

---

# **Analog I/O**

**ECE 476 Advanced Embedded Systems**

**Jake Glower - Lecture #9**

Please visit [Bison Academy](#) for corresponding  
lecture notes, homework sets, and solutions

---

## Introduction:

Up to now, we have been dealing with binary inputs and outputs (I/O).

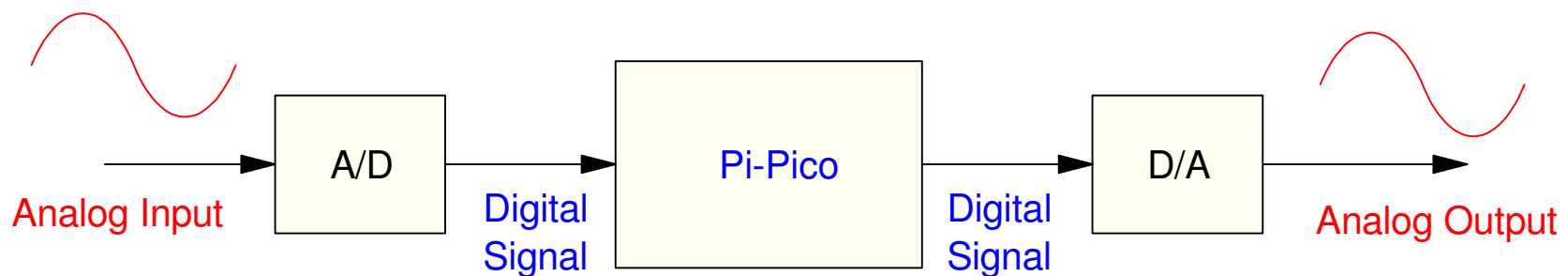
The real world is typically analog, however.

- Voltages can take on any value - not just 0V and 3.3V
- Resistance's can take on any positive value
- Temperatures can be any value

Microcontrollers like the Pi-Pico usually allow analog I/O

- A/D: Read an analog input
- D/A: Drive an analog output

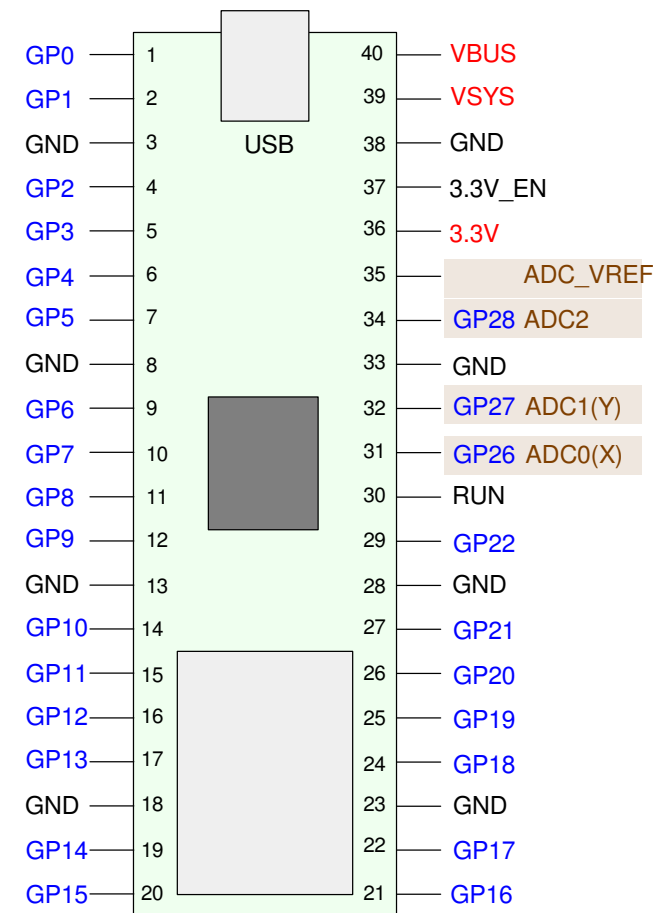
This lecture looks at how to input and output analog signals from a Pi-Pico.



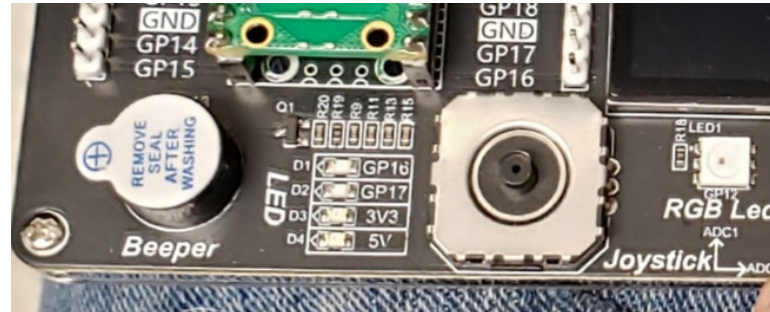
# Analog Inputs: A/D

The Pi-Pico has five 12-bit A/D ports built in  
- with three connected to I/O pins.

- ADC0: joystick x position
- ADC1: joystick y position
- ADC2: voltage on GP28
- ADC3: is not connected, and
- ADC4: the temperature of the Pi-Pico



## Reading the Joystick Position



**x = ADC0 (pin 26)**

$y = \text{ADC1}$  (pin 27)

- $0V = 0$
- $3.3V = 65,535$

```
from machine import ADC
from time import sleep_ms
```

```
a2d0 = ADC(0)
a2d1 = ADC(1)
```

```
while(1):
    x = a2d0.read_u16()
    y = a2d1.read_u16()

    print(x, y)
    sleep_ms(200)
```

---

## Left vs. Right Justified

The Pico has a 12-bit A/D

- 0 to 4095 if right justified
- 0 to 65,535 if left justified

Left justified allows the code to be independent of the A/D resolution

- Max reading is 65,536 regardless of the number of bits
- Makes the code more transportable
- Kind of becoming the standard

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
12-bit A/D reading (0x000 to 0xFFF)												0	0	0	0

---

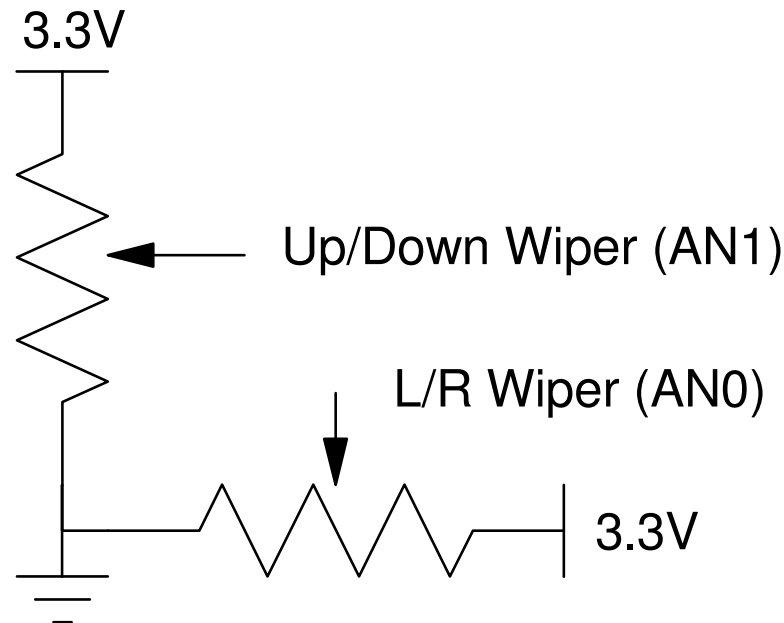
## Reading 0V to 3.3V (Joystick Input)

The joystick consists of two potentiometers

- Endpoint = 0V and 3.3V
- Wiper = A/D input

As you move the joystick, the voltage goes from 0V to 3.3V

- A/D reading is 0 to 65,535

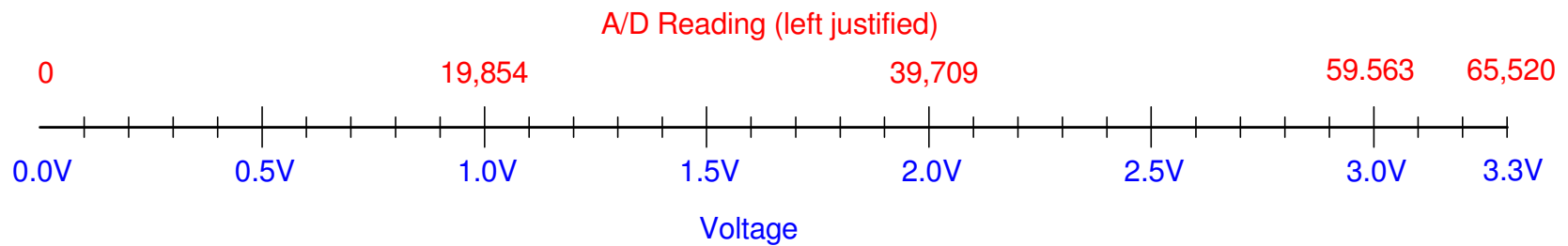


---

## Converting A/D Reading to Voltage

The A/D reading is proportional to voltage

- $0V = 0$
- $3.3V = 65,535$



Analog Inputs: A/D reading is proportional to voltage

---

---

## Reading Voltage (cont'd)

Voltage is proportional to the A/D reading

$$V_i = \left( \frac{3.30V}{65,520} \right) \cdot A2D_i$$

Sensitivity is one count (12-bits = 4095)

$$dV = \left( \frac{3.3V}{4095} \right) = 806.86\mu V$$

A/D channels 0-2 read voltage

- Connected to GPIO pins

A/D channel 4 measures the temperature of the Pi-Pico

$$^{\circ}C = 0.02927 * (14940 - A2D_4)$$

resolution = 0.468 degrees C

---



---

## Code

Read the voltage on

- Joystick X (ADC0)
- Joystick Y (ADC1)

Read the temperature of the Pi-Pico

- ADC4

Results:

- $V_x = 1.447679V$
- $V_y = 1.499254V$
- $temp = 23.21111C$

```
from machine import ADC
from time import sleep_ms

a2d0 = ADC(0)
a2d1 = ADC(1)
a2d4 = ADC(4)

k = 3.3 / 65535

while(1):
    a0 = a2d0.read_u16()
    a1 = a2d1.read_u16()
    a4 = a2d4.read_u16()

    V0 = a0 * k
    V1 = a1 * k
    Temp = 0.02927*(14940 - a4)

    print(V0, V1, Temp)
    sleep_ms(200)
```

shell

Vx	Vy	degrees C
1.447679	1.499254	23.21111
1.435554	1.501671	23.21111

---

## Reading 0-10V:

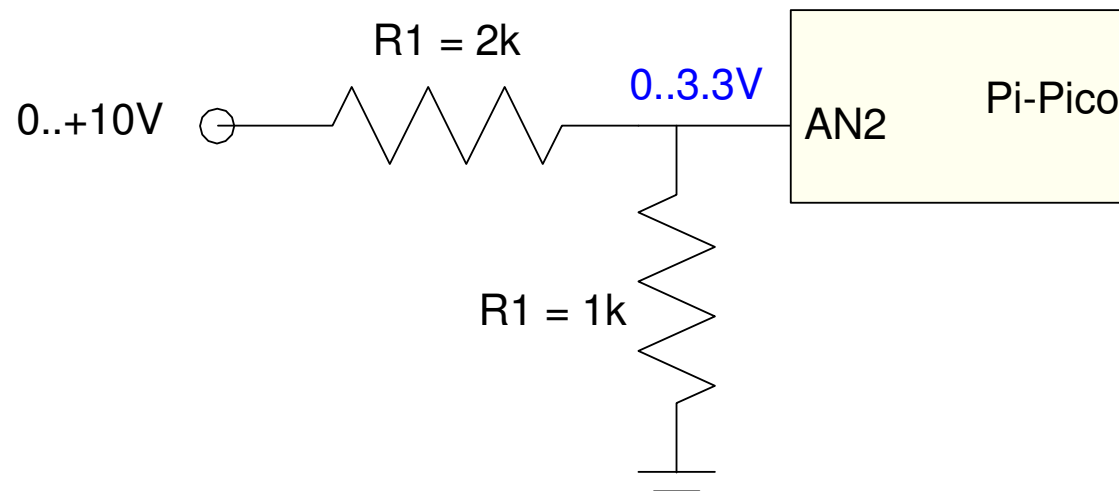
The Pi-Pico always reads 0V to 3.3V

To read 0..10V

- Convert 0-10V to 0-3.3V
- Voltage divider with a gain of 0.33

If  $R_1 = 1k$ , then

$$R_2 = \left( \frac{1-0.33}{0.33} \right) R_1 = 2.03k \approx 2k$$



---

## Code for Reading 0..10V

- Scale the computed voltage accordingly:

```
from machine import ADC
from time import sleep_ms

a2d2 = ADC(2)

k = 10.0 / 65535

while(1):
    a2 = a2d2.read_u16()
    V2 = a2 * k

    print(V2, ' Volts')
    sleep_ms(200)
```

shell

```
4.9932 Volts
5.0221 Volts
```

---

---

## Reading -10V to +10V:

To convert -10V to +10V to 0 to 3.3V, use three resistors

- Other solutions exist

If

- x: -10V to +10V
- B: +3.3V
- C: 0V

the output should be

$$y = (x + 10) \left( \frac{3.3V}{20V} \right) = 0.1650x + 1.650$$

You can set this up as a weighted average of {A, B, C}

$$y = (0.1650x + 0.500B)$$

Add a term so the coefficients add to 1.000

$$y = 0.1650x + 0.500B + 0.335C$$

---

---

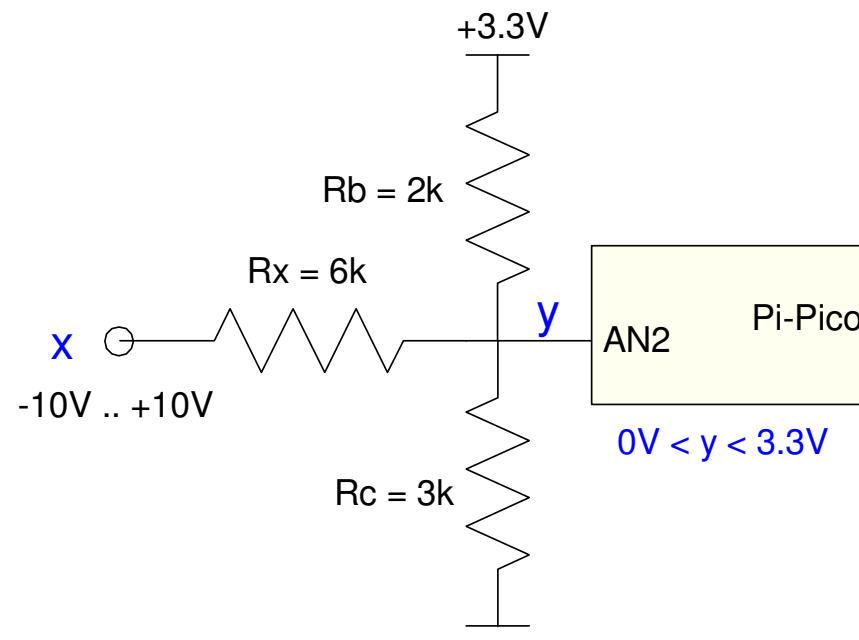
Pick your favorite resistor value, such as 1k.

The weightings tell you how the resistors are scaled:

$$R_x = \frac{1k}{0.1650} = 6.06k$$

$$R_b = \frac{1k}{0.500} = 2k$$

$$R_c = \frac{1k}{0.335} = 2.98k$$



---

The corresponding Python code would be:

- -10V reads as 0
- +10V reads as 65,535

```
from machine import ADC
from time import sleep_ms

a2d2 = ADC(2)

k = 20.0 / 65535

while(1):
    a2 = a2d2.read_u16()
    V2 = k * a2 - 10

    print(V2, ' Volts')
    sleep_ms(200)
```

**shell**

```
0.12131 Volts
0.01231 Volts
```

---

---

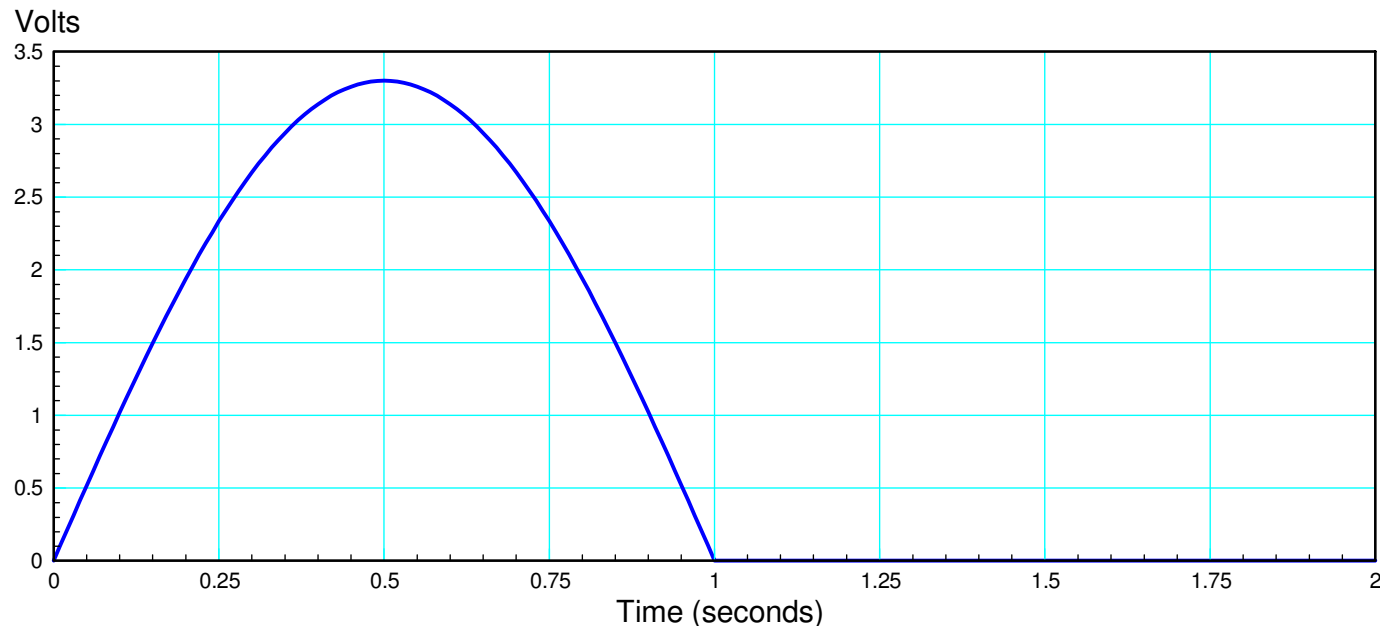
## Analog Outputs:

Analog outputs can assume multiple values between 0V and 3.3V

- In theory, infinite values
- In practice, a finite but large number of values

Example: output a half-rectified sine wave with a period of 2 seconds:

- Using PWM, PWM + Filter, D/A



---

## PWM (pulse width modulation)

From before, the *machine* library has a PWM function

This allows you to output

- A square wave,
- At a given frequency,
- With a given duty cycle.

For example, the following code outputs a

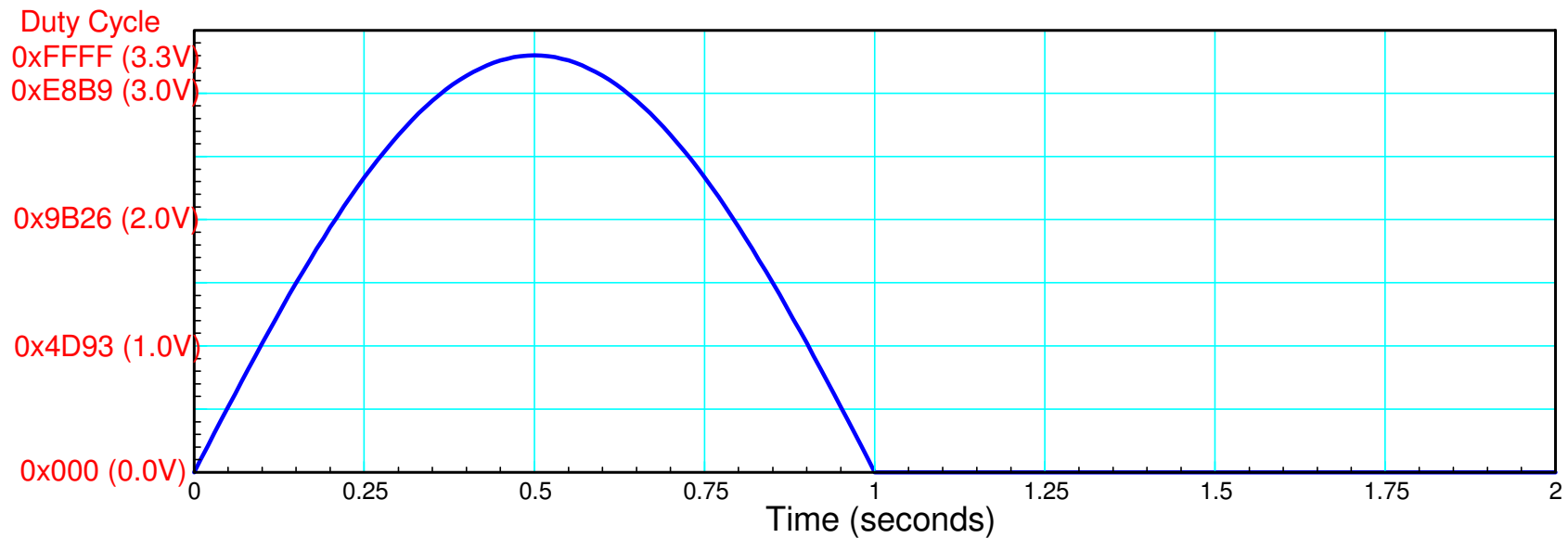
- 1kHz square wave (line 5)
- with a frequency of 1kHz (line 6: 10% of 65,535 = 6553)

```
1  from machine import Pin, PWM
2
3  Aout = Pin(16, Pin.OUT)
4  Aout = PWM(Pin(16))
5  Aout.freq(1000)
6  Aout.duty_u16(6553)
7  while(1):
8      pass
```



---

If you adjust the duty cycle, you can vary the average from 0V (0%) to 3.3V (100%). To output a half-rectified sine wave, vary the duty cycle according to the figure below:



By varying the duty cycle, you can output any voltage from 0.0V to 3.3V

---

---

## Code:

To speed up execution

- A look-up table is used
- $\sin()$  is computed prior to the main loop

```
from machine import Pin, PWM
from time import sleep_ms
from math import sin, pi

Aout = Pin(16, Pin.OUT)
Aout = PWM(Pin(16))
Aout.freq(1000)

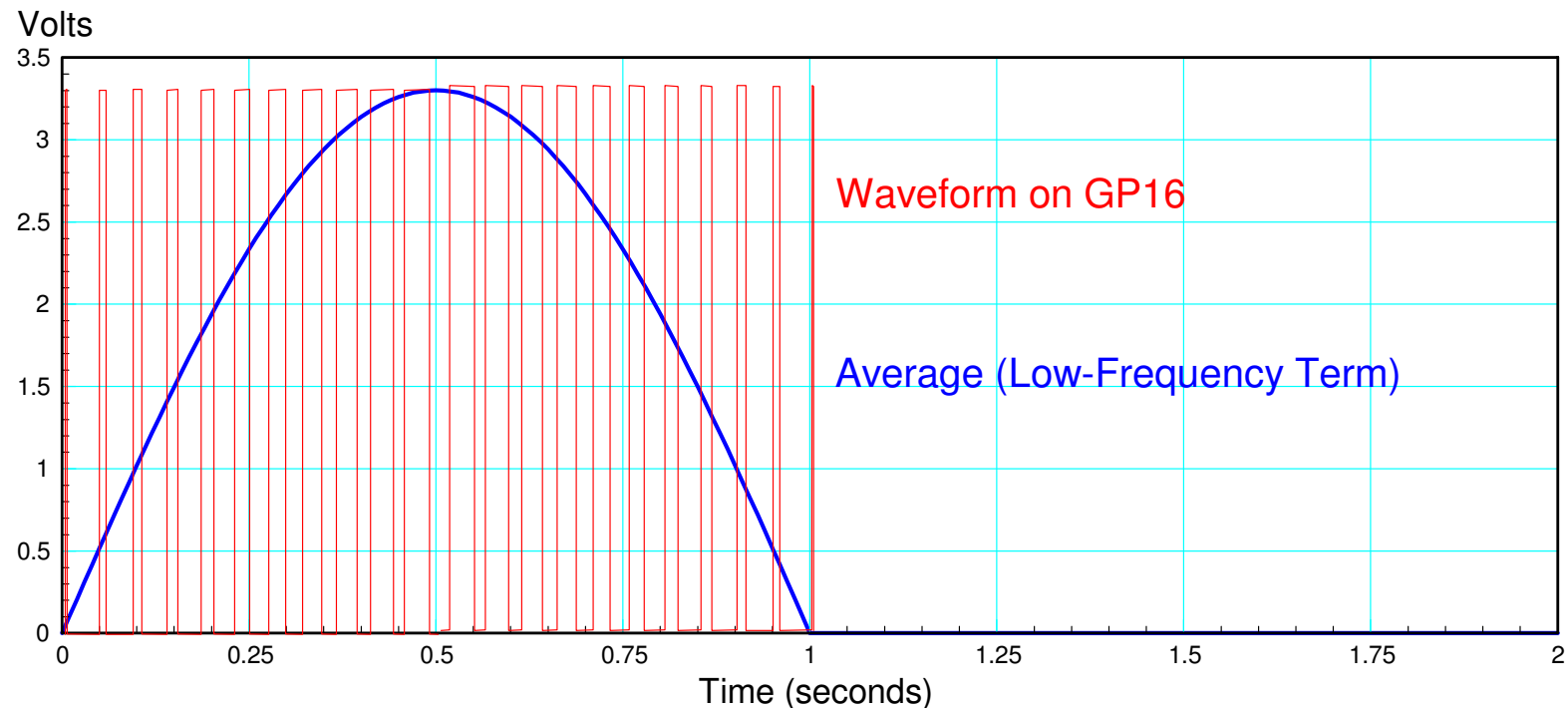
Table = []
for i in range(0,100):
    Table.append(int(65535*sin(i*pi/100)))
for i in range(0,100):
    Table.append(0)

i = 0
while(1):
    i = (i + 1) % 200
    Aout.duty_u16(Table[i])
    sleep_ms(10)
```

---

---

If you look at an LED attached to GP16, the LED will be fading in and out as desired. If you look on an oscilloscope, however, you'll see noise:

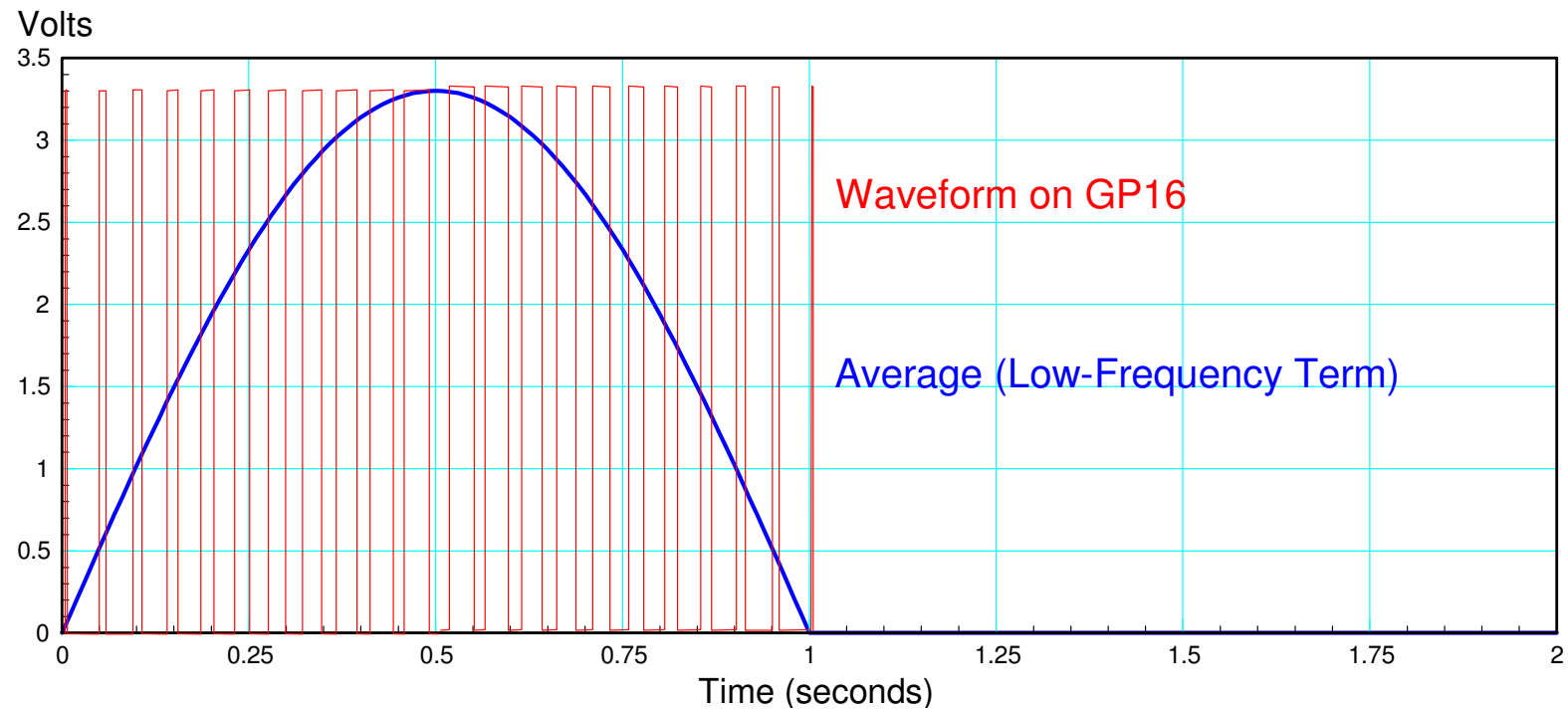


---

# PWM & Frequency Content

The PWM signal contains

- The signal (blue line) at 1Hz, and
- Harmonics of the 1kHz PWM waveform (red line)

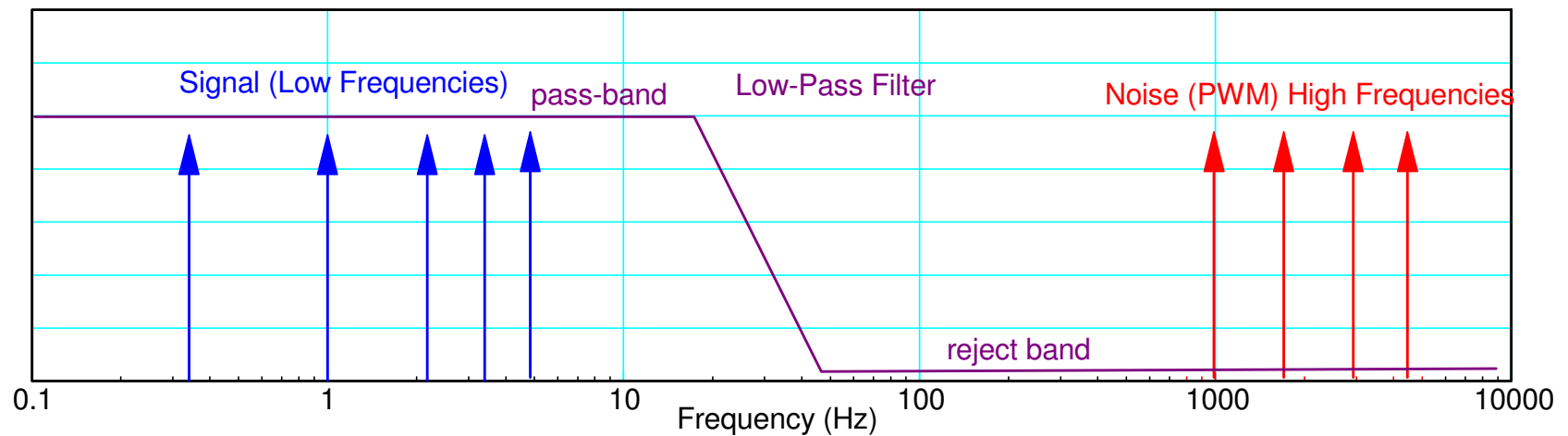


---

To output a clean signal, you want to

- Pass the low frequency terms (frequencies below something like 10Hz), and
- Reject frequencies above 1kHz.

In short, you need to add a low-pass filter.



Add a low-pass filter to pass the signal (frequencies below 10Hz) and reject noise (frequencies above 1kHz)

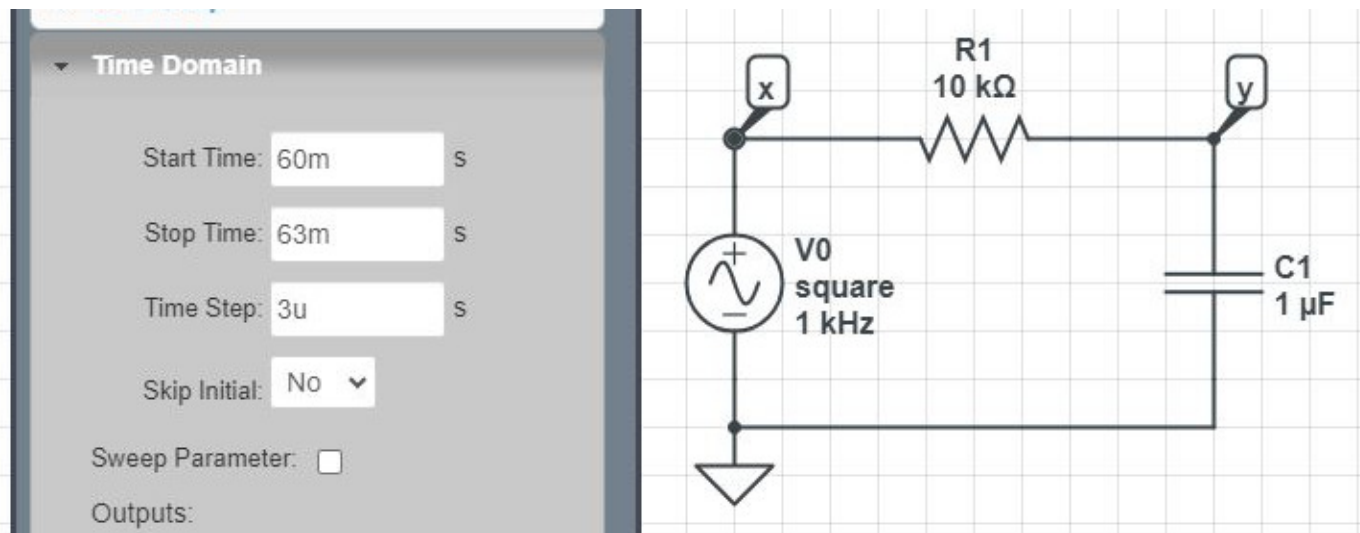
---

---

There are many types of low-pass filters. Two of these are

- A single-stage RC low-pass filter, and
- An active 2nd-order low-pass filter

**RC Low-Pass Filter:** RC filters are simple and not very good. It works OK in this application since there is a large separation between the signal (<10Hz) and the noise (PWM at 1kHz).



---

Typically, you place the corner frequency ( $1/RC$ ) in-between the pass-band (10Hz) and the reject band (1kHz). Assume for convenience you pick 100 rad/sec or 15.9Hz:

$$\frac{1}{RC} = 100 \frac{rad}{sec} = 15.9 Hz$$

The gain of the filter is then

$$y = \left( \frac{\frac{1}{RC}}{s + \frac{1}{RC}} \right) x = \left( \frac{100}{s + 100} \right) x$$

What this filter does is

- For frequencies below 100 rad/sec (15.9Hz), the gain is approximately one.
- For frequencies above 100 rad/sec, the gain drops off as  $1/s$ .
- At 1kHz (6280 rad/sec), the gain is 0.016: noise at 1kHz is attenuated by a factor of 0.016

---

---

Going back to the 1/2-wave rectified sine wave. What you expect the output of the filter to be is:

- A DC term (unchanged - the DC gain is one), and
- An AC term which has been attenuated.

You can approximate the amplitude of the AC term (i.e. the noise on the signal) as

$$Output = Gain \cdot Input$$

At 1kHz, the PWM signal is a 3.3Vpp square wave.

At 1kHz (6280 rad/sec), the gain of the filter is

$$gain = \left( \frac{100}{s+100} \right)_{s=j6280}$$

---



---

Putting it together, the AC (noise) component of the filter's output should be:

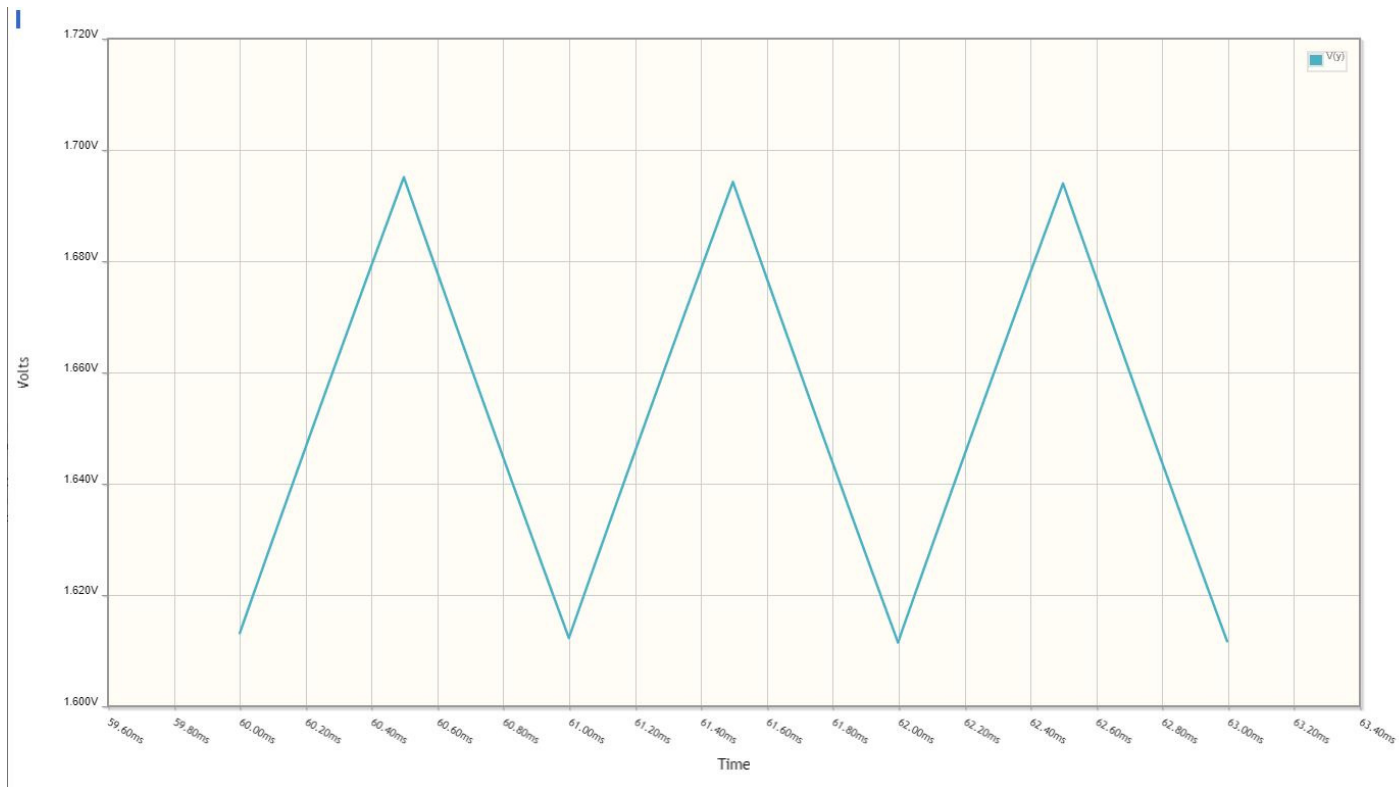
$$y = \left( \frac{100}{s+100} \right) x$$

$$|y| \approx \left( \frac{100}{s+1000} \right)_{s=j6280} \cdot 3.3V_{pp}$$

$$|y| \approx 0.0525V_{pp}$$

The signal at y should have the DC term plus a 52.5mVpp ripple.

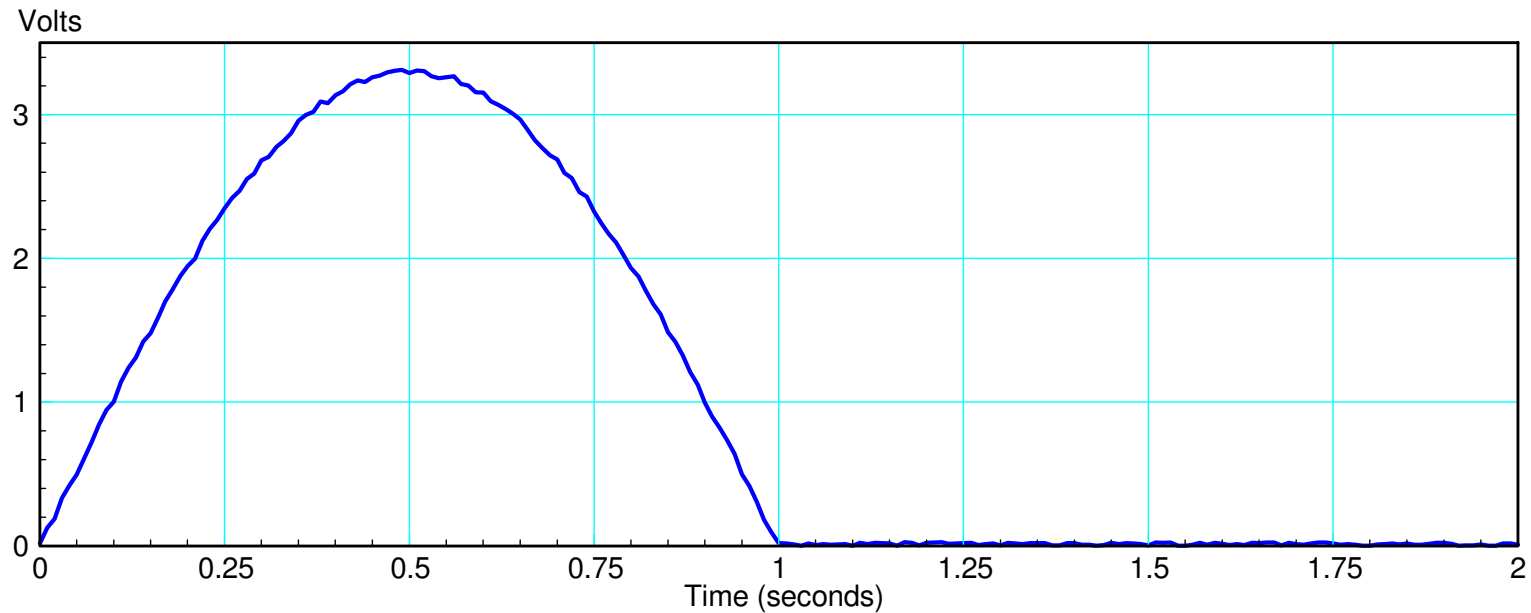
You can check this in CircuitLab. Running a time-domain simulation results in the ripple actually being 80mVpp



Time-Domain Simulation: The RC filter passes the DC term (average voltage is 1.65V)  
The ripple has been reduced from 3.3Vpp to 80mVpp

---

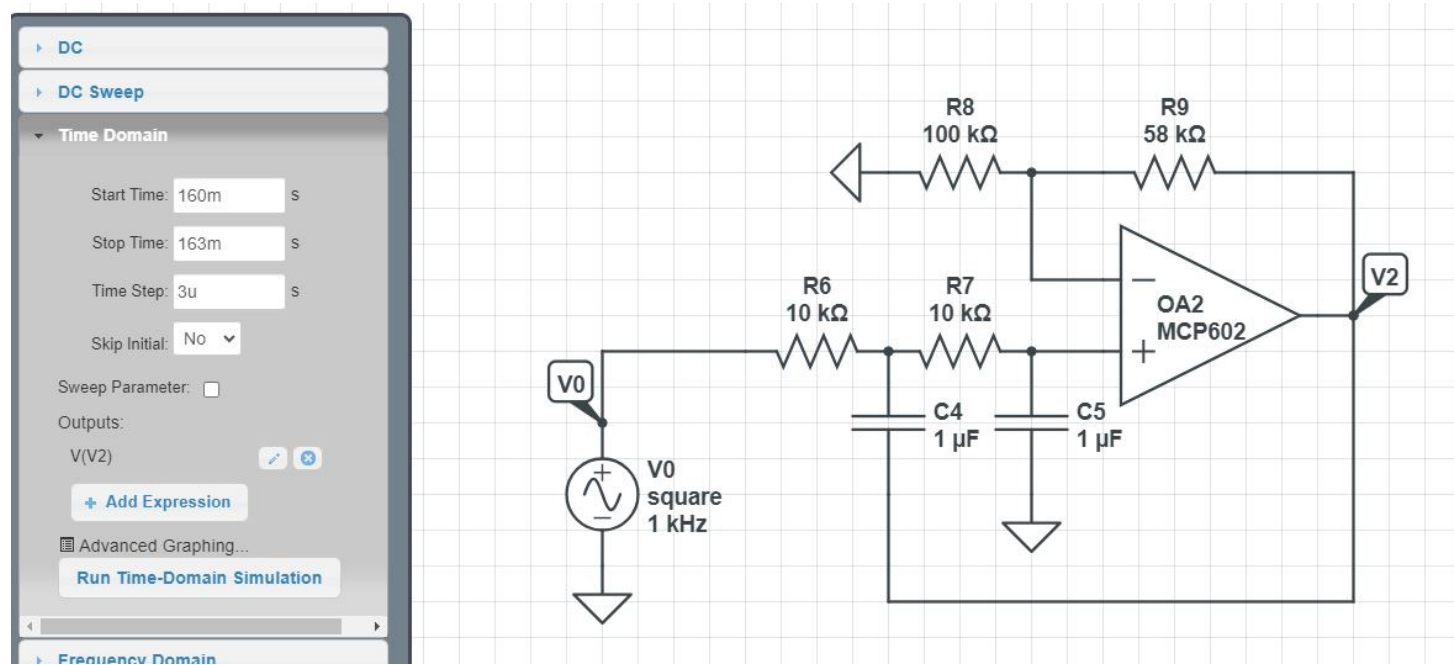
If you apply the RC filter to the output of the PWM signal, the output should look like this:



Output of the PWM & RC Low-Pass Filter.  
The noise on the signal is due the PWM signals at 1kHz getting through the filter.

---

**Active 2nd-Order Low-Pass Filter:** A better filter would be a 2nd-order Butterworth low-pass filter:



2nd-order Butterworth low-pass filter with corner at 100 rad/sec V0 represents the PWM output of a Pi-Pico

---

The magnitude of the poles (aka the corner frequency) is set by  $R_5 * C_4$

$$\frac{1}{R_5 C_4} = 100$$

The angle of the poles is set by  $R_8$  and  $R_9$

$$k = 1 + \frac{R_9}{R_8}$$

$$3 - k = 2 \cos \theta$$

The above circuit gives poles at 100 with an angle of 45 degrees (a 2nd-order Butterworth low-pass filter).

---

The gain of this filter is

$$y = \left( \frac{100^2 \cdot k}{(s+100\angle 45^\circ)(s+100\angle -45^\circ)} \right) x$$

or

$$y = k \left( \frac{100^2}{s^2 + 141s + 100^2} \right) x$$

The DC gain is 1.58 (k), meaning the DC term at x (0..3.3V) will be (0..5.21V) at y.

---

---

The ripple at x is  $3.3V_{pp}$

The ripple at y will be approximately

$$y \approx k\left(\frac{100^2}{s^2+141s+100^2}\right)_{s=j6280} \cdot 3.3V_{pp}$$

$$|y| \approx 0.0013$$

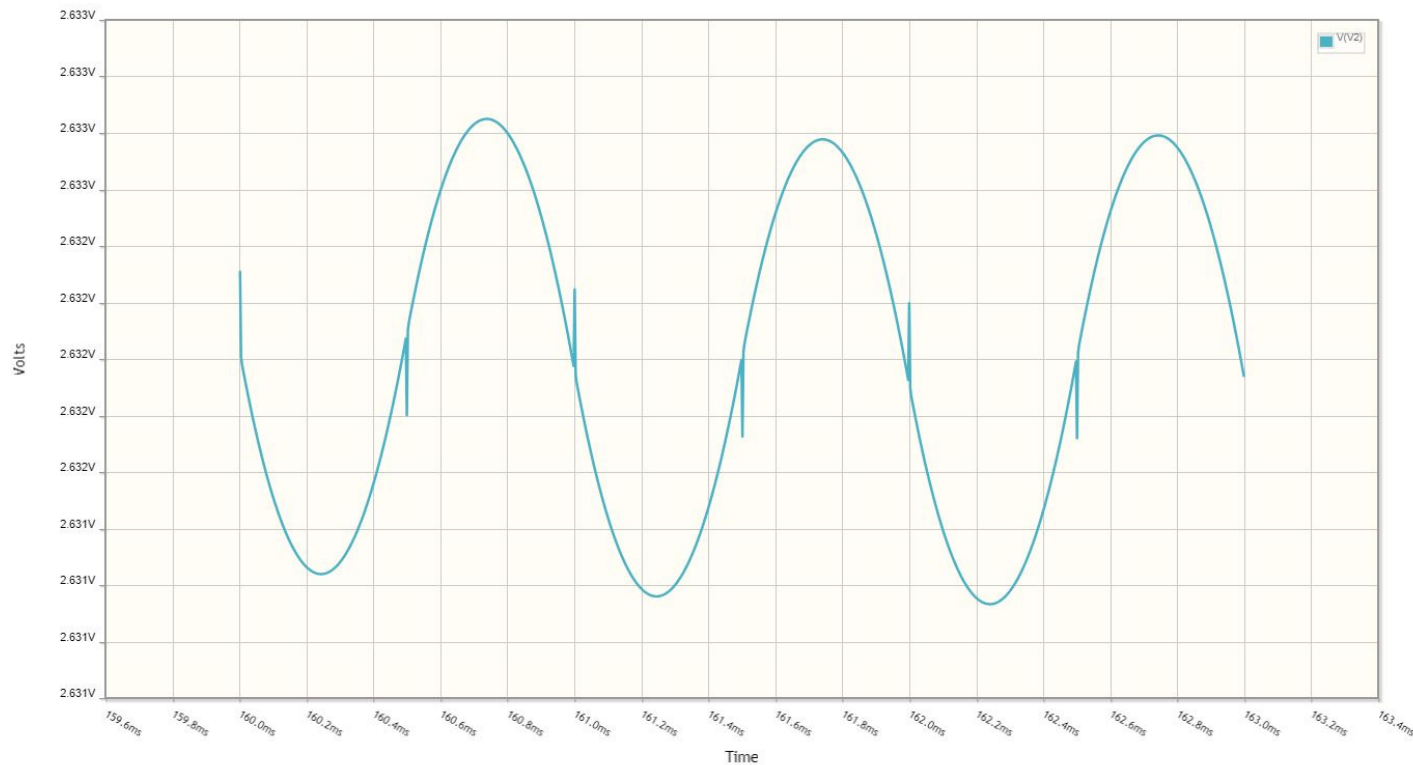
Meaning  $y(t)$  should have

- A DC term of  $2.61V$  ( $1.58 * 50\%$  of  $3.3V$ )
- An AC term of  $1.3mV_{pp}$

---

In CircuitLab, you can check the actual result is

- DC term = 2.632V (as expected)
- AC term = 1.6mVpp (slightly larger than calculated)



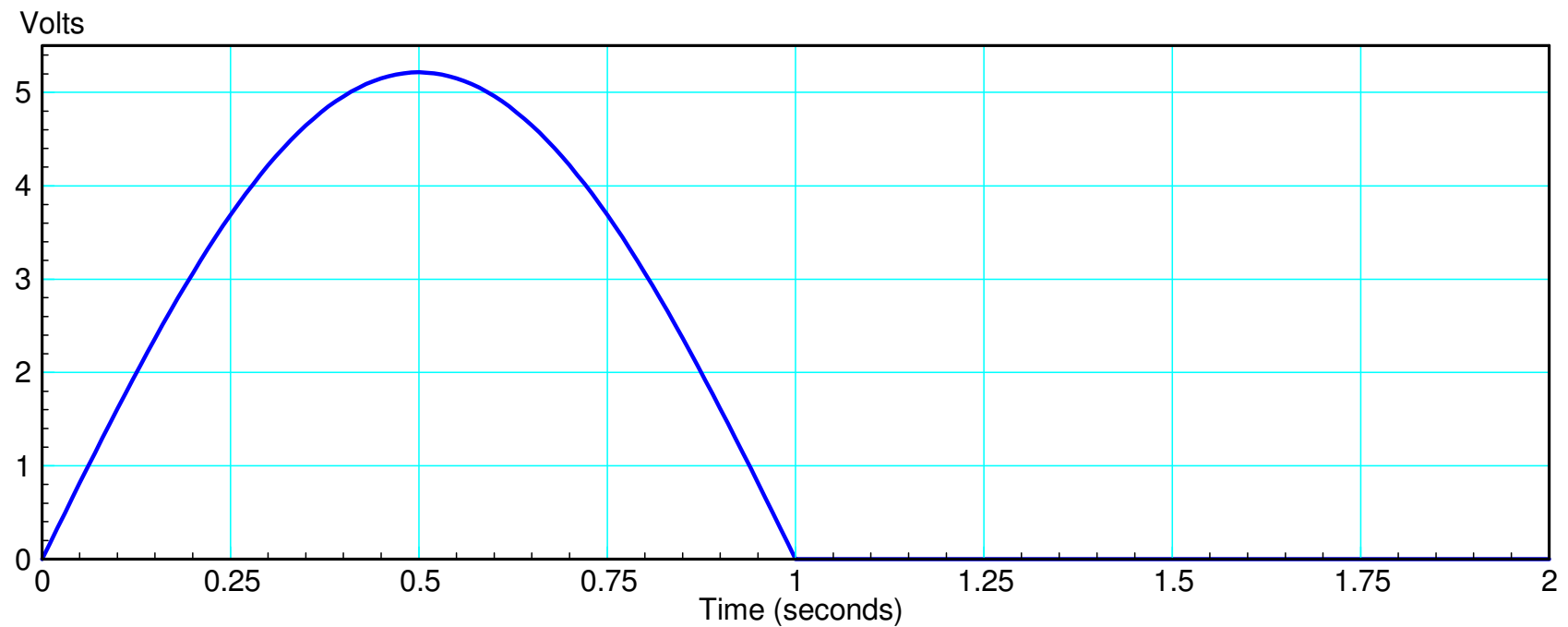
Output of the Butterworth Low-Pass Filter: A DC term with 1.6mVpp ripple

---



---

The output of the PWM signal plus filter looks much smoother than the RC filter.



Output of PWM signal & a 2nd-order Active Low-Pass Filter  
Less noise at 1kHz gets through the filter, producing a smoother output

---

---

## D/A Converter: MCP4921

D/A: Digital to Analog converter

- Outputs 0V to V<sub>dd</sub> analog
  - $2.7V < V_{dd} < 5.5V$
- 12-bit D/A (meaning 4096 steps)
- SPI data interface
- \$2.81 each (Digikey as of June 7, 2024)

Like the A/D, the D/A's voltage is proportional to the number written to it:

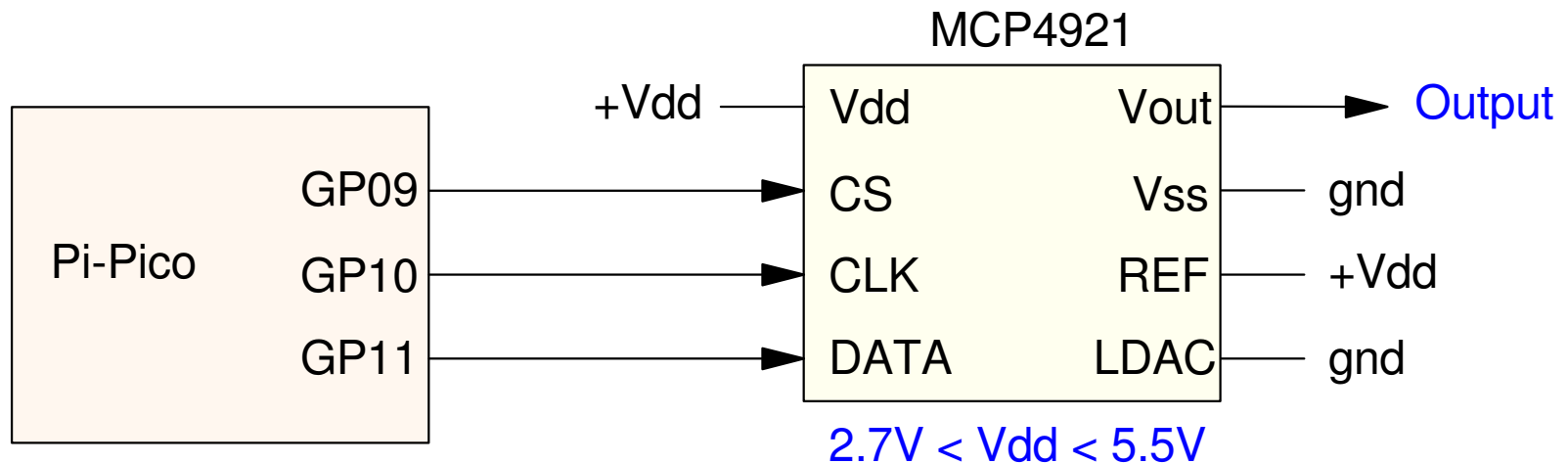
$$D/A\_Out = \left( \frac{\#}{4095} \right) \cdot V_{dd}$$

V<sub>dd</sub> can be anything from 2.7V to 5.5V

- 5.5V doesn't hurt a Pi-Pico
- No data is sent back to the Pi-Pico

---

The connections for a MCP4921 to a Pi-Pico are:



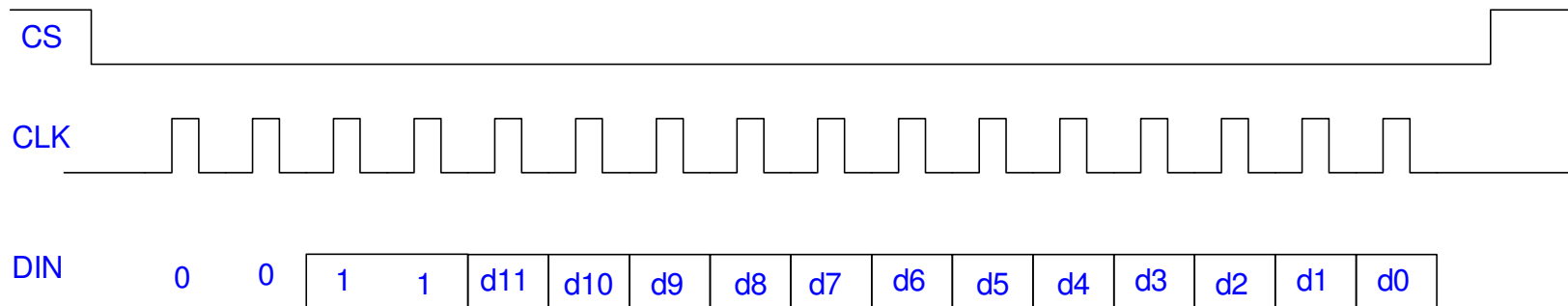
Three output pins are needed to drive the MCP4921.  
Any pins can be used for bit-banging, the SPI port is needed if using the SPI module in machine

---

To send a 12-bit number,

- Pad the first four bits of the message with binary 0011
- Then, pull chip-select low (CS = 0)
- Clock in each bit, starting with the most significant bit.
- Once all 16 bits have been sent, pull CS high.

At that point, a voltage should appear on Vout



Timing diagram for a MCP4921

---

You can implement this in software with a bit-banging routine:

```
from machine import Pin
from time import sleep_ms, sleep_us, ticks_us

CLK = Pin(10, Pin.OUT)
DATA = Pin(11, Pin.OUT)
CS = Pin(9, Pin.OUT)

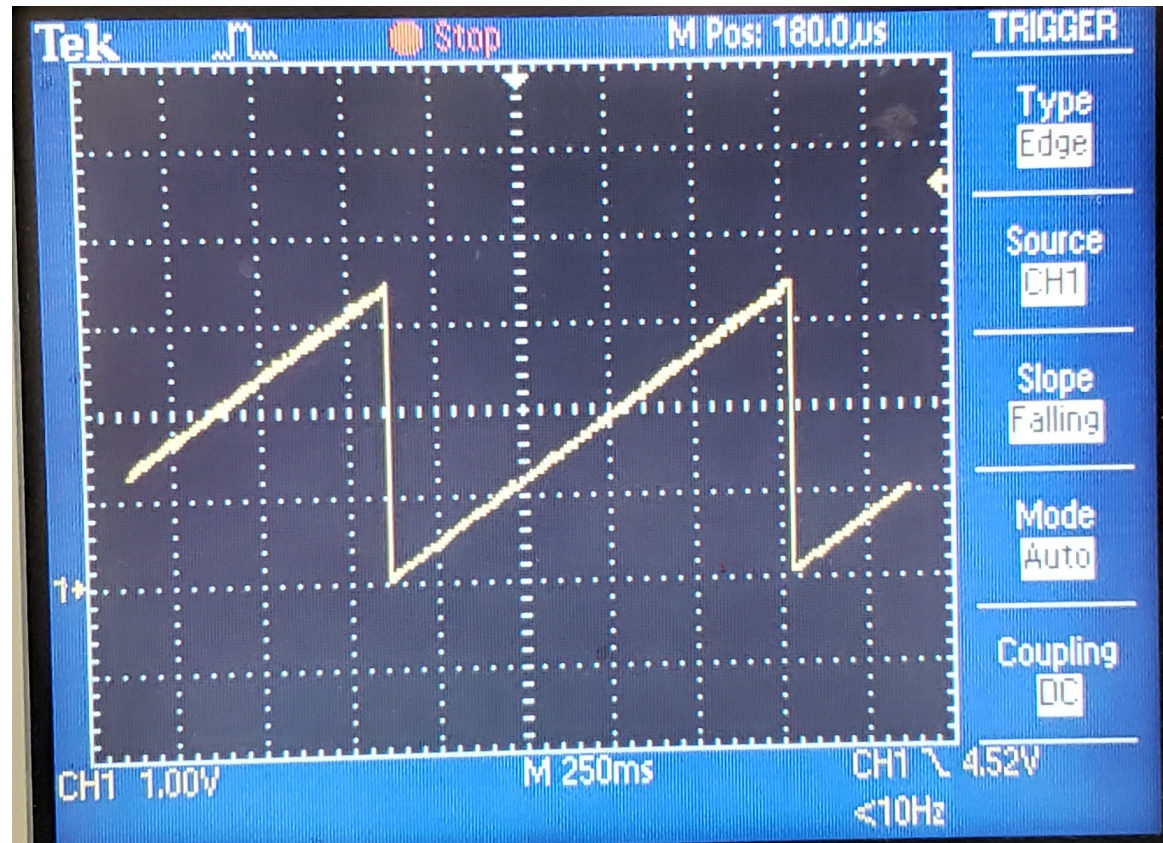
def MCP4921(X):
    X = X & 0xFFFF
    X = X | 0x3000

    CS.value(0)
    CLK.value(0)
    sleep_us(1)
    for i in range(0,16):
        if(X & (0x8000 >> i)):
            DATA.value(1)
        else:
            DATA.value(0)
        CLK.value(1)
        sleep_us(1)
        CLK.value(0)
        sleep_us(1)
    CS.value(1)
    DATA.value(0)
    sleep_ms(1)

x = 0
while(1):
    x = (x + 10) & 0xFFFF
    MCP4921(x)
    sleep_ms(1)
```

## Counting results in a sawtooth wave

- Count goes from 0 to 4095
- Voltage goes from 0V to 3.3V (Vdd)



If you look at the CLK and DATA lines on an oscilloscope, you can see the data being clocked out as well:



CLK (yellow) and DATA (blue) lines going to the MCP4921 D/A chip



---

## Using the SPI function is better

- 1700us with bit-banging
- 120us with SPI

Also simplifies the code

```
from machine import Pin, SPI
from time import sleep_ms, sleep_us,
ticks_us

CS    = Pin(9, Pin.OUT)

spi = SPI(1, baudrate=10_000_000,
polarity=0, phase=1, bits=8, sck=10,
mosi=11, miso=12)

def MCP4921(X):
    X = X & 0x0FFF
    X = X | 0x3000
    Y = bytearray()
    Y.append(X >> 8)
    Y.append(X & 0xFF)
    CS.value(0)
    spi.write(Y)
    CS.value(1)

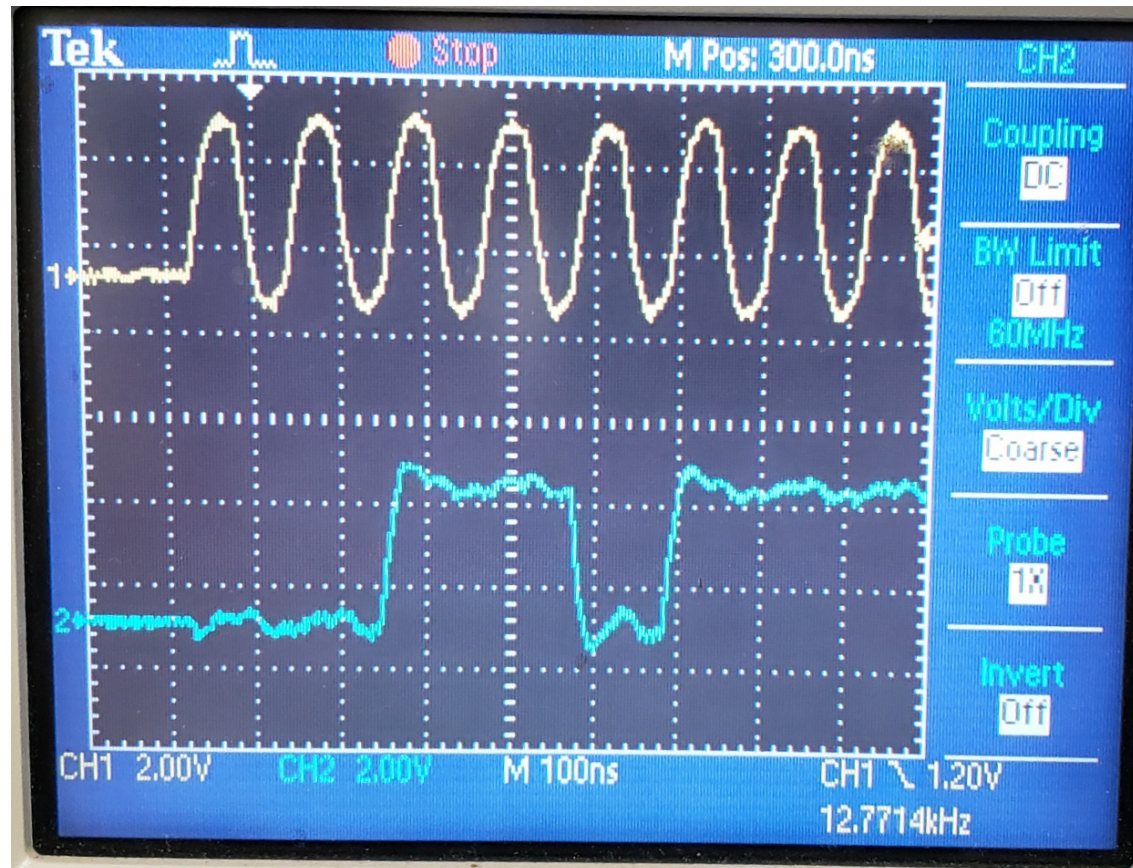
x = 0
while(1):
    x = (x + 10) & 0x0FFF
    MCP4921(x)
    sleep_ms(1)
```



---

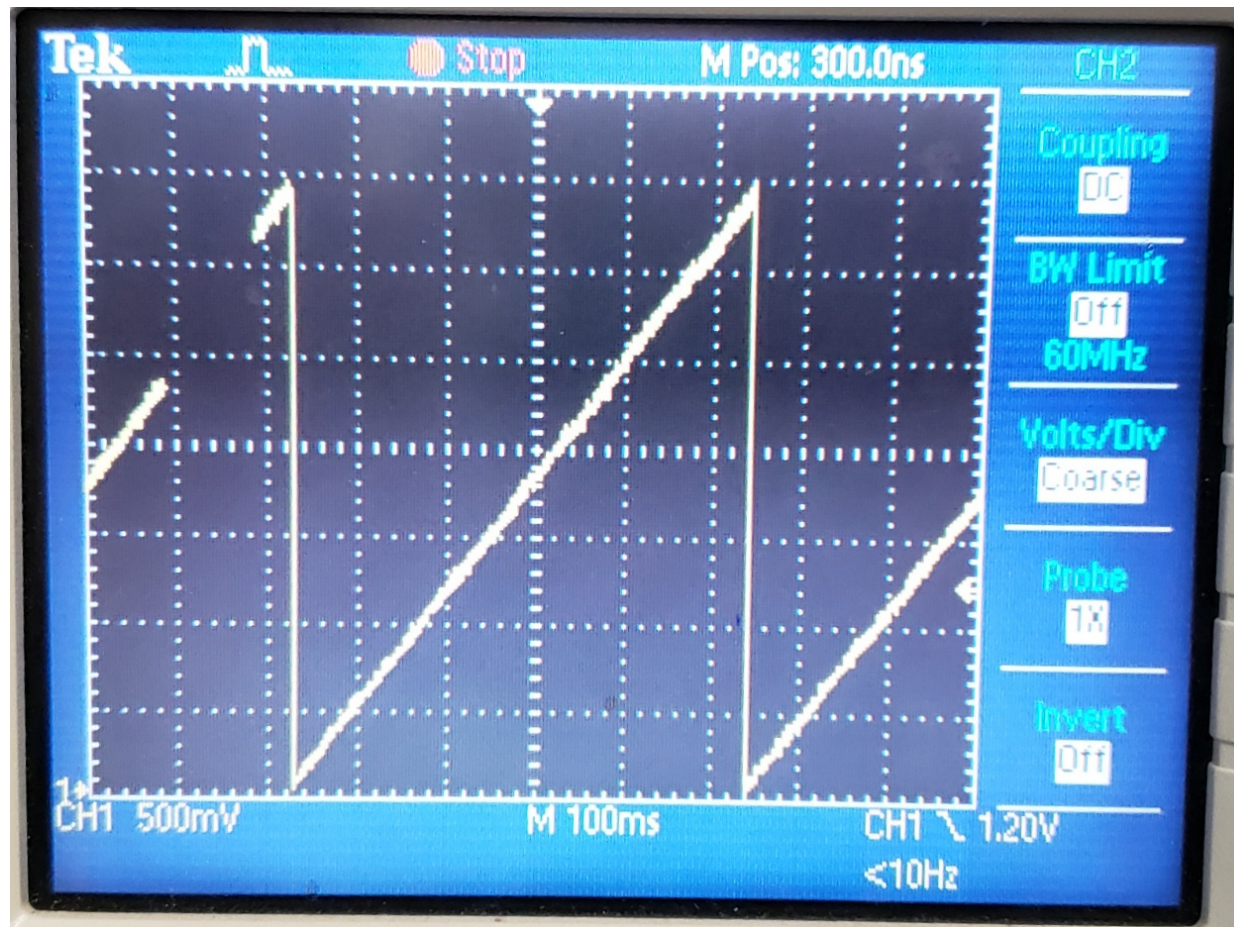
By using the SPI function, data is sent at 10MHz

- 14x faster than bit-banging
- Oscilloscope traces show limitations of a 60MHz scope.



CLK and DATA lines as seen on a 60MHz oscilloscope

---



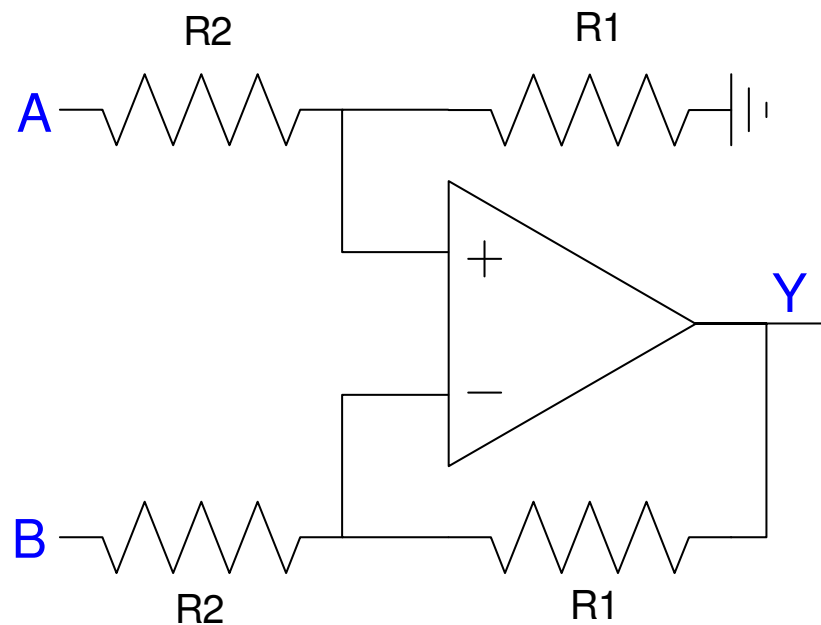
Output when the SPI bus is clocked at 10MHz.  
The D/A is working at this speed: the output is a sawtooth wave when counting

---

## Output -10V to +10V

Finally, if you want to output something other than 0V to 3.3V, an instrumentation amplifier can be added to the output. The gain of the following circuit is

$$Y = \left( \frac{R_1}{R_2} \right) (A - B)$$



Instrumentation Amplifier

---

---

If you want the output to go from -10V to +10V and x represents the D/A output

$$x = 0V..3.3V \quad \text{D/A output}$$

then

$$y = 6.06x - 10$$

Rewriting this in the form of the previous equation

$$y = 6.06(x - 1.65)$$

---

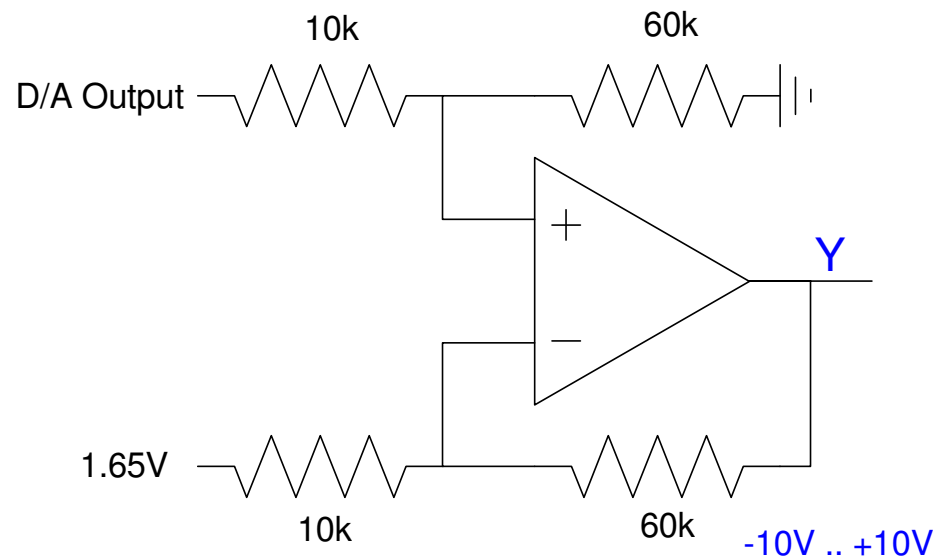
---

Let

- $A = x$
- $B = 1.65V$
- $(R1/R2) = 6.06$

then adding an instrumentation amplifier to the output of the D/A will produce

- -10V when you write 0x000 to the D/A
- +10V when you write 0xFF to the D/A



---

## Summary:

With a little code and some hardware, it isn't hard to

- Read analog voltages or
- Output analog voltages

When using a chip with an SPI input, you can use bit-banging to drive this device. Bit-banging has the advantage that you have complete control over how data is sent - but it tends to be slow. Using the built-in SPI port allows 10MHz or more data transfers, speeding up the process considerable (it also simplifies the code.)

---

# References

## Pi-Pico and MicroPython

- [https://github.com/geeekpi/pico\\_breakboard\\_kit](https://github.com/geeekpi/pico_breakboard_kit)
- [https://micropython.org/download/RPI\\_PICO/](https://micropython.org/download/RPI_PICO/)
- <https://learn.pimoroni.com/article/getting-started-with-pico>
- <https://www.w3schools.com/python/default.asp>
- <https://docs.micropython.org/en/latest/pyboard/tutorial/index.html>
- <https://docs.micropython.org/en/latest/library/index.html>
- <https://www.fredscave.com/02-about.html>

## Pi-Pico Breadboard Kit

- <https://wiki.52pi.com/index.php?title=EP-0172>

## Other

- <https://docs.sunfounder.com/projects/sensorkit-v2-pi/en/latest/>
  - <https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/>
  - <https://peppe8o.com/adding-external-modules-to-micropython-with-raspberry-pi-pico/>
  - <https://randomnerdtutorials.com/projects-raspberry-pi-pico/>
  - <https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/>
-

---

---