**Dylan Falzone**
**10/11/2023**
**4539-8897**

# EE Design 1 Technical Report
# Microcontroller SPI & DAC

## Introduction

Communication. It's key to surviving and thriving in a society, it's essential to maintaining relationships, and it is the plight of engineers everywhere. It's no secret that engineers have long struggled to use their voices and bodies to produce effective, non-infuriating interactions with other people. Yet, history's engineers were not discouraged by this affliction. The introduction of digital technologies warranted the creation of a new way to exchange information, and engineers jumped at the chance to create a simpler form of communication, free from human nuance. Communication protocols like SPI allow for simple, yet very capable conversation between machines. In this lab, we will use SPI to facilitate communication between our microcontroller and DAC to implement a waveform generator.

## Methods

### Hardware

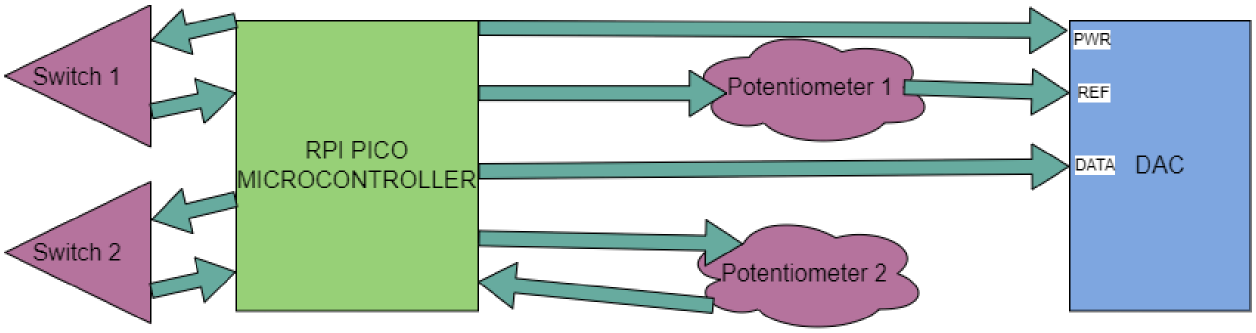


Figure 1: Hardware Block Diagram for the Waveform Generator system.

Since the waveform generator needed to feature an adjustable frequency and amplitude, as well as have 4 different types of waveforms, 4 input methods were used. Potentiometers controlled the frequency and amplitude, while the switches selected between the different waveforms. Potentiometers 1 and 2 featured different wiring, which I will discuss more in a moment. Besides the input methods, the pico and DAC were the only other main components. All data and power transfer between the two was in one direction, from the pico to the DAC.

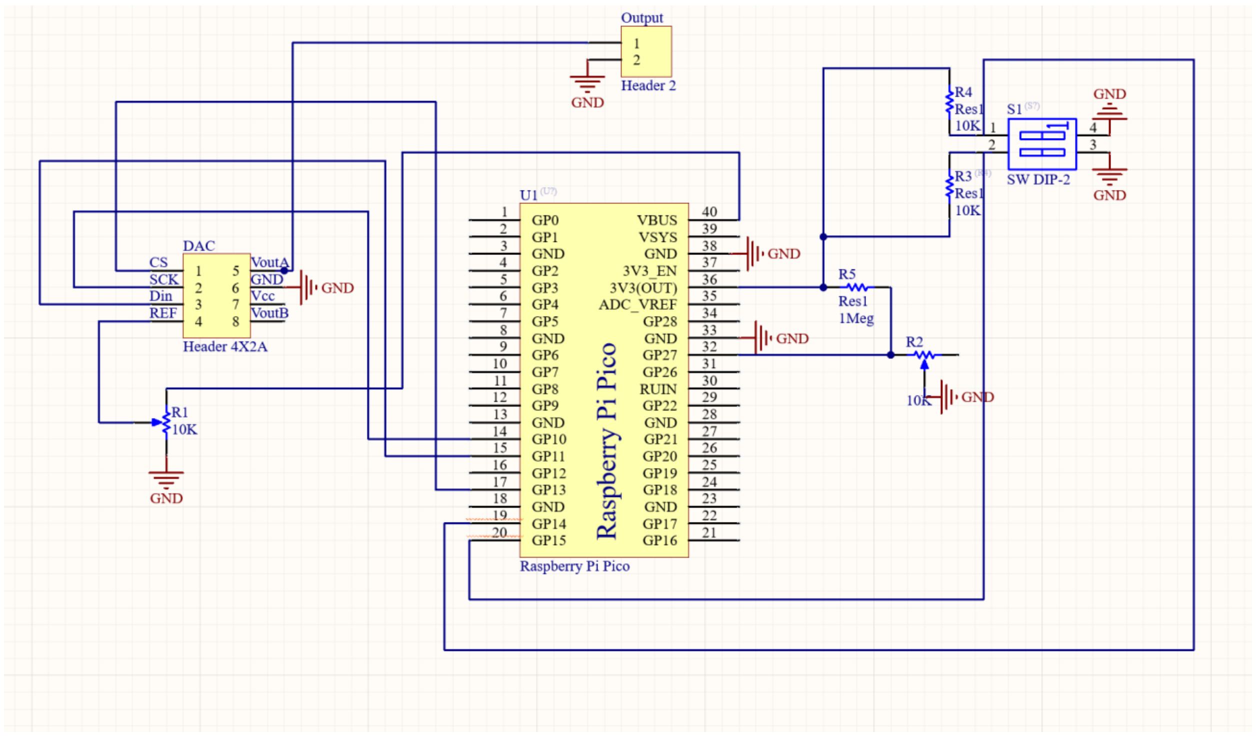


Figure 2: Altium schematic of the Waveform Generator system.

Figure 2 shows the schematic with all the components. The wiring was necessarily messy, due to the large number of input devices required, as well as 7/8 of the DAC's pins needing to be connected.

As mentioned previously, the potentiometers were wired in different configurations. Initially, I had planned to wire each of them the same way, using the ADC to take both of their resistance levels as inputs to the microcontroller, dealing with the amplitude and frequency modulation in code. Thankfully, I discovered that the REF pin on the DAC could instead be manipulated to vary the waveform's amplitude, so I rewired the potentiometer to provide varying voltage levels to that pin, which saved some complexity in the code.

Additionally, for the other potentiometer, I connected it to the ADC in a slightly unorthodox fashion. Using only two of the 3 pins, I installed it like a resistor. Instead of using it's' 3rd pin to output a voltage, I simply used it as a variable R2 for the ADC, similar to the setup of the previous lab.

Finally, for the DAC, I could not find an adequate component in the Altium library to match the number of pins, so I simply used an 4x2header instead, which actually looks a lot like the DAC.

## Software

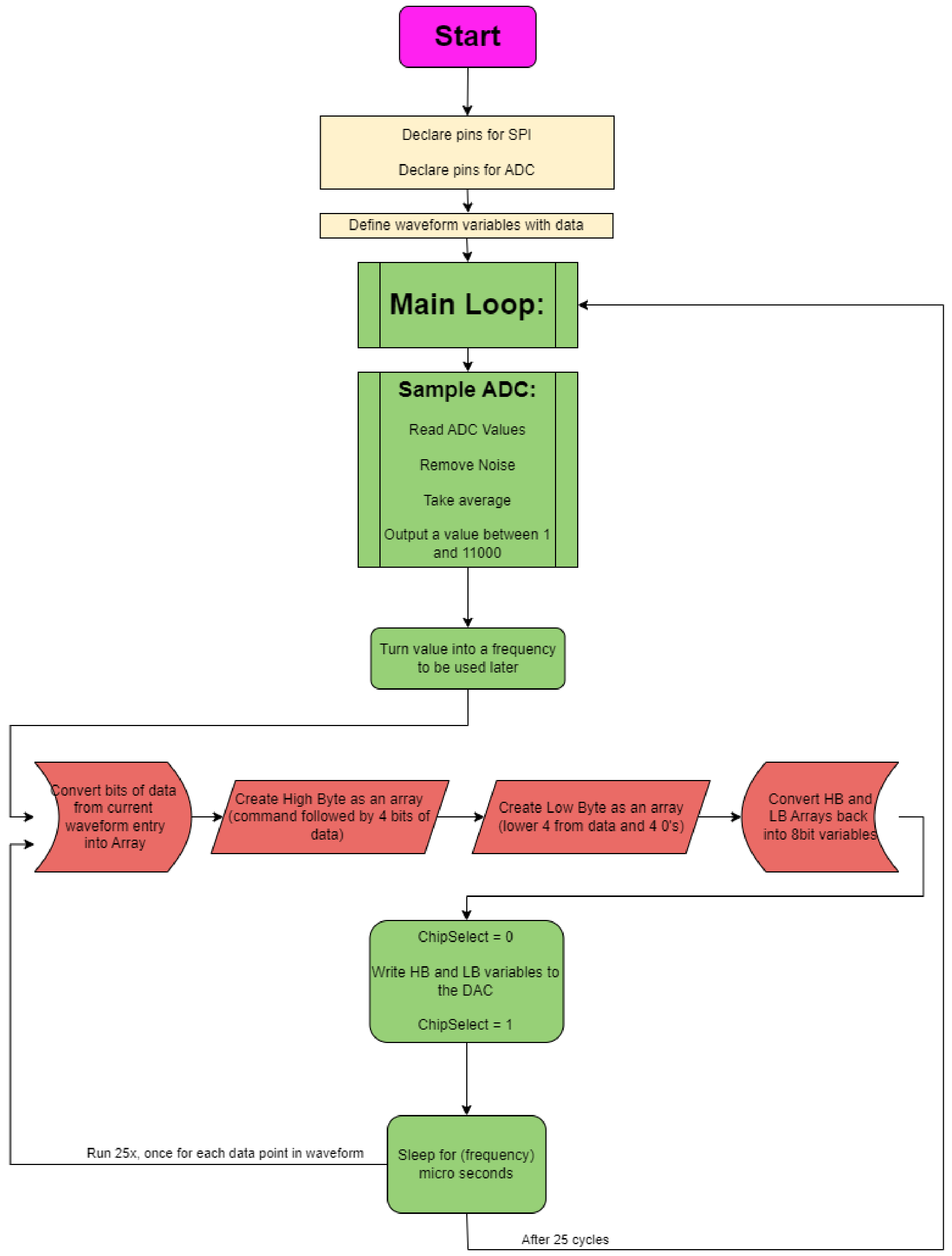

Start
Declare pins for SPI
Declare pins for ADC
Define waveform variables with data
Main Loop:
Sample ADC:
Read ADC Values
Remove Noise
Take average
Output a value between 1 and 11000
Turn value into a frequency to be used later
Convert bits of data from current waveform entry into Array
Create High Byte as an array (command followed by 4 bits of data)
Create Low Byte as an array (lower 4 from data and 4 0's)
Convert HB and LB Arrays back into 8bit variables
ChipSelect = 0
Write HB and LB variables to the DAC
ChipSelect = 1
Sleep for (frequency) micro seconds
Run 25x, once for each data point in waveform
After 25 cycles

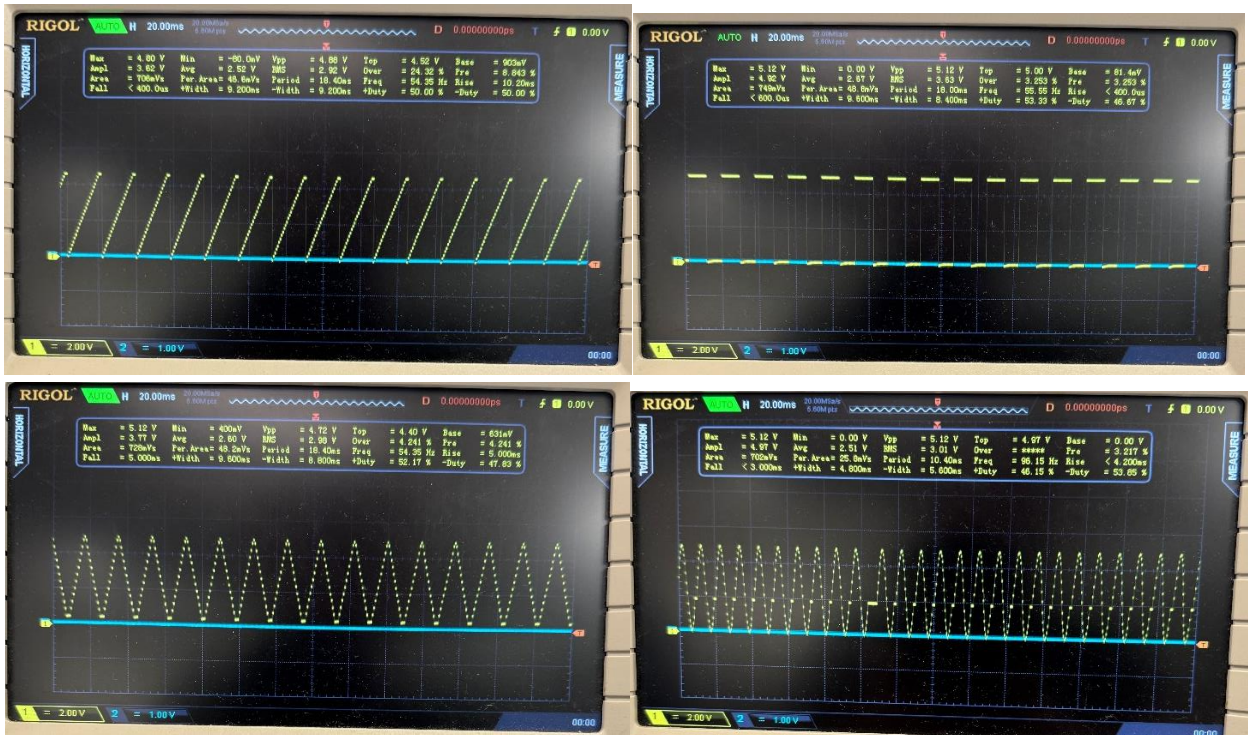
Figure 3: Software Block Diagram for the Waveform Generator system.

On the software side of things, I had some slight trouble. The code was simple enough: store some waveform data in arrays, format and send each value to the DAC, and repeat based on the frequency provided to the ADC. However, it was the 2nd of those objectives that proved quite difficult.

Having little prior experience with python, I could not figure out a way to simply shift the bits in the array data points to get them into the correct format for the DAC. If this were C, it would have been a different story, but after several hours and no progress, I decided to take a different approach. In Figure 3, the red blocks represent my less-than-optimal solution. I converted each data point to a bit-array, where each bit of the data was an entry in the array [1]. So, for example, an 8-bit integer would become an 8-entry array. I then used array operations, something I'm more familiar with than bit operations, to format the data for the DAC. And finally, I converted those formatted arrays back into individual variables to be sent via SPI to the DAC [2].

I was very surprised when I found out that other people did all of this with a single line of code, just shifting the bits in the original variables. My method worked, but it created a frequency related issue that I will discuss in the next section.

## **Results**



```
Max   = 4.80 V    Min       = -80.0mV  Vpp    = 4.88 V    Top   = 4.52 V    Base  = 903mV
Ampl  = 3.62 V    Avg       = 2.52 V   RMS    = 2.92 V    Over  = 24.32 %   Pre   = 8.843 %
Area  = 708mVs    Per.Area= 46.6mVs    Period = 18.40ms   Freq  = 54.35 Hz  Rise  = 10.20ms
Fall  < 400.0us   +Width    = 9.200ms  -Width = 9.200ms   +Duty = 50.00 %   -Duty = 50.00 %
```

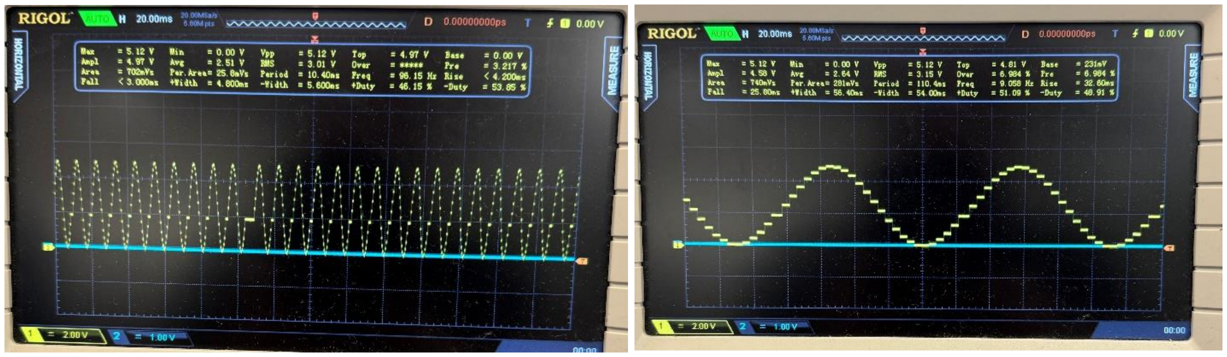
```
Max   = 5.12 V    Min       = 0.00 V   Vpp    = 5.12 V    Top   = 5.00 V    Base  = 81.4mV
Ampl  = 4.92 V    Avg       = 2.67 V   RMS    = 3.63 V    Over  = 3.253 %   Pre   = 3.253 %
Area  = 749mVs    Per.Area= 48.8mVs    Period = 18.00ms   Freq  = 55.55 Hz  Rise  < 400.0us
Fall  < 600.0us   +Width    = 9.600ms  -Width = 8.400ms   +Duty = 53.33 %   -Duty = 46.67 %
```

```
Max   = 5.12 V    Min       = 400mV    Vpp    = 4.72 V    Top   = 4.40 V    Base  = 631mV
Ampl  = 3.77 V    Avg       = 2.60 V   RMS    = 2.98 V    Over  = 4.241 %   Pre   = 4.241 %
Area  = 728mVs    Per.Area= 48.2mVs    Period = 18.40ms   Freq  = 54.35 Hz  Rise  = 5.000ms
Fall  = 5.000ms   +Width    = 9.600ms  -Width = 8.800ms   +Duty = 52.17 %   -Duty = 47.83 %
```

```
Max   = 5.12 V    Min       = 0.00 V   Vpp    = 5.12 V    Top   = 4.97 V    Base  = 0.00 V
Ampl  = 4.97 V    Avg       = 2.51 V   RMS    = 3.01 V    Over  = *****     Pre   = 3.217 %
Area  = 702mVs    Per.Area= 25.8mVs    Period = 10.40ms   Freq  = 96.15 Hz  Rise  < 4.200ms
Fall  < 3.000ms   +Width    = 4.800ms  -Width = 5.600ms   +Duty = 46.15 %   -Duty = 53.85 %
```

Figure 4: Sawtooth, square, triangle and sine waves at varying frequencies.

The resultant waveforms shown in Figure 4 were all solid. Each type worked as intended and looked correct. The data for the sine and triangle waves came from an online LUT generator, while the sawtooth and square were made by me. Pictured, all waves are at an arbitrarily chosen 54hz, while the sine wave is at 100hz. Though I do want to make it clear, all wave types were capable of hitting 100hz.



```
Max   = 5.12 V    Min       = 0.00 V   Vpp    = 5.12 V    Top   = 4.97 V    Base  = 0.00 V
Ampl  = 4.97 V    Avg       = 2.51 V   RMS    = 3.01 V    Over  = *****     Pre   = 3.217 %
Area  = 702mVs    Per.Area= 25.8mVs    Period = 10.40ms   Freq  = 96.15 Hz  Rise  < 4.200ms
Fall  < 3.000ms   +Width    = 4.800ms  -Width = 5.600ms   +Duty = 46.15 %   -Duty = 53.85 %
```

```
Max   = 5.12 V    Min       = 0.00 V   Vpp    = 5.12 V    Top   = 4.81 V    Base  = 231mV
Ampl  = 4.58 V    Avg       = 2.64 V   RMS    = 3.15 V    Over  = 6.984 %   Pre   = 6.984 %
Area  = 740mVs    Per.Area= 281mVs     Period = 110.4ms   Freq  = 9.058 Hz  Rise  = 32.60ms
Fall  = 25.80ms   +Width    = 56.40ms  -Width = 54.00ms   +Duty = 51.09 %   -Duty = 48.91 %
```

Figure 5: Sine wave at 100hz and 10hz.

Figure 5 shows that a range of 10-100hz was indeed possible with my setup. However, it was not easy or simple. After optimizing my code as much as possible, I was barely breaking 50hz. This was because my previously mentioned method for formatting the data to send to the DAC was incredibly slow compared to the bit shifting method. As a result, without a vast restructuring, my code could not make it to 100hz. So, due to the rapidly approaching demo deadline, I overclocked my pico from 120Mhz to 270Mhz. This gave just enough of a boost to get the circuit the rest of the way from 50 to 100hz. And while it worked this time, I do not plan on relying on overclocking in the future, as it can potentially harm the microcontroller in certain circumstances.

## **Conclusion**

As engineers, we may not be able to master the art of communication entirely. But we can get pretty good at SPI. In this lab, we did just that. Despite our formatting issues, we successfully used SPI to communicate between a microcontroller and DAC to create a powerful and versatile waveform generator. And while it might not have been the best practice, we did learn how to overclock a microcontroller which may prove useful if any future emergencies arise.

## ***Appendix***

```
import machine

import math

import utime


def sleep_microseconds(microseconds):

    start_time = utime.ticks_us()

    while utime.ticks_diff(utime.ticks_cpu(), start_time) < microseconds:

        pass


adc_pin0 = machine.ADC(28) #GND

adc_pin1 = machine.ADC(27) #Freq

pin_14 = machine.Pin(14, machine.Pin.IN)#sw1

pin_15 = machine.Pin(15, machine.Pin.IN)#sw2

Vref = 3.3
```

```
spi = machine.SPI(1, baudrate=1000000, polarity=0, phase=0, sck=machine.Pin(10), mosi=machine.Pin(11))
cs = machine.Pin(13, mode=machine.Pin.OUT, value=1)
ref = machine.Pin(12, mode=machine.Pin.OUT, value=1)

sine = [0x80,0x9f,0xbd,0xd7,0xeb,0xf9,0xff,0xfd,
        0xf3,0xe2,0xca,0xae,0x8f,0x70,0x51,0x35,
        0x1d,0xc,0x2,0x0,0x6,0x14,0x28,0x42,
        0x60,0x80]
square = [0,0,0,0,0,0,0,0,0,0,0,0,255,255,255,255,
        255,255,255,255,255,255,255,255,255]
triangle = [0x14,0x29,0x3d,0x52,0x66,0x7a,0x8f,0xa3,
        0xb8,0xcc,0xe0,0xf5,0xff,0xf5,0xe0,0xcc,
        0xb8,0xa3,0x8f,0x7a,0x66,0x52,0x3d,0x29,
        0x14,0x0]
sawtooth = [0,10,20,30,40,50,60,70,80,90,100,110,120,
        130,140,150,160,170,180,190,200,210,220,230,
        240,250]

while True:
    #Get desired pre_freq value from ADC
    i=0
    values1 = [None] * 10
    for i in range(10):
        adc_value0 = adc_pin0.read_u16()
        adc_value1 = adc_pin1.read_u16()

        voltage1 = (adc_value1 / 65535.0) * Vref
        ground = (adc_value0 / 65535.0) * Vref
        final_voltage1 = (voltage1-ground)
        if final_voltage1 <= 0:
```

```
        final_voltage1 = 0.000805676
    known_resistor_value = 1000000
    resistance_pre1 = (known_resistor_value)/((Vref/final_voltage1)-1)
    values1[i] = resistance_pre1

values1.sort()
total1 = sum(values1)
num_elements = 10
resistance1 = total1/ num_elements
print(resistance1)

if 1<=resistance1<1600:
    timer =1
elif 1600<=resistance1<3200:
    timer =6
elif 3200<=resistance1<4800:
    timer = 10
elif 4800<=resistance1<6400:
    timer =14
elif 6400<=resistance1<8000:
    timer =18
elif 8000<=resistance1<9600:
    timer =22
elif 9600<=resistance1<11200:
    timer =26
elif 11200<=resistance1<12800:
    timer =31
elif 12800<=resistance1<14400:
    timer =35
else:
    timer =40
print(timer)



#Convert desired array, output it to DAC
z=0
for z in range(25):

    state_14 = pin_14.value()
    state_15 = pin_15.value()

    if state_14 == 1:
        if state_15 ==1:
            #case 11
            value = sawtooth[z]
        else:
            #case 10
            value = triangle[z]
    else:
        if state_15 ==1:
            #case 01
            value = square[z]
        else:
            #case 00
            value = sine[z]




    bit_array = []
    # byte -> array
```

```
i=7
for i in range(7, -1, -1):
    bit = (value >> i) & 1
    bit_array.append(bit)

i=3
j=7
first_value = [0] * 8
first_value = [1,0,0,1,0,0,0,0]
for i in range(3, -1, -1):
    if j > 3:
        first_value[j] = bit_array[i]
    j=j-1

i=3
j=7
second_value = [0] * 8
for i in range(3, -1, -1):
    if j > 3:
        second_value[i] = bit_array[j]
    j=j-1

byte_value1 = 0
# array -> byte
for bit in first_value:
    byte_value1 = (byte_value1 << 1) | bit
byte_value2 = 0
# array -> byte
for bit in second_value:
    byte_value2 = (byte_value2 << 1) | bit
```

```
cs(0)
spi.write(bytes([byte_value1,byte_value2]))
cs(1)



if timer == 1:
    sleep_microseconds(1)
elif timer == 6:
    sleep_microseconds(50)
elif timer == 10:
    sleep_microseconds(150)
elif timer == 14:
    sleep_microseconds(300)
elif timer == 18:
    sleep_microseconds(500)
elif timer == 22:
    sleep_microseconds(500)
elif timer == 26:
    sleep_microseconds(700)
elif timer == 31:
    sleep_microseconds(1200)
else:
    sleep_microseconds(timer*100)
#utime.sleep_ms(timer)
```

## ***References***

[1] P. Sao, “Exploring bitarray in python with list of functions available,” Python Pool, https://www.pythonpool.com/python-bitarray/ (accessed Oct. 11, 2023).

[2] R. Sheldon, “What is serial peripheral interface (SPI)?: Definition from TechTarget,” WhatIs.com, https://www.techtarget.com/whatis/definition/serial-peripheral-interface-SPI (accessed Oct. 11, 2023).