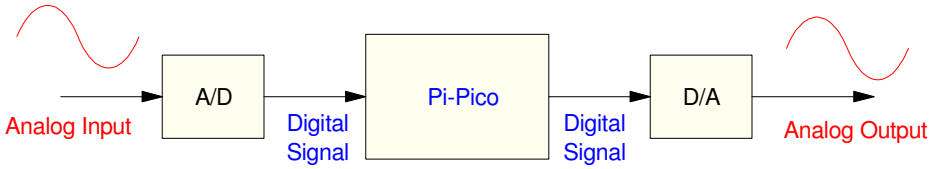
09. Analog I/O



Introduction:

Up to now, we have been dealing with binary inputs and outputs (I/O). The real world is typically analog, however. For example:

- 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

One feature that separates microcontrollers from microprocessors is the ability to input and output analog voltages. The Pi-Pico is no exception.

- On the input side, you have A/D converters (analog to digital). These A/Ds convert analog voltages to a digital number the microcontroller can read.
- On the output side, you have D/A converters (digital to analog). These allow the microcontroller to output analog voltages.

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 reads the x position of the joystick or the voltage on GP26
- ADC1 reads the y position of the joystick or the voltage on GP27
- ADC2 reads the voltage on GP28
- ADC3 is not connected, and
- ADC4 reads the temperature of the Pi-Pico

Code to read the joystick position (x, y) uses the ADC module from *machine*:

```
from machine import ADC
from time import sleep_ms

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

while(1):
    x = a2d0.read_u16() >> 4
    y = a2d1.read_u16() >> 4

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

Note that the command

```
a0 = a2d0.read_u16()
```

results in a 12-bit A/D number which is left-justified.

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

To make the number right-justifed, shift it right four times.

```
a0 = a2d0.read_u16() >> 4
```

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

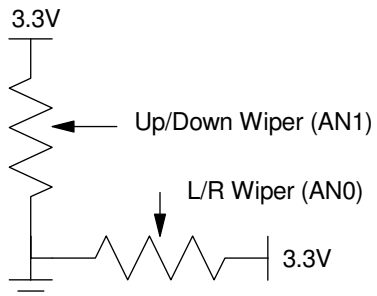
Which one you use is a matter of taste:

- Left justified results in the max value being 65,535 (regardless of whether you're using an 8-bit, 12-bit, or 16-bit A/D). This makes your code more portable.
- Right justified results in the max value being 4095 (2^{12} - 1). This makes it more clear that you're using a 12-bit A/D.

I personally prefer right justified - but there is something to be said for making your code more portable.

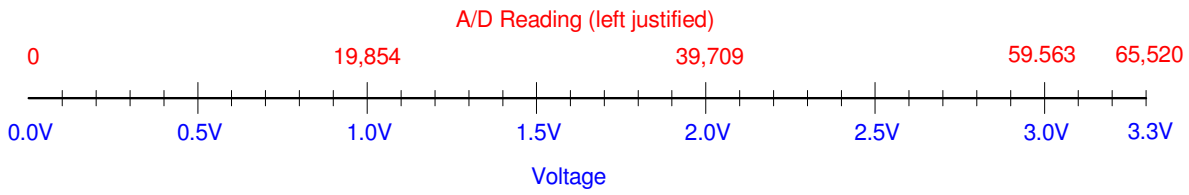
Reading 0V to 3.3V (Joystick Input)

If the input voltage is in the range of (0V, 3.3V), reading the analog input is pretty straight forward. An example of where you'd get such a voltage is the XY joystick input on your Pico-Breadboard-Kit. This has a schematic of:



Analog Inputs 0 and 1 are connected to two potentiometers (X & Y), resulting in a 0V - 3.3V signal

When you read the analog inputs, you receive an integer in the range of 0 .. 65,520 with the value proportional to the voltage:



Analog Inputs: A/D reading is proportional to voltage

This allows you to read the voltage on the GP input pins on the Pi-Pico. The conversion from A/D reading to voltage is:

$$V_i = \left(\frac{3.30V}{65{,}520}\right) \cdot A2D_i = 0.0000503663 \cdot A2D_i$$

A/D channels 0-2 are connected to I/O pins on the Pico-Breadboard Kit. A/D channel #4 is an internal A/D which reads the temperature of the Pi-Pico

$$^0C = 0.02927 * (14940 - A2D_4)$$

```
from machine import ADC
from time import sleep_ms

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

k = 3.3 / 65520

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

```
1.447679  1.499254  23.21111
1.435554  1.501671  23.21111
```

The results in the shell window tell you

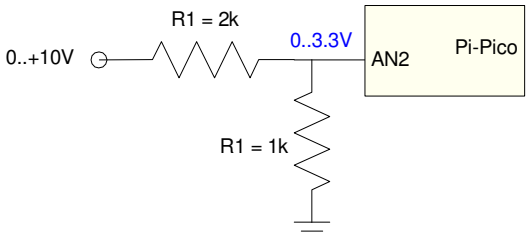
- The voltage on AD0 (1.43554V)
- The voltage on AD1 (1.501671V), and
- The temperature of the Pi-Pico (23.2111C)

Reading 0-10V: If you want to read voltages over a larger range, simply use a voltage divider. For example, to convert 0..10V to 0..3.3V, use a voltage divider with a gain of 0.33

$$gain = \left(\frac{R_2}{R_1+R_2}\right) = 0.33$$

If R1 = 1k, then

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



If you are reading 0..10V, use a voltage divider to bring the voltage down 3.3V (max)

Scale the computed voltage accordingly:

```
from machine import ADC
from time import sleep_ms

a2d2 = ADC(2)

k = 10.0 / 65520

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: Finally, if you want to read something like a signal that goes from -10V to +10V, a resistor circuit can be used (there are other solutions).

If you use resistors to create a weighted average with three inputs:

- 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}

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

The coefficients don't add up to one, so add a term times C (0V) to make them add up to 1.000

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

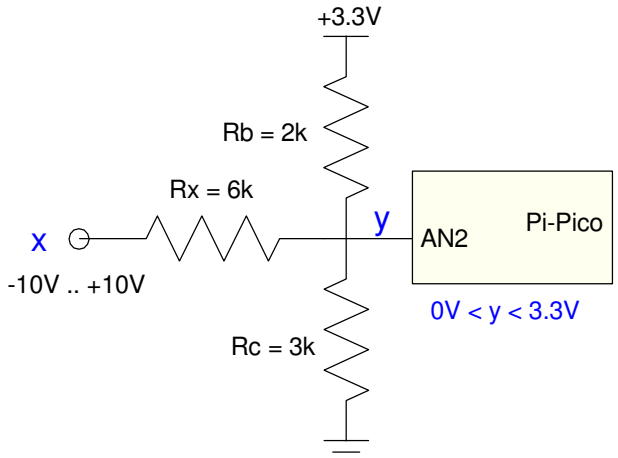
Pick your favorite resistor value, such as 1k. The weightings tell you how the resistors are scaled:

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

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

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

so a circuit which converts (-10V,+10V) to (0V, 3.3V) is



Circuit for converting (-10V, +10V) to (0V, +3.3V). (other solutions exist)

The corresponding Python code would be:

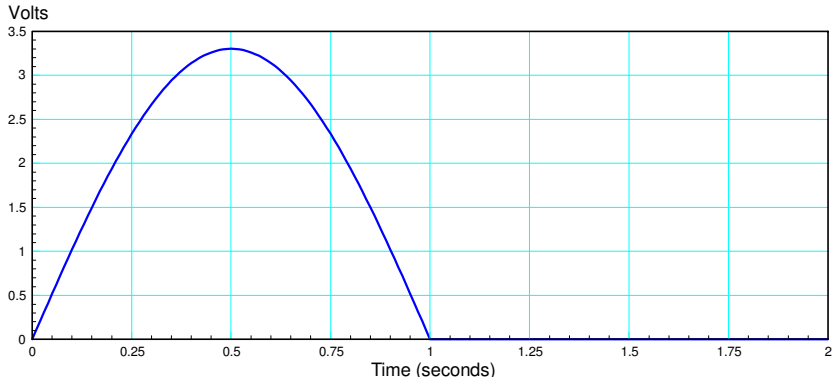
```
from machine import ADC
from time import sleep_ms

a2d2 = ADC(2)

k = 20.0 / 65520

while(1):
    a2 = a2d2.read_u16()
    V2 = k * (a2 - 32767)

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

shell

```
0.12131 Volts
0.01231 Volts
```

Analog Outputs:

With analog outputs, the goal is to output a voltage that can take on multiple value - not just binary 0V and 3.3V. For example, assume you want to output a half-rectified since wave with a period of 2 seconds:



Example of analog outputs: generate a 1/2 rectified sine wave

Here, we'll look at two ways to generate this waveform:

- PWM output
- PWM output with a low-pass filter, and
- A D/A (digital to analog) converter.

PWM (pulse width modulation)

From before, the *machine* library has a PWM function,. This allows you to output

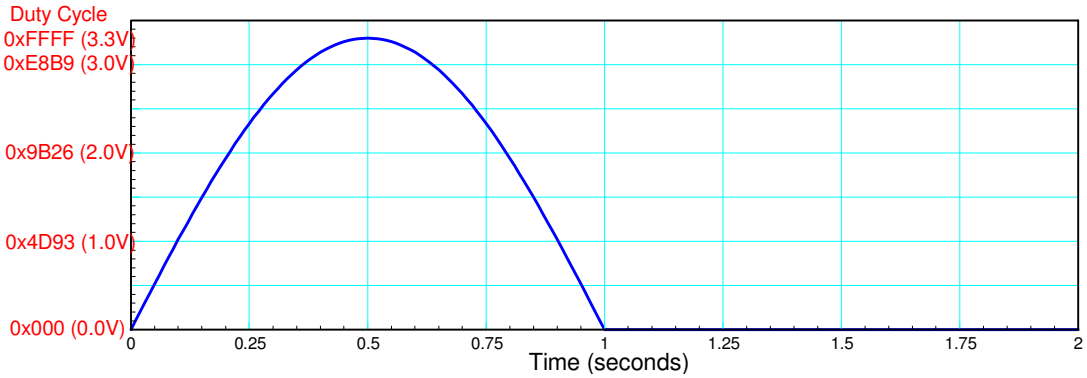
- A square wave,
- At a given frequency,
- With a given duty cycle.

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

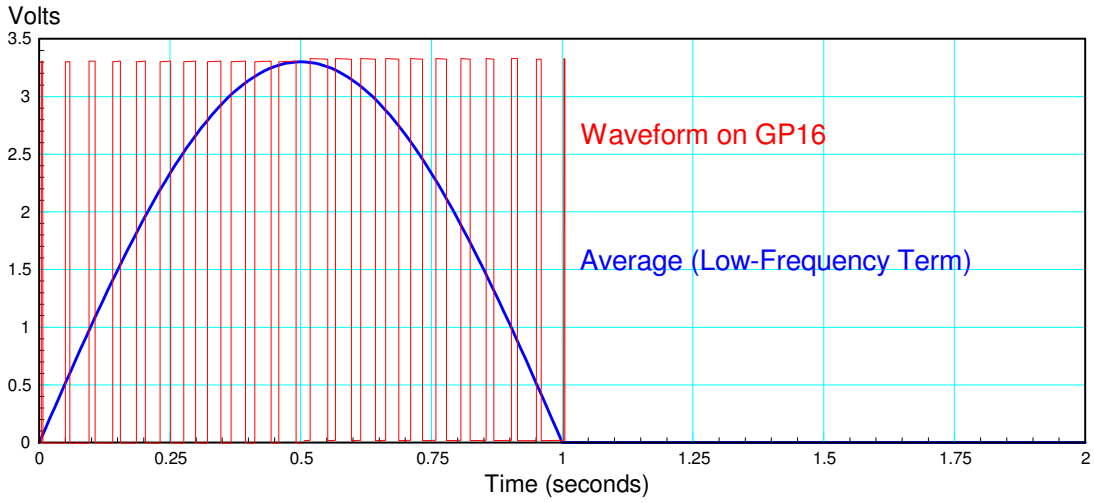
There are many ways coding this function. One way is to use a look-up table which specifies the duty cycle every 10ms. In the following program, the table is created before entering the main loop. This a little more efficient than placing the commutations inside the main loop:

- sin() is a computationally intensive function.
- By doing the computations before entering the main loop, the calculations only need to be done once.

Once you enter the main loop, you keep reusing the numbers computed previously.

```
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:



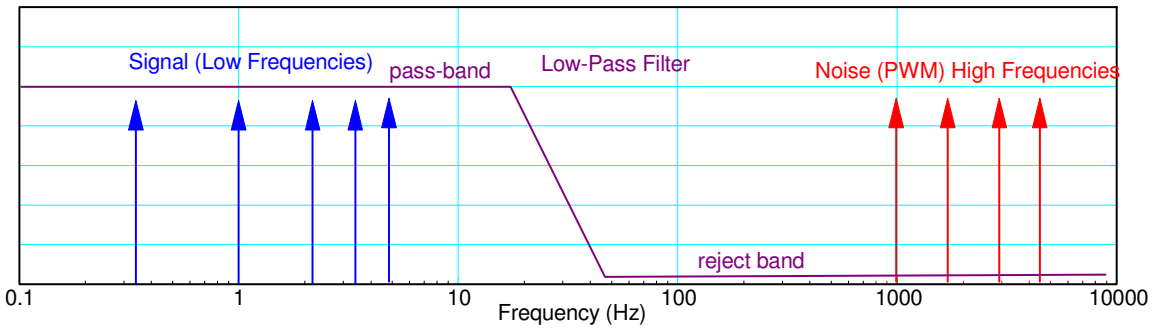
In terms of frequency content

- The signal (blue line) contains a DC term and harmonics of 1/2 Hz (the period)
- The PWM signal (red line) also contains 1kHz (the PWM frequency) and harmonics

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.

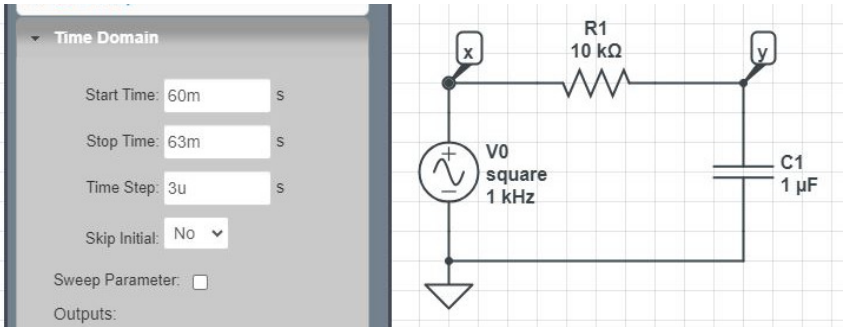


Add a low-pass filter to pass the signal (frequencies below 10Hz) and reject noise (frequencies above 1kHz)

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).



Add an RC filter to remove the 1kHz+ components from V0 (the PWM output of the Pi-Pico)
CircuitLab simulation

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:

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

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 since 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;

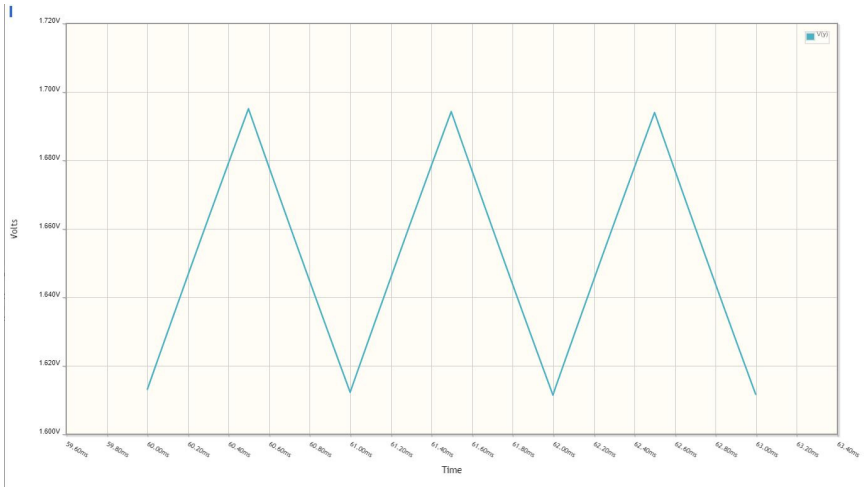
$$y = \left(\frac{100}{s+100}\right)x$$

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

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

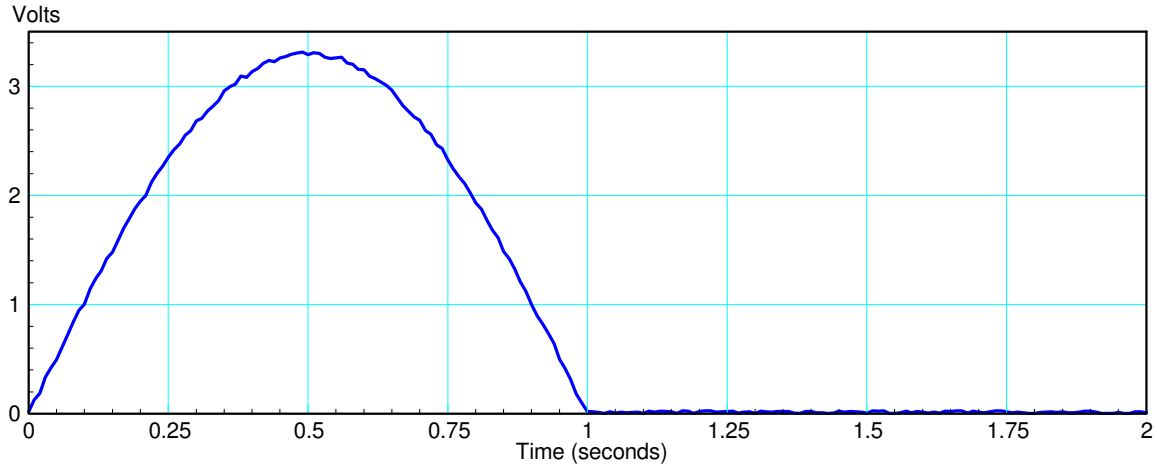
The signal at y should have the DC term plus a 52.5mVpp ripple.

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



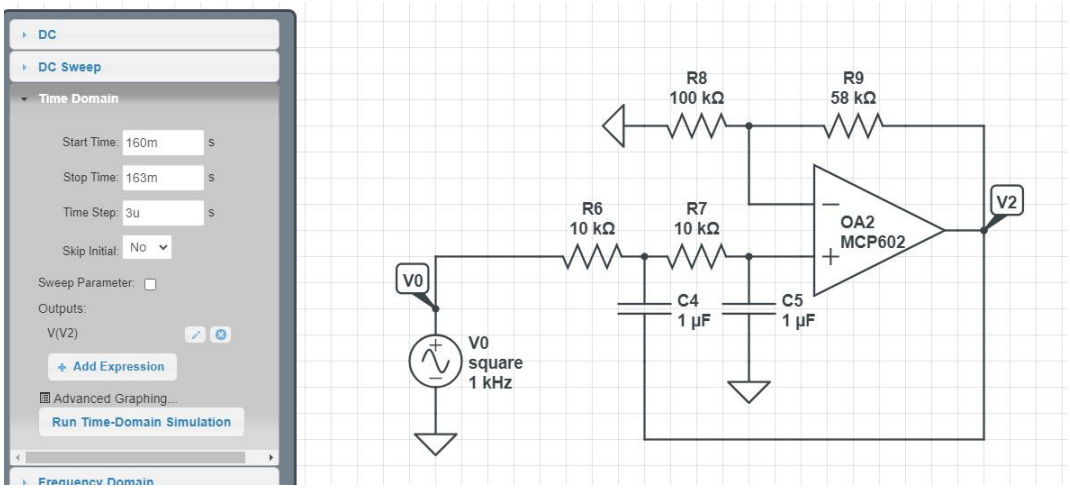
Time-Domain Simulation: The RC filter passes the DC term (average voltage is 1.65V)
The ripple has been reduced from 3.3Vpp to 80mVpp

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 R5*C4

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

The angle of the poles is set by R8 and R9

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

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

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

$$y = \left(\frac{100^2 \cdot k}{\left(s + 100\angle 45^0\right)\left(s + 100\angle -45^0\right)} \right) x$$

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 3.3Vpp

The ripple at y will be approximately

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

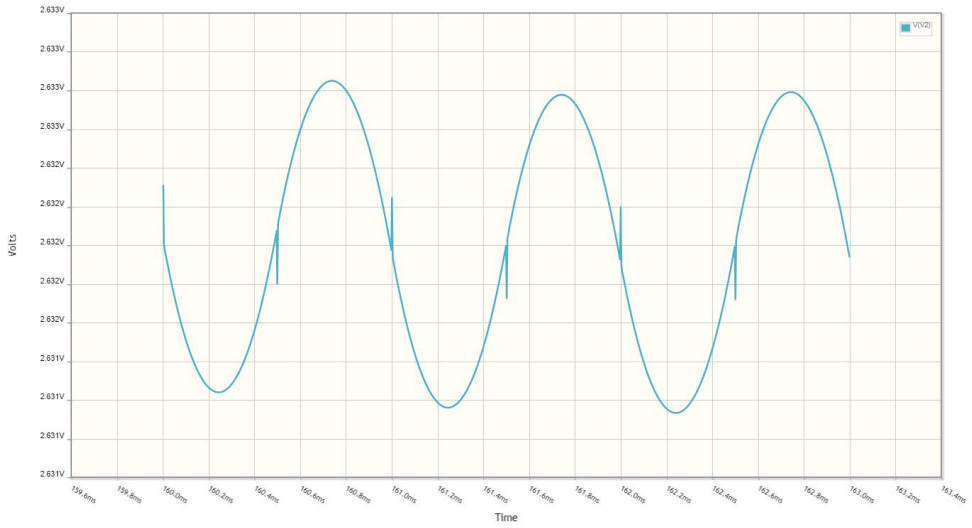
$$|y| \approx 0.0013$$

Meaning y(t) should have

- A DC term of 2.61V (1.58 * 50% of 3.3V)
- An AC term of 1.3mVpp

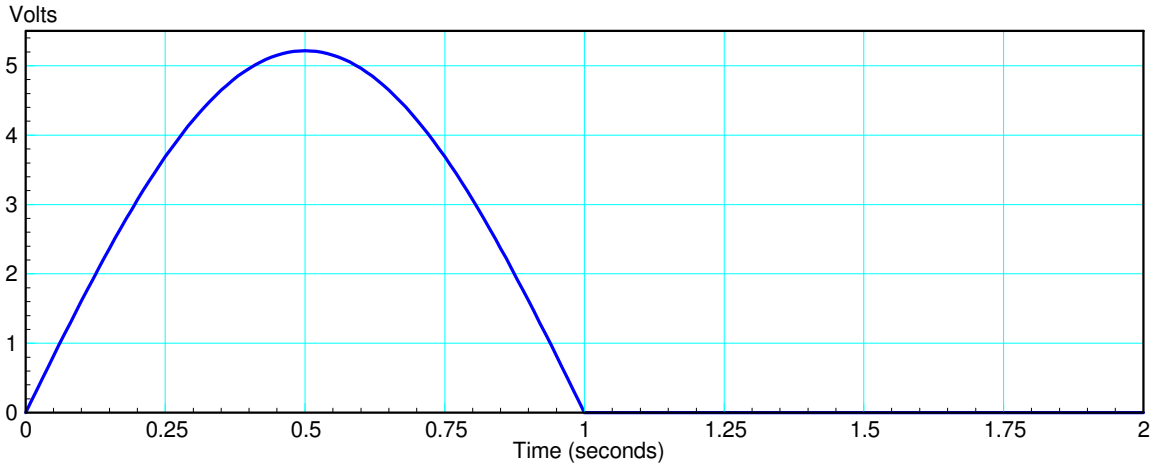
In CircuitLab, you can check the actual result is

- 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

Analog Outputs: MCP4921

A second way to output an analog voltage is to use a D/A (digital to analog) IC - such as the MCP4921. This IC is a 12-bit D/A, meaning

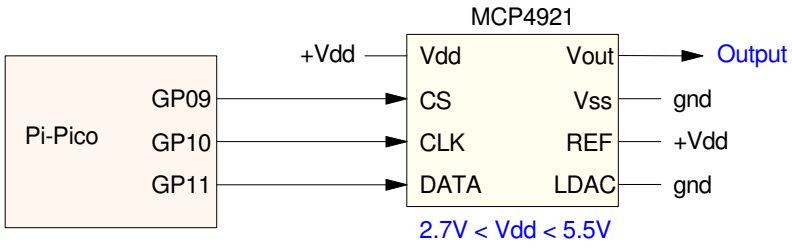
- The output can go from 0V to 5V (its power supply)
- In 4096 steps (2^{12})

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

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

Since there are no inputs to the Pi-Pico, Vdd can be anything between 2.7V and 5.5V. (Using 5V for the 4921 won't hurt the Pi-Pico since no voltages are sent back.)

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

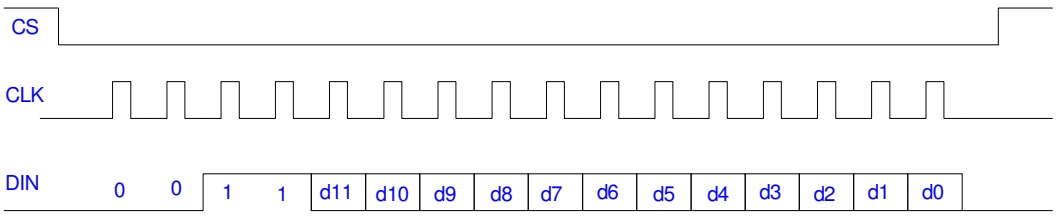


Three output pins are needed to drive the MCP4921.
Any pins can be used for bit-banging, the SPI port is needed if using the SPI module in machine

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:

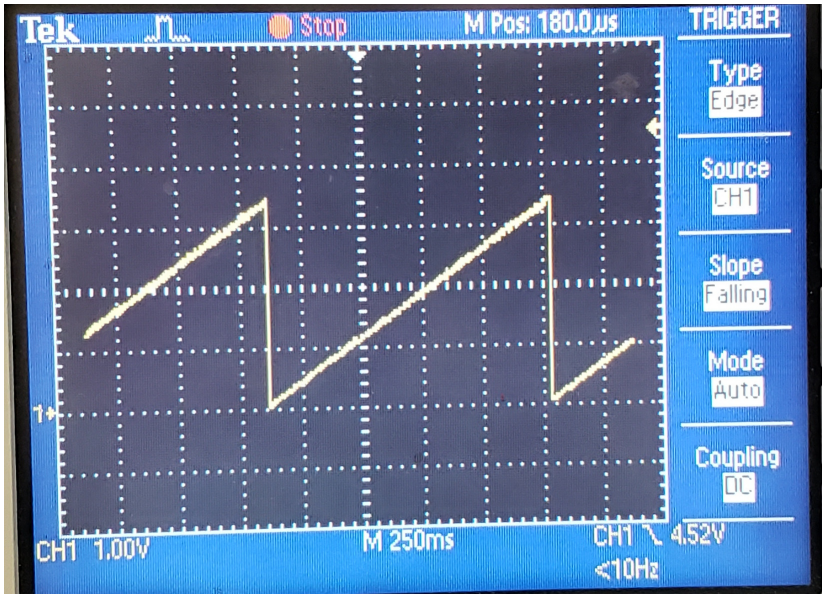
```
from machine import Pin
from time import sleep_ms, sleep_us, ticks_us

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

def MCP4921(X):
    X = X & 0x0FFF
    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) & 0x0FFF
    t0 = ticks_us()
    MCP4921(x)
    t1 = ticks_us()
    print(t1 - t0)
    sleep_ms(1)
```

shell

```
1718
1722
```

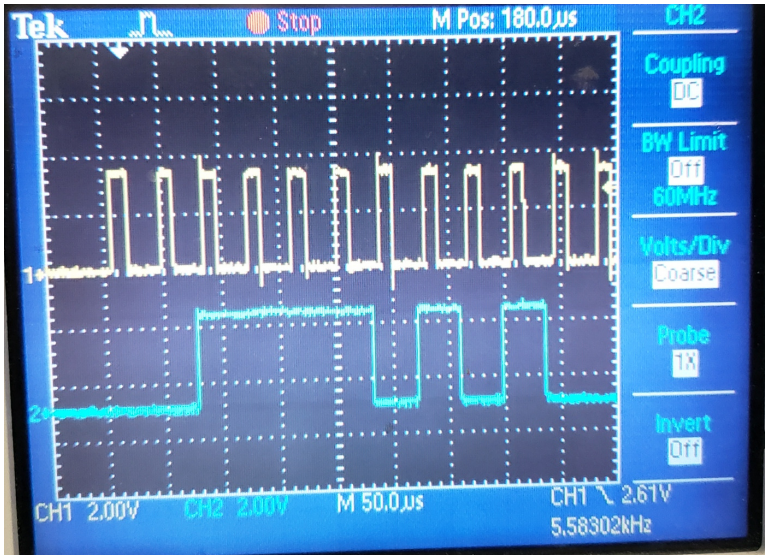
A couple of things to note:

- The code in blue isn't needed: it's just there to measure the execution time.
- Bit banging is slow: it takes 1720us to write to the D/A routine using this method.
- If you look at the output (pin 8) on an oscilloscope, you'll see a saw-thooth wave
  - The output of the D/A increases from 0V to 3.3V as you write 0x000 to 0xFFF to the D/A



Counting from 0x000 to 0xFFF results in the D/A outputting a sawtooth wave going from 0.00V to 3.3V

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

While bit-banging works, it takes considerable time, it is more efficient to use the built-in SPI port

```
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)
  #  sleep_us(1)
    spi.write(Y)
  #  sleep_us(1)
    CS.value(1)


x = 0
while(1):
    x = (x + 10) & 0x0FFF
    t0 = ticks_us()
    MCP4921(x)
    t1 = ticks_us()
    print(t1 - t0)
    sleep_ms(1)
```

shell

```
120
119
```

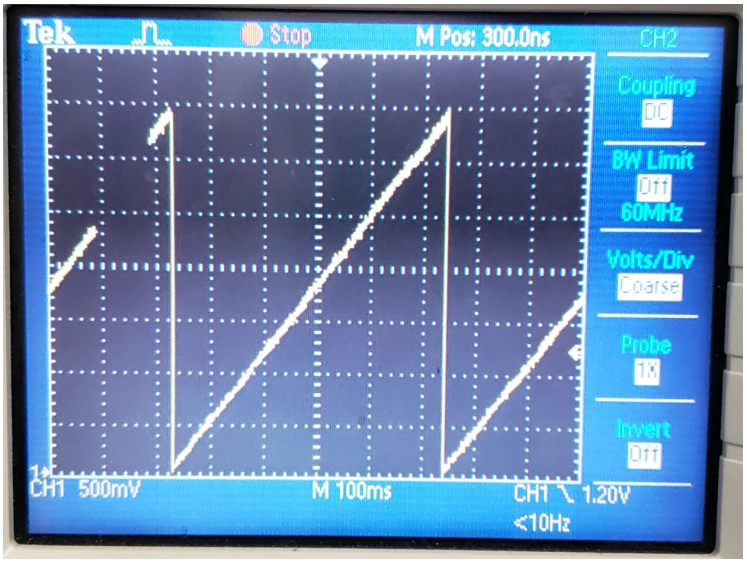
By switching to the SPI port, you get the same results (the D/A outputs a sawtooth wave from 0.0V to 3.3V), bit it now takes 120us to write to the D/A rather than 1700us.

Bit-Banging	SPI
1700us	120us

Time it takes to write to the MCP4921

With the built-in SPI, data is being sent at 10MHz - which is pushing the capabilities of my 60MHz oscilloscope. Regardless, the data is getting through.

CLK and DATA lines as seen on a 60MHz oscilloscope. 10MHz SPI communications is pushing my scope

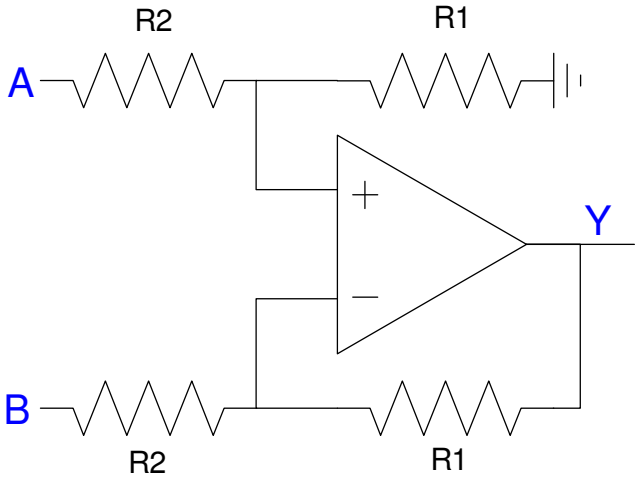


Output when the SPI bus is clocked at 10MHz.
The D/A is working at this speed: the output is a sawtooth wave when counting

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

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

then

$$y = 6.06x - 10$$

Rewriting this in the form of the previous equation

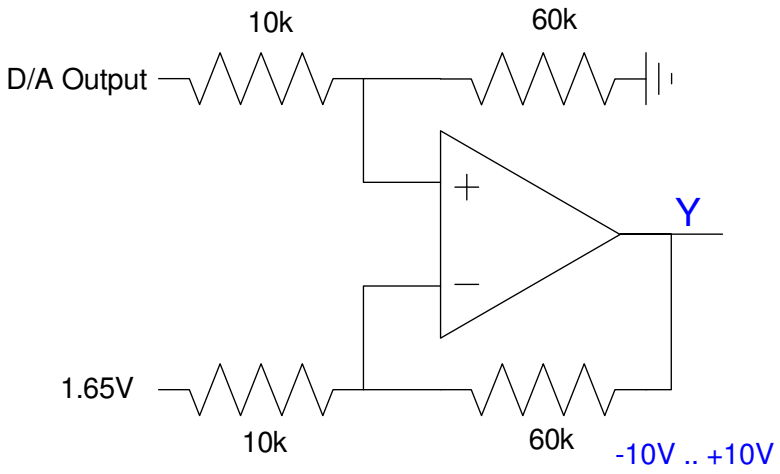
$$y = 6.06(x - 1.65)$$

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 0xFFF to the D/A



Adding an instrumentation amplifier to the output of the D/A allows the output to go from -10V to +10C

Pretty much any output voltage range is possible by playing with the gain and the offset.

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.)