

# BLE Introduction and Labs using Raspberry Pi

Lab: device connection via BLE interface of a Raspberry Pi

# BLE History

- In **2011**, the Bluetooth Special Interest Group announced the Bluetooth SMART logo scheme, intended to clarify compatibility between LE devices
- **Bluetooth Smart Ready** indicates a **dual-mode** device, typically a laptop or smartphone, whose hardware is compatible with both Classic and LE Bluetooth peripherals
- **Bluetooth Smart** indicates an **LE-only** device, typically a battery-operated sensor, which requires either a SMART Ready or another SMART device in order to function

# About Bluetooth

- History of Bluetooth



A New Era of Traffic Detection

## BLUETOOTH 5

1998

**Bluetooth "Classic":** Established ability for wireless connectivity between devices.

2011

**Bluetooth "Low Energy":** Coin cell battery-operated sensors and fitness gadgets could smoothly connect to BLE technology-enabled smart phones and tablets.

2016

**Bluetooth "5":** Combining the best aspects of both previous technologies.



**BLE and B5 use different technologies for advertising their presence, connection, and pairing.**

**Can not be detected with traffic sensors searching for in-vehicle devices using Bluetooth Classic.**

**Must combine multiple Bluetooth scanners and Wi-Fi for best detection rates.**

# Bluetooth Class Vs Bluetooth Low Energy (Bluetooth 4.0)

Technical Specification	Classic Bluetooth technology	Bluetooth low energy technology
Distance/Range	100 m (330 ft)	50 m (160 ft)
Over the air data rate	1–3 Mbit/s	1 Mbit/s
Application throughput	0.7–2.1 Mbit/s	0.27 Mbit/s
Active slaves	7	Not defined; implementation dependent
Security	56/128-bit and application layer user defined	128-bit <a href="#">AES</a> with Counter Mode <a href="#">CBC-MAC</a> and application layer user defined
Robustness	Adaptive fast frequency hopping, <a href="#">FEC</a> , fast <a href="#">ACK</a>	Adaptive frequency hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check
Latency (from a non-connected state)	Typically 100 ms	6 ms
Total time to send data (det.battery life)	100 ms	3 ms <sup>[citation needed]</sup> , <3 ms <sup>[25]</sup>
Voice capable	Yes	No
Network topology	<a href="#">Scatternet</a>	<a href="#">Star-bus</a>
Power consumption	1 as the reference	0.01 to 0.5 (depending on use case)
Peak current consumption	<30 mA	<15 mA
Service discovery	Yes	Yes
Profile concept	Yes	Yes
Primary use cases	Mobile phones, gaming, headsets, stereo audio streaming, automotive, PCs, security, proximity, healthcare, sports & fitness, etc.	Mobile phones, gaming, PCs, watches, sports and fitness, healthcare, security & proximity, automotive, home electronics, automation, Industrial, etc.

# BT Classic (Bluetooth) v.s BLE(bluetooth low energy)

- BLE (BT 4.0)
  - Not for audio
  - Not for file
  - for data sensor
  - Small quantity
  - Short range
  - Quick Connection
  - Support one (central) to many (peripherals)
- BLE (BT 4.0) is not to replace BT classic
- BLE (BT 5.2) introduces BLE Audio

# Bluetooth 5 vs Bluetooth 4

- 2x Speed
  - § 2M PHY will double the throughput up to 1.4Mbps
  - § 15-50% lower power consumption
- 4x Range
  - § 125/500kbps codec PHYs improve sensitivity /range
  - § New channel selection algorithm enables +20dBm TX
- 8x Advertisement Capacity
  - § Advertisement payload grows from 31B to 255B
  - § 37 new advertisement channels help offload 3 primary
- advertisement channels
  - § New advertisement schemes for advanced beacons
  - § Periodic Advertisement

# Bluetooth 5.2, 5.1 and 5.0

- BT 5
  - § 2x data throughput with 2Mbps PHY: Faster OTAs
  - § 4x range: Building automation
  - § 8x Enhanced Advertisements configuration: Multiple Beacons
- BT 5.1
  - § Direction finding: Asset tracking
  - § Gatt Caching : Lower power on service discovery
- BT 5.2
  - § LE Isochronous Channel: **Audio** peer to peer and Broadcast
  - § LE Power Control: Dynamic TX change, lower power more reliability

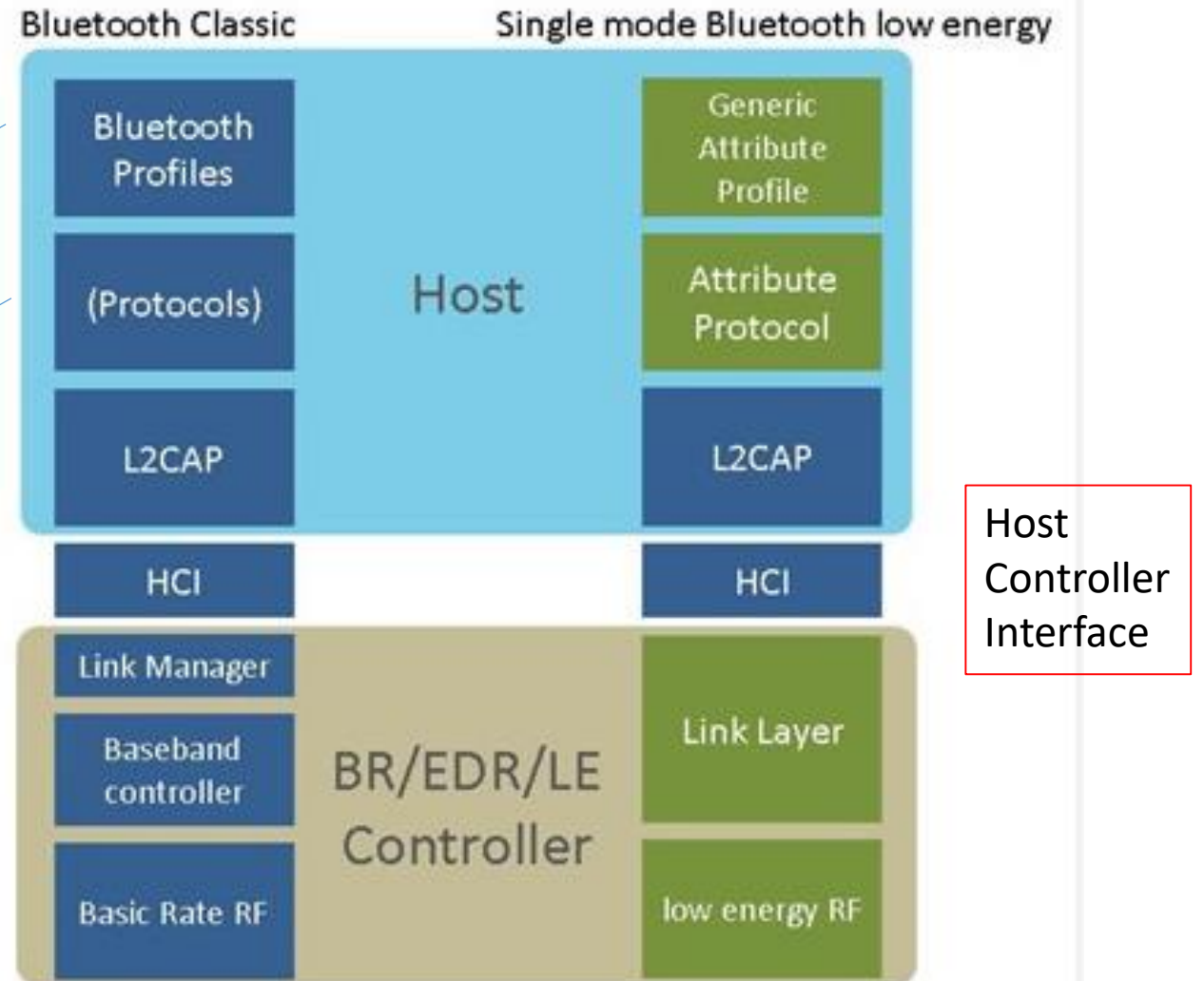
# Protocol stack

- BT classic v.s BLE (4.0)

Such as  
Headset Profiles  
SPP Profiles

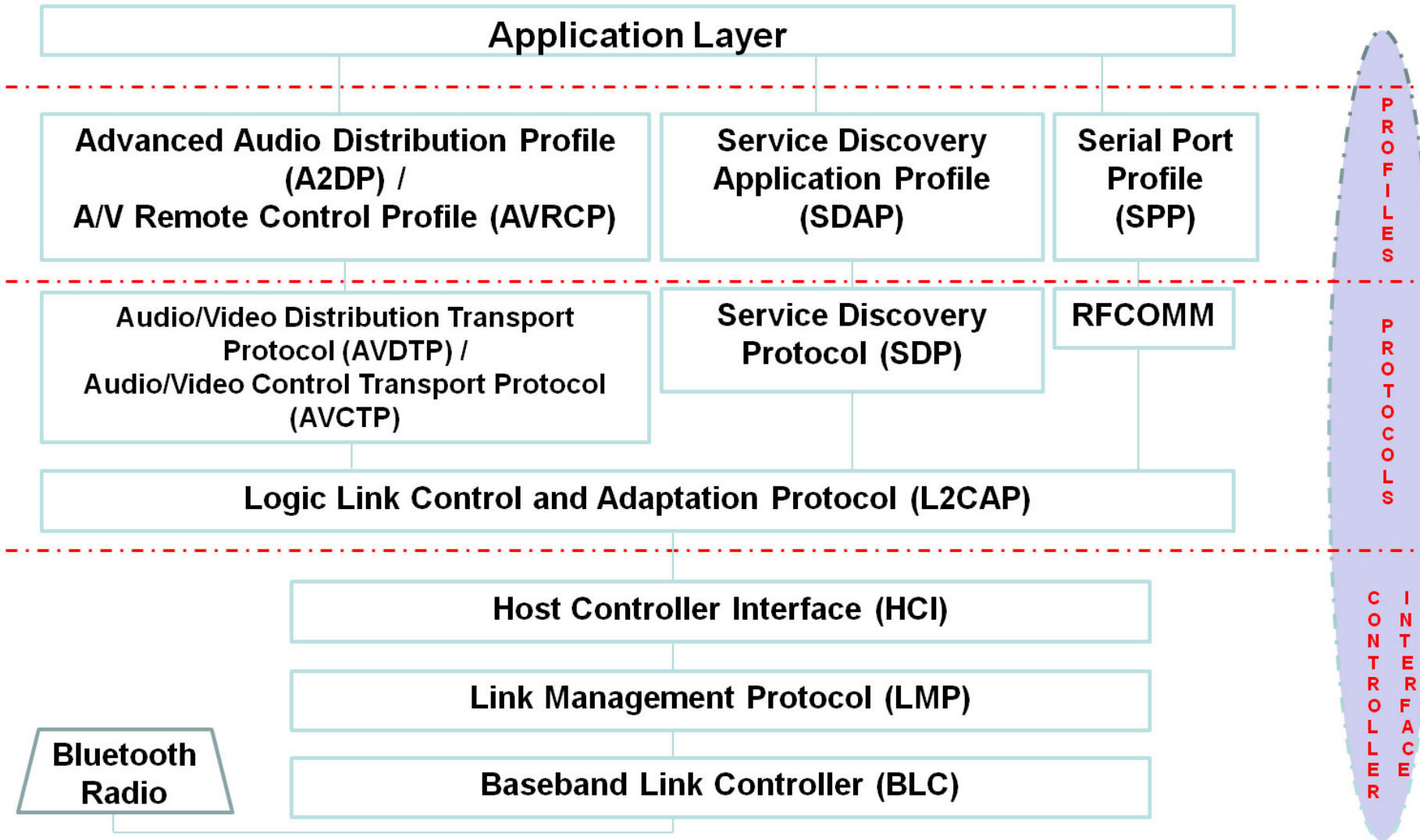
Such as  
RFCOMM Protocol

Logical Link Control and Adaptation Layer Protocol (L2CAP) is layered over the Baseband Protocol and resides in the data link layer. L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions.

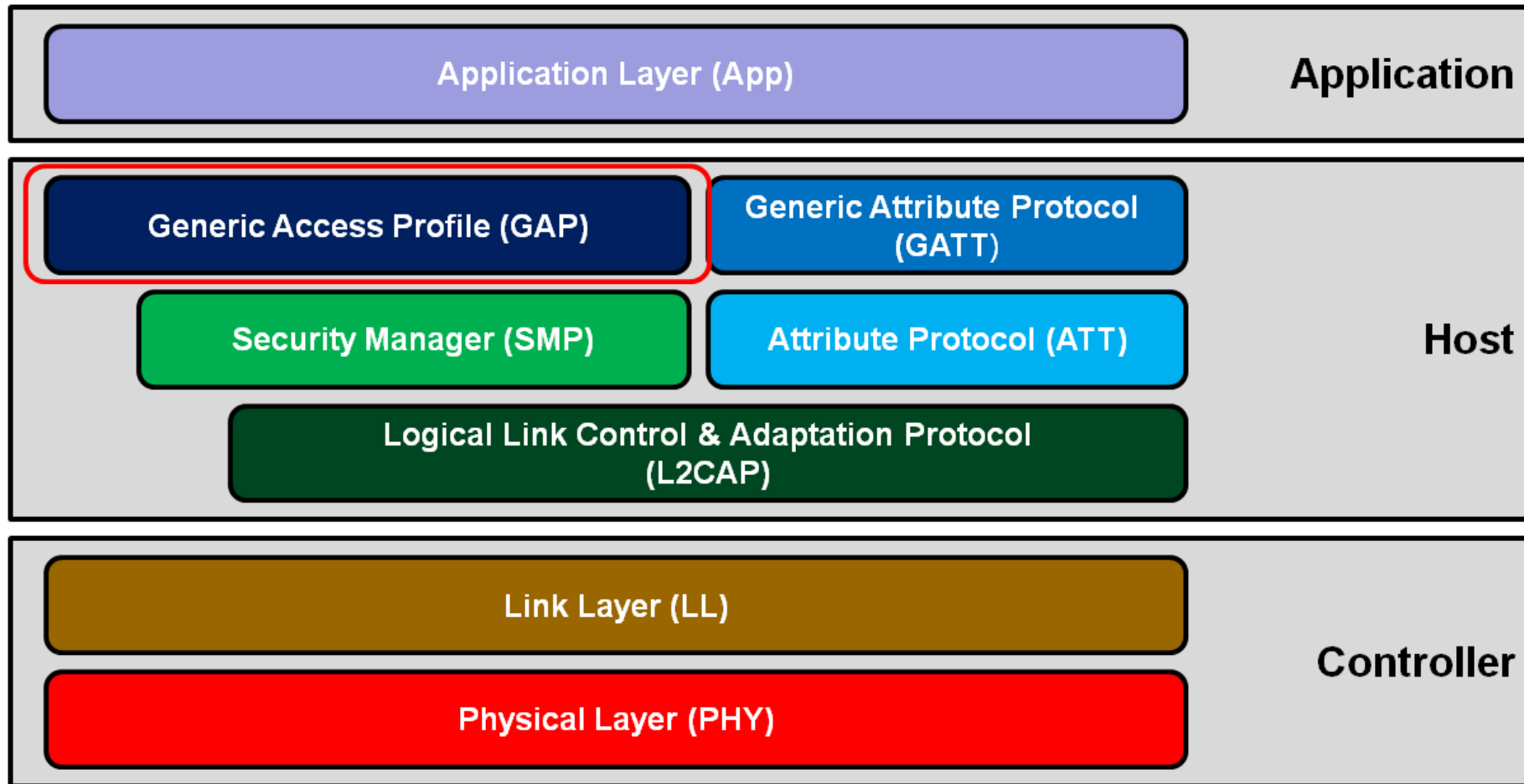




# Bluetooth Profiles and Protocols



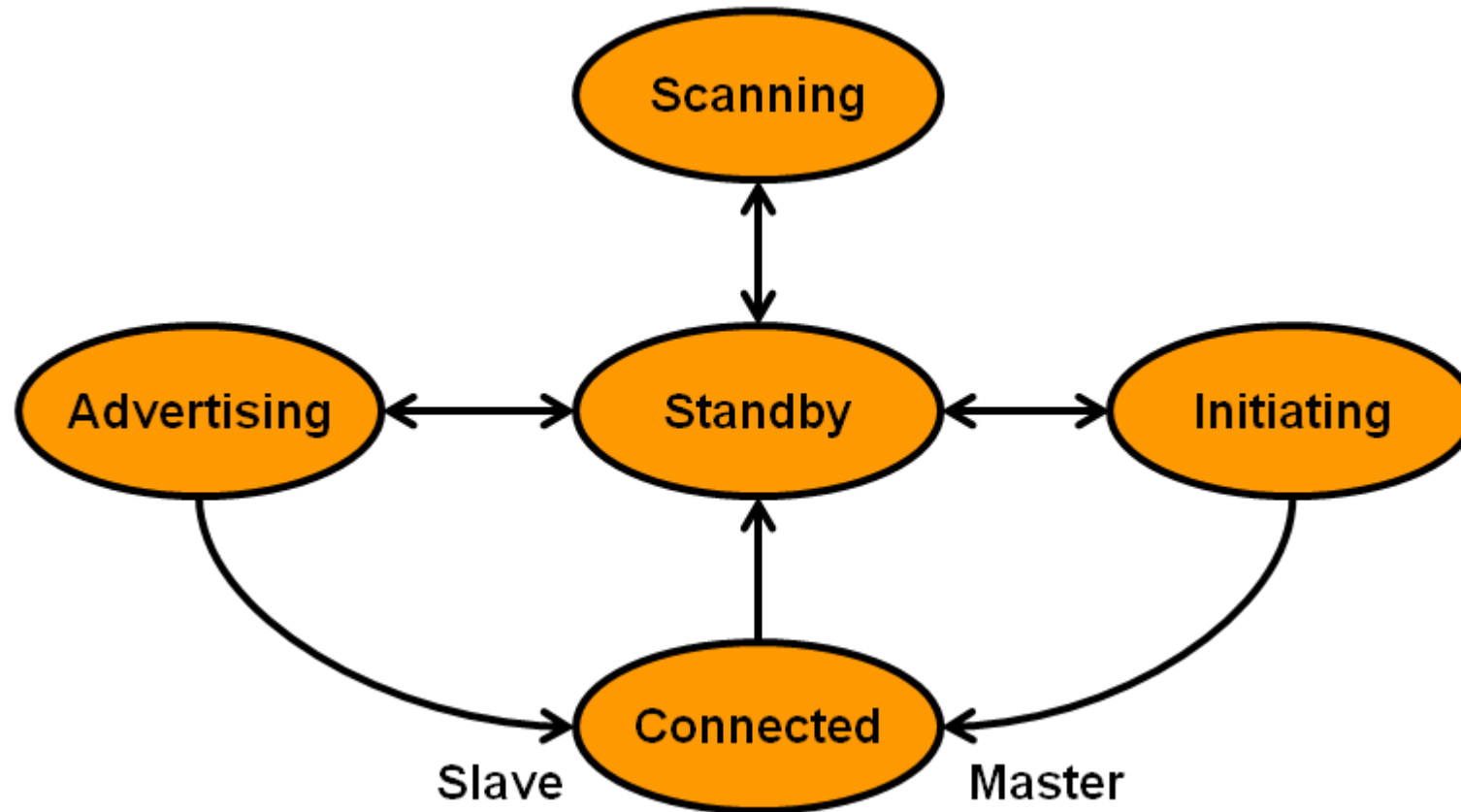
The **Generic Access Profile (GAP)** modes and procedures form the cornerstone of the Bluetooth Low Energy (BLE) '**control-plane**' operations



# ATT, GATT, UUID

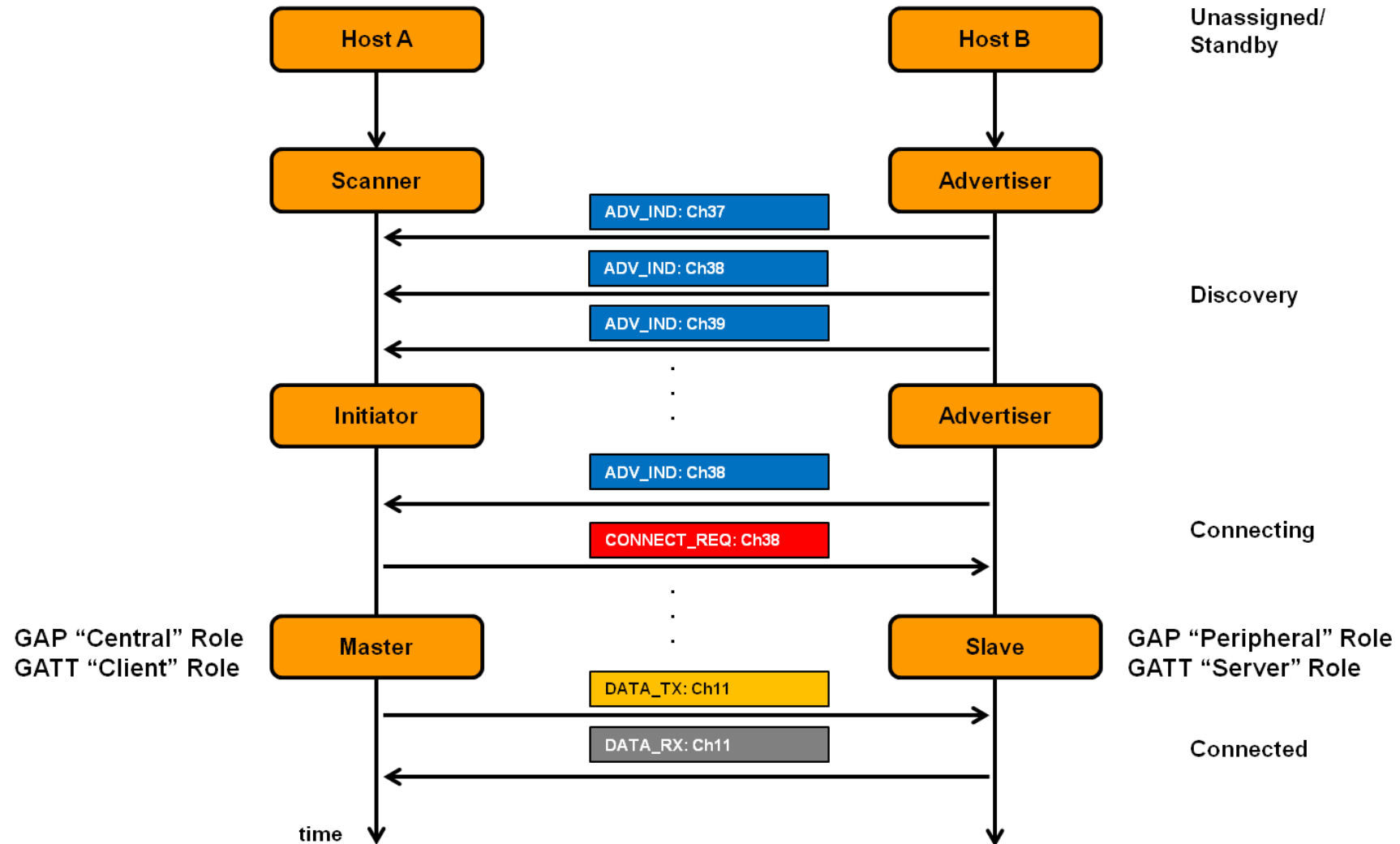
- GATT (Generic Attribute Profile) and ATT (Attribute Protocol)
  - Every BLE profile is expected to use ATT. But they can also be used over classic Bluetooth (BR/EDR).
  - Every Low Energy profile must be based on **GATT**. So, ultimately, every LE service uses ATT as the application protocol.
- ATT: Attribute Protocol
  - a 16-bit handle;
  - an UUID which defines the **attribute type**;
  - a value of a certain length.
- GATT: Generic Attribute Profile
  - **GATT is a base profile for all top-level LE profiles**. It defines how a bunch of ATT attributes are grouped together into meaningful services.

# Link Layer State Machine



Note that the Link Layer **Master** is also the **GAP Central** and **GATT Client**, while the Link Layer **Slave** is the **GAP Peripheral** and **GATT Server**.

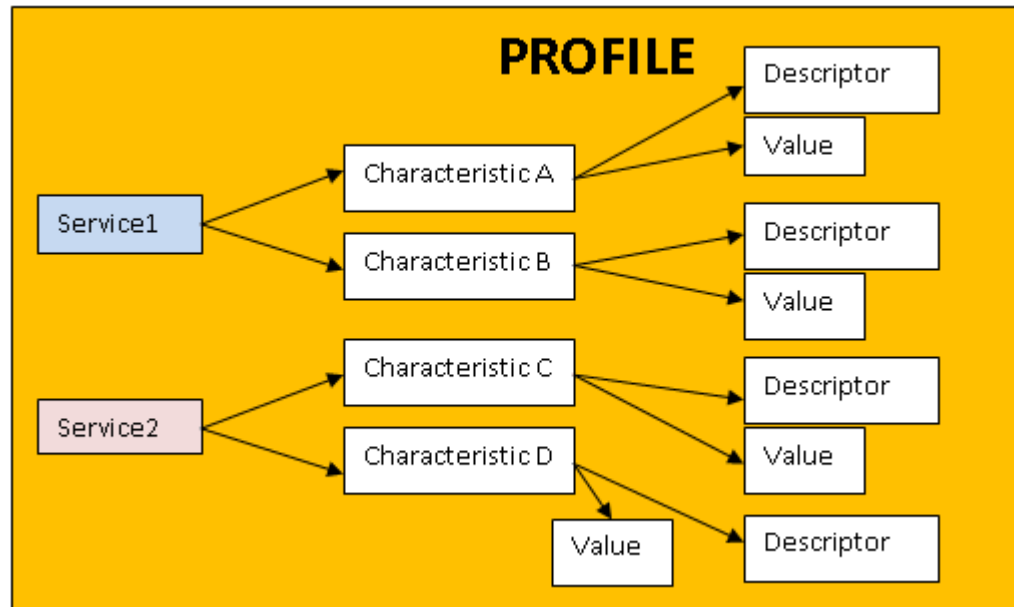
# Unicast (Peer-Peer) Connection



# GATT: Generic Attribute Profile

Ref: [https://epxx.co/artigos/bluetooth\\_gatt.html](https://epxx.co/artigos/bluetooth_gatt.html)

- **GATT services**



Handle	UUID	Description
0x0100	0x2800	Service A definition
...	...	Service details
0x0150	0x2800	Service B definition
...	...	Service details
0x0300	0x2800	Service C definition
...	...	Service details

# Standard Bluetooth UUID base

- For efficiency, and because 16 bytes would take a large chunk of the 27-byte data payload length of the Link Layer, the BLE specification adds two additional UUID formats: 16-bit and 32-bit UUIDs.
  - Examples: UUIDprimary service (0x2800) and UUIDsecondary service (0x2801)
  - These shortened formats can be used only with UUIDs that are defined in the Bluetooth specification (i.e., that are listed by the Bluetooth SIG as standard Bluetooth UUIDs).
- To reconstruct the full 128-bit UUID from the shortened version, insert the 16- or 32-bit short value (indicated by xxxxxxxx, including leading zeros) into the Bluetooth Base UUID:  
xxxxxxxx-0000-1000-8000-00805F9B34FB

# Example: a GATT Heart Rate Service

	handle	type (UUID)	description or property	value
Service	0x0021	SERVICE	READ	HRS
Characteristic	0x0024	CHAR	READ	NOT 0x0027 HRM
	0x0027	HRM	NONE	bpm
Descriptor	0x0028	CCCD	READ/WRITE	0x0001
Characteristic	0x002A	CHAR	READ	RD 0x002C BSL
	0x002C	BSL	READ	<i>finger</i>



# CCCD

- CCCD is an abbreviation for **Client Characteristic Configuration Descriptor**.
  - This descriptor is defined in the Core Specification, BLE.
- The use of it is for a GATT Client to control what kind of packets the GATT Server can send to it.
  - As you may know, there are some so-called server initiated properties, **Notifications and Indications**.
  - A Server can only send such packets to the Client if the Client have written either 1 (to enable **notifications**) or 2 (to **enable indications**) to the CCCD for the characteristic in question.

For 16-bit UUID, please see

<https://btprodspecificationrefs.blob.core.windows.net/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf>

# Hands-on Lab

You can use other Linux host (such as a Linux Notebook) to replace RPi

- Using tools: `hcitool`, `hciconfig`, ...
- Try the following commands in your **target** (RPi)
- Scan BLE devices in the nearby
  - (target)`sudo hciconfig`
  - (target)`sudo hcitool lescan`
- Set self to discoverable and try using Android BLE Scanner App to connect to your RPi
  - `sudo hciconfig hci0 leadv 0`

What is your BLE address in your RPi?

Note: The mac address (or hw address) for a **BLE peripheral** is often generated randomly.

# Android App –BLE scanner— as a helper

- Install BLE scanner App (by Bluepixel) at your Android phone
  - iPhone probably has the similar app



BLE Scanner: Read,Write,Notify

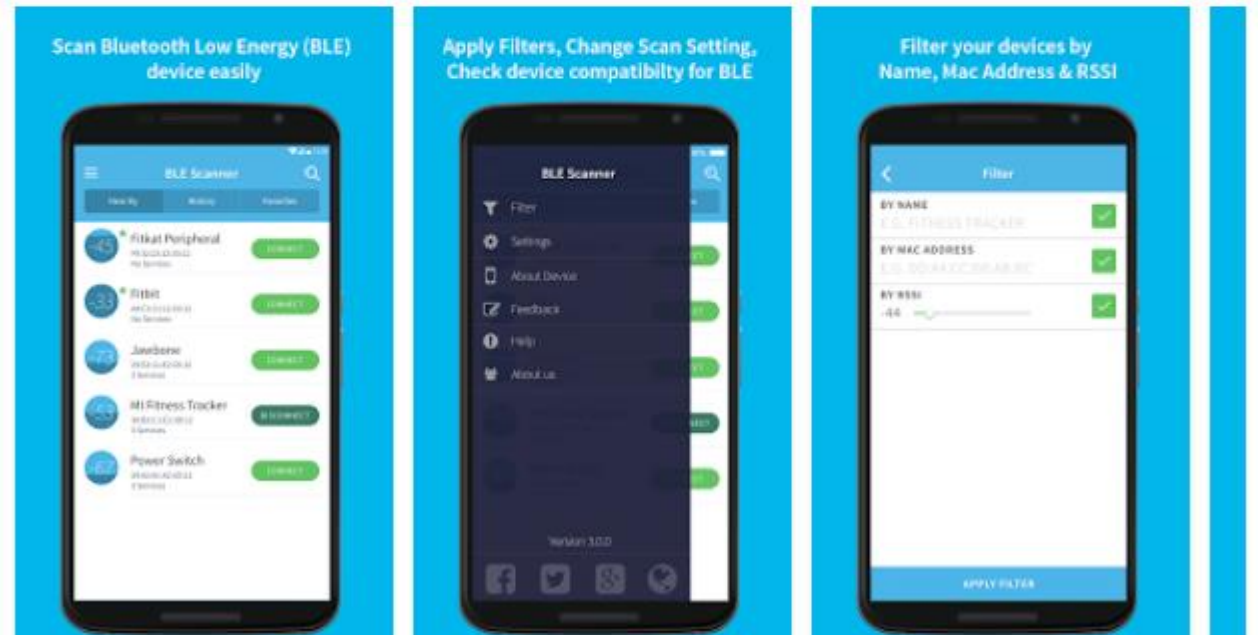
Bluepixel Technologies LLP 工具

★★★★★ 538

3+

加入願望清單

安裝



下午9:02

 **echo** DISCONNECT

Status: CONNECTED  
NOT BONDED

### GENERIC ACCESS

0x1800  
PRIMARY SERVICE

### GENERIC ATTRIBUTE

0x1801  
PRIMARY SERVICE

### CUSTOM SERVICE

0000EC00-0000-1000-8000-00805F9B34FB  
PRIMARY SERVICE

CUSTOM CHARACTERISTIC



UUID: 0000EC0E-0000-1000-8000-00805F9B34FB


Properties: READ,WRITE,NOTIFY  
Write Type:WRITE REQUEST


#### Descriptors:

Client Characteristic Configuration



UUID: 0x2902

下午9:03

 **echo** DISCONNECT

Status: CONNECTED  
NOT BONDED

### GENERIC ACCESS

0x1800  
PRIMARY SERVICE

### GENERIC ATTRIBUTE

0x1801  
PRIMARY SERVICE

### CUSTOM SERVICE

0000EC00-0000-1000-8000-00805F9B34FB  
PRIMARY SERVICE

CUSTOM CHARACTERISTIC



UUID: 0000EC0E-0000-1000-8000-00805F9B34FB

Properties: READ,WRITE,NOTIFY  
Value:abcd  
Hex: 0x61626364  
Write Type:WRITE REQUEST

#### Descriptors:

Client Characteristic Configuration



UUID: 0x2902

# Key concepts: BLE Pairing and Bonding

- Pairing: process where devices exchange the information necessary to establish an encrypted connection. It involves authenticating the identity of the two devices to be paired, encrypting the link, and distributing keys to allow security to be restarted on a reconnection.
- Bonding: process where the information from the pairing process is stored on the devices, so that the pairing process does not have to be repeated every time the devices reconnect to each other.

# BLE central using gatttool

- BLE Peripheral: Use a RPi running `bleno` as a peripheral, say A
  - Use the `echo` server in `bleno` example directory
- BLE Central
  - Use `gatttool` in another RPi, say B, to connect to A
  - or
  - Use BLE scanner App in your Android phone to connect to A
- **Note: if using `gatttool` to connect a peripheral that is using random BLE addresses**  
`gatttool -b <peripheral address> -t random -I`

otherwise, `gatttool` will show an error message

```
Attempting to connect to <peripheral address>  
Error: connect error: Connection refused (111)
```

**Please also note that the random peripheral address will change from time to time or always.**

# gattlib

- Gattlib:

- Provides API for C, based on Bluez
- gattlib requires Bluez to work with Bluetooth Low Energy (BLE) devices.
- <https://github.com/labapart/gattlib>

```
pi@raspberrypi:~$ sudo apt show bluez
Package: bluez
Version: 5.55-3.1+rpt1
Priority: optional
Section: admin
```

- Python BLE interface

- <https://bitbucket.org/OscarAcena/pygattlib>
- <https://github.com/IanHarvey/bluepy>
- [https://github.com/adafruit/Adafruit\\_Python\\_BluefruitLE](https://github.com/adafruit/Adafruit_Python_BluefruitLE)
- ...

BlueZ provides support for the core Bluetooth layers and protocols

- Bluez is the Linux Bluetooth Protocol Stack implementation.
- <http://www.bluez.org/about/>

# Using Bluetooth LE with Python

- Installing **bluepy** (for latest version of OS, python3 is used.)

```
(target) sudo apt install python3-pip
```

```
(target) sudo apt install libglib2.0-dev
```

```
(target) sudo pip install bluepy
```

- Bluepy contains some command line program
  - `blescan` (similar to `hcitool lescan`)
  - `sensortag`: connects to various versions of SensorTag



# Bluepy scan, connect, get characteristic example

```
# ble_scan_connect.py:
from bluepy.btle import Peripheral, UUID
from bluepy.btle import Scanner, DefaultDelegate

class ScanDelegate(DefaultDelegate):
    def __init__(self):
        DefaultDelegate.__init__(self)
    def handleDiscovery(self, dev, isNewDev, isNewData):
        if isNewDev:
            print ("Discovered device", dev.addr)
        elif isNewData:
            print ("Received new data from", dev.addr)
scanner = Scanner().withDelegate(ScanDelegate())
devices = scanner.scan(10.0)
n=0
addr = []
for dev in devices:
    print ("%d: Device %s (%s), RSSI=%d dB" % (n, dev.addr,
dev.addrType, dev.rssi))
    addr.append(dev.addr)
    n += 1
    for (adtype, desc, value) in dev.getScanData():
        print (" %s = %s" % (desc, value))
```

```
number = input('Enter your device number: ')
print ('Device', number)
num = int(number)
print (addr[num])
#
print ("Connecting...")
dev = Peripheral(addr[num], 'random')
#
print ("Services...")
for svc in dev.services:
    print (str(svc))
#
try:
    testService = dev.getServiceByUUID(UUID(0xff0))
    for ch in testService.getCharacteristics():
        print (str(ch))
#
    ch = dev.getCharacteristics(uuid=UUID(0xff1))[0]
    if (ch.supportsRead()):
        print (ch.read())
#
finally:
    dev.disconnect()
```

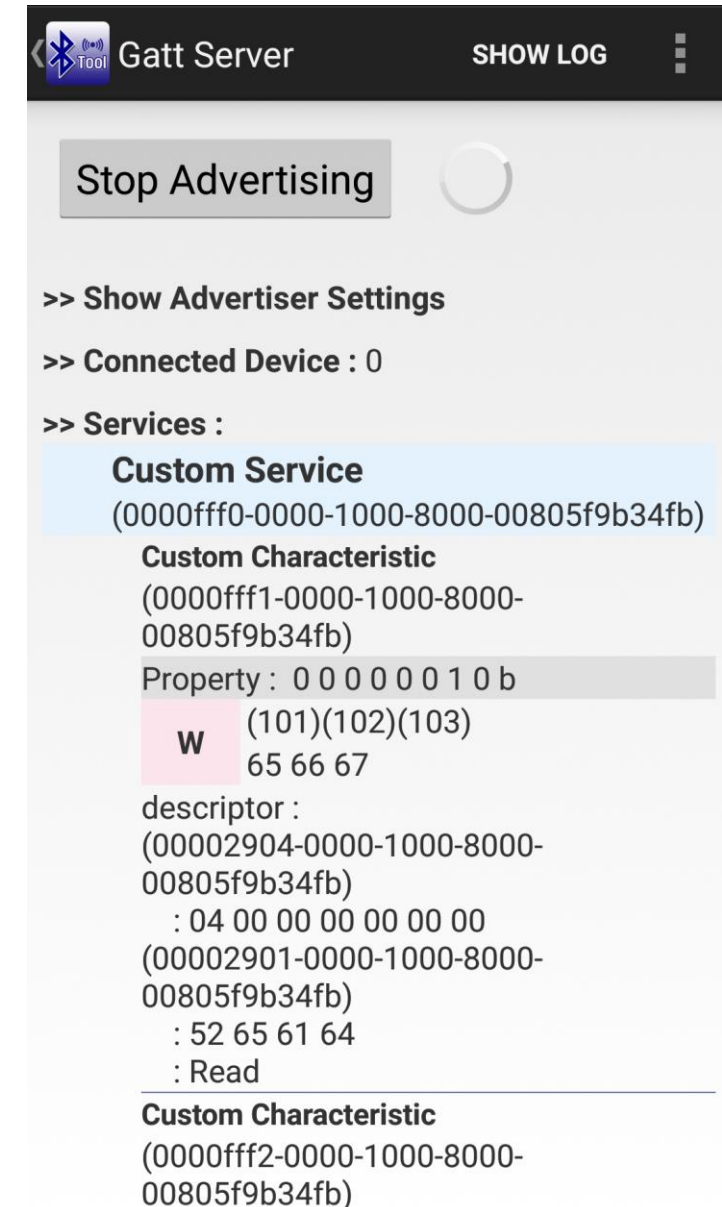
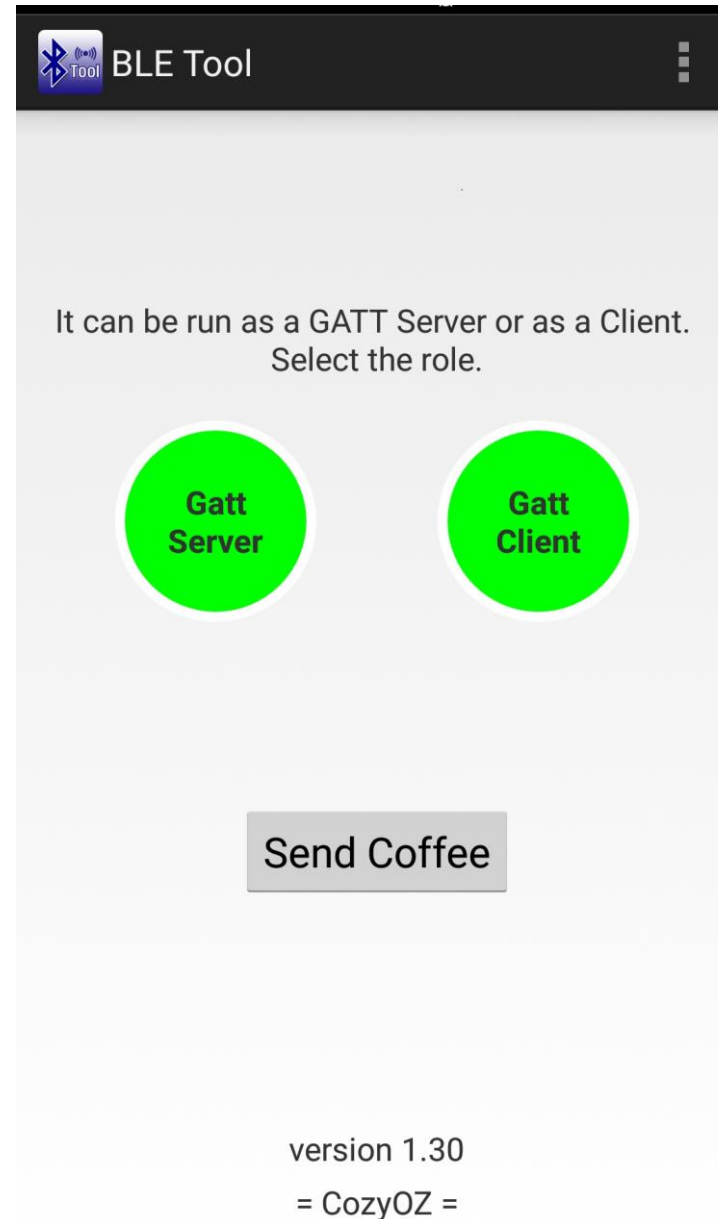
# Testing the BLE python program

- RPi

```
sudo python ble_scan_connect.py
```

- Android Phone:

BLE Tool -> Gatt Server -> Start Advertising -> Press "W" change the first characteristic value to be 0x65 0x66 0x67



# sudo python ble\_scan\_connect.py

```
pi@raspberrypi:~ $ sudo python ble_scan_connect.py
Discovered device 40:d2:2d:95:59:54
Discovered device 48:39:a0:8d:af:15
Discovered device 45:18:1f:7a:dc:27
0: Device 45:18:1f:7a:dc:27 (random), RSSI=-74 dB
  Flags = 02
  Tx Power = f9
  Complete Local Name = ASUS_Z01KD
1: Device 40:d2:2d:95:59:54 (random), RSSI=-93 dB
  Flags = 06
  Manufacturer = 4c0010020b00
2: Device 48:39:a0:8d:af:15 (random), RSSI=-65 dB
  Flags = 02
  Tx Power = f9
  Complete 16b Services = 0000fff0-0000-1000-8000-00805f9b34fb
  Complete Local Name = zf4
Enter your device number: 0
('Device', 0)
45:18:1f:7a:dc:27
Connecting...
Services...
Service <uuid=Generic Attribute handleStart=1 handleEnd=3>
Service <uuid=Generic Access handleStart=20 handleEnd=26>
Service <uuid=fff0 handleStart=40 handleEnd=65535>
Characteristic <fff1>
Characteristic <fff2>
Characteristic <fff3>
Characteristic <fff4>
efg
```

Note  
the  
device  
name