MOBILE APP DEV

4 Tips to Make Android BLE Actually Work

September 10, 2019 | By: Chee Yi Ong

Android development is hard, and it's no thanks to its relatively fragmented ecosystem and having a million OEM's out there. Throw BLE—which is extremely hardware and Bluetooth spec. dependent!—into the mix and the platform's behavior becomes unpredictable. Unlike iOS, it's not feasible financially or logistically to attain all the Android devices out there for testing purposes. You may have heard that BLE development on Android can be a nightmare, and we're here to tell you that all the bad things you heard are most likely true, but that's no reason to quit—with the right tools and a methodical mindset, it's possible to achieve robust and high-performing BLE on Android.

In this post, we'll go over some of the common pitfalls in developing a BLE-capable Android app and how you can work around those, as well as provide some tips that we think will help along the way. A quick note: it helps if you're at least familiar or somewhat comfortable with Android BLE-related APIs, and have attempted to write an Android app that communicates via BLE. But, if you don't have any experience whatsoever, feel free to keep reading and see why BLE development on Android isn't as scary as most people think.

Tip 1: Target a minimum API level of 21

BLE support was added to the Android SDK back when Android 4.3 (Jelly Bean, API level 18) was released, but our experiences with BLE on Android 4.3 and 4.4 (KitKat, API level 19) had been less than ideal: there were threading issues in the BLE scan callback that developers had to work around by manually juggling threads (which is always a bad idea), and certain Samsung and LG devices required a restart of their Bluetooth radios to kick-off a BLE scan that had previously been stopped. We also tried to filter scan results by service UUID using the provided API method, but the scan simply stopped returning results altogether.

Needless to say, those were extremely frustrating problems that we hoped we'd never have to deal with again, and Google seemed to have gotten the not-so-subtle hints of developers complaining on Stack Overflow as they released Android 5.0 (Lollipop, API level 21) with "more powerful Bluetooth low energy capabilities" that also fixed a lot of pain points around working with BLE on Android. Therefore, **we recommend targeting a minimum API level of 21 (Android 5.0)** due to more developer-friendly APIs like `BluetoothLeScanner` and `ScanFilter`, as well as a more stable experience in general—with scans working more reliably and connections dropping less often.

Sure, around 7.4% of Android 4.3 and 4.4 users won't be able to use your app, but we think this is a worthwhile tradeoff rather than having to manage version-specific—or worse, OEM-specific—code paths and behaviors.

Tip 2: Don't perform BLE operations in rapid-fire succession

As developers, we often expect SDK's to abstract implementation restrictions away from us so we can focus on writing app code instead of spending time wrestling with some lower-level implementation details. Alas, a proper, synchronized queuing mechanism is vital to BLE operations succeeding on Android. The gist of this statement is to **only do something once you've heard back from Android about the previous operation**. This is especially true for read and write operations that need to happen back-to-back when a large payload is being transferred, or for connection setup tasks like service discovery and requesting a new MTU.

It's also highly recommended to **take advantage of language-specific synchronization features** like marking Java methods and fields with the `synchronized` keyword and to use thread-safe data structures like `ConcurrentLinkedList` and `ConcurrentLinkedQueue` if a method or field is ever going to be accessed from your concrete implementation of any Bluetooth callback interfaces. Your queuing mechanism can be implemented using a `ConcurrentLinkedQueue` that executes operations on a FIFO basis, for instance.

Tip 3: Handle error codes in the BluetoothGattCallback interface

For `BluetoothGattCallback` methods like `onCharacteristicWrite(...)` which have a status argument in them, *always* ensure that the status value is equal to `BluetoothGatt.GATT_SUCCESS` before proceeding. For values other than the success case, we typically handle it by checking if the error is something the user would understand (e.g. `BluetoothGatt.GATT_READ_NOT_PERMITTED`) before surfacing it up to the user. If an error happens multiple times in a row and you've taken the utmost care to ensure things are working just fine, the unfortunate truth is that things would sometimes stop working randomly. Most random problems should fix themselves if the app reestablishes the connection from scratch by calling `BluetoothGatt.close()` followed by `BluetoothDevice.connectGatt(...)` again. Note that this process can be transparent to the user, similar to how most apps automatically attempt to reconnect to a device once a connected device goes out of range.

While we're on the topic of Android Bluetooth error codes, there are status codes like "133" which are commonly seen, but their severity is quite understandably often blown out of proportion by the fact that they're undocumented. Since Android is open-source, we can take a look at the gatt_api.h file that helps shed some light as to what some of these error codes mean. In our case, "133" is 0x85 in hex, which is defined as `GATT_ERROR` in the list of constants found in the header file. Even though there isn't any documentation around what `GATT_ERROR` means, at least we know it implies something bad that is likely irrecoverable, and we can surface this information up to our users and ask them to try again if the error is caused by a user action.

Bonus: in our experience, the "133" error code is usually present when a connection request was refused by the peer or a `connectGatt(...)` call has timed out after approximately 30 seconds, likely because the remote device has moved out of range.

Tip 4: Don't rely on usages of private APIs shown in old example code

A significant amount of blog posts and Stack Overflow answers out there recommend calling a `refresh()` method on a `BluetoothGatt` object to flush the cache of discovered services, but since `refresh()` is not a public method on `BluetoothGatt`, the only way to call it is by using Java reflection. Not only is the actual behavior undocumented and subject to change at any time, but Google has started restricting usages of private APIs since Android P came out last year.

Another method that is often recommended as a solution for BLE bonding issues is a different, private variant of `BluetoothDevice`'s `createBond()` method that accepts an `int transport` argument, where developers then pass in `BluetoothDevice.TRANSPORT_LE` as the value of that argument.

Both `refresh()` and `createBond(int transport)` are currently in Android P's list of greylisted APIs, meaning they can still be used by apps targeting Android P, but in the likely event that Google decides to move them over to the dark greylist for future Android releases, developers will no longer be able to use these APIs if they still want to target the latest release of Android every year.

In the case of `refresh()`, a logical replacement would be to utilize the "Service Changed" indication on the BLE firmware side to let Android know that it should refresh its internal cache of discovered services, and as for `createBond(int transport)`, we're left with no other option except to use the public `createBond()` method where the transport mode is set to `BluetoothDevice.TRANSPORT_AUTO` by default. If you're having issues bonding using the public `createBond()` method, you can also try having encrypted characteristics on your firmware that the app can use to trigger an insufficient authentication error by reading off of it. The read operation would error out, but the result should lead to *most* Android devices automatically starting the bonding process. We say *most*, because, well, in the Android world, hardly anything is guaranteed. This way of triggering BLE bonding also happens to be recommended by Apple while working with iOS devices, so it's likely in your best interest to have both platforms start the bonding process by triggering an insufficient authentication read error.

Android has a lot to learn from iOS

We work on a lot of iOS and Android BLE projects, and the consensus among myself and other mobile team members is that the Android platform has a lot to learn from iOS when it comes to consistency of platform behavior—for starters, performing the same API calls should result in consistent behavior across most, if not all devices. The Android BLE APIs also leave a lot to be desired: on the iOS side, the OS handles queuing of operations automatically, and there are also some really useful features such as Core Bluetooth's background processing and state restoration APIs, as well as a way (since iOS 11) to perform successive writes without response at a high throughput while at the same time having very minimal packet loss.

There are also more weird issues that can sometimes surface on Android, for example, calling `BluetoothGatt`'s `discoverServices()` from a thread that is also used by the OS for delivering `onConnectionStateChange(...)` callbacks can sometimes silently fail, but we hope that by sharing some of the most useful lessons we've learned while developing BLE-capable Android apps that our readers can benefit from them and go on to produce reliable, solid apps.

Chee Yi Ong

Mobile Software Engineer at Punch Through. He specializes in Bluetooth-connected iOS and Android apps. Smart as a whip, talented writer, and teacher. He enjoys gaming, photography, coffee, and traveling.

RECENT POSTS

Meet the Team – Matt Rengo
Erin Moore

Bluetooth Module vs Chip-Down: What is Best for Your Design?
Matt Dunham

Architecting iOS and Android Apps for IoT BLE Systems
Gretchen Walker

Meet the Team – Mike Waddick
Erin Moore

CONTACT

MINNEAPOLIS
123 N. 3rd St,
Suite 500
Minneapolis, MN
55401

info@punchthrough.com