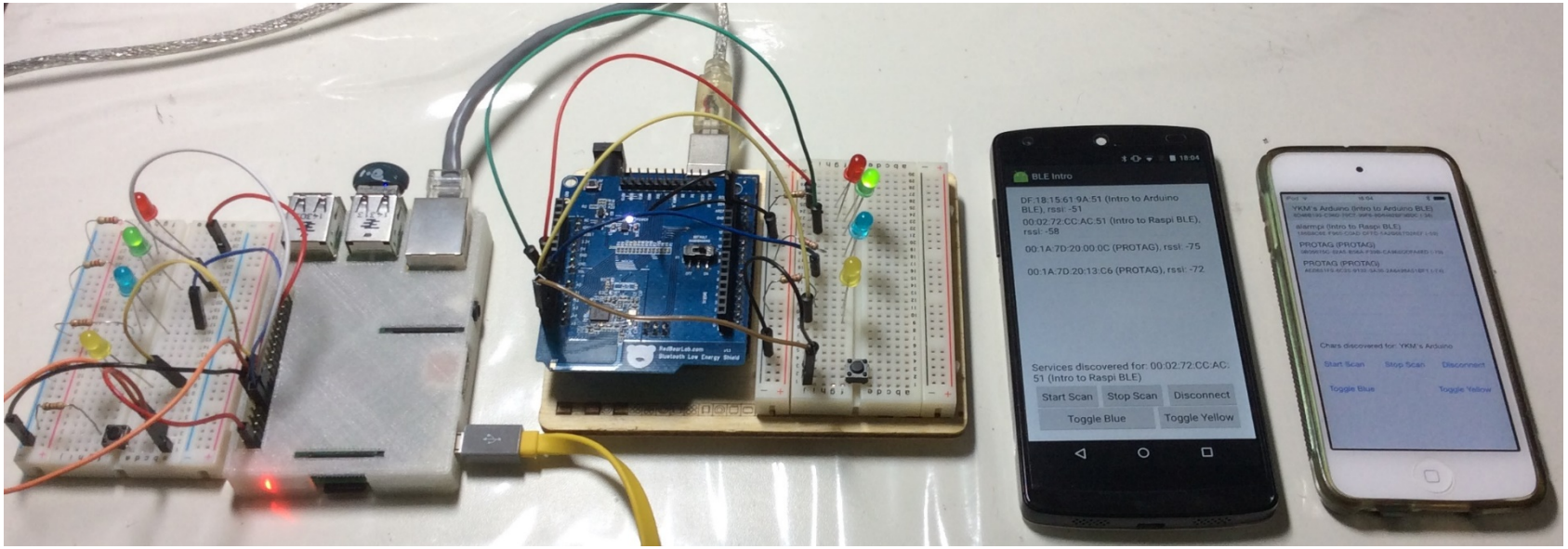
Introduction to Bluetooth Low Energy (BLE)

with



Tech Talk Tuesdays @OMG (16 Feb 2016)
Friday Hacks #98 @NUS Hackers (2 Oct 2015)
Hackware v0.8 (9 June 2015)
Hackware v0.7 (13 May 2015)
Hackers and Painters (10 April 2015)

By: Yeo Kheng Meng (yeokm1@gmail.com)
https://github.com/yeokm1/intro-to-ble

About Me

- Graduated from NUS Computer Science in 2015
- Worked in 2 startups so far
  - Both BLE-related

Where I started from?

- Innova Technology
  - Makes anti-loss BLE tags with companion phone app
  - “Protags”
- Android Dev
- 2013 – 2014
  - Era before Android officially supported BLE
  - Fragmentation like you have never seen

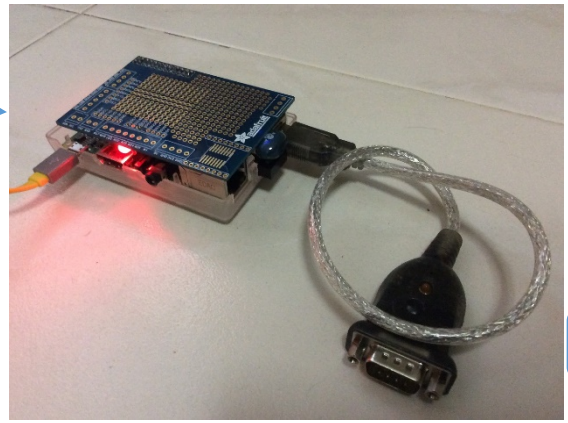
Where I am now?

algoaccess

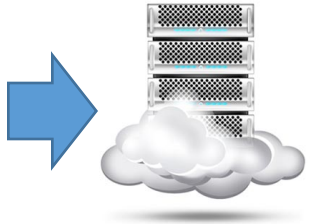
- Algoaccess
  - Med-tech startup: targeting at eye-professionals
  - Help them to retrieve, manage and process the data
  - Roles: many….
  - 2014 - present

How things work?

OD, OS, CYL…



OD, OS, CYL…



Intro: Bluetooth Classic

- The “conventional” Bluetooth
- 2.4GHz
- Range: 1m - 100m (10m typical)
- Connection-oriented: audio, file transfer, networking
- Reasonably fast data rate: 2.1 Mbps
- Power consumption:
  - High but still < Wifi < 3G

Intro: BLE

- Introduced in Bluetooth 4.0 specification (2010)
- Also known as
  - Bluetooth SMART
  - Single-Mode
  - Dual-Mode = Classic + Single-Mode
- Target applications
  - Wireless battery-powered sensors eg. heart rate, thermometer, fitness
  - Location tracking and information serving eg. iBeacons
- Requirements for target applications
  - Low-power
  - Low-cost
  - Low bandwidth: ~100 kbps
  - Low latency: Connectionless (fast setup and teardown of connection in ~10ms)
- How?
  - Radio chip off most of the time
  - Small packets
    - MTU: 20 bytes/packet for application
    - Less time transmitting -> less heat -> no need compensatory circuits -> save more power

Bluetooth Classic vs SMART

- An actual battery-life comparison
- Innova's anti-loss products

vs

Protag G1 (Classic)
Released: 2012
Battery Capacity: 3.7V, 270mAh
Battery Life: 1 - 2 weeks

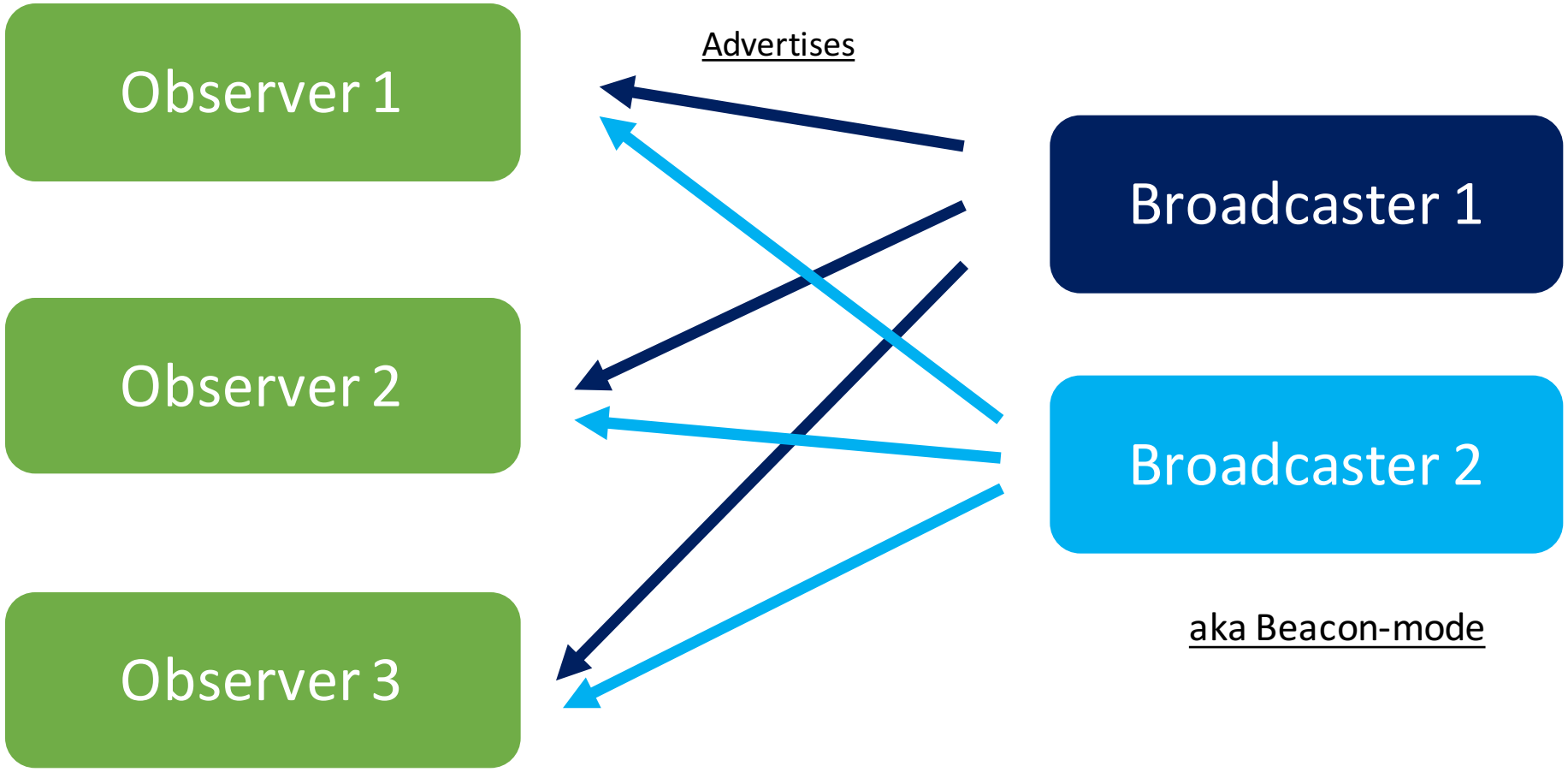
Protag Elite (SMART)
Released: 2013
Battery Capacity: 3.7V, 150mAh
Battery Life: 6 months to 1 year

What's on the agenda?

1) **BLE theoretical concepts***
   a. Device Role 1: Broadcaster vs Observer
   b. Device Role 2: Central vs Peripheral
   c. OS/Device Compatibility
   d. UUID, Attribute, GAP, GATT, Service, Characteristic, Descriptor
   e. BLE connection procedure
2) **Peripheral hardware design and software planning**
   a. Functional requirements
   b. Hardware setup
   c. Peripheral architecture plan
3) **Execution**
   a. Arduino (C)
   b. Central architecture plan (iOS and Android)
   c. iOS (Swift)
   d. Raspberry Pi (JavaScript)
   e. Android (Java)
4) **Issues and tips**
   a. General issues
   b. iOS
   c. Android (past, today, production app tips)
5) **BLE layer model and packet concepts**
6) **BLE Sniffer**
7) **Further reading**
8) **Extra questions**

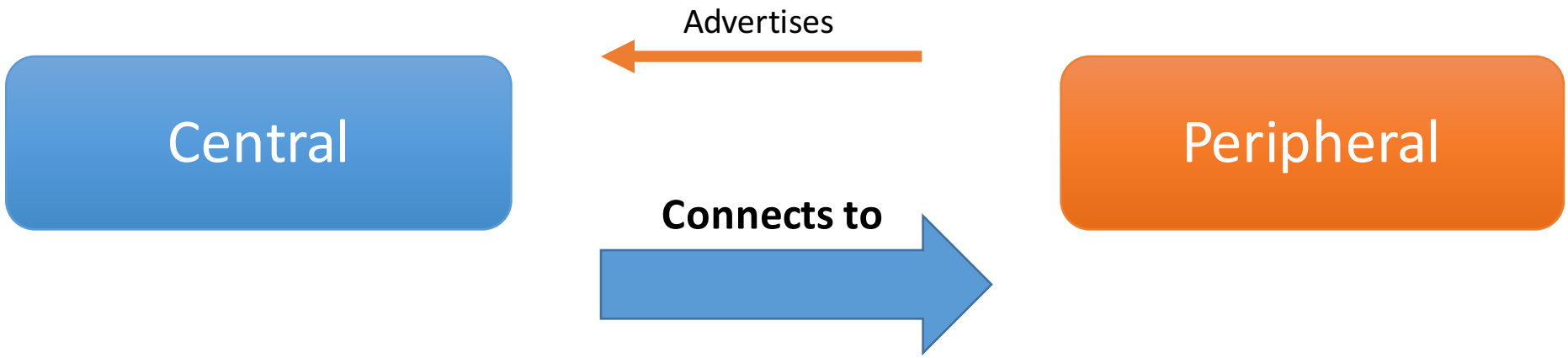
* Exact definitions are not used to aid ease of explanation

1a. Device Role 1: Broadcaster vs Observer



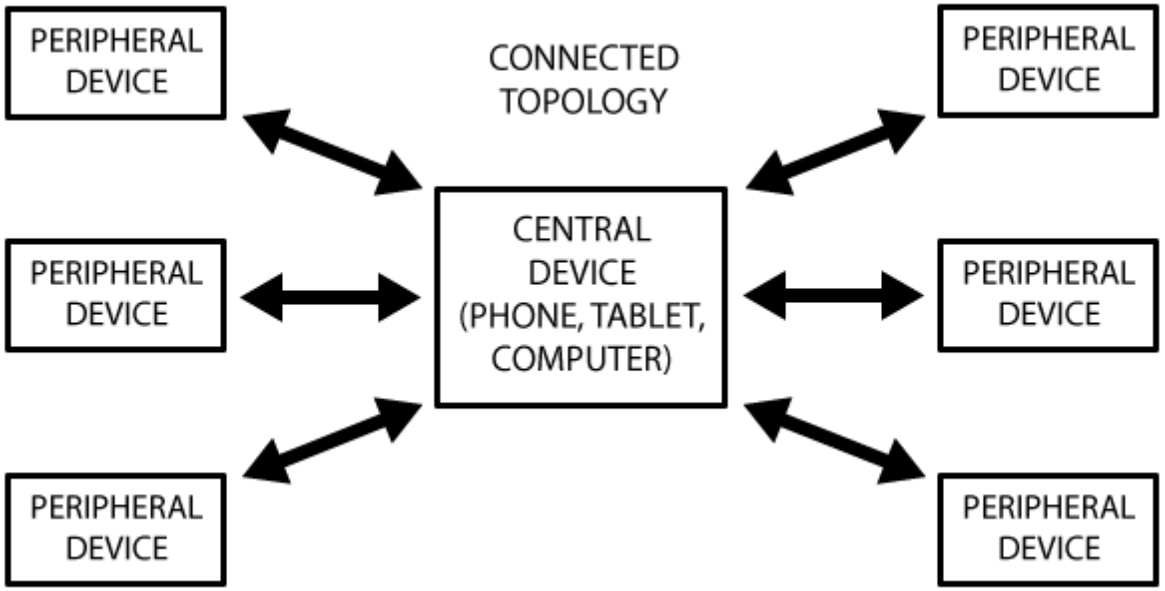
One-way advertisement information transfer from broadcaster to observer(s)

1b. Device Role 2: Central vs Peripheral



Platform	Terms they prefer (generally mean the same thing)
iOS	Central/Peripheral
Android	Client/Server
Chipset manufacturers	Master/Slave

1b. Device Role 2: Central vs Peripheral



Source: https://learn.adafruit.com/assets/13826

Central can connect to **many peripherals** at the same time
Peripheral can connect to **only one central** at any one time.

1c. OS/Device Compatibility

Observer/Central	Broadcaster/Peripheral
1. iOS 5	1. iOS 6
2. Windows 8	2. Windows 10
3. Mac OS X 10.7 (Lion)	3. Mac OS X 10.9 (Mavericks)
4. Linux kernel 3.5	4. Linux kernel 3.5
• Bluez 5.0	• Bluez 5.0
5. Android 4.3 (Jelly Bean MR2)	5. Android 5.0 (Lollipop)
• X - Nexus 7 (2012), 10*	• X - Nexus 4, 5, 7 (2013)*
• X - Galaxy Nexus*	• ✔ - Nexus 6, 9, 5X, 6P
6. Hardware chipsets	6. Hardware chipsets
• CC2540	• CC2540
• CSR1010	• CSR1010
• NRF51/52	• NRF51/52
	• NRF8001

*Hardware capable but not certified by Bluetooth SIG-> disabled in OS , custom ROMs may enable these BLE features

1d. UUID, Attribute

- Universally Unique Identifier (UUID)
  - 128-bit eg. “12345678-ABCD-EF90-1234-00805F9B34FB”
  - To ensure practical uniqueness if randomised
  - $2^{128} = 3.4 \times 10^{38}$
  - 16-bit for Bluetooth Special Interest Group (SIG) defined services/characteristics/descriptors
    - Combined inside Bluetooth Base UUID
    - 0000xxxx-0000-1000-8000-00805F9B34FB
- Attribute
  - Anything that has a UUID
  - Refers to Services, Characteristics and Descriptors

1d. GAP, GATT (defined by Peripheral)

- Generic Access Profile (GAP) or Advertising
  - Information advertised to central before connection
  - Name of peripheral
  - Is it connectable?
  - Supported features (services)
- Generic Attribute Profile (GATT)
  - How to exchange data once connected
  - Identifies Services, Characteristics and Descriptors

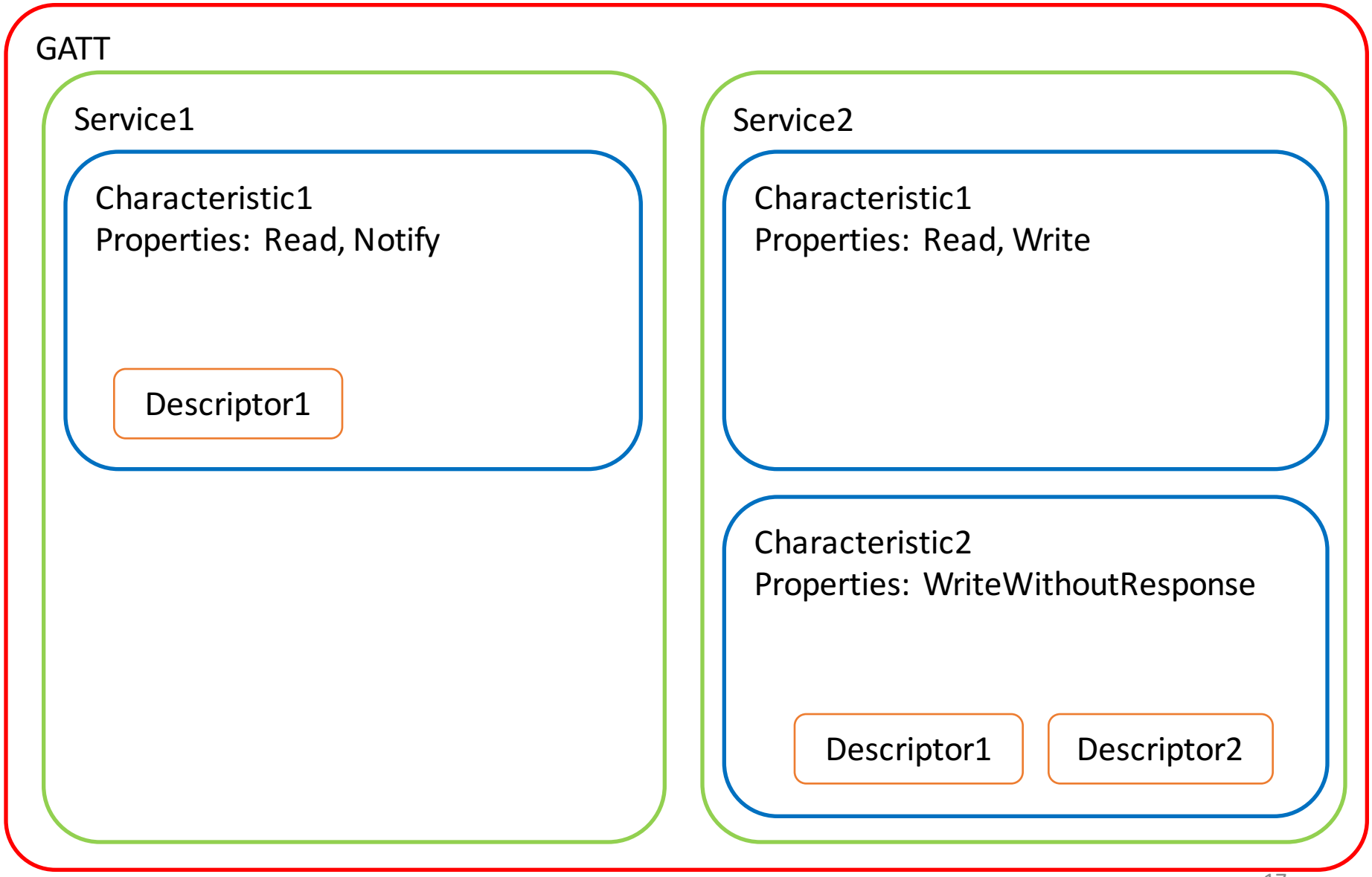
*Both GAP and GATT are theoretical concepts, you don't usually see those terms in coding APIs.

1d. Service, characteristic, descriptor

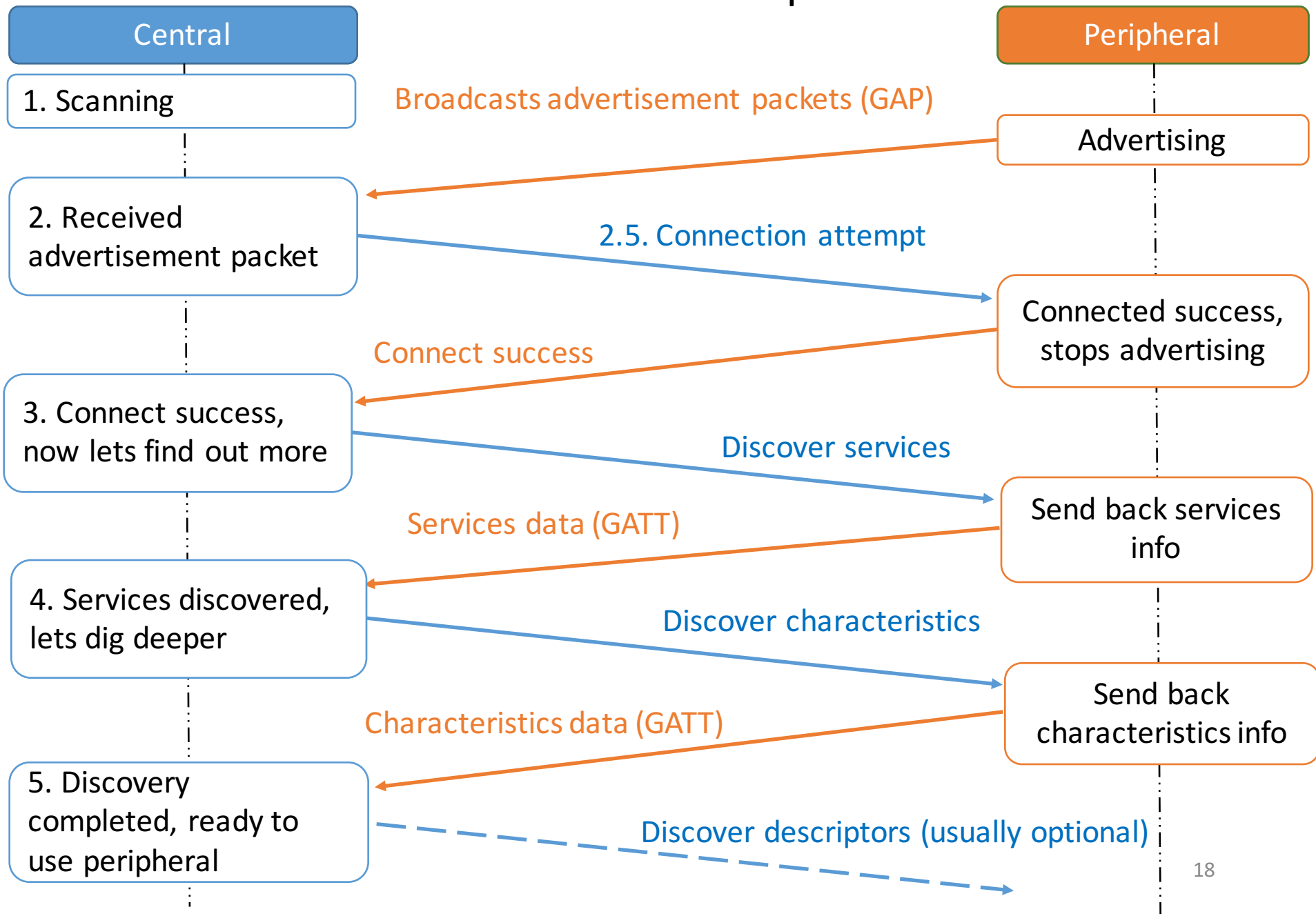
(All these are part of a peripheral's GATT)

- **Service**
  - 16-bit SIG services: Battery, Heart rate, Immediate Alert, Tx Power
  - 128-bit UUID for custom services
  - Collection of characteristics
- **Characteristic**
  - Holds a value: String, Int, Char…..
  - Can take on multiple properties:
    - Read: Central can read this value directly
    - Write: Central can write/change this value and be notified if executed successfully
    - WriteWithoutResponse: Central just "fire and forget"
    - Notify: Central gets alerted if the value has changed
    - Others: Broadcast, Indicate, SignedWrite, QueuedWrite, WritableAuxiliaries
  - Collection of optional descriptors
- **Descriptor:** usually optional
  - Holds a value
  - Used to describe a characteristic (meta-data)
  - Special case: Client Characteristic Configuration Descriptor (0x2902)
    - Usually automatically created for characteristics with "notify" property

1d. Service, characteristic, descriptor



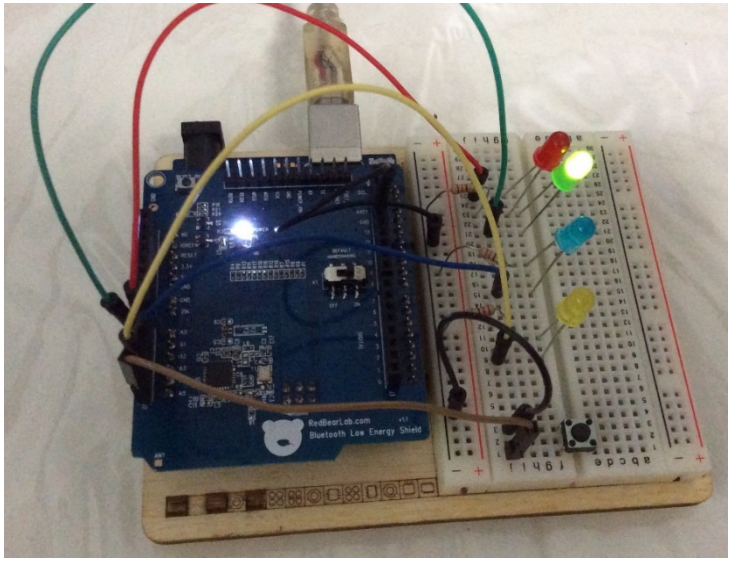
1e. BLE connection procedure

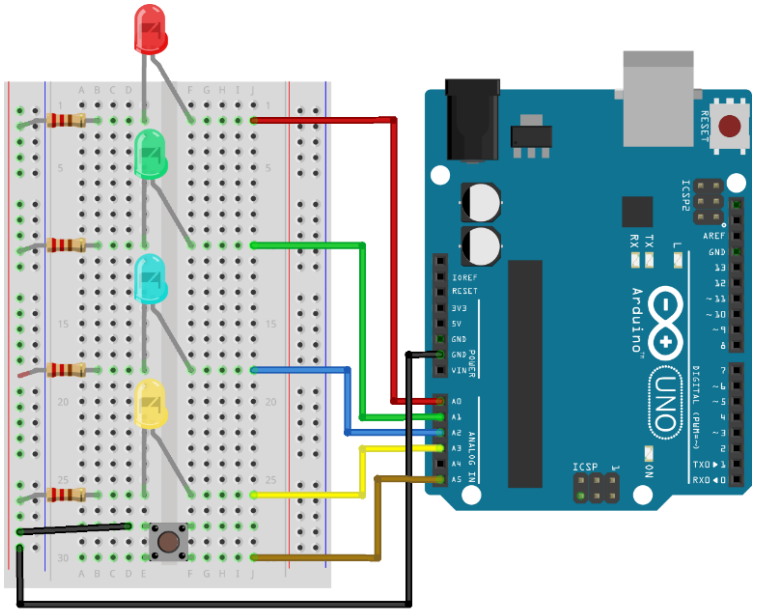


2a. Functional Requirements

- Connection Status
  - Red LED to indicate no connection
  - Green LED to indicate active connection with central
- Controllable via BLE
  - Let central toggle blue LED
  - Let central toggle yellow LED
  - Button to trigger sending data back to central

2b. Hardware setup (Arduino)

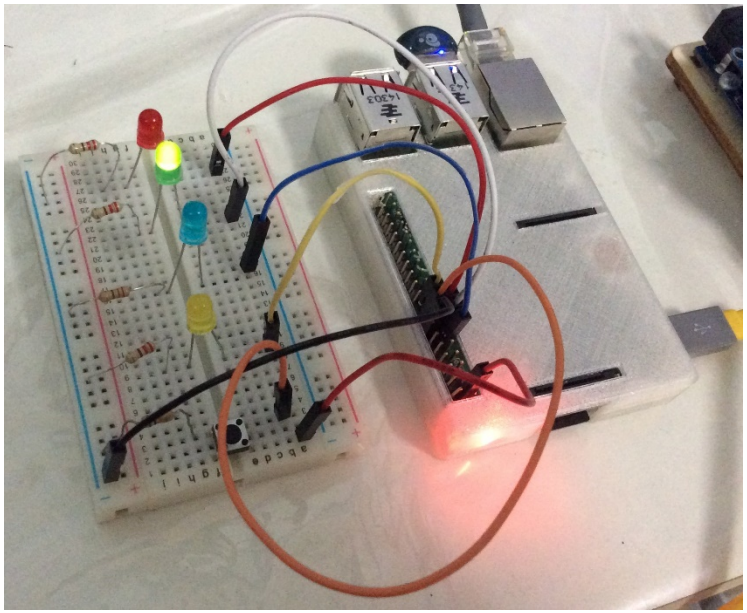


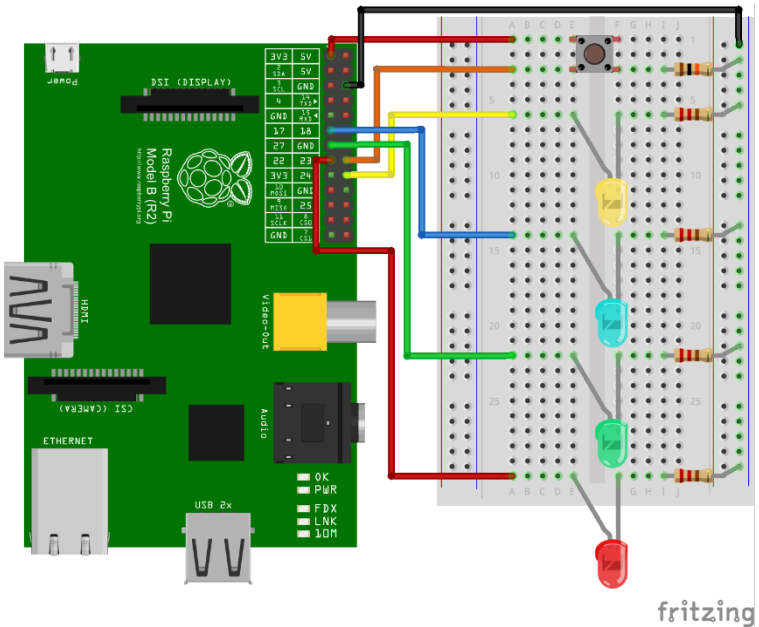


Arduino Parts list

1. Arduino Uno R3
2. RedBearLab BLE (Single-Mode) Shield v.1.1
   - (Not shown in schematic)
   - NRF8001 chipset
3. Red LED
4. Green LED
5. Blue LED
6. Yellow LED
7. 4x 220ohm resistors
8. Push Button

2b. Hardware setup (Raspberry Pi)





Raspberry Pi Parts list

1. Raspberry Pi 2 Model B
2. IOGear GBU521 USB BLE (Dual-Mode) adapter
   - BCM20702 chipset
3. Red LED
4. Green LED
5. Blue LED
6. Yellow LED
7. 4x 220ohm resistors
8. Push Button
9. 10k ohm pull-down resistor

2. Peripheral Architecture Plan

Generic Access Profile (GAP)

Field	Value
Device name (general)	YKM's Arduino (Not accessible via Android APIs)
Local name (specific):	Intro to Arduino BLE
isConnectable	Yes
Services	1 service: UUID = "12345678-9012-3456-7890-123456789012"

Generic Attribute Profile (GATT)

Service 1 (UUID : "12345678-9012-3456-7890-123456789012")

Characteristic 1 (LED)
Value type: char (1-byte character)
UUID : "00000000-0000-0000-0000-000000000010"
Properties: Read, WriteWithoutResponse

Characteristic 2 (Button)
Value type: String
UUID : "00000000-0000-0000-0000-000000000020"
Properties: Read, Notify

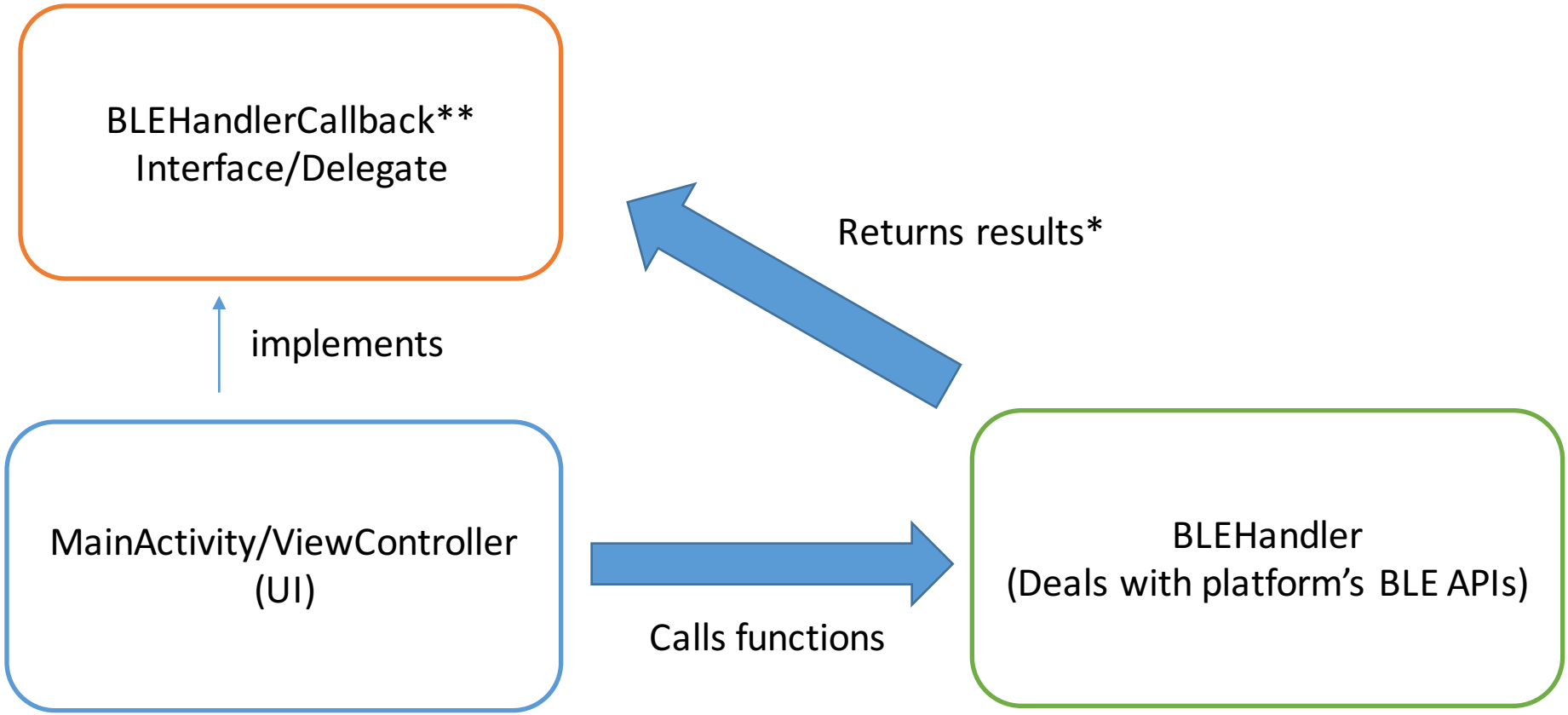
LED characteristic
- Toggles blue LED if central writes "b"
- Toggles yellow LED if central writes "y"

Button characteristic
- Notifies central if button is pressed
- Sends back incrementing number

3a. Arduino code

- Programming Language: C
- Arduino IDE 1.6.7
- Libraries Used
  - ble-sdk-arduino for NRF8001 (By Nordic)
    - https://github.com/NordicSemiconductor/ble-sdk-arduino
  - arduino-BLEPeripheral (By Sandeepmistry)
    - Abstraction over ble-sdk-arduino
    - https://github.com/sandeepmistry/arduino-BLEPeripheral

3b. Central architecture plan (iOS and Android)



*BLE APIs are asynchronous in nature.

**Use BLEHandlerCallback to avoid tight coupling between UI and BLEHandler

3c. iOS Code

- Platform
  - Device: iPod Touch 6G
  - OS: iOS 9.2.1
- Programming Language: Swift 2
- Xcode 7.2.1

3b. Raspberry Pi code

- Platform
  - Device*: Pi 2 Model B
  - OS*: Arch Linux ARM
- Programming Language: Javascript
- Framework used: Nodejs
- Nodejs BLE Library
  - Bleno (by Sandeepmistry again)
  - Abstraction over Linux's Bluez stack/API
  - Aggressive maintenance
  - https://github.com/sandeepmistry/bleno
- Why not others, Python, Go or C?
  - Bleno is more "mature" and "easier to use"

*Others will work too

3c. Android code

- Platform
  - Device: Nexus 5
  - OS: Android 6.0.1
- Programming Language: Java
- Android Studio 1.5.1

4a. General Issues

- Limit data transfer to 20-byte chunks
- Peripheral
  - Characteristics support UTF-8 values
    - I use ASCII for Arduino compatibility, but UTF-8 is generally safe
- Central
  - All callbacks from BLE APIs are not on UI thread
  - Must rescan upon Bluetooth/phone restart
    - Existing CBPeripheral (iOS) and BluetoothDevice (Android) references becomes invalid

4b. iOS issues

- Cannot retrieve Mac Address
  - Generated UUID specific to iOS device
  - Identification issues across iOS devices /Android
  - Solution:
  - Peripheral embeds Mac Address in advertisement (GAP) data
    - Manufacturer data field (Innova Technology)
    - In device/local name fields (Algo Access)
- Aggressive caching of GATT data
  - Receive out-of-date GATT data during peripheral development
  - Solution:
    - Restart iOS's Bluetooth after every change in peripheral software/firmware
- Max number of BLE connections
  - ~20 (online anecdotes)

4c. Android issues (the past)

- Before Android 4.3 (July 2013)
  - Fragmentation hell
  - Proprietary Libraries by OEMs, Android <= 4.2
    - Samsung (quite reliable)
    - HTC – buggy, unreliable
    - Motorola (reliable but conflicts with Android 4.3)
  - Architecture issues
- Testing issues

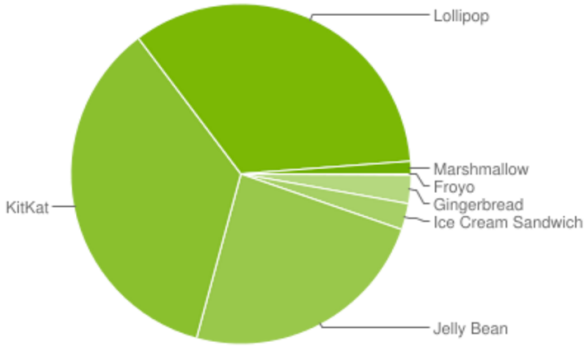
4c. Android issues (today)

1. OS fragmentation

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.5%
4.1.x	Jelly Bean	16	8.8%
4.2.x		17	11.7%
4.3		18	3.4%
4.4	KitKat	19	35.5%
5.0	Lollipop	21	17.0%
5.1		22	17.1%
6.0	Marshmallow	23	1.2%

Data collected during a 7-day period ending on February 1, 2016.

Any versions with less than 0.1% distribution are not shown.



- 74.2% of Android devices support BLE
- Few support peripheral mode: 35.3% minus Nexus 4, 5, 7 (2012/2013)

4c. Android issues (today)

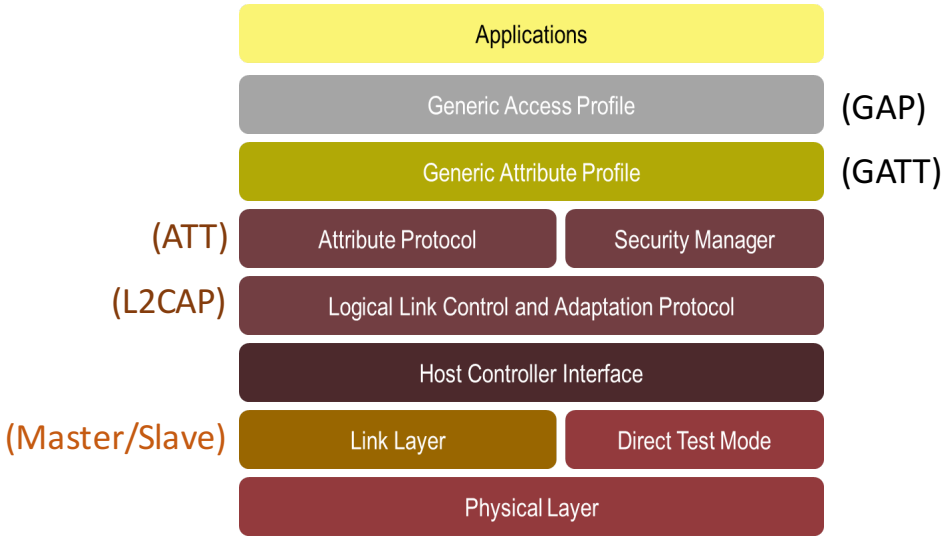
2. APIs considered new, some functions are buggy
3. Frequent connection drops (< 5.0)
4. Max BLE connections:
   - Software cap in Bluedroid code: BTA_GATTC_CONN_MAX, GATT_MAX_PHY_CHANNEL
   - Android 4.3: 4
   - 4.4 - 5.0+: 7
5. No API callback to indicate scanning has stopped
   - Scan indefinite on some phones, Samsung phones: 12 minutes
   - Solution: Restart scan at regular intervals
6. Different scan return result behaviours (See further reading)
7. Bugs on Samsung phones at least < 5.0
   - Scan using service UUID filtering does not work -> no results returned
   - connectGatt() must be called from UI thread
8. HTC
   - Slow LE scan
   - Classic scan returns both Classic and SMART peripherals

4c. Tips for production Android app

- Use Nexus (reference phone) or Motorola for initial development
- Get many models from differing manufacturers

5. BLE layer model

- Link-layer:
  - Defines how two BLE devices communicate. Advertising, Scanning, Connecting, Packet Format
  - Convention is to use Master/Slave instead of Central/Peripheral
- L2CAP:
  - Segmentation and reassembly of packets
  - 4-byte header
  - 23 bytes for MTU
  - Protocol multiplexing
    - 0x0004: ATT Channel (usually used)
    - 0x0005: LE signalling
    - 0x0006: Security Manager
- ATT
  - Action to be taken (Read/Write/…)
  - 1-byte instruction opcode
  - 2-byte handle (ID of relevant service/characteristic/descriptor)
  - 20-byte MTU for application



Source: https://developer.bluetooth.org/KnowledgeCenter/PublishingImages/GATT_stack.png

5. BLE Data Link-layer Packet Structure

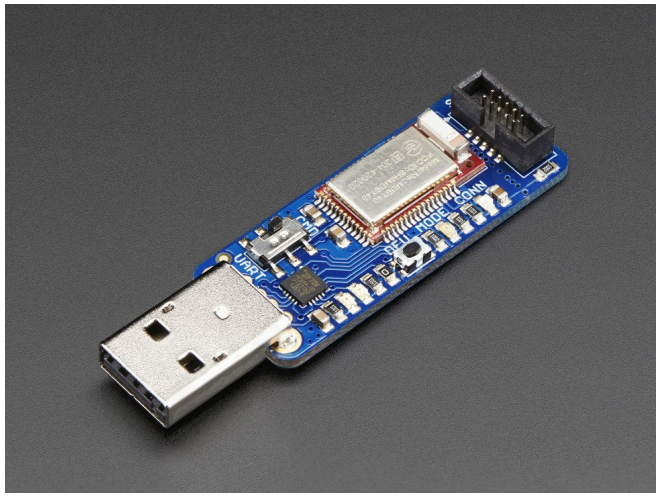
LSB

Field size (bits)	8	32	8	8	0-296	24
Field name	Preamble (Alternating bits for receiver calibration)	Advertising /Data Access Address	Header	Length	Payload	CRC

Protocol/Packet Data Unit (PDU)

- Only 1 packet structure
- Two types of packets
  - Advertising
    - Advertising Access Address: Always 0x8E89BED6
  - Data
    - Data Access Address: Random for every connection
      - Allows Master/Slave to distinguish packets associated with a connection
      - Mac Address no longer used for data packets
    - Usually carries L2CAP/ATT payload
- PDU header format for Advertising != Data

6. BLE Sniffer



- Adafruit Bluefruit LE Sniffer
- Based on Nordic nRF51822
- Required software:
  - Nordic nRF Sniffer (Windows-only)
  - Results piped to Wireshark
- Alternative: Ubertooth One

6. Sniffer: Advertising

Link layer format

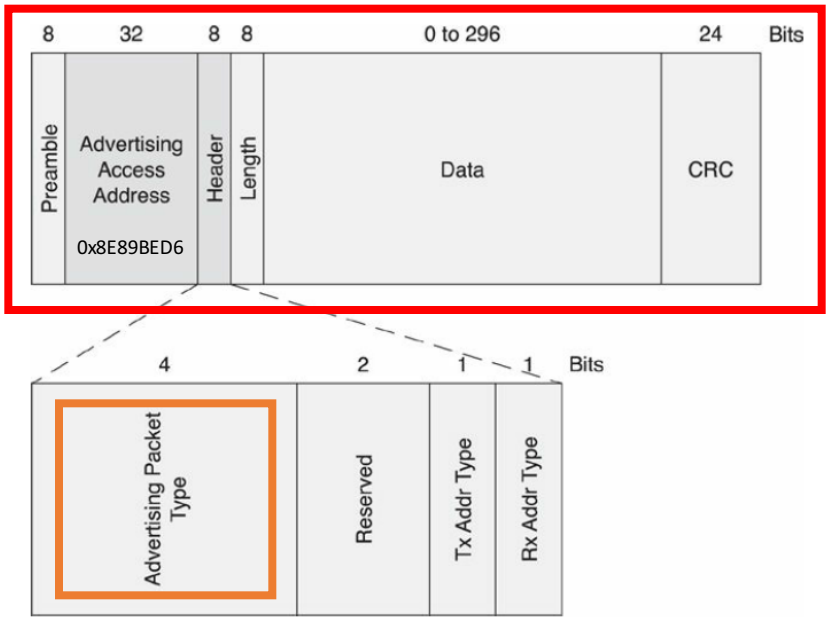


Figure 7–10. The contents of an advertising packet header

Source: BLE: The Developer's Handbook by Robin Heydon, pg82

- ADV packets' payload contains GAP data:
  - Mac Address
  - Service UUID
  - Supported Bluetooth features: Dual/Single mode
  - TX Power (Optional)
  - Name (Optional)
- **PDU/Advertising Type:**
  - **4-bit field determines type of ADV Packet**
- Slave is connectable
  - **0000: ADV_IND (Undirected connectable mode)**
    - No need to connect in a hurry
  - 0001: ADV_DIRECT_IND (Directed connectable mode)
    - To indicate to master that slave wants to be connected quickly.
    - Max 1.28s in this mode
- Slave is not connectable
  - 0010: ADV_NONCONN_IND (Not scannable)
    - Will not respond to scan (SCAN_REQ) requests for more info
  - 0110: ADV_SCAN_IND
    - Will response to SCAN_REQ with SCAN_RSP

6. Sniffer: Scan

- ADV Packets may not hold all advertising info
  - Central can issue SCAN_REQ to ask for more

- 0011: SCAN_REQ (Active Scan Request)
  - Master -> Slave
  - Ask peripheral for complete GAP data
- 0100: SCAN_RSP (Response)
  - Slave -> Master
  - Contains slave's name, TX power, …

6. Sniffer: Connection

- PDU Type:
- 0101: Connect_REQ (Connect Request)
  - Master -> Slave
  - Master selects and sends a random data access address
    - Link-layer data -> Access address field
- 0110: Empty PDU (Keep-alive packet)
  - Sent at connection interval between Master <-> Slave
  - Filter "*not btle.data_header.llid==0001*" to ignore in Wireshark

6. Sniffer: Data Packets

Link layer format

- Payload usually contains L2CAP/ATT data

- Link-layer identifier (LLID) – 2 bits
  - 11 : Control Packet
  - 10 : Start/Full Packet
  - 01: Continuation of fragmented packet

- If LLID == 11 (Control Packet)
  - Header format changes to have control and error fields
  - Does not contain L2CAP/ATT payload data
  - 0x0c: LL_VERSION_IND: Negotiate supported Bluetooth Spec
  - 0x01: LL_CHANNEL_MAP_REQ: Channel hop (Master -> Slave)
  - 0x02: LL_TERMINATE_IND: Terminate connection

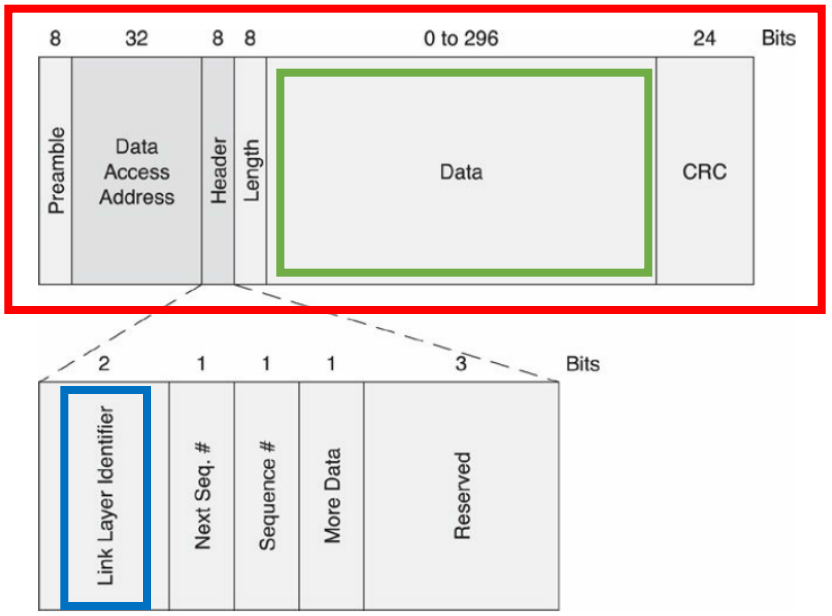


Figure 7–11. The contents of a data packet header

Source: BLE: The Developer's Handbook by Robin Heydon, pg83

6. Sniffer: Discover services/characteristics

- ATT opcodes
- 0x10: Read by Group Type Request (Discover Services)
  - Master -> Slave
- 0x11: Read by Group Type Response
  - Slave -> Master
  - Returns Services Requested
- 0x08: Read by Type Request (Discover Characteristics)
  - Master -> Slave
- 0x09: Read by Type Response
  - Slave -> Master
  - Returns Characteristics Requested

You may notice some "hidden" services during sniffing

- Generic Access Service: 0x1800 (Contains generic info, name, type etc about peripheral)
- Generic Attribute Service: 0x1801 (I don't know what this is)

6. Sniffer: Data transfer

- 0x52: Write Command (Write to Characteristic)
  - Master -> Slave
- 0x1b: Handle Value Notification (Notify Characteristic Changed)
  - Slave -> Master

7. Further reading

- BLE 4.0-4.1 Security (Passive) Weaknesses (19:58 to 23:14)
  - Video: https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan
  - Paper: https://lacklustre.net/bluetooth/Ryan_Bluetooth_Low_Energy_USENIX_WOOT.pdf
- In-depth introduction by Nordic Semiconductor
  - https://www.youtube.com/watch?v=BZwOrQ6zkzE
- Acceptable types of Characteristic values
  - https://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorViewer.aspx?u=org.bluetooth.descriptor.gatt.characteristic_presentation_format.xml
- BLE Sniffer (by Adafruit)
  - https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-sniffer
- Android 4.3 BLE unstable
  - http://stackoverflow.com/questions/17870189/android-4-3-bluetooth-low-energy-unstable
- Android different scan results behaviour
  - http://stackoverflow.com/questions/19502853/android-4-3-ble-filtering-behaviour-of-startlescan
- Android 5.0 BLE APIs improvement vs 4.3
  - https://www.youtube.com/watch?v=qx55Sa8UZAQ
- BLE Advertising Packet Format
  - http://j2abro.blogspot.sg/2014/06/understanding-bluetooth-advertising.html
- Bluetooth Core (Adopted) Specification
  - https://www.bluetooth.org/en-us/specification/adopted-specifications

8a. Can Peripheral prevent unwanted connections from unknown Central?

- Not possible to block connection attempt
- But peripheral can disconnect the central after connected
  - Wait for key-exchange
  - Mac address whitelist
- Disconnect APIs
  - arduino-BLEPeripheral
    - *blePeripheral.disconnect();*
  - Bleno
    - *bleno.disconnect();*

8b. Who defines the attributes?

- Peripheral always defines the attributes
  - Services, characteristics and descriptors

- Then why did I do this on the Central?

- Android:

```
public class BLEHandler {

    private static final String TAG = "BLEHandler";

    private static final UUID UUID_SERVICE =     UUID.fromString("12345678-9012-3456-7890-123456789012");
    private static final UUID UUID_CHAR_LED =    UUID.fromString("00000000-0000-0000-0000-000000000010");
    private static final UUID UUID_CHAR_BUTTON = UUID.fromString("00000000-0000-0000-0000-000000000020");
```

- iOS:

```
class BLEHandler : NSObject, CBCentralManagerDelegate, CBPeripheralDelegate{

    let TAG = "BLEHandler"

    let UUID_SERVICE : CBUUID     = CBUUID(string: "12345678-9012-3456-7890-123456789012")
    let UUID_CHAR_LED : CBUUID    = CBUUID(string: "00000000-0000-0000-0000-000000000010")
    let UUID_CHAR_BUTTON : CBUUID = CBUUID(string: "00000000-0000-0000-0000-000000000020")
```

- Reason:
  - I hardcoded the characteristic UUIDs to address the characteristics directly since I already know their purpose

8c. BLE Security?

- Bluetooth pairing
- < Bluetooth 4.2:
  - Strongly discouraged to use native BLE security features Key-exchange protocol weakness
  - See video in Further Reading
- Security issues fixed in 4.2 (Dec 2014)
  - But many devices in the market have not adopted this

8d. Data loss from using writeWithoutResponse instead of write property?

- Possibility exists but unlikely to happen in practice
- Rough Analogy:
  - write vs writeWithoutResponse -> TCP vs UDP
  - Possible to lose data if central sends faster than peripheral can process