IoT Engineering 5: Local Connectivity with Bluetooth LE

Slides: tmb.gr/iot-5

n|w

Overview

These slides introduce *Bluetooth Low Energy*.

Examples for the peripheral and central role.

Designing BLE services and characteristics.

Prerequisites

Install the Arduino IDE and set up the nRF52840:

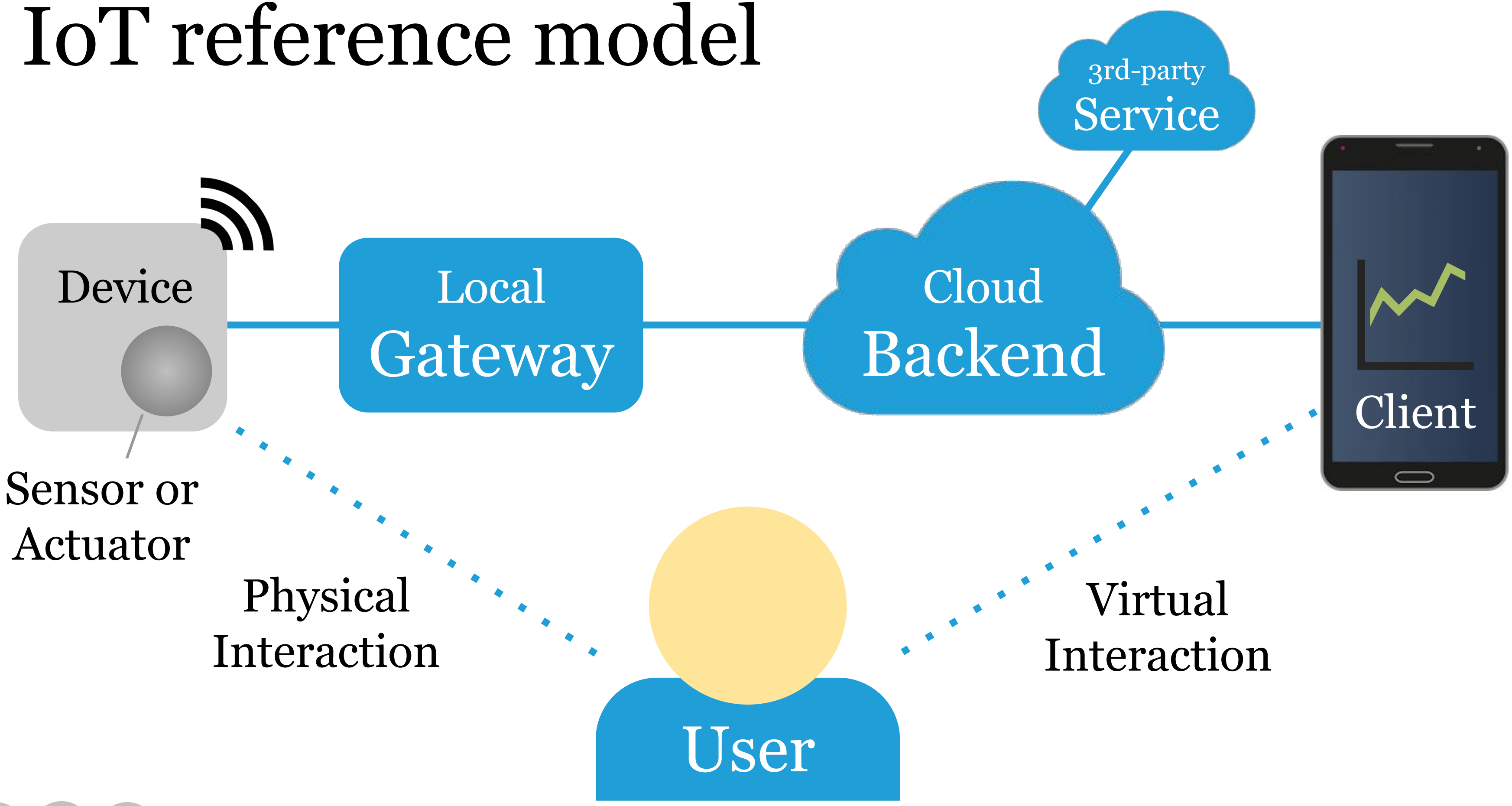
Check the Wiki entry on Installing the Arduino IDE.

Set up the Feather nRF52840 Express for Arduino.

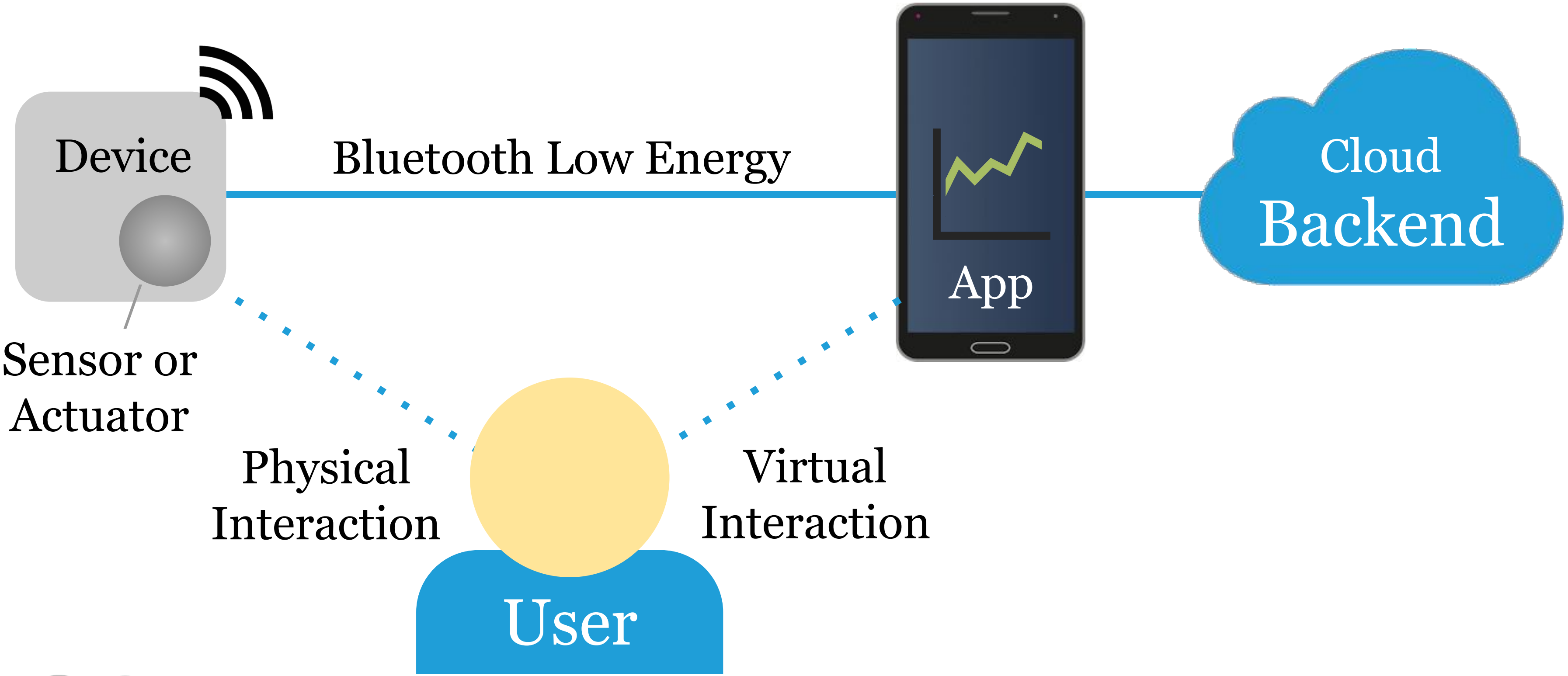
Setting up the board also installs this BLE library.

For testing, a smartphone with BLE is required.

IoT reference model



BLE reference model



Bluetooth Low Energy (BLE)

BLE is a power-efficient Bluetooth variant (since 4.0).

BLE is well suited for small, battery powered devices.

It uses less energy than Wi-Fi and way less than 4/5G.

Range is ~30 m, data rate 1 Mbps, frequency 2.4 GHz.

The standard is maintained by the Bluetooth SIG.

How BLE works

Peripherals advertise the data they have, over the air.

Centrals scan for nearby peripherals to discover them.

The central connects to a peripheral and uses its data.

Data is structured into *services* and *characteristics*.

BLE protocol stack

Application — application specific code and formats

BLE library — thin, language-specific wrapper library

GATT — services & characteristics | GAP — discovery

ATT — attribute transport | SMP — security manager

L2CAP — logical link control and adaptation protocol

Link layer — exposed via the host controller interface

Physical layer — dealing with actual radio signals

Generic Access Profile (GAP)

GAP defines the following roles, communication types:

Broadcaster and *observer* (connectionless, one-way).

Peripheral and *central* (bidirectional connection).

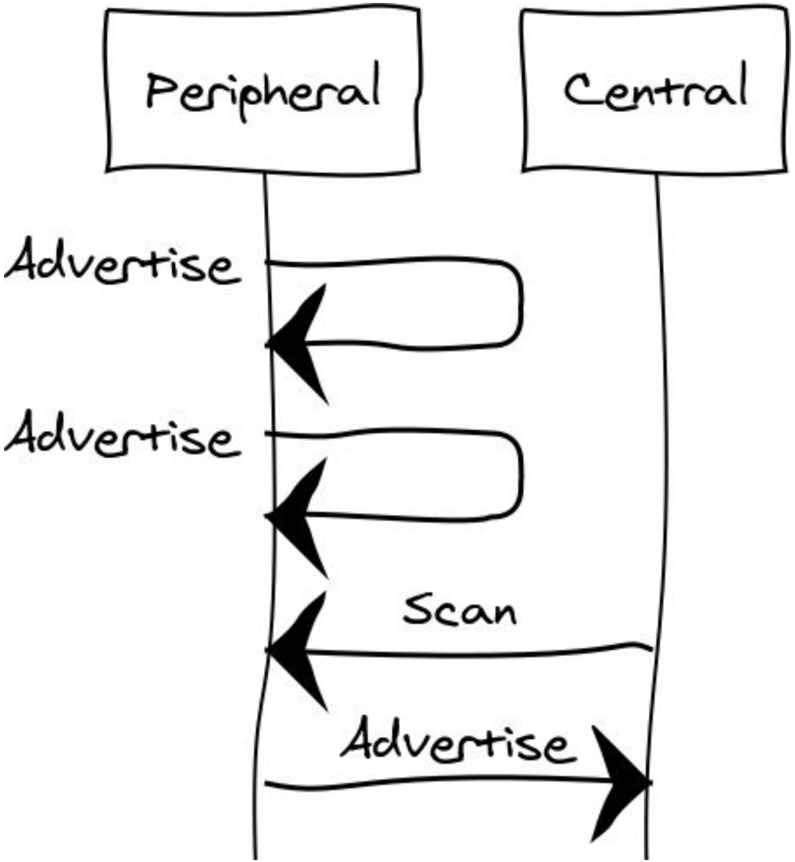
Each device supports one or more of these roles.

We start with peripheral and central roles.

Advertising

A peripheral *advertises* its services by broadcast, in a regular interval.

A central *scans* for all or a subset of services and gets device addresses and, if it's been sent, advertised data.



Attribute Transport (ATT)

ATT allows a *client* to access attributes on a *server*.

An *attribute* has a handle, a UUID and permissions.

An *attribute handle* is a server-assigned, 16-bit ID.

A *UUID* is a 16/128-bit universally unique identifier.

Permissions allow you to read, write or get notified.

See Bluetooth spec v5.1, Part F, p. 2288.

Generic Attribute Profile (GATT)

GATT is a simple application level protocol for BLE.

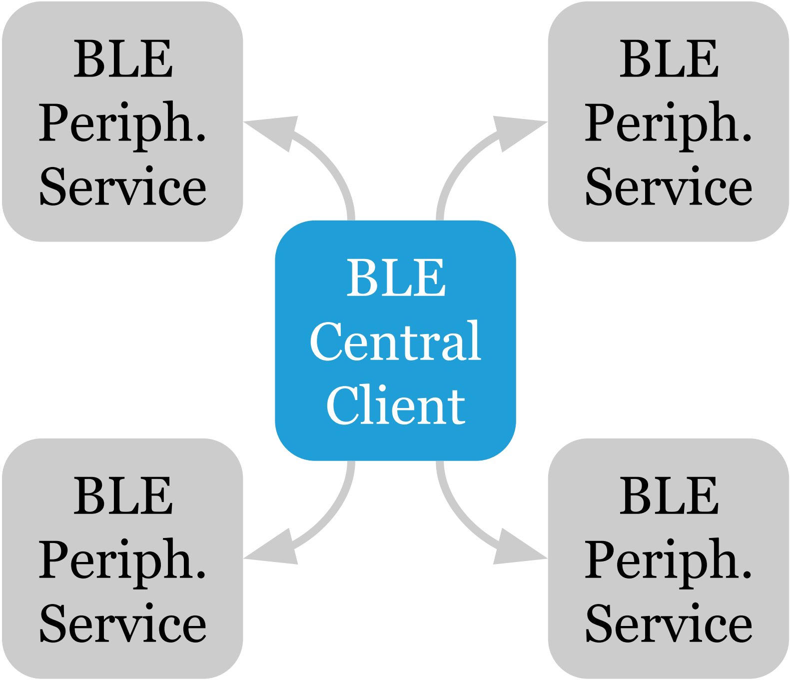
It's connection-based, with a *client* and a *server* role.

This enables a BLE device to provide a RESTful API.

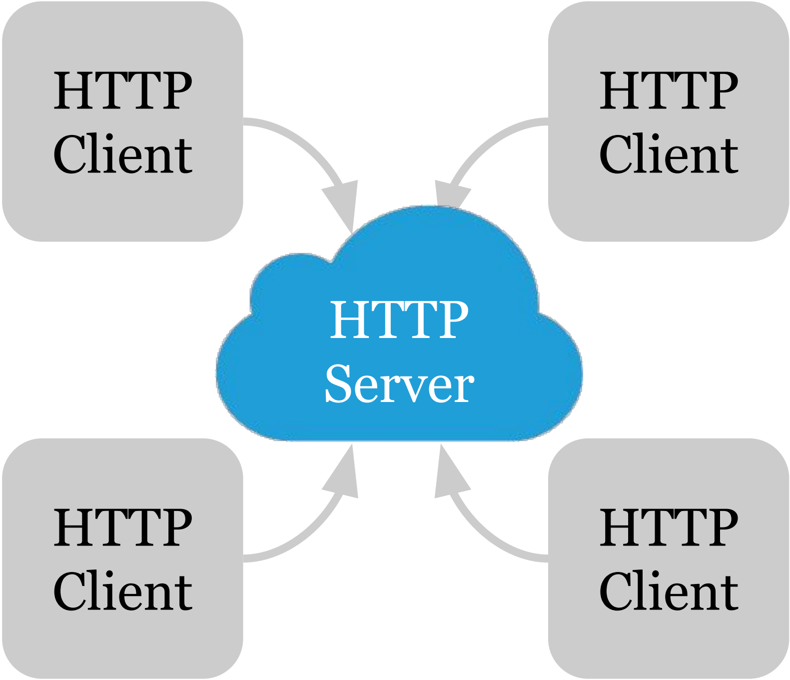
A "GATT API", or *profile*, is a collection of *services*.

Usually, the peripheral acts as a server.

BLE


BLE Periph. Service
BLE Periph. Service
BLE Central Client
BLE Periph. Service
BLE Periph. Service

HTTP


HTTP Client
HTTP Client
HTTP Server
HTTP Client
HTTP Client

Services

A GATT service is a collection of characteristics.

Services encapsulate the behavior of part of a device.

In addition, such a service can refer to other services.

There are standard and custom services and profiles.

E.g. the Battery Service or the Heart Rate Service.

Characteristics

A GATT characteristic has a value and descriptors.

A *value* encodes data "bits" that form a logical unit.

Descriptors are defined attributes of a characteristic.

Supported procedures: read, write and notifications.

E.g. a Battery Level or a Heart Rate Measurement.

Descriptors

A GATT descriptor describes a characteristic value.

E.g. Presentation Format or Valid Range descriptor.

Descriptors also allow to configure characteristics.

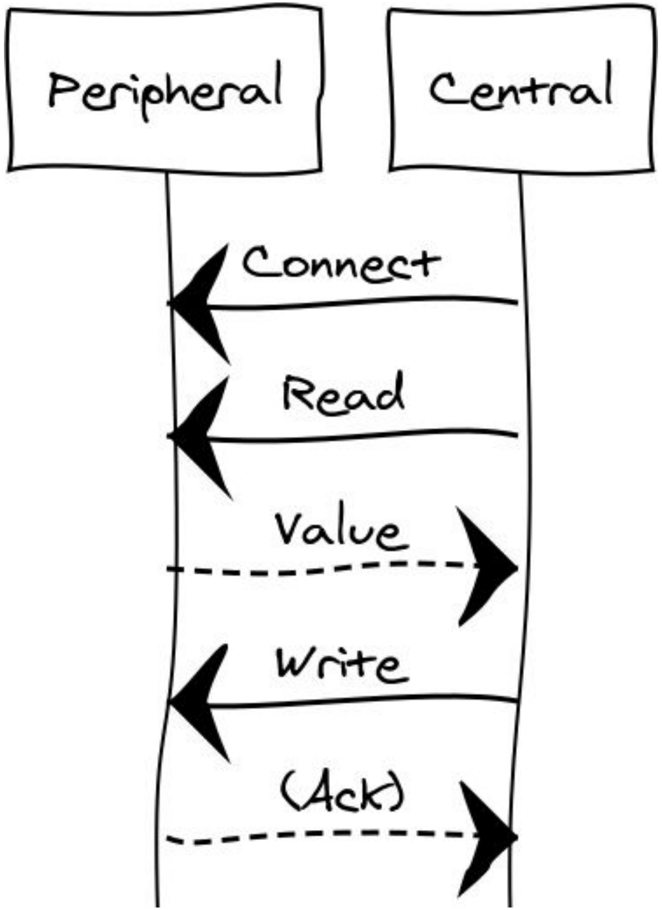
E.g. Client Characteristic Configuration descriptor allows a client to enable or disable notifications.

Read and write

Connect = the central connects to a peripherals BLE address.

Read = value of a characteristic or its descriptors is returned.

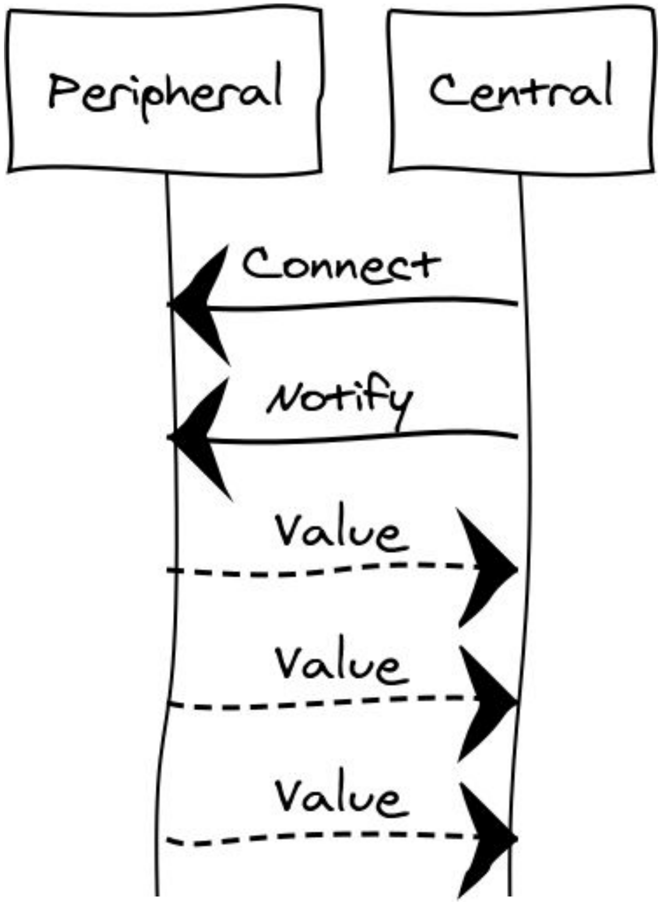
Write = characteristic value, or characteristic descriptor value is set, with/without response.



Notifications

Notify = *Client Characteristic Configuration* descriptor of a characteristic, UUID `0x2902`, is set to `0x0001` using *Write*.

Value = A *Handle Value Notification* is sent if value changes.



See Bluetooth spec v5.1, p. 2360 and p. 2389.

BLE explorer apps

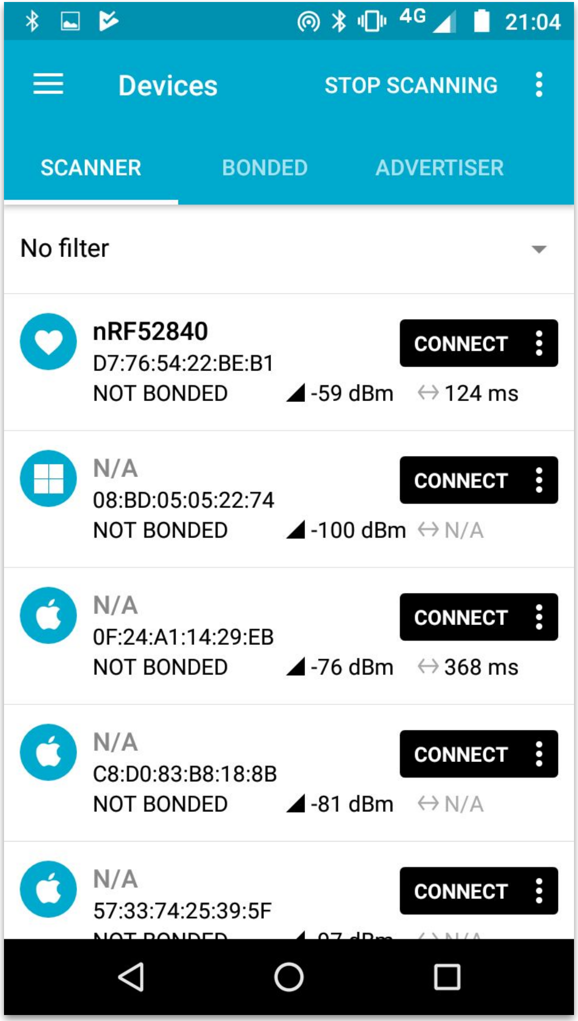
For debugging, use any generic BLE explorer app:

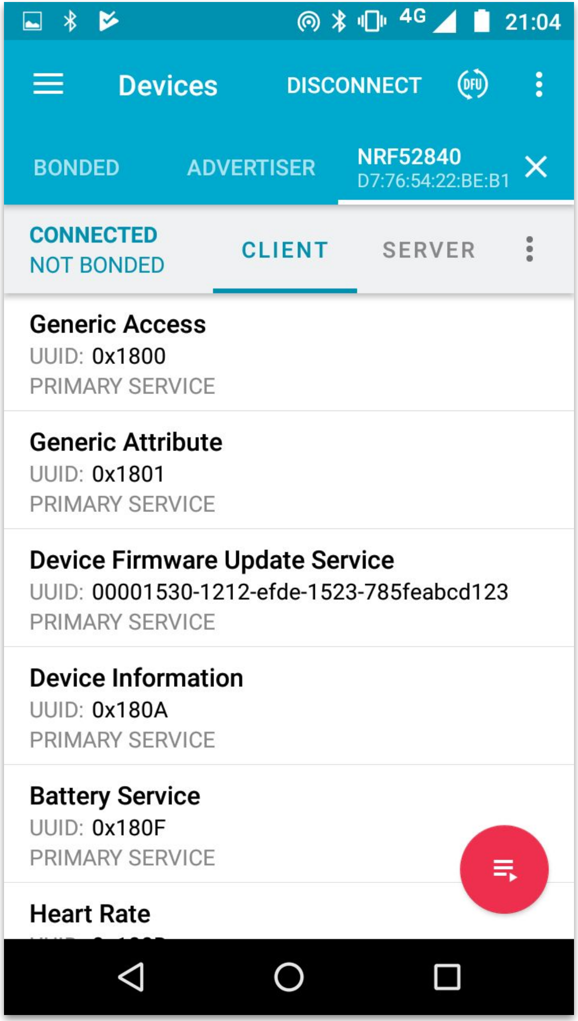
Find BLE explorer apps on the Google Play Store.

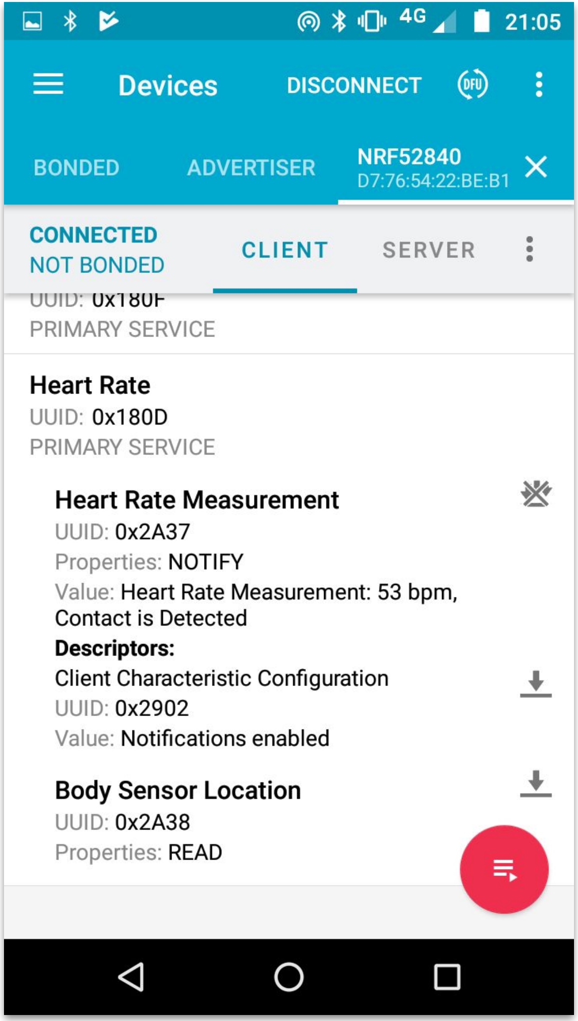
Search for "BLE explorer" in the iOS App Store.

Smartphones can act as central or peripheral.

Exploring is a great way to learn about BLE.


21:04
Devices
STOP SCANNING
SCANNER
BONDED
ADVERTISER
No filter
nRF52840
D7:76:54:22:BE:B1
NOT BONDED
-59 dBm
124 ms
CONNECT
N/A
08:BD:05:05:22:74
NOT BONDED
-100 dBm
N/A
CONNECT
N/A
0F:24:A1:14:29:EB
NOT BONDED
-76 dBm
368 ms
CONNECT
N/A
C8:D0:83:B8:18:8B
NOT BONDED
-81 dBm
N/A
CONNECT
N/A
57:33:74:25:39:5F
CONNECT


21:04
Devices
DISCONNECT
BONDED
ADVERTISER
NRF52840
D7:76:54:22:BE:B1
CONNECTED
NOT BONDED
CLIENT
SERVER
Generic Access
UUID: 0x1800
PRIMARY SERVICE
Generic Attribute
UUID: 0x1801
PRIMARY SERVICE
Device Firmware Update Service
UUID: 00001530-1212-efde-1523-785feabcd123
PRIMARY SERVICE
Device Information
UUID: 0x180A
PRIMARY SERVICE
Battery Service
UUID: 0x180F
PRIMARY SERVICE
Heart Rate


21:05
Devices
DISCONNECT
BONDED
ADVERTISER
NRF52840
D7:76:54:22:BE:B1
CONNECTED
NOT BONDED
CLIENT
SERVER
PRIMARY SERVICE
Heart Rate
UUID: 0x180D
PRIMARY SERVICE
Heart Rate Measurement
UUID: 0x2A37
Properties: NOTIFY
Value: Heart Rate Measurement: 53 bpm, Contact is Detected
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Notifications enabled
Body Sensor Location
UUID: 0x2A38
Properties: READ

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]*
```

*See also Heart Rate Service specification.

nRF52840 HRM BLE peripheral .ino

```
hrmSvc = BLEService(0x180D); // See HRM spec
hrmChr = BLECharacteristic(0x2A37); // See spec

hrmSvc.begin(); // to add characteristics
hrmChr.setProperties(CHR_PROPS_NOTIFY); …
hrmChr.begin(); // adds characteristic

uint8_t hrmData[2] = { 0b00000110, value };
hrmChr.notify(hrmData, sizeof(hrmData));
```

Hands-on, 10': HRM BLE peripheral

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Explore the HRM example using a smartphone app*.

Try to enable notifications to get value updates.

*Try nRF Connect for Android or iOS.

Nordic UART service

This service provides a serial connection over BLE:

`Nordic UART Service` `custom (128-bit) UUID:`
`0x6E400001-B5A3-F393-E0A9-E50E24DCCA9E`

This service includes the following characteristics:

```
RX (device receives data)  UUID: 0x0002 [W]
TX (device transmits data) UUID: 0x0003 [N]
```

This service is becoming a de facto standard.

nRF52840 UART BLE peripheral .ino

```
// UUID: 6E400001-B5A3-F393-E0A9-E50E24DCCA9E
uint8_t const uartSvcUuid[] = { 0x9E, 0xCA, …,
0xB5, 0x01, 0x00, 0x40, 0x6E }; // lsb first

uartSvc = BLEService(uartSvcUuid); // 128-bit
rxChr = BLECharacteristic(rxChrUuid); // 128-b.
txChr = BLECharacteristic(txChrUuid); // 128-b.

txChar.setProperties(CHR_PROPS_NOTIFY);
rxChar.setProperties(CHR_PROPS_WRITE);
```

Hands-on, 10': UART BLE peripheral

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Write bytes to *RX* with a generic BLE explorer app.

Check the serial monitor to see the received bytes*.

*Why do some bytes not show up?

nRF52840 UART BLE central .ino

```
Bluefruit.begin(0, 1); // 1 central connection
uartSvcClient.begin();
uartSvcClient.setRxCallback(rxCbck); // read
Bluefruit.Central.setConnectCallback(connCbck);

void connCbck(uint16_t connHandle) {
  if (uartSvcClient.discover(connHandle)) {
    uartSvcClient.enableTXD(); // enable notify
    uartServiceClient.print(…); // write data
    … } }
```

nRF52840 UART BLE central (ff.) .ino

```
Bluefruit.Scanner.setRxCallback(found);

void found(ble_gap_evt_adv_report_t* report) {
  if (….Scanner.checkReportForService(
    report, uartServiceClient)) {
    Bluefruit.Central.connect(report);
  } else {
    Bluefruit.Scanner.resume();
  }
}
```

Hands-on, 10': UART BLE central

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Open the Arduino serial monitor to enter a message.

Use a second nRF52840 as a UART peripheral.

Beacons

Beacons, e.g. Apple iBeacon are *broadcaster* devices.

Any *observer* can read the data which they advertise.

Lookup of "what a beacon means" requires an app.

Except for Physical Web / Eddystone beacons.

These contain an URL to be used right away.

nRF52840 beacon BLE observable .ino

```
BLEBeacon beacon(
  beaconUuid, // AirLocate UUID
  beaconMajorVersion,
  beaconMinorVersion,
  rssiAtOneMeter);
beacon.setManufacturer(0x004C); // Apple
startAdvertising();

suspendLoop(); // save power
```

Hands-on, 10': Beacons

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to find the source code.

Test the beacon with a dedicated iOS/Android app.

Which information is transferred by a beacon?

Start a scan in a public place, e.g. Zürich HB.

nRF52840 scanner BLE central .ino

```
Bluefruit.begin(0, 1); // Central
Bluefruit.Scanner.setRxCallback(found);
Bluefruit.Scanner.start(0);

void found(ble_gap_evt_adv_report_t* report) {
  Serial.printBufferReverse( // little endian
    report->peer_addr.addr, 6, ':');
  if (Bluefruit.Scanner.checkReportForUuid(…))…
  Bluefruit.Scanner.resume();
}
```

Hands-on, 10': Scanner BLE Central

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to find the source code.

Add a *checkReportForUuid()* for the Battery Service.

Can you spot the UUID in the advertising data?

Consider working in teams => more nRF52840.

Security

BLE has security mechanisms for pairing and more.

Pairing: exchanging identity and keys to set up trust.

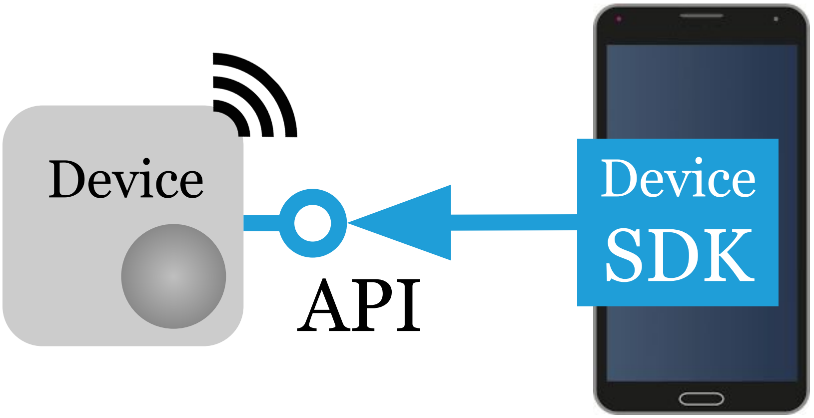
Device chooses *Just Works*, *Passkey Entry* or OOB.

Or numeric comparison and ECDH for key exchange.

Some apps add encryption on the application layer.

Device API vs. SDK

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




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

A platform specific *device SDK* simplifies integration.

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

BLE on Android

There is an official introduction to BLE on Android.

Building a robust BLE app on Android can be tricky.

Use the Nordic Semiconductor Android-BLE-Library.

As an example app, look at the nRF Toolbox project.

Writing a plugin for nRF Toolbox is a good start.

BLE on iOS

On iOS the official BLE library is Core Bluetooth.

Its documentation is a great introduction to BLE.

In iOS there's no way to get a device BLE address.

Instead, a UUID is assigned, as a handle, by iOS.

iOS devices change their Bluetooth MAC address.

BLE on Raspberry Pi

On Raspberry Pi Zero W there are many options, e.g.

Node.js libraries: Noble (central), Bleno (peripheral)

Python library: PyBluez, BluePy

Linux C library: Bluez

CLI: bluetoothctl

Summary

BLE provides low power, personal area connectivity.

A BLE central scans for peripherals, who advertise.

Each BLE peripheral provides one or more services.

Services allow to read/write characteristic values.

Descriptors allow to configure notifications.

Next: Raspberry Pi as a Local Gateway.

Challenge

Design and implement an API for the DHT11 sensor.

Create UUIDs for your service and its characteristics.

Allow the central to read temperature and humidity.

Chose a data format that fits the sensor value range.

Test your peripheral with a generic BLE explorer.

Done? Consider adding support for notifications.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.