# Mobile Computing Bluetooth Low Energy on Android Smartphones



Slides: tmb.gr/mc-cen

Source on GDocs

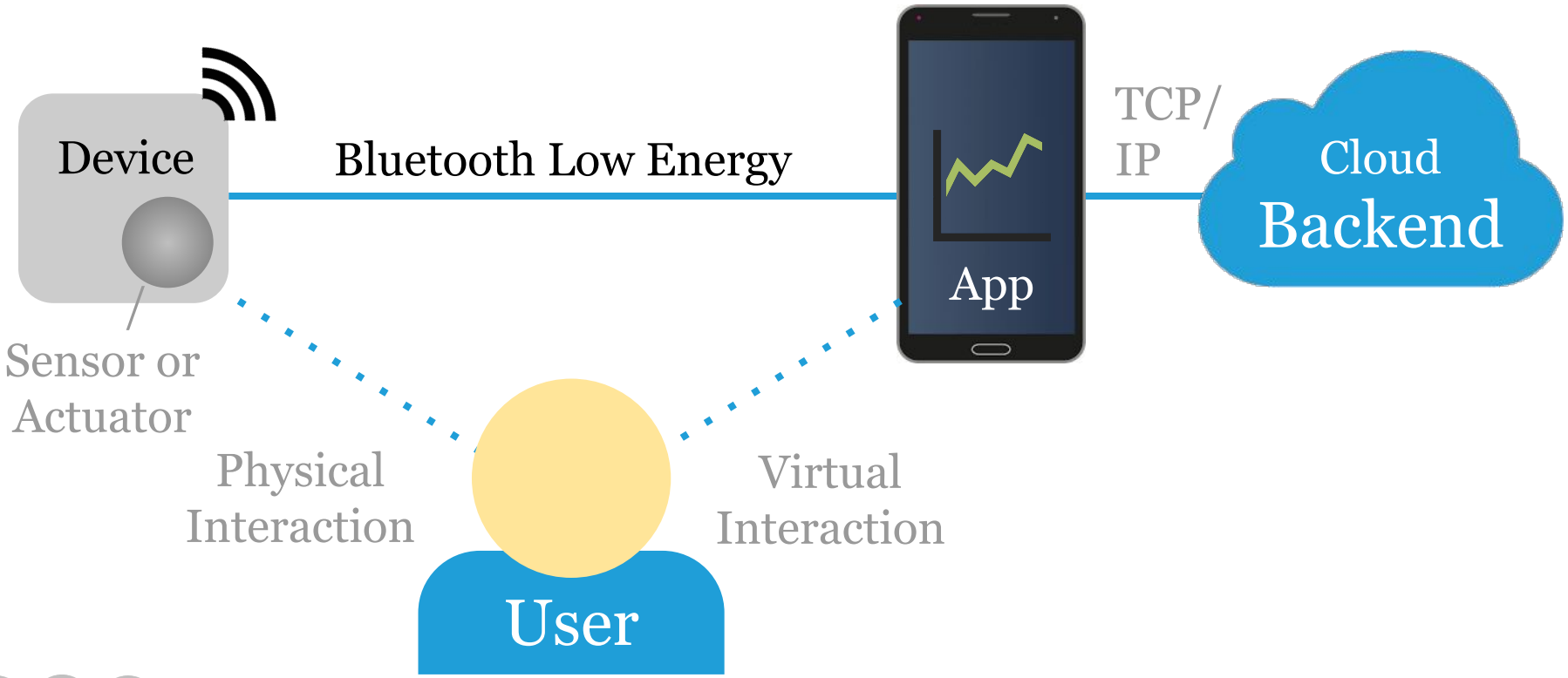
# Overview

These slides introduce *BLE on Android smartphones*.

How to implement the central role, as a GATT client.

Scanning, reading, writing and getting notifications.

# Reference model



# BLE on Android

Android has built-in BLE support* since API level 18.

A smartphone can take the central or peripheral role.

The Android SDKs BLE API is rather tricky to use.

Luckily there are some good 3rd party libraries.

*Also Bluetooth Classic for Headset, A2DP, HDP.

# Adding BLE permissions .xml|.html

Add these permissions to *AndroidManifest.xml*

```
android.permission.BLUETOOTH                    ≤30
android.permission.BLUETOOTH_ADMIN
android.permission.BLUETOOTH_SCAN                31
android.permission.BLUETOOTH_CONNECT
```

If your code just scans, `_CONNECT` is not needed.

# Adding location permissions .xml|.html

Also add these permissions to *AndroidManifest.xml*

```
android.permission.ACCESS_COARSE_LOCATION ≤28
android.permission.ACCESS_FINE_LOCATION   ≤30
android.perm….ACCESS_BACKGROUND_LOCATION  ≤30
```

A scan reveals nearby devices/beacons, often such a "fingerprint" is all it takes to derive a user's location.

Therefore, location permission is required for BLE.

# Checking hardware capability .xml | .kt

Indicate that BLE is required in *AndroidManifest.xml*

```
<uses-feature
android:name="android.hardware.bluetooth_le"
android:required="true"/>
```

And test if the current system has BLE as a feature:

```
private fun hasBle(): Boolean { …
  return app.packageManager.hasSystemFeature(
    PackageManager.FEATURE_BLUETOOTH_LE) }
```

# Checking if Bluetooth is enabled .kt

Check the BluetoothAdapter via BluetoothManager.

```
private fun isBleEnabled(): Boolean { …
  val btManager = app.getSystemService(
    BLUETOOTH_SERVICE) as BluetoothManager
  val btAdapter = btManager.adapter
  return btAdapter != null &&
    btAdapter.isEnabled
}
```

# Asking the user to enable Bluetooth .kt

If Bluetooth is not enabled, ask the user to enable it.

```
fun askToEnableBle() {
  val enableBtIntent = Intent(
    BluetoothAdapter.ACTION_REQUEST_ENABLE)
  startActivityForResultLauncher.launch(
    enableBtIntent)
}
```

# Asking for permissions (at runtime) .kt

```
fun askForPermission() {
  if (Build.VERSION.SDK_INT >=
…VERSION_CODES.S){
      …Launcher.launch(arrayOf(
        Manifest.permission.BLUETOOTH_SCAN,
        Manifest.permission.BLUETOOTH_CONNECT))
  } else if (….VERSION.SDK_INT >= …_CODES.Q) {
    …Launcher.launch(….ACCESS_FINE_LOCATION)
  } else { ….launch(….ACCESS_COARSE_LOCATION }}
```

# Starting a scan

.kt

Get a BluetoothLeScanner to scan for BLE devices:

```
val scanner = btAdapter.bluetoothLeScanner
val handler = Handler(Looper.getMainLooper())
handler.postDelayed({
  scanner.stopScan(scanCallback)
}, SCAN_PERIOD_MS)
scanner.startScan(scanCallback)
```

Note that the Handler runs on another thread.

# Getting scan results

.kt

```
val scanCallback: ScanCallback =
  object : ScanCallback() {
    override fun onScanResult(
      callbackType: Int, result: ScanResult) {
      super.onScanResult(callbackType, result)
      Log.d(TAG, "onScanResult, result = " +
        result.device.address) }
    override fun onScanFailed(errorCode: Int) {
      Log.d(TAG, "onScanFailed, " +
        "errorCode = $errorCode") } }
```

# Hands-on, 10': Android BLE scanner

Build and run the MyBleScannerApp example code.

Check the Logcat output with filter "package:mine".

If it works, you should see BLE devices around you.

Test the app by disabling Bluetooth, location, etc.

Done? Add a button to the UI to start a scan.

# Heart rate service

This service is intended for fitness heart rate sensors:

`Heart Rate Service UUID (16-bit): 0x180D`

This service includes the following characteristics:

```
Heart Rate Measurement      UUID: 0x2A37 [N]
Body Sensor Location        UUID: 0x2A38 [R]
Heart Rate Control Point    UUID: 0x2A39 [W]*
```

Standard service, defined by the Bluetooth SIG.

# Base UUID for registered services

For a custom service UUID we define all 128-bits, e.g.
`0x6E400001-B5A3-F393-E0A9-E50E24DCCA9E`

For well known, registered services the *base UUID* is*
`0x00000000-0000-1000-8000-00805F9B34FB`

A 16-bit service "UUID", e.g. `0x180D` corresponds to:
`0x0000180D-0000-1000-8000-00805F9B34FB`

*See Bluetooth spec v5.3, p.1181 ff.

# Connecting to discover GATT services

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() {
  @Override
  public void onConnectionStateChange(…) { … }
    gatt.discoverServices();
  } …
};
BluetoothGatt gatt = device.connectGatt(
  this, /* reconnect */ true, gattCallback);
```

# Reading a GATT characteristic

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() { …
  @Override
  public void onServicesDiscovered(…) { …
    BluetoothGattCharacteristic characteristic
      = gattService.getCharacteristic(uuid);
    gatt.readCharacteristic(characteristic);
  } …
};
```

# Getting a GATT characteristic value

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() { …
  @Override
  public void onCharacteristicRead(…) { …
    if (characteristic.getUuid().equals(…)) {
      byte[] value = characteristic.getValue();
    }
  } …
};
```

# Writing a GATT characteristic

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() { …
  @Override
  public void onServicesDiscovered(…) { …
    BluetoothGattCharacteristic characteristic
      = gattService.getCharacteristic(uuid);
    characteristic.setValue(value, format, 0);
    gatt.writeCharacteristic(characteristic);
  } … };
```

# Getting GATT characteristic write status

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() { …
  @Override
  public void onCharacteristicWrite(…) { …
    if (characteristic.getUuid().equals(…)) {
      Log.d(TAG, "write, status = " + status);
    }
  } …
};
```

# Enabling GATT char. notifications

```
@Override
public void onServicesDiscovered(…) { …
  gatt.setCharacteristicNotification(…, true);
  UUID configUuid = UUID.fromString("2902");
  BluetoothGattDescriptor descriptor =
    characteristic.getDescriptor(configUuid);
  descriptor.setValue(BluetoothGattDescriptor.
    ENABLE_NOTIFICATION_VALUE);
  gatt.writeDescriptor(descriptor);
}
```

# Getting GATT characteristic notifications

```
final BluetoothGattCallback gattCallback =
  new BluetoothGattCallback() { …
  @Override
  public void onCharacteristicChanged(…) { …
    if (characteristic.getUuid().equals(…)) {
      byte[] value = characteristic.getValue();
    }
  } …
};
```

# Hands-on, 10': Android BLE central

Build and run the MyBleCentralApp example code.

Use the nRF52840 with HRM BLE peripheral code.

Check the Logcat output with filter "package:mine".

If it works, you should see heart rate measurements.

Done? Read the Android app source code in detail.

# What's missing in the example code

BLE related code should be moved into a Service.

Starting a scan should be triggered by the user*.

Discovered devices should be presented in a list.

Read, write, etc. operations should be queued.

*On API level 26+, try companion device pairing.

# Hands-on, 10': Android BLE issues

It looks easy, but BLE on Android has many issues.

Read some of the tips how to make it actually work.

Also check this list of issues to get an impression.

iOS BLE seems more stable. Why could that be?

Done? Find some stats about the Android OS.

# BLE libraries from 3rd parties

Writing robust apps based on the BLE API is hard.

There are a number of 3rd party libraries to fix this:

The Nordic Android BLE library is written in Java.

Nordic also has a Kotlin BLE library for Android*.

RxAndroidBle library uses the RxJava framework.

*Uses Kotlin coroutines.

# Hands-on, 10':  nRF Blinky app example

Android nRF Blinky is a complete BLE app example.

It shows how to use the Nordic Android BLE library.

Read the library docs and study the app source code.

Which parts become easier by using this library?

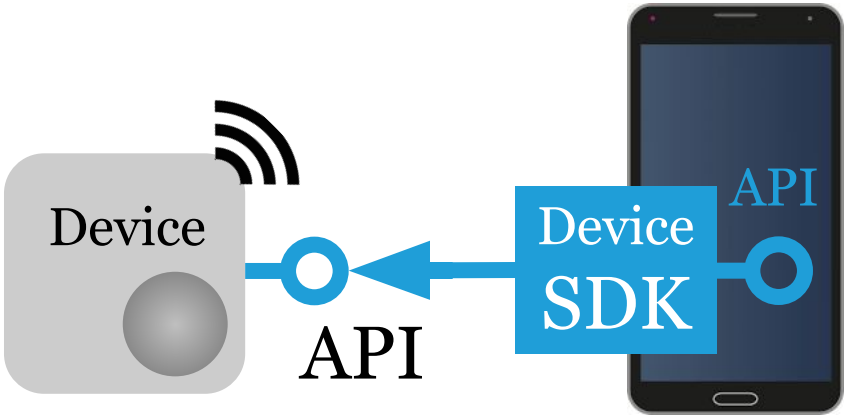
Done? Sketch the BLE API of the peripheral.

# Device API vs. SDK

A *device API* specifies how to talk to the device, from any client (here via BLE).

A platform specific *device SDK* simplifies integration.

E.g. *iOS device SDK* to talk to a device API from iOS.




```
Battery Service
  Battery Level [R]
vs.
p = ble.conn(addr);
b = sdk.getBatt(p);
x = b.getLevel();
```

# Summary

The Android BLE API allows to implement a central.

Scan, read, write and notify operations use callbacks.

Service and characteristic UUID define the interface.

3rd party libraries can help to write robust BLE code.

# Challenge: nRF Toolbox plugin

*nRF Toolbox* is a "container app" for BLE demos.

Build and run the nRF Toolbox app source code.

Write a plugin for a custom SHT30 BLE service.

Test the app with a nRF52840 BLE peripheral.

Done? Create a custom icon for your service.

# Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.