Android Programming

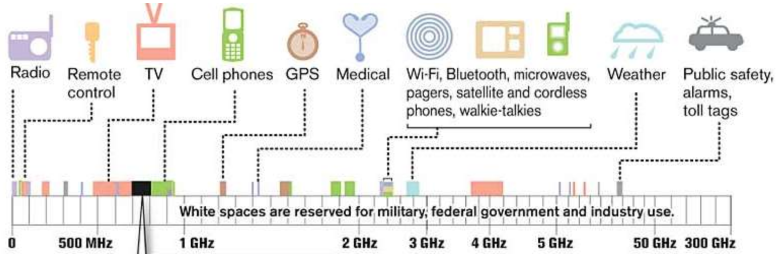
TX00CK66 Sensor Based Mobile Applications

Lecture Bluetooth Beacons

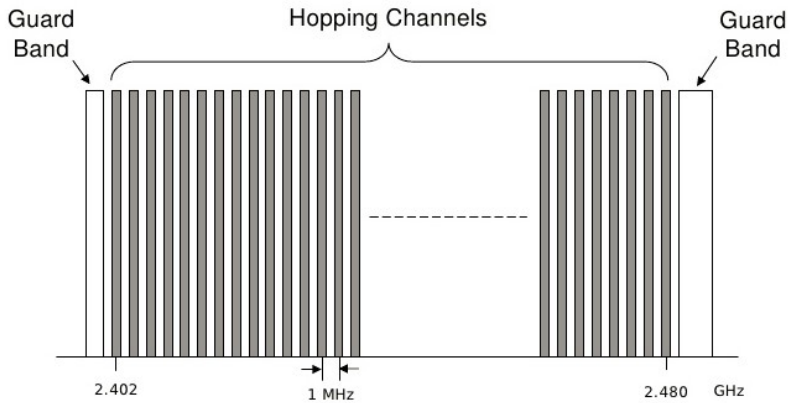
Jarkko.Vuori@metropolia.fi

Bluetooth Tech side

- Universal Wireless Standard
- Small & Cheap Low Power Wireless (<0.1W active power)
  - 2.4 GHz ISM band (Industry, Science, Medical; license free)
  - 79 radio channels of 1 MHz wide (f = 2402 + k MHz, k = 0,…,78) with GFSK modulation
  - Low Energy version (Bluetooth LE, BLE, Bluetooth Smart) uses only 40 channels with GFSK modulation, i.e., 40 channels of 2 MHz wide (f = 2402 + 2k MHz, k = 0,…,39)
    - It has wider bandwidth and higher data rate (in order to reduce transmitter active time, i.e., saving battery)
    - Power consumption is 1/100th of the Bluetooth classic
- Adaptive Frequency Hopping (AFH)
  - Avoid interference in ISM
  - 1600 hops/s (625 μs time slots)
  - Pseudo random sequence based on master BD address
- TDD (Time Division Duplex) for send/receive
- Transmit power 1 – 100 mW
- Data rates 721 kbps – 3 Mbps (BLE: 125kbit/s – 2 Mbit/s)
  - 270 kbps – 2.1 Mbps in real life (BLE: 0.27 – 1.37 Mbit/s)
  - 50 Mbps using WLAN co-operation
- Range: 5m (1 mW transmit power) – 100m (100 mW transmit power)
- Secure (AES encryption)
- BD address: 48 bit IEEE registered
  - Guarantied to be unique



Radio spectrum usage. Higher the frequency, more resemblance to the light.



Bluetooth Tech side

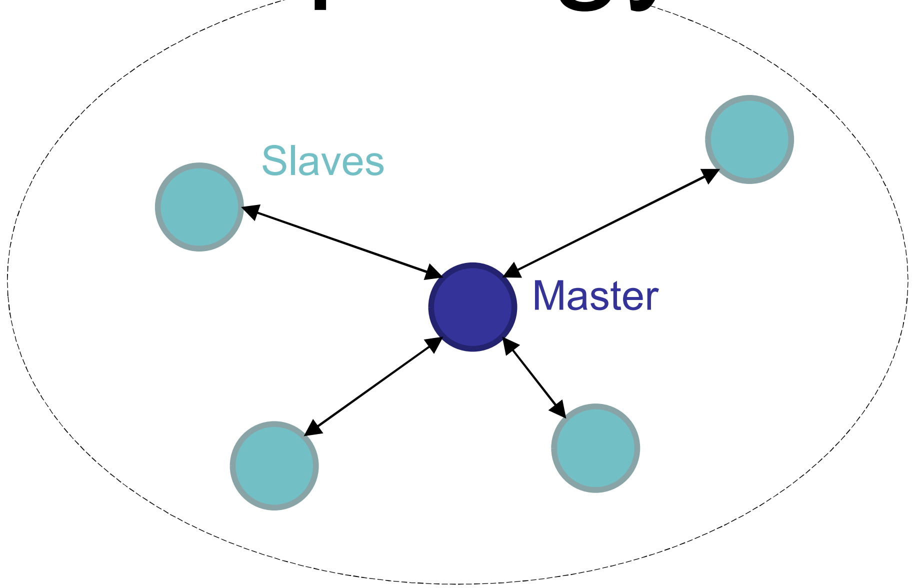
- Native Bluetooth support to Android 2013
- Bluetooth Low Energy (BLE) support at Jelly Bean (4.3) version
  - This is designed to provide significantly lower energy consumption
  - This allows Android apps to communicate with BLE devices that have low power requirements, such as proximity sensors, heart rate monitors, fitness devices, and so on
  - But the energy consumption on the mobile phone side is not so small
- Bluetooth API changed at Lollipop (5.0) version

- Low energy here means that CR2032 battery (≈ 230 mAh, 3V) can power Bluetooth device "few years"

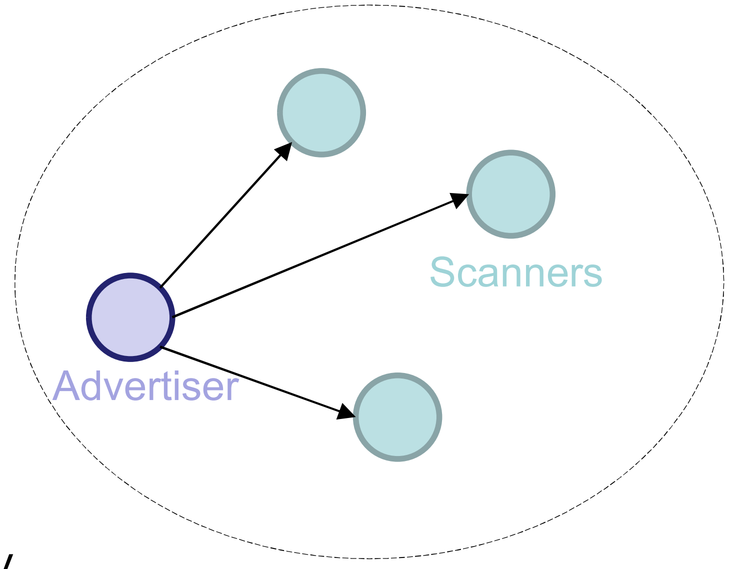
Name	Bluetooth Version	Max Data Rate	Spec
Basic rate (BR)	1.X	721.1 kbps	1999
Enhanced Data Rate (EDR)	2.X	2.1 Mbps	2004
High Speed (HS)	3.X	54 Mbps	2009
Bluetooth Low Energy (BLE)	4.X	1 Mbps	2010
Bluetooth Internet of Things (IoT)	5.X	2 Mbps	2016
Concurrent transactions, improved power control, LE Audio	5.2	2 Mbps	2019

Bluetooth Network Topology

- Piconet: star topology
  - Slaves connect to the master
  - Master coordinates (synchronize) transmissions

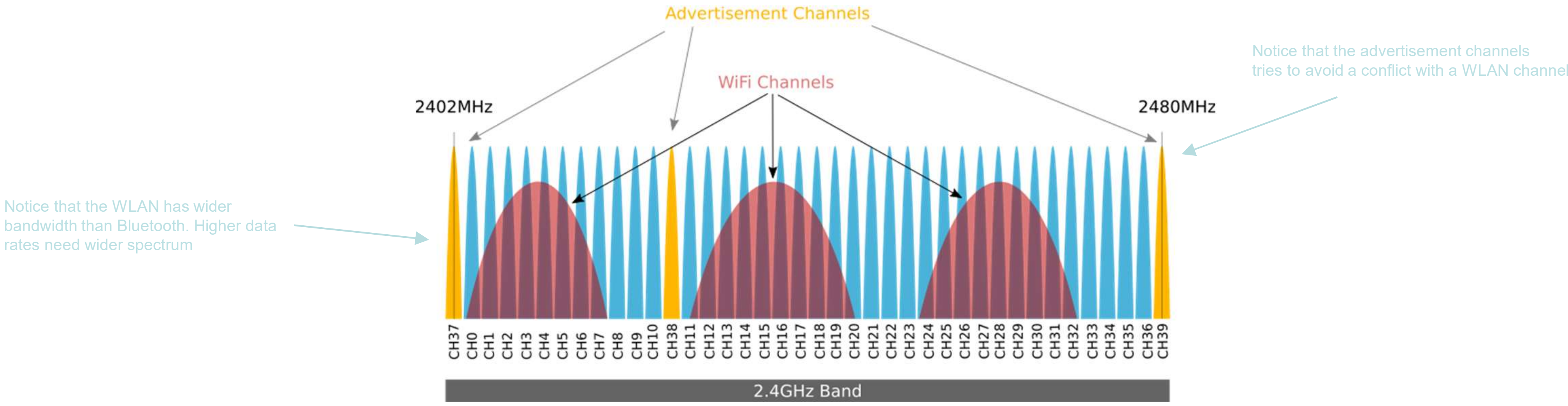


- Broadcast group
  - Advertiser periodically informs services it is giving
  - Scanners search for services



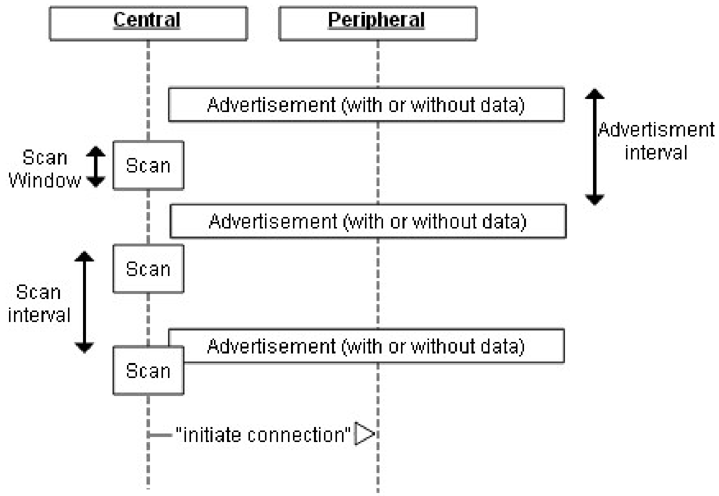
Bluetooth connections, advertiser

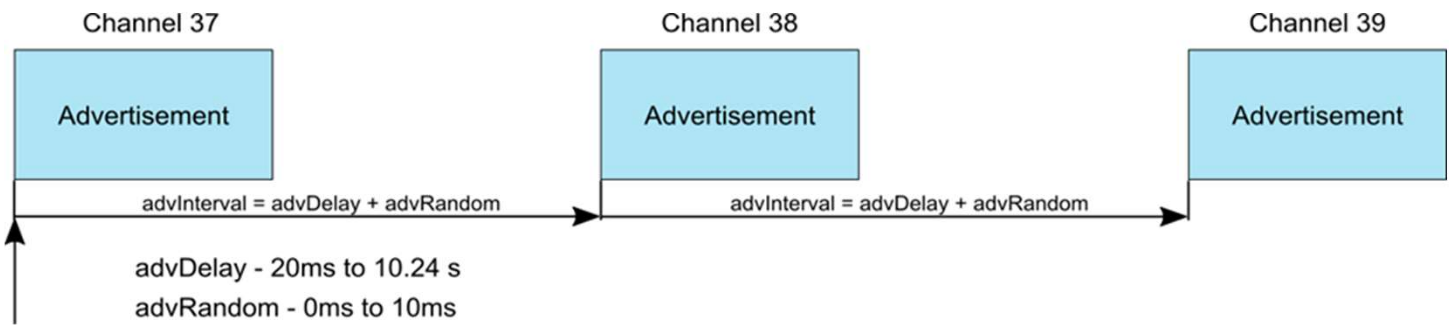
- It is possible for a BLE device to play multiple roles and change its function dynamically
- Most systems will probably be designed explicitly as either **clients** or **servers** of resources (*masters* or *slaves* of the connection); in Bluetooth terminology **peripheral** and **central**
- A device which wants to provide information/services to remote clients will start by first becoming an advertiser
  - In this state, it will periodically broadcast a message to indicate its existence to any potential masters in the area
  - This advertisement is a little packet that contains the address of the slave and up to 31 bytes of extra data called as Scan Response Data
  - This device retains the role of advertiser until someone comes along to connect



Bluetooth connections, scanner

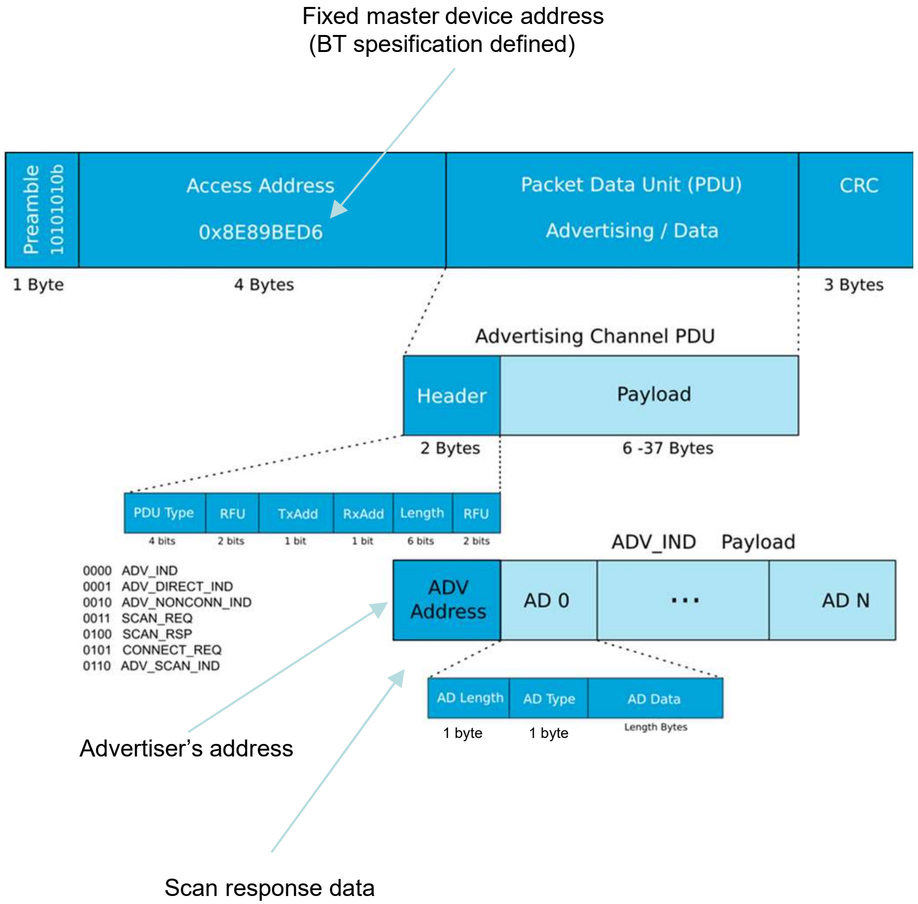
- When in the mood for contact, the eventual master gets into the character of a scanner
  - In this role, it listens on (one or more of) the advertisement channels for broadcasts from potential friends
    - There are three fixed broadcast channels in BLE (the remaining 37 are for the data communication in AFH way)
    - Because the master can listen only one channel at a time and slave is able to transmit only on one channel, it takes time for a master to detect the slave
  - The time interval between packets has both a fixed interval and a random delay
    - The fixed interval can be from 20ms to 10.24 seconds, in steps of 0.625ms
    - The random delay is a pseudo-random value from 0ms to 10ms that is automatically added
      - This randomness helps reduce the possibility of collisions between advertisements of different devices





The BLE Advertising Packets

- The Packet Data Unit (PDU) for the advertising channel (called the Advertising Channel PDU) includes a 2-byte header and a variable payload from 6 to 37 bytes
  - The actual length of the payload is defined by the 6-bit Length field in the header of the Advertising Channel PDU
- There are several PDU types for the advertisements
  - ADV_IND is a generic advertisement and usually the most common
  - It's generic in that it is not directed, and it is connectable, meaning that a central device can connect to the peripheral that is advertising, and it is not directed towards a particular Central device
- When a peripheral device sends an ADV_IND advertisements, it is helping Central devices such as Smartphones to find it
  - Once found, a Central device can begin the connection process
- ADV_NONCONN_IND is the advertisement type used when the peripheral does not want to accept connections, which is typical in Beacons



Scan Responce Data

- Scan Responce Data (PDU Type: SCAN_RSP) is sent periodically (about once in a second or less) in BLE advertising packets payload
- Size is 31 bytes
- It is divided into what are called AD structures
  - They are sequences of bytes of various size
    - The first byte represent the number of bytes left to the end of the AD structure
    - The second byte is the ID of an AD structure type (defined by the Bluetooth reference)
    - The rest of the bytes are data that is structured in a predefined way depending on what AD type was defined by the previous byte

AD Length	AD type	Data

- Most beacon protocols, if not all, have only 2 AD structures

Scan Responce Data

- Nearly every Beacon has a Flag AD
  - The first byte will be: 0x02 because we only count the following bytes.
  - The second byte is: 0x01 which indicates we have a "Flag" AD type.
  - The last byte represents theses flags. These flags express whether the emitting device is, in "Limited Discoverable Mode", "General Discoverable Mode", etc… The byte is computed on the following way:
- The 5 flags are represented by the first 5 bits of a byte. The value of these bits defines whether the flag is ON or OFF. The binary number is then written as a hexadecimal value which will be advertised
  - An example on the right clears things up

bit 0	OFF	LE Limited Discoverable Mode
bit 1	ON	LE General Discoverable Mode
bit 2	OFF	BR/EDR Supported
bit 3	ON	Simultaneous LE and BR/EDR to same Device Capable (controller)
bit 4	ON	Simultaneous LE and BR/EDR to same Device Capable (host)

The resulting binary value hence becomes: b00011010.
Converted into a hex, we get: 0x1A

Scan Response Data																	
AD Structure 1			AD Structure 2														
0x02	0x01	0x1A	0x1B	0xFF	0xE0	0x00	0xBE	0xAC	0x0C	[...]	0xBB	0x00	0x09	0x00	0x06	0xBA	0x00
Remaining length	AD type	Data	Remaining length	AD Type	Manufacturer ID		Beacon Prefix		UUID (16 bytes)			Major		Minor		TX Power	
2	Flags	Flag	27	Manufacturer Specific	224: Google							9		6		-70	

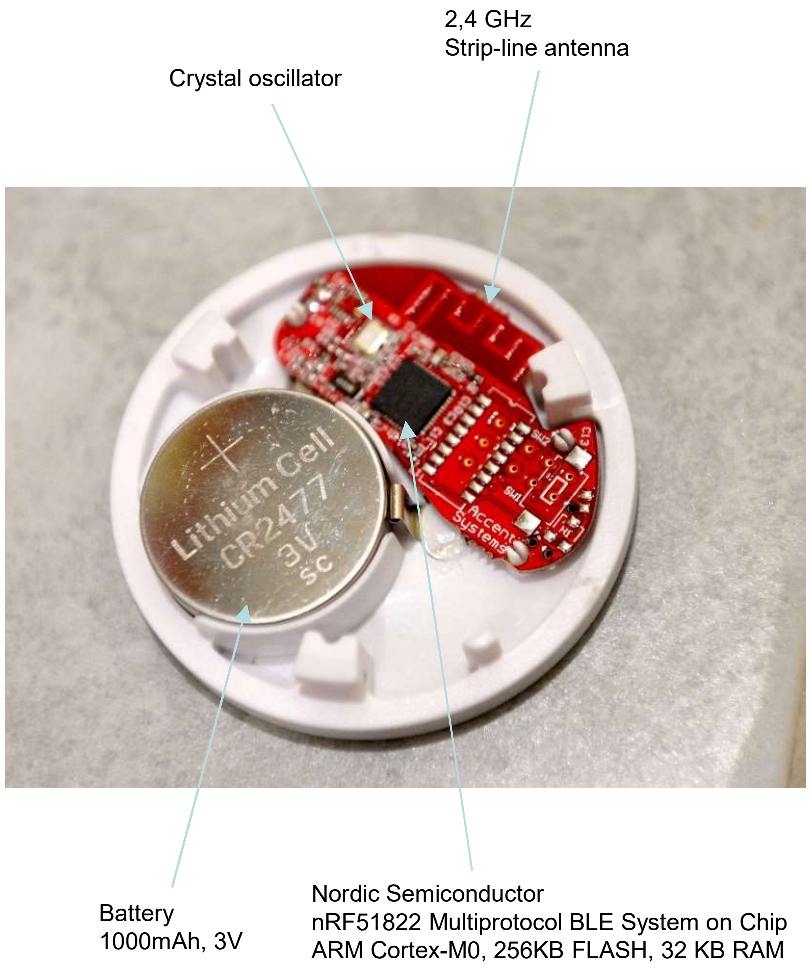
Scan Responce Data

- The second structure can be of different size according to the protocol
- First byte is 0x1B (27 in hexadecimal) which means we are taking all of the last available byte of our 31 byte scan response
  - This can vary according to protocols
- The next byte is always 0xFF which means we have a “Manufacturer Specific” type of AD structure
- As a result, the 2 following bytes represent the company identifier as defined on bluetooth.org
  - Company here means the Bluetooth chip manufacturer, we’ll simplify this by using Google’s manufacturer ID which is 224
    - In hexadecimal value, this is equal 0x00E0. The ID, written as little endian takes up the 2 bytes. Here it will be 0x0E0 0x00 in this order
- The rest is Manufacturer specific data! This is what changes the most between protocols

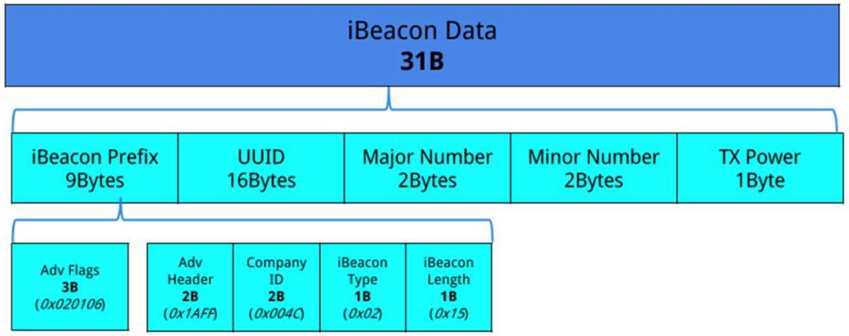
Scan Response Data																	
AD Structure 1			AD Structure 2														
0x02	0x01	0x1A	0x1B	0xFF	0xE0	0x00	0xBE	0xAC	0x0C	[...]	0xBB	0x00	0x09	0x00	0x06	0xBA	0x00
Remaining length	AD type	Data	Remaining length	AD Type	Manufacturer ID		Beacon Prefix		UUID (16 bytes)			Major		Minor		TX Power	
2	Flags	Flag	27	Manufacturer Specific	224: Google							9		6		-70	

Beacon Tech side

- Beacons are Bluetooth LE-transmitters that send some information or advertising data that can be accepted by the smartphones and tablets within the range of the transmitter
  - i.e., their task is simple—serially send data packages (advertisement packets)
- There are different types of Beacons
  - iBeacon
    - Developed by Apple in 2013
  - Eddystone
    - Developed by Google in 2015
  - Accent Systems
    - Developed by Accent Systems 2015
    - Capable of sending both protocol beacons (at different times)
    - Capable of doing measurements and sending them in an advertisement
  - Etc. (e.g., Ruuvi tag, https://ruuvi.com/)



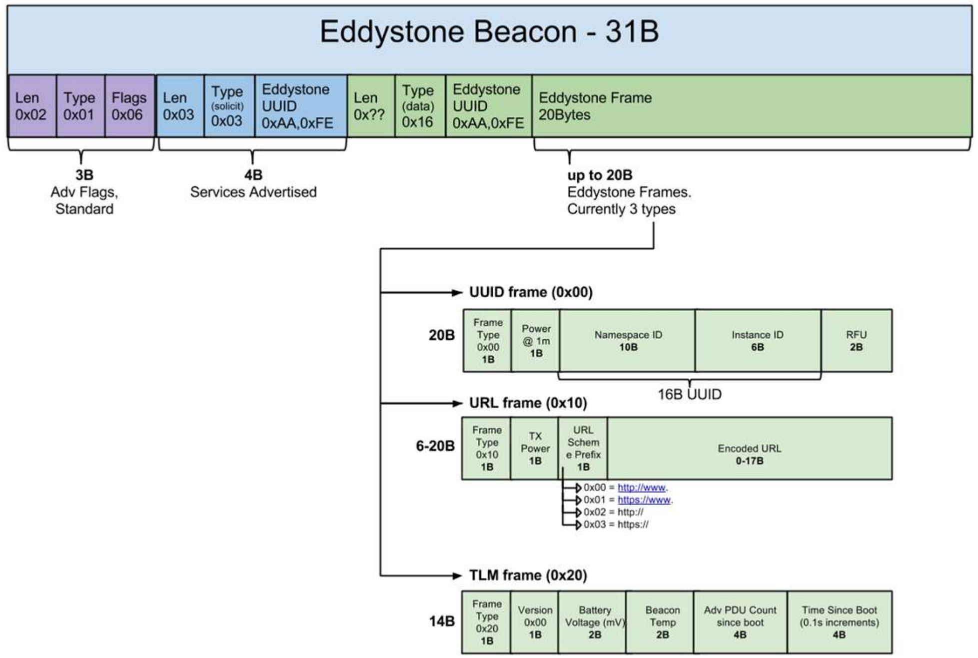
iBeacon



- iBeacons broadcast four pieces of information:
  1. A UUID that identifies the beacon
  2. A Major number identifying a subset of beacons within a large group
  3. A Minor number identifying a specific beacon
  4. A TX power level in 2's complement number, indicating the signal strength one meter from the device
     - This number must be calibrated for each device by the user or manufacturer.
- A scanning application reads the UUID, major number and minor number and references them against a database to get information about the beacon; the beacon itself carries no descriptive information - it requires this external database to be useful
  - The TX power field is used with the measured signal strength to determine how far away the beacon is from the smart phone
  - TxPower must be calibrated on a beacon-by-beacon basis by the user to be accurate

EddyStone

- The Eddystone Frames get swapped in and out depending on what frames you have enabled
- The only required frame type is the TLM frame, all others are optional and you can have any number enabled
  - To disable the UID or URL frames you must configure the Beacon using a special software
- The Eddystone spec recommends broadcasting 10 frames a second



UUID

- A universally unique identifier (UUID) is a 128-bit number used to identify information in computer systems (in unique way)
  - There are $2^{128} = 3{,}4 \cdot 10^{38}$ different UUIDs (there are $\approx 7{,}97 \cdot 10^{9}$ humans on the earth, so every human being can have $4{,}3 \cdot 10^{28}$ different UUIDs)
- Bluetooth Services, Characteristics and other items use UUID to uniquely identify them
- UUIDs are nothing more than unique 128-bit (16 byte) numbers:

  75BEB663-74FC-4871-9737-AD184157450E
  - It's typical to arrange the UUID in the format above 4-2-2-2-6 (where the number means how many bytes belong to that field, 16 together, 16×8=128)
  - Each pair of characters actually indicate a hexadecimal number that fits into one byte
    - Thus, 75 above is actually 0x75
- To avoid constantly transmitting 16 bytes which can be wasteful (Bluetooth is very limited in the amount of data and 16 bytes are significant), the Bluetooth SIG has adopted a standard UUID base
  - This base forms the last 96 bits (12 bytes) of the 128-bit UUID. The rest of the bits are defined by the Bluetooth SIG:

    XXXXXXXX-0000-1000-8000-00805F9B34FB
  - The top 32-bits are up to you. For 16-bit UUIDs, the highest 16-bits remain 0. For example, the short form 16-bit UUID for the Heart Rate Service is:

    0x180D
  - In reality this represents a 128-bit UUID:

    0000180D-0000-1000-8000-00805F9B34FB
- If you're using existing services or profiles that were specified by the Bluetooth SIG, you can avoid using the full 128-bit UUID
  - Custom services need a fully defined 128-bit UUID
- UUIDs must be unique
  - Use can use https://www.uuidgenerator.net/ to generate a random UUID which is unique in very high probability (because there are so many different UUIDs available)

Android Bluetooth

- Android provides Bluetooth API to perform the following different operations
  1. Scan for other Bluetooth devices
     - `CompanionDeviceManager` API mainly for Bluetooth devices that will pair up with the app
     - `BluetoothLEScanner` API for scanning Beacons
  2. Get a list of paired devices
  3. Connect to other devices through service discovery
- Android provides `BluetoothAdapter` class to communicate with Bluetooth
- An object of this class can be created by calling the static method

```
class MainActivity : ComponentActivity() {
    private var mBluetoothAdapter: BluetoothAdapter? = null

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

        val bluetoothManager = getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
        mBluetoothAdapter = bluetoothManager.adapter
```

Permissions

- You need to give static permission to the Android application to use the Bluetooth interface (see https://developer.android.com/guide/topics/connectivity/bluetooth/permissions)
  - In order to allow an Android device to search nearby Bluetooth devices, manifest must contain right to allow give location information to outside
    - Because searching other Bluetooth devices reveals the coarse location of the Android device

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

- It is also mandatory to setup runtime permissions by the user (asked only at the first time you run the application)

```
private fun hasPermissions(): Boolean {
    if (mBluetoothAdapter == null || !mBluetoothAdapter!!.isEnabled) {
        Log.d("DBG", "No Bluetooth LE capability")
        return false
    } else if (checkSelfPermission(Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        Log.d("DBG", "No fine location access")
        requestPermissions(arrayOf(Manifest.permission.ACCESS_FINE_LOCATION), 1);
        return true // assuming that the user grants permission
    }

    return true
}
```

Finding BLE Devices

```
private val mResults = java.util.HashMap<String, ScanResult>()
fun scanDevices(scanner: BluetoothLeScanner) {
    viewModelScope.launch(Dispatchers.IO) {
        fScanning.postValue(true)
        val settings = ScanSettings.Builder()
            .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
            .setReportDelay(0)
            .build()
        scanner.startScan(null, settings, leScanCallback)
        delay(SCAN_PERIOD)
        scanner.stopScan(leScanCallback)
        scanResults.postValue(mResults.values.toList())
        fScanning.postValue(false)
    }
}
```

We are running this under ViewModel

A callback method

We need this because Bluetooth devices transmit asyncronously to our program execution

```
companion object GattAttributes {
        const val SCAN_PERIOD: Long = 5000

        const val STATE_DISCONNECTED = 0
        const val STATE_CONNECTING = 1
        const val STATE_CONNECTED = 2

        val UUID_HEART_RATE_MEASUREMENT       = UUID.fromString("00002a37-0000-1000-8000-00805f9b34fb")
        val UUID_HEART_RATE_SERVICE           = UUID.fromString("0000180d-0000-1000-8000-00805f9b34fb")
        val UUID_CLIENT_CHARACTERISTIC_CONFIG = UUID.fromString("00002902-0000-1000-8000-00805f9b34fb")
    }
```

Good way to have class-wide constants

Callback function

- This callback is called if the scan finds a BLE device
  - If you don’t have given right permissions, scanning does not call the callback function
  - ScanCallback may give the same device multiple times (every time it receives the advertisement)
    - HashMap may be used to remove duplicates

```
private val leScanCallback: ScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)
        val device = result.device
        val deviceAddress = device.address
        mResults!![deviceAddress] = result

        Log.d("DBG", "Device address: $deviceAddress (${result.isConnectable})")
    }
}
```

Analyzing ScanResult

- Bluetooth LE Scan makes a callback method call when an advertisement is heard
  - This call has an argument `ScanResult`
  - From the `ScanResult`, using `getDevice()` method it is possible to obtain `BluetoothDevice` object
    - This can be using, e.g., to get the Bluetooth address, or to make a connection to the device (if possible)
      - After you have found BluetoothDevice (from the scan), it is possible to connect to the GATT server on the device `mBluetoothGatt = device.connectGatt(this, false, mGattCallback);`
      - Then the GattCallback is being called when services are found (on the Bluetooth device)
      - Services are represented as `BluetoothGattService` object
  - From the ScanResult, using getScanRecord() method it is possible to
    - Get the Bluetooth Device Name, `getDeviceName()`
    - Get the AD flags, `getAdvertiseFlags()`
    - Get all the raw data (if you want to decode all ADs by yourself), `getBytes()`
    - Get possible service UUIDs, getServiceUuids()
      - EddyStone UUID is 0xFEAA
      - Matching service data is then found by getServiceData(ParcelUuid uuid)
        - » UUID, URL or TLM frame

How to connect Composable

- You can use LiveData to transfer information from the ViewModel to the Composable function

```
class MyViewModel : ViewModel() {
    val scanResults = MutableLiveData<List<ScanResult>>(null)
    val fScanning = MutableLiveData<Boolean>(false)
    private val mResults = java.util.HashMap<String, ScanResult>()

    fun scanDevices(scanner: BluetoothLeScanner) {
            .
            .
            .
        scanResults.postValue(mResults.values.toList())
    }

    private val leScanCallback: ScanCallback = object : ScanCallback() { .. }
            .
            .
            .
}


fun ShowDevices(mBluetoothAdapter: BluetoothAdapter, model: MyViewModel = viewModel()) {
    val context = LocalContext.current
    val value: List<ScanResult>? by model.scanResults.observeAsState(null)
    val fScanning: Boolean by model.fScanning.observeAsState(false)

    Column { .. }
            .
            .
            .
}
```

Reading list

- Bluetooth technical specification
  - http://www.bluetooth.org
- Good explanation for Bluetooth broadcast advertisement
  - https://www.novelbits.io/bluetooth-low-energy-advertisements-part-1/
- Bluetooth in Android
  - https://www.tutorialspoint.com/android/android_bluetooth.htm
  - https://developer.android.com/guide/topics/connectivity/bluetooth/ble-overview
  - https://punchthrough.com/android-ble-guide/
- Eddystone beacon
  - https://github.com/google/eddystone