner which can be obtained using BluetoothAdapter#getBluetoothLeScanner(). This method returns *null* if Bluetooth is disabled.
A list of filters and scan settings, including scan mode and report delay, can be set using the ScanSettings.Builder object.

2.2.1 Scan mode

ScanSettings.Builder#setScanMode(int) sets the scan mode. There are three options to choose from:

- SCAN_MODE_LOW_LATENCY
- SCAN_MODE_BALANCED
- SCAN_MODE_LOW_POWER



Flags: GeneralDiscoverable, BrEdrNotSupported
Incomplete List of 16-bit Service UUIDs: 0xFE59
Complete Local Name: DfuTarg
Adv. Interval 4 ms

```
• 14:55:48.146        -47 dBm
↕ 25 ms
• 14:55:48.121        -47 dBm
↕ 59 ms
• 14:55:48.061        -46 dBm
↕ 34 ms
• 14:55:48.027        -46 dBm
↕ 29 ms
+ 332 more
↕ 17 ms
```

Flags: GeneralDiscoverable, BrEdrNotSupported
Incomplete List of 16-bit Service UUIDs: 0xFE59
Complete Local Name: DfuTarg
Adv. Interval 29 ms

```
• 14:57:57.650        -44 dBm
↕ 23 ms
• 14:57:57.627        -44 dBm
↕ 29 ms
• 14:57:57.598        -44 dBm
↕ 27 ms
• 14:57:57.570        -43 dBm
↕ 62 ms
+ 572 more
↕ 26 ms
```

Flags: GeneralDiscoverable, BrEdrNotSupported
Incomplete List of 16-bit Service UUIDs: 0xFE59
Complete Local Name: DfuTarg
Adv. Interval 28 ms

```
• 14:58:56.597        -42 dBm
↕ 4460 ms
• 14:58:52.137        -43 dBm
↕ 28 ms
• 14:58:52.109        -43 dBm
↕ 34 ms
• 14:58:52.074        -43 dBm
↕ 27 ms
+ 643 more
↕ 26 ms
```

SCAN_MODE_LOW_LATENCY SCAN_MODE_BALANCED SCAN_MODE_LOW_POWER

Figure 1 Scan modes

In the balanced and low power modes, the interval is 5 seconds. In the balanced mode, it scans for 2 seconds, then waits for 3 seconds, while in low power mode, it scans for 0.5 seconds and waits for 4.5 seconds.

Packets should also be reported when another app scans in a more aggressive mode.
There is also a fourth, special scan mode, opportunistic, which means that your app will receive advertising packets only when another application, for example Google Play Services, is scanning at the same time.

2.2.2 Report delay

ScanSettings.Builder#setReportDelay(long) sets report delay in milliseconds. Instead of notifying about each received packet separately when it's found, the app gets a list of packets every number of milliseconds. This has been added to save battery as batching may be done on a very low level.

Bug: On Samsung S6 and Samsung S6 Edge all reported packets using report delay > 0 have equal RSSI (for older version, or RSSI = -120 dBm in the current version). Therefore, if RSSI is important this method should not be used.

2.3 Android 6+

2.3.1 Callback type

ScanSettings.Builder#setCallbackType(int) sets the callback type. There are three types available:

- CALLBACK_TYPE_ALL_MATCHES - Trigger a callback for every Bluetooth advertisement found that matches the filter criteria. If no filter is active, all advertisement packets are reported.
- CALLBACK_TYPE_FIRST_MATCH - A result callback is only triggered for the first advertisement packet received that matches the filter criteria. This is useful if "beacon in range" scenario is needed.
- CALLBACK_TYPE_MATCH_LOST- Receive a callback when advertisements are no longer received from a device that has been previously reported by a first match callback. Useful to detect that a beacon is out of range.

nRF Connect 4.8 does not allow to set the callback type. Only the ALL_MATCHES (default) mode is used in scanning.

2.3.2 Match mode

ScanSettings.Builder#setMatchMode(int) sets the match mode. There are two types available:

- MATCH_MODE_AGGRESSIVE - In Aggressive mode, hw will determine a match sooner even with feeble signal strength and few number of sightings/match in a duration. This mode is default.
- MATCH_MODE_STICKY - For sticky mode, higher threshold of signal strength and sightings is required before reporting by hw. The threshold RSSI may be chip-dependent.

See also: Android-6.0-Bluetooth-HCI-Reqs.pdf Chapter 6.1.3, *Batch_scan_Discard_Rule.*
nRF Connect 4.8 does not allow to modify this parameter and only the aggressive (default) option is used.

2.3.3 Number of matches

ScanSettings.Builder#setNumOfMatches(int) allows to set the number of matches for Bluetooth LE scan filters hardware match. It seems to be related with Report Delay and says how many packets we want from a device in a single batch. Three types are available:

- MATCH_NUM_ONE_ADVERTISEMENT - Match one advertisement per filter.
- MATCH_NUM_FEW_ADVERTISEMENT - Match a few advertisements per filter, depends on current capability and availability of the resources in hw.
- MATCH_NUM_MAX_ADVERTISEMENT - Match as many advertisements per filter as hw could allow, depends on current capability and availability of the resources in hw.

nRF Connect 4.8 does not allow to set this parameter.

2.4 Receiving packets from a single device

On some devices, for example Nexus 4 and 7, a scanning application receives only one packet from connectable devices per scan – only one Scan Request is sent. Subsequent advertising packets are ignored. This makes listening to RSSI changes complicated. An app would have to stop and start scanning every couple of seconds. Such a mode can be configured in nRF Connect by setting *Settings -> Scanner -> Continuous scanning* to Disabled. However, if a peripheral advertises as a non-connectable device, every packet received is reported to the application.

The majority of Android devices, however, send a Scan Request for each Advertising Packet received and both Advertising Packet and Scan Response are reported multiple times.

3. BluetoothDevice object

In order to connect to a BLE device, an app needs to obtain a BluetoothDevice object. This can be done using one of three methods:

- By scanning: a BluetoothDevice object is returned for each scanned advertising packet in a callback parameter ScanResult.
- By obtaining a list of bonded devices using BluetoothAdapter#getBondedDevices().
- By creating a device object using BluetoothAdapter#getRemoteDevice(String address).

When a BluetoothDevice object is obtained using the last method (or by creating a new object with *new BluetoothDevice(address)*), a connection attempt to it will work only if:

- The device has a PUBLIC address or is bonded, or
- The device has been scanned at least once before connection attempt since Bluetooth adapter have been started

Explanation: In order to connect to a BLE device, a master device needs to know its 48-bit address and the address type (public/random). The method above takes only an address (in the format AA:BB:CC:DD:EE:FF) as a parameter, which means that the "type" information is missing. Android has a "security database" in the stack layer where it stores triples: address, device type, and the address type of all scanned addresses. Every time a new device is found, a row is added to this database.

If a device hasn't been scanned, Android assumes that the given address is a public one, the logic being that random addresses can change any time, and if it is known without scanning, most probably it is a public one. It is a known Android bug and the work is being done to fix it. The result should be available in Android 7.2 or later.

This may cause a problem for example when starting a service that should reconnect to a device after a reboot. The Android app may have the address saved, but until it's scanned and found, the connection will not succeed as the Connect Request will have the flag set to Public Address. Currently, the recommended way of restoring a connection to non-bonded device it is to scan for it and start connecting (with autoConnect true or false) when found (note that the device does not have to be in range when the phone is restarted/Bluetooth enabled).

Bug: When a connection is established with *autoConnect=true* to an unknown device (one that has not been scanned since Bluetooth was enabled) the Android assumes the address is RANDOM, not PUBLIC. This is the opposite to when this flag is cleared. This is the commit (2016-02-02) that changed this: https://android.googlesource.com/platform/system/bt/+/d36b421035fe3b7d086f5d7737d8ba9fbdc471b3%5E%21/ and https://android-review.googlesource.com/#/c/200376/. See the last else-block, i.e. under "not a known device, i.e. attempt to connect to device never seen before". Now, instead of always choosing public address it seems they do some heuristics assuming random address, but if it is neither a non-resolvable, resolvable or static random address, then it is certainly a public address. The problem is that the bit-check is written incorrectly to always return false, so random address is always chosen. That was however fixed by https://android.googlesource.com/platform/system/bt/+/42e055353ca397f3aaf350b99e79d42f79a478e1%5E!/. That commit does not seem to be included in the Android 7.1 Nougat releases, so the change may be available since next Android release.

4. Connection

On Android, a peripheral BLE device may be used by more than one application at the same time. The physical connection is managed by the system, while each app “thinks” it’s the only one connected to this device. There is no way to check if another app is also connected without getting this information from the peripheral (checking if it’s advertising, reading a value, etc.). When an app disconnects from the device, it will receive a callback BluetoothGattCallback#onConnectionStateChange(BluetoothGatt, int status, int newState) with newState = STATE_DISCONNECTED but it doesn’t mean that the device is in fact disconnected. To verify that a device has actually disconnected, an app may get a list of connected devices using BluetoothManager#getConnectedDevices(int) with the GATT parameter and check if the peripheral is listed there.

To ensure that the device will be disconnected, some kind of disconnect operation must be implemented on the peripheral side (for example OpCode “Disconnect”, etc.), and it is the peripheral that must close the connection.

4.1 The autoConnect parameter

Connecting to a BLE device is done by calling the BluetoothDevice#connectGatt(Context, boolean autoConnect, BluetoothGattCallback) method. The second parameter, autoConnect, indicates if Android should keep connecting to this device whenever it is discovered until the BluetoothGatt#close() method is called, or the Bluetooth adapter is turned off. If it’s *true*, the only difference regarding that is that the address is provided by the white list instead of the HCI LE Create Connection command. Also the standard configuration (which may be different between Android phone vendors) is to use more active scanning parameters when using autoConnect=false which leads to faster connection setups. Whenever a device from the whitelist is discovered, the system will try to connect to it.

There are more differences: once the device disconnects with a reason other than the gatt object was disconnected/closed, a new connection attempt will be made automatically if either the autoConnect parameter was true or the connect() method was called on the gatt object, i.e. if you used autoConnect=false no further connection attempt will be made. With autoConnect=false the connection attempt will also get a timeout of 30 seconds. autoConnect=true has no timeout. Lastly, autoConnect=false will temporarily cancel all other pending connections to devices in the white list and autoConnect=false attempts are queued up. You should also be aware of https://code.google.com/p/android/issues/detail?id=228633 and certainly of https://code.google.com/p/android/issues/detail?id=69834 if you use *autoConnect=true* (the latter issue was fixed in Android Nougat)

An address may be added to the whitelist only if at least one of the following is true:

- A device is bonded
- The address is a public one
- The address is a random static address

Random non-resolvable and random resolvable addresses will not be added to this list if a device is not bonded, which means that a connection to such address may never be completed (there is no callback at all, even no timeout) as it would with this parameter set to *false*.

4.2 BLE operations

After a connection to a device is established, an app may send or receive data using asynchronous calls using the BluetoothGatt object API. This object is returned by BluetoothDevice#connectGatt(Context, boolean autoConnect, BluetoothGattCallback).
A corresponding callback must be received before a new method is called, so it is not possible to send data in a loop just by writing characteristics. The correct way of sending more than one packet is to write a characteristic, wait for a callback, write the next packet, wait for a callback, etc.

To make this easier, the BleManager class in nRF Toolbox v.2.0 introduced a queue of requests. An app using this manager may now enqueue many requests and they will be performed one after another.

The BluetoothGattCallback#onCharacteristicWrite(BluetoothGatt, BluetoothGattCharacteristic, int status) callback is called for both Write Request (when a response has been received) and Write Command (when the packet has been added to the local outgoing buffer). In the latter case, when the buffer gets full, the BLE may either stack (Android 4.3 – 5x) or the callback may be postponed until the buffer is empty (Android 5.1+(?)). If the device loses the connection while the internal queue is full, no further GATT operations will work on that GATT object once it reconnects. See https://code.google.com/p/android/issues/detail?id=223558.

Handling a full buffer is shown in the following image:

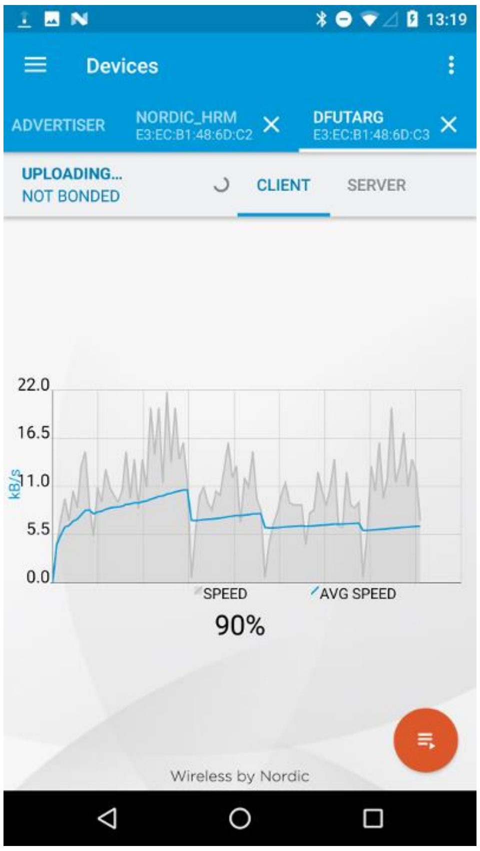


Figure 2 Handling full buffer

Gray color indicates how fast a characteristic may be written using Write Without Response (write, wait for callback, write, callback, etc.). The speed drops down to 0 when the callback is postponed until the buffer empties.

4.3 Connection parameters

Since Android 5 (API 21), three connection priorities can be requested using BluetoothGatt#requestConnectionPriority(int). This method is asynchronous (like all BLE related methods) but there is no callback called when it's completed/rejected. This is a known bug, which will be fixed in the future.

Table 1 Connection priorities

	Connection Interval (ms)		Latency	Timeout (sec)
	Android 5	Android 6+		
CONNECTION_PRIORITY_HIGH	**7.5 – 10**	**11.25 – 15**	**0**	**20**
CONNECTION_PRIORITY_BALANCED	**30 – 50**		**0**	**20**
CONNECTION_PRIORITY_LOW_POWER	**100 – 125**		**2**	**20**

The commit in AOSP (Android Open Source Project) that has changed these values can be found here:
https://android.googlesource.com/platform/packages/apps/Bluetooth/+/434bd21a120078b2944a22828f64714df414c5d9%5E1..434bd21a120078b2944a22828f64714df414c5d9/

The current configuration (from master) is here:
https://android.googlesource.com/platform/packages/apps/Bluetooth/+/master/src/com/android/bluetooth/gatt/GattService.java#1632

The values are set from a config file:
https://android.googlesource.com/platform/packages/apps/Bluetooth/+/0e94b62d60f3a729b03841a1891af3a889c9748f%5E2..0e94b62d60f3a729b03841a1891af3a889c9748f/

The current version of the config file can be found here:
https://android.googlesource.com/platform/packages/apps/Bluetooth/+/master/res/values/config.xml#39

This method works only if Android is the master. If it has advertised and another device gets connected to it, this method seems to be doing nothing.
If another value is needed, it has to be requested from the peripheral. Mind, that a new restriction in Android 6+ is that connections interval lower than 11.25 ms. **are not possible anymore** (which is important if you use the Connection Parameter Update Request from the slave side) due to some performance issues Google had.

4.4 MTU

Since Android 5 (API 21), MTU may be requested using BluetoothGatt#requestMtu(int).
The lowest MTU is **23** (default), the highest is **517**. There is a callback called BluetoothGattCallback#onMtuChanged(BluetoothGatt, int, int) when this asynchronous operation completes with the status and final MTU.

Both Connection Priority and MTU may be tested using the Android version of nRF Connect 4.3 or newer.

4.5 Packets per connection parameter

Different Android phones send and receive a different number of packets in a single connection interval, depending on their Bluetooth chip capabilities. Here's a short list of devices that have been tested using a precompiled sample Secure DFU bootloader from SDK 12.1 by sending a sample HRM app using nRF Connect 4.8 (Android) and nRF Toolbox (iOS). During the test, a phone/tablet was sending eight times 205 Writes Without Response packets as fast as possible (one object is 4096 bytes which requires 205 20-byte-long packets).

On Android, each write must be done when a callback for the previous write operation has been received, so the system has a way to pause the sending process when the buffer gets full (handling this situation was fixed in Android 5 or 5.1). There is no such callback on iOS when writing without response, and therefore sending bytes in a loop may cause a buffer overflow.
Regarding older devices where overflow is not handled well, as well as on iOS, an easy workaround is to send each 15th packet or so as a Write With Response. Then you don't need to create any special acknowledgement logic like Packet Receipt Notifications in DFU on the peripheral side. It of course reduces speed a bit, but rather that than dropped packets.

Table 2 Number of packets in a connection interval

Device	OS Version	Connection interval (ms) [1]	Packets sent in connection interval	PRN Required /max PRN [2]
Nexus 6P	7.1.1 (beta)	30	4	No
Nexus 5X	7.1.1 (beta)	30	12[3]	No
Nexus 9	7.0	30	4	No
Nexus 6	6.0.1	30	4	No
Nexus 4	5.1.1	30	1	No
Samsung S3	4.3	48.75	1	Yes / ~340 [4]
Samsung S7	6.0.1	30	5	No
HTC One M8	6.0.1	30	9+[5]	No
Sony Xperia Z5 Compact	6.0.1	30	4	No

For comparison, measurements done on iOS:

Table 3 Measurements on iOS

Device	OS Version	Connection interval (ms)	Packets sent in connection interval	PRN Required /max PRN
iPhone 7	10.1	30	9+	Yes / 22
iPhone 6 Plus	10.1	30	7	Yes / 29
iPhone SE	9.3.3	30	8	Yes / <~20
iPhone 5s	10.0.2	30	7	Yes / <~20
iPhone 4S	8.1.3	30	7	Yes / <~20
iPad Mini	9.3.4	30	6	Yes /<~20
iPad Air	9.3	30	6	Yes / <~20

[1] This is not the minimum connection interval, but the actual one during performing DFU.
[2] Max supported PRN – the higher the number, the better. No value is even better as it means that PRNs are not required.
[3] Based on a comment here.
[4] Tested using Legacy DFU. As the maximum object size in Secure DFU (4096 bytes) is 205 packets long, no PRNs are required when doing Secure DFU on that phone model.
[5] The DFU Bootloader wasn't sending ACK for the 9th packet. If it did there could perhaps have been more packets sent.

4.6 Protected services

Android does not allow to read or write data to the following characteristics (and their descriptors):

1. HID Service (since Android 5):
    a. HID Information
    b. Report Map
    c. HID Control Point
    d. Report
2. FIDO (https://fidoalliance.org/) (since Android 6)
    a. U2F (0000FFFD-0000-1000-8000-00805F9B34FB)

Only applications with BLUETOOTH_PRIVILEGED, that is, applications signed with the same certificate as the system, are able to read and write data to those characteristics.

Explanation: On Android 4.3 and 4.4.x applications are able to sniff to HID devices and read passwords and other private information. U2F is used by security tokens.

5. GATT server

GATT server is supported since the beginning, that is, since Android 4.3.
An app may add 0+ services to the phone server but it is not aware of any other service that other apps have already added to it. A peripheral device will discover a sum of services created by all apps using GATT server + 2 additional, namely Generic Access and Generic Attribute, which are added by the system. Therefore, it may happen that the app service will never be used if the same service has already been created before.

Opening the server must be done using BluetoothManager#openGattServer(Context, BluetoothGattServerCallback) method. When an app has opened the server, it will immediately receive a BluetoothGattServerCallback#onConnectionStateChange(BluetoothDevice, int status, int newState) callback for each device that is already connected or getting connected to the Android device. This is how nRF Connect discovers devices that connected when the app got launched or was in the background.

If the app intends to use the connection, it should call the BluetoothGattServer#connect(BluetoothDevice, boolean autoConnect) method, giving the device as a parameter. Otherwise Android can close the connection if another (or this) app disconnects as not being used. To release a server connection, the BluetoothGattServer#cancelConnection(BluetoothDevice) method must be called. However, calling this method will not disconnect the device from the server. The peripheral device should disconnect itself when it gets disconnected by the client.

Example:
A Proximity tag nRF Proximity by Pebble is not releasing the server connection when disconnected.

1. In nRF Connect, configure the GATT server and create the predefined Link Loss and Immediate Alert services.
2. Connect to the tag and test two-way communication.
3. Disconnect by pressing DISCONNECT button in the app.
4. Observe in the log that server.cancelConnection(device) method has been called and the device has got disconnected (Android has disconnected from it). You may still click the button on the tag to trigger an alarm.

6. Advertising

Advertising is supported on newer devices with Android 5+. Devices that got updated to Android 5 but were designed for an older version do not support it (for example Nexus 4 or Nexus 5). To verify that an Android 5+ device supports advertising (or has a high chance of this), use the BluetoothAdapter#isMultipleAdvertisementSupported() method. However, this method always returns *false* if the *Bluetooth*® adapter is disabled, or on some devices, e.g. Sony Xperia™ Z1 Compact, it returns *true* even though the device in practice does not support advertising (AdvertiseCallback#ADVERTISE_FAILED_TOO_MANY_ADVERTISERS error is returned even with one advertisement set).

For more information, see the StackOverflow post.
In order to start advertising, an app must obtain a BluetoothLeAdvertiser object using BluetoothAdapter#getBluetoothLeAdvertiser(). This method also returns *null* if *Bluetooth* is disabled on the phone/tablet, or when advertising is not supported. On Sony Xperia Z1 Compact, and perhaps on some other devices, it returns an object.

The advertiser API allows to configure the advertising data, scan the response data, and set AdvertiseSettings. The data for both packets can be configured using AdvertiseData.Builder with the following API:

1. addManufacturerData(int manufacturerId, byte[] data)
   Adds a manufacturer data field
   Bug: Only the data added first is in fact added to each packet, all the following ones are ignored,
2. addServiceData(UUID serviceUUID, byte[] data)
   Adds service data field
   Bug: Only 16-bit UUIDs are supported, 128-bit UUIDs are truncated to 16-bit ones,
   Bug: Only the first data is advertised in each packet,
3. addServiceUuid(UUID serviceUUID)
   Adds a Complete List of Service UUID field,
4. setIncludeDeviceName(boolean include)
   Adds the device name to the packet or removes it. The name can't be set to a custom value using this API, but it may be set in Bluetooth Settings for the Android device,
5. setIncludeTxPowerLevel(boolean include)
   Includes TX power information. TX power may be set in AdvertiseSettings. The value advertised is automatically set to a proper value.

The allowed settings:
1. Advertise mode to one of the following:
   a. ADVERTISE_MODE_LOW_LATENCY – Advertising interval: **100 ms**
   b. ADVERTISE_MODE_BALANCED – Advertising interval: **250 ms**
   c. ADVERTISE_MODE_LOW_POWER – Advertising interval: **1000 ms**
2. TX power to one of the following:
   a. ADVERTISE_TX_POWER_HIGH – TX power: **+1 dBm**
   b. ADVERTISE_TX_POWER_MEDIUM – TX power: **-7 dBm**
   c. ADVERTISE_TX_POWER_LOW – TX power: **-15 dBm**
   d. ADVERTISE_TX_POWER_ULTRA_LOW – TX power: **-21 dBm**
3. The connectability of the device using setConnectable(boolean connectable)
4. Advertising timeout using setTimeout(int milliseconds)

For more information, see:
https://android.googlesource.com/platform/packages/apps/Bluetooth/+/master/src/com/android/bluetooth/gatt/AdvertiseManager.java#258

Usually an Android device can advertise with five–six packets set at the same time. When trying to start the next advertisement, an error AdvertiseCallback#ADVERTISE_FAILED_TOO_MANY_ADVERTISERS is returned by the callback.