Networked Embedded Systems Lab

Xenofon (Fontas) Fafoutis
xefa@dtu.dk

2025-01

Contents

Preliminaries

Welcome to the lab of 02226/02116 Networked Embedded Systems.

This lab complements the course lectures with practical exercises that aim at providing you hands-on experience with building networked embedded systems applications. The lab exercises are based on the Zephyr Embedded Operating Systems and the RF Connect SDK for embedded platforms that are based on Nordic Semiconductor Systems-on-Chip.

The lab is primarily based on two external resources[1], namely the Nordic Developer Academy Courses and the Edge Impulse Tutorials. These external resources are enhanced with additional comments, guides, tips, and questions.

Hardware

Unfortunately, purchasing and managing hardware for such a large classroom is not easy, therefore the lab is BYOH (Bring Your Own Hardware). However, the lab is carefully designed so that you are required to buy only one relatively affordable platform, namely the Nordic Thingy:53, while it takes also advantage of hardware that you already own (e.g. network connectivity is achieved using Bluetooth LE between Thingy:53 and a smartphone).

The Thingy:53 is available from all major electronics providers. A list of providers is available here, although you can also buy it from a different store. It is strongly recommended that you choose a store that is based in the EU and ships from the EU to avoid duty charges and shipping delays.

In addition to the Nordic Thingy:53, you will also need the following hardware to complete the lab: (i) a Windows or macOS laptop or desktop computer; (ii) an Android or iOS phone; and (iii) a USB-C to USB-C cable.

While Thingy:53 is relatively cheap (costing less than a typical university textbook), it is understandable that some students may not be able to afford it. If that is the case, to reduce the per-person costs, it is recommended to work in groups and share the hardware among the members of the group. If this is also not technically possible, it is still possible to do a large part of the labs without having access to a Thingy:53, including the theoretical sections and the practical sections up to the point of deployment (i.e., you can write the embedded software and compile it on your laptop). In any case, it is possible (albeit not advisable) to pass the course without completing this lab.

Assessment

These lab exercises are not mandatory, in the sense that you are not asked to deliver any solutions or report at the end. They are based on the principle of self-assessment: reference solutions are provided to you, so you can assess your own solutions against them. It is recommended that you do not blindly copy-paste the solutions, but make sure that you understand what you are doing. Furthermore, there will be two 4-hour lab sessions, whereby you can get support from the teaching assistants. Although not technically mandatory, you should expect that **there will be questions in the final exam that are based on this lab**.

Alignment to lectures

You are completely free to do these lab exercises at your own pace. Table 1 provides an alignment between each lab exercise and the lectures. According to your learning style, you may do the lab exercises before or after the corresponding lecture. If you prefer to study the theory first and then get practical experience, make sure that you do the lab after the corresponding lecture. Otherwise, if you prefer to first get your hands dirty and then dive into the theoretical aspects, make sure that you do the lab before the corresponding lecture.

[1] This lab relies heavily on external resources that may get updated at any time. Please report any broken links at `xefa@dtu.dk`.

Table 1: Alignment of Labs and Lectures

Lab Number	**Lab Title**	**Lecture Title**
1	Introduction to the nRF toolchain	Introduction to Networked Embedded Systems
2	GPIOs: LEDs and Buttons	MCU I/O
3	nRF Connect SDK Apps	Embedded Software
4	Printing and Logging	Serial Communication
5	Serial Communication	Serial Communication
6	Introduction to BLE	Wireless Embedded Systems
7	BLE Advertisements	Wireless Embedded Systems
8	BLE Connections	Wireless Embedded Systems
9	Embedded AI: Motion Recognition	Embedded Data Processing and Embedded AI

1 Introduction to the nRF toolchain

This lab is based on the Nordic Semiconductor DevAcademy course nRF Connect SDK Fundamentals and specifically in Lesson 1 – nRF Connect SDK Introduction.

The objective of the first lab is to introduce you to the nRF Connect Software Development Kit (SDK). The nRF Connect SDK is a unified set of libraries and drivers for Nordic Semiconductor's nRF series platforms. The nRF Connect SDK leverages ZephyrOS, which is an open source Embedded Operating System (OS). Developing networked embedded systems applications using an Embedded OS and an SDK offers abstraction and portability, which in turn reduce development time.

To familiarize yourself with the nRF toolchain start by reading the nRF Connect SDK structure and content.

The full documentation of the nRF Connect SDK is available here. The provided link points to documentation of the latest release. It is generally a good idea to use the latest release if you are starting a new project, but if you are continuing an older project make sure that you are looking at the version of the documentation that matches your project. Unless otherwise stated, this lab is based on the nRF Connect SDK v3.0.0 (full documentation available here).

1.1 Installing the tools

We will now install all the necessary tools required for developing embedded software with the nRF Connect SDK, by following the guide in Exercise 1 – Installing nRF Connect SDK and VS Code. When using the guide make sure that you have selected the tab `3.0.0` to match the version of the nRF Connect SDK that we use. To the extent possible, also ensure that you install the exact version of each tool suggested in the guide in order to avoid unexpected incompatibilities.

The guide recommends the installation of several tools to ensure wide compatibility with a wide range of Nordic platforms. This lab uses a specific hardware platform (i.e. Nordic Thingy: 53); therefore not all tools are needed. It may be a good idea to install them, if you want your setup to be ready for all platforms, and if you want to avoid any unexpected incompatibilities.

- **SEGGER J-Link Software** provides the software for using the SEGGER J-Link programmer and debugger chips. Several of the larger Nordic Semiconductor development kits (e.g. the nRF5340-DK) incorporate a SEGGER J-Link debugger chip which allows you to program and debug the firmware in the nRF chip via the SWD programming and debugging interface. Microcontrollers are often also programmable via other serial interfaces, e.g. UART, yet this solution cannot be used for debugging purposes. The platform that we use in this lab (i.e. Thingy:53) does not have a SEGGER J-Link debugger. Therefore, installing the SEGGER J-Link software is not technically necessary.
- **nrfutil** is a command line tool for Nordic Semiconductor platforms. It technically does not require installation as it is provided as a binary. We will not need it.
- **VS Code** is a popular editor for programming. Nordic Semiconductor have created a plugin for VS Code; it is therefore very convenient to use it for developing our embedded software. If you do not have VS Code already installed in your system, you can find it here.
- **nRF Connect Extension Pack** is a plugin for VS Code enables us to develop embedded software and deploy it on the embedded hardware using VS Code.
- The **Toolchain** is a set of tools for building embedded software using the nRF Connect SDK. This is installed using the nRF Connect Extension Pack for VS Code and you need to install version `3.0.0`.
- **nRF Connect SDK** is also installed using the nRF Connect Extension Pack for VS Code and you need to install version `3.0.0`.

In addition to the tools above, you will also need to install the nRF Connect for Desktop.

1.2 Blinky: the `hello world` of embedded software

Next, we will create the `hello world` of embedded software, i.e. embedded software that blinks the LED of our platform, by following the guide Exercise 2 – Build and flash your first nRF Connect SDK application.

In steps 1-4 of the guide, we will create a new application based on a template (called `sample`) and compile it into an executable binary. In step 4.1 ensure that you select `thingy53/nrf5340/cpuapp/ns` to match the target platform of the lab (i.e. Thingy:53).

Steps 5 and 6 explain how to deploy the binary to the target platform using the SEGGER J-Link debugger. Importantly, Thingy:53 does not have a debugger, so we will not be able to program it using the steps 5 and 6. Instead, we will use program it over the serial interface with the help of the bootloader as explained in this guide. The bootloader is a program that runs when a microcontroller boots: it first initializes the system and then typically loads the main application binary into memory for execution. In this guide, we press a specific button while powering on our platform in order to change the default behaviour of the bootloader. If the bootloader finds this specific button pressed, instead of loading the application binary from flash memory into RAM, it reads the application from the serial interface (i.e. UART over USB). This offers an alternative means of updating the application binary of microcontrollers without using the debugger interface.

Note: The application binary that we need to upload to our platform is the file `dfu_application.zip`. Different to the screenshots of the guide, in my setup this is available right under the `build` directory in `ncsfund/l1_e2/build/`.

IMPORTANT: Before uploading the software ensure that the setting `Enable MCUboot` is enabled. It should be enabled by default, but it is safer to double-check.

In Step 7 of the guide, we will make a minor change in the source code to make the LED blink at a higher speed.

2 GPIOs: LEDs and Buttons

This lab is based on the Nordic Semiconductor DevAcademy course nRF Connect SDK Fundamentals and specifically in Lesson 2 – Reading buttons and controlling LEDs.

The objective of this lab is to demonstrate how hardware is described in the nRF Connect SDK and how the General-Purpose Input/Output (GPIO) driver can be used to detect that a button has been pressed and control the LEDs.

The first part of the lab describes how hardware is described using the devicetree. This section concludes with the inspection of nRF52833 as an example. In addition to (or instead of) this example, inspect the device tree of the platform that we use in this lab: the Nordic Thingy:53. In the `Welcome` menu of nRF Connect, navigate to menu `Manage SDKs` and click on `Open SDK Directory`. Then navigate to the path `<SDK>/zephyr/boards/nordic/thingy53` and open the file `thingy53_nrf5340_common.dtsi`.

Task: Try to identify the GPIOs where the buttons and LEDs of the Thingy:53 are connected. Open the schematics of Thingy:53 to confirm your findings. The answer is available in the Solutions[1].

The next part of this lab describes the device driver model before further diving in the GPIO Generic API. This GPIO driver provides all the necessary functions for configuring and controlling the state of GPIOs. We will use this driver for the LEDs and buttons: for example, turning on a LED is done by setting the corresponding GPIO as output and changing its state to close the circuit.

The next part of this lab goes back to the `blinky` sample that we used in Lab 1.2 and explain how it uses the GPIO driver to make the LED blink. In Lab 1.2, we changed a single line in our source code to alter the frequency the LED toggles.

Task: Make the blue LED toggle instead by changing a single line in the source code. The answer is available in the Solutions[2].

2.1 Turn on the LED by pushing a Button (polling based)

Next, we will turn on the LED by pressing a button, following the guide in Exercise 1 – Controlling an LED through a button (polling based). The guide is following the polling method which is checking the state of the button periodically using a periodic timer.

Task: Observe the source code and explain why the polling based method is very inefficient. The answer is available in the Solutions[3].

2.2 Turn on the LED by pushing a Button (interrupt based)

In the last part of this lab we will turn on the LED by pressing a button, following the guide in Exercise 2 – Controlling an LED through a button (interrupt based). This solution is based on interrupts. Instead of periodically checking the state of the GPIO connected to the button, the interrupt controller is instructed to monitor the state of the GPIO and trigger an interrupt handler when a change is detected.

Task: You will notice that with this solution the LED changes state every time we push the button. Change the source code so that the LED turns on when you press the button and turns off when you release it. The answer is available in the Solutions[4].

3 nRF Connect SDK Apps

This lab is based on the Nordic Semiconductor DevAcademy course nRF Connect SDK Fundamentals and specifically in Lesson 3 – Elements of an nRF Connect SDK application.

The lab begins with a description of the various configuration files. In the paragraph about board configuration, also inspect the configuration file of the board that we use in this lab: the Nordic Thingy:53. In the `Welcome` menu of nRF Connect, navigate to menu `Manage SDKs` and click on `Open SDK Directory`. Then navigate to the path `<SDK>/zephyr/boards/nordic/thingy53` and open the file `thingy53_nrf5340_cpuapp_ns_defconfig`.

The guide continues with an overview of the Devicetree overlays, CMake, and build systems.

3.1 Creating an App

This lab guides you in creating a basic `hello world` app that prints a message periodically on the serial output. To see the serial output in your monitor, you will need a serial terminal app. A serial terminal app is provided within the nRF Connect for Desktop.

Note that the Thingy:53 will not appear in the `Connected Devices` menu in VS Code because this board does not have a programmer/debugger chip.

This app prints a message on UART. As we know from theory, the transmitter and receiver in UART need to have a pre-agreed frame configuration and baud rate to successfully communicate. This works out of the box because the Nordic serial terminal app is preconfigured to the same default settings. However, this is not always the case.

Task: Find the default UART configuration and baud rate of the Thingy:53. You should look at the device tree of Thingy:53 and in the UART serial driver of ZephyrOS for Nordic devices (since the app does not override the default settings). The answer is available in the Solutions[5].

3.2 Customizing an App

This lab guides you in customizing the basic `hello world` app.

In the first part of the lab we will learn how to include a custom source file in the build. In the second part of the lab we will learn how to modify the hardware configuration in a way that applies only to our app (and not all apps that use the same hardware).

Tip: When changing the build files ensure that you make a pristine build to make sure that all changes apply. Step 14 in this guide explains how to do a pristine build, but you should also do this in Step 10.

4 Printing and Logging

This lab is based on the Nordic Semiconductor DevAcademy course nRF Connect SDK Fundamentals and specifically in Lesson 4 – Printing messages to console and logging.

The lab begins with a description of the `printk()` function which is the equivalent of `printf()`. It then continues with a description of the Logger module which offers more advanced logging functionality including colour coding, time stamps, and the ability to enable/disable print messages at compile time or runtime.

4.1 Printing to the terminal

This lab teaches you how to print to the console using the `printk()` function.

Note that the Thingy:53 will not appear in the `Connected Devices` menu in VS Code because this board does not have a programmer/debugger chip. To see the printed output, you will need to set up a serial terminal similarly to the previous lab.

4.2 Using the logger

This lab teaches you how to use the logger module. It essentially shows how to provide a similar functionality as the previous lab using the logger module instead of simple printing.

Notice that this lab also shows you how to print a variable in hex format using `LOG_HEXDUMP_INF`. This is a particularly useful tool in debugging whenever you need to examine a variable or some memory space at binary or hex level.

4.3 Configuring the logger

This lab teaches you how to configure the logger module. Throughout the lab, you will learn how to enable a minimal logger implementation that can be very useful in severely memory constrained environments.

Task: Let's assume that while you are debugging you want to filter out all the clutter and see only the error messages. Configure the logger in the project configuration file to override the setting in the source code to print only the error messages. You should look for how to do it in the documentation of the logger. The answer is available in the Solutions[6].

5 Serial Communication

This lab is based on the Nordic Semiconductor DevAcademy course nRF Connect SDK Fundamentals and specifically in Lesson 6 – Serial communication (I2C).

The nRF Connect SDK Fundamentals course also includes a lesson on Serial Communication based on UART. We have already used UART in Lab 3 and Lab 4 for printing to the terminal. This lesson goes beyond that by demonstrating bidirectional communication over UART. Unfortunately, this UART lesson is not compatible with our hardware platform, Thingy:53, and thus we will skip it. However, you may study the theoretical chapters of this UART lesson if you are interested to learn more about the UART Protocol (also covered in the respective course lecture) or the UART Driver implemented in ZephyrOS.

Instead, in this lab we will focus on Serial Communication using I2C. The lab begins with a description of the I2C Protocol and the I2C Driver implemented in ZephyrOS. In turn, we will use the I2C driver to communicate with one of the sensors on the Thingy:53, namely the BH1749 Ambient Light Sensor.

Task: Before starting the lab assignment, let us understand the I2C bus of Thingy:53. Open the schematics of Thingy:53 and study the I2C bus. Try to figure out the following: (i) who is I2C controller? (ii) how is the I2C controller connected to the I2C bus? (iii) what is the value of the I2C pull-up resistors? (iv) who are the I2C targets and how many they are? (v) what are the I2C addresses of the targets? The answer is available in the Solutions[7].

5.1 Connecting to the BH1749 Ambient Light Sensor

Let us now proceed with Exercise 2 – Connecting to the BH1749 Ambient Light Sensor on the Thingy:91 and Thingy:53. Specifically, you will use I2C to configure the sensor, initiate the measurements, and periodically read the measurements. Note that we skip Exercise 1 as it is not compatible with our hardware.

Observe that we communicate with the sensor by reading and writing to its registers. To do so, we need to identify from the datasheet (i) the addresses of the relevant registers and (ii) what is the meaning of each value.

Tip: Observe that the MCU and the sensor are not synchronized in this implementation. In step 9, we configured the sensor to take measurements every 120 ms. Yet, in `main.c` the sleep time between readings is defined as 1000 ms. Furthermore, these periodic events are triggered by different clocks. This means that every time the MCU reads the data registers of the sensor, it will get the most recent measurements which may be up to 120 ms old. An implementation with interrupts will not have this issue.

6 Introduction to BLE

This lab is based on the Nordic Semiconductor DevAcademy course Bluetooth Low Energy Fundamentals and specifically in Lesson 1 – Bluetooth LE Introduction.

The lab starts with a General Introduction to Bluetooth Low Energy and continues with an introduction to the Device Roles and Topologies, Data Representation, and Radio Modes. BLE is not covered in the lectures of the course, so these theoretical lessons can act a general introduction to BLE.

6.1 Testing a BLE connection to your mobile phone

The lab continues with testing a Bluetooth LE connection to your mobile phone. In this exercise you will not modify the source code, instead you will compile and run an existing sample.

Tip: The guide instructs you to compile the sample directly from the nRF Connect SDK without making a copy of the sample to your working directory. This is fine because you will not make any modifications to the source code. However, it may not be trivial to find your compiled binaries. You can find them in the path: `<SDK>/nrf/samples/bluetooth/peripheral_lbs/build/dfu_application.zip`.

In this BLE example, you use your phone to control the LED of the board. Thingy:53, however, uses an RGB LED instead of three individual red, green and blue LEDs, meaning that the three LEDs are packed inside the same small package. As a result, turning on multiple LEDs appears as a mixed colour combination.

7 BLE Advertisements

This lab is based on the Nordic Semiconductor DevAcademy course Bluetooth Low Energy Fundamentals and specifically in Lesson 2 – Bluetooth LE Advertising.

The lab starts with a brief introduction of the BLE Advertising Process and continues with an introduction to the Advertising Types and Bluetooth Addresses. It then continues with an examination of the Advertising Packet Format. BLE is not covered in the lectures of the course, so these theoretical lessons can act a general introduction to BLE.

Tip: As in Lab 6, we will use our phone to capture and inspect the BLE packets broadcasted by Thingy:53. You may use either an Android or iOS phone to complete this lab. Note, however, that iOS filters out a lot of the information in the advertisement packets from the apps; therefore a lot of information will not be visible from iOS. If you are using an iOS phone, it is recommended to also switch to the Android tab in the solutions and have a look at the Android screenshots.

7.1 Broadcasting Beacons

In the first exercise, we will create an application that periodically broadcasts non-connectable advertisements that contain a URL.

7.2 Broadcasting Beacons with Custom Data

In the second exercise, we will change some advertisement parameters and enhance our advertisements with custom data. The custom data will be a counter that tracks the number of times the button is pressed.

7.3 Connectable Advertising

In the third exercise, we will create an application that periodically broadcasts connectable advertisements. We will also learn how to change the BLE Address.

8 BLE Connections

This lab is based on the Nordic Semiconductor DevAcademy course Bluetooth Low Energy Fundamentals and specifically in Lesson 3 – Bluetooth LE Connections.

The lab starts with a description of the BLE Connection Process and continues with an introduction to some Connection Parameters. BLE is not covered in the lectures of the course, so these theoretical lessons can act a general introduction to BLE.

8.1 Connecting to your phone

In the first exercise, we will create an application that connects to the phone. We will, in turn, use your phone to read the status of a button on Thingy:53.

8.2 Updating the connection parameters

In the second exercise, we will update the default BLE connection parrameters.

Tip: Make sure that you make a **pristine** build to ensure that the changes are incorporated in the new binaries.

9 Embedded AI: Motion Recognition

This lab is based on the Edge Impulse tutorial on Motion Recognition and on the Edge Impulse tutorial for Thingy:53.

Edge Impulse provides a framework for Embedded AI and Edge AI having hardware, like Thingy:53, in the loop. Indeed, Thingy:53 comes out of the box with the Edge Impulse firmware preinstalled. However, we have overwritten this firmware during the previous labs. So, the first step would be to re-install it.

To complete this lab you will need to create a free Edge Impulse developer account.

9.1 Installing the Edge Impulse Firmware on Thingy:53

Download (and unzip) the Edge Impulse firmware (`dfu_application.zip`) from the Edge Impulse using this link. If the link is outdated, check out the 'Updating the firmware' section of the Edge Impulse tutorial for Thingy:53.

Upload the firmware on Thingy:53 similarly to the previous labs.

After the firmware is successfully installed, you may remove the cable and disconnect it from your laptop. This will make it easier later to collect data without having the cable in the way.

We will then connect Thingy:53 to your mobile phone via BLE, following this guide. Install to your mobile phone the `nRF Edge Impulse` app for iPhone or Android. We will use this app to deploy AI models on Thingy:53 via BLE. Log in to the app and connect Thingy:53 to your mobile phone.

Tip: In iOS, when your phone gets in sleep mode, the BLE connection between Thingy:53 and iOS is disconnected. Ensure that the BLE connection is re-established before trying to communicate with your Thingy:53.

9.2 Data Collection

We are now ready to start the tutorial on building a continuous motion recognition system based on the accelerometer data of Thingy:53.

We will first use the nRF Edge Impulse app on your phone to collect accelerometer data using Thingy:53. Read the guide carefully and follow it as accurately as possible. Your Machine Learning model will be as good as the data you use to train it, so ensure that you collect sufficient data (i.e. 12 recordings of 10 seconds for each of the 4 classes) and ensure that there is some variation in the way you perform the movements. Data collection is tedious and time-consuming; so, Edge Impulse provides a pre-collected dataset for those that want to skip this step. However, for learning purposes, skipping this step is not recommended.

You will notice that the data that you collect using the phone app over BLE automatically synchronize to your Edge Impulse account, and you can see and edit them via the Web version of the Edge Impulse studio. You can also visualize the data by clicking on each recording. Inspect data from the 4 classes and try to observe if there are any patterns visible to the human eye.

9.3 Pre-Processing and Training

We will now use the data that we collected to create a motion classifier, following the section of the tutorial titled 'Designing an impulse'. We will start by splitting the data into windows and performing some spectral analysis to extract features. In turn, will provide these features as input data to a neural network and train it. After the network is trained, we'll be able to see how the network performs on the training data.

9.4 Testing

In the next step we will test the model, by classifying at real time data collected from Thingy:53. This step is based on the tutorial section titled 'Classifying new data'. However, the tutorial guides you through performing the live classification via the web app. Yet, in the previous steps, we connected Thingy:53 via BLE to our mobile phones; therefore it is easier to collect testing data via the nRF Edge Impulse app.

To do so, first ensure that the device is connected via BLE to the phone and then go to the 'Data Acquisition' tab. Select the 'Testing' data tab and collect some testing data sample similarly to Step 9.2 (i.e. using the accelerometer, 10-second window, 62.5 Hz sampling frequency). Once the data sample is collected using the phone app, the data will synchronize to the web app, whereby you can see the result of the classification.

Note that in this step, we are only using Thingy:53 to collect accelerometer data. The row data is transmitted to the phone via BLE, and the neural network is executed off-device. We do not yet use Thingy:53 for performing the inference on-device (i.e. Embedded AI).

9.5 Anomaly Detection

In the next step, we'll set up a K-means based anomaly detector to identify unknown classes. This step is based on the tutorial section titled 'Anomaly detection'.

Similarly to the previous step, it is easier to use the nRF Edge Impulse app to communicate with Thingy:53 via BLE. To do so, first ensure that the device is connected via BLE to the phone and then go to the 'Data Acquisition' tab. Select the 'Testing' data tab and collect an anomalous data sample similarly to Step 9.2 (i.e. using the accelerometer, 10-second window, 62.5 Hz sampling frequency). Following the tutorial this shall be a shaking motion labelled 'Shake'. Once the data sample is collected using the phone app, the data will synchronize to the web app, whereby you can see the result of the anomaly detection using K-means.

9.6 Model Deployment

In the next step, we will deploy the model on Thingy:53 and perform on-device inferencing (i.e. Embedded AI). The motion recognition tutorial instucts how to do this using the edge impulse command line tools. Instead, it is easier to do it via the nRF Edge Impulse app, following the guide on the Edge Impulse tutorial for Thingy:53: specifically the section titled 'deployment' and 'inferencing'.

Finally, have a look at the tutorial's final remarks.

Further Projects

You have reached the end of the Networked Embedded Systems lab. Do you want to do more projects with your Thingy:53 to learn more?

Here's a list of ideas that (probably) do not require additional hardware:

- Lesson 6 and 7 of nRF Connect Fundamnetals for building multithreaded applications in Zephyr OS.
- Lesson 4 and 5 of Bluetooth Low Energy Fundamentals for diving a little deeper into BLE.
- The Audio tutorials of Edge Impulse.
- Edge Impulse Community projects built for Thingy:53.
- Nordic Samples and Applications for Thingy:53
- Repeat Exercise 2 of Lesson 6 of nRF Connect Fundamentals but this time try to read values from some different sensor of Thingy:53.

These projects are unguided, unsupported, untested, and not part of the course.

Solutions

[1]The red, green and blue LEDs are connected to pins P1.08, P1.06 and P1.07 respectively. The two buttons are connected to pins P1.14 and P1.13.

[2]In the device tree we can see that `led2` is alias for the blue LED. Therefore, we need to go to line 15 and change it to `#define LED0_NODE DT_ALIAS(led2)`. In the real world it would have been good practice to also change the name of the variable.

[3]The device wakes up once every 100 ms to read the state of the button and set the state of the LED. This happens also when nobody presses the button consuming energy unnecessarily. Furthermore, there is a max 100 ms delay between the moment the user presses the button and the LED turns on, which is not perceivable in this application but can be an issue in delay-sensitive applications.

[4]We need to do two modifications. In line 44, where we configure the initial state of the LED, we should make sure that the LED is initialized as inactive using the configuration `GPIO_OUTPUT_INACTIVE`. In line 54, where we configure the interrupts, we should make sure that an interrupt is generated on both edges (both rising and falling edge) using the configuration `GPIO_INT_EDGE_BOTH`. The functionality now looks identical to Lab 2.1, but importantly with this interrupt based implementation the devices wakes up only when the button is pressed or released (not periodically).

[5]The default baud rate is defined in the device tree file of the board. In the `Welcome` menu of nRF Connect, navigate to menu `Manage SDKs` and click on `Open SDK Directory`. Then navigate to the path `<SDK>/zephyr/boards/nordic/thingy53` and open the file `thingy53_nrf5340_common.dtsi`. The default baud rate is defined in `uart0` in line 265 and is 115200. The UART driver for Nordic devices can be found in `<SDK>/zephyr/drivers/serial/uart_nrfx_uart.c`. The default configuration is 8N1 defined in structure `uart_nrfx_uart0_data` in line 1124. These default settings can be overridden at project level.

[6]In `prj.conf` you need to set `CONFIG_LOG_OVERRIDE_LEVEL=1`. This overrides the log level to printing just the error messages.

[7]In Sheet 1 of the schematics, we can see an I2C label. This label represents a set of wires related to I2C. We can observe that this I2C set of wires is connected to Sheet 2, Sheet 4 and Sheet 5, so these are the sheets that we need to further examine. In Sheet 2 (nRF5340 GPIO pin mapping, Sensors and LEDs), we can observe that the I2C label is expanded into two wires, namely SDA and SCL, these are in turn connected to labels P1.02 and P1.03. In Sheet 2 (nRF5340 MCU, Port 1 GPIO's), we can observe that these labels are connected to the respective GPIOs P1.02 and P1.03 of our MCU. So, (i) the I2C controller is the MCU (nRF5340) and (ii) it is connected to GPIOs P1.02 and P1.03. We can confirm this by looking at the devicetree files of our board (see `thingy53_nrf5340_common-pinctrl.dtsi`). In Sheet 4, we can see that the SDA and SCL wires are connected to two resistors R22 and R23. These are the pull-up resistors, which (iii) have a value of 4.7 KΩ. In Sheet 5, we can trace the I2C wires, SDA and SCL, to (iv) four I2C targets: the sensors BMM150, BME688, BH1749 and VM3011. Normally, we would have to find the datasheets of these sensors to find their I2C addresses, but the schematic of Thingy:53 makes our life easy, by stating the I2C addresses (v): 0x10 for BMM150, 0x76 for BME688, 0x38 for BH1749 and 0x60 for VM3011. Observe that the addresses are also specified in the devicetree file and, indeed, the I2C driver finds the I2C address of each sensor in the devicetree file, so we do not have to explicitly specify this at application level.