EN 173 Lab Guides

Lab 5

Version 2026, 2025-03-04

A new way to blink an LED

A pulse-width modulated signal is one way to make an LED blink. The dedicated pulse-width hardware will take over the task, letting the microcontroller do other things once the initial configuration is done.

1. Create a new application.
2. Edit `prj.conf` to enable both logging and pulse-width modulation:

```
CONFIG_LOG=y
CONFIG_PWM=y
```

3. Generate a build configuration and create an overlay.
4. Edit the overlay file. This requires more work than we have seen previously in the overlay so the additions you need to make will be described in several steps. The first step is to create the node identifier related to the external LED.

```
/{
  pwmleds { ①
    compatible = "pwm-leds"; ②
    pwm_ext_led: pwm_ext_led { ③
      pwms = <&pwm0 0 PWM_MSEC(1) PWM_POLARITY_NORMAL>; ④
    };
  };
};
```

① First, we create an new entry in a section that already exists called pwmleds.

② A hardware category known as `pwm-leds` already exists so we can use that.

③ The PWM-controlled LED is given the node identifier (and node label) `pwm_ext_led`.

④ There are four separate PWM controllers on the nRF52840 and each controller supports up to four channels. This adds a channel to the first PWM controller (`pwm0`) for an external LED. The pin correspond to this channel will be set later. A default period of 1 ms is set and the channel has normal polarity (the pulse width specifies the time at a high voltage).

5. Continuing to edit the overlay, create custom settings for the PWM controller so it knows what to do during normal power mode and during power-saving sleep mode.

```
&pwm0 {
  status = "okay"; ①
  pinctrl-0 = <&pwm0_my_default>;
  pinctrl-1 = <&pwm0_my_sleep>;
  pinctrl-names = "default", "sleep"; ②
};

&pinctrl {
```

```
  pwm0_my_default: pwm0_my_default { ③
    group1 {
      psels = <NRF_PSEL(PWM_OUT0, 0, 3)>; ④
    };
  };

  pwm0_my_sleep: pwm0_my_sleep {
    group1 {
      psels = <NRF_PSEL(PWM_OUT0, 0, 3)>;
      low-power-enable; ⑤
    };
  };
};
```

① A status of `okay` turns on a peripheral. The off status is called `disabled`.

② The standard names for the two operational modes are `default` and `sleep`. We stick with those for compatibility with other systems.

③ The name of our custom default setting is `pwm0_my_default`.

④ A link is made between channel 0 (specified by `PWM_OUT0`) and P0.03. If you wanted to define additional channels (up to four are allowed per PWM controller), they would follow this.

⑤ Modifies the behavior for low power mode.

6. Enter Program 1 into `main.c`.

Program 1. Blink an external LED using pulse-width modulation.

```
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
#include <zephyr/drivers/pwm.h>

LOG_MODULE_REGISTER(Lab05_Exercise1, LOG_LEVEL_INF);

#define PWM_LED_NI DT_NODELABEL(pwm_ext_led)
const struct pwm_dt_spec pwm_led = PWM_DT_SPEC_GET(PWM_LED_NI); ①

int main(void) {
    int err;

    if (!pwm_is_ready_dt(&pwm_led)) { ②
        LOG_ERR("Error: PWM device is not ready");
        return -1;
    }
    pwm_set_dt(&pwm_led, PWM_MSEC(250), PWM_MSEC(125)); ③
}
```

① The process of getting the information for a PWM output is similar to that for a GPIO.

② PWM also has its own ready-reporting function.

③ The period and the pulse width are set. The convenience macro `PWM_MSEC` converts a time in milliseconds to one in nanoseconds (the unit of time that PWM functions use).

7. Connect GND on the development board to the ground bus on a breadboard. Connect the short leg of a red LED to the ground bus and the long leg to a socket in a terminal strip (the main section of the breadboard). Connect a 330 Ω resistor in series with the LED. The other end of the resistor should be connected to P0.03 on the development board.
8. Build your application and flash it to the development board.
9. You should observe a rapidly flashing LED. Unfortunately, the PWM controllers on the nRF52840 have a maximum period of about 260 ms so this method cannot be used to create an LED that has a slower blink rate.

Observing PWM on the oscilloscope

Exercise 5.1

Your goal is to observe the output of your pulse-width-modulated output on the oscilloscope and compare to what is expected from the code.

1. Begin by connecting **1+** to junction between P0.03 and the resistor on the breadboard, **1-** to ground, and ↓ also to ground.
2. Set the Time base to 50 ms/div and the Trigger level to 2 V. Choose appropriate Channel 1 settings so the PWM trace fills most of the display.
3. Open the Measurements tab and add the horizontal measurements **Period**, **PosDuty**, and **PosWidth**.
4. Acquire a single acquisition and compare the measured period, positive pulse width, and duty cycle to that expected from the code.
5. Change the code to produce a positive pulse width of 25 ms (a duty cycle of 10% when the period is 250 ms) and verify with the oscilloscope.
6. Change the code to produce a positive pulse width of 225 ms (a duty cycle of 90% when the period is 250 ms) and verify with the oscilloscope.

Averaging produces output with more than two states

Exercise 5.2

If the period is decreased below the ability of your eye to see individual flashes you instead perceive the **average** brightness. In other words, a low duty cycle will look dim and a high duty cycle will look bright. You will use the same program to vary the brightness of an external LED. In this case your eye is doing the averaging.

1. Modify Program 1 by:

a. adding the following `define` statements before the `main` function:

```
#define PWM_PERIOD_MS   1
#define PWM_PERIOD_NS   PWM_PERIOD_MS*1000000
```

b. replace the code after the `if` statement checking that the PWM controller is ready with the following:

```
    err = pwm_set_dt(&pwm_led, PWM_PERIOD_NS, PWM_PERIOD_NS);
    if (err) {
        LOG_ERR("Error %d in pwm_set_dt()", err);
        return -1;
    }
    LOG_INF("PWM period is %d ms",PWM_PERIOD_MS);
    k_msleep(1000);

    while (true) {
        for (int i = 0; i <= 10; ++i) {
            err = pwm_set_dt(&pwm_led, PWM_PERIOD_NS, i*PWM_PERIOD_NS/10);
            if (err) {
                LOG_ERR("Error %d in pwm_set_pulse_dt()", err);
                return -1;
            } else {
                LOG_INF("Duty cycle = %d%%",i*10);
            }
            k_msleep(2000);
        }
    }
}
```

2. Build and flash the application. If you are a typical human being you should not be able to detect that the LED is actually turning off and on very rapidly.
3. Open a terminal connection (from Connected Devices) and observe the logger output.
4. Repeat with periods of 10 ms, 20 ms, 50 ms, 100 ms, and 200 ms. When do you first notice the flicker?

When you have finished your observations, discuss the results with the instructor.

Exercise 5.3

You will use the same program to produce a voltage between 0 and and about 3 V using a resistor and a capacitor. The results will be observed with the oscilloscope.

1. Remove the LED and 330 Ω resistor from the breadboard.
2. Form an *RC* low-pass filter by connecting the short leg of a 10 µF capacitor to the ground

bus. This is a polarized capacitor and will be damaged if you use it backwards. Connect the long leg of the capacitor to a socket in a terminals strip.

3. Connect a 10 kΩ resistor (brown-black-orange) to the same terminal strip as the capacitor. The other leg of the resistor should be connected to P0.03 (via a jumper wire).
4. Set the period in the program to 10 ms.
5. Use the oscilloscope to observe the output of the low-pass filter (**1**+ to the junction between the resistor and capacitor, **1**- to ground, and ⏚ to ground). Use a time base of 2 s/div, a time position of 10 s, a channel 1 offset of -2 V, and a range of 500 mV/div.
6. Now observe what happens when you change the resistance. Replace the 10 kΩ resistor with a 1 kΩ resistor (brown-black-red). What differences do you observe on the oscilloscope?
7. Replace the resistor with a 330 Ω one. What differences do you observe on the oscilloscope?

When you have finished your observations, discuss the results with the instructor.

Controlling a servo

The position of a servo is controlled using pulse-width modulation. It expects a period of 20 ms and then the positive pulse width determines the position. For the Hitec HS-422 servo a pulse width of 1500 µs sends it 0°. Changing that pulse width by 10 µs changes the angle by 1°. This means -90° is produced with a pulse width of 600 µs and +90° with 2400 µs. This servo should not be driven outside of those ranges.

1. Create a new application.
2. Create a new folder named `dts` at the top-level of your application (not inside any folder other than the one holding application itself). Inside of the `dts` folder create another folder `bindings`. A *binding* is the name used in Zephyr for a file that provides a high-level description of a type of hardware. Zephyr looks for these in this particular folder.
3. Create a file named `pwm-servo.yaml` inside of the `bindings` folder. Add the following to that file:

```
description: PWM-driven servo
compatible: "pwm-servo" ①
include: base.yaml ②
properties:
  pwms: ③
    required: true
    type: phandle-array
    description: PWM specifier driving the servo
  min-pulse: ④
    required: true
    type: int
    description: Minimum pulse width (nanoseconds)
  max-pulse: ⑤
```

```
    required: true
    type: int
    description: Maximum pulse width (nanoseconds)
  deg-to-pw: ⑥
    required: true
    type: int
    description: Conversion factor from degrees to pulse width (nanoseconds)
  center-pw: ⑦
    required: true
    type: int
    description: Pulse width for center position (nanoseconds)
```

① This is the name we will use in the devicetree overlay to indicate that this binding should be used.

② Bindings can be layered on top of others. In this case this one is built on the `base` binding which provides properties expected for all bindings.

③ The servo requires a PWM specifier, the same as the `pwm-leds` used earlier.

④ A new property to hold the minimum pulse width is added. Because servos may be damaged if driven outside of their operating range it is a required property.

⑤ The maximum pulse width is also required.

⑥ The conversion factor from degrees to pulse width should be specified in the servo description.

⑦ The pulse width for the center position is the final property needed to describe operation of the servo.

4. Edit `prj.conf` to enable both pulse-width modulation and logging. We will also set the logging level for the PWM module so that only errors will be displayed.

```
CONFIG_PWM=y
CONFIG_PWM_LOG_LEVEL_ERR=y
CONFIG_LOG=y
```

5. Generate a build configuration and create an overlay.
6. Edit the overlay file. We are going to add the servo using the binding just created.

```
/{
  servo: hs422_servo {
    compatible = "pwm-servo"; ①
    pwms = <&pwm0 0 PWM_MSEC(20) PWM_POLARITY_NORMAL>; ②
    min-pulse = <PWM_USEC(600)>; ③
    max-pulse = <PWM_USEC(2400)>;
    center-pw = <PWM_USEC(1500)>;
    deg-to-pw = <PWM_USEC(10)>;
  };
};
```

① The `compatible` property indicates the binding that should be used.

② The default period for a servo is set.

③ A property holding the minimum allowed pulse width is created and set to 600 µs using one of the convenience macros (which actually converts this to a value in nanoseconds).

7. Continuing to edit the overlay, create custom settings for the PWM controller so it knows what to do during normal power mode and during power-saving sleep mode. These also define which pin is controlled by the PWM controller.

```
&pwm0 {
  status = "okay";
  pinctrl-0 = <&pwm0_my_default>;
  pinctrl-1 = <&pwm0_my_sleep>;
  pinctrl-names = "default", "sleep";
};

&pinctrl {
  pwm0_my_default: pwm0_my_default {
    group1 {
      psels = <NRF_PSEL(PWM_OUT0, 0, 3)>;
    };
  };

  pwm0_my_sleep: pwm0_my_sleep {
    group1 {
      psels = <NRF_PSEL(PWM_OUT0, 0, 3)>;
      low-power-enable; ⑤
    };
  };
};
```

8. You are now ready for the actual application code in `main.c`.

Program 2. Send a servo to a series of pre-defined angles.

```
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
#include <zephyr/drivers/pwm.h>

LOG_MODULE_REGISTER(Lab05_Servo, LOG_LEVEL_INF);

#define SERVO DT_NODELABEL(servo)
const struct pwm_dt_spec servo = PWM_DT_SPEC_GET(SERVO);

/* Use DT_PROP() to get servo properties from overlay */
#define SERVO_MIN_PULSE_WIDTH DT_PROP(SERVO, min_pulse) ①
#define SERVO_MAX_PULSE_WIDTH DT_PROP(SERVO, max_pulse)
#define SERVO_CENTER DT_PROP(SERVO, center_pw)
#define SERVO_DEG_CONV DT_PROP(SERVO, deg_to_pw)
```

```
/** ②
 * @brief Convert angle to PWM pulse width
 *
 * @param angle Servo angle (integer degrees)
 * @return int representing the pulse width (in nanoseconds)
 */
int angle_to_pulsewidth(int angle) { ③
    int pw = SERVO_CENTER + angle*SERVO_DEG_CONV;
    if (pw < SERVO_MIN_PULSE_WIDTH) { ④
        pw = SERVO_MIN_PULSE_WIDTH;
        LOG_WRN("Out of servo range. Attempted to set to %d", angle);
    }
    if (pw > SERVO_MAX_PULSE_WIDTH) {
        pw = SERVO_MAX_PULSE_WIDTH;
        LOG_WRN("Out of servo range. Attempted to set to %d", angle);
    }
    return pw;
}

int main(void) {
    int angles[] = {0, +30, -30, +60, -60, +90, -90}; ⑤
    int num_angles = 7;

    if (!pwm_is_ready_dt(&servo)) {
        LOG_ERR("PWM controller is not ready");
        return -1;
    }

    while (true) {
        for (int i = 0; i < num_angles; i++) {
            LOG_INF("Angle set to %d deg", angles[i]); ⑥
            pwm_set_pulse_dt(&servo, angle_to_pulsewidth(angles[i]));
            k_msleep(2000);
        }
    }
}
```

① Access the servo-specific properties from the hardware overlay. Notice that the - found in the devicetree property name becomes _ when referring to devicetree property names in the C code. Notice that the C code is now independent of the specific servo used. If you wanted to use a different servo you would only need to change the overlay file.

② It is good practice to provide a comment block describing the inputs and output of a function as well as what it does. This one is formatted in the Doxygen style.

③ A function is defined that takes one input argument (an integer representing an angle in degrees) and the output is also an integer (the number of nanoseconds the pulse width should be for the servo to go to that angle).

④ The minimum and maximum pulse widths are not automatically enforced. The code you write needs to do that. In this case, if the pulse width is too small, the value is set to the

minimum and an error message is sent to the logger module.

⑤ Seven angles are stored in an array of integers.

⑥ The angles stored in the array are accessed by their index. C starts counting from 0 so the first angle is accessed through `angles[0]` and the seventh angle through `angles[6]`.

9. The Hitec HS-422 servo requires more power than can supplied directly by the nRF52840 DK development board. An external battery pack is required, but it must be used carefully to avoid damaging your development board.

An alternative approach is to use a micro servo such as the TowerPro SG92R. This less powerful servo can be driven directly from voltages supplied by the development board.

   a. Connect the ground bus of a breadboard to a GND socket on the development board.
   b. Connect the black lead from a 6 V battery pack to the same ground bus.
   c. Connect the black lead of the servo to the ground bus. It is important that all parts of the system agree on the ground voltage.

The power bus on the breadboard should have no connections other than those to be described. Connecting the 6 V of the battery pack (directly or indirectly) to the development board will damage it.

   d. Connect the red lead of the battery pack to the power bus.
   e. Connect the red lead of the servo to the power bus.
   f. Connect the yellow lead of the servo to P0.03.

10. Build your application and flash it to the development board.
11. Start a terminal connection so you can observe logging messages.
12. You should observe the servo rotating through a sequence of angles.

Your Turn

Servo controller

Assignment 5.1

In this exercise you will move the servo to a location selected by button presses.

- BUTTON 1: decreases the current angle by 10°
- BUTTON 2: increases the current angle by 10°
- BUTTON 3: sets the angle to 0°
- BUTTON 4: sets the angle to 90° if the current angle is positive or -90° if the current angle is negative

1. Access the GitHub Classroom link for this assignment on Blackboard.
2. Follow the usual steps for getting started with a repository from GitHub Classroom.
3. Write your code to control the servo. A button should only perform the specified behavior when it has been pressed and then released.
4. Test your program.
5. Update the `README.md`.

When your program and circuit are working successfully, remember to push the commits to the remote repository. Also, take a video of its successful operation (along with your reflection) and upload this to Blackboard.

Dimmer control

Assignment 5.2

In this assignment you will use three buttons to control the brightness of an external LED. The LED should have brightness levels ranging from 0% (off) to 100% (maximum brightness), adjustable in increments of 10%.

- BUTTON 1 increases the brightness (duty cycle) by 10%.
- BUTTON 2 decreases the brightness (duty cycle) by 10%.
- BUTTON 3 toggles the light off or on. The brightness setting should be remembered when it is toggled off so that when toggled on again the previous brightness setting is restored.

Each button press and release should trigger the corresponding action just once.

1. Create the repository using the GitHub Classroom link on Blackboard.
2. Update `main.c` and `README.md`.
3. Test your program.

When your program and circuit are working successfully, remember to push the commits to the remote repository. Also, take a video of its successful operation (along with your reflection) and upload this to Blackboard.