Hes·so VALAIS WALLIS
School of Engineering

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

nRF 5 – Programming cheatsheet

1. Bluetooth – Theory

https://www.novelbits.io/basics-bluetooth-low-energy/

1.1. Introduction

What are the low energy requirements?

Low Power

- Radio off longer
- Short burst of data
- Low transfer speeds

Bluetooth Low Energy (BLE) is a low power wireless technology used for connecting devices with each other. BLE operates in the 2.4 GHz ISM (Industrial, Scientific, and Medical) band, and is targeted towards applications that need to consume less power and may need to run on batteries for longer periods of time—months, and even years.

Bluetooth LE vs Bluetooth Classic

Bluetooth LE	**Bluetooth Classic**
Low-bandwidth applications (sensor data, control of devices)	Streaming applications (audio, files transfert)
Low power, low duty data cycles	Higher data rate
40 RF channels	79 RF channels
Quick Connections (discovery on 3 channels).	Slow Connections (discovery on 32 channels)

BLE use frequency hopping to counteract narrowband interference problems.

Benefits and limitations of BLE

- ✓ Low power consumption, by keeping the radio off as much as possible and sending small amounts of data at low transfer speeds.
- ✓ Free access to officials specifications documents
- ✓ Cheap modules and chipsets
- ✓ Already equipped in most smartphones in the market
- ✗ Design for short range application, there are a few factors that limit the range of BLE:
  - 2.4 GHz ISM spectrum is greatly affected by obstacle
  - Performance and design of the antenna
  - Physical enclosure of the device
  - Device orientation

1.2. Bluetooth Low Energy Architecture

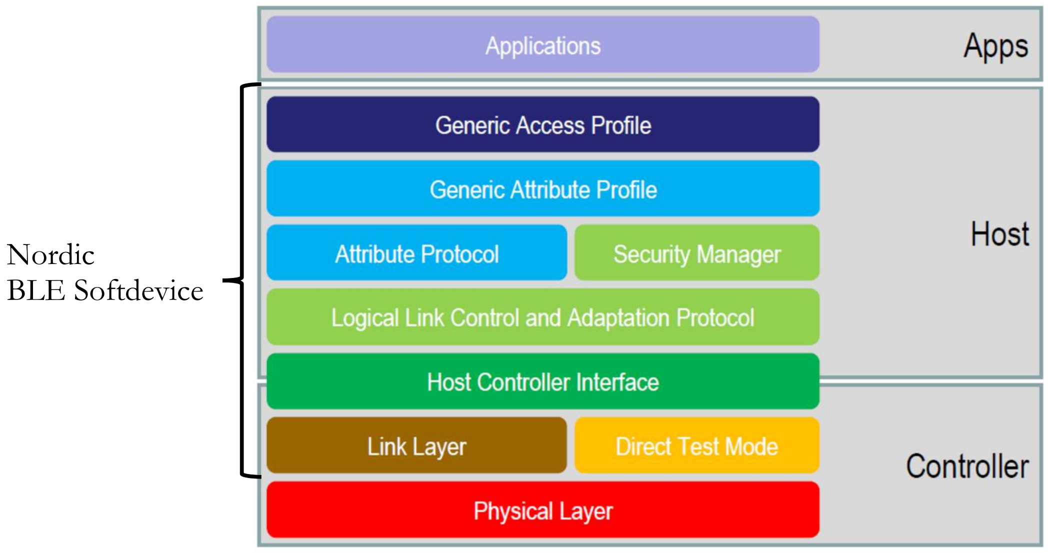


Figure 1: BLE Architecture

The Soft device implement all the layer between the Appliction and the Physical layer. As a developer you won't have to worry much about the layers below the Security Manager and Attribute Protocol. But lets at least cover the definitions of these layers.

- **The physical layer (PHY)** refers to the physical radio used for communication and for modulating/demodulating the data. It operates in the ISM band (2.4 GHz spectrum).
- **The Link Layer** is the layer that interfaces with the Physical Layer (Radio) and provides the higher levels an abstraction and a way to interact with the radio (through an intermediary level called the HCI layer). It is responsible for managing the state of the radio as well as the timing requirements for adhering to the Bluetooth Low Energy specification.
- **Direct Test Mode**: the purpose of this mode is to test the operation of the radio at the physical level (such as transmission power, receiver sensitivity, etc.).
- **The Host Controller Interface (HCI) layer** is a standard protocol defined by the Bluetooth specification that allows the Host layer to communicate with the Controller layer. These layers could exist on separate chips, or they could exist on the same chip.
- **The Logical Link Control and Adaptation Protocol (L2CAP) layer** acts as a protocol multiplexing layer. It takes multiple protocols from the upper layers and places them in standard BLE packets that are passed down to the lower layers beneath it.

Controller layer

In addition of the Bluetooth Controller provided by the Softdevice, we can use the Zephyr RTOS.

1.3. The Generic Access Profile (GAP)

The GAP provides a framework that defines how BLE devices interact with each other. This includes:

- Roles of BLE devices
- Advertisements (Broadcasting, Discovery, Advertisement parameters, Advertisement data)
- Connection establishment (initiating connections, accepting connections, Connection parameters)
- Security

GAP topology

4 distinct roles of BLE usage:

- Connection-oriented
  - **Central** Devices: phones or PC's with a higher CPU processing power.
  - **Peripheral** Devices: Sensors or low power devices, which connect to the central device.
- Connection-less roles
  - **Broadcaster**: sending out BLE advertisements, e.g. a smart beacon
  - **Observer**: scanning for BLE advertisements

A single device may operate in multiple Roles at the same time. For example, your smartphone can operate in the Central role when communicating with your smartwatch, and also act in the Peripheral role while communicating with a PC.

Advertising

In the Advertising state, a device sends out packets containing useful data for others to receive and process. The packets are sent at a fixed interval defined as the Advertising Interval.
Devices can advertise for 4 reasons (PDU type):

- Advertising Indications, requests connection to any central device (ADV_IND)
- Connection request is directed at a specific central device (ADV_DIRECT_IND)
- Non connectable devise, broadcast information like a beacon (ADV_NONCONN_IND)
- Broadcast, additional information are available via scan response (ADV_SCAN_IND)

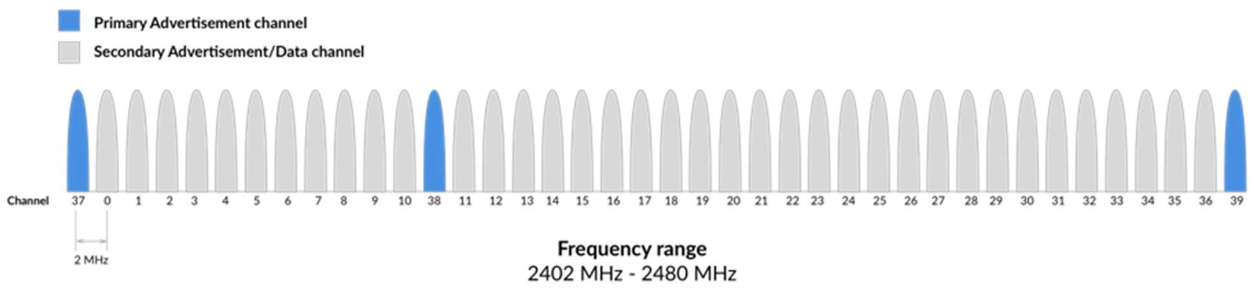


Figure 2: RF Channels

Advertisements always start with Advertisement Packets sent on the 3 Primary Advertising Channels This allows Centrals to find the Advertising device (Peripheral or Broadcaster) and parse its Advertising packets. The Central can then Initiate a Connection if the Advertiser allows it (Peripheral devices). The remaining 37 channels are used for Data Packet transfer during a Connection.

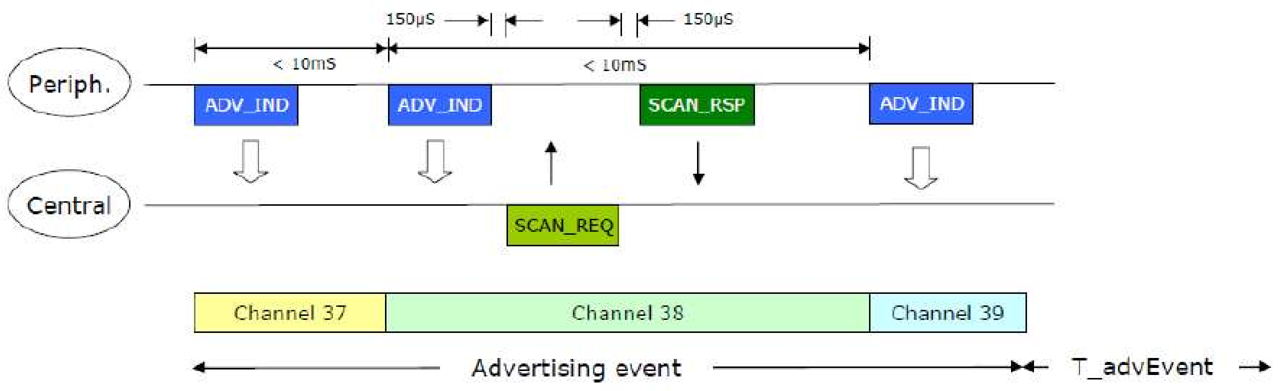


Figure 3: Advertising events

ADVERTISING DATA TYPE

Advertising packet can consist of no more than 31 bytes, so you might need to select your data with care:

- Flags
- Service UUIDs
- Local Name
- TX Power Level
- Slave Connection Interval Range
- Signed Data
- Service Solicitation
- Service Data
- Manufacturer Specific Data (company UUID for SIG members)
- Appearance
- Others advertising data types

SCAN RESPONSE DATA

For advertising more than 31 bytes we can use scan response data, this is an optional "secondary" advertising payload which allows scanning devices that detect an advertising device to request a second advertising packet. BLE packet type

Connection

In order for two BLE devices to connect to each other, the following needs to happen:

1. The **Peripheral** needs to **start Advertising** and send out Connectable Advertisement packets.
2. The **Central** device needs to be **Scanning** for Advertisements while the Peripheral is Advertising.
3. If the Central happens to be listening on an Advertising Channel that the Peripheral is Advertising on, then the Central **device discovers** the Peripheral and is able to read the Advertisement packet and all the necessary information in order to establish a Connection.
4. The Central then sends a **Connection Request** packet.
5. The peripheral always listens for a short interval on the same Advertising Channel after it sends out the Advertising packet. This allows it to receive the Connection Request packet from the Central device — which triggers the forming of the **Connection** between the two devices.

After that, the Connection is considered "created", but not yet "established". A Connection is considered "established" once the device receives a packet from its peer device.

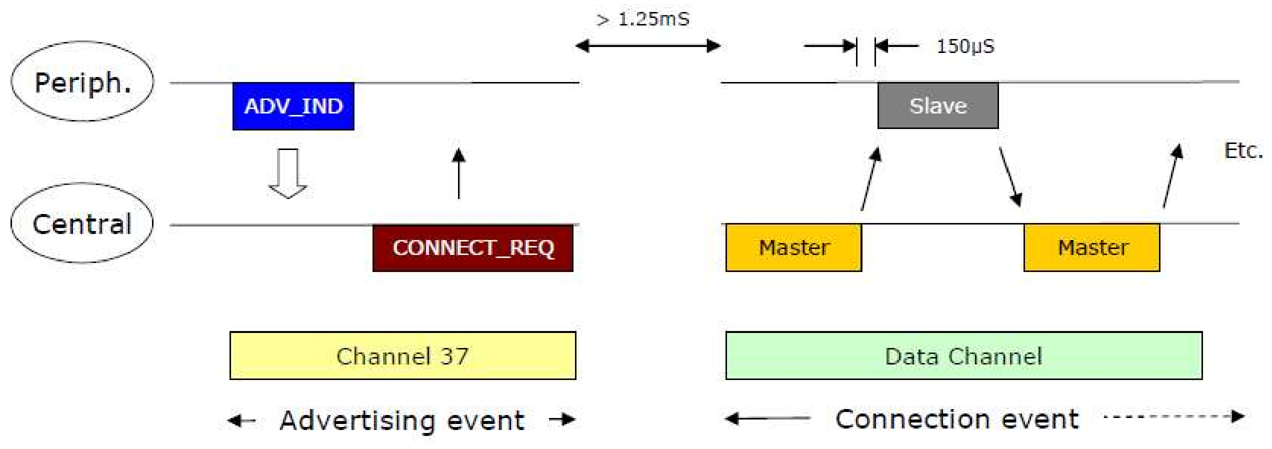


Figure 4: Data Transaction

Once a connection is made:

- Master informs slave of hopping sequence and when to wake up
- All subsequent transactions are performed in the 37 data channels
- Transactions can be encrypted
- Both devices can go into deep sleep between transactions

CONNECTION EVENTS

During a Connection Event, the Master and Slave alternate sending data packets to each other until neither side has data to send. Here are a few aspects of Connections that are very important to know:

- A Connection Event contains at least one packet sent by the Master (central).

- The Slave(peripheral) always sends a packet back if it received a packet from the Master.
- If the Master does not receive a packet back from the Slave, the Master will close the Connection Event — it resumes sending packets at the next Connection Event.
- The Connection Event can be closed by either side.

CONNECTION PARAMETERS

The most important parameters that define a Connection include:

- **Connection Interval**: the interval at which two connected BLE devices wake up the radio and exchange data (at each Connection Event).
- **Slave Latency**: this value allows the Peripheral to skip a number of consecutive Connection Events and not listen to the Central at these Connection Events without compromising the Connection.
- **Supervision Timeout**: the maximum time between two received data packets before the Connection is considered lost.

GAP service

The final item that GAP includes as part of its section in the core specification is the GAP Service, a mandatory GATT service that every device must include among its attributes. The service is freely accessible(read-only) to all connected devices with no security requirements whatsoever, and it contains the following three characteristics:

- **Device Name characteristic**
- **Appearance characteristic:** generic category, typically used by the GATT client to display an icon that represents the given category. This characteristic can also be made available in the advertising packet with the Appearance AD Type.
- **Peripheral Preferred Connection Parameters** (PPCP) **characteristic**
- **Central Address Resolution characteristic**

Security

Words to know:

- **Connecting** is the act of establishing a communication link. No pairing or bonding is required to communicate over Bluetooth LE.
- **Pairing** is the act of exchanging keys after connection, typically to set up and maintain an encrypted connection. Or, in the words of the
- **Bonding** is the act of storing the exchanged keys after pairing, typically to re-establish an encrypted connection without needing to exchange these keys again.

GAP also defines authentication, encryption or signing for security establishment. In addition to these you can choose the device address between:

- **Public:** address permanently programmed into a device.
- **Static:** random address, can be regenerated only after power cycle.
- **Random Private Resolvable**: generated at any time, and requiring an *Identity Resolving Key (IRK)* to be resolve. Avoid the device to be identified and tracked by an unknown scanning device. Used By iOS and Android
- **Random Private Non-Resolvable**: Shared between bonded devices for use during reconnection. This changes with each connection.

1.4. Security Manager Protocol (SMP)

SMP is used by Bluetooth Low Energy implementations for pairing and transport specific key distribution.
In the Nodic ble Softdevice, the SDK module **Peer manager** implement connection and security

1.5. The Generic Attribute Profile (GATT)

The Generic Attribute Profile (GATT) defines the **format of the data (device tree - hierarchical data structure)** exposed by a BLE device. It also defines the **procedures** needed to **access** the data exposed by a device.

There are two Roles within GATT: **Server** and **Client**. The Server is the device that exposes the data it controls or contains, and possibly some other aspects of its behaviour that other devices may be able to control. BLE device can act as the Server and a Client at the same time.

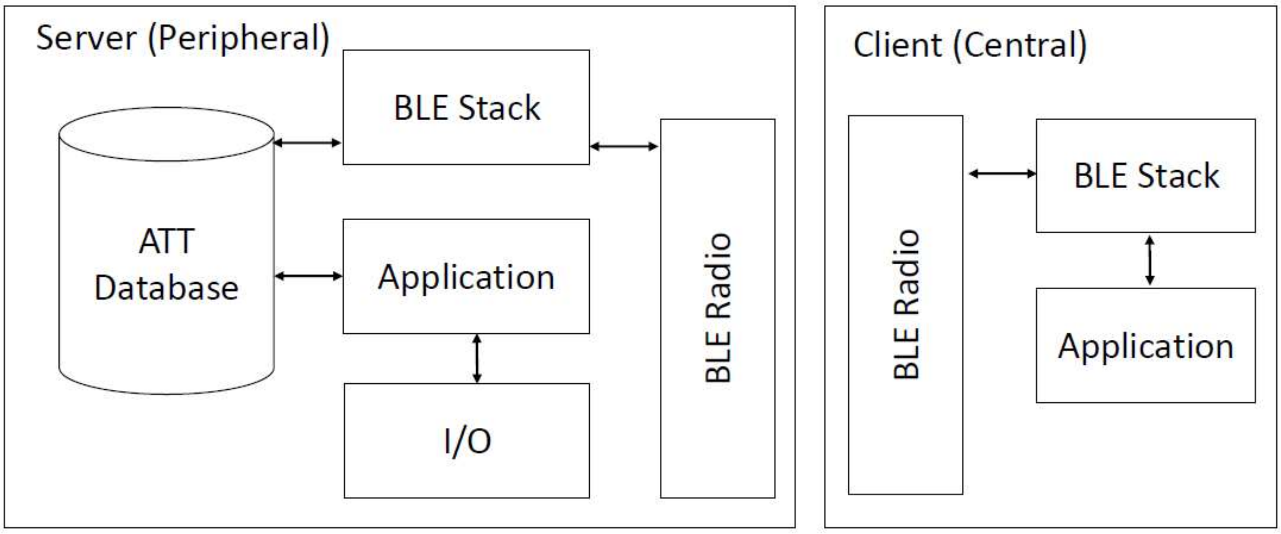


Figure 5: ATT Server - Client

Services and Characteristics

- **Attributes**: a generic term for any type of data exposed by the Server and defines the structure of this data. For example, Services and Characteristics are types of Attributes.
- **Characteristics**: a Characteristic is always part of a Service and it represents a piece of information/data that a Server wants to expose to a client. For example, the Battery Level Characteristic represents the remaining power level of a battery in a device which can be read by a Client.
- **Services**: a grouping of one or more Attributes (some of which are Characteristics by functionality.
- **Profiles**: Profiles are much broader in definition from Services. They are concerned with defining the behavior of both the Client and Server when it comes to Services, Characteristics and even Connections and security requirements.
  Services and their specifications, on the other hand, deal with the implementation of these Services and Characteristics on the Server side only.

In BLE, there are six types of operations on Characteristics:

1. **Commands**: sent by the Client to the Server and do not require a Response (defined below). One example of a Command is a **Write Command**, which does not require a Response from the Server.
2. **Requests**: sent by the Client to the Server and require a Response. Some examples of Requests include: Read Requests and Write Requests.
3. **Responses**: sent by the Server in response to a Request.
4. **Notifications**: sent by the Server to the Client to let the Client know that a specific Characteristic Value has changed. In order for this to be triggered and sent by the Server, the Client has to enable Notifications for the Characteristic of interest. Note that a Notification does not require a Response from the Client to acknowledge its receipt.

5. **Indications**: sent by the Server to the Client. They are very similar to Notifications but require an acknowledgment to be sent back from the Client to let the Server know that the Indication was successfully received.
   Note: Notifications and Indications are exposed via the Client Characteristic Configuration Descriptor (**CCCD**) Attribute. Writing a "1" to this Attribute value enables Notifications, whereas writing a "2" enables Indications. Writing a "0" disables both Notifications and Indications.
6. **Confirmations**: sent by the Client to the Server. These are the acknowledgment packets sent back to the Server to let it know that the Client received an Indication successfully.

Attribute

Each attribute in the table contain:

- a type, defined by a UUID
- a Handle
- Permissions
- a value

Universally Unique ID (UUID)

Unique number used to identify attributes. These IDs are transmitted over the air so that i.e. a peripheral can inform a central what services it provides. To save transmitting airtime and memory space in your nRF52 there are two kinds of UUIDs:

- short **16-bit UUID**, energy and memory efficient, but like there is limited number of unique IDs there is a rule; you can only transmit the predefined **Bluetooth SIG** UUIDs directly over the air.
  Note: *Bluetooth Special Interest Group (SIG) has predefined attribute UUIDs.*
- **128-bit UUID**, "custom services and characteristics - **vendor specific**" It looks something like this: 4A98xxxx-1CC4-E7C1-C757-F1267DD021E8 and is called the "base UUID". The four x's represent a field where you will insert your own 16-bit IDs for your custom services and characteristics and use them just like a predefined UUID. This way you can store the base UUID once in memory, forget about it, and work with 16-bit IDs as normal.

Handle

The attribute handle uniquely identifies an attribute on a server, it can be considered as the row number in the attribute table.

Attribute Permissions

Apply to the attribute value, allows a client to look through a server's attribute table and discover what the server can provide.

- **Service Declaration**: Always: Read Only, No Authentication, No Authorization required
- **Characteristic Declaration**: Always: Read Only, No Authentication, No Authorization required
- **Characteristic Value Declaration** and **Descriptor Declaration** permissions can be choose:
  - Access Permissions
    - None
    - Read/Write only
    - Read and Write
  - Encryption
    - None
    - Encryption required
    - Authenticated encryption required
  - Authorization
    - None
    - Authorization required

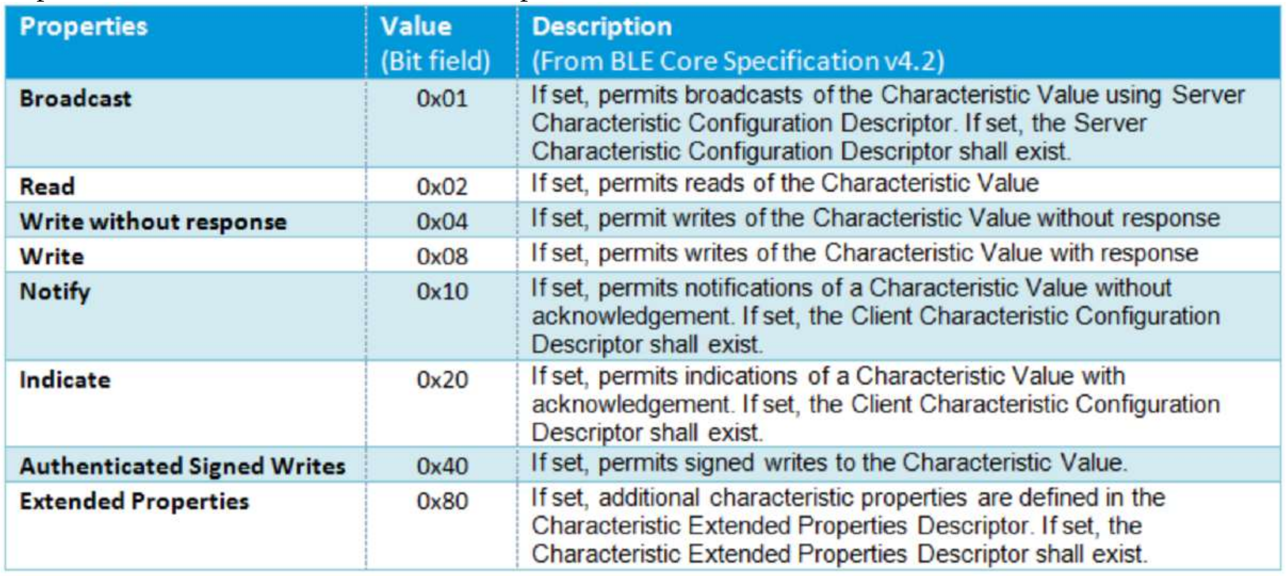
Attribute Values

Each attribute type has a different value:

- **Service Declaration:** contain a UUID descripting what kind of service it is.
  Holds a value or information about where to find other attributes and their properties.
- **Characteristic Declaration:** holds information about the subsequent Characteristic Value Declaration (Properties, Handle, and Type)
- **Characteristic Value Declaration:** contain the value
- **Descriptor Declaration:** attribute with additional information about the characteristic (CCCD, etc...)

Attribute properties

Affect the characteristic Value Declaration, it has a large impact on transmission speed, i.e. write without response is 2 times faster as write with response.

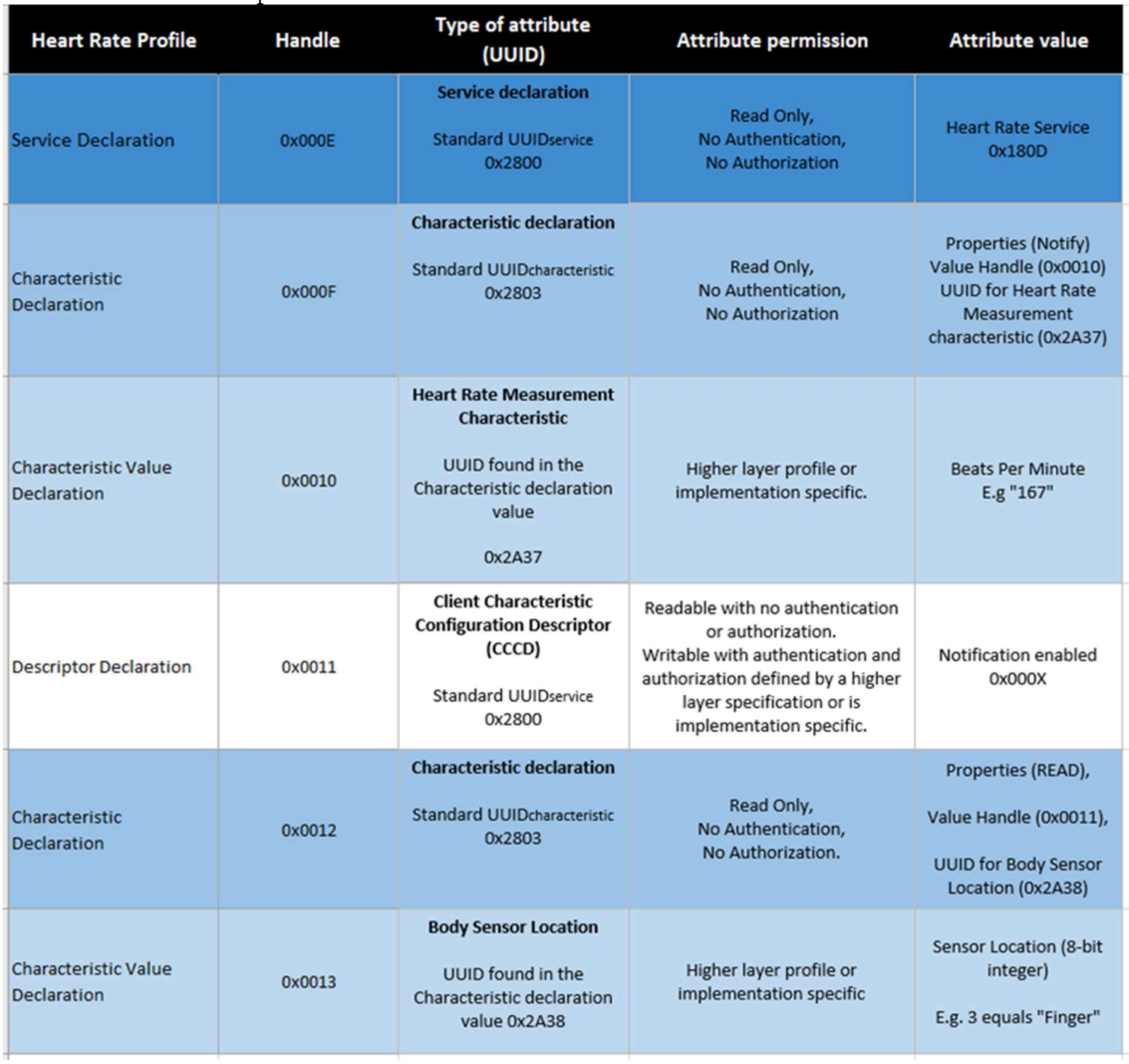


Properties	Value (Bit field)	Description (From BLE Core Specification v4.2)
Broadcast	0x01	If set, permits broadcasts of the Characteristic Value using Server Characteristic Configuration Descriptor. If set, the Server Characteristic Configuration Descriptor shall exist.
Read	0x02	If set, permits reads of the Characteristic Value
Write without response	0x04	If set, permit writes of the Characteristic Value without response
Write	0x08	If set, permits writes of the Characteristic Value with response
Notify	0x10	If set, permits notifications of a Characteristic Value without acknowledgement. If set, the Client Characteristic Configuration Descriptor shall exist.
Indicate	0x20	If set, permits indications of a Characteristic Value with acknowledgement. If set, the Client Characteristic Configuration Descriptor shall exist.
Authenticated Signed Writes	0x40	If set, permits signed writes to the Characteristic Value.
Extended Properties	0x80	If set, additional characteristic properties are defined in the Characteristic Extended Properties Descriptor. If set, the Characteristic Extended Properties Descriptor shall exist.

GATT – Exemple

This example about Heart Rate Profile, show use the use of default Bluetooth SIG UUIDs:

- 0x2800 “Service Declaration”,
- 0x2803 “Characteristic Declaration”
- 0x2902 “Descriptor Declaration”



Heart Rate Profile	Handle	Type of attribute (UUID)	Attribute permission	Attribute value
Service Declaration	0x000E	**Service declaration** Standard UUID$_{service}$ 0x2800	Read Only, No Authentication, No Authorization	Heart Rate Service 0x180D
Characteristic Declaration	0x000F	**Characteristic declaration** Standard UUID$_{characteristic}$ 0x2803	Read Only, No Authentication, No Authorization	Properties (Notify) Value Handle (0x0010) UUID for Heart Rate Measurement characteristic (0x2A37)
Characteristic Value Declaration	0x0010	**Heart Rate Measurement Characteristic** UUID found in the Characteristic declaration value 0x2A37	Higher layer profile or implementation specific.	Beats Per Minute E.g "167"
Descriptor Declaration	0x0011	**Client Characteristic Configuration Descriptor (CCCD)** Standard UUID$_{service}$ 0x2800	Readable with no authentication or authorization. Writable with authentication and authorization defined by a higher layer specification or is implementation specific.	Notification enabled 0x000X
Characteristic Declaration	0x0012	**Characteristic declaration** Standard UUID$_{characteristic}$ 0x2803	Read Only, No Authentication, No Authorization.	Properties (READ), Value Handle (0x0011), UUID for Body Sensor Location (0x2A38)
Characteristic Value Declaration	0x0013	**Body Sensor Location** UUID found in the Characteristic declaration value 0x2A38	Higher layer profile or implementation specific	Sensor Location (8-bit integer) E.g. 3 equals "Finger"

2. nRF52840 Development Kit

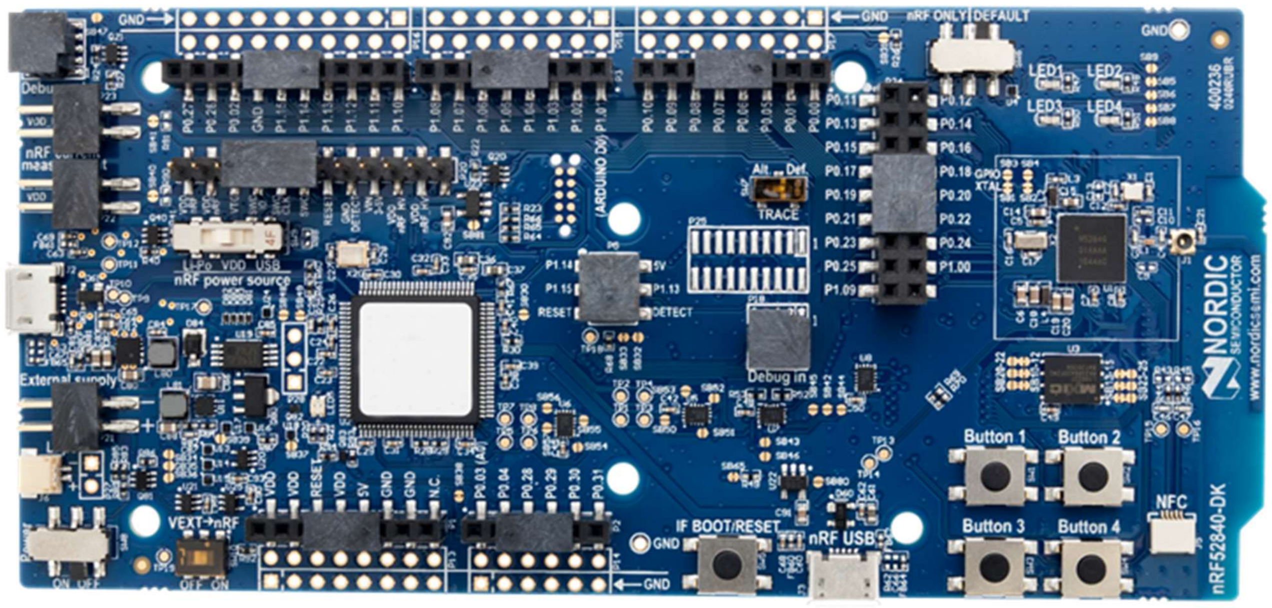


Figure 6: nRF52 DK - pca10056

2.1. Key features

- Supports Bluetooth LE, Bluetooth mesh, NFC, Thread and Zigbee
- User-programmable LEDs(4) and buttons(4)
- 2.4 GHz and NFC antennas
- SWF RF connector for direct RF measurements
- On-board SEGGER J-Link debugger/programmer
- Pins for measuring power consumption
- 1.7-5.0 V supply from USB, external, Li-Po battery or CR2032 coin cell battery
- (Arduino shild compatible → amelioration, cheap hardware)

3. nRF5 Software

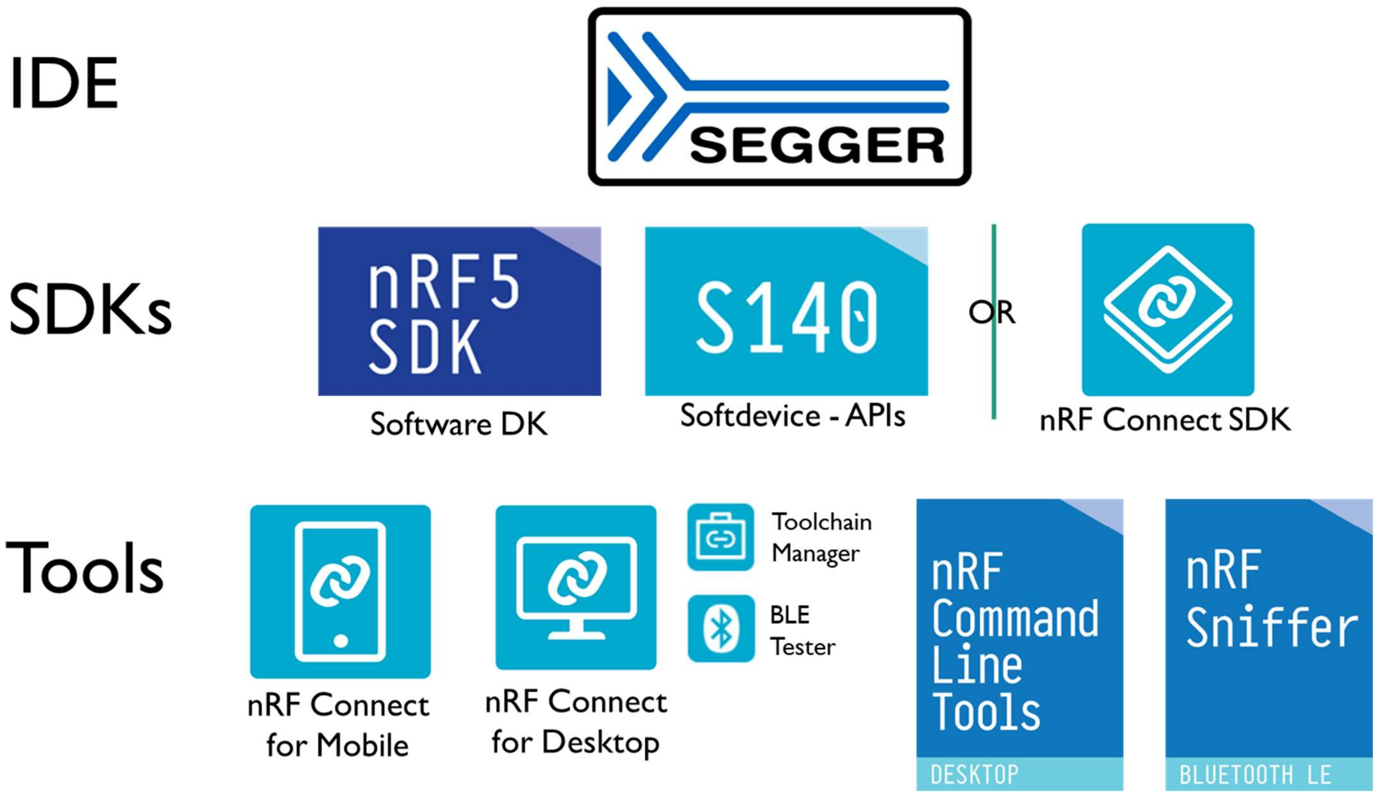
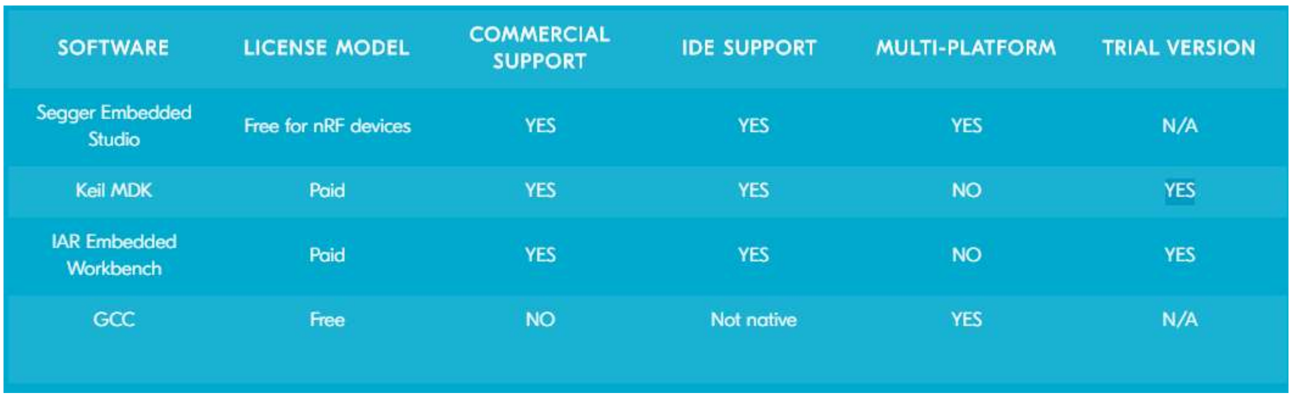


Figure 7 : nRF5 SoftwRE

3.1. IDE

nRF5 SoCs support different IDE, we choose the Segger Embedded Studio because it Free license when working with Nordic Semiconductor. It's also the default IDE used and develop by Nordic, more example and documentation are available.



SOFTWARE	LICENSE MODEL	COMMERCIAL SUPPORT	IDE SUPPORT	MULTI-PLATFORM	TRIAL VERSION
Segger Embedded Studio	Free for nRF devices	YES	YES	YES	N/A
Keil MDK	Paid	YES	YES	NO	YES
IAR Embedded Workbench	Paid	YES	YES	NO	YES
GCC	Free	NO	Not native	YES	N/A

Figure 8: Nordic Semiconductor - Supported IDE

Users can choose between the included Clang/LLVM or GCC C/C++ compiler options or use third party compilers.

For use with nRF Connect SDK, get the SEGGER Embedded Studio Nordic Edition - downloadable from the Toolchain Manager app for nRF Connect for Desktop.

3.2. Software Development Kit

For the nRF52 Series SoCs, Nordic offers the choice between using our standard nRF5 software development kit (SDK) together with our Bluetooth Low Energy (LE) protocol stacks (called SoftDevices) or using the nRF Connect SDK, based around the open source Zephyr RTOS.

SoftDevices + nRF5 SDK and Application-specific SDKs

Nordic Semiconductor's most popular solutions today, this solution is very mature and in use on hundreds of millions of products in the market today. This software solution has enormous scope and is qualified to Bluetooth 5.1. There is support for all of the nRF52 Series SoCs. Additionally, there are application-specific SDKs for mesh technologies such as Bluetooth mesh, Thread and Zigbee and HomeKit

nRF5 SDK – V15.3

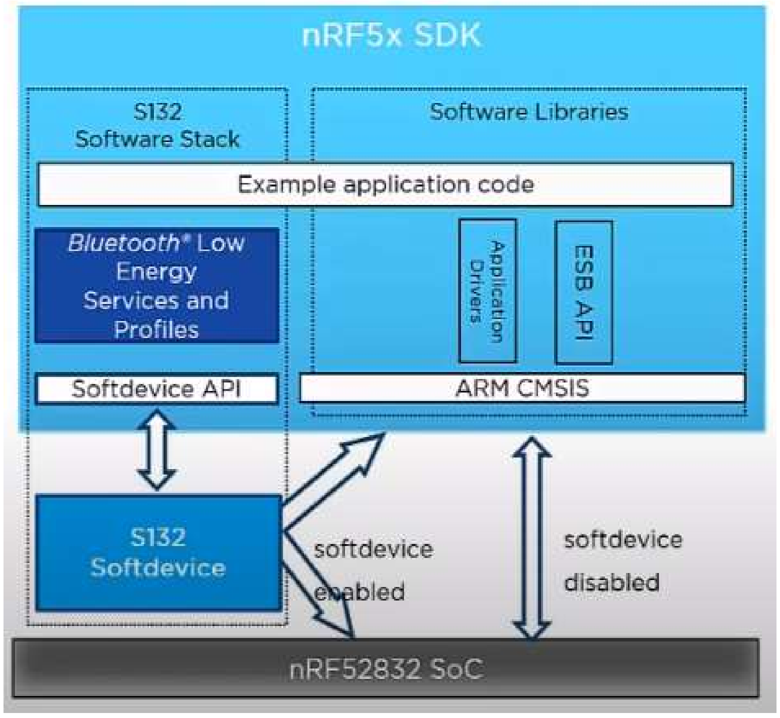
Software development kit, very useful to save time, it includes a lot of varied modules and examples like:

- Bluetooth Low Energy profiles
- Device Firmware Upgrade (DFU)
- GATT serializer
- Driver support for all peripherals

Softdevice 140

Feature-rich Bluetooth Low Energy protocol stack, Key features:

- Bluetooth 5.1 qualified
- High-throughput 2 Mbps
- Long Range
- Advertising Extensions
- CSA #2
- LE Secure Connections
- Privacy 1.2
- Configurable ATT Table
- Configurable ATT MTU
- Custom UUID
- LE Data Packet Length Extension
- L2CAP connection-oriented channels
- Concurrent multiprotocol support



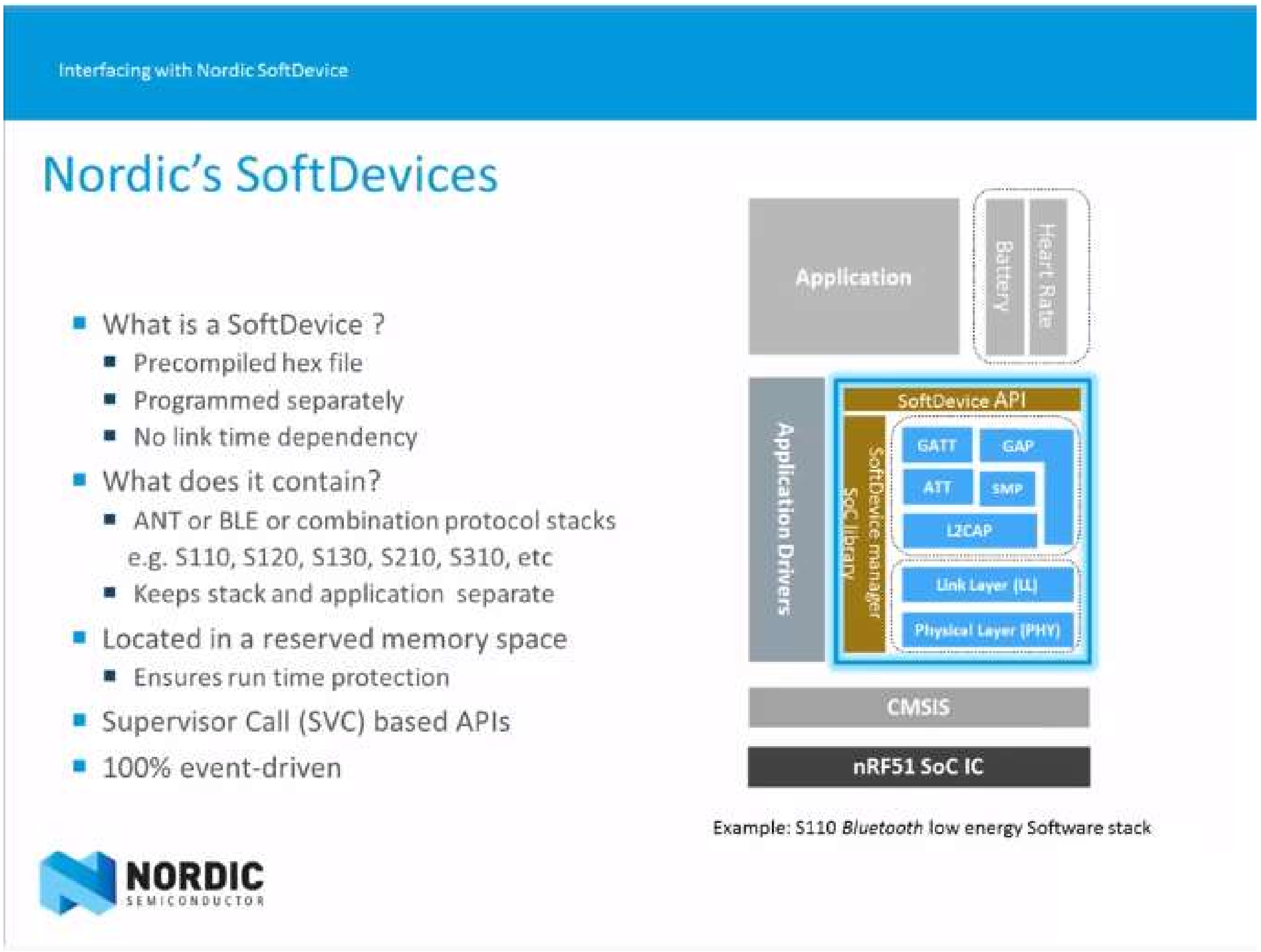
9 - SDK Introduction video

Peer Manager - Modules

The Peer Manager module handle bonded connections, which includes controlling encryption and pairing procedures.

BSP - Button Support Package - Modules

The Button Support Package (bsp) module handle the LEDs and the buttons.


Application
Battery
Heart Rate
Application Drivers
SoftDevice API
SoftDevice manager
SoC library
GATT
GAP
ATT
SMP
L2CAP
Link Layer (LL)
Physical Layer (PHY)
CMSIS
nRF51 SoC IC
NORDIC SEMICONDUCTOR

Interfacing with Nordic SoftDevice

Nordic's SoftDevices

- What is a SoftDevice ?
  - Precompiled hex file
  - Programmed separately
  - No link time dependency
- What does it contain?
  - ANT or BLE or combination protocol stacks e.g. S110, S120, S130, S210, S310, etc
  - Keeps stack and application separate
- Located in a reserved memory space
  - Ensures run time protection
- Supervisor Call (SVC) based APIs
- 100% event-driven

Example: S110 *Bluetooth* low energy Software stack

https://www.youtube.com/watch?v=tZjlixQPO-Q
ARM CMSIS: Cortex Microcontroller Software Interface Standard, hardware abstraction layer.

nRF Connect SDK (NCS) – V1.2

New, open-source and more scalable long-term evolution for development on Nordic devices. From the resource constrained, to the ever more complex high-end solutions. Based around the open source Zephyr RTOS, it uses a specific IDE SEGGER Embedded Studio Nordic Edition.

- Zephyr™ Real-time operating system (RTOS), which is built for connected low power product
- West: swiss-army knife command line tool for Zephyr, West's built-in commands provide a multiple repository management like Git. Zephyr uses this feature to provide conveniences for building applications, flashing and debugging them, and more.

3.3. Development tools

nRF Connect

- **nRF Connect for Desktop – V3.4.0**
  Contain useful app like:
  - Toolchain Manager: Install and manage tools for nRF Connect SDK (NCS)
  - Bluetooth Low Energy: BLE testing, need one additional nRF DK or Dongue
- **nRF Connect for Mobile – V4.24.1**
  Allow to scan and explore your Bluetooth Low Energy devices and communicate with them. nRF Connect for Mobile supports several Bluetooth **SIG** adopted profiles, as well as the Device Firmware Update profile (**DFU**) from Nordic Semiconductor or Eddystone from Google.
- **nRF Connect for Cloud**
  For cellualar IoT application

nRF Toolbox

The nRF Toolbox is a container app that stores your Nordic Semiconductor apps for Bluetooth® Low Energy in one location.

nRF Command Line Tools

Used for development, programming and debugging of Nordic Semiconductor's nRF52 devices.

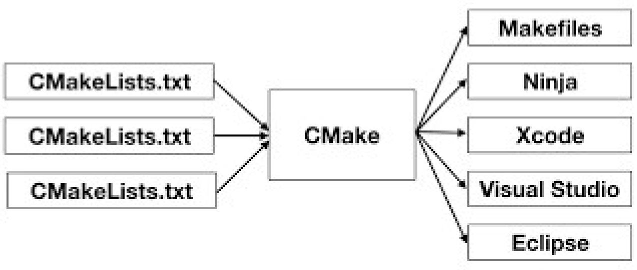
- **nrfjprog** executable - tool for programming through SEGGER J-LINK programmers and debuggers
- **mergehex** executable - enables you to combine up to three .HEX files into one single file
- **nrfjprog DLL** - a DLL that exports functions for programming and controlling nRF51, nRF52, nRF53 and nRF91 Series devices and lets developers create their own development tools using the DLLs API
- SEGGER J-Link software and documentation pack (only included in the Windows installer)

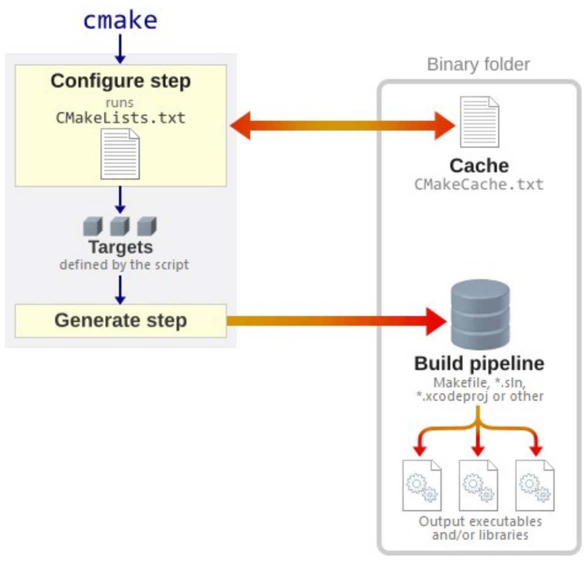
nRF Sniffer for Bluetooth LE

Useful tool for debugging and learning about Bluetooth Low Energy applications, allows near real-time display of Bluetooth LE packets. Equipment needed nRF5x DK or Dongle.

3.4. CMAKE & GCC

Used by the nRF Connect SDK





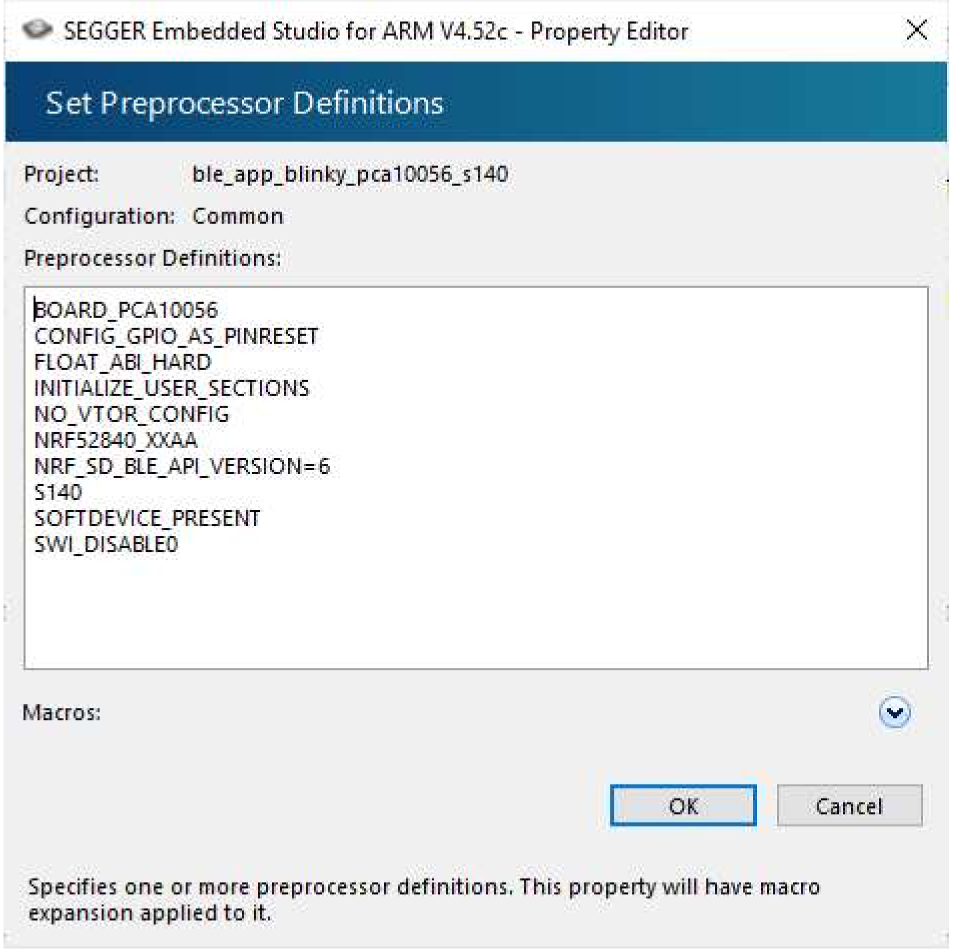
https://stackoverflow.com/questions/39761924/understanding-roles-of-cmake-make-and-gcc

4. Installation

Nordic Tools and download

4.1. SoftDevices + nRF5 SDK and Application-specific SDKs

1) Download the nRF5 SDK with all the examples
2) Download the softdevice precompiled (.hex) file
3) Install Segger Embedded Studio **for ARM**
    a. Open xxx_pca10056**.emProject** to open one project, according to your programmer
4) !!! Make sure you are using the right board:
    a. Project+clkD>>Option>>Code>>Preprocessor>>Preprocessor Definition


SEGGER Embedded Studio for ARM V4.52c - Property Editor
Set Preprocessor Definitions
Macros:
OK
Cancel

Project: ble_app_blinky_pca10056_s140

Configuration: Common

Preprocessor Definitions:

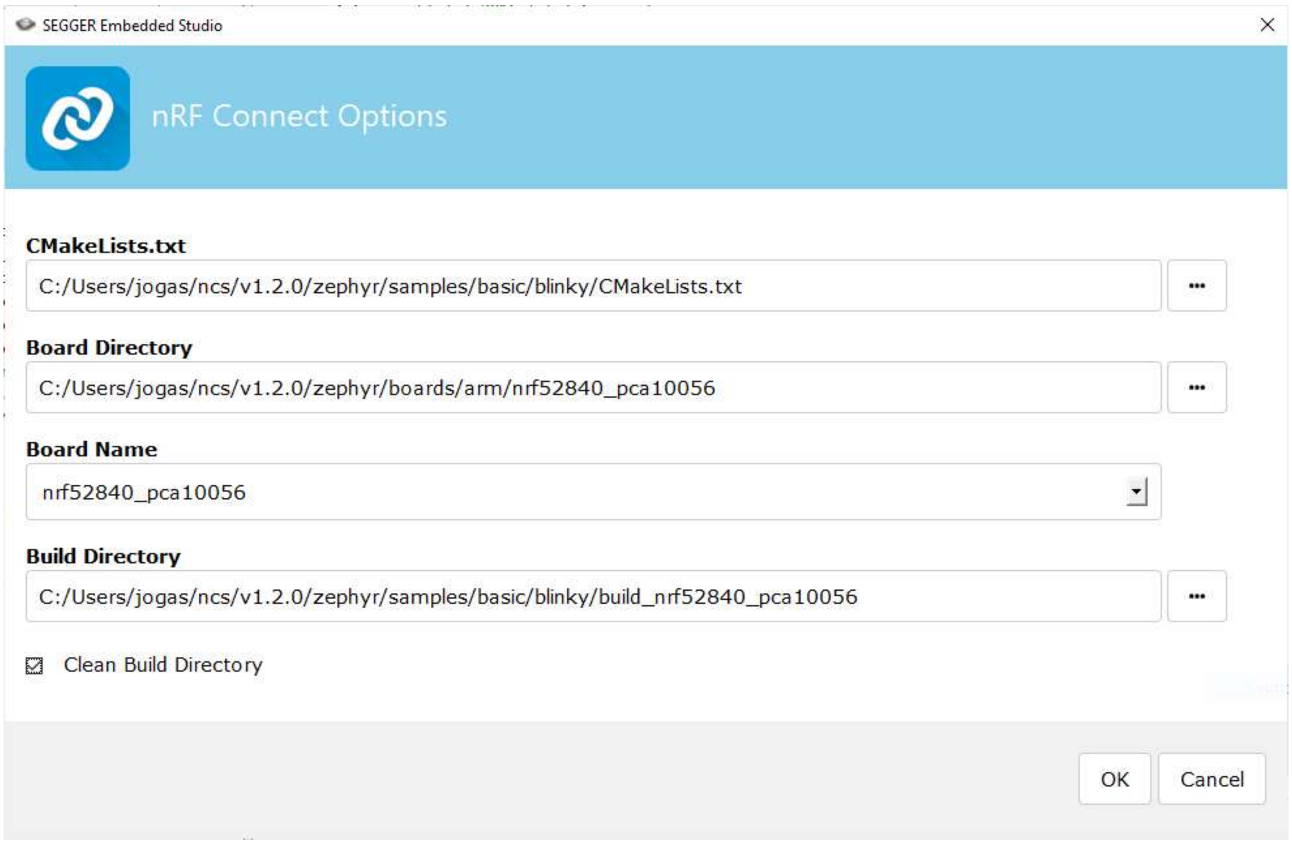
```
BOARD_PCA10056
CONFIG_GPIO_AS_PINRESET
FLOAT_ABI_HARD
INITIALIZE_USER_SECTIONS
NO_VTOR_CONFIG
NRF52840_XXAA
NRF_SD_BLE_API_VERSION=6
S140
SOFTDEVICE_PRESENT
SWI_DISABLE0
```

Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.

    b. Project+clkD>>Option>>Debug>>Debugger>>Target Device
    c. Control that nrf_peripheral.h use the correct define

4.2. nRF Connect SDK (NCS

1. Install nRF Connect for Desctop
2. Install the Toolchain Manager app
   → Install nRF Connect SDK and Segger Embedded Studio **Nordic Edition**
3. Checkbuild configuration: File>>Open nRF Connect SDK Project



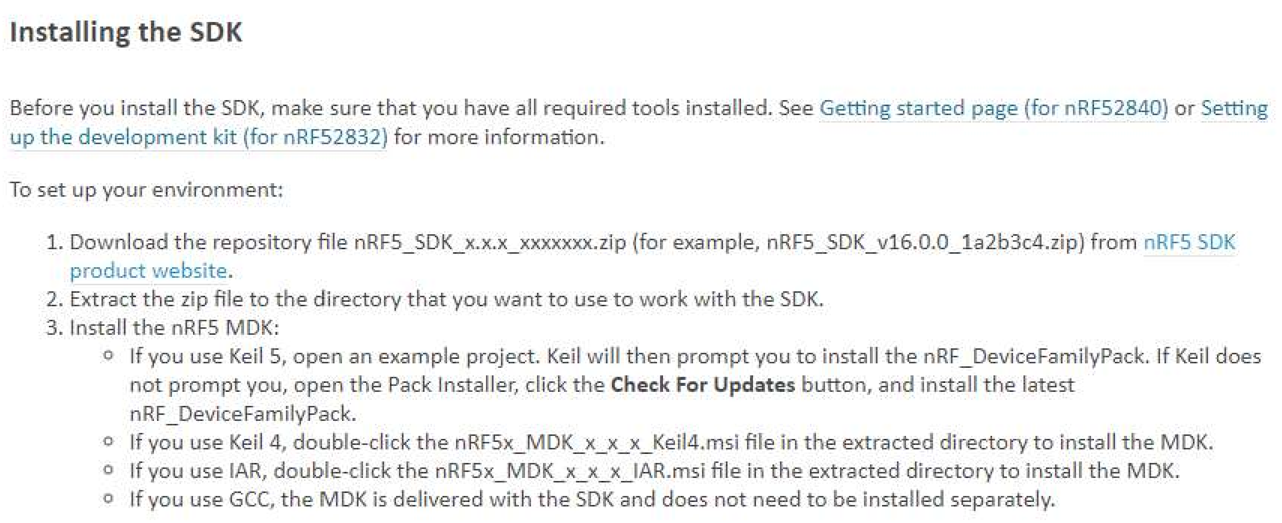
CMakeLists.txt
C:/Users/jogas/ncs/v1.2.0/zephyr/samples/basic/blinky/CMakeLists.txt

Board Directory
C:/Users/jogas/ncs/v1.2.0/zephyr/boards/arm/nrf52840_pca10056

Board Name
nrf52840_pca10056

Build Directory
C:/Users/jogas/ncs/v1.2.0/zephyr/samples/basic/blinky/build_nrf52840_pca10056

☑ Clean Build Directory



Installing the SDK

Before you install the SDK, make sure that you have all required tools installed. See Getting started page (for nRF52840) or Setting up the development kit (for nRF52832) for more information.

To set up your environment:

1. Download the repository file nRF5_SDK_x.x.x_xxxxxxx.zip (for example, nRF5_SDK_v16.0.0_1a2b3c4.zip) from nRF5 SDK product website.
2. Extract the zip file to the directory that you want to use to work with the SDK.
3. Install the nRF5 MDK:
   - If you use Keil 5, open an example project. Keil will then prompt you to install the nRF_DeviceFamilyPack. If Keil does not prompt you, open the Pack Installer, click the **Check For Updates** button, and install the latest nRF_DeviceFamilyPack.
   - If you use Keil 4, double-click the nRF5x_MDK_x_x_x_Keil4.msi file in the extracted directory to install the MDK.
   - If you use IAR, double-click the nRF5x_MDK_x_x_x_IAR.msi file in the extracted directory to install the MDK.
   - If you use GCC, the MDK is delivered with the SDK and does not need to be installed separately.

4.3. Getting started

Getting started with the software development
Exemples

Drag and drop a **file.HEX** on the usb J-LINK peripheral to Flash the MCU.

4.1. NRF_LOG_INFO()

1. Install: CMSIS Configuration Wizard
2. Sdk_config.h+ClkD>>CMSIS conf Wizard>>nrf_log>>

NRF_LOG_BACKEND_RTT_ENABLED>>enable

3. Disable: NRF_FPRINTF_FLAG_AUTOMATIC_CR_ON_LF_ENABLED

4.2. Debug an external Board with nRF5 DK

If it is inconvenient to have a separate power supply on the external board, the nRF52840 DK can supply power through the Debug out connector (P19). To enable this, short solder bridge SB47. Note that as long as SB47 is shorted, it is not possible to program the onboard nRF52840 SoC even if the external board is unplugged.

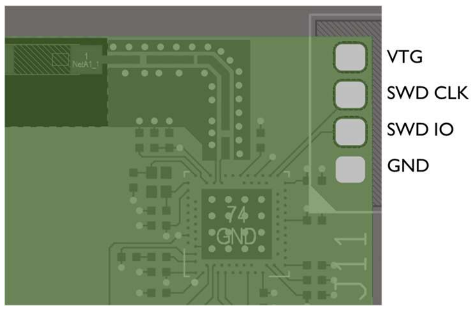


Figure 10: BMS - nRF52840 Debug Connector

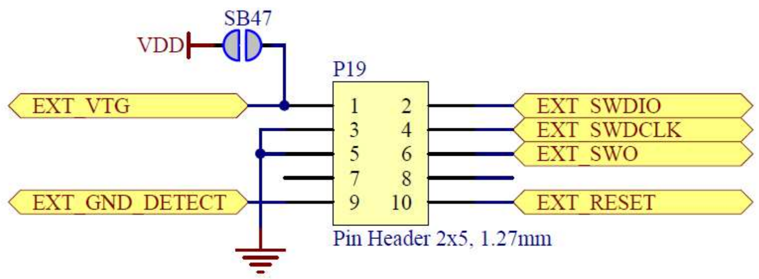


Figure 12: Debug Out p19 Connector (nRF52840 DK Schematic)

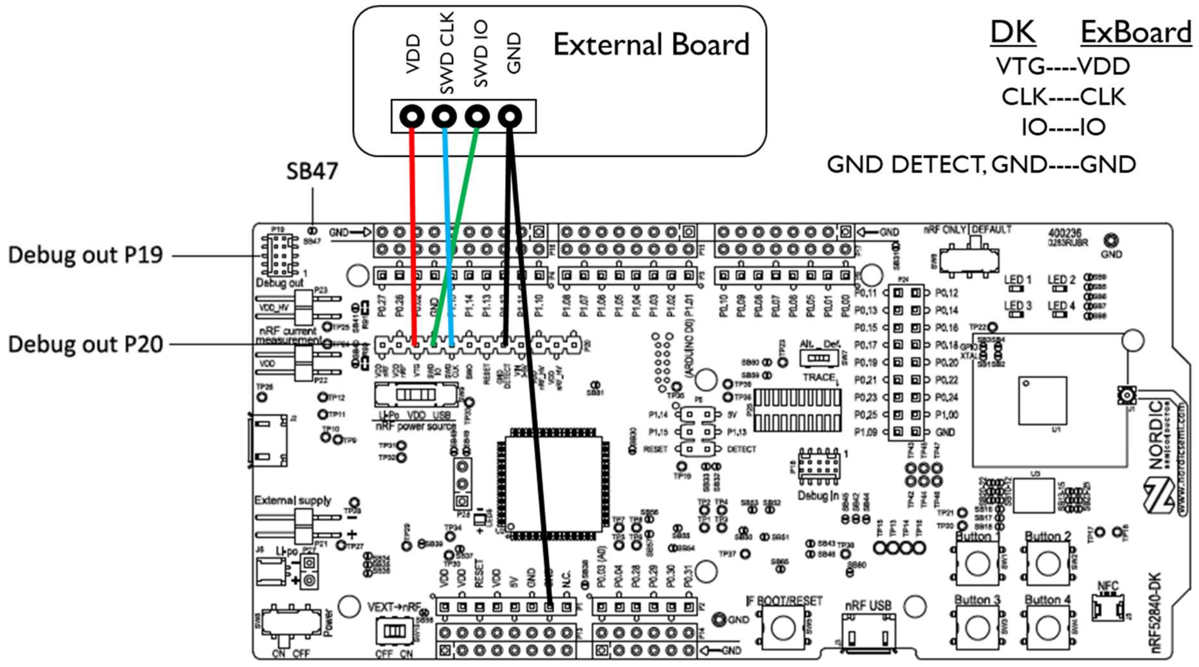


Figure 11: External Board debug connection

4.3. Custom Services error

Debug tuto

1. Go to this tutorial
2. For newer SDKs, go to sdk_config.h and set "#define NRF_SDH_BLE_VS_UUID_COUNT 1"

Add more RAM

3. You need to adjust the RAM size and start address.Simply add 16 bytes to the start and subtract 16 from the size per each new UUID you add.
   Go to: Project>>Option>>Common (instead Release)>>Linker>>Section Placement Macros
   And change this settings:
   - RAM_START=0x20002218
   - RAM_SIZE=0xdde8
     the new setting would be
   - RAM_START=0x20002218 + 0x10 =0x20002228
   - RAM_SIZE=0xdde8 -0x10 =0xDDD8

4.4. Compile an example with an old SDK

Problem: nrf_log.h no such file or directory
Fix:
1. Make a copy of this directoty .../examples/ble_peripheral/ble_app_template
2. Rename it to advertising_tutorial
3. Copy and replace the main.c file provided by this tutorial into the newly created project directory
4. Compile and enjoy!

Explanation:
This tutorial project had been made for SDK V15.0.0. By now we have newer versions of SDK which are not really compatible with the project distributed by the tutorial. But we can still use the source file :)

4.5. Error code

Research the code number in those files

- nrf_error.h (app_error>>dependencies
- sdk_errors.h

4.6. Modifying an example

Modify an example

Open a project

Search the file: softdevice/ses/xxx_pca10056.emProject

- F7: Build the project -> create the zypher.elf file
- F5: Debug/Go

Open it directly form the Toolchain Manager

Adding files

- Modify *CMakeList.txt* directly with Segger
- Use tags:

```
# NORDIC SDK APP START
target_sources(app PRIVATE src/main.c)
# NORDIC SDK APP END
```

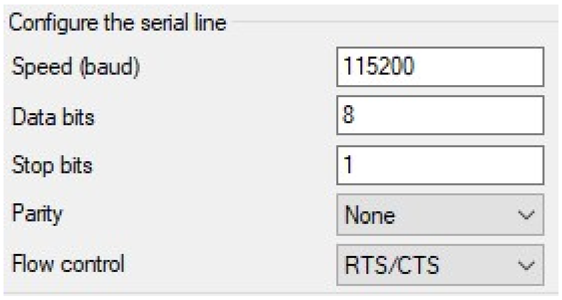
4.7. Configuring your application in Debug

set CONFIG_DEBUG_OPTIMIZATIONS to y
Default configuration for:

- A Library: `Kconfig` (& `prj.conf` permanently changes)
- A Board: `*_defconfig` & `Kconfig.defconfig`

(Write your devicetree n the Zephyr documentation for more information)
[Kconfig Configuration]

4.8. UART communication



Configure the serial line

Speed (baud)	115200
Data bits	8
Stop bits	1
Parity	None
Flow control	RTS/CTS

5. BLE Tutorials

These 3 tutorials will help a lot implementing a complete Bluetooth-LE profile and exchange some data

5.1. BLE Advertising

Tuto ble-advertising link

Appearance - Add an Icon to a device

The central device can display an icon representing the generic device type of the peripheral device. If the device type is conformed to Bluetooth SIG an icon will be displayed automatically.

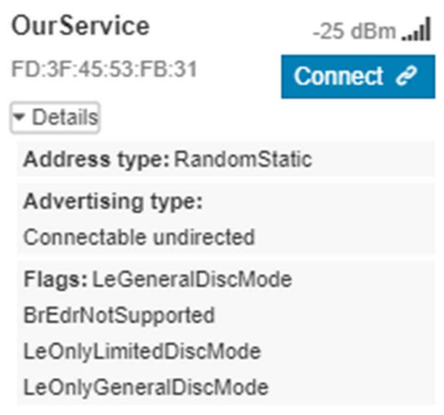
- In the function gap_params_init() add:
  ```
  sd_ble_gap_appearance_set(BLE_APPEARANCE_CYCLING_POWER_SENSOR)
  ```

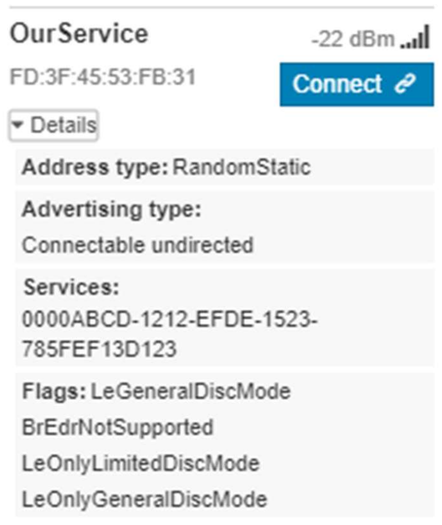
5.2. BLE Services

Tuto ble-services link

Add service, create a custom service with a custom base UUID and a service UUID.

Create a Service, add it to the GATT in our_service_init(), define it in advertise_init to appear when the central do a SCAN_REQ. We made our own custom service with a custom base UUID and a service UUID.

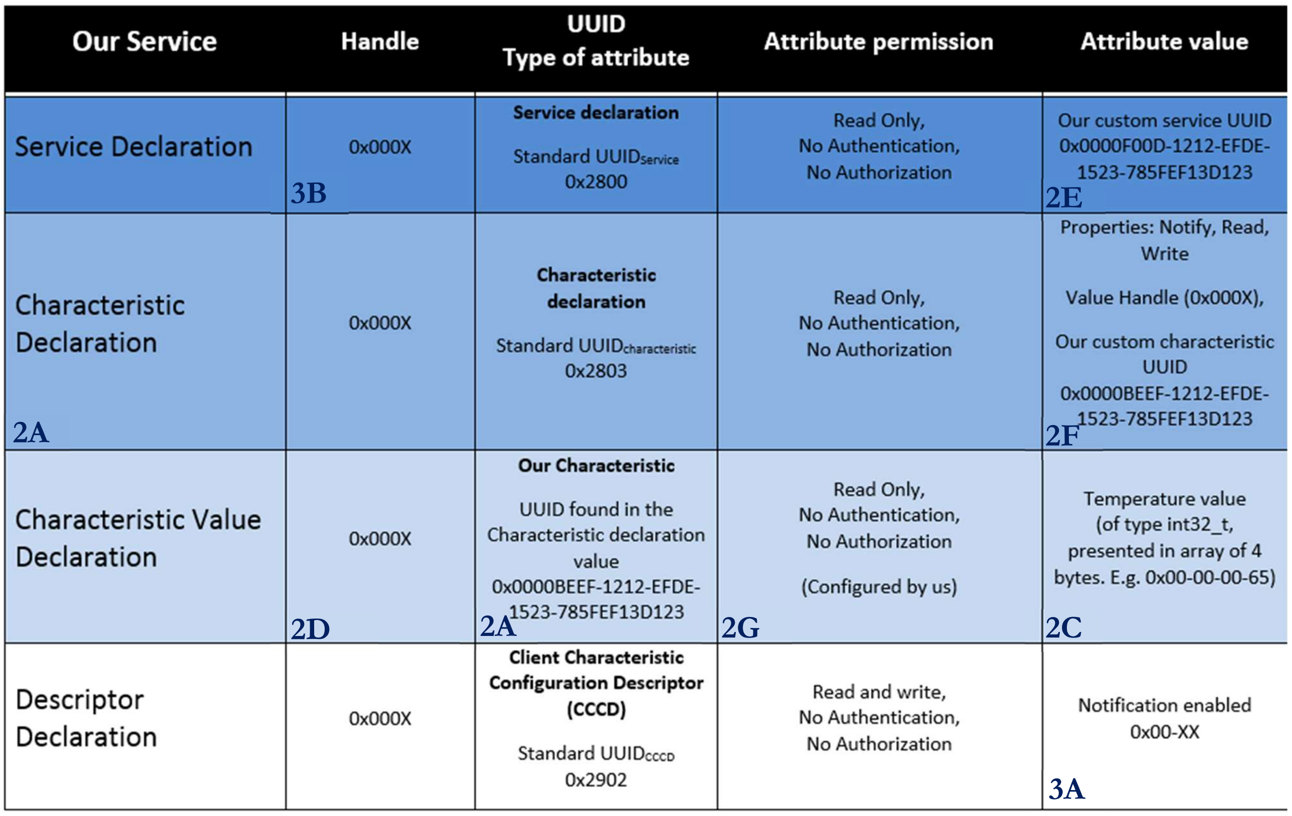

OurService -25 dBm
FD:3F:45:53:FB:31
Connect
Details
Address type: RandomStatic
Advertising type: Connectable undirected
Flags: LeGeneralDiscMode BrEdrNotSupported LeOnlyLimitedDiscMode LeOnlyGeneralDiscMode


OurService -22 dBm
FD:3F:45:53:FB:31
Connect
Details
Address type: RandomStatic
Advertising type: Connectable undirected
Services: 0000ABCD-1212-EFDE-1523-785FEF13D123
Flags: LeGeneralDiscMode BrEdrNotSupported LeOnlyLimitedDiscMode LeOnlyGeneralDiscMode

5.3. BLE Characteristics

Tuto ble-characteristics link
https://devzone.nordicsemi.com/f/nordic-q-a/12653/integers-in-characteristics-little-or-big-endian

Adding a characteristic



Our Service	Handle	UUID Type of attribute	Attribute permission	Attribute value
Service Declaration	0x000X 3B	**Service declaration** Standard UUID$_{Service}$ 0x2800	Read Only, No Authentication, No Authorization	Our custom service UUID 0x0000F00D-1212-EFDE-1523-785FEF13D123 2E
Characteristic Declaration 2A	0x000X	**Characteristic declaration** Standard UUID$_{Characteristic}$ 0x2803	Read Only, No Authentication, No Authorization	Properties: Notify, Read, Write Value Handle (0x000X), Our custom characteristic UUID 0x0000BEEF-1212-EFDE-1523-785FEF13D123 2F
Characteristic Value Declaration	0x000X 2D	**Our Characteristic** UUID found in the Characteristic declaration value 0x0000BEEF-1212-EFDE-1523-785FEF13D123 2A	Read Only, No Authentication, No Authorization (Configured by us) 2G	Temperature value (of type int32_t, presented in array of 4 bytes. E.g. 0x00-00-00-65) 2C
Descriptor Declaration	0x000X	**Client Characteristic Configuration Descriptor (CCCD)** Standard UUID$_{CCCD}$ 0x2902	Read and write, No Authentication, No Authorization	Notification enabled 0x00-XX 3A

Figure 13: GATT table Tuto-Char

Step 2: Add the Characteristic
Step 2.A, Use custom UUID to define characteristic value type
Step 2.B, Configure the Attribute Metadata
Step 2.C, Configure the Characteristic Value Attribute
Step 2.D, Add handles for the characteristic to our struct
Step 2.E, Add the new characteristic to the service
Step 2.F, Add read/write properties to our characteristic value
Step 2.G, Set read/write permissions to our characteristic
Step 2.H, Set characteristic length

Updating the characteristic and sending notifications

Step 3: Client Characteristic Configuration Descriptor (CCCD)
Step 3.A, Configuring CCCD metadata
Step 3.B, Housekeeping part 1: Give our service connection handle a default value
Step 3.C, Housekeeping part 2: Responding to connect and disconnect events
Step 3.D, Housekeeping part 3: Handling BLE events related to our service
Step 3.E, Update characteristic value
Step 3.F, Update the characteristic with temperature data
Step 3.G, Declare a timer ID and a timer interval
Step 3.H, Initiate the timer
Step 3.I, Start our timer

Housekeeping

The SoftDevice Handler dispatch BLE events information, in addition with “housekeeping” or keeping the connection Handle update, allow us to execute some code when connected only.

5.4. Adding Multiple Characteristics to your Service and updating their values

Step 1

You would need to create another 16-bit characteristic UUID

```
#define BLE_UUID_CHARACTERISTIC_1 0xBEEF // Just a random, but recognizable value
#define BLE_UUID_CHARACTERISTIC_2 0xB00B
```

Step 2

In our_service.h navigate to Step 2.D where we define the structure ble_os_t.

We need to add the characteristic handles to our structure. We will be using these handle instances when updating a corresponding characteristic.

```
ble_gatts_char_handles_t char_handles_1; // Adding handles for the characteristic to our structure
ble_gatts_char_handles_t char_handles_2; // Adding handles for the characteristic to our structure
```

The code should look something like that:

```
typedef struct
{
uint16_t conn_handle; /**< Handle of the current connection (as provided by the BLE stack, is
BLE_CONN_HANDLE_INVALID if not in a connection).*/
// keeps track of the current connection and has nothing to do with attribute table handles
uint16_t service_handle; /**< Handle of Our Service (as provided by the BLE stack). */
// OUR_JOB: Step 2.D, Add handles for the characteristic attributes to our struct
ble_gatts_char_handles_t char_handles_1;
// Adding handles for the characteristic to our structure
ble_gatts_char_handles_t char_handles_2;
// Adding handles for the characteristic to our structure
}ble_os_t;
```

Step 3

Add an extra characteristic

In our_services.c create a copy of the our_char_add which will create a new characteristic. You should assign it a different name. You should assign it a unique UUID. You must assign it a unique characteristic handle (such as char_handles_1 and char_handles_2). You may choose to set different max_len depending on your application.

```
static uint32_t our_char_add_1(ble_os_t * p_our_service)
{
...

// OUR_JOB: Step 2.A, Add a custom characteristic UUID
char_uuid.uuid = BLE_UUID_CHARACTERISTIC_1; // specifying our characteristic UUID (16-bit)

...

// OUR_JOB: Step 2.E, Add our new characteristic to the service.
err_code = sd_ble_gatts_characteristic_add(p_our_service->service_handle,
// adding a service handle to the attribute table
&char_md, // adding characteristic metadata to the attribute table
&attr_char_value, // adding the attribute characteristic value to the attribute table
&p_our_service->char_handles_1); // adding characteristic handles to the attribute table

...
}

static uint32_t our_char_add_2(ble_os_t * p_our_service)
{
...

char_uuid.uuid = BLE_UUID_CHARACTERISTIC_2; // specifying our characteristic UUID (16-bit)

...

err_code = sd_ble_gatts_characteristic_add(p_our_service->service_handle,
// adding a service handle to the attribute table
&char_md, // adding characteristic metadata to the attribute table
&attr_char_value, // adding the attribute characteristic value to the attribute table
&p_our_service->char_handles_2); // adding characteristic handles to the attribute table

...
}
```

Step 4

Now we will be initialising those characteristics

```
void our_service_init(ble_os_t * p_our_service)
{
...

// OUR_JOB: Call the function our_char_add() to add our new characteristic to the service.
our_char_add_1(p_our_service);
our_char_add_2(p_our_service);

...
}
```

Step 5

Now let's make a function that updates the characteristics. Basically, you would clone your update function.

```
void characteristic_value_update_1(ble_os_t *p_our_service, char machine_serial_number[])
{
...

hvx_params.handle = p_our_service -> char_handles_1.value_handle; //the handle needs to know what

...
}

void characteristic_value_update_2(ble_os_t *p_our_service, char machine_serial_number[])
{
...

hvx_params.handle = p_our_service -> char_handles_2.value_handle; //the handle needs to know what

...
}
```

5.5. Receiving (retrieve or write) values from characteristics and making use of them

Step 1

In main.c somewhere at the top of the file add the write handler

```
// BLE_WRITE:
/**@brief Function for handling write events to the LED characteristic.
 *
 * @param[in] characteristic1_value     value that was received from the phone
 */
// called from our_services.c from on_write();
// Make a note of the arguments that are passed to this handler, we will use that later on
static void characteristic1_value_write_handler(uint32_t characteristic1_value)
{
       NRF_LOG_INFO("We have received the characteristic1 value into our App:  %d",
characteristic1_value);
}
// Add other handlers here...
```

Step 2

In our_service.h file, you need to add the init structure at the top of the file.

```
// BLE_WRITE:
/** @brief Our Service init structure. This structure contains all options and data needed for
 *        initialization of the service.*/
 //This is used to pass the write handlers for different characteristics from main.c
 //This is essentially like public constructor. All of the content will be copied to instance.
 //Note that "uint32_t characteristic1_value" part had to match from Step 1
typedef void (*ble_os_characteristic1_value_write_handler_t) (uint32_t characteristic1_value);

// Add other handlers here...
typedef struct
{
       /**< Event handler to be called when the Characteristic1 is written */
    ble_os_characteristic1_value_write_handler_t characteristic1_value_write_handler;
    // Add other handlers here...

} ble_os_init_t;
```

Step 3

In our_service.h file modify the structure.

```
// This structure contains various status information for our service.
// The name is based on the naming convention used in Nordics SDKs.
// 'ble indicates that it is a Bluetooth Low Energy relevant structure and
// os is short for Our Service).
typedef struct
{
    uint16_t conn_handle;    /**< Handle of the current connection (as provided by the BLE
stack, is BLE_CONN_HANDLE_INVALID if not in a connection).*/
// keeps track of the current connection and has nothing to do with attribute table handles
    uint16_t service_handle; /**< Handle of Our Service (as provided by the BLE stack). */

       ...

// BLE_WRITE: Write handlers. Upon BLE write, these handler will be called
// Their implementation is in the main.c
    ble_os_characteristic1_value_write_handler_t characteristic1_value_write_handler;  /**<
Event handler to be called when the Characteristic1 is written. */
    // Add other handlers here...

}ble_os_t;
```

Step 4

In our_service.c file modify the function.

```
/**@brief Function for initiating our new service.
 *
 * @param[in]   p_our_service        Our Service structure.
 * @param[in]   init                 Our Service init structure. (BLE_WRITE)
 *
 */
void our_service_init(ble_os_t * p_our_service, ble_os_init_t * init)
{
    uint32_t   err_code; //Variable to hold return codes from library and softdevice functions

        ...

    // BLE_WRITE: transfer the pointers from the init instance to the module instance
    p_our_service->characteristic1_value_write_handler = init-
>characteristic1_value_write_handler;

        ...

    APP_ERROR_CHECK(err_code);
}
```

Step 5

In main.c file modify the function. We have to init our service module and let it know about the write handler.

```
/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    ret_code_t         err_code;
    nrf_ble_qwr_init_t qwr_init = {0};

    // BLE_WRITE: Initialize Our Service module.
    ble_os_init_t init = {0}; // Init Our Service module
    init.characteristic1_value_write_handler = characteristic1_value_write_handler;
    // Add other handlers here...

    ...

        // BLE_WRITE: We need to add the init instance pointer to our service instance
    // Initialize our service
    our_service_init(&m_our_service, &init);

    ...

}
```

Step 6

In main.c file, make sure that the following macro exists.

```
/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    ret_code_t err_code;

    ...

    // BLE_WRITE: Make sure this macro exists
    // OUR_JOB: Step 3.C Call ble_our_service_on_ble_evt() to do housekeeping of ble
connections related to our service and characteristics
    // Needed for associating the observer with the event handler of the service
    NRF_SDH_BLE_OBSERVER(m_our_service_observer, APP_BLE_OBSERVER_PRIO,
ble_our_service_on_ble_evt, (void*) &m_our_service); // Modules that want to be notified about
SoC events must register the handler using this macro.

}
```

Step 7

In our_service.c file modify the function.

```
// ALREADY_DONE_FOR_YOU: Declaration of a function that will take care of some housekeeping of
ble connections related to our service and characteristic
void ble_our_service_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
{
    ble_os_t * p_our_service =(ble_os_t *) p_context;
    // OUR_JOB: Step 3.D Implement switch case handling BLE events related to our service.

    switch(p_ble_evt -> header.evt_id)
    {

              ...

        // BLE_WRITE:
        // Write: Data was received to the module
        case BLE_GATTS_EVT_WRITE:
            on_write(p_our_service, p_ble_evt);
            break;

        //
        default:
          // No implementation needed
          break;
    }

}
```

Step 8

In our_service.c somewhere at the top of the file add this function

```
// BLE_WRITE:
/**@brief Function for handling the Write event.
 *
 * @param[in] p_our_service     Our Service structure.
 * @param[in] p_ble_evt         Event received from the BLE stack.
 */
static void on_write(ble_os_t * p_our_service, ble_evt_t const * p_ble_evt)
{
    NRF_LOG_INFO("on_write: called");

    ble_gatts_evt_write_t const * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;

    if ((p_evt_write->handle == p_our_service-
    >characteristic1_value_write_handler.value_handle))
    {
        NRF_LOG_INFO("characteristic1_value: Write Happened!");


        // Make sure that the data is 4 bytes (or whatever the size of your characteristic)
        // It has to match the exact byte size of the characteristic to avoid problems
        int8_t len = p_evt_write->len;
        if (len != 4)
        {
            NRF_LOG_INFO("ERROR: incomplete package");
            NRF_LOG_INFO("len: %d", len);
            return;
        }

        // Data must be sent from in Little Endian Format and 4 bytes
        // Convert the little endian 4 bytes of data into 32 bit unsigned int
        uint32_t *characteristic1_value_adr;
        uint32_t characteristic1_value_val;
        characteristic1_value_adr = (uint32_t*) p_evt_write->data;
        characteristic1_value_val = *characteristic1_value_adr;

        NRF_LOG_INFO("characteristic1_value: %d", characteristic1_value_val);

        // Call the write handler function. Implementation is in the main.
        p_our_service->characteristic1_value_write_handler(characteristic1_value_val);
    }
}
```

6. Android Studio

Figure 14: Android Studio Application IDE V4.0

Figure 17: Android BLE Library from Nordic Semi-conductor V2.2.0

Figure 16: Android Studio supported languages

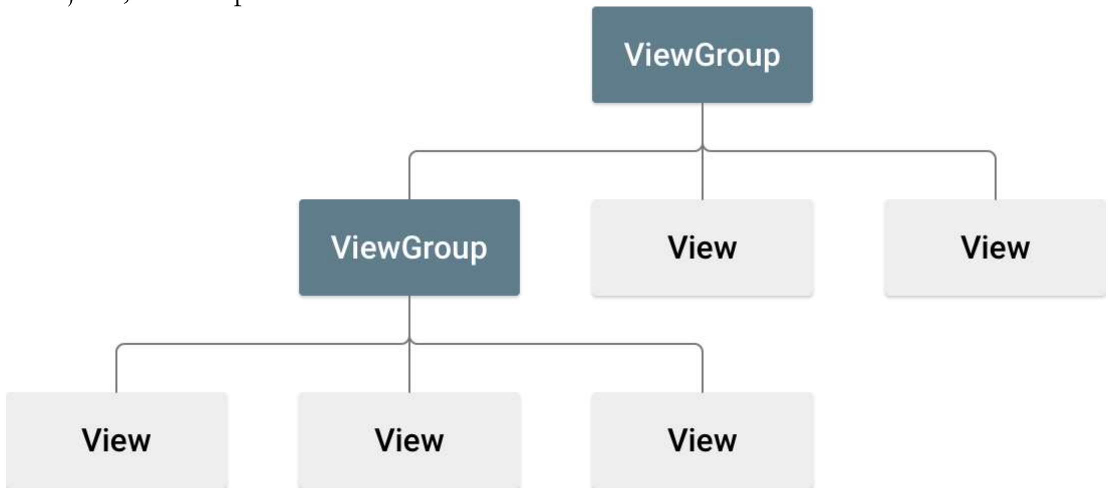
Figure 15: Android Studio Build Tool

Gradle

Gradle is an open-source build automation system = CMake for android/java.
Gradle was designed for multi-project builds, which can grow to be quite large. It supports incremental builds by intelligently determining which parts of the build tree are up to date; any task dependent only on those parts does not need to be re-executed.

6.1. Tutorial

Make your first app
The user interface (UI) for an Android app is built as a hierarchy of layouts and widgets. The layouts are **ViewGroup** objects, containers that control how their child views are positioned on the screen. Widgets are **View** objects, UI components such as buttons and text boxes.



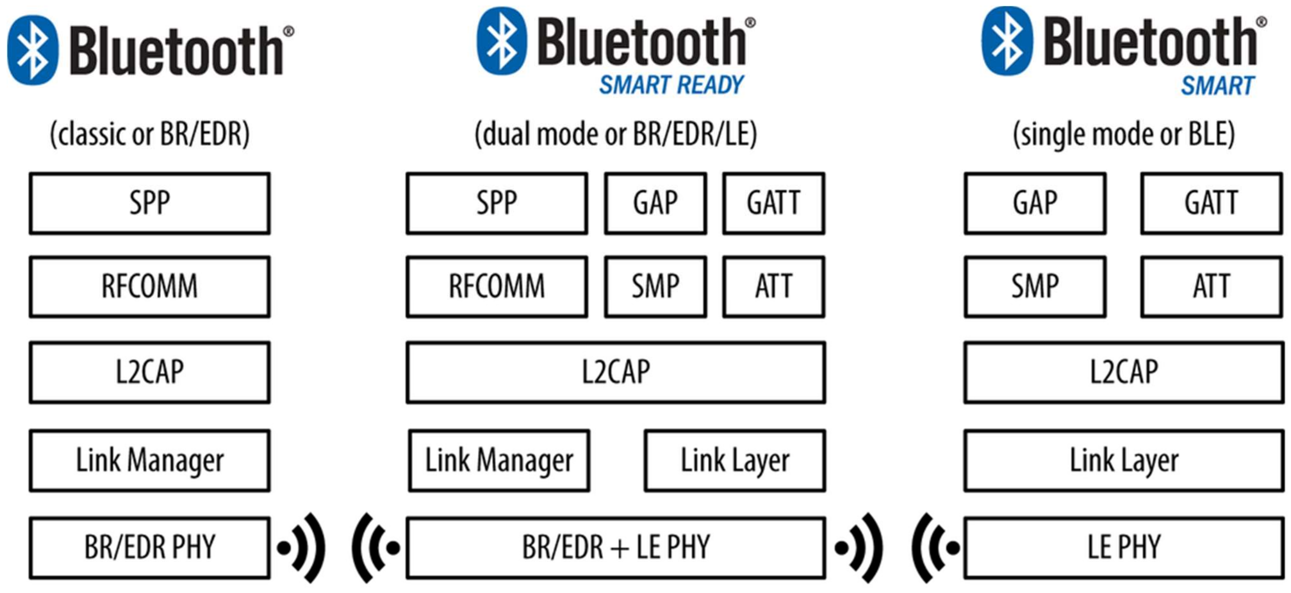
Android Studio's Layout Editor writes the XML for you as you drag and drop views to build your layout.

6.2. Drawable - Vector Asset

1. Download one SVG icon on flaticon.com
2. Save it on: app\src\main\res\drawable
3. Right click on: res>>new>>**Vector Asset**
4. Resize, change color: `android:tint="@color/colorSecondary"`

7. Acronyms

- MCU: microcontroller unit
- SoC: System on Chips = MCU
- SPI : Serial Peripheral Interface
- API: application programming interface
- IDE: Integrated Development Environment
- SDK: Software Development Kits (SDKs) are your starting point for software development on the nRF51 and nRF52 Series. They contain source code libraries and example applications covering wireless functions, libraries for all peripherals, bootloaders, Wired and OTA firmware upgrades, RTOS examples, serialization libraries and more. Some of the specialized SDKs are installed on top of the generic nRF5 SDK and extend its functionality.
- Nordic Semiconductor provides Software Development Kits to facilitate firmware development for different devices and applications. The SDKs contain examples that are tailored to run on Nordic Semiconductor's Development Kits.
- MDK Microcontroller Development Kit
- GNU: **G**NU's **N**ot **U**NIX, operating system
- NCS: nRF Connect SDK
- DLL: Dynamic-link library, Modules used by a program are loaded from individual shared library into memory at load time or runtime, rather than being copied by a linker when it creates a single monolithic executable file for the program.
- CMSIS: Cortex Microcontroller Software Interface Standard, hardware abstraction layer.
- IPSP: Bluetooth Internet Protocol Support Profile API provides functions for supporting of exchanging IPv6 packets between devices over the BLE transport.
- Device tree: all the GATT hierarchical structure
- BR/EDR: Bluetooth Basic Rate/ Enhanced Data Rate = Bluetooth classic
- Bluetooth SMART = Bluetooth LE


Bluetooth
(classic or BR/EDR)
SPP
RFCOMM
L2CAP
Link Manager
BR/EDR PHY
Bluetooth SMART READY
(dual mode or BR/EDR/LE)
SPP
GAP
GATT
RFCOMM
SMP
ATT
L2CAP
Link Manager
Link Layer
BR/EDR + LE PHY
Bluetooth SMART
(single mode or BLE)
GAP
GATT
SMP
ATT
L2CAP
Link Layer
LE PHY