

# Zephyr: Devicetree, GPIO, ISR and Callbacks Lab

BME554L - Spring 2026 - Palmeri

Dr. Mark Palmeri, M.D., Ph.D.

2026-02-01

## Table of contents

Overview . . . . .	1
Nordic DevAcademy Modules . . . . .	1
Git Version Control . . . . .	2
Repository Setup . . . . .	2
Kernel Configuration . . . . .	3
Devicetree . . . . .	3
Firmware . . . . .	4
State Diagram . . . . .	6
How do I ask for help? . . . . .	6
What to Submit & Grading . . . . .	7

## Overview

This week we will start programming the firmware that will be controlling your **nRF52833DK**. We will be using buttons to trigger device events and LEDs will be used to visualize output activity for the device. While LEDs might seem underwhelming, they are a universal way to represent how digital output signals could be used to control other devices (e.g., motors, actuators, etc.).

## Nordic DevAcademy Modules

Please complete the following Nordic DevAcademy modules:

1. [Reading Buttons and Controlling LEDs](#)

## 2. Debugging & Troubleshooting

### Tip

Remember that you will need to show completion of the entire Fundamentals course by the end of the semester, so be sure to be logged into your Nordic DevAcademy account to track your progress.

## Git Version Control

- Use best practices for version control (branching, commit messages, etc.).
- Do all development on a dedicated branch that is merged into **main** once it is functional.
- Commits should be very specific to the changes/additions you are making to your code. This will help you and others understand what you did and why you did it.
- On a given development branch, try to implement one small piece of functionality at a time, commit it, and then move on to the next piece of functionality.

### Important

You do not want one, monolithic git commit right before you submit your project.

## Repository Setup

1. Fork the [Duke-BME554-ECG-TEMP-BLE-Lab](#) template repository into your personal GitLab user space.
2. Add Dr. Palmeri (mlp6) as a **Maintainer** of your project ASAP. This will allow him to add your TAs, a faster CI runner and some project labels to your repository.
3. Clone your forked project to your local laptop.
4. Update the pipeline badge URLs in **README.md** to point to your forked repository instead of the upstream parent repository. (Do this as a discrete commit.)
5. Make sure that you can build the Zephyr application “as is” after your clone the repository to your local machine.
  - You should be using the v3.2.1 SDK and toolchain.
  - The **CMakePresets.json** file in the repository should properly link to all of the appropriate project configuration and board files for you.

### ! Important

Make sure that the SDK and toolchain are the same exact versions!!

## Kernel Configuration

Take note of kernel configurations enabled in `application/prj.conf`.

## Devicetree

- We have introduced the concept of the Devicetree (DT) for the nRF52833DK. The development kit has a preconfigured DT (`nrf52833dk_nrf52833.dts`), which is selected when we create a board configuration (the present selected in `CMakePresents.json`).
- To make changes or additions to this DT, we will create a DT overlay file that will be compiled into the firmware. The default name of this file is `nrf52833dk_nrf52833.overlay` and it is located in the `application/boards/` directory.
- The nRF52833DK has 4 integrated LEDs. Edit the DT overlay file for your project to create alias for 4 LEDs on the following pins (all 4 of which are the integrated LEDs), which will then be associated with the specified firmware GPIO pin struct names:

Firmware gpio_dt_spec Struct Name		DT Default Node Name	Physical GPIO Pin	DK Part Label
heartbeat_led	heartbeat	led0	P0.13	LED1
iv_pump_led	ivpump	led1	P0.14	LED2
buzzer_led	buzzer	led2	P0.15	LED3
error_led	error	led3	P0.16	LED4

- The nRF52833DK also has 4 integrated buttons. Edit the DT overlay file for your project to create aliases for these 4 buttons that are the same as their default DTS node names:

Firmware gpio_dt_spec Struct Name		DT Default Node Name	Physical GPIO Pin	DK Part Label
sleep_button	sleepbutton	button0	P0.11	Button 1
frequp_button	frequpbutton	button1	P0.12	Button 2

Firmware gpio_dt_spec Struct Name	DT Alias	DT Default Node Name	Physical GPIO Pin	DK Part Label
freq_down_button	freqdownbutton	button2	P0.24	Button 3
reset_button	resetbutton	button3	P0.25	Button 4

[https://docs.nordicsemi.com/bundle/ug\\_nrf52833\\_dk/page/UG/dk/hw\\_buttons\\_leds.html](https://docs.nordicsemi.com/bundle/ug_nrf52833_dk/page/UG/dk/hw_buttons_leds.html)

## Firmware

Implement the following firmware functionality as a **state machine** (switch/case):

- Check that the GPIO0 interface is ready.
- Initialize all LED output pins as GPIO\_OUTPUT\_ACTIVE or GPIO\_OUTPUT\_INACTIVE, as dictated by the functional specifications below. Note that ACTIVE for the LEDs on this DK corresponds to driving them LOW.
- Be sure to capture all function exit codes and have conditional statements to capture any returned error codes.

Implement the following control logic:

- The `heartbeat` LED blinks at a fixed 1 Hz while `main()` is being executed. Issue a `LOG_INF()` statement each time the `heartbeat` LED is toggled.
- The 2 “action” LEDs (`buzzer_led` and `iv_pump_led`) blink out of phase with one another at 2 Hz by default.
- This 2 Hz default blink frequency should be defined using a preprocessor macro: `#define LED_BLINK_FREQ_HZ 2`.
- Issue a `LOG_INF()` statement each time the “action” LEDs are toggled.
- `freq_up_button` increases the blink frequency of the “action” LEDs by 1 Hz each time it is pressed.
- This incremental increase in blink frequency should be defined by the preprocessor macro: `#define FREQ_UP_INC_HZ 1`.
- Issue a descriptive `LOG_INF()` statement each time this button is pressed, which indicates the new blink frequency.
- `freq_down_button` decreases the blink frequency of the “action” LEDs by 1 Hz each time it is pressed.

- This incremental increase in blink frequency should be defined by the preprocessor macro: `#define FREQ_DOWN_INC_HZ 1`.
- Issue a descriptive `LOG_INF()` statement each time this button is pressed, which indicates the new blink frequency.
- If the blink frequency for the “action” LEDs is  $< 1$  Hz or  $> 5$  Hz, then:
  - The “action” LEDs should both be off, and
  - The `error` LED is continuously illuminated.
  - Issue a descriptive `LOG_ERR()` statement.
  - The `heartbeat` LED should continue to blink at 1 Hz.
  - `freq_up_button`, `freq_down_button`, and `sleep_button` should be disabled (interrupts disabled).
  - The only way to exit the “error state” is to press the `reset` button.
- Define each of these min/max blinking frequency limits using preprocessor macros (you can choose appropriate names).
- Pressing the `reset` button resets the “action” LEDs to blink at their default 2 Hz rate, out of phase with one another, and re-enables the `freq_up_button`, `freq_down_button`, and `sleep_button`.
- The `reset_button` can be pressed from any state and resets the device back to the default state.
- A descriptive `LOG_INF()` statement should be issued each time the `reset` button is pressed.
- At any point in time when the “action” LEDs are blinking (i.e., not the error state), if the `sleep_button` is pressed:
  - The current blink frequency for the “action” LEDs is stored,
  - Both “action” LEDs are turned off,
  - The `heartbeat` LED continues to blink,
  - A descriptive `LOG_INF()` statement is issued,
  - The “sleep state” can be exited by:
    - Pressing the `sleep` button again, at which time the device returns to blinking the “action” LEDs at the same frequency and relative phase before being put to sleep. Relative phase means the appropriate relative timing between the different “action” LEDs.
    - Pressing the `reset_button`, at which time the device returns to blinking the “action” LEDs at their default 2 Hz rate, out of phase with one another.

### Note

- You may want to make your state diagram before writing any code.
- Do not use `k_msleep()` in your code to control LED blink timing.
  - Sleep is **blocking** and paralyzes a single-threaded application.
    - \* Instead, have your main while loop run as fast as possible and use the `k_uptime_get()` function to determine when to toggle the LEDs.
    - \* Spoiler alert: We will be system kernel timers / threads to control the LED blink timing in the next lab.
- Consider using a **struct** for each LED to bookkeep the LED state, phase, blink frequency, etc.

### Warning

Be careful with how you check to see what button(s) have been pressed in your **ACTION** state. If multiple buttons have been pressed before you test for the Boolean values that represent whether a button has been pressed, then you might have some “unexpected” behavior. For example, if you check for `freq_up_button` being pressed before checking for `reset_button` being pressed, then if both buttons are pressed at the same time, you might end up increasing the frequency and then immediately resetting it back to default, which is not what you want. You will need to think carefully about the order of your conditional statements in your state machine.

## State Diagram

1. Create a state diagram of your firmware using [PlantUML](#), or your diagraming program of choice.
2. Include a PNG of your block diagram in your git repository called `state_diagram.png`.

## How do I ask for help?

1. If you have a general / non-coding question, you should ask your TAs / Dr. Palmeri on Ed to allow any of them to respond in a timely manner.
2. Push your code to your GitLab repository, ideally with your active development on a non-main branch.
3. Create an [Issue](#) in your repository.

- Add as much detail as possible as to your problem, and add links to specific lines / section of code when possible.
  - Assign the label “Bug” or “Question”, as appropriate.
  - Be sure to specify what branch you are working on.
  - Assign the Issue to one of the TAs.
  - If your TA cannot solve your Issue, they can escalate the Issue to Dr. Palmeri.
4. You will get a response to your Issue, and maybe a new branch of code will be pushed to help you with some example syntax that you can use `git diff` to visualize.

## What to Submit & Grading

- Make sure you are committing as you develop your code, and have all of those commits pushed to GitLab (i.e., do not “squash” them).
- Make sure all of the CI pipelines are passing.
- Merge all development branches into `main` and make sure that the `main` branch is up to date with all of your commits.
- Create an annotated tag called `v1.0.0` at the commit that represents your completed lab on the `main` branch.
- Push your final repository and the annotated tag to GitLab.
- Create an Issue assigned to Dr. Palmeri titled `v1.0.0 ready for feedback` and assign it the Label `Review`.

### Note

You do not need to upload anything to Canvas/Gradescope. You will be receiving feedback through your GitLab repository.

- Grading feedback will be given on:
  - Git Usage
  - Code Functionality
  - Efficiency of code logic
  - State diagram completeness and matching the firmware implementation
  - “Readability”
    - \* “Readability” does not mean a lot of verbose comments

- \* “Readability” means that the structure of the code, the naming of variables, etc. convey meaning and logical flow.