Lab 1: ADC and PWM

ECE342, Winter 2026

Department of Electrical and Computer Engineering

University of Toronto

Introduction

In this lab, you will explore how to interface the Nucleo platform with external devices. First, you will examine the analog signal capabilities of the STM32F446ZE board. Using its analog-to-digital converter (ADC), you will measure the voltage across a variable resistor (a potentiometer). Next, you will study pulse-width modulation (PWM), a common technique in embedded systems that allows digital platforms to emulate analog behavior. You will apply PWM to control the brightness of an LED using both software delays and hardware timers, with the brightness controlled by the potentiometer reading.

1. Digitizing Analog Values

In this section, you will use a potentiometer (a variable resistor) as an external control input. The analog signal from the potentiometer must be converted into a digital value that the CPU can process.

1.1 Potentiometer

A potentiometer is a variable resistor with three terminals. It can be modeled as two resistors, R_1 and R_2, connected in series. The total resistance between the outer terminals is fixed ($R_{\text{const}} = R_1 + R_2$).

By turning the dial of the potentiometer, you adjust the relative values of R_1 and R_2. If the dial is turned fully in one direction, $R_1 = 0$ and $R_2 = R_{\text{const}}$. If turned fully the other way, $R_2 = 0$ and $R_1 = R_{\text{const}}$.

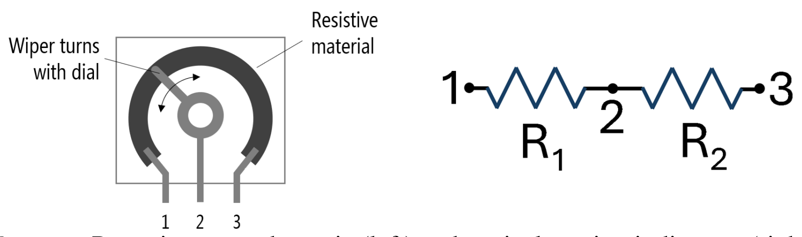

Figure 1. Potentiometer schematic (left) and equivalent circuit diagram (right).

Our system cannot directly measure resistance. To determine the potentiometer's setting, we connect it as a *voltage divider*. For example, if terminal 1 is connected to 3.3 V and terminal 3 to 0 V, then the voltage at terminal 2 is given by:

$$V_{\text{out}} = \frac{R_2}{R_1 + R_2} \cdot V_{\text{in}}$$

where $V_{\text{in}} = 3.3\,\text{V}$. This output voltage varies smoothly between 0 V and 3.3 V as the dial is turned, and can be read by the Nucleo board using its analog-to-digital converter (ADC).

Note: Because the potentiometer is a resistor, orientation does not matter.

1.2 Analog to Digital Conversion

As you saw in class, an analog-to-digital converter (ADC) is used to convert an analog signal (such as a voltage) into a digital value that the CPU can process. To begin, we must connect the potentiometer to the board so the ADC can read the voltage. Figure 2 shows the CN8 and CN9 headers on the Nucleo platform. Make the following connections between your board and the potentiometer:

1. Terminal 1 $\rightarrow$ 3V3 pin on CN8
2. Terminal 2 $\rightarrow$ PF_5 pin on CN9 (labelled 'A4' on the board)
3. Terminal 3 $\rightarrow$ GND pin on CN8

Next, we will read the voltage at terminal 2 using the ADC. The Nucleo platform provides several functions to interact with the ADC:

1. **HAL_ADC_Start()**: initiates a conversion in the ADC hardware. This function is non-blocking, allowing the processor to perform other tasks while waiting.

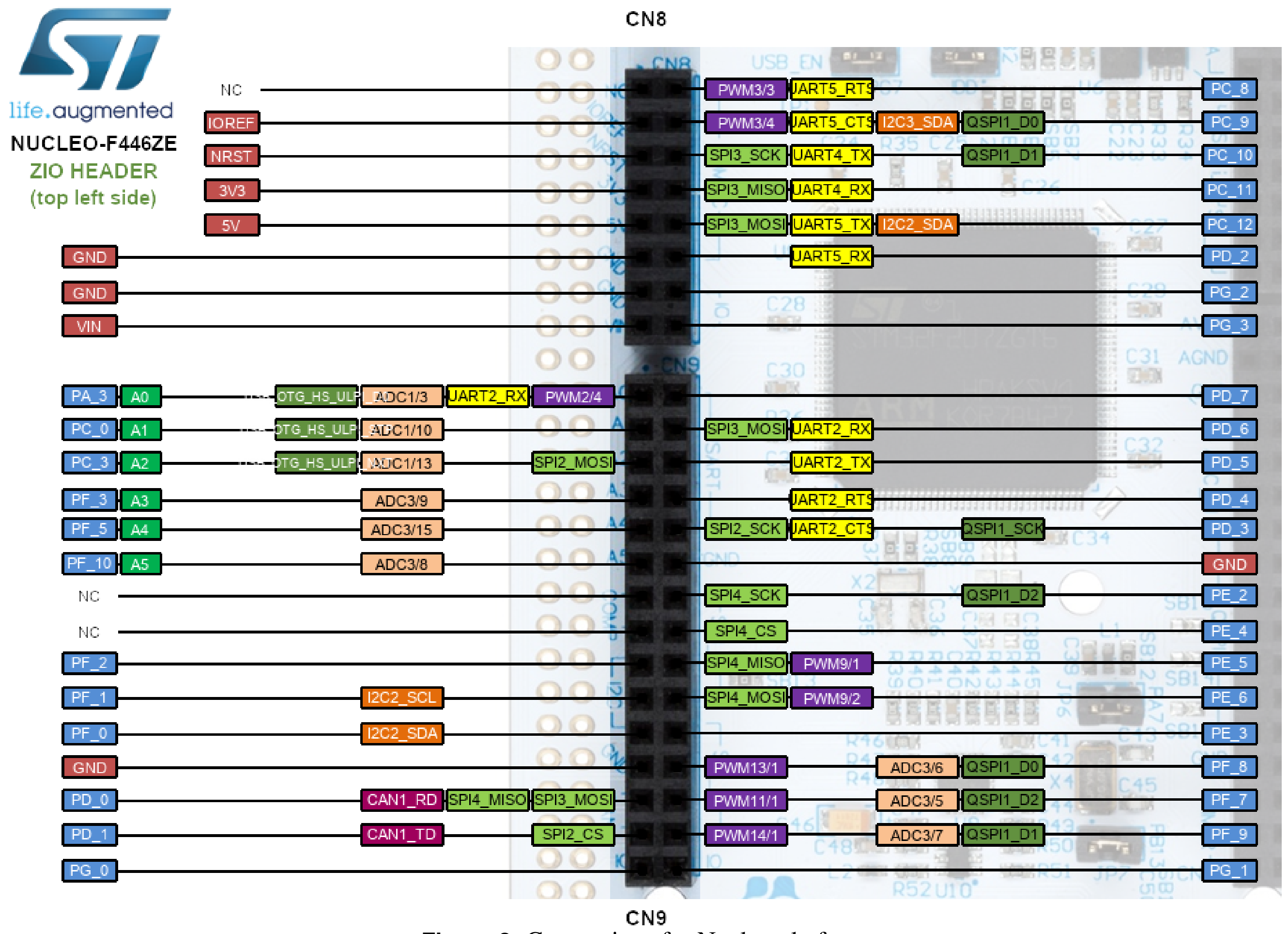

Figure 2. Connections for Nucleo platform

2. **HAL_ADC_PollForConversion**(): blocks execution until the ADC conversion has completed.
3. **HAL_ADC_GetValue**(): returns the result of the ADC conversion.

The lab kit includes example code that demonstrates how to call these functions and prints the ADC value over the serial port. Modify this code to continuously sample the potentiometer using the ADC. The 12-bit ADC on the Nucleo board can return $2^{12} = 4096$ distinct values. Thus, as you turn the potentiometer, you should see values ranging approximately from 0 to 4095.

Note: If you place the print command directly inside the `while(1)` loop without any delay, the serial terminal will be flooded with updates. To avoid this, add a delay of at least 500 ms to the loop.

2. Pulse-width Modulation (PWM)

Pulse-width modulation (PWM) is a technique that allows a digital system to emulate analog behavior by rapidly switching a signal on and off. The **duty cycle** is the fraction of each cycle that the signal is “on.” By adjusting the duty cycle, we can control the average voltage delivered to a device. For example, a 30% duty cycle with a 3.3V supply delivers an average voltage of approximately 1V.

In this lab, you will use PWM to adjust the brightness of LEDs on the Nucleo board.

2.1 LED Brightness with Toggle

Begin by downloading the provided Lab 1 kit and examining the code inside the `while(1)` loop. If you run the code as-is, the LED will blink with a period of 2 seconds using the function `HAL_GPIO_TogglePin()`.

To observe how changing the toggle delay affects LED brightness, try reducing the `HAL_Delay()` value to 100 ms. What do you notice? Next, adjust the delay to 10 ms, and then to 1 ms. How does the LED's brightness change as you reduce the delay?

For small delay values, the LED appears to be ON continuously but looks dimmer. The LED itself is just as bright when ON, but because it toggles between ON and OFF states, our eyes perceive the average brightness. With toggle, the LED is ON for 50% of the time and OFF for 50% of the time. We can think of this as the LED being "50% ON." However, with this approach the LED can only be 0% ON (always off), 50% ON (toggling), or 100% ON (always on). To achieve finer control of brightness, we need true PWM control.

2.2 Software PWM with HAL_GPIO_WritePin()

For finer control of LED brightness, we can adjust the ratio of ON time to OFF time using `HAL_GPIO_WritePin()`. This function requires an additional parameter specifying the desired pin state. To set the GPIO pin high (1), use `GPIO_PIN_SET`; to set it low (0), use `GPIO_PIN_RESET`.

For example, if you set the LED ON using `HAL_GPIO_WritePin()` with `GPIO_PIN_SET`, delay for 1 ms, then set it OFF with `GPIO_PIN_RESET`, and delay for 9 ms, the LED will be ON for 1 ms and OFF for 9 ms. This corresponds to a duty cycle of 10%, and you should observe that the LED appears much dimmer than with toggle.

Next, experiment with different duty cycle values to see how brightness changes. Consider whether you can perceive a difference between 10% and 11% duty cycle, or between 30% and 40%. You can create a list of duty cycle values in your code and update the value each time the button is pressed.

Finally, investigate the smallest duty cycle at which the human eye begins to detect the LED turning on and off. At very small duty cycles, the ON pulses are spaced far enough apart that the LED no longer appears steady, and you can clearly perceive flickering. For instance, try a duty cycle of 1% and observe whether the LED looks like it is blinking rather than continuously dim.

Note: Human Eye and Flicker

The human eye cannot respond to infinitely fast changes in light. Instead, it has a threshold known as the *critical flicker fusion frequency*, typically around 60 Hz under normal conditions. If the PWM frequency is above this threshold, the LED appears steady and only its average brightness (set by the duty cycle) is perceived. If the PWM frequency is below this threshold, or if the duty cycle is very small so that ON pulses are widely spaced, the eye can detect the LED turning on and off as flicker.

2.3 Hardware PWM with Timers

The previous approach used software delays to generate PWM signals. While this works, it is inefficient because the CPU must actively manage the timing. A better approach is to use hardware timers, which can generate PWM signals automatically without consuming CPU cycles.

In this section, you will configure a hardware timer on the STM32F446ZE to generate a PWM signal that controls a third LED on the board. The timer will be set up to:

1. Generate a PWM signal at a specific frequency (e.g., 1 kHz)
2. Allow you to adjust the duty cycle based on the ADC reading from the potentiometer
3. Output the PWM signal to control the third LED's brightness

The STM32 HAL library provides functions to configure and control timer-based PWM:

1. **HAL_TIM_PWM_Start()**: starts the PWM generation on a specific timer channel
2. **__HAL_TIM_SET_COMPARE()**: adjusts the duty cycle by setting the compare value

The lab kit includes example code showing how to configure a timer for PWM output. You will need to:

1. Configure the timer's period (determines PWM frequency)
2. Configure the timer's pulse (determines duty cycle)
3. Map the ADC value to an appropriate pulse width
4. Update the pulse width dynamically as the potentiometer is adjusted

With hardware PWM, the LED brightness can be controlled smoothly and the CPU remains free to perform other tasks. Compare the performance and smoothness of hardware PWM versus software PWM using delays.

3. Mapping ADC to PWM Duty Cycle

Now that you can read ADC values from the potentiometer and generate PWM signals, the next step is to convert the ADC value into a PWM duty cycle to control LED brightness.

At first glance, you might consider mapping all 4096 ADC values directly to duty cycles, ranging from 0.0244% (1/4096) up to 99.97% (4095/4096). But is this the best approach? Since we are setting the duty cycle using the `HAL_Delay()` function for software PWM, extremely small duty cycles (e.g., 0.01%) are not practical.

The experiments you performed earlier should guide you in determining a reasonable number of duty cycle steps. Once you decide on this number, think carefully about how to map the ADC values into those duty cycle levels.

Worked Example

Suppose you decide that 10 distinct brightness levels are sufficient. You can divide the ADC's 4096 values into 10 equal ranges, each corresponding to one duty cycle. For instance:

$$\text{Duty cycle} = \frac{\text{ADC value}}{4096} \times 100\%$$

If the ADC value is 2048 (approximately half of the maximum), then:

$$\text{Duty cycle} = \frac{2048}{4096} \times 100\% \approx 50\%$$

In practice, you would round the ADC value into one of the 10 bins (e.g., 0–409, 410–819, etc.), and assign each bin to a duty cycle such as 0%, 10%, 20%, ..., 100%. This way, turning the potentiometer smoothly adjusts the LED brightness in clear, perceptible steps.

For hardware PWM using timers, the mapping is similar, but instead of calculating delay times, you map the ADC value to a compare value for the timer. If your timer period is set to 1000, then:

$$\text{Compare value} = \frac{\text{ADC value}}{4096} \times \text{Timer Period}$$

For example, with an ADC value of 2048 and a timer period of 1000:

$$\text{Compare value} = \frac{2048}{4096} \times 1000 = 500$$

This gives a 50% duty cycle. You can use `__HAL_TIM_SET_COMPARE()` to update this value dynamically as the potentiometer is adjusted.

Be prepared to explain: How many brightness levels you chose, why that number is practical, and how you mapped the ADC values to duty cycles for both software and hardware PWM during your lab session.

4. In-lab Demo

During your lab session, you are required to demonstrate the following to your TA:

1. Show that you can read ADC values from the potentiometer and display them over the serial port.
2. Use the potentiometer to adjust the brightness of an LED using software PWM (HAL_Delay method).
3. Use the potentiometer to adjust the brightness of the third LED using hardware PWM (Timer method).
4. If the potentiometer control is not functioning correctly, show that you can change the LED brightness by directly modifying the duty cycle values in your code and re-running the program.
5. Explain your choice of brightness levels and how you mapped ADC values to duty cycles.