
DTU

GPIOS: Leds & Buttons

Lab2

Device Tree

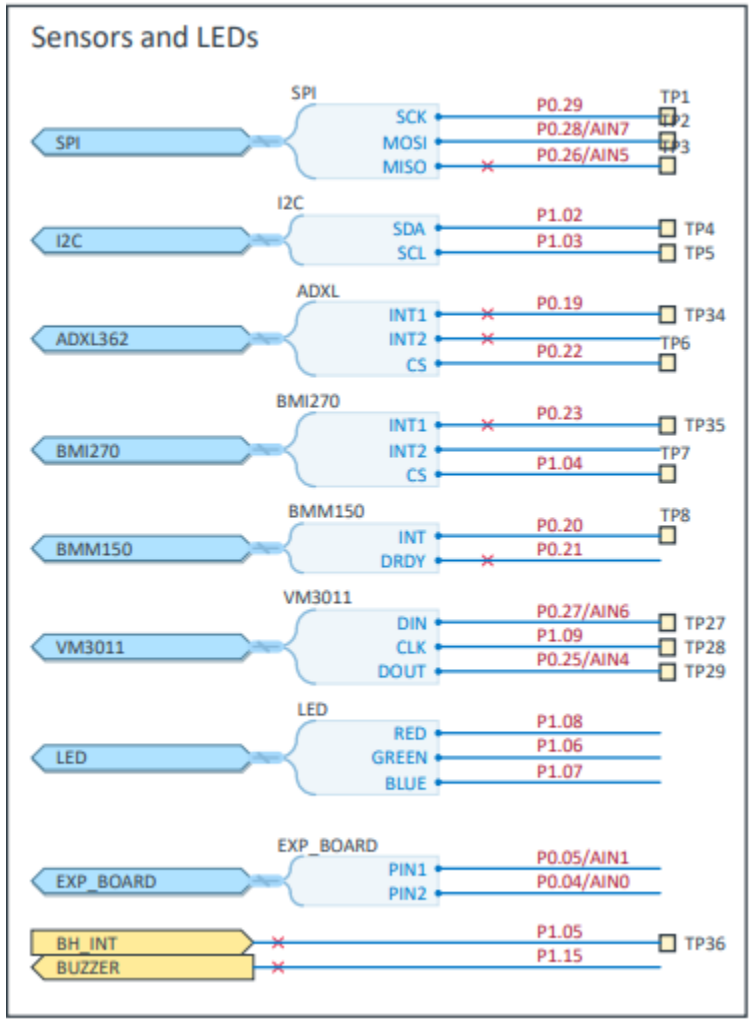
- Specified by file thingy53_nrf5340_common.dtsi

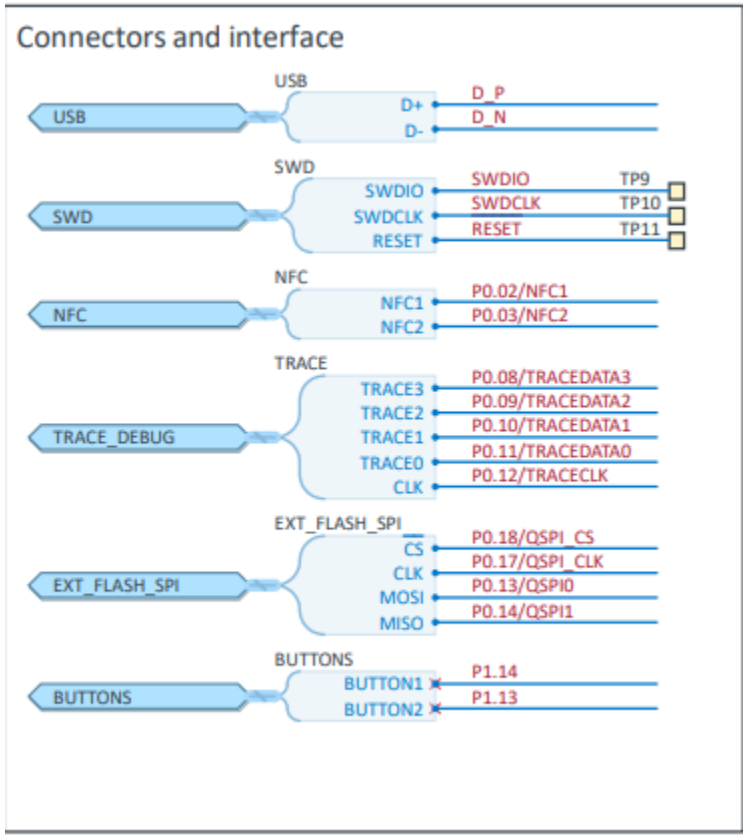
```
leds {
    compatible = "gpio-leds";
    red_led: led_1 {
        gpios = <&gpio1 8 GPIO_ACTIVE_HIGH>;
        label = "RGB red LED";
    };
    green_led: led_2 {
        gpios = <&gpio1 6 GPIO_ACTIVE_HIGH>;
        label = "RGB green LED";
    };
    blue_led: led_3 {
        gpios = <&gpio1 7 GPIO_ACTIVE_HIGH>;
        label = "RGB blue LED";
    };
};
```

```
aliases {
    sw0 = &button0;
    sw1 = &button1;
    led0 = &red_led;
    led1 = &green_led;
    led2 = &blue_led;
    pwm-led0 = &red_led_pwm;
    pwm-led1 = &green_led_pwm;
    pwm-led2 = &blue_led_pwm;
    magn0 = &bmm150;
    watchdog0 = &wdt0;
    accel0 = &adxl362;
    mcuboot-button0 = &button1;
    mcuboot-led0 = &blue_led;
};
```

```
/* The devicetree node identifier for the "led2" alias. */
#define LED2_NODE DT_ALIAS(led2)
```

Schematics


Sensors and LEDs
SPI
SPI
SCK
MOSI
MISO
P0.29
P0.28/AIN7
P0.26/AIN5
TP1
TP2
TP3
I2C
I2C
SDA
SCL
P1.02
P1.03
TP4
TP5
ADXL
ADXL362
INT1
INT2
CS
P0.19
P0.22
TP34
TP6
BMI270
BMI270
INT1
INT2
CS
P0.23
P1.04
TP35
TP7
BMM150
BMM150
INT
DRDY
P0.20
P0.21
TP8
VM3011
VM3011
DIN
CLK
DOUT
P0.27/AIN6
P1.09
P0.25/AIN4
TP27
TP28
TP29
LED
LED
RED
GREEN
BLUE
P1.08
P1.06
P1.07
EXP_BOARD
EXP_BOARD
PIN1
PIN2
P0.05/AIN1
P0.04/AIN0
BH_INT
BUZZER
P1.05
P1.15
TP36


Connectors and interface
USB
USB
D+
D-
D_P
D_N
SWD
SWD
SWDIO
SWDCLK
RESET
SWDIO
SWDCLK
RESET
TP9
TP10
TP11
NFC
NFC
NFC1
NFC2
P0.02/NFC1
P0.03/NFC2
TRACE
TRACE_DEBUG
TRACE3
TRACE2
TRACE1
TRACE0
CLK
P0.08/TRACEDATA3
P0.09/TRACEDATA2
P0.10/TRACEDATA1
P0.11/TRACEDATA0
P0.12/TRACECLK
EXT_FLASH_SPI
EXT_FLASH_SPI
CS
CLK
MOSI
MISO
P0.18/QSPI_CS
P0.17/QSPI_CLK
P0.13/QSPI0
P0.14/QSPI1
BUTTONS
BUTTONS
BUTTON1
BUTTON2
P1.14
P1.13

Polling based solution

```c
while (1) {
    /* STEP 6.1 - Read the status of the button and store it */
    bool val = gpio_pin_get_dt(&button);

    /* STEP 6.2 - Update the LED to the status of the button */
    gpio_pin_set_dt(&led, val);

    k_msleep(SLEEP_TIME_MS); // Put the main thread to sleep for 100ms for power
                             // optimization
}
```

- The device checks the pin's status every 100ms
- Even if the button is not pressed the gpio status will be checked periodically
- Worst case delay is 100ms between the pressing of the button and the led turning on

Interrupt based solution

```c
/* STEP 4 - Define the callback function */
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    gpio_pin_toggle_dt(&led);
}
```

```c
/* STEP 3 - Configure the interrupt on the button's pin */
ret = gpio_pin_interrupt_configure_dt(&button, GPIO_INT_EDGE_TO_ACTIVE);

/* STEP 6 - Initialize the static struct gpio_callback variable   */
gpio_init_callback(&button_cb_data, button_pressed, BIT(button.pin));

/* STEP 7 - Add the callback function by calling gpio_add_callback()   */
gpio_add_callback(button.port, &button_cb_data);
```

- The device wakes up only when the button is pressed

Interrupt based solution

```
ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_INACTIVE);
```

- Configure the led to be initialized as inactive

```
ret = gpio_pin_interrupt_configure_dt(&button, GPIO_INT_EDGE_BOTH);
```

- GPIO pin interrupt will be triggered on pin rising or falling edge

nRF Connect SDK Apps

Lab3

Device Tree overlays

- The Device Tree datastructure describes hardware
- If we wish to modify properties of it, we can do it through overlay files
- Overlay files has the extension ".overlay", and should be placed in root directory

```
&spi1{
  status = "okay";
};

&pinctrl {
  spi1_default: spi1_default {
    group1 {
      psels = <NRF_PSEL(SPIM_MOSI, 0, 25)>;
    };
  };
  spi1_sleep: spi1_sleep {
    group1 {
      psels = <NRF_PSEL(SPIM_MOSI, 0, 25)>;
    };
  };
};
```

```
&spi1{
  status = "okay";
};
```

Enables spi1 node

```
&pinctrl {
  spi1_default: spi1_default {
    group1 {
      psels = <NRF_PSEL(SPIM_MOSI, 0, 25)>;
    };
  };
  spi1_sleep: spi1_sleep {
    group1 {
      psels = <NRF_PSEL(SPIM_MOSI, 0, 25)>;
    };
  };
};
```

Changes SPIM_MOSI to pin 25, in both default and sleep states

Configuration files

- Each nRF application has an application configuration file, usually called prj.conf
- Each line in the configuration specifies whether a module is enabled for the build

```
CONFIG_<symbol_name>=<value>
```

Prefix | Specific module | Value: y/n

Thingy:53 config file

```
# SPDX-License-Identifier: Apache-2.0

# Enable MPU
CONFIG_ARM_MPU=y

# Enable hardware stack protection
CONFIG_HW_STACK_PROTECTION=y

# Enable TrustZone-M
CONFIG_ARM_TRUSTZONE_M=y

# This Board implies building Non-Secure firmware
CONFIG_TRUSTED_EXECUTION_NONSECURE=y

# Enable GPIO
CONFIG_GPIO=y
```

"Hello World" terminal app

We create an application with the minimum amount of files needed:

- prj.conf
- CMakeLists.txt
- Src/main.c

CMakeLists.txt:

```
# Build specifications
cmake_minimum_required(VERSION 3.20.0)# Min CMake version required to build
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})# Creates CMake target
project(hello_world)# Name of project
target_sources(app PRIVATE src/main.c)# Add source file to build
```

Main.c:

```
1. #include <zephyr/kernel.h>
2. #include <zephyr/sys/printk.h>
3. int main(void){
4.     while (1) {
5.         // Loop: Prints, then sleeps for 1 sec
6.         printk("Hello World!\n");
7.         k_msleep(1000);
8. }}
```

Build

Terminal output:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Custom source files

Now, we want to include custom c code, found in other files than main.c

Custom file, myfunction.c:

```c
#include "myfunction.h"

int sum(int a, int b){
    return a+b;
}
```

Myfunction.h header file:

```c
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

int sum(int a, int b);

#endif
```

Add to CMake file:

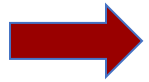
```
target_sources(app PRIVATE src/myfunction.c)
```

Add include in main.c:

```c
#include "myfunction.h"
```

Change main.c to use new code:

```c
int main(void)
{
  int a = 3, b = 4;
  while(1){
    printk("The sum of %d and %d is %d\n", a, b, sum(a,b));
    k_msleep(1000);
  }
}
```



Terminal output:

```
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
```

Adding custom configurations

Another way to include custom code. Allows custom code paths based on enabling or not. This is to avoid including functions in the build, that are not used for the particular application

"Kconfig" file, in app directory:

```
source "Kconfig.zephyr"

config MYFUNCTION
  bool "Enable my function"
  default n
```

Modify header in main.c:

```
#ifdef CONFIG_MYFUNCTION
#include "myfunction.h"
#endif
```

Add this to CMake file, instead of old

```
7   #target_sources(app PRIVATE src/myfunction.c)
8   target_sources_ifdef(CONFIG_MYFUNCTION app PRIVATE src/myfunction.c)
```

Eanble in prj.conf:

```
CONFIG_MYFUNCTION=y
```

```
int main(void)
{

  while (1) {
#ifdef CONFIG_MYFUNCTION
    int a = 3, b = 4;
    printk("The sum of %d and %d is %d\n", a, b, sum(a, b));
#else
    printk("MYFUNCTION not enabled\n");
    return 0;
#endif
    k_msleep(1000);
  }
  return 0;
}
```

If enabled

```
The sum of 3 and 4 is 7
The sum of 3 and 4 is 7
```

If NOT enabled

```
MYFUNCTION not enabled
```

Changing the baud rate through overlay

- Create a folder called "boards"
- Create an overlay file with the name of the board: board.overlay
- For the thingy53 -> thingy53_nrf5340.overlay

Change the baud rate in the .overlay file

```
&uart0 {
    current-speed = <9600>;
};
```

This determines the speed at which the device sends information to the terminal

Printing and Logging

Lab 4

Exercise 1 – Printing to the console

- Use printk() to send messages from your application.

1) Include the header file of the printk() function:

```
#include <zephyr/sys/printk.h>
```

2) Print a simple banner:

```
printk("nRF Connect SDK Fundamentals - Lesson 4 - Exercise 1\n");
```

```
*** Booting nRF Connect SDK ***
*** Using Zephyr OS ***
nRF Connect SDK Fundamentals - Lesson 4 - Exercise 1
```

Exercise 1 – Printing to the console

- Use printk() to send messages from your application.

Define the macro **MAX_NUMBER_FACT:**

```
#define MAX_NUMBER_FACT 10
```

Replace the callback function **button_pressed():**

```
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    int i;
    long int factorial = 1;

    printk("Calculating the factorials of numbers from 1 to %d:\n", MAX_NUMBER_FACT);
    for (i = 1; i <= MAX_NUMBER_FACT; i++) {
        factorial = factorial * i;
        printk("The factorial of %2d = %ld\n", i, factorial);
    }
    printk("_____\n");
```

Formatted string that contains one integer %d :

```
printk("Calculating the factorials of numbers from 1 to %d:\n", MAX_NUMBER_FACT);
```

Formatted string containing an integer with a width specifier of %2d and a long integer %ld :

```
printk("The factorial of %2d = %ld\n", i, factorial);
```

String that contains one long underline

```
printk("_____\n");
```

Exercise 1 – Printing to the console

- Use printk() to send messages from your application.

Terminal output:

```c
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    int i;
    long int factorial = 1;

    printk("Calculating the factorials of numbers from 1 to %d:\n", MAX_NUMBER_FACT);
    for (i = 1; i <= MAX_NUMBER_FACT; i++) {
        factorial = factorial * i;
        printk("The factorial of %2d = %ld\n", i, factorial);
    }
    printk("_____\n");
```

```
*** Booting nRF Connect SDK ***
*** Using Zephyr OS ***
nRF Connect SDK Fundamentals - Lesson 4 - Exercise 1
Calculating the factorials of numbers from 1 to 10:
The factorial of  1 = 1
The factorial of  2 = 2
The factorial of  3 = 6
The factorial of  4 = 24
The factorial of  5 = 120
The factorial of  6 = 720
The factorial of  7 = 5040
The factorial of  8 = 40320
The factorial of  9 = 362880
The factorial of 10 = 3628800
```

Exercise 2 – Using the Logger module

1) Enable the logger module in the application configuration file prj.conf :

```
CONFIG_LOG=y
```

2) Include the header file of the logger module:

```
#include <zephyr/logging/log.h>
```

3) Register your code with the logger module.

```
LOG_MODULE_REGISTER(Less4_Exer2,LOG_LEVEL_DBG);
```

Exercise 2 – Using the Logger module

4) Print logging information:

```
int exercise_num = 2;
uint8_t data[] = {0x00, 0x01, 0x02, 0x03,
                  0x04, 0x05, 0x06, 0x07,
                  'H', 'e', 'l', 'l','o'};
// Printf-like messages
LOG_INF("nRF Connect SDK Fundamentals");
LOG_INF("Exercise %d",exercise_num);
LOG_DBG("A log message in debug level");
LOG_WRN("A log message in warning level!");
LOG_ERR("A log message in Error level!");
// Hexdump some data
LOG_HEXDUMP_INF(data, sizeof(data),"Sample Data!");
```

Exercise 2 – Using the Logger module

4) Print logging information:

```
int exercise_num = 2;
uint8_t data[] = {0x00, 0x01, 0x02, 0x03,
                  0x04, 0x05, 0x06, 0x07,
                  'H', 'e', 'l', 'l','o'};
// Printf-like messages
LOG_INF("nRF Connect SDK Fundamentals");
LOG_INF("Exercise %d",exercise_num);
LOG_DBG("A log message in debug level");
LOG_WRN("A log message in warning level!");
LOG_ERR("A log message in Error level!");
// Hexdump some data
LOG_HEXDUMP_INF(data, sizeof(data),"Sample Data!");
```

Terminal output:

```
*** Booting nRF Connect SDK v3.1.1-e2a97fe2578a ***
*** Using Zephyr OS v4.1.99-ff8f0c579eeb ***
[00:00:00.007,995] <inf> Less4_Exer2: nRF Connect SDK Fundamentals
[00:00:00.007,995] <inf> Less4_Exer2: Exercise 2
[00:00:00.008,026] <dbg> Less4_Exer2: main: A log message in debug level
[00:00:00.008,026] <wrn> Less4_Exer2: A log message in warning level!
[00:00:00.008,026] <err> Less4_Exer2: A log message in Error level!
[00:00:00.008,056] <inf> Less4_Exer2: Sample Data!
                                      00 01 02 03 04 05 06 07  48 65 6c 6c 6f          |........ Hello
```

Exercise 2 – Using the Logger module

4) Change the callback function **button_pressed()** to use the asynchronous logger API:

```
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    int i;
    long int factorial = 1;

    LOG_INF("Calculating the factorials of numbers 1 to %d:",MAX_NUMBER_FACT);
    for (i = 1; i <= MAX_NUMBER_FACT; i++) {
        factorial = factorial * i;
        LOG_INF("The factorial of %2d = %ld",i,factorial);
    }
```

Exercise 2 – Using the Logger module

4) Change the callback function **button_pressed()** to use the asynchronous logger API:

```c
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    int i;
    long int factorial = 1;

    LOG_INF("Calculating the factorials of numbers 1 to %d:",MAX_NUMBER_FACT);
    for (i = 1; i <= MAX_NUMBER_FACT; i++) {
        factorial = factorial * i;
        LOG_INF("The factorial of %2d = %ld",i,factorial);
    }
```

Terminal output:

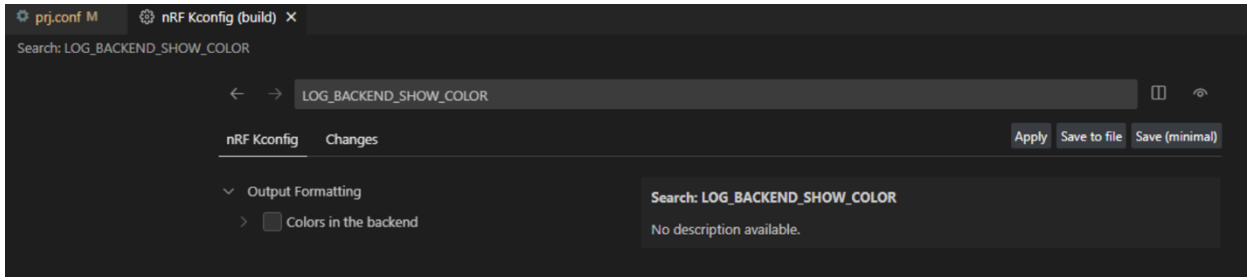
```
[00:24:47.245,513] <inf> Less4_Exer2: Calculating the factorials of numbers 1 to 10:
[00:24:47.245,544] <inf> Less4_Exer2: The factorial of  1 = 1
[00:24:47.245,544] <inf> Less4_Exer2: The factorial of  2 = 2
[00:24:47.245,544] <inf> Less4_Exer2: The factorial of  3 = 6
[00:24:47.245,574] <inf> Less4_Exer2: The factorial of  4 = 24
[00:24:47.245,574] <inf> Less4_Exer2: The factorial of  5 = 120
[00:24:47.245,574] <inf> Less4_Exer2: The factorial of  6 = 720
[00:24:47.245,574] <inf> Less4_Exer2: The factorial of  7 = 5040
[00:24:47.245,574] <inf> Less4_Exer2: The factorial of  8 = 40320
[00:24:47.245,605] <inf> Less4_Exer2: The factorial of  9 = 362880
[00:24:47.245,635] <inf> Less4_Exer2: The factorial of 10 = 3628800
```

Exercise 3 – Exploring the Logger module features

1) Disable colors in the backend in the application configuration file prj.conf :

```
CONFIG_LOG_BACKEND_SHOW_COLOR=n
```

2) Confirm that this feature has been disabled by opening Kconfig:



Terminal output (the error and warning messages are no longer colored) :

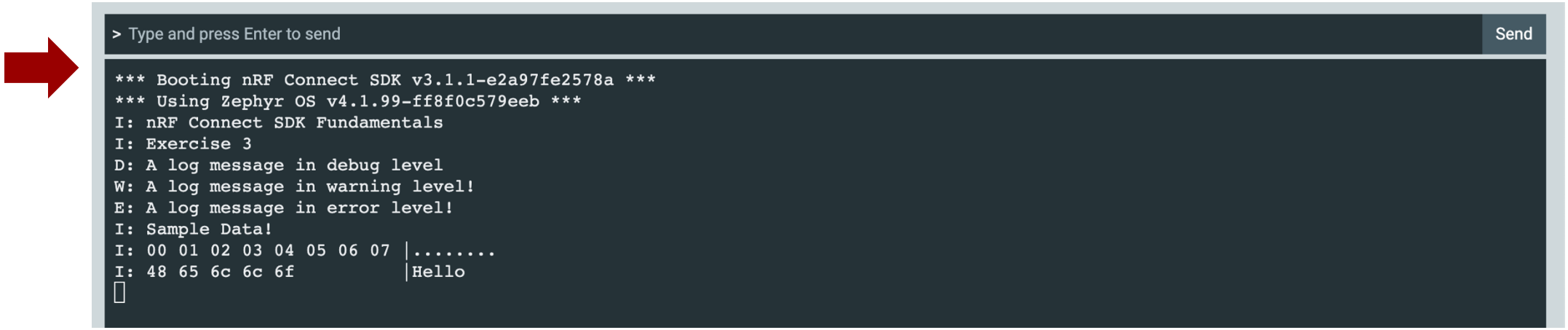
```
*** Booting nRF Connect SDK v3.1.1-e2a97fe2578a ***
*** Using Zephyr OS v4.1.99-ff8f0c579eeb ***
[00:00:00.007,995] <inf> Less4_Exer3: nRF Connect SDK Fundamentals
[00:00:00.007,995] <inf> Less4_Exer3: Exercise 3
[00:00:00.008,026] <dbg> Less4_Exer3: main: A log message in debug level
[00:00:00.008,026] <wrn> Less4_Exer3: A log message in warning level!
[00:00:00.008,026] <err> Less4_Exer3: A log message in error level!
[00:00:00.008,056] <inf> Less4_Exer3: Sample Data!
                                      00 01 02 03 04 05 06 07  48 65 6c 6c 6f          |........ Hello
```

Exercise 3 – Exploring the Logger module features

3) Enable a minimal logging implementation in the application configuration file prj.conf :

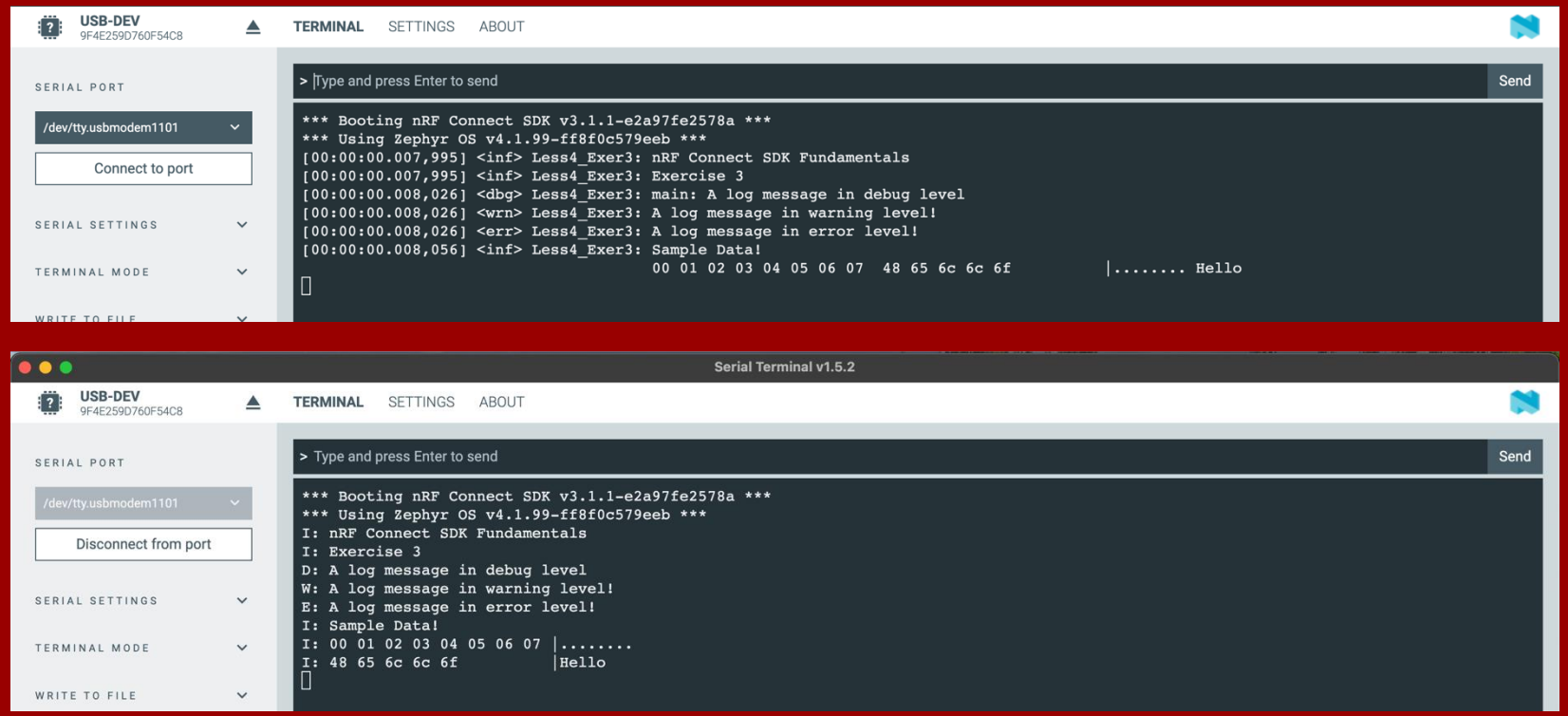
```
CONFIG_LOG_MODE_MINIMAL=y
```

Terminal output (log levels are indicated by letters and timestamps and prefixed module names are not displayed):



Exercise 3 – Exploring the Logger module features

(DEMO)



Extra Task – Filtering Error Messages

- **Goal**: Reduce log clutter and display **only error messages** during debugging.
- **Solution**: Override the log level in prj.conf with CONFIG_LOG_OVERRIDE_LEVEL=1.

  `CONFIG_LOG_OVERRIDE_LEVEL`: It overrides module logging level when it is not set or set lower than the override value.

- **Result**: All messages below the error level are ignored; only LOG_ERR() messages appear in the terminal.

Terminal output:


DTU

DTU

I2C communication with sensors

NES Lab 5

First things first . . .

1. Clone the exercise repo:
   - `git clone https://github.com/NordicDeveloperAcademy/ncs-fund`
2. In VS Code do:
   1. `Welcome view`.
   2. `Open an existing application` using the `l6_e2` folder from the exercise repo.
   3. Implement missing code in `main.c` and `prj.conf` according to the instructions on `DevAcademy`.
   4. Build project.
   5. Check out `l6_e2_sol` for the polling based solution.
3. Flash to Thingy53 using the nRF-Connect desktop GUI
4. Connect to Thingy53 using your favorit serial terminal
   - `ls /dev/ttyACM*`
   - `picocom -b 115200 /dev/ttyACM0`

Interrupt based sensor reading

Startup

1. Configure sensor for interrupt mode:
   - As in `l6_e2_sol` + write 0x11 to sensor interrupt reg. (0x60)
   - Clear sensor latch by reading interrupt reg. (0x60)
2. Configure interrupt pin (like in l6_e2_sol):
   - `#include <zephyr/drivers/gpio.h>`
   - `int_pin = GPIO_DT_SPEC_GET(I2C_NODE, int_gpios);`
   - `gpio_pin_configure_dt(&int_pin, GPIO_INPUT);`
   - `gpio_pin_interrupt_configure_dt(&int_pin, GPIO_INT_LEVEL_LOW);`

Runtime loop

INT low → ISR: disable IRQ → submit work → Work: check VALID → burst-read RGB → print → clear latch → re-enable IRQ

`main.c` (no error checking so it fits on a slide)

```
int main(void) { // H
  // Configure sensor and interrupt pin
  bh1749_configure();
  gpio_pin_configure_dt(&int_pin, GPIO_INPUT);
  gpio_pin_interrupt_configure_dt(&int_pin,
                                  GPIO_INT_LEVEL_LOW);

  // Init. work and callback struct and register callback
  k_work_init(&bh1749_work, bh1749_work_handler);
  gpio_init_callback(&bh1749_cb_data,
                     bh1749_isr,
                     BIT(int_pin.pin));
  gpio_add_callback(int_pin.port,
                    &bh1749_cb_data);
  while (1) { k_sleep(K_FOREVER); } // Sleep forever
```

bh1749_isr

```
static void bh1749_isr(const struct device *dev,
                       struct gpio_callback *cb,
                       uint32_t pins) {

  // Disable further interrupts until work is done
  gpio_pin_interrupt_configure_dt(&int_pin,
                                  GPIO_INT_DISABLE);
  k_work_submit(&bh1749_work); // Schedule work item
}
```

bh1749_work_handler

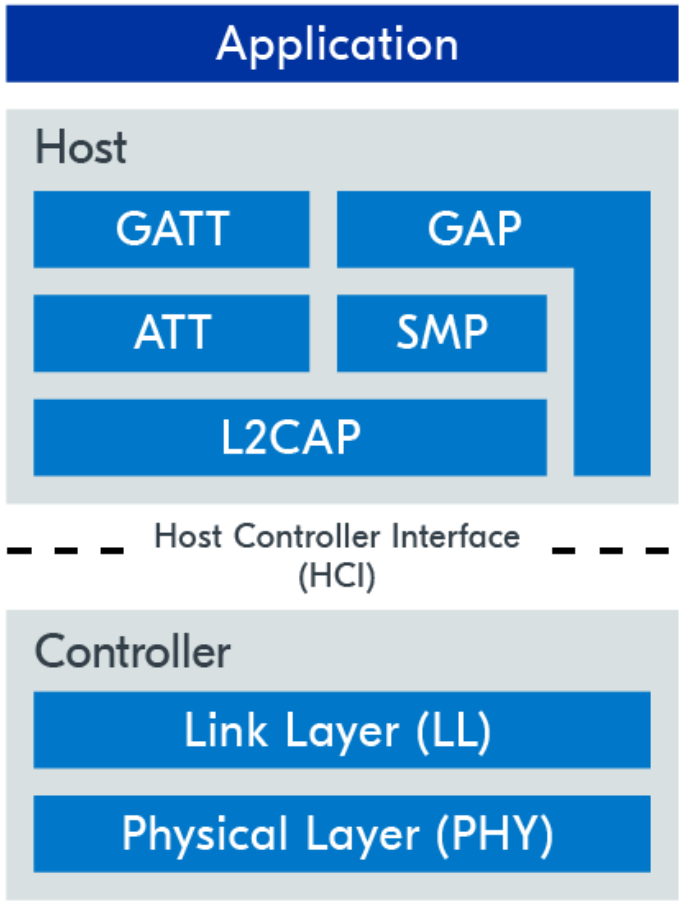
```
static void bh1749_work_handler(struct k_work *work) {
  bh1749_read_and_print(); // Do the work
  uint8_t interrupt_status;
  int ret = i2c_reg_read_byte_dt(&dev_i2c,
                                 BH1749_INTERRUPT,
                                 &interrupt_status);
  if (ret != 0) {
    printk("Failed to clear BH1749 interrupt");
    return; }
  ret = gpio_pin_interrupt_configure_dt(&int_pin,
                                        GPIO_INT_LEVEL_LOW);
  if (ret < 0) {
    printk("Failed to re-enable BH1749 interrupt");
  }
}
```

nRF Connect SDK Apps

Lab 6

Bluetooth LE Introduction

- Used in low-power wearables or massive IoT applications
- Frequent battery charging is not feasible
- High data-transfer speeds are not required



GAP: Device roles and topologies

Communication Styles	
Connection-oriented	Dedicated connection between devices
Broadcast	Broadcast of data packets

Network Topologies	
Broadcast	Data transfer without establishing a connection. Data are broadcasted to any device in range to receive them.
Connected	Connection is established before data transfer. Communication is bi-directional
Multi-role	Devices can both act as Central and Peripheral.

Device Roles	
Advertising	Transmission of advertising packets
Scanning	Listening for advertising packets
Central	Scans and initiates connections with peripherals
Peripheral	Advertises and accepts connections from centrals
Broadcaster	Peripheral that broadcasts advertisement packets without accepting connection requests
Observer	Listens to advertising packets without initiating a connection

PHY: Radio modes

- Physical layer of Bluetooth LE stack

1M PHY

- 1 Mbps
- Mode that will be used when a connection is initiated

2M PHY

- 2 Mbps
- Introduced in Bluetooth v5.0
- Data transmitted at higher data rate, radio needs to stay on for less time
- Decreases receiver sensitivity

Coded PHY

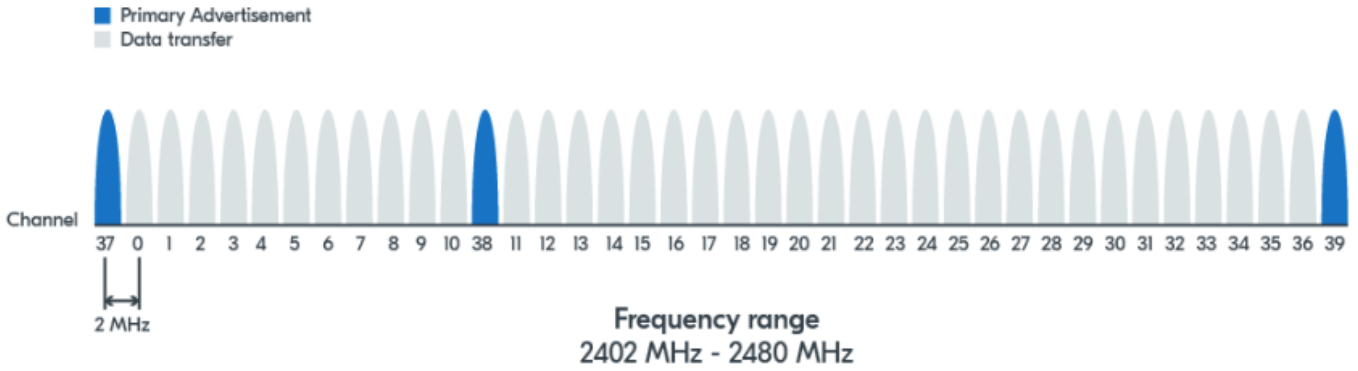
- Longer communication range
- Sacrifices data rate
- Coding schemes to correct packet errors
- A single bit is represented by more than one symbol

nRF Connect SDK Apps

Lab 7

Brief theory

- Bluetooth LE devices can enter **advertising** state
  - This means it sends out advertising packets periodically in predefined intervals, to anounce that it is present and to (potentially) connect to other devices
- Bluetooth LE devices communicate through 40 channels
  - 3 of them are primary channels (37,38,39), used mainly for advertising packets
  - The rest are secondary, mainly for data transfers but also sometimes for advertisement



  - When a device is in advertising state, other devices can send a request for additional data
    - This is a **scan request**
    - The response is called a **scan response**, transmited back over the primary channels

Brief theory

Connectable vs. non-connectable: Determines whether the central can connect to the peripheral or not.

Scannable vs. non-scannable: Determines if the peripheral accepts scan requests from a scanner.

Directed vs. undirected: Determines whether advertisement packets are targeted to a specific scanner or not.

	Connectable	Scannable	Directed
ADV_IND	x	x	
ADV_DIRECT_IND	x		x
ADV_SCAN_IND		x	
ADV_NONCONN_IND			

Exercise preparation

Clone the Github repository:
bt_fund

The following exercises will use the folders in l2:
l2_e1
l2_e2
l2_e3

Exercise 1 – Broadcasting Beacons

1) Modify prj.conf

```
# Bluetooth LE
# STEP 1 - Include the Bluetooth LE stack in your project
CONFIG_BT=y
# STEP 2 - Set the Bluetooth LE device name
CONFIG_BT_DEVICE_NAME="Nordic_Beacon"
```

2) Add headers in main.c to include the needed bluetooth functions inside the file

```
#include <zephyr/bluetooth/bluetooth.h>
#include <zephyr/bluetooth/gap.h>
```

Exercise 1 – Broadcasting Beacons

3) Create advertising packet

```
static const struct bt_data ad[] = {
    /* Set the advertising flags:
    BT_DATA_BYTES(type,value)
    type = flag
    value = Only LE enabled, not classic Bluetooth */
    BT_DATA_BYTES(BT_DATA_FLAGS, BT_LE_AD_NO_BREDR),
    /* Set the advertising packet data:
    BT_DATA(type,pointer to value,length of value)
    type = device name
    pointer = pointer to device name from config
    length = length variable */
    BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};
```

4) Create scan response packet

```
static unsigned char url_data[] ={0x17,'/','/','a','c','a','d','e','m','y','.',
                                  'n','o','r','d','i','c','s','e','m','i','.',
                                  'c','o','m'};
/* Declare the scan response packet */
static const struct bt_data sd[] = {
        /* Include the URL data in the scan response packet:
        type = URL
        pointer = Our URL link above, URL of Nordic Developer Academy
        length = simply the size of the URL */
        BT_DATA(BT_DATA_URI, url_data,sizeof(url_data)),
};
```

4) Enable bluetooth functions in main() itself

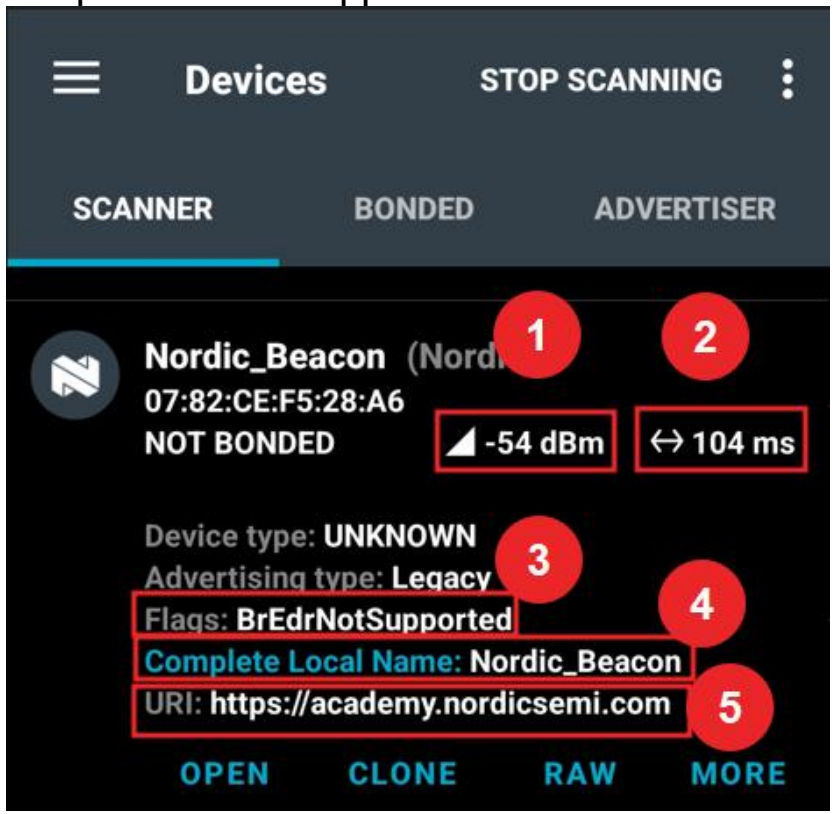
```
err = bt_enable(NULL);
/* Enables the use of bluetooth functions*/
if (err) { /* Standard error handling */
    LOG_ERR("Bluetooth init failed (err %d)\n", err);
    return -1;
}
LOG_INF("Bluetooth initialized\n");
```

5) Start advertising

```
err = bt_le_adv_start(
    BT_LE_ADV_NCONN,
    ad, ARRAY_SIZE(ad),
    sd,
    ARRAY_SIZE(sd));
if (err) {
    LOG_ERR("Advertising failed to start (err %d)\n", err);
    return -1;
}
LOG_INF("Advertising successfully started\n");
```

Exercise 1 – Broadcasting Beacons

Output from the app:



Exercise 2 – Broadcasting with custom data

This exercise builds upon the previous exercise, but continues in l2_e2

1) Create a struct for modifying the advertising interval

```
static const struct bt_le_adv_param *adv_param =
    /* BT_LE_ADV_PARAM(options, min adv interval, max adv interval, directed or not)*/
    BT_LE_ADV_PARAM(BT_LE_ADV_OPT_NONE, /* No options specified */
                    800, /* Min Advertising Interval 500ms (800*0.625ms) */
                    801, /* Max Advertising Interval 500.625ms (801*0.625ms) */
                    NULL); /* Set to NULL for undirected advertising */
```

2) Create our custom data, then include it in the to include in the advertising packet

```
typedef struct adv_mfg_data {
  uint16_t company_code; /* Company Identifier Code. */
  uint16_t number_press; /* Number of times Button 1 is pressed */
} adv_mfg_data_type;
```

```
#define COMPANY_ID_CODE 0x0059 /* Arbitrary ID */
```

```
static adv_mfg_data_type adv_mfg_data = { COMPANY_ID_CODE, 0x00 };
```

```
BT_DATA(BT_DATA_MANUFACTURER_DATA, (unsigned char *)&adv_mfg_data, sizeof(adv_mfg_data)),
```

Exercise 2 – Broadcasting with custom data

3) setup button actions

```c
static int init_button(void)
{
    int err;
    err = dk_buttons_init(button_changed);
    if (err) {
        printk("Cannot init buttons (err: %d)\n", err);
    }
    return err;
}
```

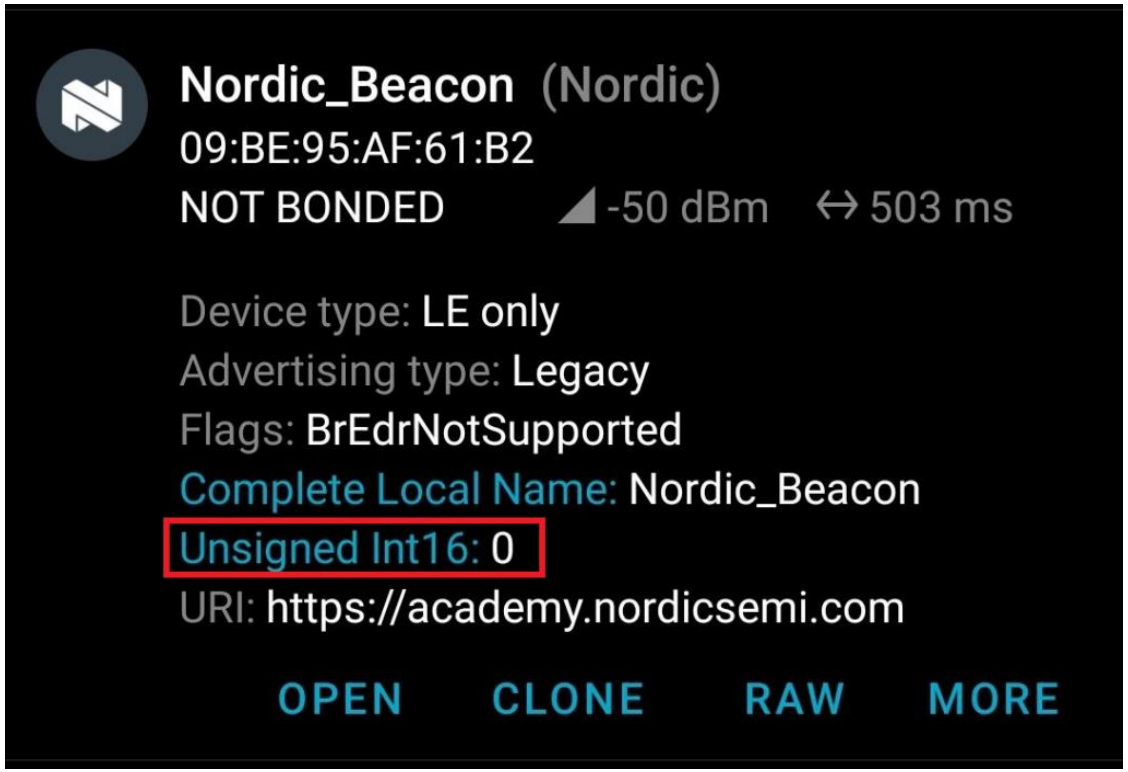
```c
static void button_changed(uint32_t button_state, uint32_t has_changed)
{
    /* If the button is pressed:
    increment our button press counter,
    update the advertising packet
    */
    if (has_changed & button_state & USER_BUTTON) {
        adv_mfg_data.number_press += 1;
        bt_le_adv_update_data(ad, ARRAY_SIZE(ad), sd, ARRAY_SIZE(sd));
    }
}
```

4) Call it in main()

```c
err = init_button();
if (err) {
  printk("Button init failed (err %d)\n", err);
  return -1;
}
```

Exercise 2 – Broadcasting with custom data

Output from the app:



Note that you need to keep scanning to see it getting changed, since it is only advertising and not connecting!!

Exercise 3 – Connecting

1) Modify prj.conf

```
CONFIG_BT_PERIPHERAL=y
CONFIG_BT_DEVICE_NAME="Nordic_Peripheral"
```

2) Add header files

```
#include <zephyr/bluetooth/addr.h>
#include <zephyr/bluetooth/uuid.h>
```

Exercise 3 – Connecting

3) Add to our scan response data

```
/* Include the 16-bytes (128-Bits) UUID of the LBS service in the scan response packet */
BT_DATA_BYTES(BT_DATA_UUID128_ALL, BT_UUID_128_ENCODE(0x00001523, 0x1212, 0xefde, 0x1523, 0x785feabcd123)),
```

4) Add to our advertising data

```
/* As before, however this time we add the BT_LE_AD_GENERAL flag;
   this makes our device act as a peripheral, rather than a beacon, allowing us to connect to it  */
BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
```

5) To be able to make a connectable advertising, we need to set a static address

```
bt_addr_le_t addr;
err = bt_addr_le_from_str("FF:EE:DD:CC:BB:AA", "random", &addr);
if (err) {
    printk("Invalid BT address (err %d)\n", err);
}
err = bt_id_create(&addr, NULL);
if (err < 0) {
    printk("Creating new ID failed (err %d)\n", err);
}
```

Exercise 3 – Connecting

5) Add a flag that enabled connectable advertising

```
/* As before, this time however we add the BT_LE_ADV_OPT_CONN to allow for connection*/
static const struct bt_le_adv_param *adv_param = BT_LE_ADV_PARAM((BT_LE_ADV_OPT_CONN|BT_LE_ADV_OPT_USE_IDENTITY),
                800, /*Min Advertising Interval 500ms (800*0.625ms) */
                801, /*Max Advertising Interval 500.625ms (801*0.625ms)*/
                NULL); /* Set to NULL for undirected advertising*/
```

6) Start advertising in main()

```
k_work_init(&adv_work, adv_work_handler);
advertising_start();
```

Exercise 3 – Connecting

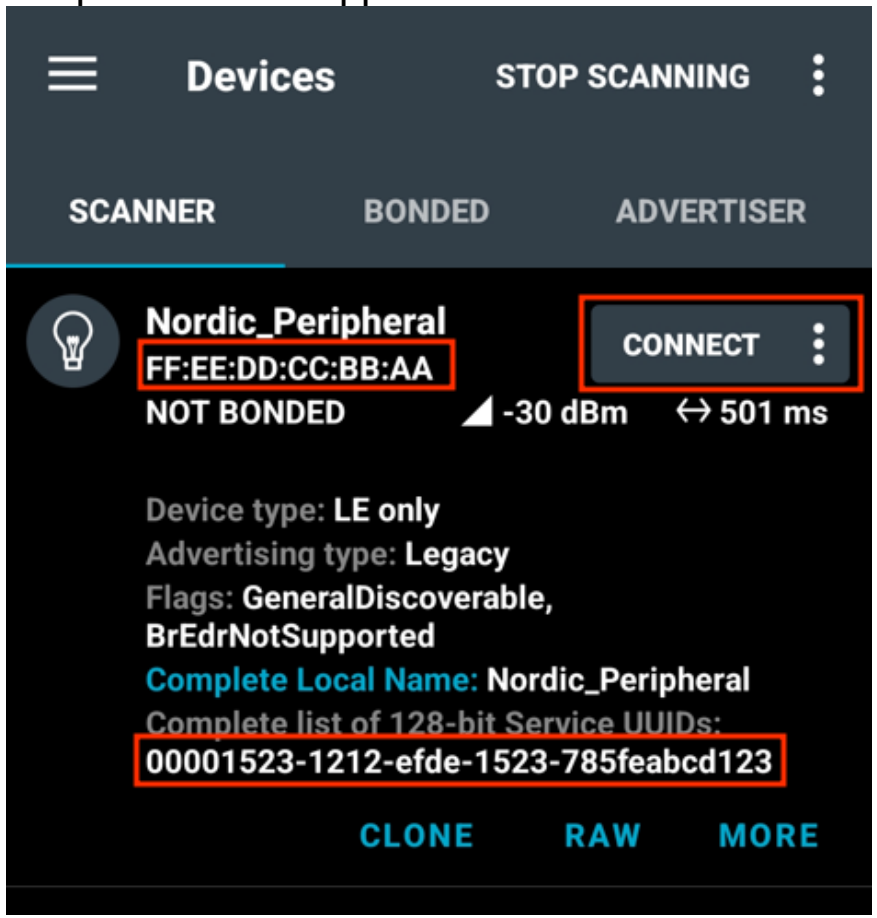
7) OPTIONAL: Add functionality to resume advertising after disconnection

```c
static void adv_work_handler(struct k_work *work)
{
    int err = bt_le_adv_start(adv_param, ad, ARRAY_SIZE(ad), sd, ARRAY_SIZE(sd));
    if (err) {
        printk("Advertising failed to start (err %d)\n", err);
        return;
    }
    printk("Advertising successfully started\n");
}
static void advertising_start(void)
{
    k_work_submit(&adv_work);
}
static void recycled_cb(void)
{
    printk("Connection object available from previous conn. Disconnect is complete!\n");
    advertising_start();
}

BT_CONN_CB_DEFINE(conn_callbacks) = {
    .recycled = recycled_cb,
};
```

Exercise 3 – Broadcasting Beacons

Output from the app:



BLE Connections

Lab 8

Overview

A connection grants you benefits like higher data throughput, mechanisms to handle packet loss, and added security

A Bluetooth LE connection exposes a set of connection parameters.
Default parameters often work for simple applications.
However, parameters can be tuned for:

- **Lower power consumption**
- **Higher throughput**
- **Application-specific behavior**

Focus of the lab:
Explanation of what each parameter does and how to adjust them on the **peripheral's perspective** and its limitations.

- Establish bi-directional communication between the Nordic device and a smartphone.
- Read current connection parameters.
- Request updated parameters from the central device.

BLE Connection Process

Two roles involved: a *peripheral* (advertising) and a *central* (scanning). Connection is only possible when a central picks up a peripheral's advertising.

- **Connection initiation:** the central sends a *connection request* to the peripheral after receiving an advertisement that it wants to connect to.
- **Acceptance:** The peripheral automatically accepts the connection request (unless using an accept-list/whitelist).
- **Switch to data channels:** After connection, devices stop using advertising channels; they start using data channels (channel hopping across data channels 0–36) for data exchange.
- **Low-power operation using connection intervals:** Devices agree on a *connection interval*
  i.e. how often they wake up to exchange data. Between intervals they sleep to save power.
- **Connection events:** At every connection interval there is a "connection event" where the central (and peripheral) exchange packets. Data is sent during these events; if no data is pending, empty packets are sent to maintain timing/synchronization. **Data throughput / packet handling:** If there is more data than can fit in one connection interval, it gets split across multiple intervals.
- **Disconnection possibilities:** The connection remains until either:
  1. One device initiates a termination (application-level disconnect), or
  2. A "supervision timeout" occurs (no packets/acknowledgments for a certain time).

BLE Connection Parameters

When a peripheral and central device connect in BLE, they exchange a set of parameters that govern how the connection behaves. Some are set by default, others are chosen by the central device at connection-time.

Connection interval : how often the two devices wake up and exchange data.
Supervision timeout : how long the connection stays alive if no packets are received; if no communication occurs before this timeout, the connection is considered lost.
Peripheral (slave) latency : number of connection intervals the peripheral is allowed to skip if it has no data to send.

Other parameters that can influence performance:

•**PHY radio mode** — the physical layer used: default is 1 Mbit/s, but can change (e.g. 2M or coded PHY) to affect bandwidth vs. range.
•**Data length & MTU (Maximum Transfer Unit)** — define how many bytes can be sent per packet (data length) and per GATT operation (MTU). Default MTU = 23 bytes, data length = 27 bytes.
Using features like “Data Length Extension” can increase data-length (on-air packet size) up to 251 bytes, thus improving throughput and reducing radio-on time (power saving + efficiency).

Exercise 1 – Connecting to your smartphone

1) Add the header file needed for handling the Bluetooth Low Energy connections.

```
#include <zephyr/bluetooth/conn.h>
```

2.1) Set up callbacks to be triggered when a connection has been established or disconnected.

structure called connection_callbacks of type bt_conn_cb.
This structure is used for tracking the state of the connection

```
struct bt_conn_cb connection_callbacks = {
    .connected              = on_connected,
    .disconnected           = on_disconnected,
    .recycled               = on_recycled,
};
```

Exercise 1 – Connecting to your smartphone

2.2) Define the callback functions on_connected and on_disconnected.

```c
void on_connected(struct bt_conn *conn, uint8_t err)
{
    if (err) {
        LOG_ERR("Connection error %d", err);
        return;
    }
    LOG_INF("Connected");
    my_conn = bt_conn_ref(conn);

    /* STEP 3.2  Turn the connection status LED on */
}

void on_disconnected(struct bt_conn *conn, uint8_t reason)
{
    LOG_INF("Disconnected. Reason %d", reason);
    bt_conn_unref(my_conn);

    /* STEP 3.3  Turn the connection status LED off */
}

void on_recycled(void)
{
    advertising_start();
}
```

! my_conn parameter is a bt_conn pointer that will be used to keep track of the connection in the next exercise.

2.3) Register the callback structure connection_callbacks.

```c
err = bt_conn_cb_register(&connection_callbacks);
if (err) {
  LOG_ERR("Connection callback register failed (err %d)", err);
}
```

This needs to be called before we start advertising, to avoid a connection being established before we have registered the callback.

Exercise 1 – Connecting to your smartphone

3. Configure an LED to indicate the current connection status.

3.1) Define the connection status LED

```
#define CONNECTION_STATUS_LED    DK_LED2
```

3.2) In the connected event, turn the connection status LED on.

```
dk_set_led(CONNECTION_STATUS_LED, 1);
```

3.3) In the disconnected event, turn the connection status LED off.

```
dk_set_led(CONNECTION_STATUS_LED, 0);
```

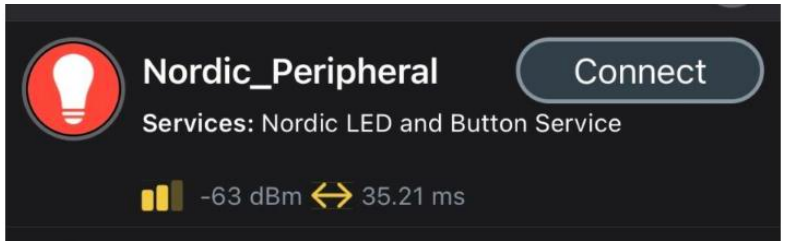
Exercise 1 – Connecting to your smartphone

4) Build and flash the application to your board

5) Open a terminal to see the log output from the application.

```
*** Booting nRF Connect SDK ***
*** Using Zephyr OS ***
[00:00:00.004,302] <inf> Lesson3_Exercise1: Starting Lesson 3 - Exercise 1
[00:00:00.007,659] <inf> Lesson3_Exercise1: Bluetooth initialized
[00:00:00.008,605] <inf> Lesson3_Exercise1: Advertising successfully started
```

6) Use your smartphone to scan and connect to your board.



LED on your board indicates a connection, and you should also be seeing the following log output :

```
[00:00:26.831,720] <inf> Lesson3_Exercise1: Connected
```

7) Disconnect the device from your smartphone

```
[00:00:38.627,004] <inf> Lesson3_Exercise1: Disconnected. Reason 19
```

Exercise 1 – Connecting to your smartphone

8) Change the application to send a message whenever button 1 is pressed.

8.1) Include the LED Button Service (LBS). in the application

```
CONFIG_BT_LBS=y
```

8.1) Include the LED Button Service header file near the top of main.c.

```
#include <bluetooth/services/lbs.h>
```

8.3) Add a callback to be notified when button 1 is pressed.

```
static void button_changed(uint32_t button_state, uint32_t has_changed)
{
  int err;
  bool user_button_changed = (has_changed & USER_BUTTON) ? true : false;
  bool user_button_pressed = (button_state & USER_BUTTON) ? true : false;
  if (user_button_changed) {
    LOG_INF("Button %s", (user_button_pressed ? "pressed" : "released"));

    err = bt_lbs_send_button_state(user_button_pressed);
    if (err) {
      LOG_ERR("Couldn't send notification. (err: %d)", err);
    }
  }
}
```

We want to send the button state of button 1 to the central device (your phone) whenever button 1 on the device is pressed.

Exercise 1 – Connecting to your smartphone

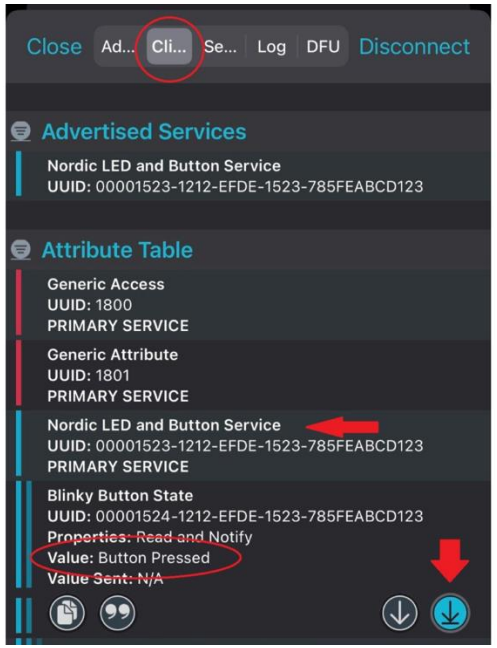
8.4) Complete the implementation of the init_button() function.

```
err = dk_buttons_init(button_changed);
if (err) {
    LOG_ERR("Cannot init buttons (err: %d)", err);
}
```

9) Build and flash the application to your board and connect to it via your phone.

- Open the **Client (Cli…)** tab to view all services.
- Find and expand the **Nordic LED and Button Service**.
- Enable notifications (icon with multiple downward arrows).
- Press **Button 1**.
- The value should change between **Button Pressed** and **Button Released**.

! If you press the button **without subscribing**, you'll get the error:
Couldn't send notification. err: -13.



Exercise 2 – Updating the connection parameters

1) Get the connection parameters for the current connection

1.1) Declare a structure to store the connection parameters in your on_connected() callback function

```
struct bt_conn_info info;
  err = bt_conn_get_info(conn, &info);
  if (err) {
    LOG_ERR("bt_conn_get_info() returned %d", err);
    return;
  }
```

1.2) Add the connection parameters to your log in the end of your on_connected() callback function

```
double connection_interval = info.le.interval*1.25; // in ms
uint16_t supervision_timeout = info.le.timeout*10; // in ms
LOG_INF("Connection parameters: interval %.2f ms, latency %d intervals, timeout %d ms", connection_interval, info.le.latency, supervision_timeout);
```

2) Build and flash the sample to your board.

```
*** Booting nRF Connect SDK ***
[00:00:00.251,098] <inf> Lesson3_Exercise2: Starting Lesson 3 - Exercise 2
[00:00:00.008,453] <inf> Lesson3_Exercise2: Bluetooth initialized
[00:00:00.009,490] <inf> Lesson3_Exercise2: Advertising successfully started
```

Exercise 2 – Updating the connection parameters

4) Modify the callbacks to be notified when the parameters for an LE connection have been updated

The bt_conn_cb structure also has the member le_param_updated. This is used to tell our application that the connection parameters for an LE connection have been updated.

4.1) Modify your connection_callbacks parameter, by adding the following line:

.le_param_updated = on_le_param_updated

4.2) Add the on_le_param_updated() event.

```c
void on_le_param_updated(struct bt_conn *conn, uint16_t interval, uint16_t latency, u
{
    double connection_interval = interval*1.25;         // in ms
    uint16_t supervision_timeout = timeout*10;          // in ms
    LOG_INF("Connection parameters updated: interval %.2f ms, latency %d intervals, t
}
```

Exercise 2 – Updating the connection parameters

4) Modify the callbacks to be notified when the parameters for an LE connection have been updated

The bt_conn_cb structure also has the member le_param_updated. This is used to tell our application that the connection parameters for an LE connection have been updated.

4.1) Modify your connection_callbacks parameter, by adding the following line:

.le_param_updated = on_le_param_updated

4.2) Add the on_le_param_updated() event.

```c
void on_le_param_updated(struct bt_conn *conn, uint16_t interval, uint16_t latency, u
{
    double connection_interval = interval*1.25;         // in ms
    uint16_t supervision_timeout = timeout*10;          // in ms
    LOG_INF("Connection parameters updated: interval %.2f ms, latency %d intervals, t
}
```

Exercise 2 – Updating the connection parameters

5) Change the preferred connection parameters

```
CONFIG_BT_PERIPHERAL_PREF_MIN_INT=800
CONFIG_BT_PERIPHERAL_PREF_MAX_INT=800
CONFIG_BT_PERIPHERAL_PREF_LATENCY=0
CONFIG_BT_PERIPHERAL_PREF_TIMEOUT=400
CONFIG_BT_GAP_AUTO_UPDATE_CONN_PARAMS=y
```

6) Build and flash your application and connect to it with your smartphone.

```
[00:06:35.863,922] <inf> Lesson3 _Exercise2: Connection parameters: interval 30.00 ms, latency 0 intervals, timeout 240 ms

[00:06:41.113,983] <inf> Lesson3_Exercise2: Connection parameters updated: interval 1005.00 ms, latency 0 intervals, timeout 4000 ms
```

Exercise 2 – Updating the connection parameters

5) Change the preferred connection parameters

```
CONFIG_BT_PERIPHERAL_PREF_MIN_INT=800
CONFIG_BT_PERIPHERAL_PREF_MAX_INT=800
CONFIG_BT_PERIPHERAL_PREF_LATENCY=0
CONFIG_BT_PERIPHERAL_PREF_TIMEOUT=400
CONFIG_BT_GAP_AUTO_UPDATE_CONN_PARAMS=y
```

6) Build and flash your application and connect to it with your smartphone.

```
[00:06:35.863,922] <inf> Lesson3 _Exercise2: Connection parameters: interval 30.00 ms, latency 0 intervals, timeout 240 ms

[00:06:41.113,983] <inf> Lesson3_Exercise2: Connection parameters updated: interval 1005.00 ms, latency 0 intervals, timeout 4000 ms
```

Networked embedded systems - Lab 9

Embedded AI: Motion Recognition

Today's Lab: Motion Recognition on Thingy:53

What we're building:

- Train a gesture classifier for 4 motions: idle, snake, wave, updown
- Deploy ML model to run *on-device* (Embedded AI)
- Use Edge Impulse platform with BLE workflow

The workflow:
Thingy:53 $\rightarrow$ BLE $\rightarrow$ Mobile App $\rightarrow$ Edge Impulse Cloud $\rightarrow$ Deploy back to device

Step 1: Installing Edge Impulse Firmware

First, flash the Edge Impulse firmware:

- Download `dfu_application.zip` from Edge Impulse
- Flash it to Thingy:53 (same process as before)
- Create free Edge Impulse developer account

Then connect via mobile:

- Install **nRF Edge Impulse** app (iOS or Android)
- Connect Thingy:53 to your phone via BLE
- Log in to the app with your Edge Impulse account

Step 2: Data Collection (via Mobile App)

Collect accelerometer data over BLE:

- Use nRF Edge Impulse app
- 10-second recordings at 62.5Hz
- ~10 recordings per gesture:
  - `idle` - sitting still
  - `snake` - sliding on desk
  - `wave` - left to right
  - `updown` - up and down
- Data auto-syncs to Edge Impulse cloud

Quality matters:

- Add variation in speed and intensity

Step 3: Creating the Impulse Pipeline

Design your impulse in the "Create impulse" tab (3 blocks):

1. **Time series data** (Input)
   - Window size: 2000ms (long enough to capture gesture pattern)
   - Window increase: 80ms (sliding window for continuous recognition)
2. **Spectral Analysis** (Processing)
   - Converts time-domain signal $\rightarrow$ frequency-domain features
3. **Classification** (Learning)
   - Neural Network (Keras)
   - Input: spectral features
   - Output: 4 gesture classes (idle, snake, wave, updown)

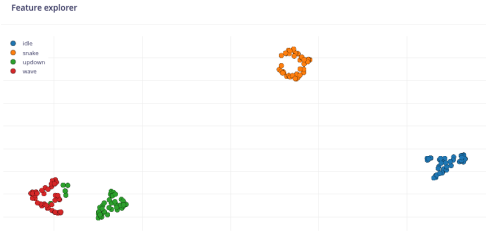
Step 4: Configuring Spectral Features

Configure the DSP block in the "Spectral features" tab:

Feature vector per window:

- FFT extracts frequency content
- Per axis (accX, accY, accZ):
  - 16 frequency bins
  - 1 RMS value (signal energy)
  - Peak frequency
- **Total:** ≈50-60 features per window

Tip: Click "Autotune parameters" to optimize DSP settings automatically!



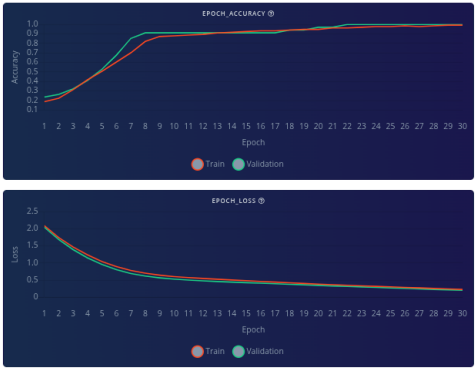
Feature explorer uses PCA to project high-dimensional features to 2D (PC1, PC2)

Step 5: Training the Neural Network

Train in the "NN Classifier" tab:

Training process:

- Start with 1 epoch (baseline)
- Gradually increase to 30 epochs
- Watch accuracy improve
- Confusion matrix shows classification errors
- Goal: >90% accuracy



Training results showing 96% accuracy after 30 epochs

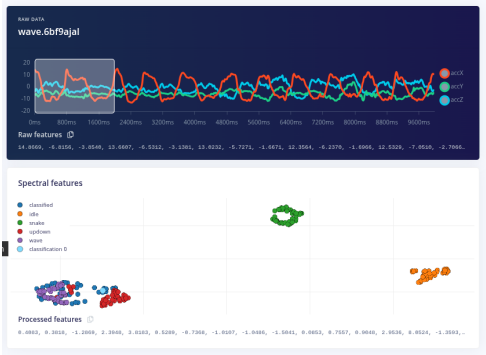
Step 6: Testing (Back to Mobile App)

Test your model with live data:

- Return to nRF Edge Impulse app
- Collect 10-second test samples
- Data syncs to cloud
- View classification results in web interface

Important:

- Currently: Thingy:53 only collects data
- Flow: Raw data → BLE → Phone → Cloud → Inference
- Next step: On-device inference



Live classification results showing gesture predictions

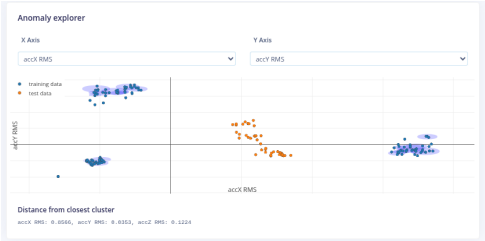
Step 7: Anomaly Detection - The Unknown Unknown

The problem:

- Neural networks always predict one of the 4 trained classes
- Unknown gestures (e.g., shake) get misclassified

Solution: Add K-means clustering learning block to the pipeline

- Creates 32 clusters around training data
- Measures distance to nearest cluster
- Large distance → flag as anomaly



Feature space: training data (blue) vs. shake anomaly (orange)

Step 8: Deploy to Thingy:53 (On-Device Inference)

Deploy the model for true Embedded AI:

In Edge Impulse Studio (web):

- Go to "Deployment" tab → Build firmware

In nRF Edge Impulse app:

- Go to "Deployment" tab
- Download model to Thingy:53 over BLE
- Firmware includes: signal processing + neural network + classifier

Benefits of on-device inference:

- Ultra-low latency (milliseconds)
- Works offline - no internet needed
- Privacy - data never leaves device
- Low power consumption

Test: Perform gestures and watch live classification results in the app

Key Takeaways: Embedded ML Workflow

1. **Feature extraction is critical**
   - Raw sensor data $\rightarrow$ FFT $\rightarrow$ spectral features $\rightarrow$ effective learning
2. **Data quality matters most**
   - Diverse, well-labeled samples beat large datasets
3. **Edge computing enables new applications**
   - Low latency, privacy, offline operation
4. **Anomaly detection for robustness**
   - Real-world deployment requires handling unknown inputs
5. **BLE bridges development and deployment**
   - Mobile app serves as interface to embedded device