

*nRF 5 – Programming cheatsheet*

[1. Bluetooth – Theory 2](#_Toc43451191)

[1.1. Introduction 2](#_Toc43451192)

[1.2. Bluetooth Low Energy Architecture 3](#_Toc43451193)

[1.3. The Generic Access Profile (GAP) 4](#_Toc43451194)

[1.4. Security Manager Protocol (SMP) 1](#_Toc43451195)

[1.5. The Generic Attribute Profile (GATT) 2](#_Toc43451196)

[2. nRF52840 Development Kit 6](#_Toc43451197)

[2.1. Key features 6](#_Toc43451198)

[3. nRF5 Software 7](#_Toc43451199)

[3.1. IDE 7](#_Toc43451200)

[3.2. Software Development Kit 8](#_Toc43451201)

[3.3. Development tools 10](#_Toc43451202)

[3.4. CMAKE & GCC 10](#_Toc43451203)

[4. Installation 11](#_Toc43451204)

[4.1. SoftDevices + nRF5 SDK and Application-specific SDKs 11](#_Toc43451205)

[4.2. nRF Connect SDK (NCS 11](#_Toc43451206)

[4.3. Getting started 12](#_Toc43451207)

[4.4. Custom Services error 12](#_Toc43451208)

[4.5. Compile an example with an old SDK 12](#_Toc43451209)

[4.6. Error code 13](#_Toc43451210)

[4.7. CMSIS Configuration Wizard 13](#_Toc43451211)

[4.8. Modifying a sample application 13](#_Toc43451212)

[4.9. Configuring your application in Debug 13](#_Toc43451213)

[4.10. UART communication 13](#_Toc43451214)

[4.11. Debug an external Board with nRF5 DK 14](#_Toc43451215)

[5. BLE Tutorials 15](#_Toc43451216)

[5.1. BLE Advertising 15](#_Toc43451217)

[5.2. BLE Services 15](#_Toc43451218)

[5.3. BLE Characteristics 16](#_Toc43451219)

[5.4. Adding Multiple Characteristics to your Service and updating their values 17](#_Toc43451220)

[5.5. Receiving (retrieve or write) values from characteristics and making use of them 19](#_Toc43451221)

[6. Acronyms 22](#_Toc43451222)

# C programming

## Volatile

volatile uint8\_t dummy**;**

Require the compilator to read the register:

* for example PORTB change accordingly to external conditions, the compilator can’t say “it’s okay I have already read this register.
* If a register have to be read but is not used, the variable won’t be deleted at compilation.

For example the SPI rx register have to be read before sending an another byte

dummy **=** SPI1RXB**;**

# Bluetooth – Theory

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

## 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

## Bluetooth Low Energy Architecture / BLE Stack

Figure 1: BLE Architecture



Nordic

BLE Softdevice

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.

## 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)

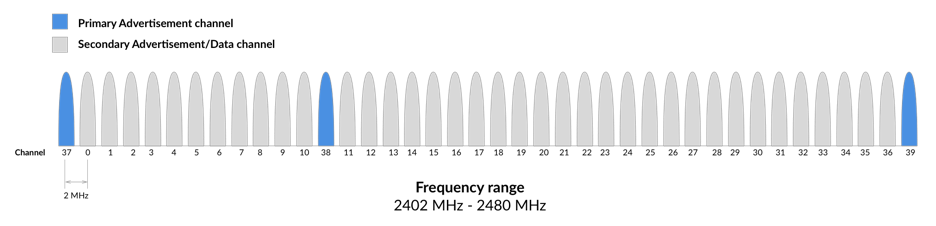


Figure 2: [RF Channels](https://www.novelbits.io/basics-bluetooth-low-energy/)

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.

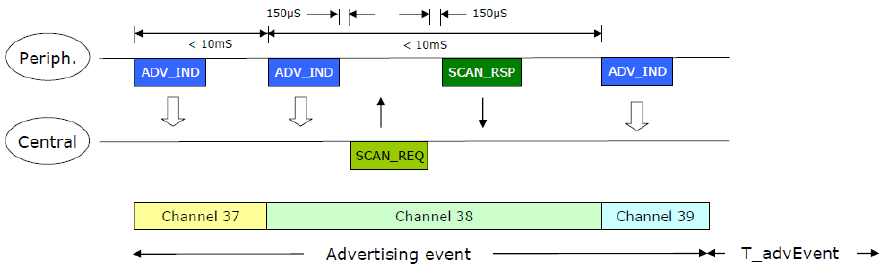


Figure 3: Advertising events

#### 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](https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/)

#### 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](https://microchipdeveloper.com/wireless:ble-link-layer-packet-types#:~:text=Bluetooth%C2%AE%20Low%20Energy%20Packet%20Types,1%20is%202%2D39%20bytes.)

### 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.

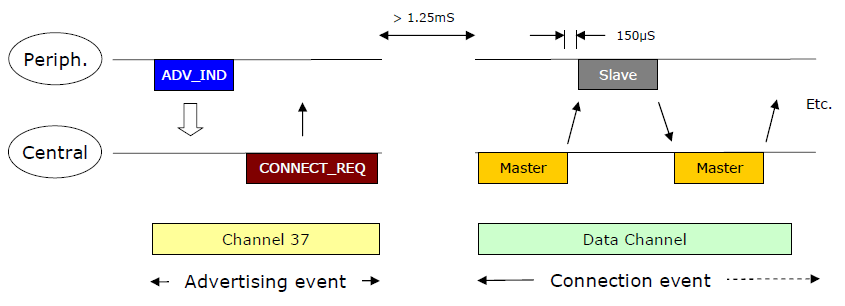


Figure 4: Data Transaction

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](http://what-when-how.com/Tutorial/topic-505haskm/Getting-Started-with-Bluetooth-Low-Energy-64.html), 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.
* **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.

## 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

## 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.

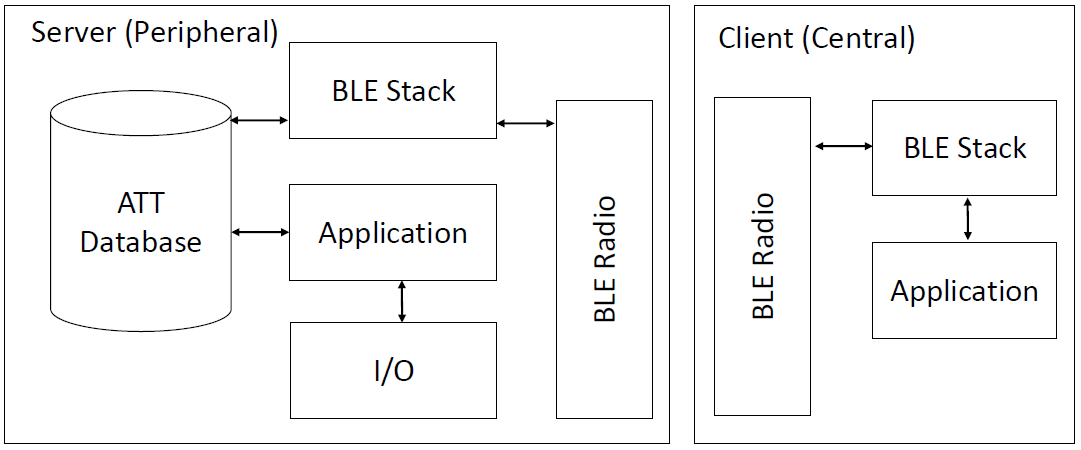


Figure 5: ATT Server - Client

### 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.

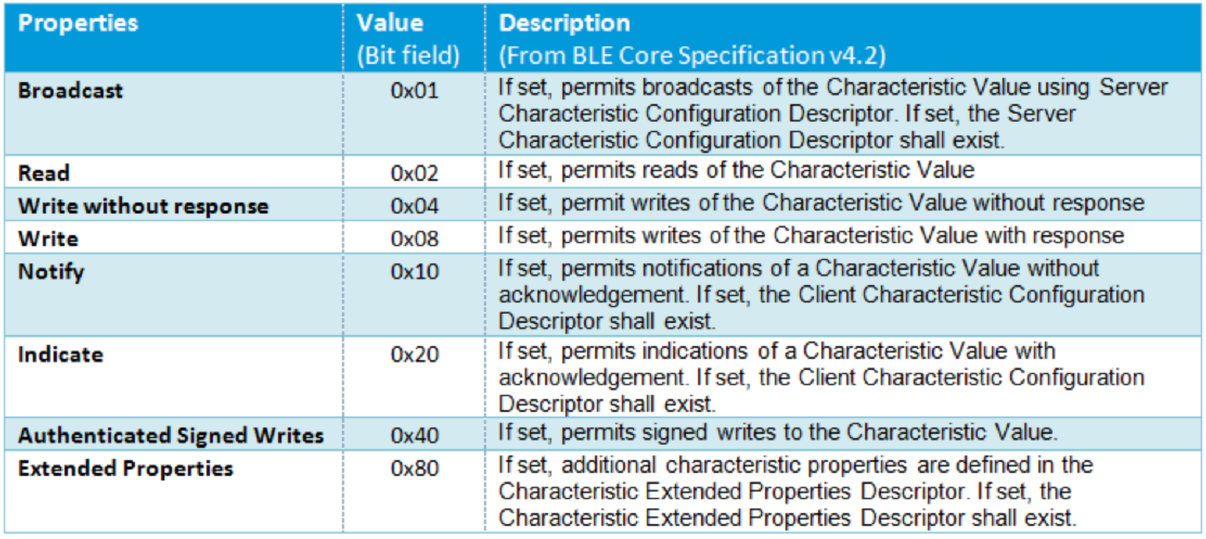
Holds a value or information about where to find other attributes and their properties.

* **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...](https://www.bluetooth.com/specifications/gatt/descriptors/))

##### Attribute properties

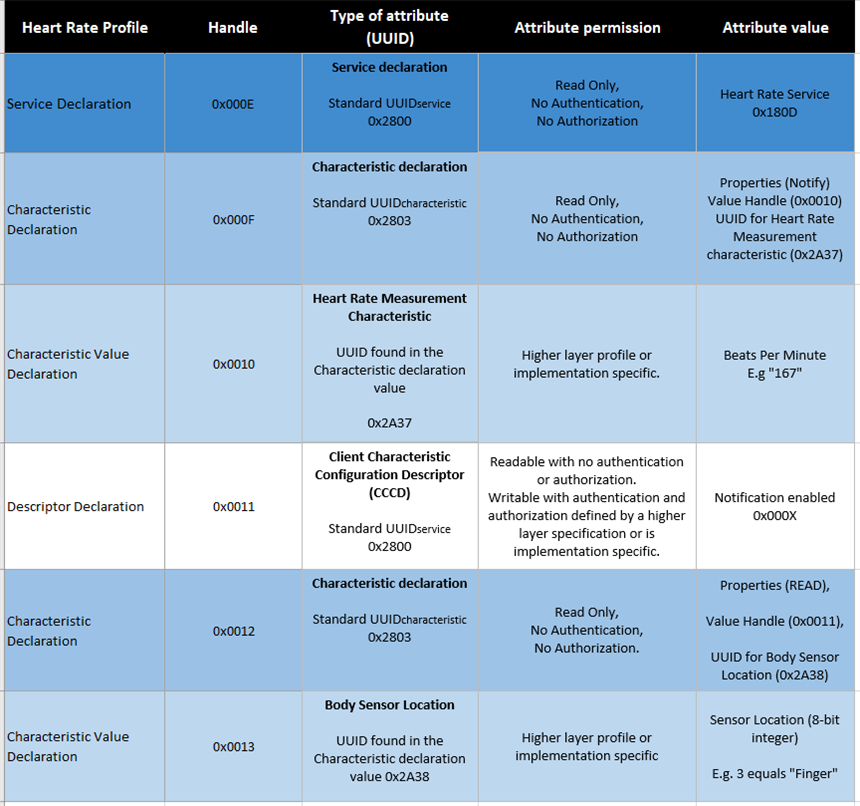
Affect the characteristic Value Declaration, it has a large impact on transmission speed, i.e. write without response is 2 times faster as write with response.



### 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”



# nRF52840 Development Kit

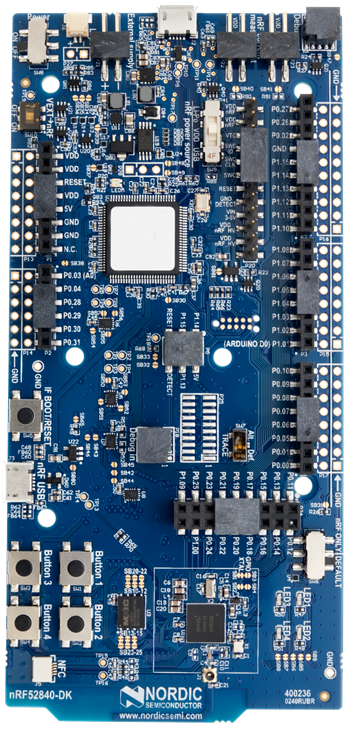


Figure 6: nRF52 DK - pca10056

## 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)

# nRF5 Software

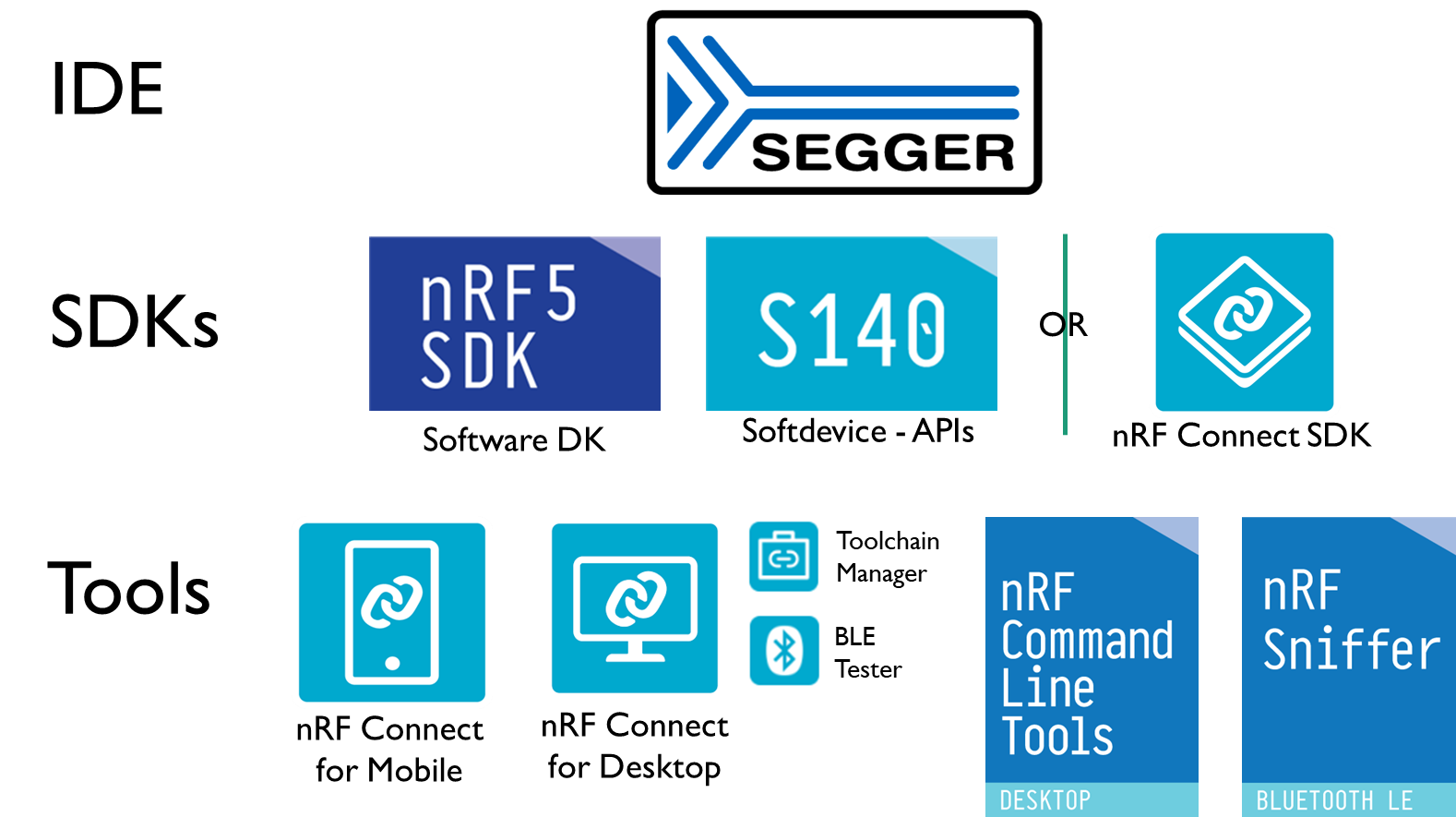


Figure 7 : nRF5 SoftwRE

## 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.



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.

## 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.

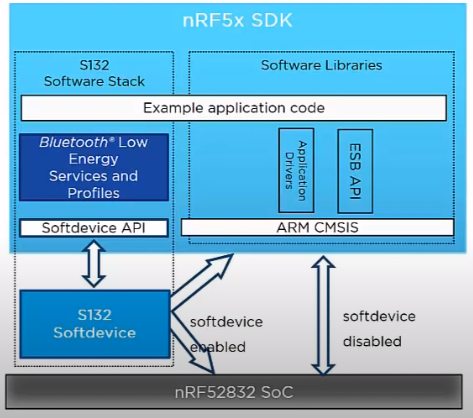
### SoftDevices + nRF5 SDK and Application-specific SDKs

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



9 - SDK Introduction video

* Device Firmware Upgrade (DFU)
* GATT serializer
* Driver support for all peripherals

#### Softdevice 140

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

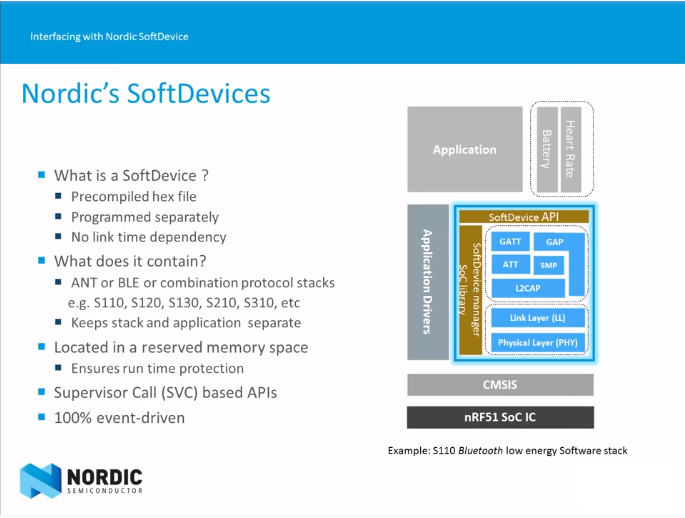
* Bluetooth 5.1 qualified
* High-throughput 2 Mbps
* Long Range
* Advertising Extensions
* CSA #2
* LE Secure Connections
* Privacy 1.2
* Configurable ATT Table
* Configurable ATT MTU
* Custom UUID
* LE Data Packet Length Extension
* L2CAP connection-oriented channels
* Concurrent multiprotocol support

##### Peer Manager - Modules

The [Peer Manager](https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsdk_nrf5_v16.0.0%2Flib_pm_architecture.html) 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.



<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.

## 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

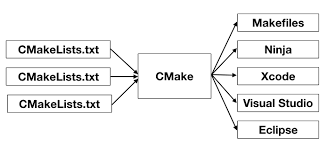
Used for development, programming and debugging of Nordic Semiconductor's nRF52 devices.

* **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.

## How to Build a CMake-Based ProjectCMAKE & GCC

Used by the nRF Connect SDK

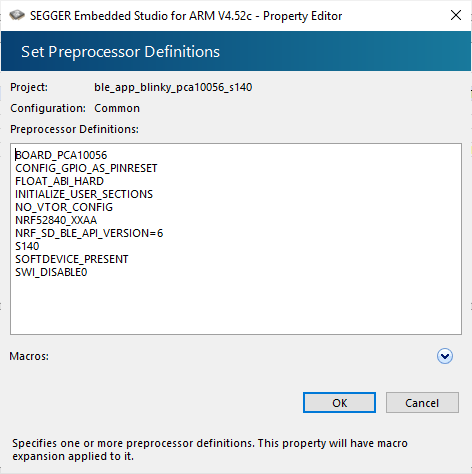
<https://stackoverflow.com/questions/39761924/understanding-roles-of-cmake-make-and-gcc>

# Installation

[Nordic Tools and download](https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_getting_started%2FUG%2Fcommon%2Fnordic_tools.html&cp=1_0_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**
   1. Open xxx\_pca10056**.emProject** to open one project, according to your programmer
4. !!! Make sure you are using the right board:
   1. Project+clkD>>Option>>Code>>Preprocessor>>Preprocessor Definition



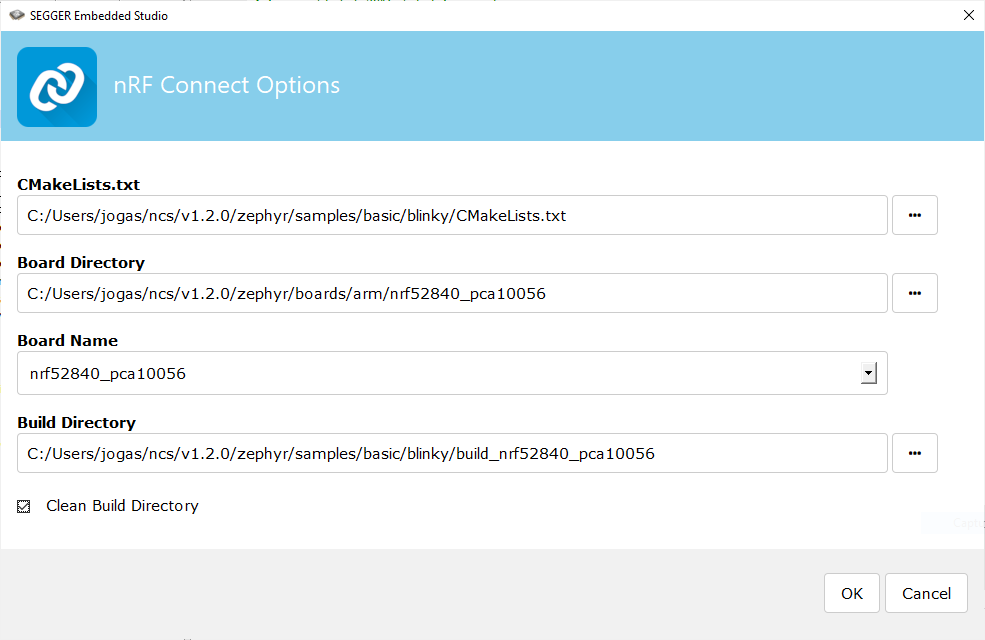
* 1. Project+clkD>>Option>>Debug>>Debugger>>Target Device
  2. Control that nrf\_peripheral.h use the correct define

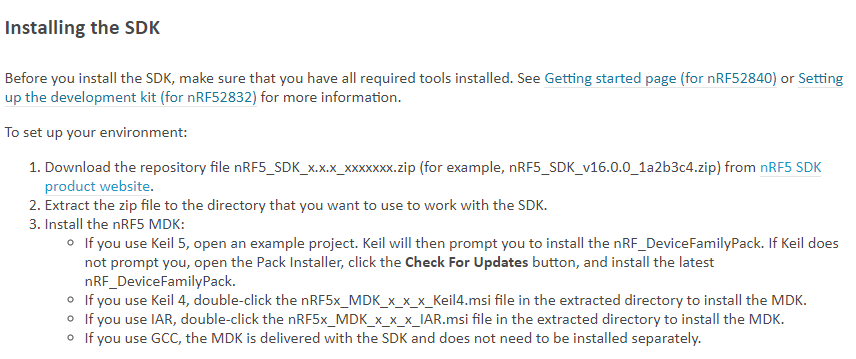
## nRF Connect SDK (NCS

1. Install nRF Connect for Desctop
2. Install the Toolchain Manager app

🡪 Install nRF Connect SDK and Segger Embedded Studio **Nordic Edition**

1. Checkbuild configuration: File>>Open nRF Connect SDK Project





# Segger Embedded Studio

[User guide](http://studio.segger.com/index.htm?http://studio.segger.com/ide_project_explorer.htm)

## Getting started

Getting started with the [software development](https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_getting_started%2FUG%2Fgs%2Fdevelop_sw.html&cp=1_0_2)

[Exemples](http://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/examples.html)

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

## NRF\_LOG\_INFO()

1. Install: [CMSIS Configuration Wizard](https://www.youtube.com/watch?v=b0MxWaAjMco)
2. Sdk\_config.h+ClkD>>CMSIS conf Wizard>>nrf\_log>>

NRF\_LOG\_BACKEND\_RTT\_ENABLED>> Enable

1. NRF\_FPRINTF\_FLAG\_AUTOMATIC\_CR\_ON\_LF\_ENABLED>> Disable

## 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 11: BMS - nRF52840 Debug Connector

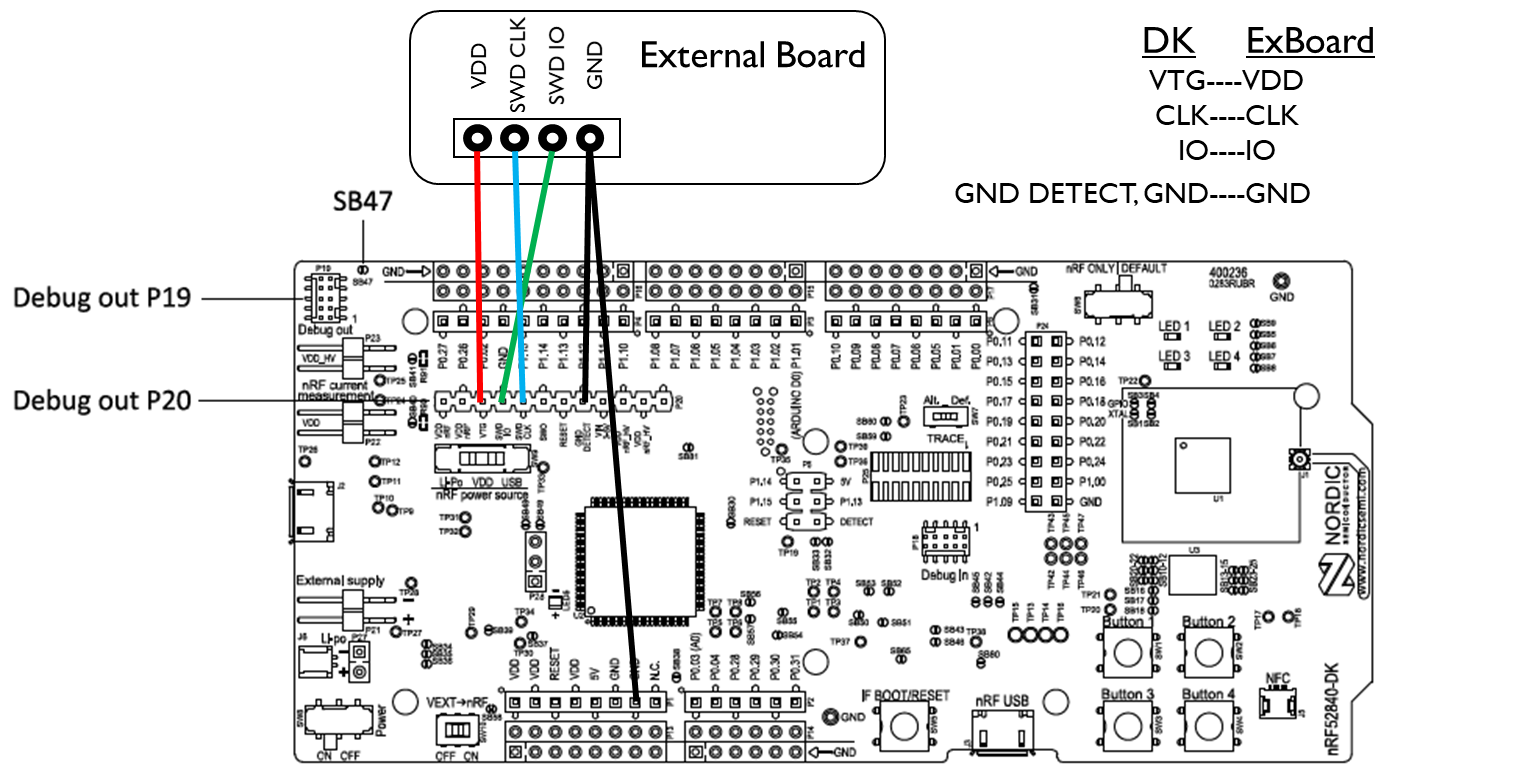


Figure 10: External Board debug connection

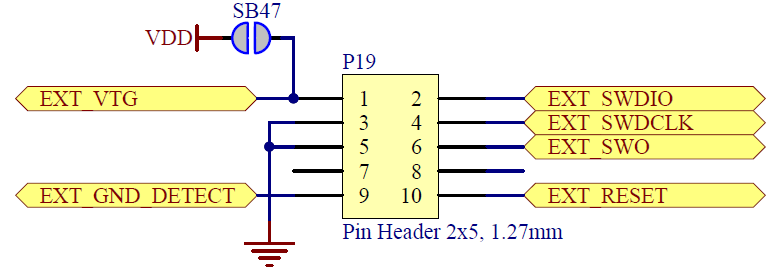


Figure 12: Debug Out p19 Connector (nRF52840 DK Schematic)

## Custom Services error

### Debug tuto

1. Go to this [tutorial](https://www.youtube.com/watch?v=uP8RYgYGRvI&hd=1&feature=youtu.be&t=254)
2. For newer SDKs, go to sdk\_config.h and set "#define NRF\_SDH\_BLE\_VS\_UUID\_COUNT 1"

### Add more RAM

1. 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

## 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 :)

## Error code

Research the code number in those files

* nrf\_error.h (app\_error>>dependencies
* sdk\_errors.h

## Modifying an example

[Modify an example](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/gs_modifying.html)

### 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*

## Configuring your application in Debug

set CONFIG\_DEBUG\_OPTIMIZATIONS to y

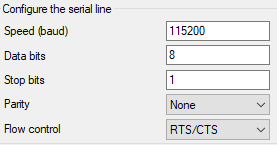
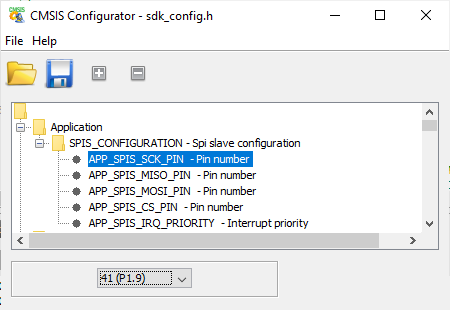
Default configuration for:

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

(Write your devicetree n the Zephyr documentation for more information)

[Kconfig Configuration](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/application/index.html#application-kconfig)

## UART communication

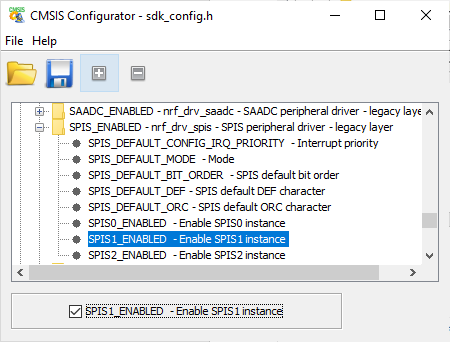


## SPI Slave

Add the drivers

Enable the drivers in **sdk\_config.h** or CMSIS config wizard

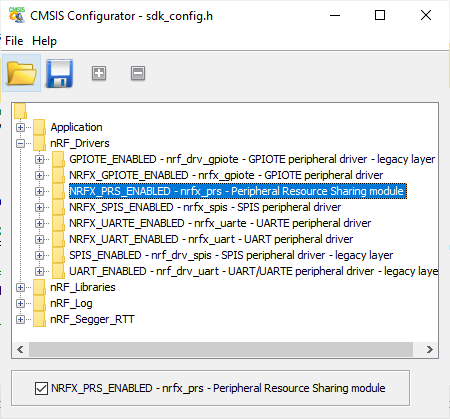
1. Add Pin description to sdk\_config.h

* APP\_SPIS\_CS\_PIN;
* APP\_SPIS\_MISO\_PIN;
* APP\_SPIS\_MOSI\_PIN;
* APP\_SPIS\_SCK\_PIN;

1. Enable SPIS\_ENABLE and SPIS1\_ENABLE
2. Enable the other drivers listed below
3. Import the driver to the project:

Right+Click on nRF\_Driver folder>>Import existing file>>RF5\_SDK\_XX.X.X\_59ac345

* \integration\nrfx\legacy\nrf\_drv\_spis.c
* \modules\nrfx\drivers\src\nrfx\_spis.c



# nRF Command line tool

***nrfjprog -f nrf52 –eraseall***

Erase every thing on the board, also the bootloader, use *nrfjprog --recover –log* to recover and flash a new program

***nrfjprog --recover –log***

Give indication about the nRF state and try to repair it

***nrfjprog -f nrf52 –program***

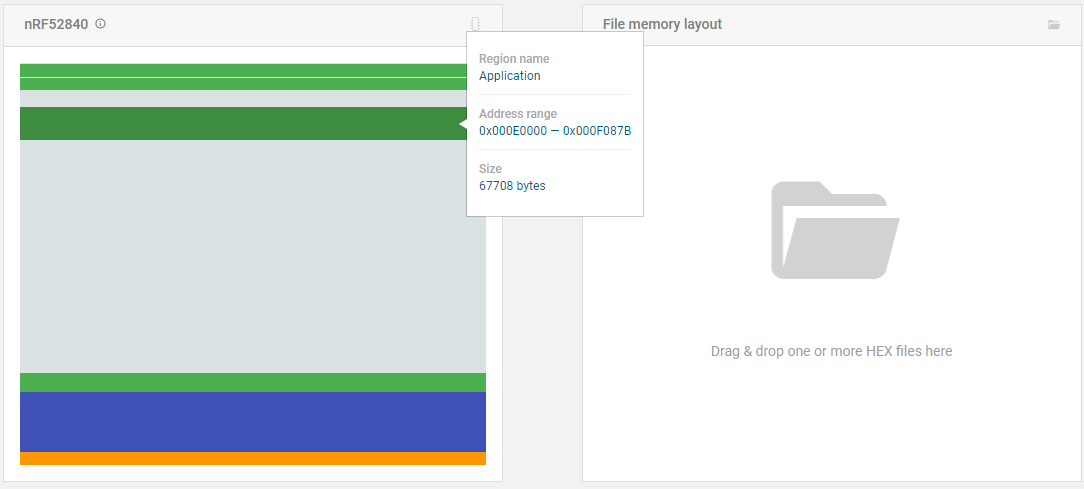
1. Go the the Segger project location

2. This command will flash the .emProject into the nRF

# nRF Connect

## Programmer

It’s a simplified version of the Command line, you also can see the flash content.



# BLE Tutorials

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

## BLE Advertising

[Tuto ble-advertising link](https://devzone.nordicsemi.com/nordic/short-range-guides/b/bluetooth-low-energy/posts/ble-advertising-a-beginners-tutorial)

### Appearance - Add an Icon to a device

The central device can display an icon representing the generic device type of the peripheral device.

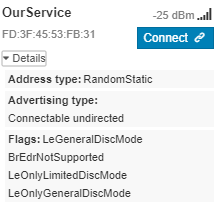
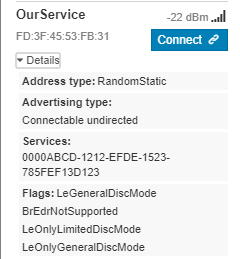
If the device type is conformed to Bluetooth SIG an icon will be displayed automatically.

* In the function gap\_params\_init() add: sd\_ble\_gap\_appearance\_set**(**BLE\_APPEARANCE\_CYCLING\_POWER\_SENSOR**)**

## BLE Services

[Tuto ble-services link](https://devzone.nordicsemi.com/nordic/short-range-guides/b/bluetooth-low-energy/posts/ble-services-a-beginners-tutorial)

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.

## BLE Characteristics

[Tuto ble-characteristics link](https://devzone.nordicsemi.com/nordic/short-range-guides/b/bluetooth-low-energy/posts/ble-characteristics-a-beginners-tutorial)

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

### Adding a characteristic

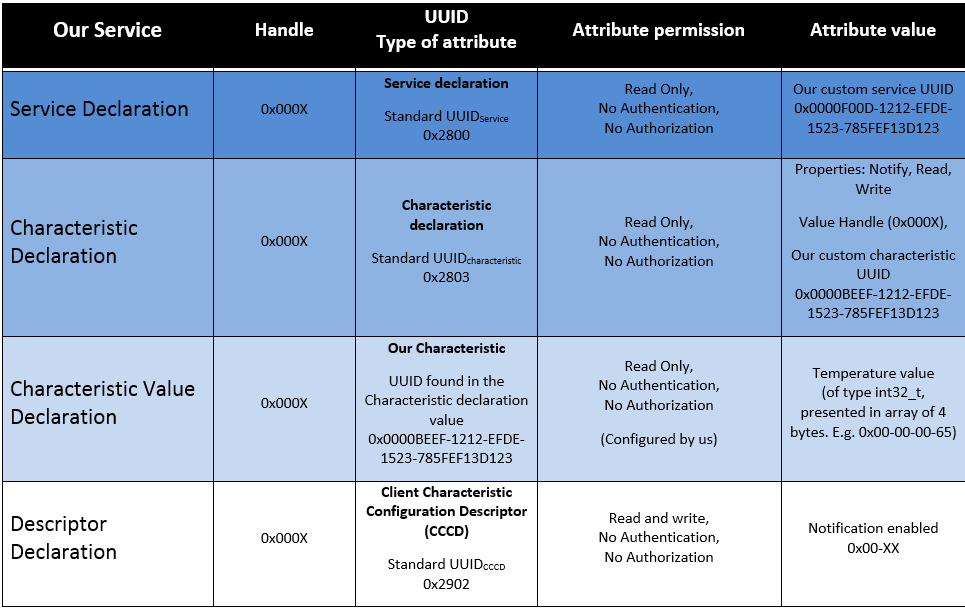


Figure 13: GATT table Tuto-Char

**2A**

**3A**

**2F**

**2E**

**2D**

**2A**

**3B**

**2C**

**2G**

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.

## 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.



The code should look something like that:



**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.



**Step 4**

Now we will be initialising those characteristics



**Step 5**

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



## 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



**Step 2**

In our\_service.h file, you need to add the init structure at the top of the file.



**Step 3**

In our\_service.h file modify the structure.



**Step 4**

In our\_service.c file modify the function.



**Step 5**

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



**Step 6**

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



**Step 7**

In our\_service.c file modify the function.



**Step 8**

In our\_service.c somewhere at the top of the file add this function



# Android Studio

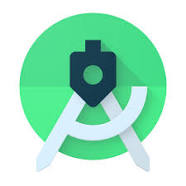


Figure 16: Android Studio

Application IDE

V4.0



Figure 14: Android Studio

Build Tool

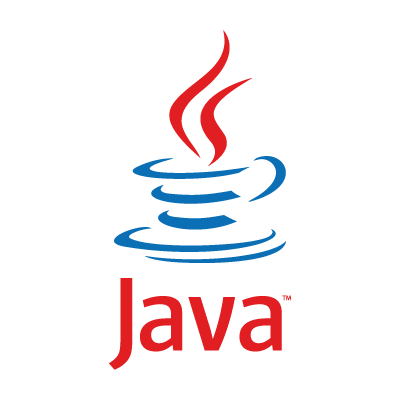


Figure 15: Android Studio supported languages



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

***Gradle***

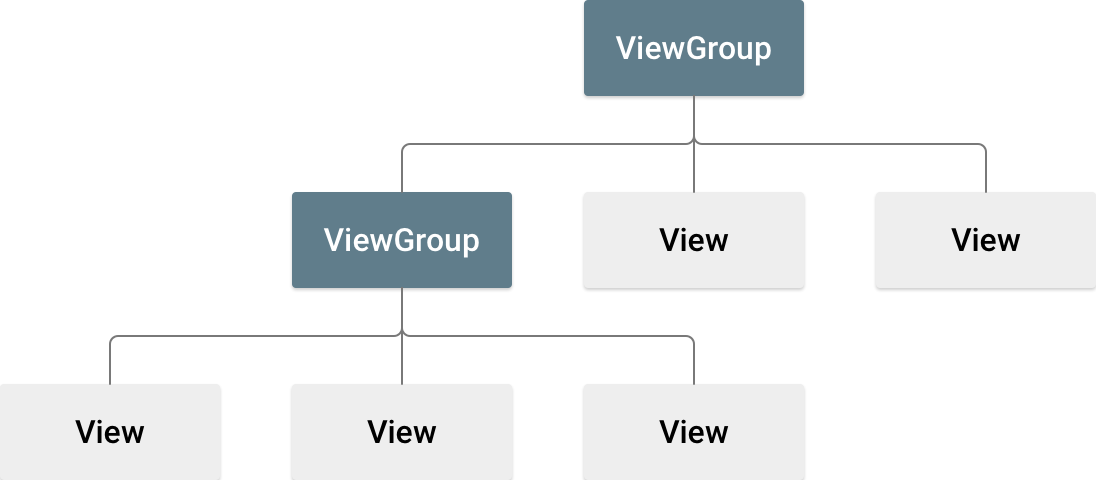
Gradle is an open-source build automation system = CMake for android/java.

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.

## Tutorial

Make your [first app](https://developer.android.com/training/basics/firstapp)

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.

## Java Operator

### Lambda functions

the -> separates the parameters (left-side) from the implementation (right side).

The general syntax for using lambda expressions is

(Parameters) -> { Body } where the -> separates parameters and lambda expression body.

The parameters are enclosed in parentheses which is the same way as for methods and the lambda expression body is a block of code enclosed in braces.

Useful when you want to implement a functional interface:

Runnable r = ()-> System.out.print("Run method");

is equivalent to

Runnable r = new Runnable() {

@Override

public void run() {

System.out.print("Run method");

}

};

### Method references

Double colon operator (“::”)

## String resources - string.xml

A string resource provides text strings for your application with optional text styling and formatting. There are three types of resources that can provide your application with strings:

* [**String**](https://developer.android.com/guide/topics/resources/string-resource#String)

XML resource that provides a single string.

* [**String Array**](https://developer.android.com/guide/topics/resources/string-resource#StringArray)

XML resource that provides an array of strings.

* [**Quantity Strings (Plurals)**](https://developer.android.com/guide/topics/resources/string-resource#Plurals)

XML resource that carries different strings for pluralization.

**Resource reference:**

In Java: R.string.*string\_name*

In XML: @string/*string\_name*

**Application code retrieves a string**



More information to [android.com](https://developer.android.com/guide/topics/resources/string-resource#java)

All the resources in this .xml file are create and use for components, you can assigned one to a component like that:

Select a component>>Under **hint**>>Click on Pick a Ressource (…)

All the application string are located in one file, like that a language translator can easily be installed. You can add French, German, English…

## Drawable - Vector Asset

1. Download one SVG icon on [flaticon.com](https://www.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"**

# MPLABX

Pickit3 connection failed: Run MPLSBX in administrator mode

Ctrl+space for code suggestion

Ctrl+click: enter a function definition

Ctrl+Shift+C: comment a block of code

# 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

