Dylan Falzone
9/20/23
4539-8897

EE Design 1 Technical Report
Microcontroller Basics

Introduction

One of the most defining traits of human beings is our creativity. Admittedly, other species do possess some creativity, evidenced by their ability to design structures like webs or hives, but such things are tiny compared to what the human mind is capable of. We make paintings, cities, 5-star meals, video games, and so much more. Yet despite our boundless creativity, in many cases we are restricted to thinking within a set of parameters. Architects must consider material costs, spatial limitations, and what is possible within Earth's gravity. Chefs must consider the ingredients they have on hand, as well as their costs too. All things considered, it is rare to find a situation in which imagination is truly the only limiting factor. Luckily, we get to work with microcontrollers.

A microcontroller is like a million blank canvasses, ready to be painted. While microcontrollers do technically have limiting factors like processing speed and storage space, clever optimizations and proper planning can make these a non-factor for the majority of projects. With these small devices, countless combinations of components and systems can be designed and intelligently driven, allowing for endless possibilities. In this lab, we will discuss one such possibility, as an example of the true power of the microcontroller.

Methods

Circuit

Using our microcontroller of choice, the Raspberry Pi Pico, the goal was to design a system capable of manually toggling between two different outputs: a 2Hz flashing LED, and a buzzer running at 2kHz.

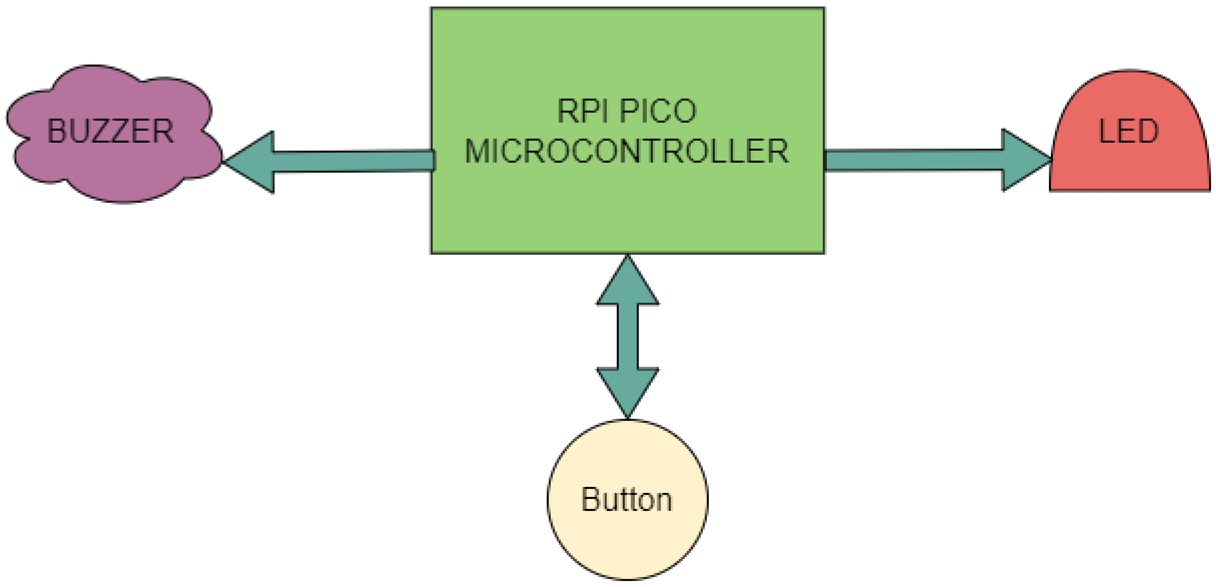


Figure 1: Software block diagram.

As shown in figure 1, such a system would require a button, an LED, a buzzer, and of course, a microcontroller. The microcontroller, being the brains of the circuit, are in the middle, surrounded by the other components. The LED and Buzzer are output devices, so the arrows indicate that power to them is flowing in one direction. The button, however, uses a bi-directional arrow. This is because, at this point in the process, I was unsure on whether I would use the button in a pull-up or pull-down configuration. A pull-up configuration would have the button press grounding the pin it is connected to, while a pull-down configuration would have the button press providing power to said pin [1].

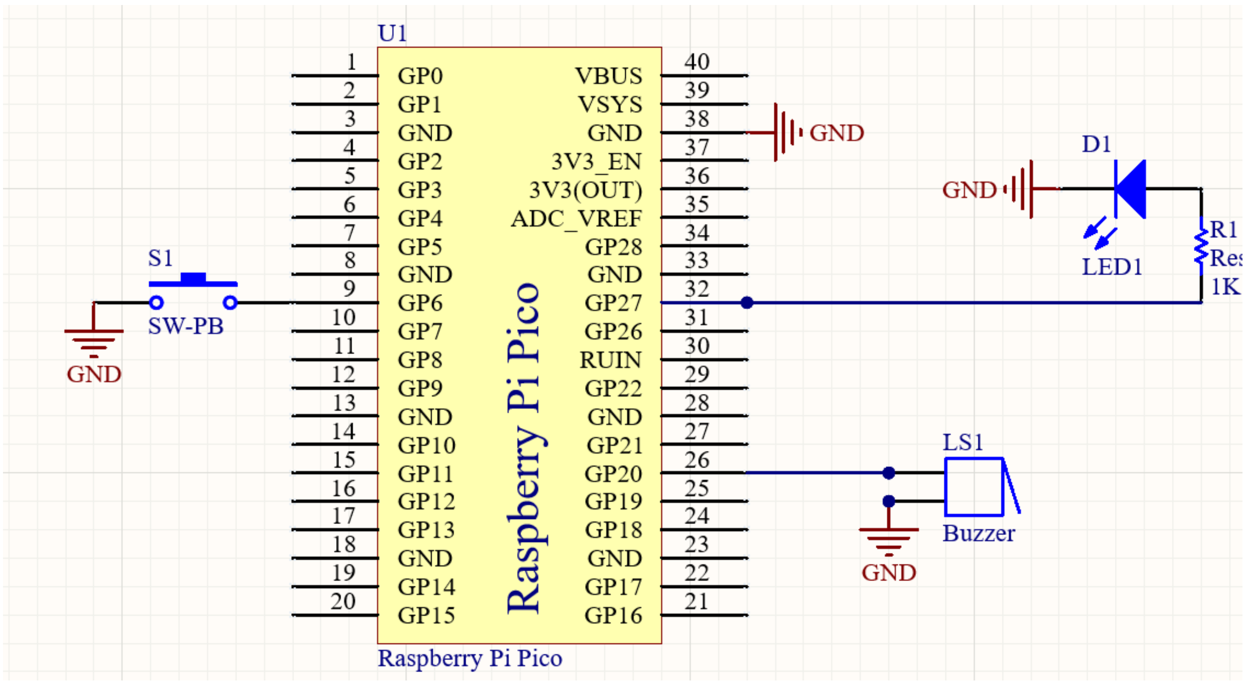


Figure 2: Altium schematic for the circuit.

After making the simple block diagram for the circuit, the logical next step was to create the schematic. I began with the components that I knew I would need and wired them up to pins of my choosing. Thankfully, the PICO was included in one of the given Altium libraries. I was careful to choose the right pins, since the numbering on the pins is different from their GP numbering, and some of them have specific functions like the ground pins and the pins on the top right. I used GP pins 27, 20, and 6 for the LED, buzzer, and switch respectively. I also made sure to add a resistor to the LED so as not to burn it out.

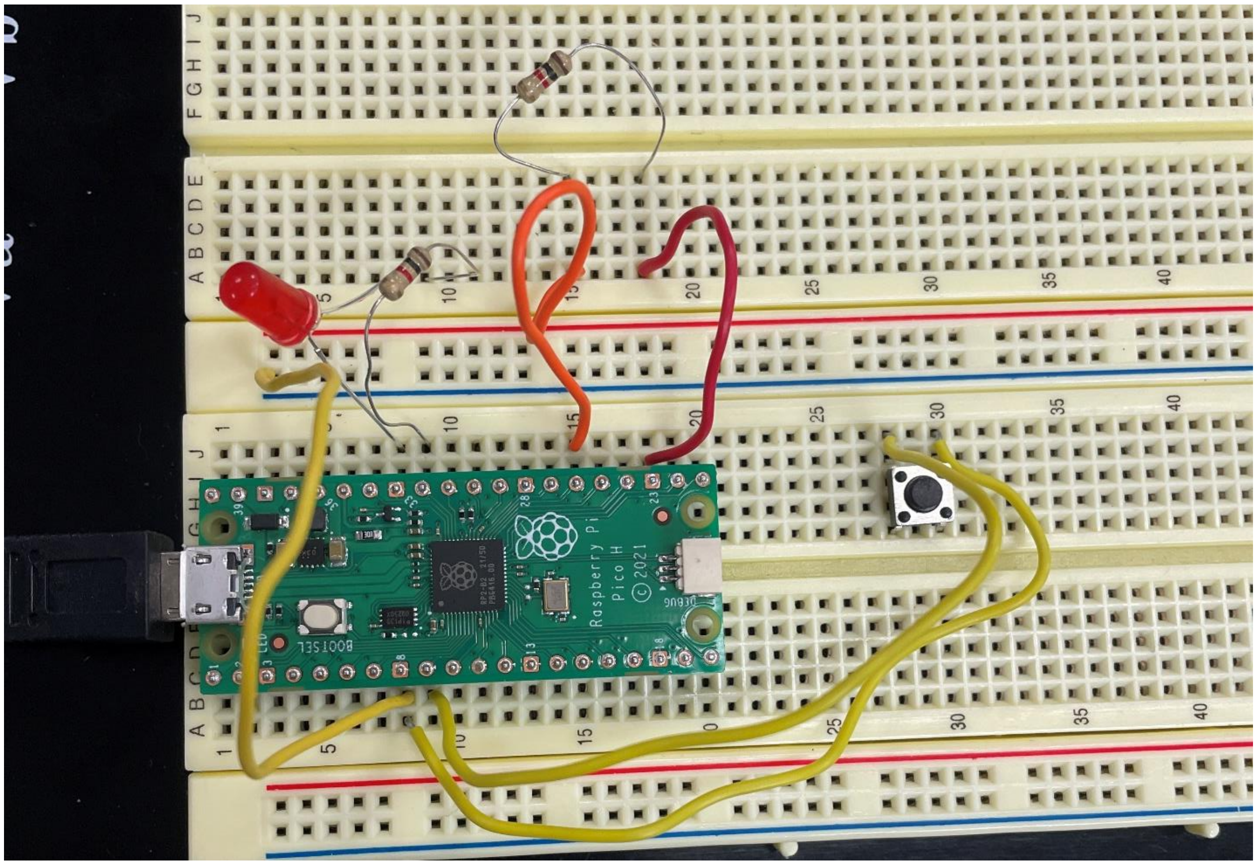


Figure 3: Circuit on breadboard.

For the final version of the circuit, the button was used in a pull-up configuration where it connects the pin to ground when pressed. This helped significantly with debouncing, as when it was in the other configuration, soft 1's caused significantly more bouncing. Additionally, for most of the lab I used a simple 1k resistor in place of the buzzer, to preserve the ear health of those around me. However, I did try the buzzer at one point and confirmed that it worked as intended.

Software

The software would prove to be the most difficult part of this project. The primary challenge being that the most optimal way to program a Raspberry Pi Pico is using python, or more specifically, micropython. Not knowing either of these languages, I had to utilize a wide variety of internet resources to get the basics down as quickly as possible. Thankfully, the logic of the required code wasn't too complex, as shown below.

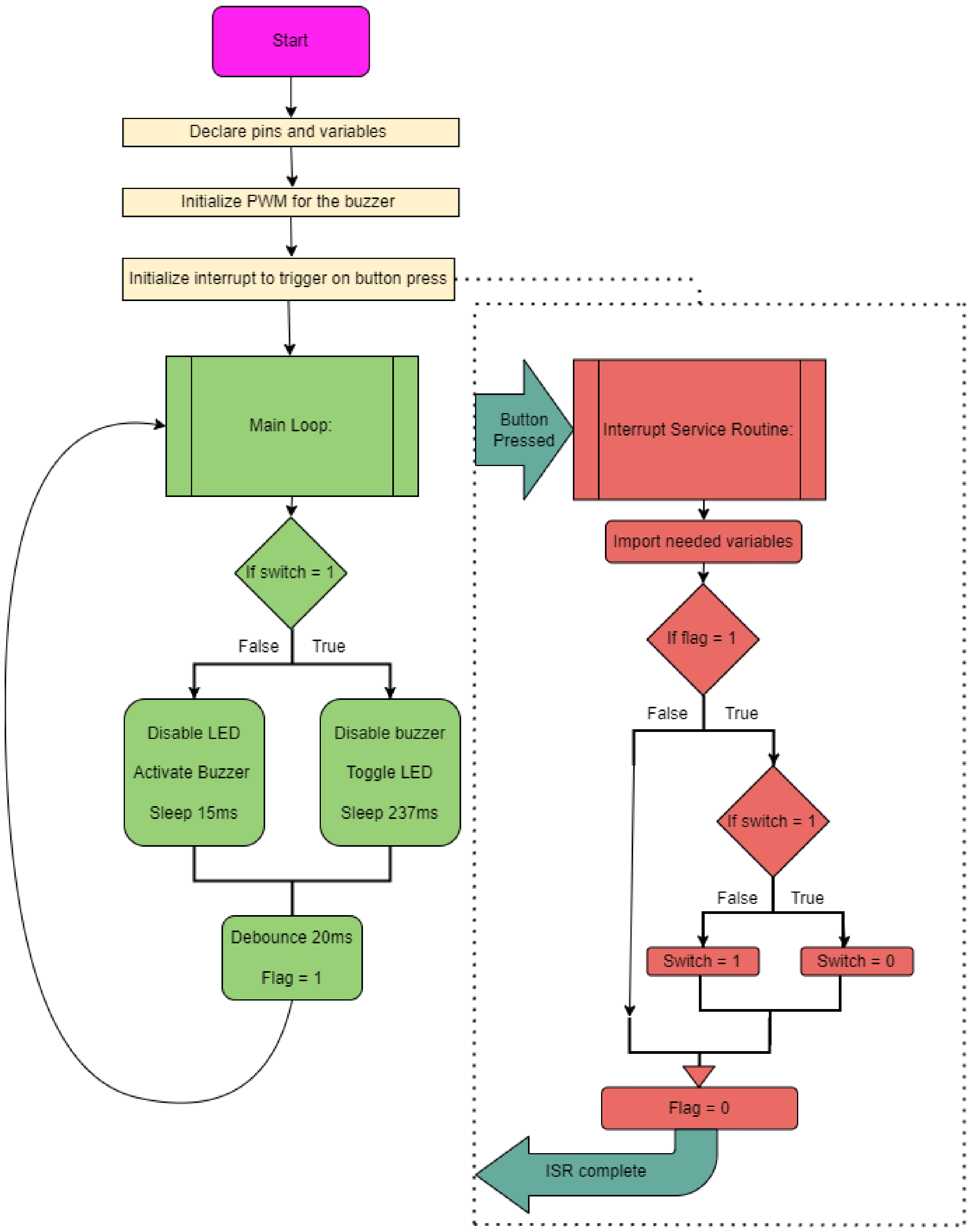


Figure 4: Software block diagram for the system.

Starting with the setup section pictured in beige, variables were needed for the pins, the interrupt flag, and the system toggle (switch). PWM turned out to be the best method for powering the buzzer, unlike the LED, so the next step was to initialize the PWM for the correct frequency. After that came the interrupt initialization which was particularly confusing at first. The most helpful tool in understanding this foreign language was Chat GPT, which was instrumental to my success in this lab. It helped me to understand that there needed to be a line initializing the interrupt separately from interrupt service routine itself, which is similar to C in concept [2].

The main loop was the simplest part for me to write, as it only consisted of basic logic and was not so reliant on formatting knowledge. An 'if else' statement was used to determine which state the system was in, outputting to the LED or the buzzer. Each path would disable the opposing output and enable the correct input. Sleep statements were needed here to ensure proper functioning at the correct frequencies, but I will go more into that in the results section. The main loop ends with setting the interrupt flag back to 1, allowing the interrupt to be triggered again for the next loop. There is also a 20ms wait time for debouncing. This ensures that the interrupt cannot be triggered again for 20ms after it has been triggered, which effectively debounces the button very well [2].

Finally, in the ISR, pictured in red. There are two nested 'if else' statements, allowing the ISR to be skipped if it is triggered twice within the same loop, since flag would still have the value 0, which it is set to at the end of the ISR. The logic here is just to toggle switch from its current value to the opposite whenever the ISR is triggered.

<u>Results</u>

The results should be displayed in this section, which include figures and descriptive text for each figure. Figure captions need to be given as well, occurring below the figure. Use subheadings as necessary.

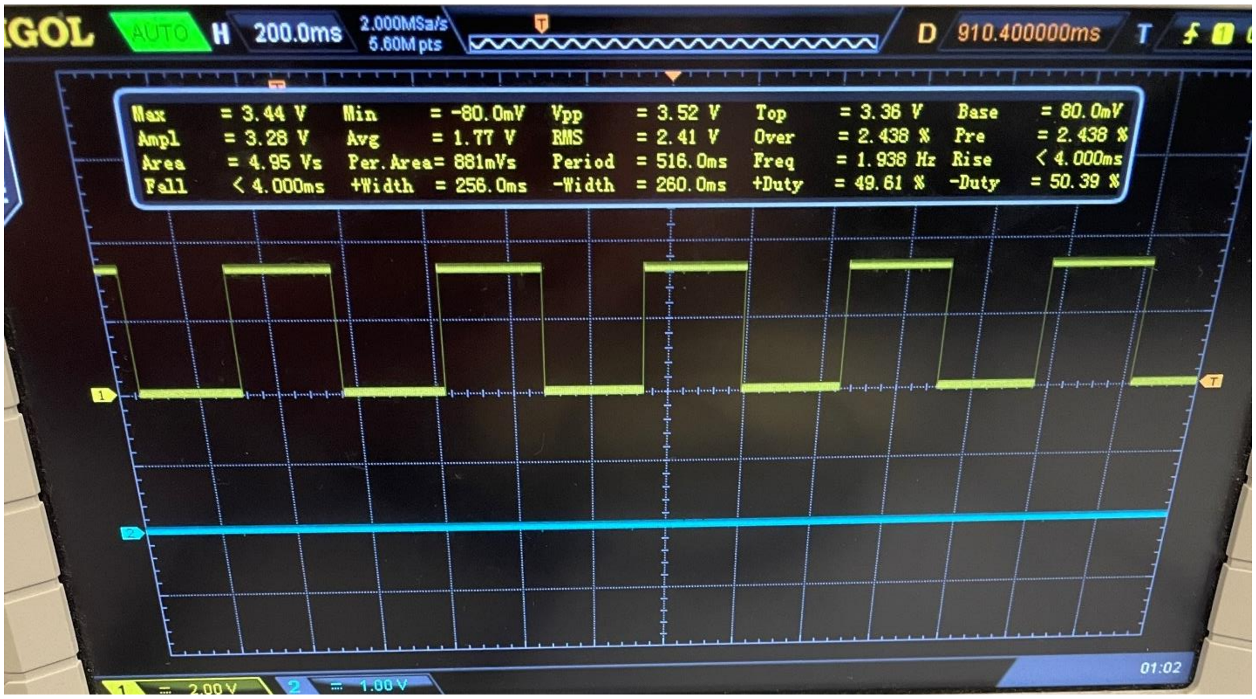
Figures needed:

- Breadboard Circuit Results:
  - Figure of waveform at the LED output showing a 2 Hz square wave
  - Figure of waveform at the buzzer output showing a 2 kHz square wave

Specific questions to answer in this section:

1) *How did you define the frequency of the output digital signal; when does the specified frequency not give you what you expect?*
2) *Explain the difference between polling and interrupts and examples of when to use them.*

LED

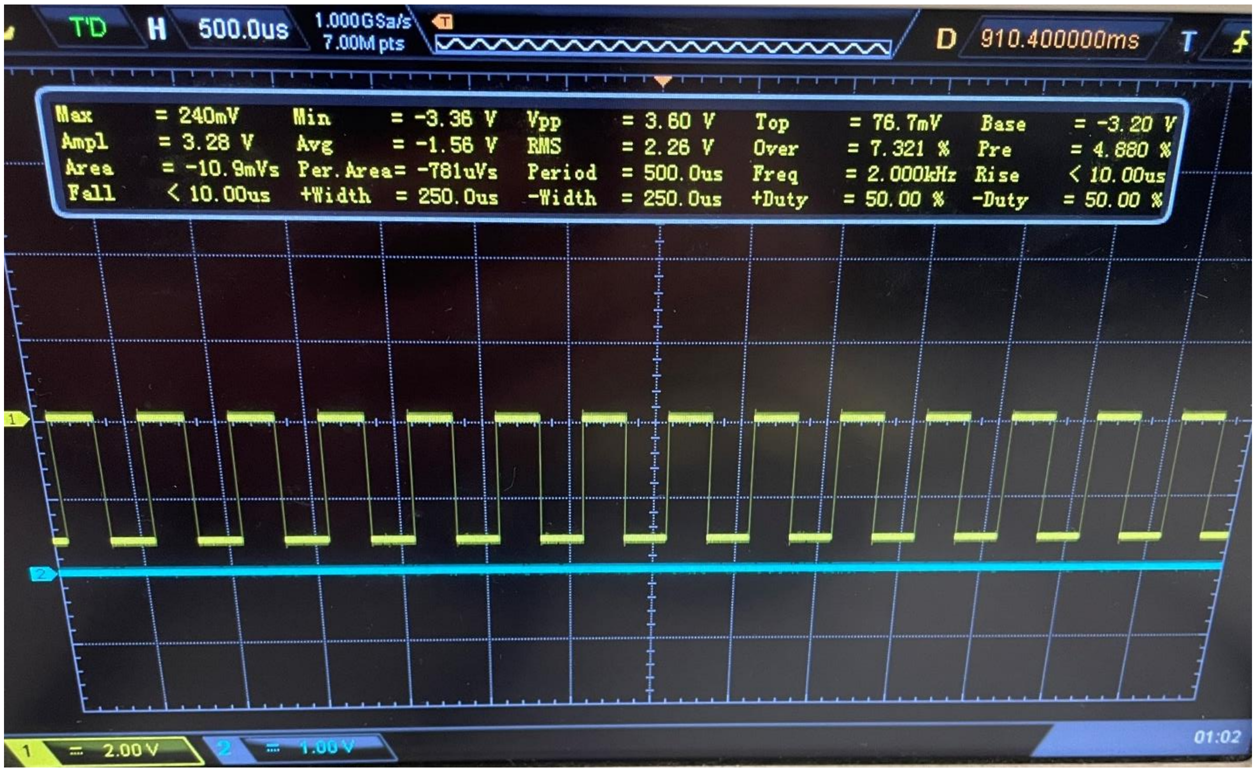


```
Max   = 3.44 V    Min      = -80.0mV  Vpp    = 3.52 V    Top   = 3.36 V    Base  = 80.0mV
Ampl  = 3.28 V    Avg      = 1.77 V   RMS    = 2.41 V    Over  = 2.438 %   Pre   = 2.438 %
Area  = 4.95 Vs   Per.Area= 881mVs    Period = 516.0ms   Freq  = 1.938 Hz  Rise  < 4.000ms
Fall  < 4.000ms   +Width   = 256.0ms  -Width = 260.0ms   +Duty = 49.61 %   -Duty = 50.39 %
```

Figure 5: Voltage output waveform over the LED.

Figure 5 shows that the LED receives a square wave voltage, with an average frequency of 1.938Hz (within the 5% tolerance of 2Hz). Achieving this had to be done in a different way than the buzzer, since 2Hz was too small to use the PWM function. So, as previously discussed, it was done by using a simple sleep ms function. This needed to be tweaked to find the exact correct value, but eventually it was within range and close enough to 2Hz.

Buzzer



```
Max   = 240mV     Min       = -3.36 V   Vpp    = 3.60 V    Top   = 76.7mV     Base   = -3.20 V
Ampl  = 3.28 V    Avg       = -1.56 V   RMS    = 2.26 V    Over  = 7.321 %    Pre    = 4.880 %
Area  = -10.9mVs  Per.Area= -781uVs     Period = 500.0us   Freq  = 2.000kHz   Rise   < 10.00us
Fall  < 10.00us   +Width    = 250.0us   -Width = 250.0us   +Duty = 50.00 %    -Duty  = 50.00 %
```

Figure 6: Voltage output waveform over the buzzer.

The buzzer was much easier to implement correctly. As shown, it has an exact frequency of 2kHz. This is because the PWM function was used, so processing speed and sleep functions did not interfere with its time scale whatsoever. This is also why the LED waveform was not as perfect. I was very happy with how this waveform turned out and in the future, I will attempt PWM before I try manual timing, since it is much more consistent.

Additionally, interrupts can cause problems with manual timing but not PWM. Since the interrupts take the focus of the processor, the code gets delayed, and a manually timed waveform will become incorrect. For reasons like this, interrupts are much more useful than manual polling as well. Polling takes extra cycles, delaying the system even further, as well as only being trigger-able at a certain point in the code. For example, if we used polling for this lab instead, you would be unable to switch modes until a full iteration of the code had run. But with interrupts, we can press the button in the middle of the loop, and it will work without any waiting. Polling may still be useful in specific scenarios, especially when little to no complexity is required in the main loop, however, I find interrupts much more generally useful.

Conclusion

In this lab, we dipped our toes into the ocean of possibilities contained within the microcontroller. We used PWM and interrupts to make a simple circuit, a toggle between an LED and a buzzer. Yet there are countless other ways in which this could have been accomplished, such as polling and manual timing. Such is the nature of microcontrollers, there are many roads that lead to the same place and there are countless places. The knowledge we gained from this lab will undoubtedly bolster our future efforts and act as a framework for labs and projects to come.

References

[1] Utmel, “What are the differences between pull up and pull down resistors?,” Utmel, https://www.utmel.com/blog/categories/resistor/what-are-the-differences-between-pull-up-and-pull-down-resistors (accessed Sep. 20, 2023).

[2] “Raspberry pi pico interrupts &amp; button interfacing tutorial using MicroPython.,” Electrocredible, https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/ (accessed Sep. 20, 2023).

[3] “Debouncing a pin input,” 1. Debouncing a pin input - MicroPython latest documentation, https://docs.micropython.org/en/latest/pyboard/tutorial/debounce.html (accessed Sep. 20, 2023).

Appendix

```
import machine

import utime



# Define GPIO pin numbers

button_pin = machine.Pin(6, machine.Pin.IN, machine.Pin.PULL_UP)  # Button pin

led_pin = machine.Pin(27, machine.Pin.OUT)  # LED pin

buzzer_pin = machine.Pin(20, machine.Pin.OUT)

switch = 1

flag = 1
```

```
buzzer_pwm = machine.PWM(buzzer_pin)

buzzer_pwm.freq(2000)

buzzer_pwm.duty_u16(32768)


# Function to toggle the LED

def interrupt(pin):

    global flag

    global switch

    if flag == 1:

        if switch ==1:

            switch = 0

        else:

            switch = 1

    flag = 0




# Configure the button pin to trigger the interrupt on a rising edge

button_pin.irq(trigger=machine.Pin.IRQ_RISING, handler=interrupt)


# Main loop

while True:

    if switch == 1:

        buzzer_pwm.duty_u16(0)
        led_pin.toggle()

        utime.sleep_ms(237)

    else:

        led_pin.value(0)

        buzzer_pwm.duty_u16(32768)

        utime.sleep_ms(15)

    utime.sleep_ms(20)

    flag = 1
```