# Pulse width modulation and timers

## 1 Introduction

Up to now, all the communications with the Arduino have used digital interfaces. Reading the state of the light sensor. Activating a solenoid. Lighting an LED. Here, you will explore how the ATmega328 emulates *analog outputs*, voltages that range between 0 – 5 V.

The ATmega328 is not actually capable of producing a true analog signal. While there are microcontrollers that do have on-board digital-to-analog converters, the ATmega328 can only produce 0V or 5V signals – not 3V, or 1.8V, or anywhere else in between. Instead, the chip uses a common technique called *pulse width modulation*, or simply PWM, to *mimic* analog voltages. In short, the microcontroller toggles a pin between 0V and 5V at a very high frequency so that the *average* voltage approximates the desired voltage. By changing the *duty cycle*, the proportion of the time that the pin is `HIGH`, the apparent voltage can be controlled very precisely. For many devices – motors, LEDs, and so forth – the switching occurs so fast that it has no practical difference from a true analog voltage.

The machinery to produce PWM signals can be complicated – it involves setting several control registers – but Arduino has created a convenient function, `analogWrite()`, to make PWM easy. Still, in this lab you'll see how you can change frequencies, and use that knowledge to create different tones with a piezo buzzer, which will serve as the alarm for your alarm system.

### 1.1 Objectives

Upon successful completion of this lab, the student will be able to:

- Implement PWM using Arduino libraries and direct control,
- Produce square waves of different frequencies, and

## 2 Background materials

Before coming to lab, the student should review the following materials:

- *Pulse width modulation.* Everything you need to know should be included below.

**Each student must complete the Pre-lab Worksheet and turn it in at the start of lab.**

## 2.1 Timers

There are six pins on the Arduino that are capable of create a PWM signal, indicated with a $\sim$ next to the pin. Each of three internal timers controls two of the pins, as shown in Table 1, which also indicates the size of the timer (number of bits) for reference.

| Timer | N (bits) | pin A | pin B |
|---|---|---|---|
| 0 | 8 | 6 | 5 |
| 1 | 16 | 9 | 10 |
| 2 | 8 | 11 | 3 |

Table 1: Timers on the ATmega328.

`Timer0` and `Timer2` are *8-bit* timers and `Timer1` is a *16-bit* timer. The size of the timer will affect the range of frequencies that you can create with it. As described in class, the timers each have an independent *pre-scaler* that is used to set how fast the timer counts. If the pre-scaler is set to 1, the timer will increment by 1 for each tick of the system clock. Since the clock on the Arduino runs at 16 MHz, this means it will take an 8-bit timer $256/16MHz = 16\mu s$ to roll-over. By setting the pre-scaler to 8, it will take 8 ticks of the system clock to increment the timer by 1. As such it will take an 8-bit timer $256 \times 8/16MHz = 256\mu s$ to roll-over. For an N-bit timer, the general formula for calculating the rollover period, $\Delta T$ is

$$\Delta T = \frac{2^N \times P}{f_{sys}}$$

where $P$ is the pre-scaler and $f_{sys}$ is the frequency of the system clock. Alternatively, the rollover *frequency* is given by $f_{timer} = 1/\Delta T$, or

$$f_{timer} = \frac{f_{sys}}{2^N \times P} \tag{1}$$

When using the timer for PWM, the duty cycle is changed by setting a compare value, `OCRnx`, where `n` is the timer (0, 1, or 2) and `x` is one of the two pins connected to that timer, A or B. In Fast PWM mode, the timer counts up from $0 \rightarrow 2^N - 1$ and then rolls back over to 0. Whenever it rolls over, it sets the corresponding pin to `HIGH` and when it reaches 1 greater than the compare value, it switches the pin back to `LOW`. Thus, for an N-bit timer, the duty cycle can be calculated as:

$$dutycycle = \frac{OCRnx + 1}{2^N} \tag{2}$$

When you call `analogWrite(<pin>, <value>)`, all the Arduino library does is set the compare value, `OCRnx`, to `<value>`. Note that passing 0 to `analogWrite(<pin>, <value>)` would normally produce a small, but non-zero duty cycle; however, the Arduino library has an exception for that case where it sets the pin to `LOW` and turns off the PWM function so that the functionality is more intuitive.

### 2.2 Changing frequencies

Changing the pre-scaler alone would not give much control over the timer frequency. To make sounds of different frequencies, it is useful to have finer control over the frequency. Indeed, fine-scale control can be achieved by running the timer in CTC mode, which is short for Clear Timer on Compare Match. In CTC mode, the timer rolls over at a specific value, the `TOP` value. The general formula then becomes:

$$f_{timer} = \frac{f_{sys}}{(TOP + 1) \times P}$$

where the +1 is there because it takes an extra timer increment to get from `TOP` back to 0. In CTC mode, you can set a pin to toggle every time the timer rolls over, which will produce a square wave. Connecting the pin to a speaker (or piezo buzzer) will cause it to make a sound. Note that if you toggle a pin on rollover, the frequency will be half that of the timer, because it will either be `HIGH` or `LOW` for each timer cycle. When in CTC mode, `TOP` is set using `ORCnx`, which shouldn't be confused with it's functionality in the previous section.

## 3 In lab

### 3.1 Pulse width modulation

#### Procedure

1. Construct the circuit shown in Figure 1. Use Equation 2 to calculate the theoretical duty cycle and record it in Table 2 for each of the values there.

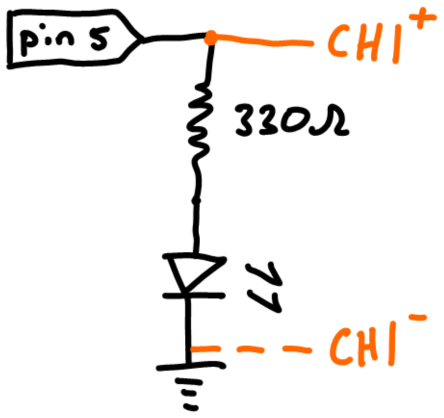

Figure 1: Yet another LED circuit. Be sure to use pin 5.

2. Write a program that declares pin 5 as an `OUTPUT` and sends a PWM command to it using `analogWrite()`. In the `loop()`, print the value in `OCR0B` once per second – you may use `delay()`.

3. For each value in Table 2, change the value that is passed to `analogWrite()` and run the code. Connect your oscilloscope to the output of pin 5 and set the trigger to capture the rising edge. Adjust the time scale until you can see four or five PWM cycles. Dig through the built-in measurements until you find "Duty cycle" and compare the measured duty cycle to the theoretical value. Complete the table.

| PWM Value | Exp. Duty Cycle | Act. Duty Cycle | Avg. Voltage | Perceived Brightness |
|---|---|---|---|---|
| 0 | | | | |
| 50 | | | | |
| 100 | | | | |
| 150 | | | | |
| 200 | | | | |
| 255 | | | | |
| 300 | | | | |

Table 2: Table for PWM values.

4. What happens to the LED when you pass 300 as the PWM parameter? Does it glow brightly? What is the measured duty cycle? Explain.

5. Instead of using `analogWrite()`, change your code to read:

```
OCR0B = 75;
```

What is the duty cycle? Is this what you expect?

6. Use your oscilloscope to determine the frequency of the PWM signal. Knowing that pin 5 is controlled by `Timer0`, an 8-bit timer, determine what pre-scaler is being used by the Arduino library using Equation 1.

7. Add a line to your `setup()` function to print the `TCCR0B` register to the Serial Monitor,

```
Serial.println(TCCR0B, HEX);
```

The last three bits of this register control the pre-scaler for `Timer0`. Consult the pages posted on collab (TCCR0B.pdf) to check what the bits correspond to. Do they agree?

**Demonstrate your working system to an instructor!**

## 3.2 Sounds

Arduino does have a `Tone` library, but not only is it more cumbersome than it needs to be[1], it is useful for you to understand how to fine-tune the frequency of a timer manually. In this section, you'll use the 16-bit `Timer1`, which gives you a huge range of frequencies, including the painful ones above 10 kHz. You'll use a piezo buzzer to make the sounds.

### Procedure

1. Build the circuit shown in Figure 2. The datasheet indicates that the piezo is meant to be driven using 3.3V – since the Arduino outputs 5V from each pin, you'll use a low-side driver. Size the resistor, R, using the rules-of-thumb for a low-side driver. Connect the oscilloscope as shown.

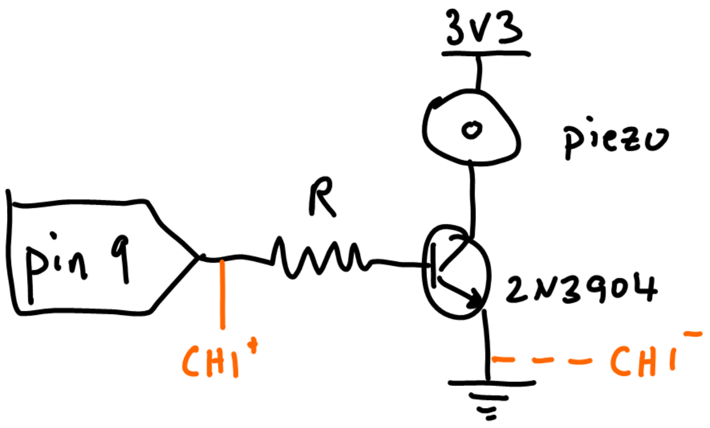

Figure 2: Circuit for the piezo element.

2. Wire a standard button to whichever pin you'd like and edit the code appropriately.
3. Download `jammin_tunes.ino` from collab. The `setup()` function sets up `Timer1` to run in CTC mode. There are comments in the code. Use the `Button` class you made last week to activate and deactivate the piezo. Use Equation 1 to determine the frequency.
4. Run the program. Press the button to turn on and off the piezo. Use the oscilloscope to determine the frequency. Do they agree?
5. Using Equation 2.2, calculate the value of `TOP` needed to create a tone with a frequency of 440 Hz (middle A) and set `OCR1A` to one less than the value you calculated.
6. Run the new program an verify that the frequency is correct.

**Demonstrate your working system to an instructor!**

[1]Stupidly cumbersome, if you ask me.

## 3.3 Pre-lab worksheet

**To be turned in at the start of lab.** Completed individually, but you are welcome to talk to others about the material.

*Answers with incomplete or incorrect units will be marked as wrong!*

1. Give three reasonably distinct examples of where an analog output might be useful.

2. What is the duty cycle of a PWM signal created with the statement `analogWrite(5, 42);`?

3. What is the frequency of an 8-bit timer on an Arduino Uno if the prescaler is 8 and `TOP` is set to 199?