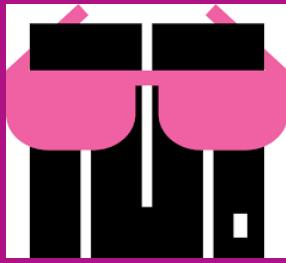
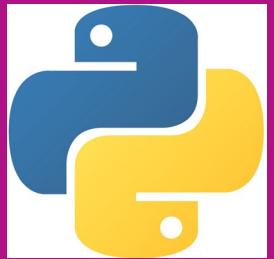
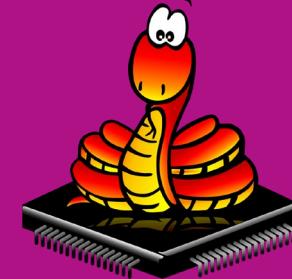


INTERNET OF THINGS (IOT) PROJECTS USING PYTHON

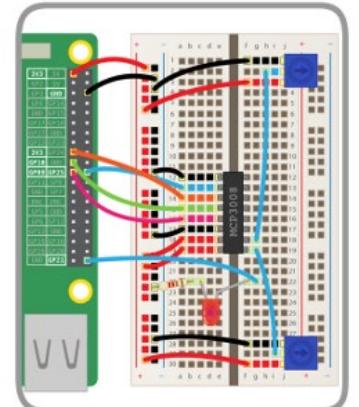
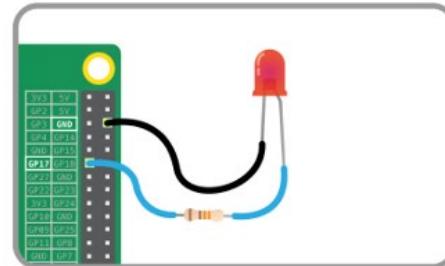
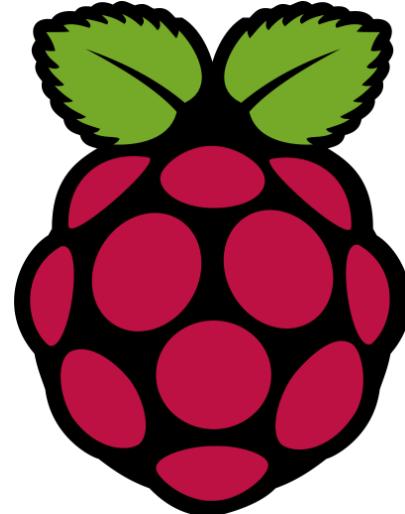


(CSE 4110)

(LECTURE – 6)



T_h



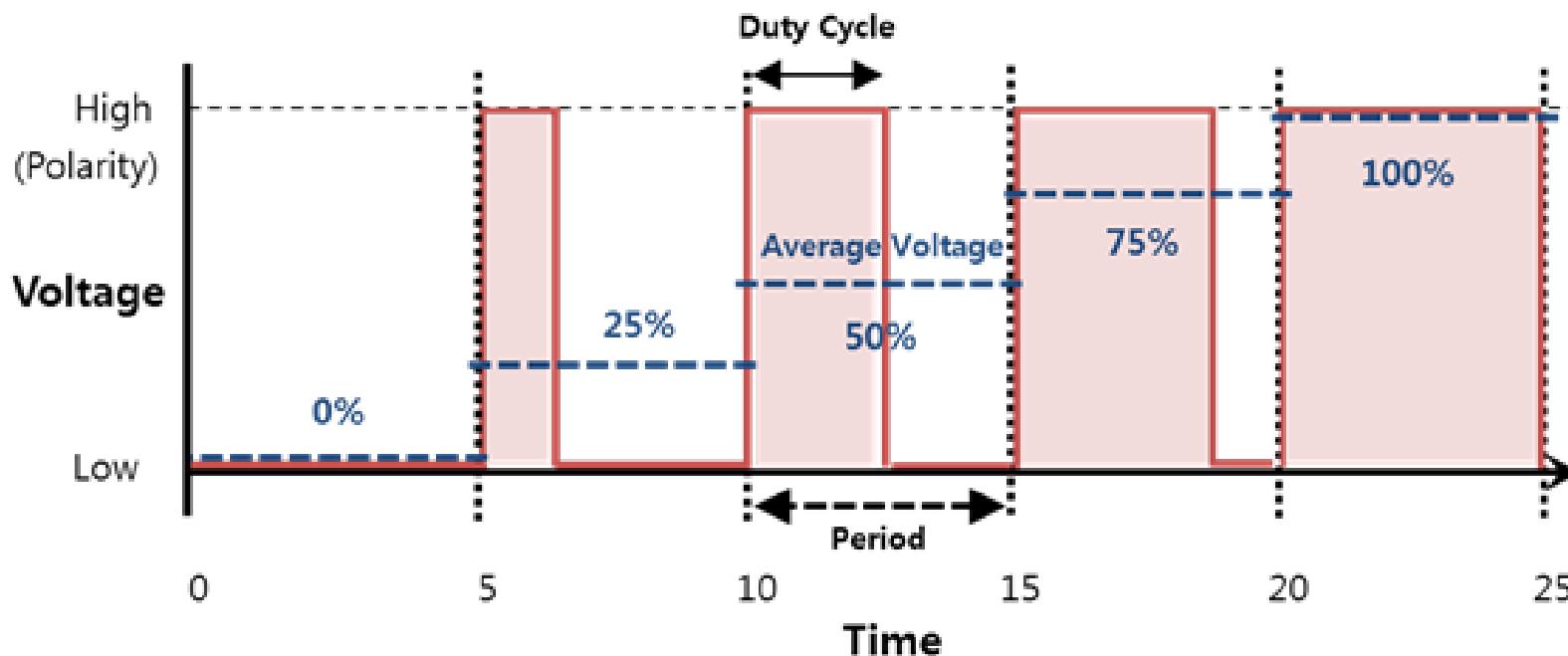
Pulse With Modulation (PWM)

PWM, a trick to generate variable voltages on digital pins

The **PWM** is a technique that allows the generation of a voltage between 0 and 3.3V using **only digital outputs** .i.e, getting analog results with digital means.

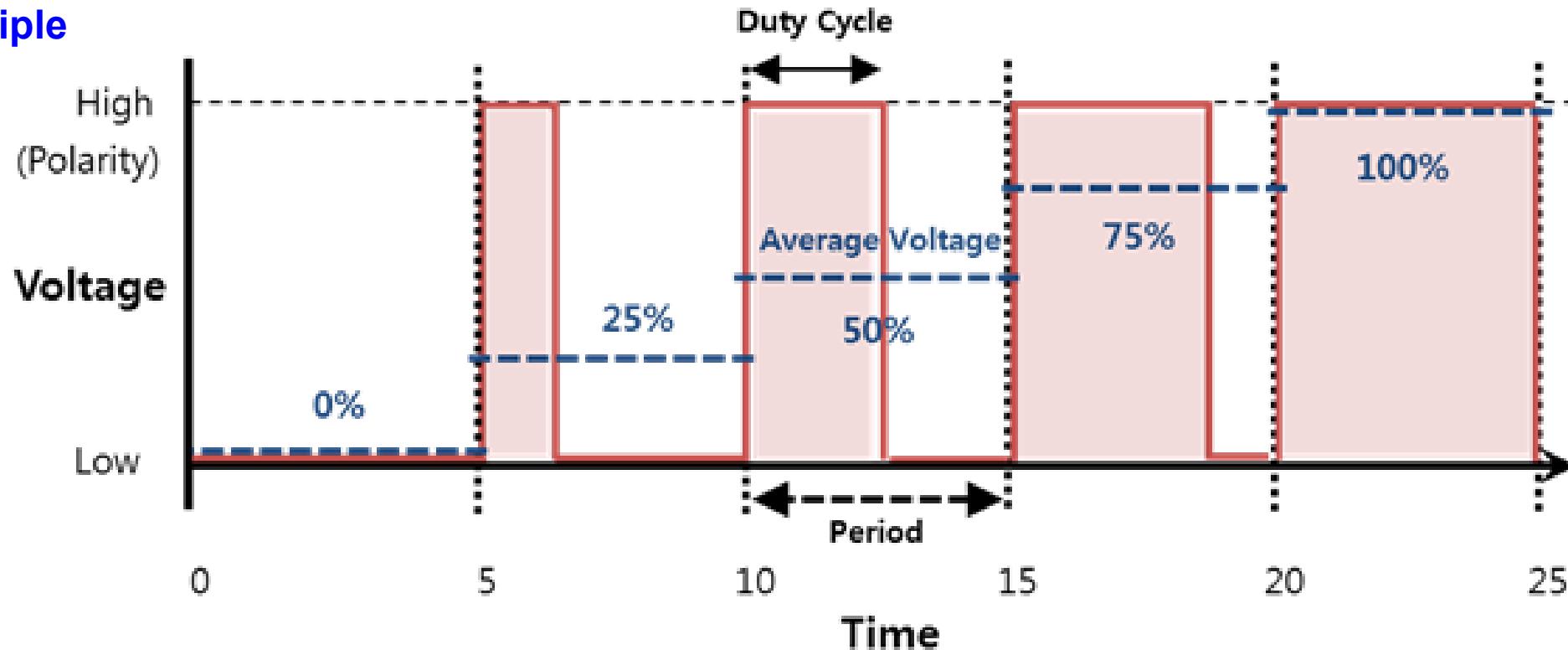
The PWM consists in varying the width of an electrical pulse.

The PWM allows the generation of constant voltages whose value can be changed. It does not generate alternating voltages as a DAC could do.



Pulse With Modulation (PWM)

PWM principle



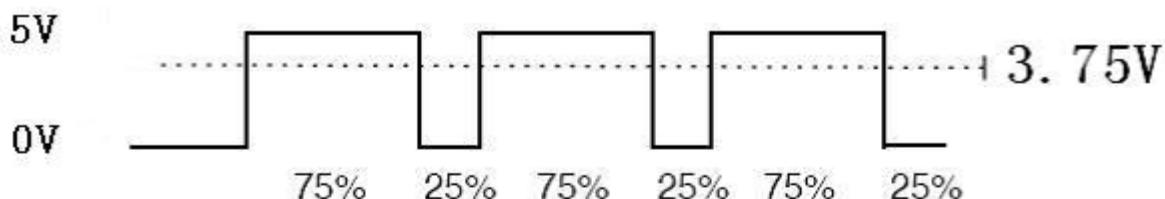
The succession of pulses with a given width is seen on average as a constant voltage between 0V and 3.3V, whose value is determined by :

$$V_{output} = V_{input} \times \alpha$$

with α , the duty cycle (the pulse width in percent)

Pulse With Modulation (PWM)

PWM principle



The duration of “on time” is called the **pulse width**. To get varying analog values, you change or modulate, that width. If you repeat this on-off pattern fast enough with some device, an LED, for example, it would be like this: the signal is a steady voltage between 0 and 5V controlling the brightness of the LED.

The smaller the PWM value is, the smaller the value will be after being converted into voltage. Then the LED becomes dimmer accordingly. Therefore, we can control the brightness of the LED by controlling the PWM value.

PWM Pins in Raspberry Pi Pico

PWM Pins

All 30 GPIO pins on RP2040 can be used for PWM:

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

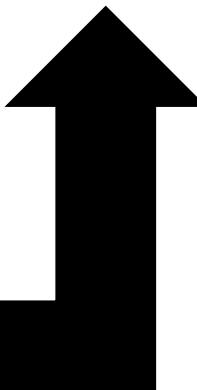
- The RP2040 microcontroller at the core of the Raspberry Pi Pico has **8 Slices** of PWM, and each Slice is **independent** of the others. This simply means that we can set the frequency of a Slice without affecting the others.
- The RP2040 PWM block has 8 identical PWM slices, each with two output channels (A/B), where the **B pin can also be used as an input for frequency and duty cycle measurement**. That means each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a **total of up to 16 controllable PWM outputs**. All 30 GPIO pins can be driven by the PWM block.

PWM Pins in Raspberry Pi Pico

PWM Pins

PWM_A[0]	GP0	1	1	VBUS
PWM_B[0]	GP1	2	2	VSYS
GND		3		GND
PWM_A[1]	GP2	4		3V3_EN
PWM_B[1]	GP3	5		3V3(OUT)
PWM_A[2]	GP4	6		ADC_VREF
PWM_B[2]	GP5	7		GP28
GND		8		PWM_A[6]
PWM_A[3]	GP6	9		33
PWM_B[3]	GP7	10		GND
PWM_A[4]	GP8	11		AGND
PWM_B[4]	GP9	12		32
GND		13		GP27
PWM_A[5]	GP10	14		PWM_B[5]
PWM_B[5]	GP11	15		31
PWM_A[6]	GP12	16		GP26
PWM_B[6]	GP13	17		PWM_A[5]
GND		18		30
PWM_A[7]	GP14	19		RUN
PWM_B[7]	GP15	20		29
				GP22
				PWM_A[3]
				28
				GND
				27
				GP21
				PWM_B[2]
				26
				GP20
				PWM_A[2]
				25
				GP19
				PWM_B[1]
				24
				GP18
				PWM_A[1]
				23
				GND
				22
				GP17
				PWM_B[0]
				21
				GP16
				PWM_A[0]

pins **21/GP16** and **22/GP17** belong to the same slice – GP16 is output A and GP17 is output B.



Pulse With Modulation (PWM)

The following factors influence the behaviour of a pulse width modulated signal.

1) duty cycle

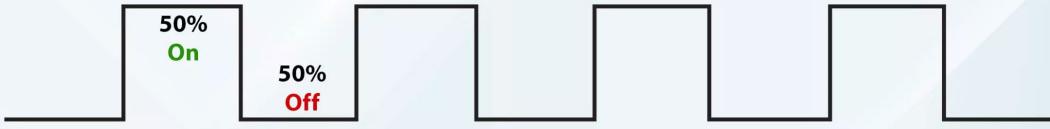
2) pulse frequency.

3) Resolution

We can modify these two parameters in MicroPython.

Duty Cycle

50% Duty Cycle



75% Duty Cycle



25% Duty Cycle



Frequency :

- Technically, frequency means “number of cycles per second”. When we toggle a digital signal (ON and OFF) at a high frequency, the result is an analog signal with a constant voltage level.
- PWM frequency is determined by the clock source.
- The frequency and resolution of PWM are inversely proportional.

Resolution:

- A PWM signal's resolution refers to the number of steps it can take from zero to full power.
- The resolution of the PWM signal can be changed.
- The Raspberry Pi Pico module has a 1 - 16 bit resolution, which means we can set up to **65536** steps from zero to full power.

Duty Cycle : It is the ratio of **ON time** (when the signal is high) to the total time taken to complete the cycle.

Pulse With Modulation (PWM)

PWM Application

In practice, PWM is used to :

- Controlling the speed of a motor
- Direction or position control of a servo motor
- Controlling the brightness of LEDs
- Loudness control in buzzers
- Generate square signals (with $\alpha=0.5$)
- Generate music notes (sound similar to retro consoles)
- Controlling the fan speed

Pulse With Modulation (PWM)

PWM performance on the Raspberry Pi Pico

The microprocessor doesn't generate PWM signals permanently but uses dedicated hardware blocks. We only need to configure the PWM blocks once in the script to permanently create the signal in the background. We associate the output of a PWM block to a pin of our board. The processor resources will then be free to execute other tasks.

Each PWM block can have an independent frequency.

PWM characteristics	Pi Pico
Frequency range of the PWM signal	7 Hz to 125 Mhz
Independent PWM frequency	8
PWM Output	16
Duty cycle resolution	16 bits

In most cases, a PWM frequency of around **1000 Hz** will be sufficient.

Pulse With Modulation (PWM)

PWM on MicroPython with Pico

MicroPython automatically selects an available PWM block: it is unnecessary to indicate which one we intend to use.

The configuration consists in associating a **PWM** object to a **Pin** object and choosing the PWM frequency

Since the `PWM` object is also in the machine module, we can combine the two imports on a single line:

```
from machine import Pin  
from machine import PWM
```

is equivalent to:

```
from machine import Pin, PWM  
  
from machine import Pin, PWM  
  
pin = Pin(25, mode=Pin.OUT)  
pin_with_pwm = PWM(pin) # Attach PWM object on a pin
```

Once the pin of the board is linked to our **PWM** object, all the functions are done directly on the **PWM** object

LED Breathing : Voltage Variation using Built-in LED

PWM on MicroPython with Pico

In python, you can insert **underscores _** to make the numbers easier to read (and thus avoid counting zeros on large numbers). For example, to write **1 MHz instead of having 1000000 , we can put 1_000_000** .

We use the function **.duty_u16()** to select the duty cycle with a value between 0 and $2^{16} - 1$ (**0-65535**). The voltage is set directly on the output of the previously selected pin

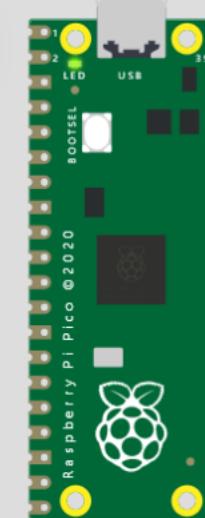
WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO 🐍

```
1 from machine import Pin, PWM
2
3 LED_BUILTLIN = 25 # For Raspberry Pi Pico
4
5 pwm_led = PWM(Pin(LED_BUILTLIN, mode=Pin.OUT)) # Attach PWM object on the LED pin
6
7 # Settings
8 pwm_led.freq(1_000)
9
10 while True:
11     for duty in range(0,65_536): # For Pi Pico
12         pwm_led.duty_u16(duty)
```

Simulation

00:21.133 21%



LED Breathing : Duty Cycle Percentage Specification

PWM on MicroPython with Pico

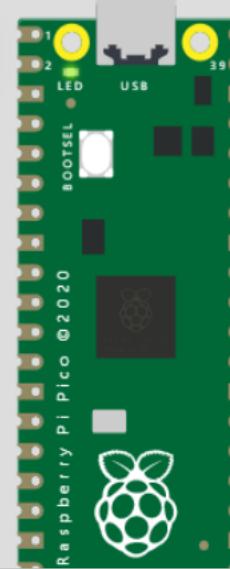
WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO

```
1 from machine import Pin, PWM
2
3 LED_BUILTIN = 25 # For Raspberry Pi Pico
4
5 pwm_led = PWM(Pin(LED_BUILTIN, mode=Pin.OUT)) # Attach PWM object on the LED pin
6 pwm_led.freq(1_000)
7
8 duty_cycle = 50 # Between 0 - 100 %
9 pwm_led.duty_u16(int((duty_cycle/100)*65_535))
```

Simulation

01:50.058 100%



MPY: soft reboot
MicroPython v1.18 on 2022-01-17; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> []

LED Breathing : Brightness Modification

PWM on MicroPython with Pico

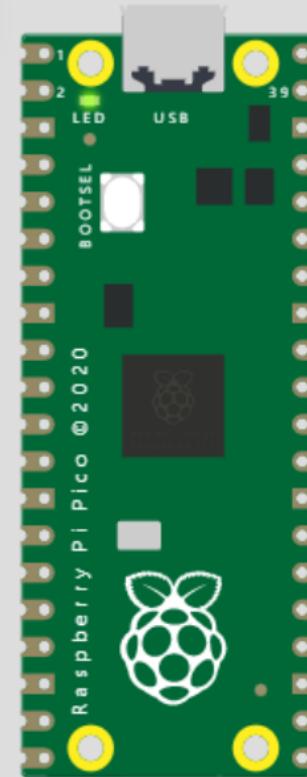
WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO

```
1 from machine import Pin, PWM
2 import time
3
4 LED_BUILTIN = 25 # For Raspberry Pi Pico
5
6 pwm_led = PWM(Pin(LED_BUILTIN, mode=Pin.OUT)) # Attach PWM object on the LED pin
7 pwm_led.freq(1_000)
8
9 while True:
10     for duty in range(101): # Duty from 0 to 100 %
11         pwm_led.duty_u16(int((duty/100)*65_535))
12         time.sleep_ms(10)
```

Simulation

01:53.652 99%



LED Breathing : Dimming the integrated inbuilt LED

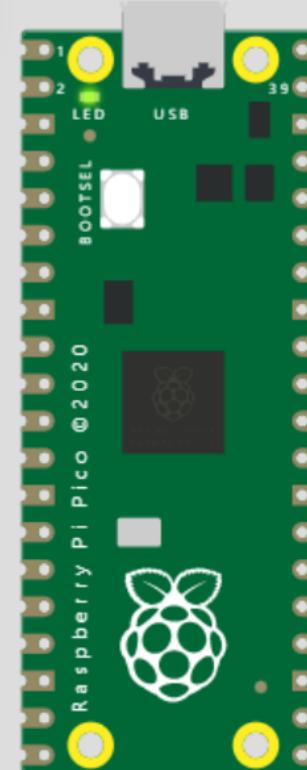
WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO

```
1 from machine import Pin, PWM
2
3 LED_BUILTLIN = 25 # For Raspberry Pi Pico
4
5 pwm_led = PWM(Pin(LED_BUILTLIN, mode=Pin.OUT)) # Attach PWM object on the LED pin
6
7 # Settings
8 pwm_led.freq(1_000)
9
10 while True:
11     for duty in range(0,65_536, 5):
12         pwm_led.duty_u16(duty)
13     for duty in range(65_536,0, -5):
14         pwm_led.duty_u16(duty)
```

Simulation

00:03.100 17%



LED Breathing : Dimming the External LED

WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO

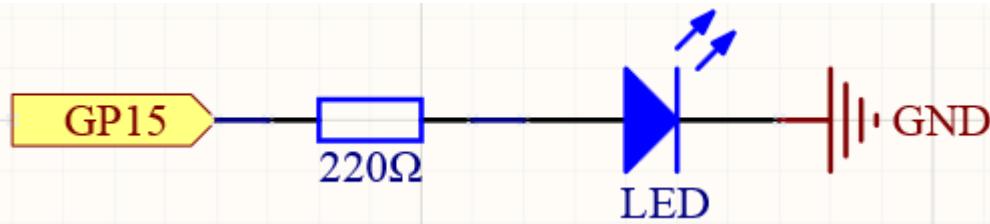
```
1 from machine import Pin, PWM
2 import utime
3
4 led = machine.PWM(machine.Pin(15))
5 led.freq(1000)
6
7 for brightness in range(0,65535,50):
8     led.duty_u16(brightness)
9     utime.sleep_ms(10)
10    led.duty_u16(0)
11
```

Simulation

00:03.516 100%

Raspberry Pi Pico © 2020

LED Breathing : Dimming the External LED

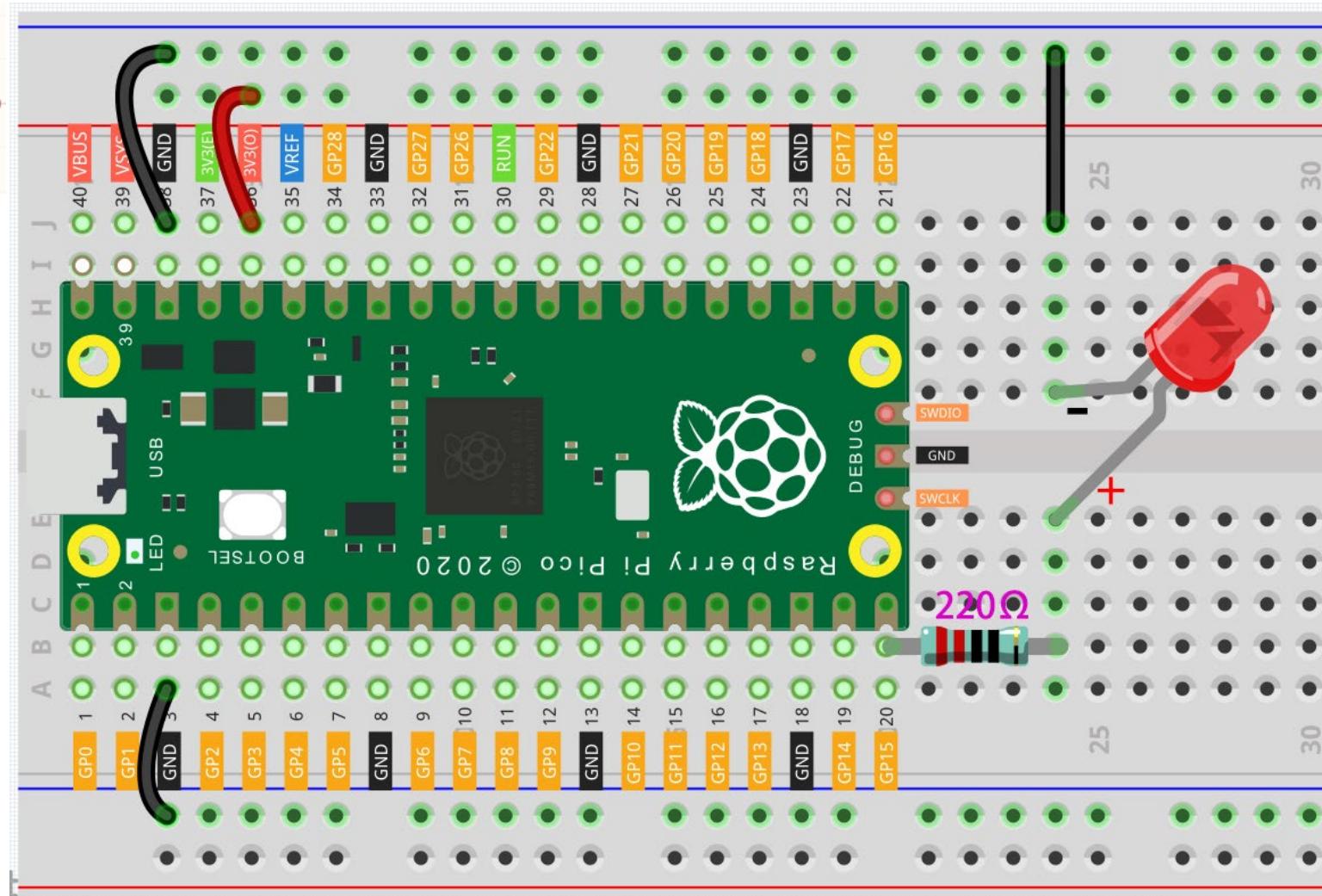


```
from machine import Pin, PWM  
import utime  
  
led = machine.PWM(machine.Pin(15))  
led.freq(1000)  
  
for brightness in range(0,65535,50):  
    led.duty_u16(brightness)  
    utime.sleep_ms(10)  
led.duty_u16(0)
```

duty cycle is 0%

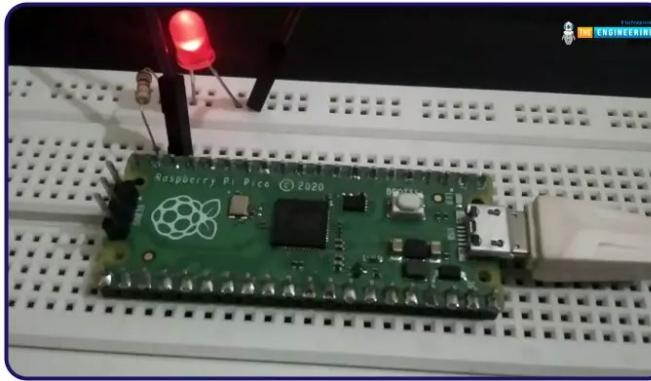


- 1) Change the duty cycle to 100%
- 2) Change the duty cycle to 50%



LED Breathing : Up and Down Fading the External LED

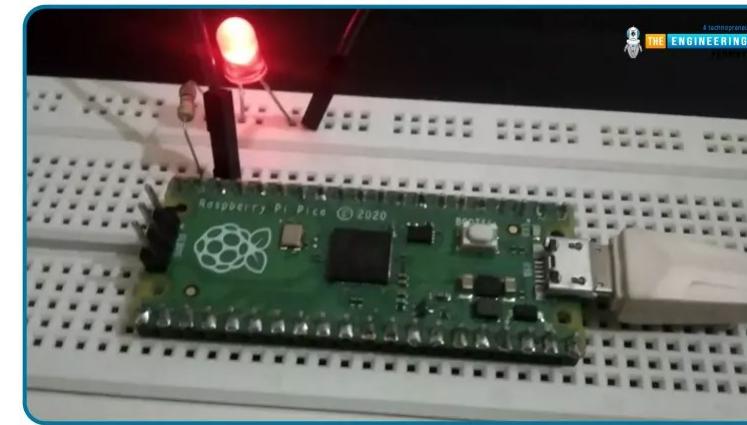
```
from machine import Pin, PWM  
import time  
  
led = PWM(Pin(14))  
  
print(led)  
  
led.freq(1000)  
  
while True:  
    for duty in range(0, 65535):  
        led.duty_u16(duty)  
        print(duty)  
        time.sleep(0.001)  
    for duty in range(65535, 1):  
        led.duty_u16(duty)  
        print(duty)  
        time.sleep(0.001)
```



Brightness level 1

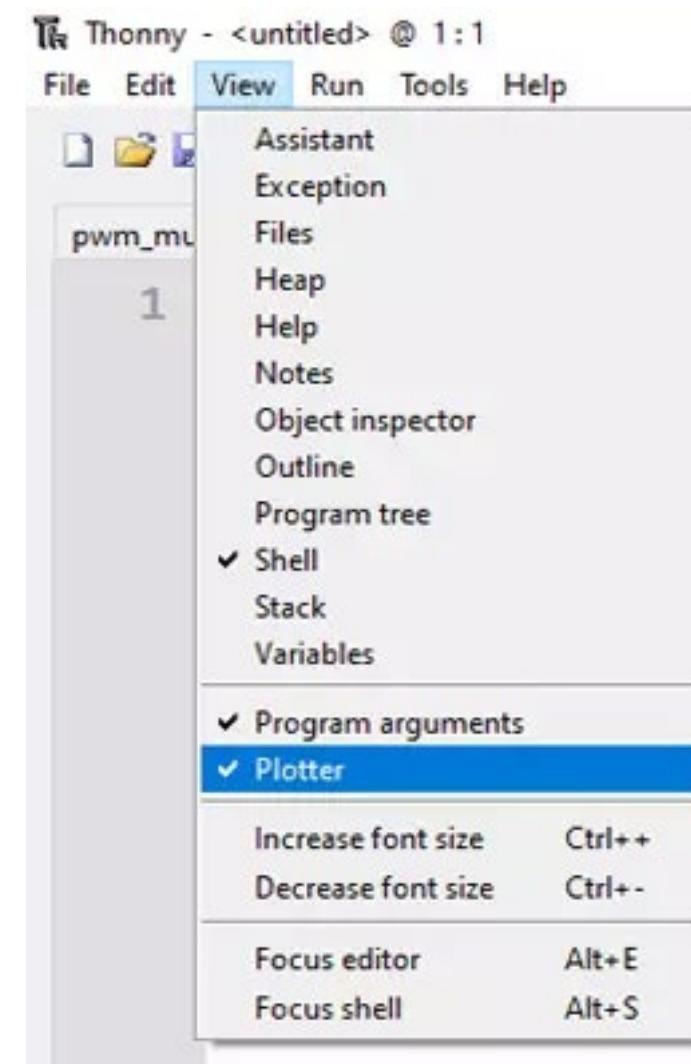


Brightness level 2

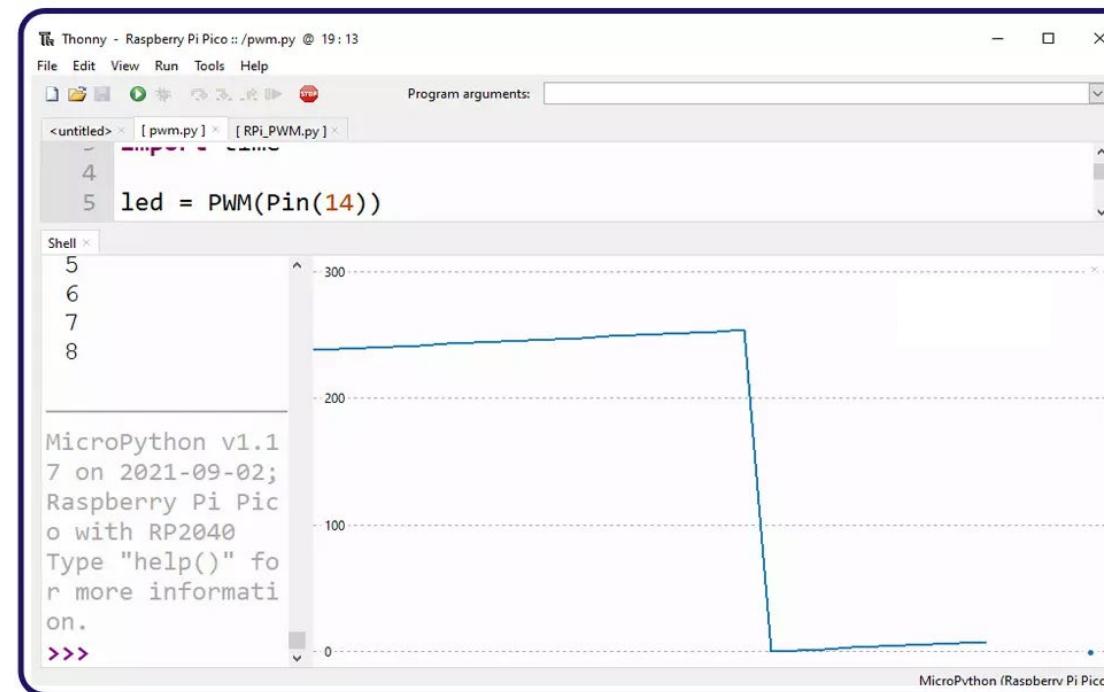


Brightness level 3

LED Breathing : Plotter in Thonny

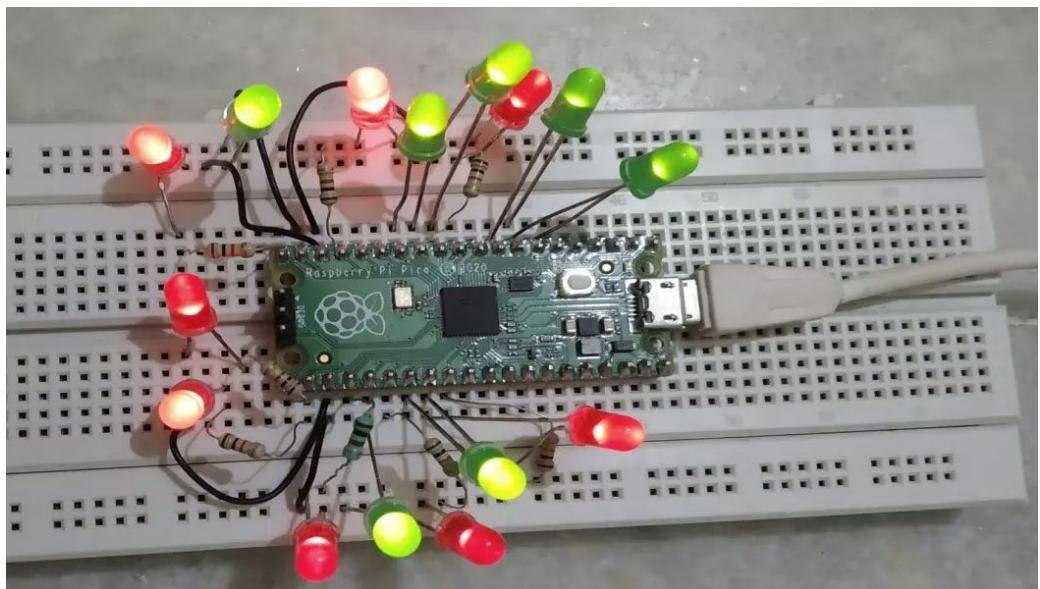
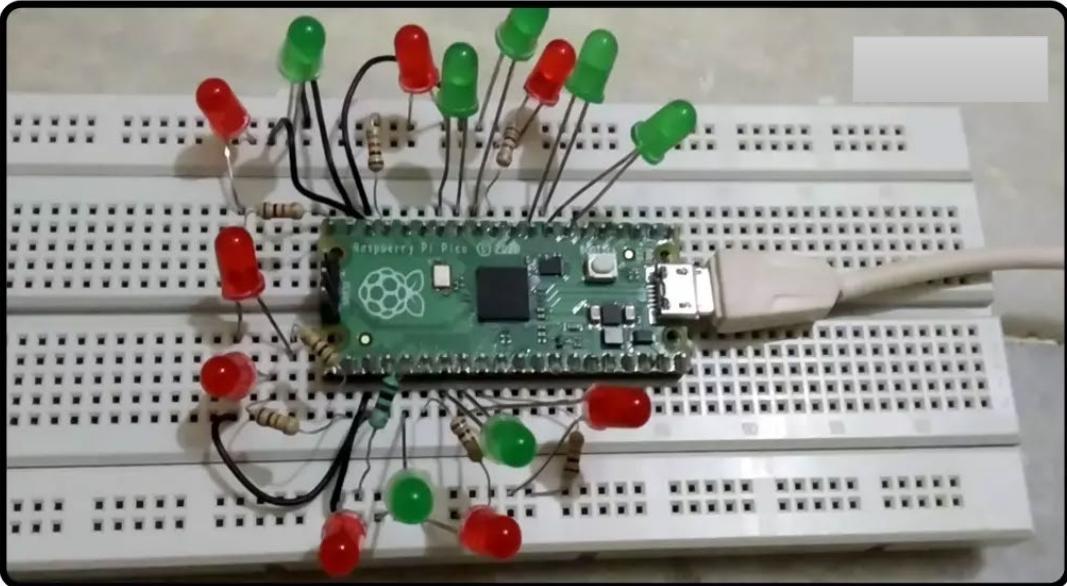


Rising PWM output 0 to
65535
(brightness increasing)

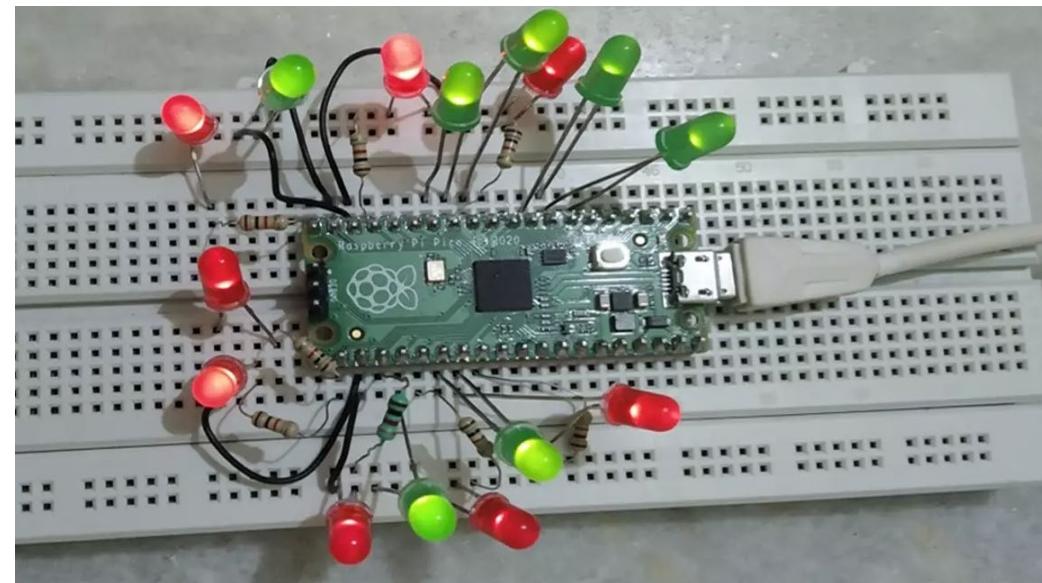


PWM output maximum
to 0

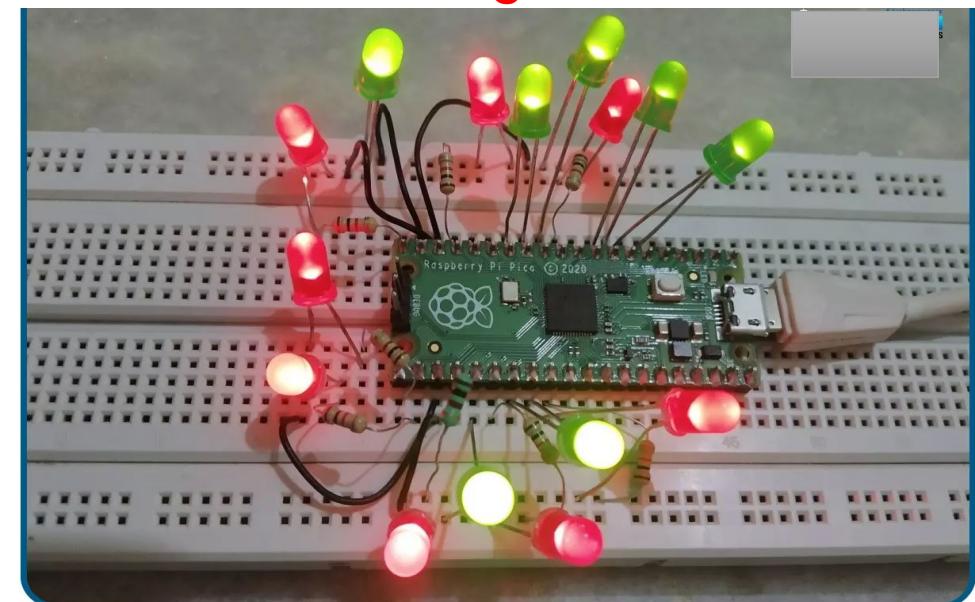
LED Breathing : Complete the Challenge



Intermediate Brightness Level



Minimum Brightness Level



Maximum Brightness Level

LED Breathing : Complete the Challenge

```
from machine import Pin, PWM
import time

led_1 = PWM(Pin(5))      # declaring led_x object for PWM pins
led_2 = PWM(Pin(6))
led_3 = PWM(Pin(8))
led_4 = PWM(Pin(9))
led_5 = PWM(Pin(10))
led_6 = PWM(Pin(13))
led_7 = PWM(Pin(14))
led_8 = PWM(Pin(15))
led_9 = PWM(Pin(16))
led_10 = PWM(Pin(17))
led_11 = PWM(Pin(18))
led_12 = PWM(Pin(19))
led_13 = PWM(Pin(20))
led_14 = PWM(Pin(21))
led_15 = PWM(Pin(22))
led_16 = PWM(Pin(26))

print(led_1)

def led_freq(x):          # Generating frequency for all LEDs
    led_1.freq(x)
    led_2.freq(x)
    led_3.freq(x)
    led_4.freq(x)
    led_5.freq(x)
    led_6.freq(x)
    led_7.freq(x)
    led_8.freq(x)
    led_9.freq(x)
    led_10.freq(x)
    led_11.freq(x)
    led_12.freq(x)
    led_13.freq(x)
    led_14.freq(x)
    led_15.freq(x)
    led_16.freq(x)

    led_freq(1000)        # setting pulse width modulation frequency
```

Try to complete the circuit connection with the help of code

LED Breathing : Complete the Challenge

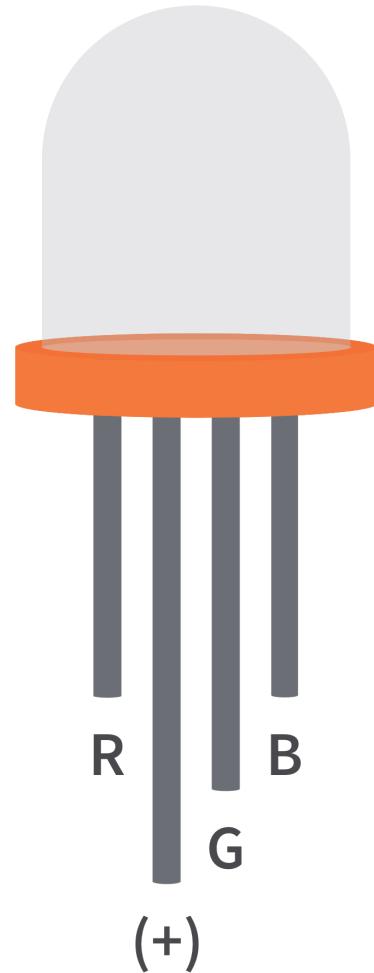
```
while True:  
    for duty in range(0, 65535):      # Increasing LED broghtness  
        led_1.duty_u16(duty)  
        led_2.duty_u16(duty)  
        led_3.duty_u16(duty)  
        led_4.duty_u16(duty)  
        led_5.duty_u16(duty)  
        led_6.duty_u16(duty)  
        led_7.duty_u16(duty)  
        led_8.duty_u16(duty)  
        led_9.duty_u16(duty)  
        led_10.duty_u16(duty)  
        led_11.duty_u16(duty)  
        led_12.duty_u16(duty)  
        led_13.duty_u16(duty)  
        led_14.duty_u16(duty)  
        led_15.duty_u16(duty)  
        led_16.duty_u16(duty)  
  
        print(duty)                  # Print the duty Cycle  
        time.sleep(0.001)
```

```
for duty in range(65535, 0):      # deccresing LED brightness  
    led_1.duty_u16(duty)  
    led_2.duty_u16(duty)  
    led_3.duty_u16(duty)  
    led_4.duty_u16(duty)  
    led_5.duty_u16(duty)  
    led_6.duty_u16(duty)  
    led_7.duty_u16(duty)  
    led_8.duty_u16(duty)  
    led_9.duty_u16(duty)  
    led_10.duty_u16(duty)  
    led_11.duty_u16(duty)  
    led_12.duty_u16(duty)  
    led_13.duty_u16(duty)  
    led_14.duty_u16(duty)  
    led_15.duty_u16(duty)  
    led_16.duty_u16(duty)  
  
    print(duty)  
    time.sleep(0.001)
```

Try to complete the circuit connection with the help of code

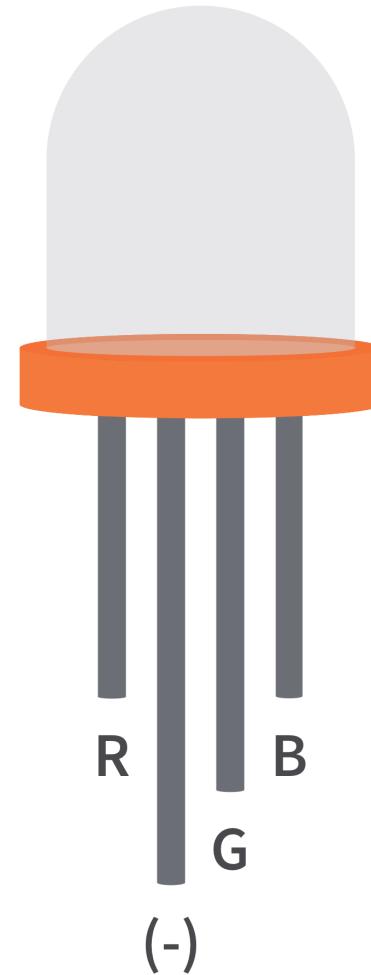
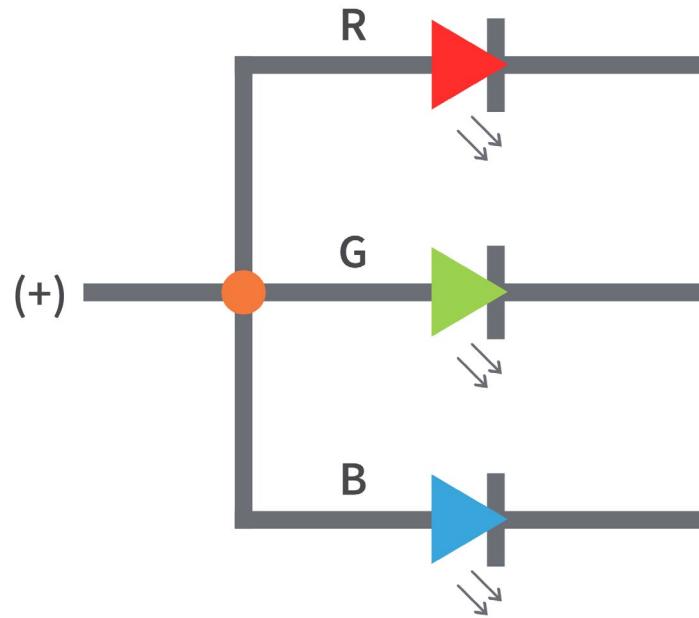
Colorful Light: How RGB LEDs work

RGB LED Types and Structure



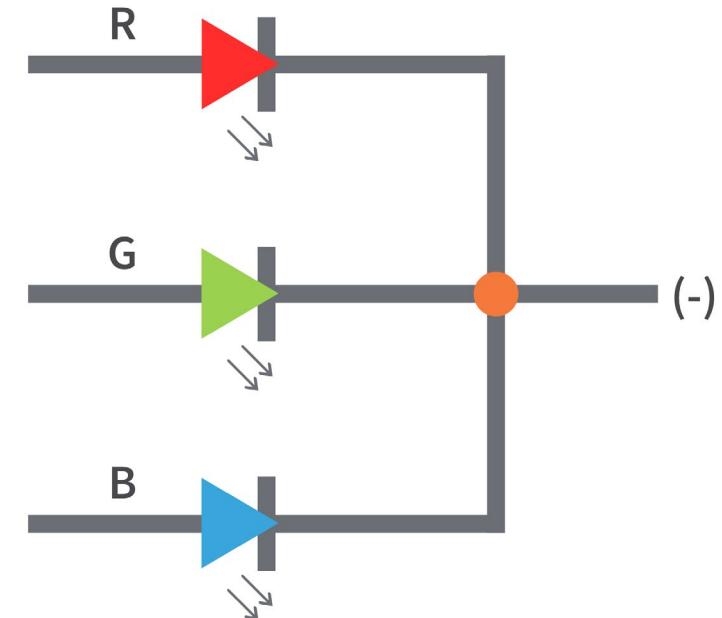
Common Anode

Common Anode RGB LED



Common Cathode

Common Cathode RGB LED



Colorful Light: Light can be superimposed

- Light can be superimposed. For example, mix **blue** light and **green** light give **cyan** light, **red** light and **green** light give **yellow** light. This is called “The additive method of color mixing”.
- Based on this method, we can use the three primary colors to mix the visible light of any color according to different specific gravity. For example, **orange** can be produced by **more red** and **less green**.
- **COMMON CATHODE RGB LED** is equivalent to encapsulating Red LED, Green LED, Blue LED under one lamp cap, and the three LEDs share one cathode pin. Since the electric signal is provided for each anode pin, the light of the corresponding color can be displayed. By changing the electrical signal intensity of each anode, it can be made to produce various colors.

Colorful Light: Alternately flashing red, green, and blue LEDs — RGB

WOKWI SAVE SHARE Docs B

main.py • diagram.json • PIO 🐍

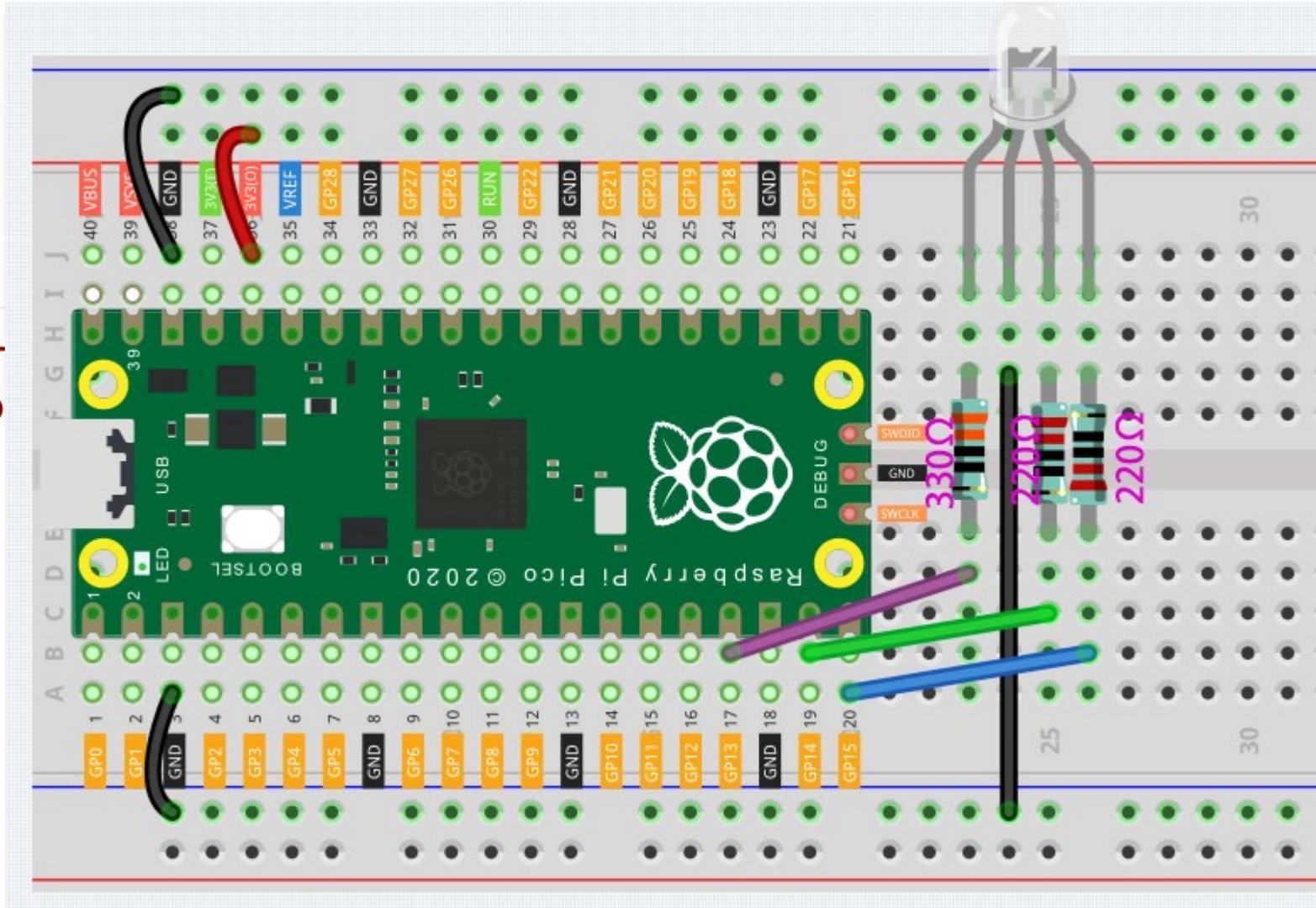
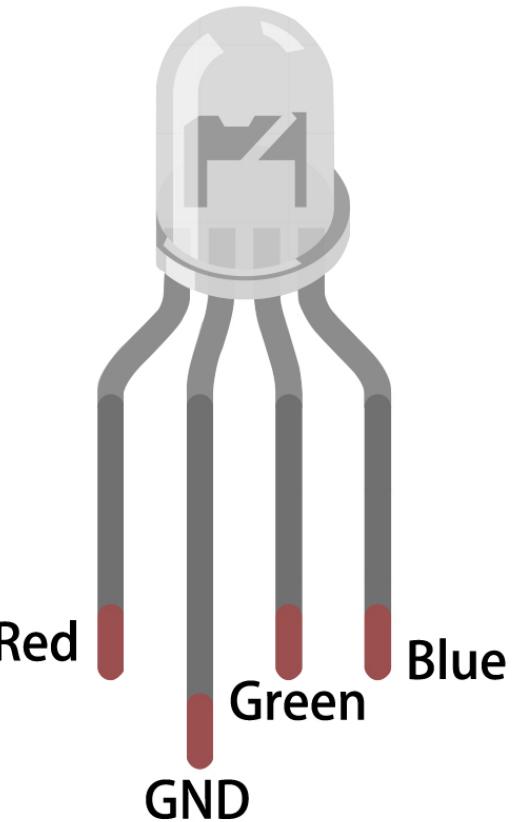
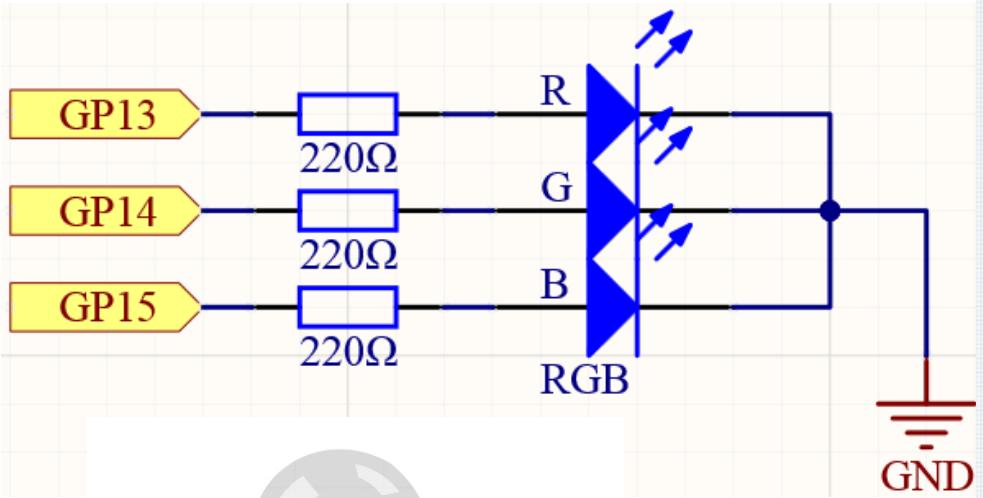
```
1  from machine import Pin
2  from utime import sleep
3
4  red = Pin(13, Pin.OUT)
5  green = Pin(14, Pin.OUT)
6  blue = Pin(15, Pin.OUT)
7
8  #Clear common cathode RGB LED
9
10 red.value(0)
11 green.value(0)
12 blue.value(0)
13
14 while True:
15     red.toggle()
16     sleep(1)
17     red.toggle()
18
19     green.toggle()
20     sleep(1)
21     green.toggle()
22
23     blue.toggle()
24     sleep(1)
25     blue.toggle()
26
27     sleep(1)
28
```

Simulation

Raspberry Pi Pico © 2020

00:26.763 100%

Colorful Light: Light can be superimposed



Colorful Light: Mystery of additive color mixing

WOKWI Docs

main.py diagram.json **rgb.py** ● PIO

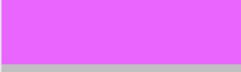
```
1  from machine import Pin, PWM
2  from utime import sleep
3
4  red = PWM(Pin(13))
5  green = PWM(Pin(14))
6  blue = PWM(Pin(15))
7
8  red.freq(1000)
9  green.freq(1000)
10 blue.freq(1000)
11
12 def interval_mapping(x, in_min, in_max, out_min, out_max):
13     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
14
15 def color_to_duty(rgb_value):
16     rgb_value = int(interval_mapping(rgb_value,0,255,0,65535))
17     return rgb_value
18
19 def color_set(red_value,green_value,blue_value):
20     red.duty_u16(color_to_duty(red_value))
21     green.duty_u16(color_to_duty(green_value))
22     blue.duty_u16(color_to_duty(blue_value))
23
24 color_set(255,128,0)
```

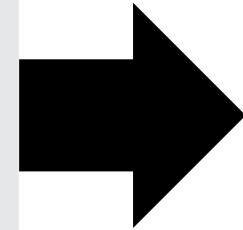
Change these values in THONNY and see the magic

Simulation

00:10.048 99%

Colorful Light: Mystery of additive color mixing

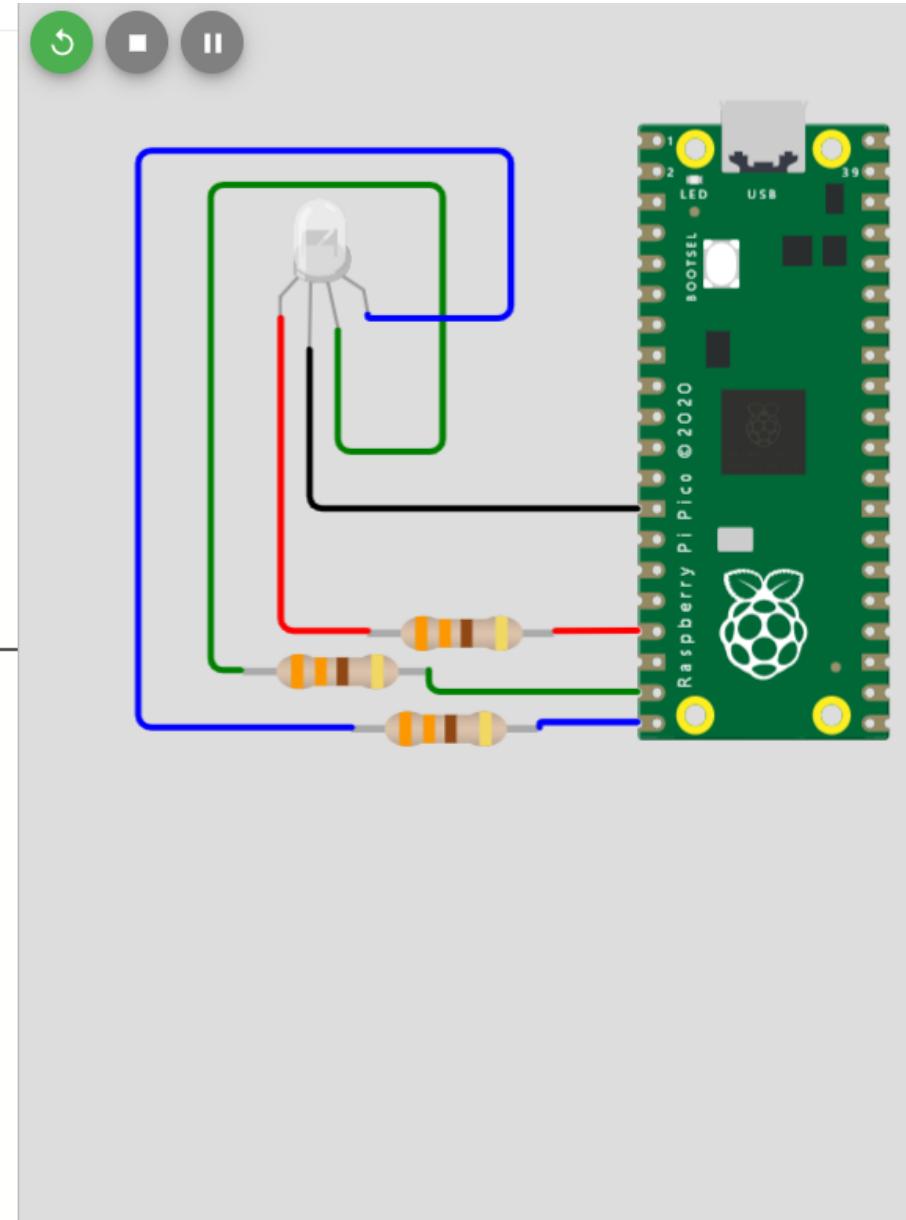
Color	HTML/CSS NAME	Decimal (R,G,B)
	Black	(0,0,0)
	White	(255,255,255)
	Red	(255,0,0)
	Lime	(0,255,0)
	Blue	(0,0,255)
	Yellow	(255,255,0)
	Cyan	(0,255,255)
	Magenta	(255,0,255)
	Silver	(192,192,192)
	Grey	(128,128,128)
	Maroon	(128,0,0)
	Olive	(128,128,0)
	Green	(0,128,0)
	Purple	(128,0,128)
	Teal	(0,128,128)
	Navy	(0,0,128)



Check these colours in your
RGB LED

Colorful Light: Mystery of additive color mixing

```
1 from machine import Pin, PWM
2 from utime import sleep
3
4 red = PWM(Pin(13))
5 green = PWM(Pin(14))
6 blue = PWM(Pin(15))
7
8 red.freq(1000)
9 green.freq(1000)
10 blue.freq(1000)
11
12 def interval_mapping(x, in_min, in_max, out_min, out_max):
13     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
14
15 def color_to_duty(rgb_value):
16     rgb_value = int(interval_mapping(rgb_value,0,255,0,65535))
17     return rgb_value
18
19 def color_set(red_value,green_value,blue_value): #let the three primary colors work together.
20     red.duty_u16(color_to_duty(red_value))
21     green.duty_u16(color_to_duty(green_value))
22     blue.duty_u16(color_to_duty(blue_value))
23
24 while True:
25     color_set(255,0,0)
26     sleep(1)
27     color_set(0,255,0)
28     sleep(1)
29     color_set(0,0,255)
30     sleep(1)
31     color_set(0,255,255)
32     sleep(1)
33     color_set(255,255,0)
34     sleep(1)
35     color_set(255,255,255)
36     sleep(1)
37     color_set(128,0,0)
38     sleep(1)
39     color_set(128,128,0)
40     sleep(1)
```



Custom Tone: Create a Melody

Buzzers

- Electronic buzzers create sound.
- Buzzers can be categorized into two different types – active buzzers and passive buzzers.
- An active buzzer has a built-in oscillator so it can produce sound with only a DC power supply.
- A passive buzzer does not have a built-in oscillator, so it needs an AC audio signal to produce sound.
- An active buzzer has a built-in oscillator so it can produce sound with only a DC power supply.
- A passive buzzer does not have oscillating source, so it will not beep if DC signals are used. But this allows the passive buzzer to adjust its own oscillation frequency and can emit different notes

Active Buzzers Vs Passive Buzzers

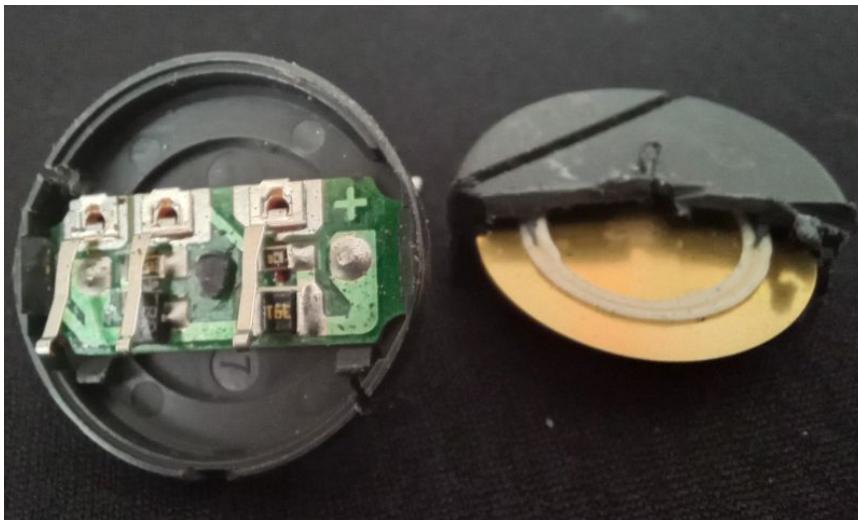
ACTIVE BUZZERS

Active buzzers are the simplest to use. They are typically available in voltages from **1.5V to 24V**. All you need to do is apply a DC voltage to the pins and it will make a sound.

Active buzzers have **polarity**. The polarity is the same as an LED and a capacitor – the longer pin goes to positive. One **downside** of active buzzers is that the **frequency of the sound is fixed** and **cannot be adjusted**.

Custom Tone: Create a Melody

Inside an ACTIVE BUZZERS



The gold disc on the right is a piezoelectric disk that vibrates when a voltage is applied. The three metal fingers on the PCB make contact with the disc, and a little transistor amplifier is on the printed circuit board.

PASSIVE BUZZERS

Passive buzzers need an AC signal to produce sound. the downside to this is that they will need more complex circuitry to control them, like an oscillating 555 timer .

Passive buzzers have the advantage that they can vary the pitch or tone of the sound. Passive buzzers can be programmed to emit a wide range of frequencies or musical notes.

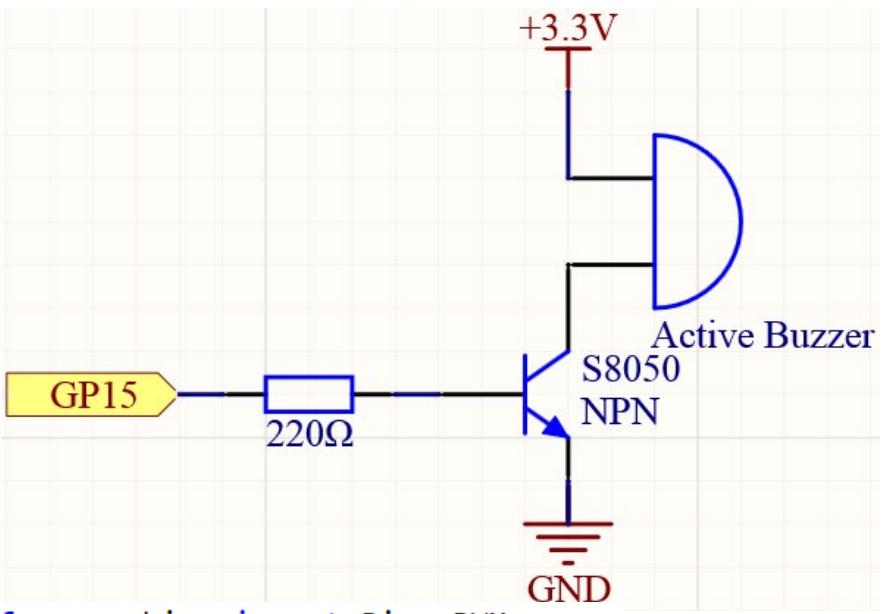
Custom Tone: Create a Melody

Inside an PASSIVE BUZZERS

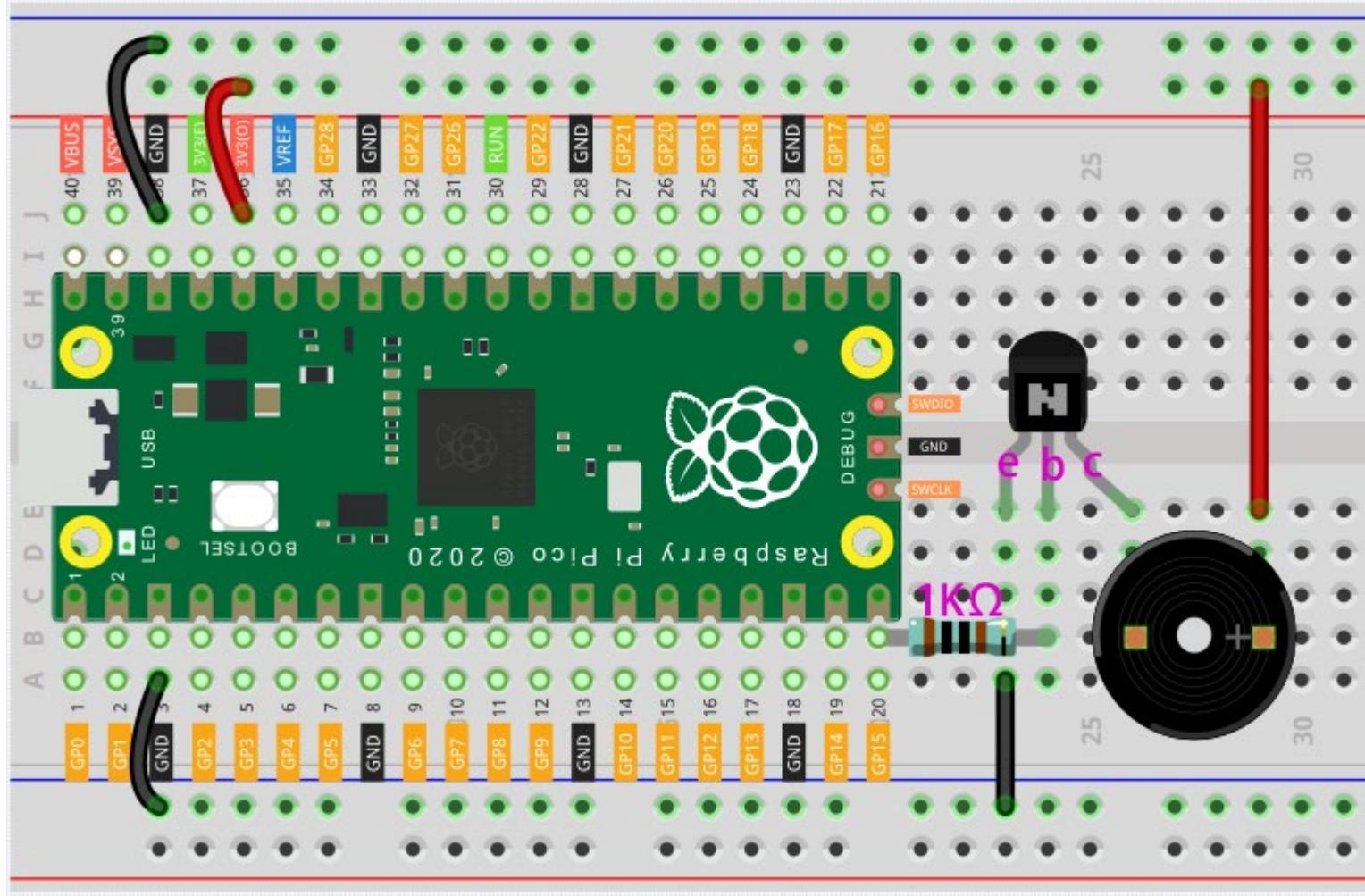


In a passive buzzer, we find an anatomy similar to a loudspeaker. There is a circular magnet surrounding an inner wire coil, with a disk that vibrates from the magnetic force generated by the electromagnetic coil.

Custom Tone: Create a Melody



```
from machine import Pin, PWM  
import utime  
  
buzzer = PWM(Pin(15))  
  
def tone(pin,frequency,duration):  
    pin.freq(frequency)  
    pin.duty_u16(30000)  
    utime.sleep_ms(duration)  
    pin.duty_u16(0)  
  
tone(buzzer,440,250)  
utime.sleep_ms(500)  
tone(buzzer,494,250)  
utime.sleep_ms(500)  
tone(buzzer,523,250)
```

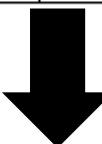


Check it in THONNY

Custom Tone: Classic Happy Birthday Melody

The frequencies of the musical notes starting from middle C (i.e. C4) are given below. To play the tune of a melody, we need to know its musical notes. Each note is played for certain duration and there is a certain time gap between two successive notes.

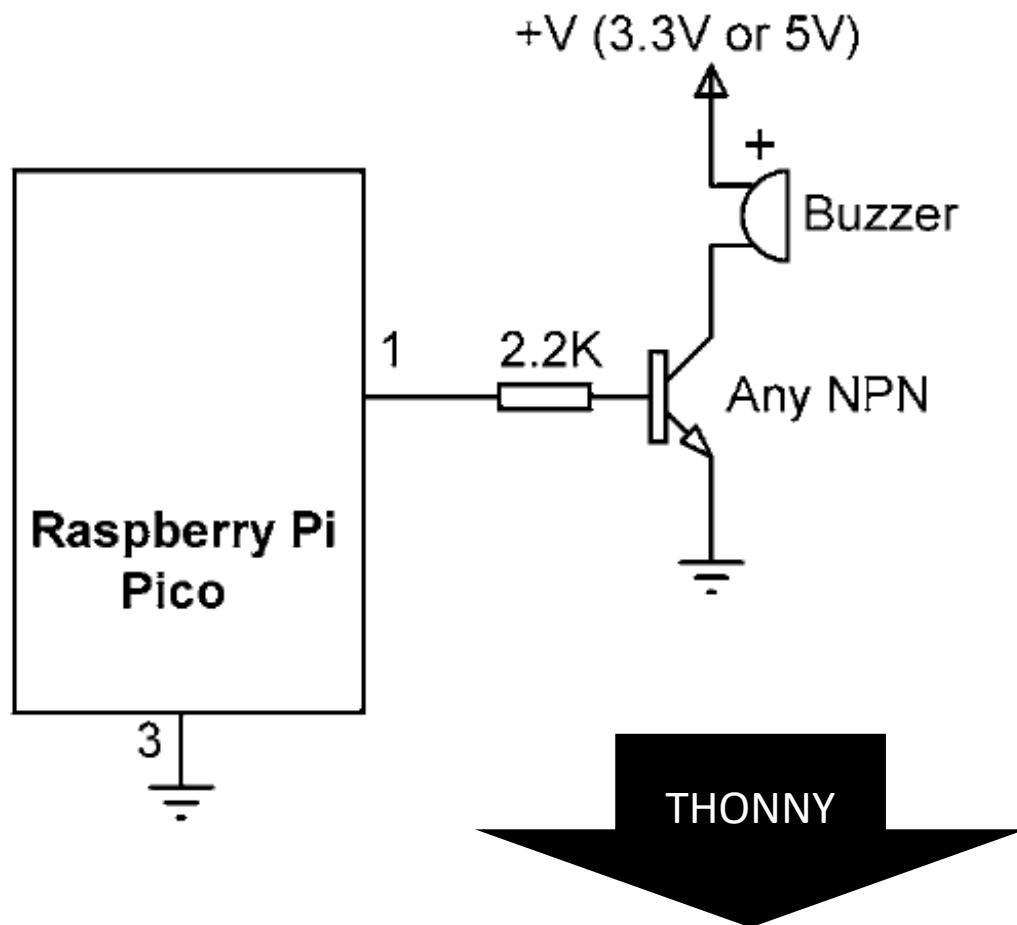
Notes	C ₄	C ₄ #	D ₄	D ₄ #	E ₄	F ₄	F ₄ #	G ₄	G ₄ #	A ₄	A ₄ #	B ₄
Hz	261.63	277.18	293.66	311.13	329.63	349.23	370	392	415.3	440	466.16	493.88



Note	C ₄	C ₄	D ₄	C ₄	F ₄	E ₄	C ₄	C ₄	D ₄	C ₄	G ₄	F ₄	C ₄
Duration	1	1	2	2	2	3	1	1	2	2	2	3	1

Note	C ₄	C ₅	A ₄	F ₄	E ₄	D ₄	A ₄ #	A ₄ #	A ₄	F ₄	G ₄	F ₄
Duration	1	2	2	2	2	2	1	1	2	2	2	4

Custom Tone: Classic Happy Birthday Melody



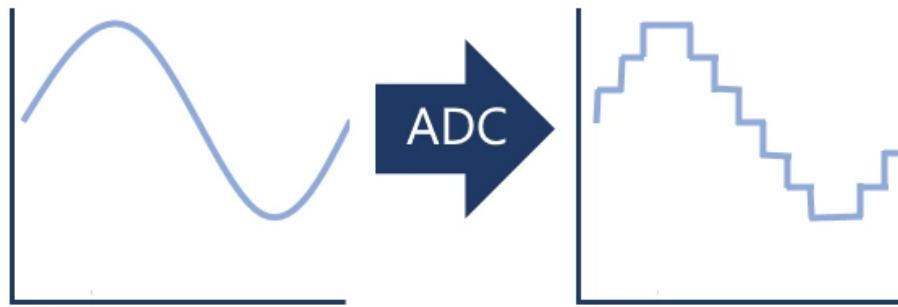
1. Change the durations between the notes and see its effects.
2. How can you make the melody run quicker?

```
from machine import Pin, PWM
import utime

ch = PWM(Pin(0))          # PWM output at GP0
MaxNotes = 25
Durations = [0]*MaxNotes
#
# Melody frequencies
#
frequency = [262,262,294,262,349,330,262,262,294,262,
392,349,262,262,524,440,349,330,294,466,
466,440,349,392,349]
#
# Frequency durations
#
duration = [1,1,2,2,2,3,1,1,2,2,2,3,1,1,2,2,2,2,
2,1,1,2,2,2,3]

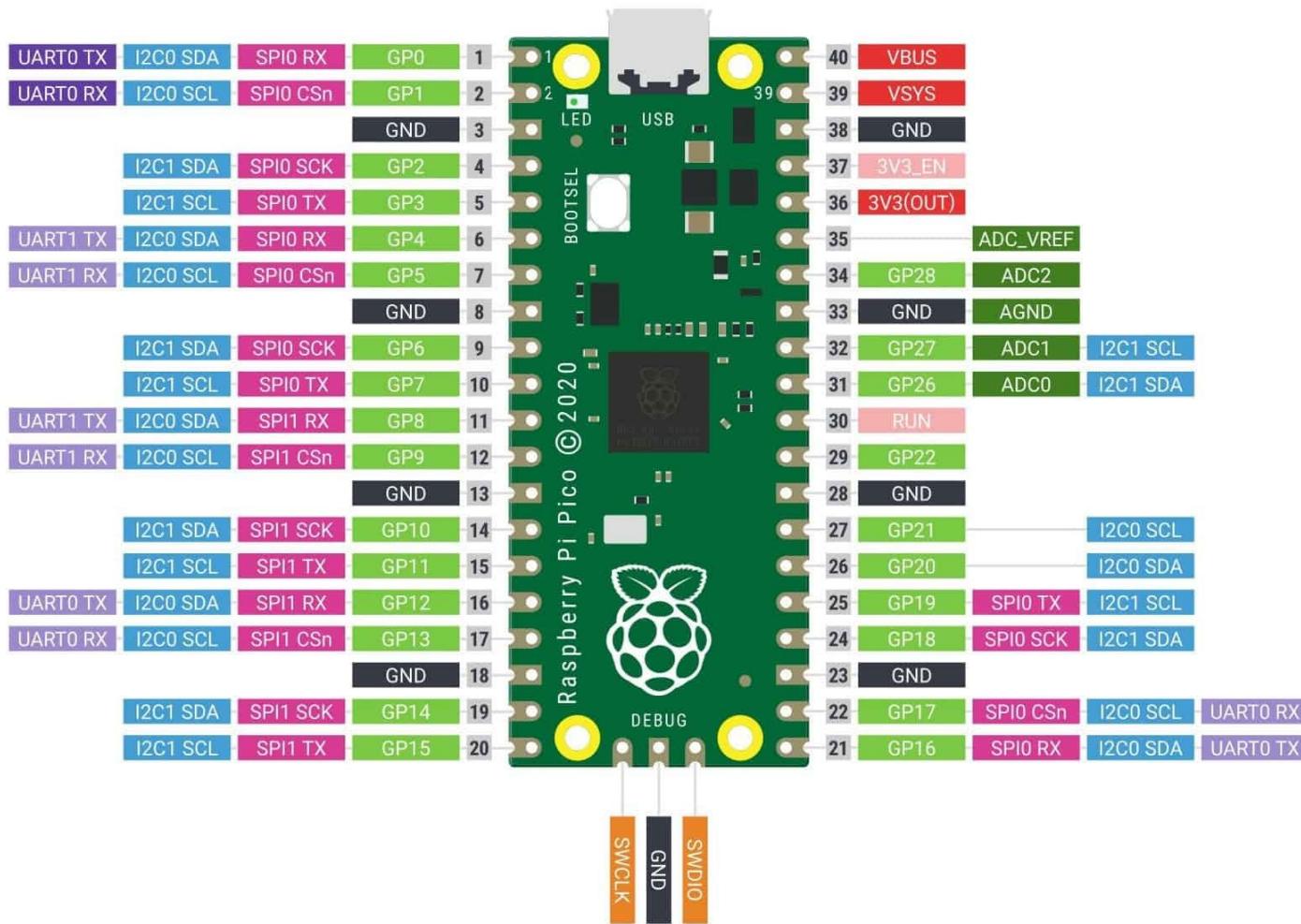
for k in range(MaxNotes):
    Durations[k] = 400 * duration[k]
while True:
    for k in range(MaxNotes):          # Do for all notes
        ch.duty_u16(32767)           # Duty cycle
        ch.freq(2*frequency[k])       # Play 2nd harmonics
        utime.sleep_ms(Durations[k])  # Durations
        utime.sleep_ms(100)            # Wait
        ch.duty_u16(0)                # Stop playing
        utime.sleep(3)                # Stop 3 seconds
```

What is Analog to Digital Converter (ADC)?



- An Analog to Digital Converter (ADC) is a very useful feature that **converts an analog voltage on a pin to a digital number**. By converting from the analog world to the digital world, we can begin to use electronics to interface with the analog world around us.
- An analog to digital converter (ADC) is a circuit that converts a continuous voltage value (analog) to a binary value (digital) that can be understood by a digital device which could then be used for digital computation.
- The Raspberry Pi Pico board exposes 26 multi-function GPIO pins from a total of 36 GPIO pins. Out of 36 GPIO Pins, there are 4 ADC pins but only 3 are usable.
- The **ADC in Raspberry Pi Pico is 12bits**, which is 4 times better than the 10 bits ADC of the Arduino. We will write a MicroPython code to learn how we can use the ADC pin value with any analog sensors. A potentiometer is the best tool to vary the input Analog Voltage.
- With the ADC, we can sense environmental parameters like light, sound, distance, gravity, acceleration, rotation, smell, gases, other particles.
- Most microcontrollers nowadays have built-in ADC converters. It is also possible to connect an external ADC converter to any type of microcontroller. ADC converters are usually 10 or 12 bits, having 1024 to 4096 quantization levels. **A 16 bits ADC has 65536 quantization levels**. **A Raspberry Pi Pico has 12 Bits ADC with a quantization level of 4096**.

How does an ADC work in a Microcontroller?



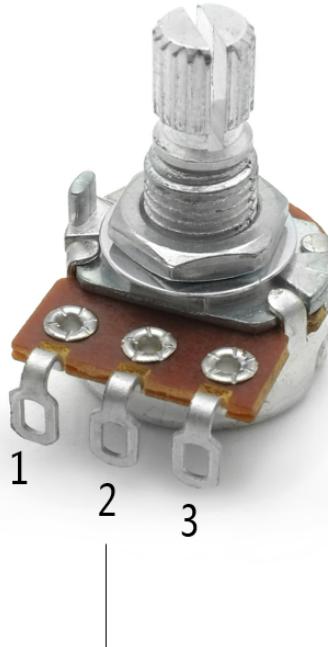
ADC Module	GPIO Pins
ADC0	GP26
ADC1	GP27
ADC2	GP28

- The Raspberry Pi Pico supports four **12-bit SAR** based analog to digital converters. Out of the 4, you can only use 3 analog channels.
- The 4th analog channel is internally connected to the **internal temperature sensor**. You can measure the temperature using build-in temperature by reading the analog value of ADC4. with GP26, GP27 & GP28 pins respectively.
- The ADC conversion speed per sample is **2μs** that is **500kS/s**. The RP2040 microcontroller operates on a 48MHz clock frequency which comes from USB PLL. So, its ADC takes a 96 CPU clock cycle to perform one conversion. Therefore, the sampling frequency is $(96 \times 1 / 48\text{MHz}) = 2 \mu\text{s}$ per sample (500kS/s).

Turn the Knob: Potentiometer/POT

A potentiometer (or pot) is a three-terminal resistor with a sliding or rotating contact that can be used to dynamically vary resistance.

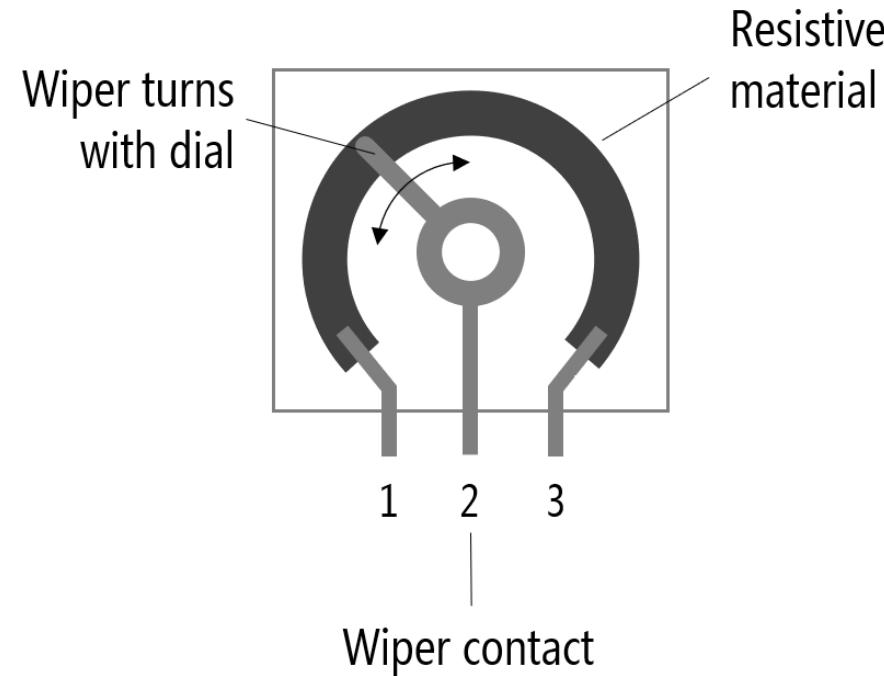
PANEL MOUNT POT 10kΩ



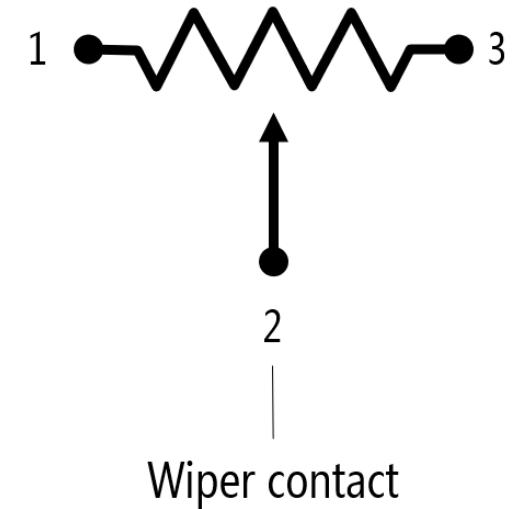
TRIM POT 10kΩ



FUNCTIONAL DIAGRAM



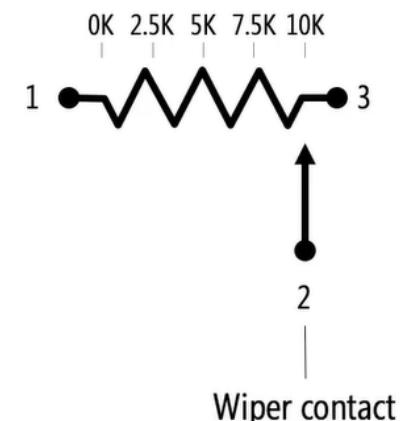
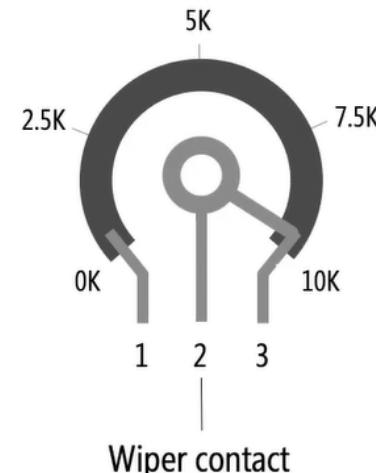
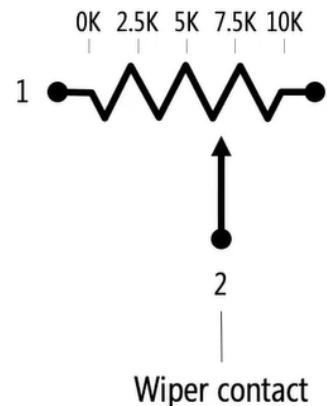
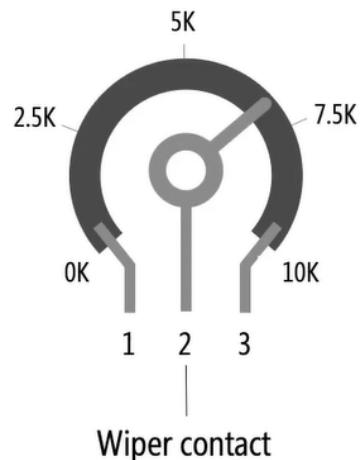
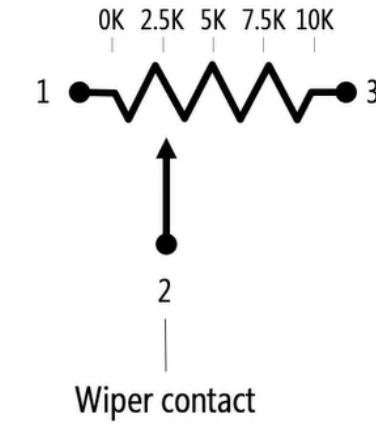
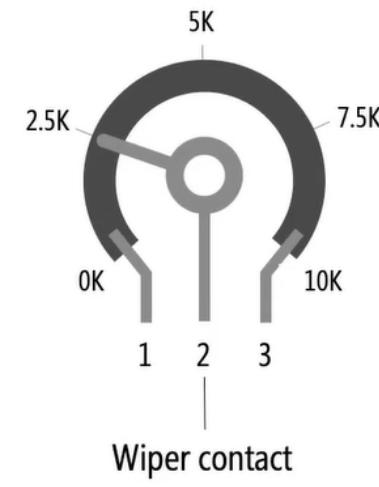
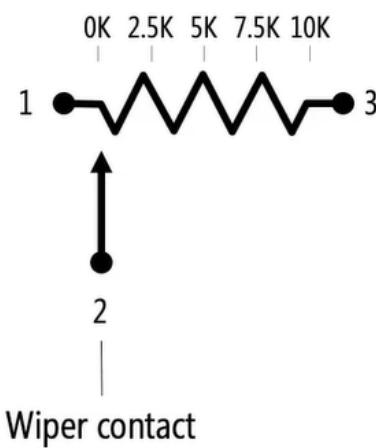
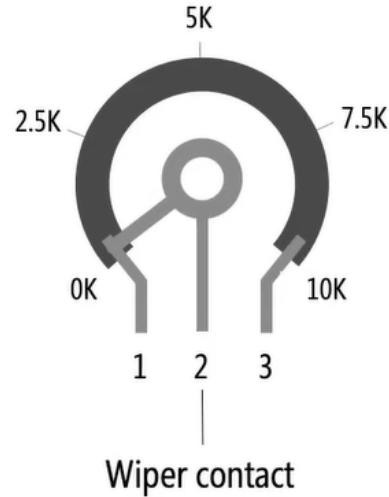
SCHEMATIC SYMBOL



Two example potentiometers commonly included in our hardware kits: a $10\text{k}\Omega$ panel mount and a $10\text{k}\Omega$ trim potentiometer.

Turn the Knob: Potentiometer/POT

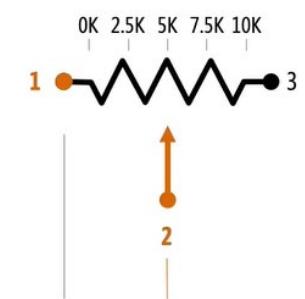
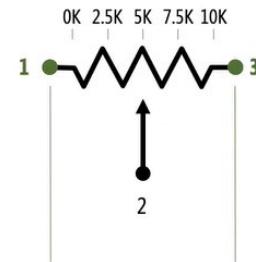
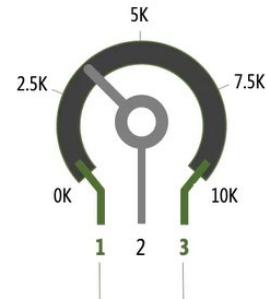
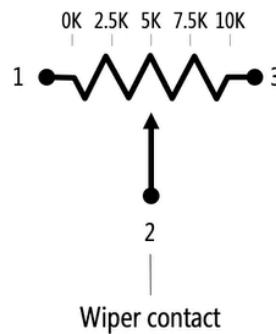
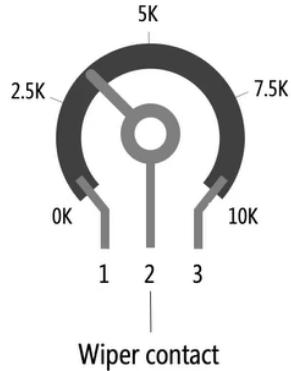
A potentiometer (or pot) is a three-terminal resistor with a sliding or rotating contact that can be used to dynamically vary resistance.



Turn the Knob: How does a potentiometer work?

Potentiometers have three legs: the resistance between the outer two legs (Leg 1 and Leg 3) will not vary. For example, if you are using a $10\text{k}\Omega$ potentiometer, then the resistance between Legs 1 and 3 will always be $10\text{k}\Omega$ regardless of wiper position (Leg 2).

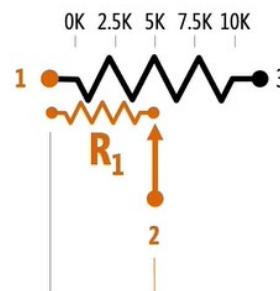
Here we have a 10 kilo ohm Potentiometer



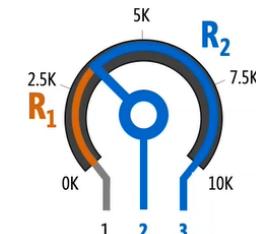
Regardless of wiper position, note that (R_{tot}) between Legs 1 and 3 is always $10\text{k}\Omega$ (this is a $10\text{k}\Omega$ potentiometer)

The resistance between Legs 1 and 2 changes depending on wiper position.

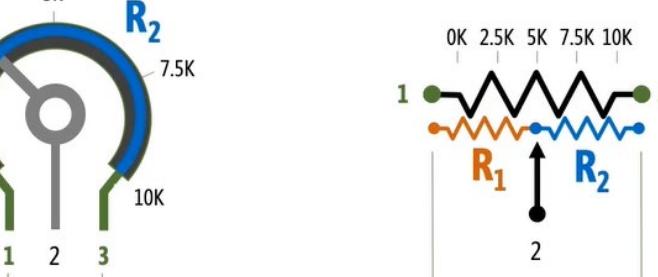
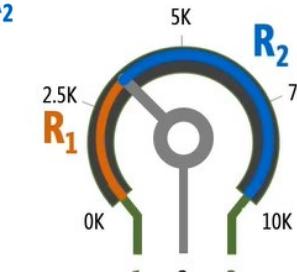
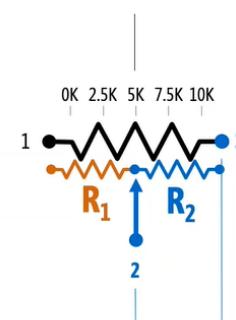
This circuit is known as a **voltage divider** because the voltage at Leg 2 is divided between R_1 and R_2



The resistance between Legs 1 and 2 changes depending on wiper position. Let's call this R_1 .



Simultaneously, the wiper also affects the resistance between Legs 2 and 3. Let's call this R_2 .

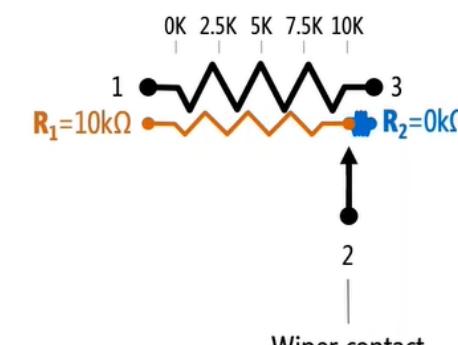
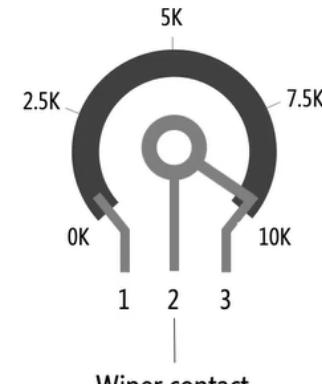
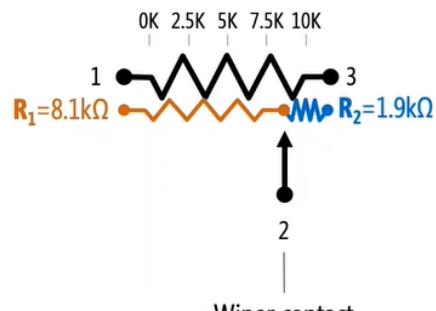
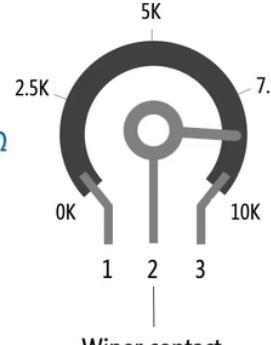
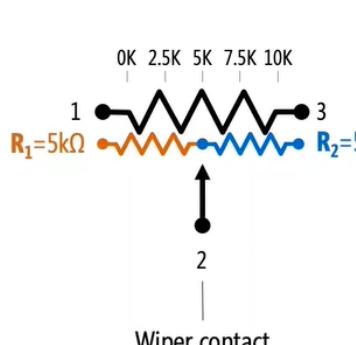
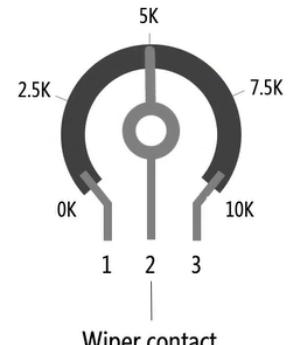
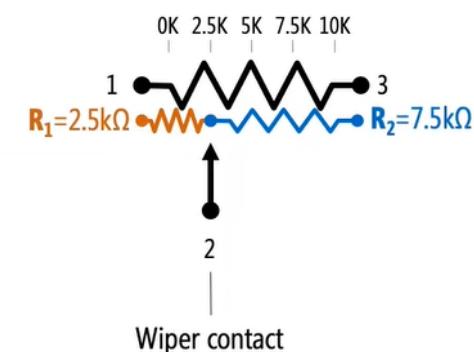
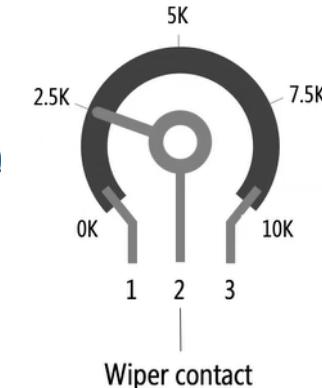
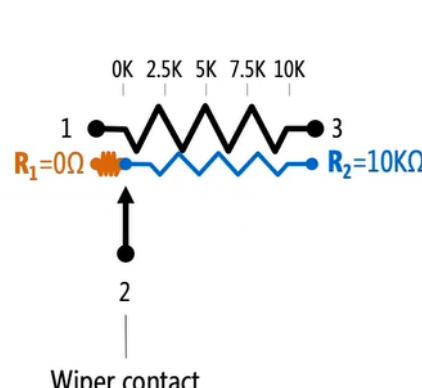
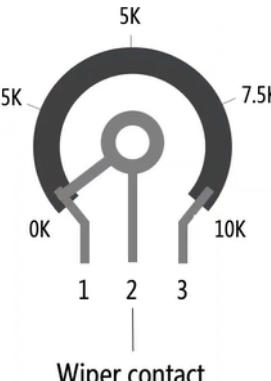
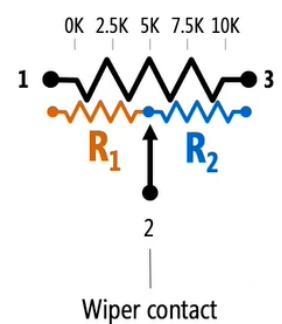
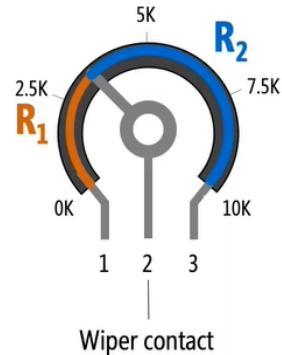


And that R_{tot} always equals $R_1 + R_2$

Turn the Knob: How does a potentiometer work?

Potentiometers have three legs: the resistance between the outer two legs (Leg 1 and Leg 3) will not vary. For example, if you are using a $10\text{k}\Omega$ potentiometer, then the resistance between Legs 1 and 3 will always be $10\text{k}\Omega$ regardless of wiper position (Leg 2).

Let's Try Few Wiper Positions



Turn the Knob: Welcome to Analog World

WOKWI SAVE SHARE Docs B

main.py • diagram.json • PIO 🐍

```
1 import machine
2 import utime
3
4 analog_value = machine.ADC(28)
5
6 while True:
7     reading = analog_value.read_u16()
8     print("ADC: ", reading)
9     utime.sleep(0.2)
10
```

Simulation

00:52.243 99%

THONNY

ADC: 65535
ADC: 65535
ADC: 65535
ADC: 65535
ADC: 65535

Turn the Knob: Control an LED using POT

WOKWi[®] SAVE SHARE Docs B

main.py • diagram.json • PIO

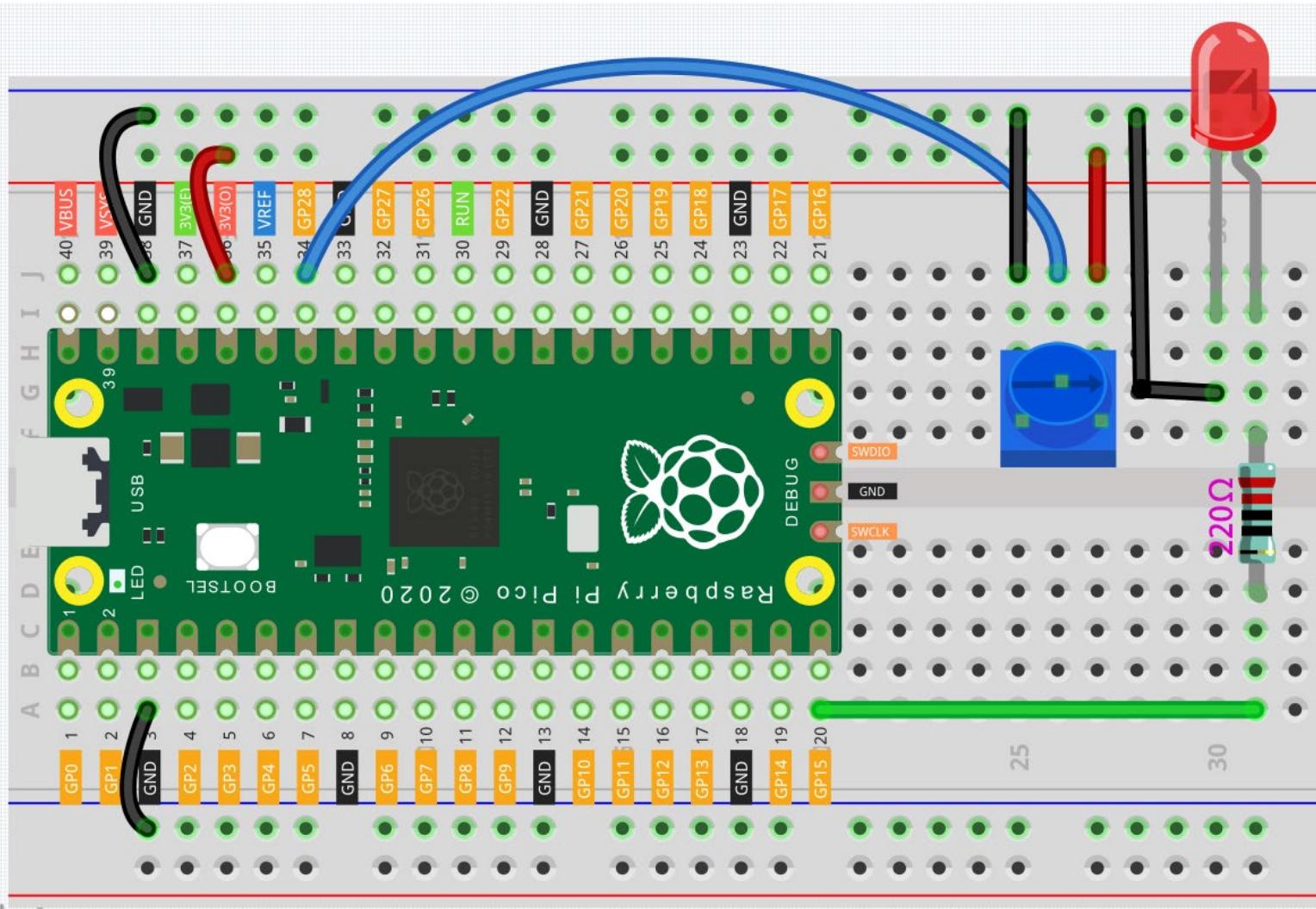
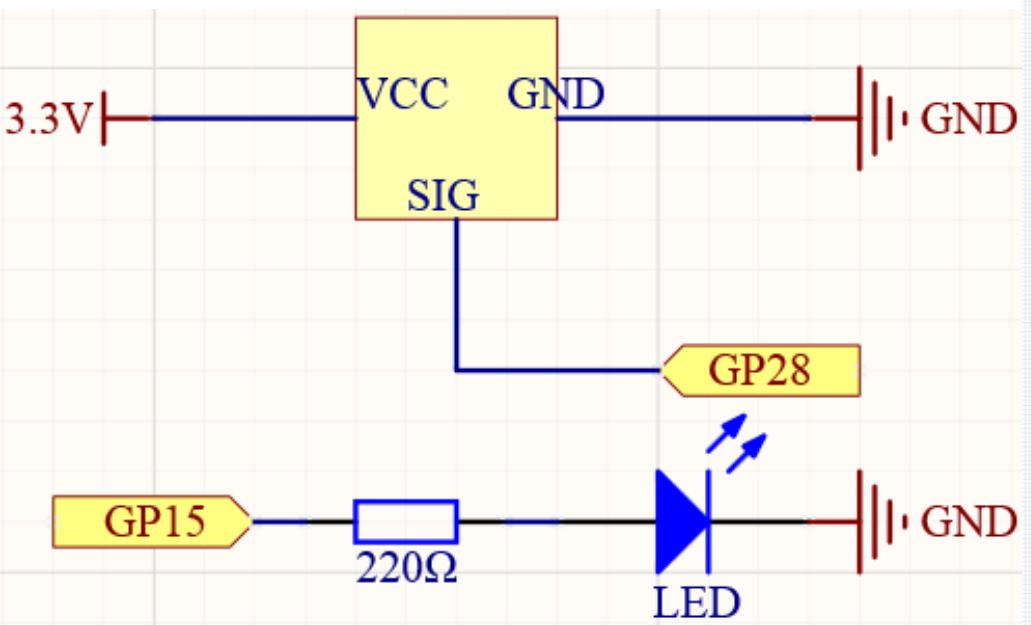
```
1 import machine
2 import utime
3
4 potentiometer = machine.ADC(28)
5 led = machine.PWM(machine.Pin(15))
6 led.freq(1000)
7
8 while True:
9     value=potentiometer.read_u16()
10    print(value)
11    led.duty_u16(value)
12    utime.sleep_ms(200)
```

Simulation

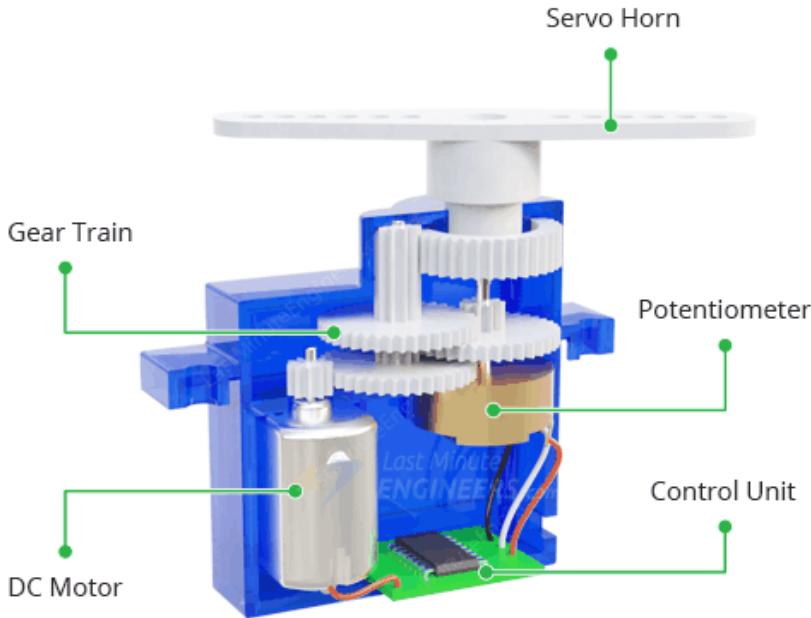
00:12.432 99%

8514
8514
8514
8514

Turn the Knob: Control an LED using POT



Swing the servo : What is a Servo and what makes it precise?



Servo motor is a component that can rotate its handle (usually between 0° and 180°). It used to control the angular position of the object.

Pinout

The servo motor used in this example includes three pins:

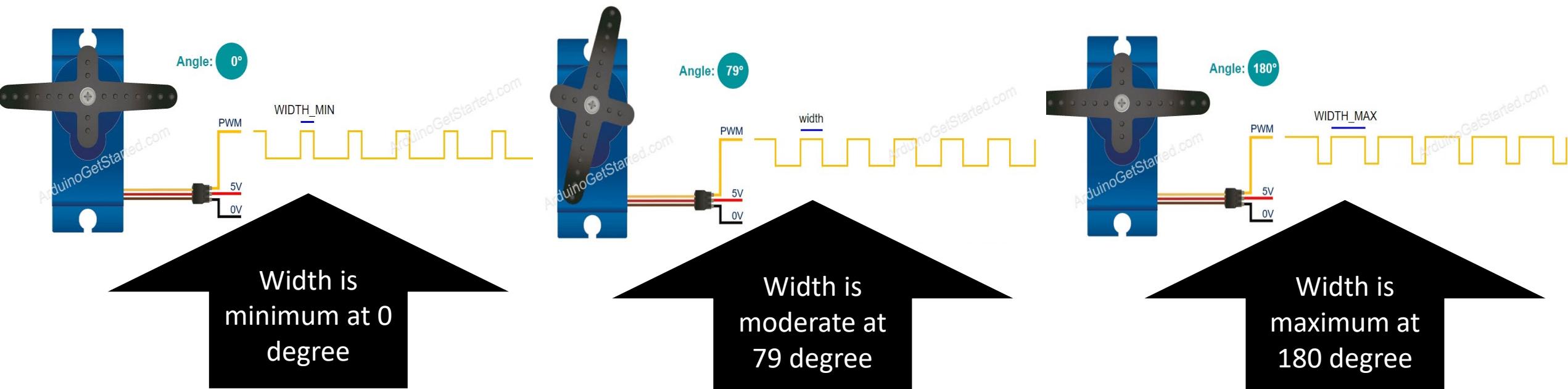
- ◆ **VCC pin:** (typically red) needs to be connected to **vcc** (5V)
- ◆ **GND pin:** (typically black or brown) needs to be connected to **GND** (0v)
- ◆ **Signal pin:** (typically yellow or orange) receives the PWM control signal from an

Swing the servo : Servo and Pulse Width

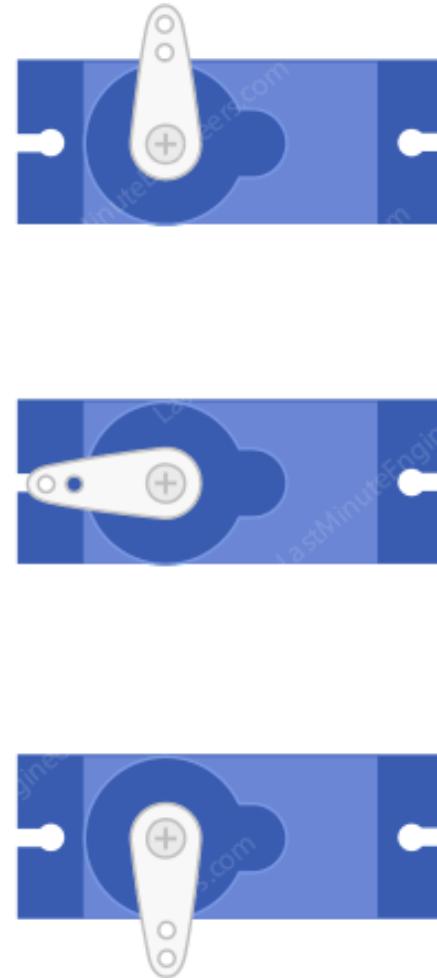
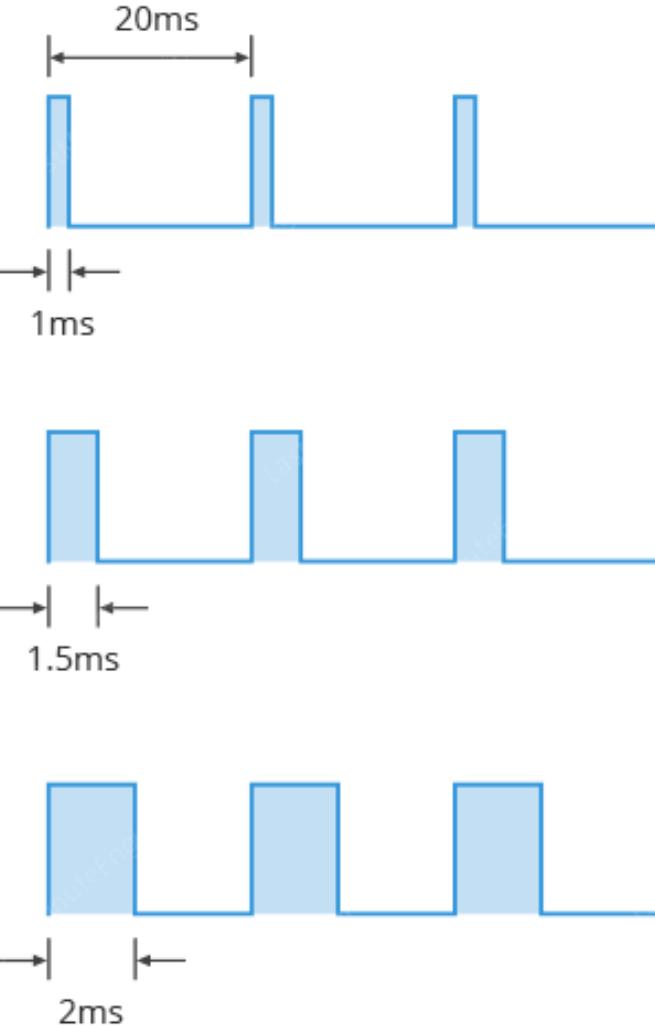
Servos are motors that allow us to **precisely control** physical movement because they generally move to a position rather than continuously rotating. They are simple to connect and control because the motor driver is built right into them.

Servos contain a small DC motor connected to the output shaft through gears. The output shaft drives a servo horn and is also linked to a potentiometer (pot).

The potentiometer provides position feedback to the error amplifier in the control unit, which compares the current position of the motor to the target position.



Swing the servo : Servo and Pulse Width



0 Degrees

A short pulse of 1 ms or less will rotate the servo to 0 degrees (one extreme).

90 Degrees

A pulse duration of 1.5 ms will rotate the servo to 90 degrees (middle position).

180 Degrees

A pulse duration of 2 ms or so will rotate the servo to 180 degrees (other extreme).

Swinging Servo

WOKWI

SAVE

SHARE

Docs

B

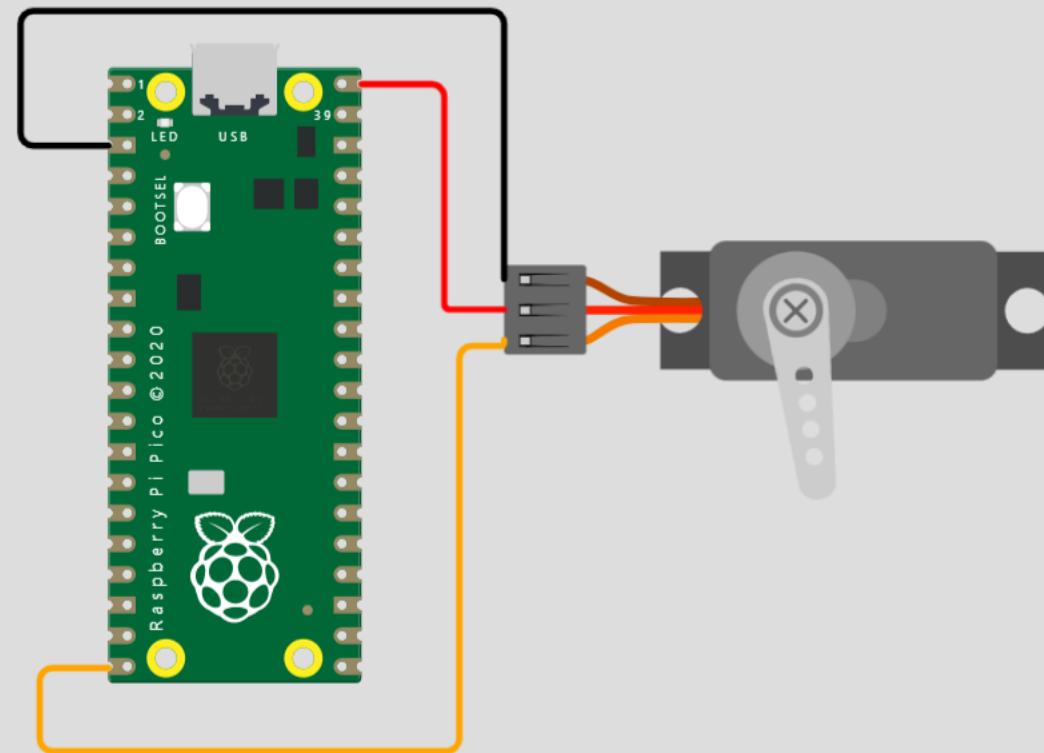
main.py • diagram.json • PIO 🚗

```
1  from machine import Pin, PWM
2  from utime import sleep_ms
3
4  servo = PWM(Pin(15))
5  servo.freq(50)
6
7  def interval_mapping(x, in_min, in_max, out_min, out_max):
8      return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
9
10 def servo_write(pin,angle):
11     pulse_width=interval_mapping(angle, 0, 180, 0.5,2.5)
12     duty=int(interval_mapping(pulse_width, 0, 20, 0,65535))
13     pin.duty_u16(duty)
14
15 while True:
16     for angle in range(180):
17         servo_write(servo,angle)
18         sleep_ms(20)
19     for angle in range(180,-1,-1):
20         servo_write(servo,angle)
21         sleep_ms(20)
22
```

Simulation

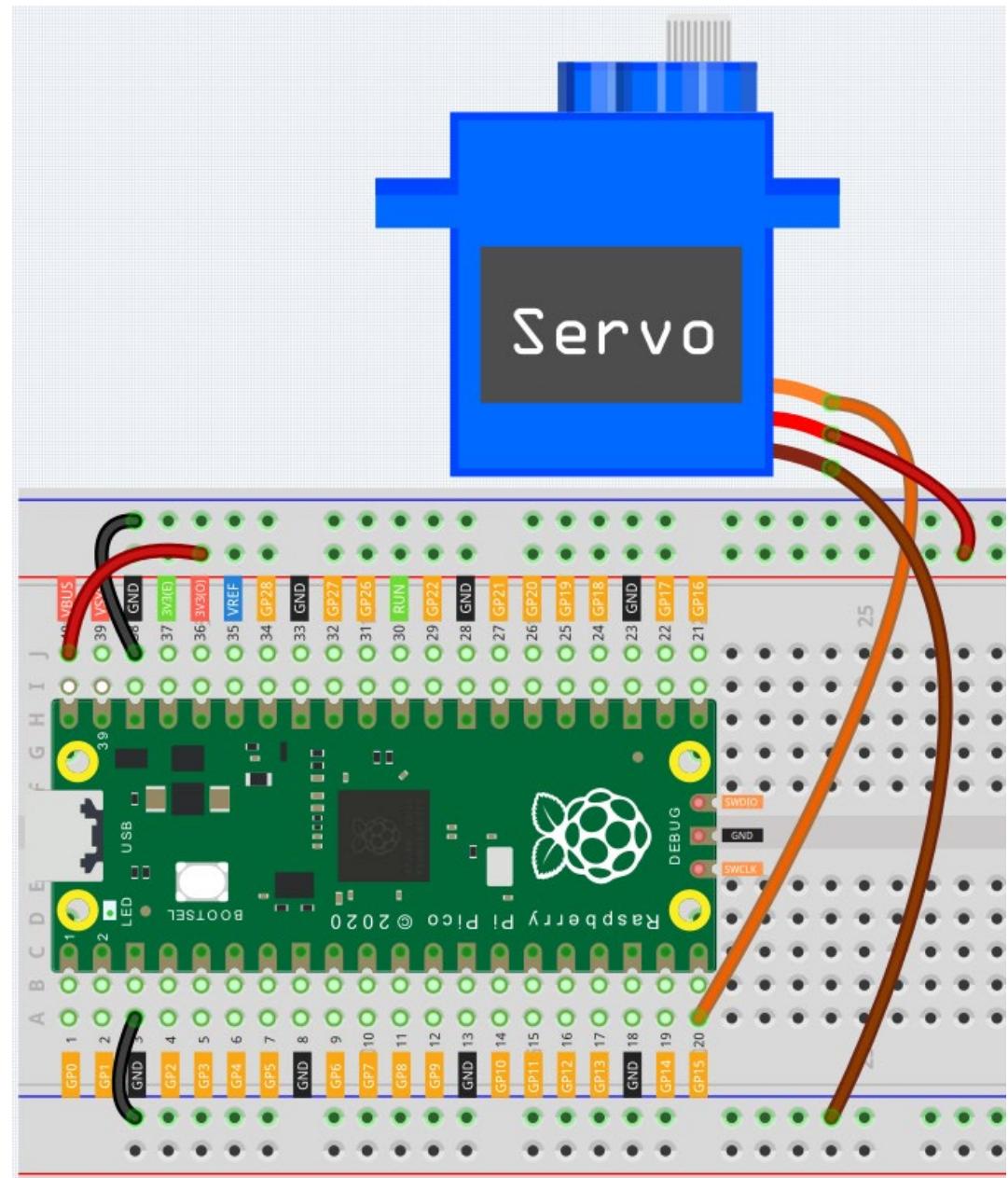
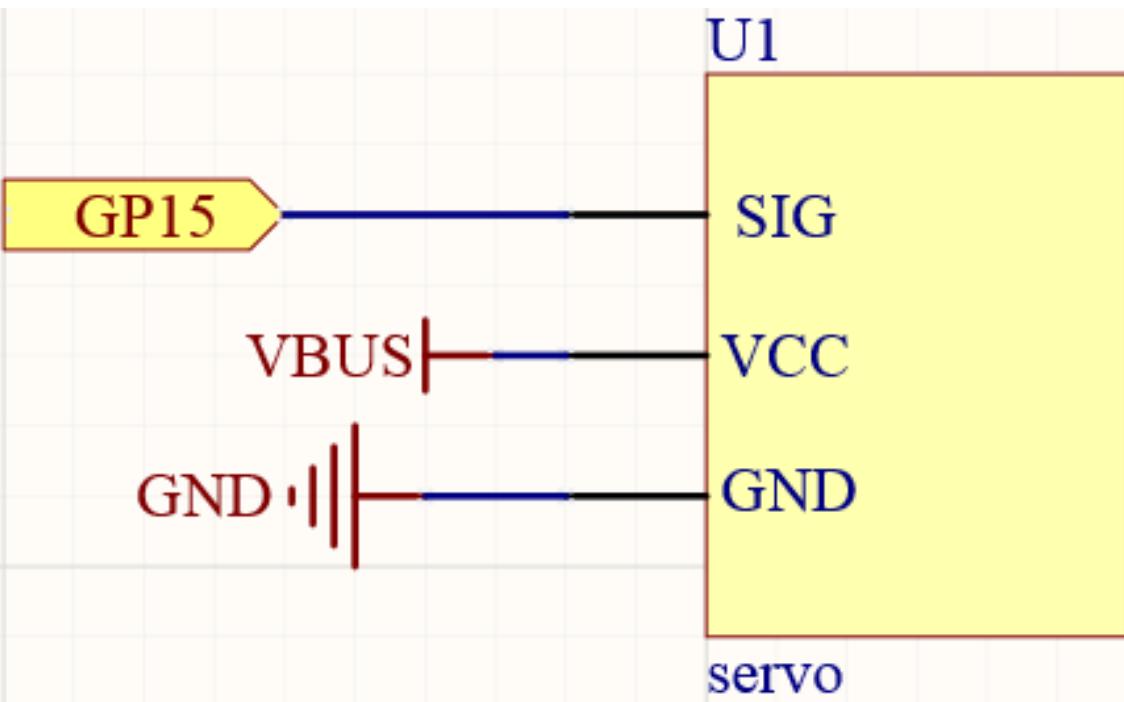


⌚ 00:25.778 ⚡ 99%



Swinging Servo

- Servo is a position (angle) servo device, which is suitable for those control systems that require constant angle changes and can be maintained. It has been widely used in high-end remote control toys, such as airplanes, submarine models, and remote control robots.



Swing the servo from left to right using POT

WOKWi

SAVE



SHARE

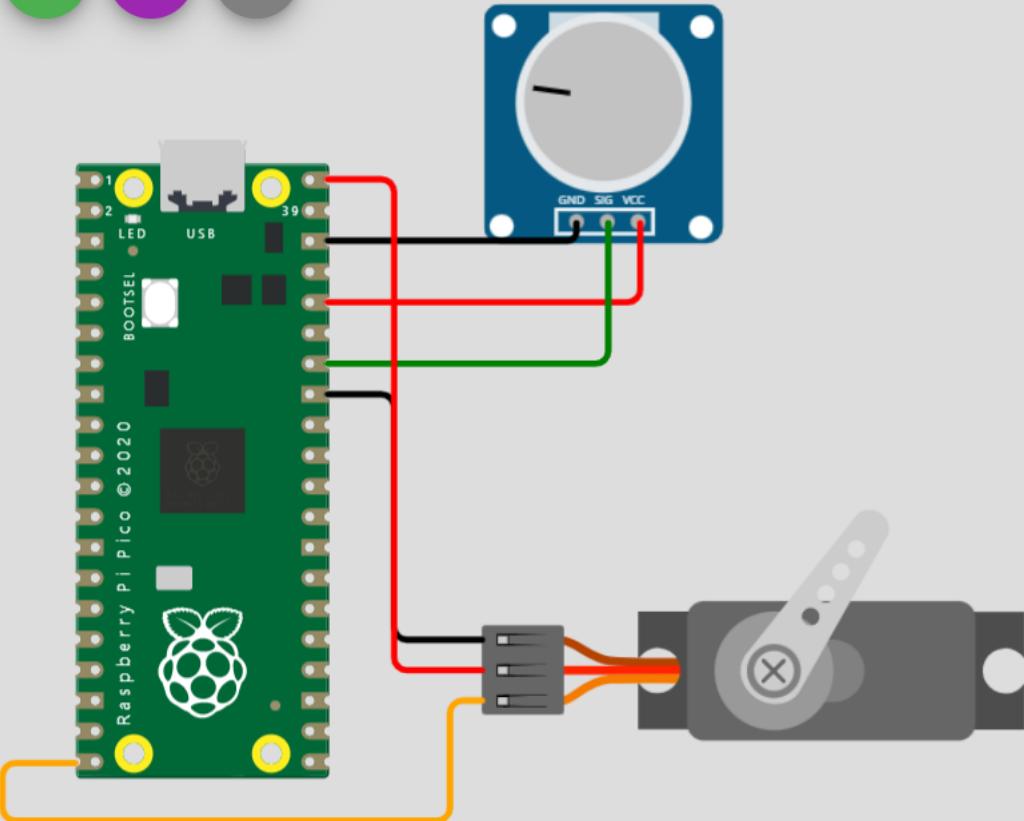
Docs

B

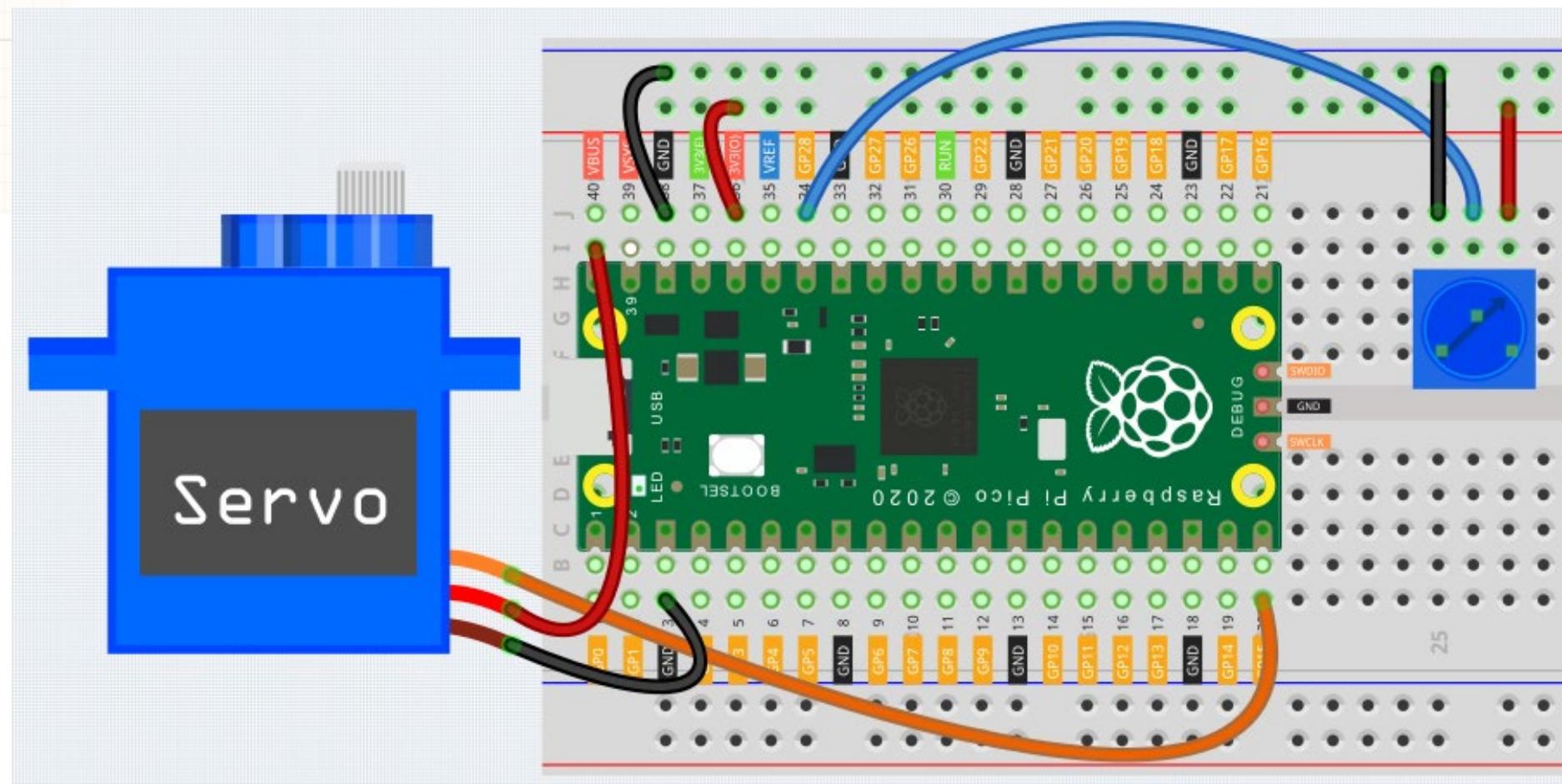
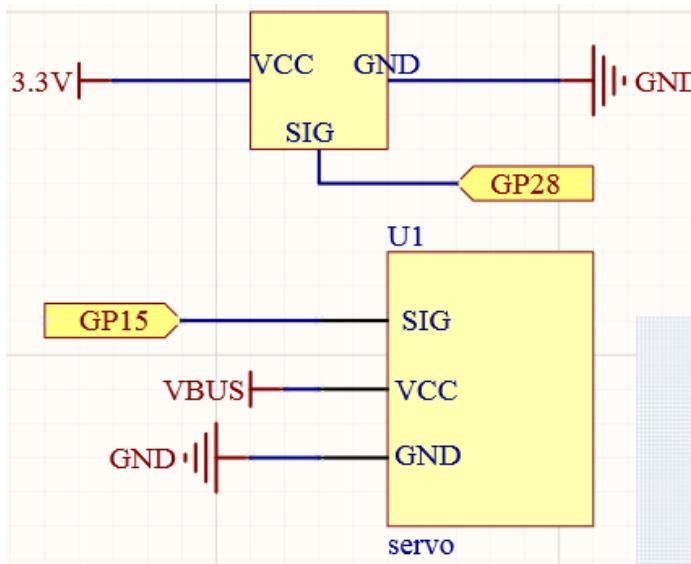
main.py ● diagram.json ●

```
1 import machine
2 import utime
3
4 potentiometer = machine.ADC(28)
5 servo = machine.PWM(machine.Pin(15))
6 servo.freq(50)
7
8 def interval_mapping(x, in_min, in_max, out_min, out_max):
9     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
10
11 def servo_write(pin,angle):
12     pulse_width=interval_mapping(angle, 0, 180, 0.5,2.5)
13     duty=int(interval_mapping(pulse_width, 0, 20, 0,65535))
14     pin.duty_u16(duty)
15
16 while True:
17     value=potentiometer.read_u16()
18     angle=interval_mapping(value,0,65535,0,180)
19     servo_write(servo,angle)
20     utime.sleep_ms(200)
21
```

Simulation



Swing the servo from left to right using POT



Thermometer: Let's Measure some temperature

Thermistors are **simple, inexpensive, and accurate** components that make it easy to get temperature data for your projects. Thermistors or **THERMally sensitive resISTORs** are **variable resistors**, whose **resistance will be changed with the temperature**. This feature enables us to read the temperature or change in temperature by measuring the resistance of the thermistors. Thermistors are used in various everyday use items like Thermostats, SMPS, surge protection circuits and rechargeable battery packs, etc.

Types of Thermistors

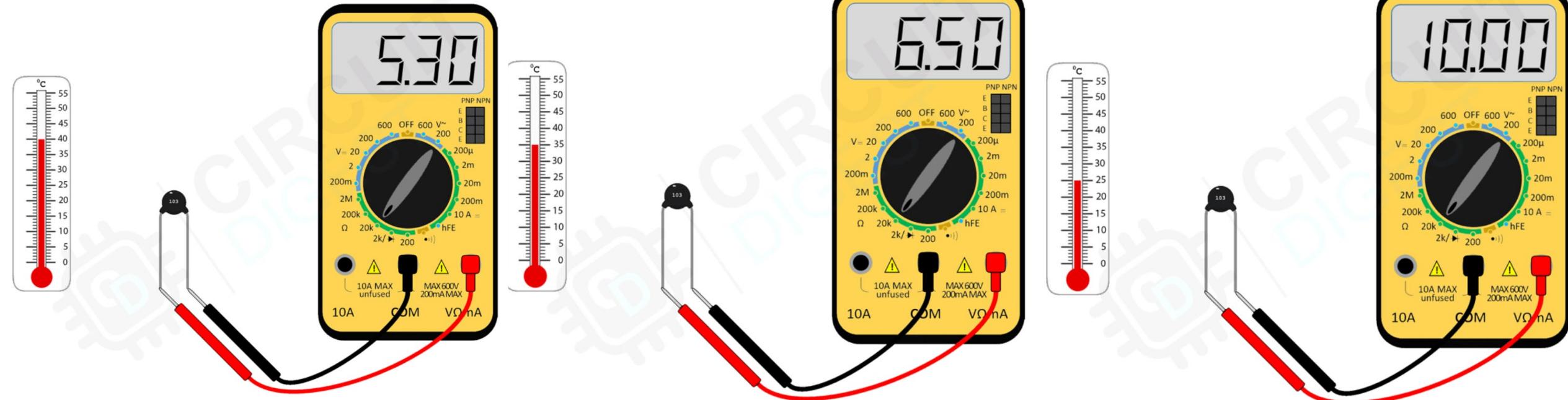
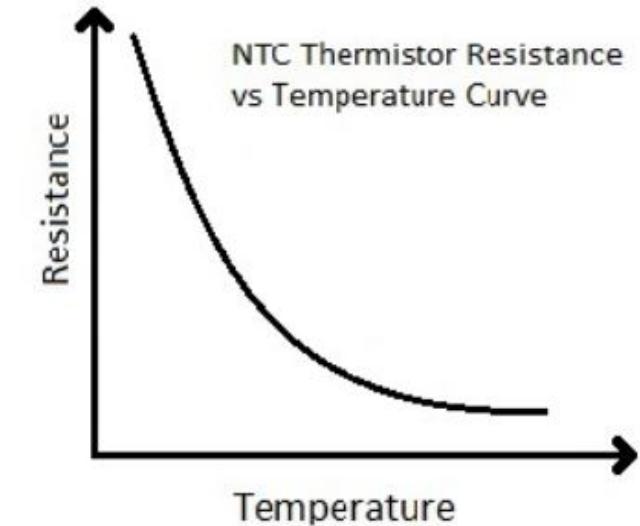
Depending on the materials used and how they react with the temperature thermistors are classified into **two types**:

- Negative Temperature Coefficient Thermistors or **NTC Thermistors**
- Positive Temperature Coefficient Thermistors or **PTC Thermistors**

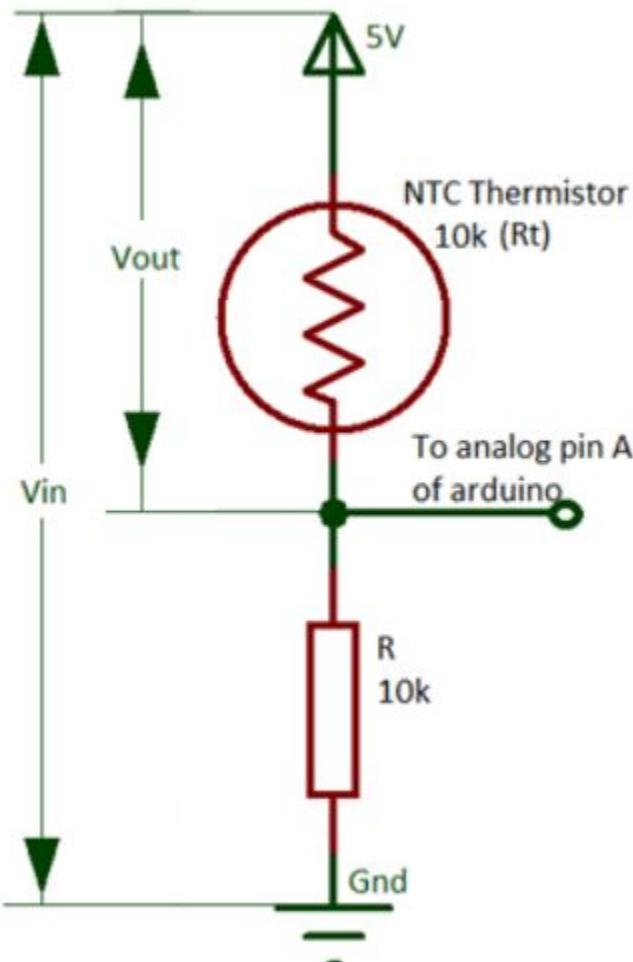
The resistance of PTC thermistor increases with temperature, while the condition of NTC is opposite to the former.

Thermometer: Let's Measure some temperature

NTC Thermistor Pinout



Thermometer: Calculation of Temperature



Get the value of V_{out} from analog pin Ao

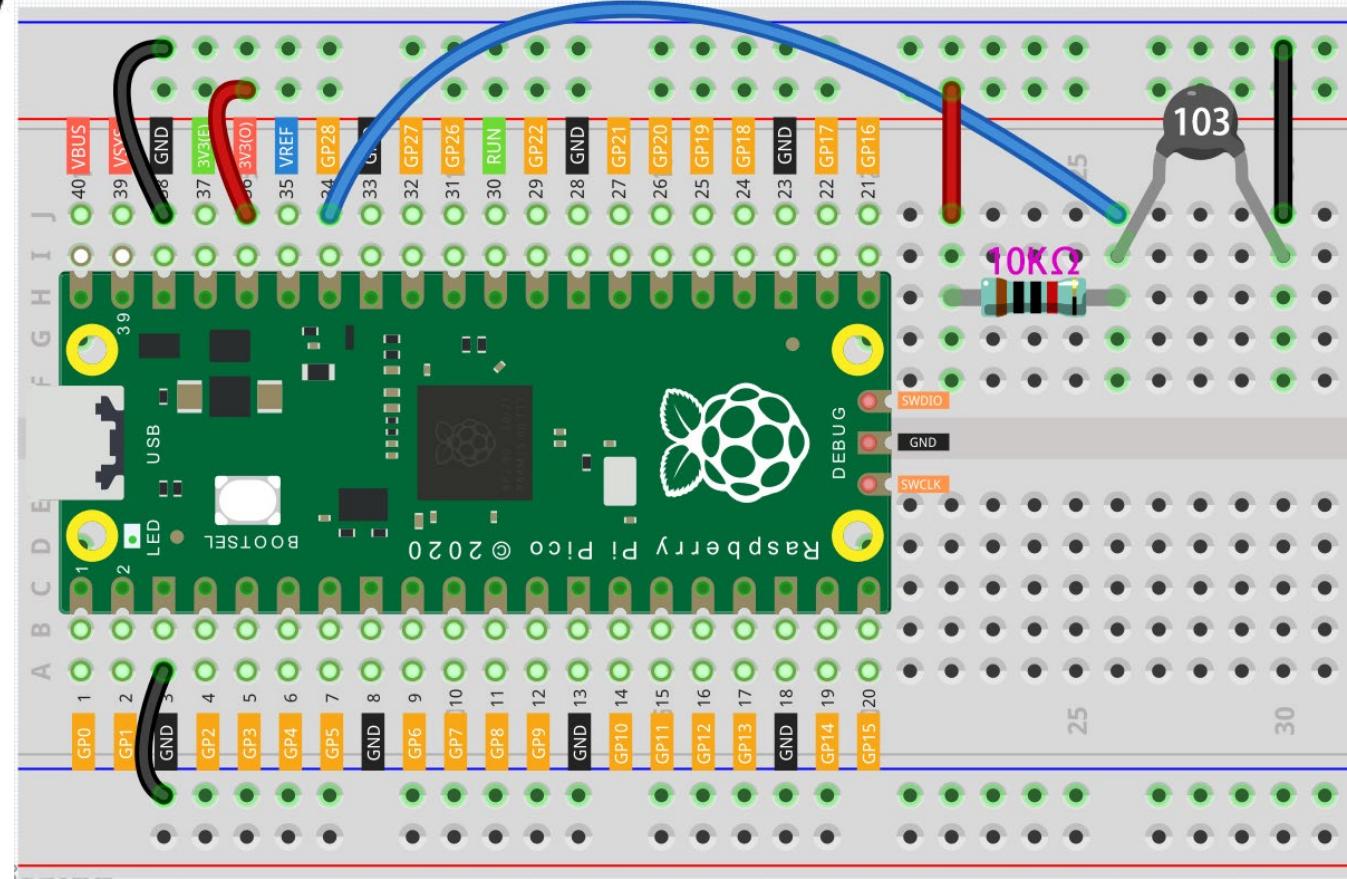
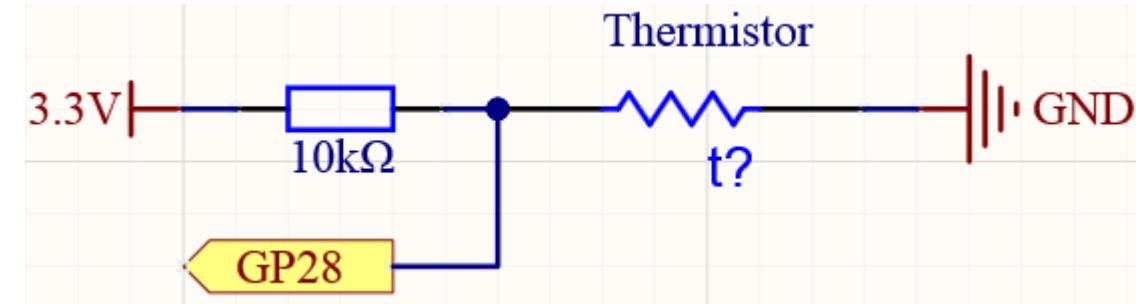
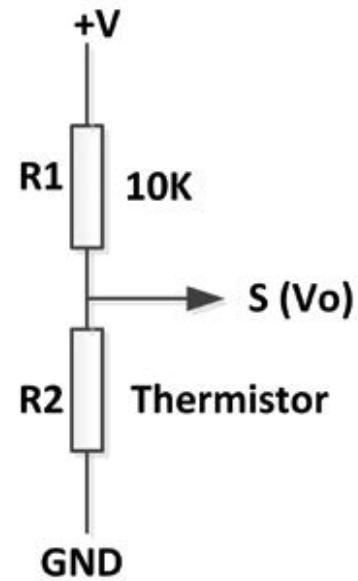
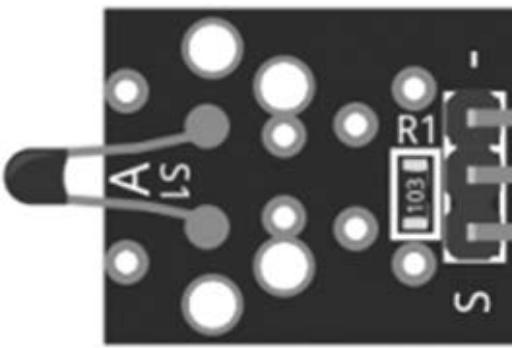
Find the value of R_t through the formula:

$$V_{out} = \frac{(V_{in} * R_t)}{R + R_t}$$

Put the value of R_t in Stein-Hart equation to get the value of temperature in kelvin

$$T = \frac{1}{A + B \ln(R_t) + C (\ln(R_t))^3}$$

Thermometer: Calculation of Temperature



Thermometer: Calculation of Temperature

WOKWi SAVE SHARE Docs B

main.py • diagram.json • PIO

```
1 from machine import ADC
2 import utime
3 import math
4
5 thermistor = ADC(28)
6
7 while True:
8     temperature_value = thermistor.read_u16()
9     Vr = 3.3 * float(temperature_value) / 65535
10    Rt = 10000 * Vr / (3.3 - Vr)
11    temp = 1/(((math.log(Rt / 10000)) / 3950) + (1 / (273.15+25)))
12    Cel = temp - 273.15
13    Fah = Cel * 1.8 + 32
14    print ('Celsius: %.2f C Fahrenheit: %.2f F' % (Cel, Fah))
15    utime.sleep_ms(200)
16
```

Simulation

NTC Temperature Sensor (analog)

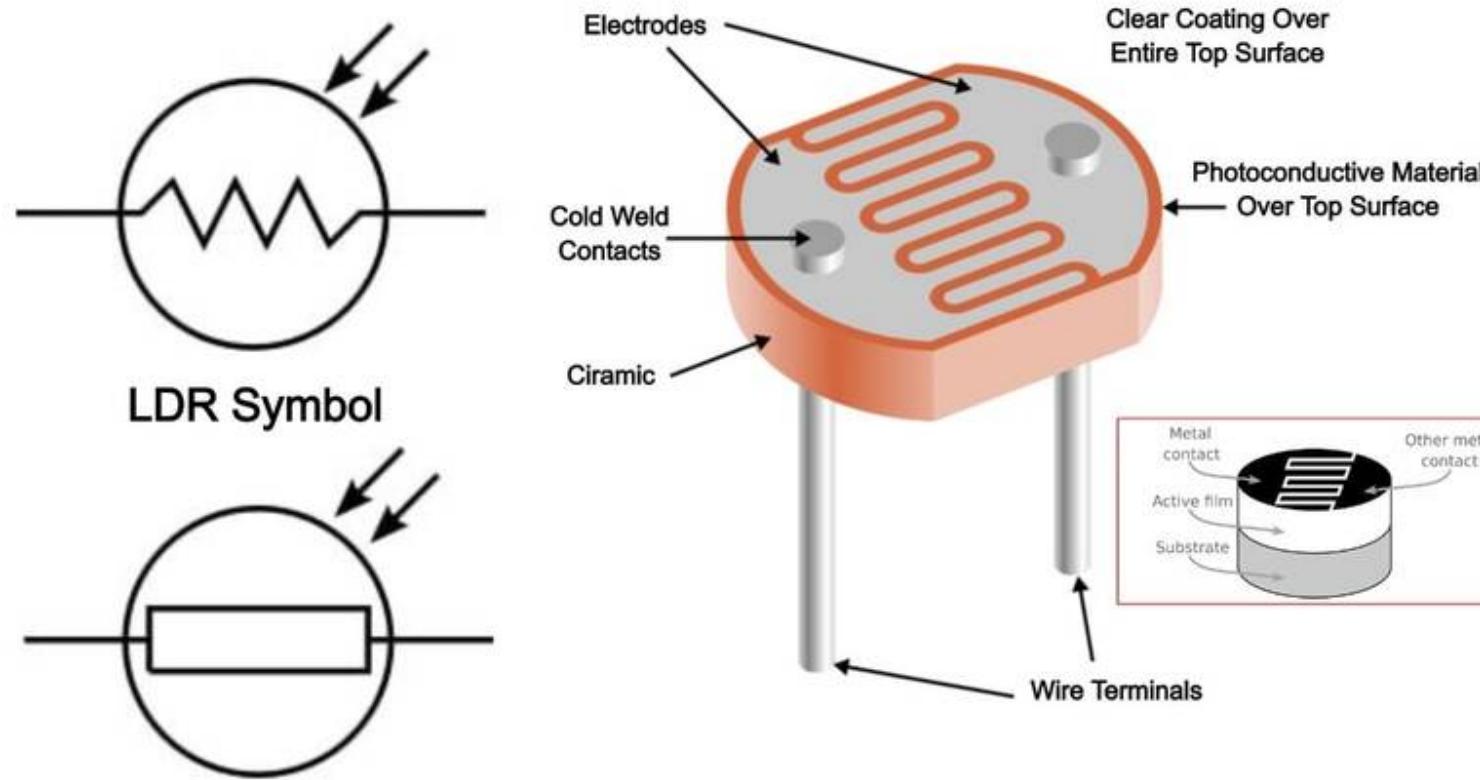
Temperature: 30.3°C

Raspberry Pi Pico

Celsius: 30.31 C Fahrenheit: 86.57 F
Celsius: 30.31 C Fahrenheit: 86.57 F
Celsius: 30.31 C Fahrenheit: 86.57 F
Celsius: 30.31 C Fahrenheit: 86.57 F

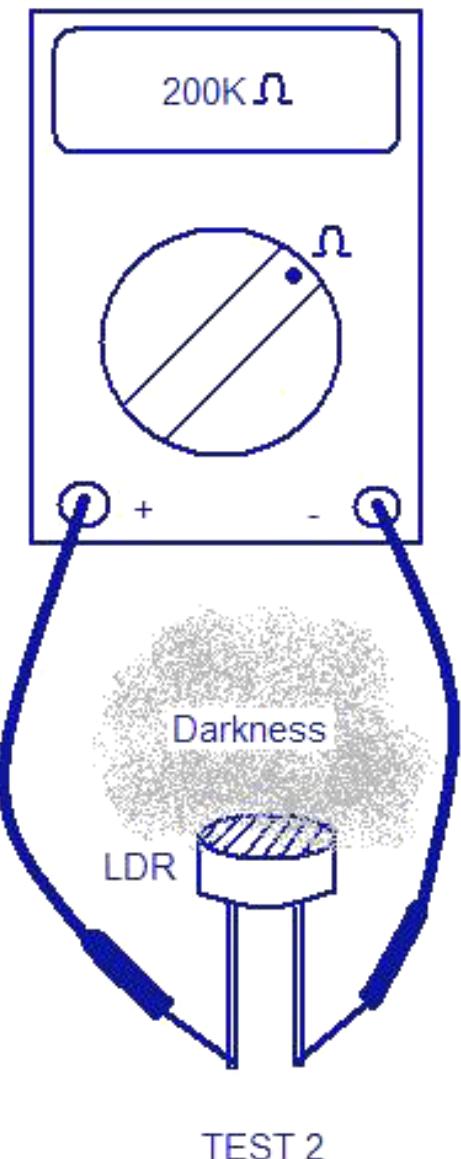
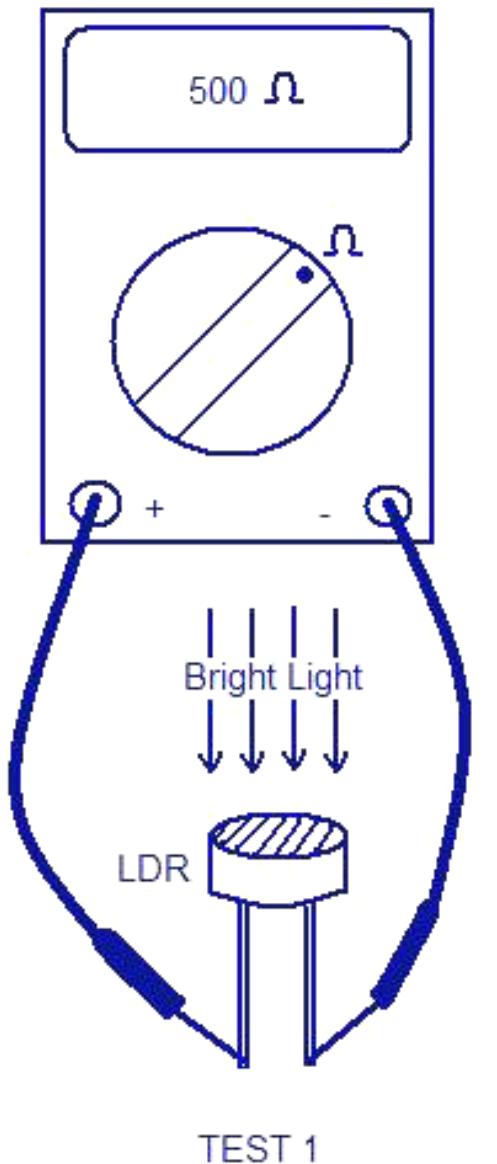
Complete on THONNY

Light Dependent Resistor (LDR)

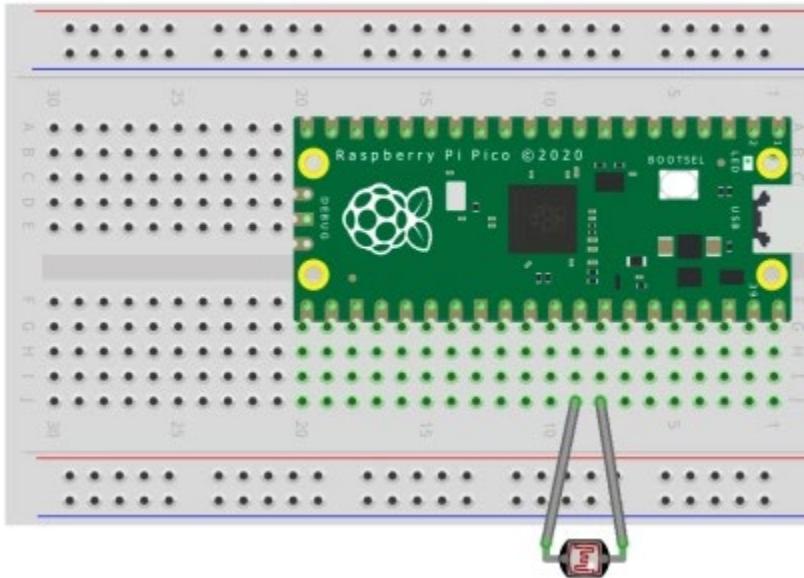


- Light Dependent Resistor or LDR or Photoresistor are electronic components that are often used in electronic circuit designs where it is necessary to detect the presence or the level of light.
- LDRs are very different from other forms of resistor like the carbon film resistor, metal oxide film resistor, metal film resistor, and the like that are widely used in other electronic designs.
- They are specifically designed for their light sensitivity and the change in resistance this causes.

LDR Testing



LDR Testing



Complete
it in
THONNY

```
from machine import Pin, ADC  
from utime import sleep
```

```
ldr = ADC(27)
```

```
while True:  
    print(ldr.read_u16())  
    sleep(2)
```

- connect one end of the LDR to GP27 (=GPIO 27 or ADC1)
- connect the other end of the LDR to a GND (ground) pin

Check LDR in

1) Dark Condition

2) Low to Bright Light
Condition

LDR Testing

WOKwi

SAVE ▾ SHARE Docs B

main.py • diagram.json • PIO

```
1 from machine import Pin, ADC
2 from utime import sleep
3
4 ldr = ADC(27)
5
6 while True:
7     print(ldr.read_u16())
8     sleep(2)
9
```

Simulation 00:54.630 96%

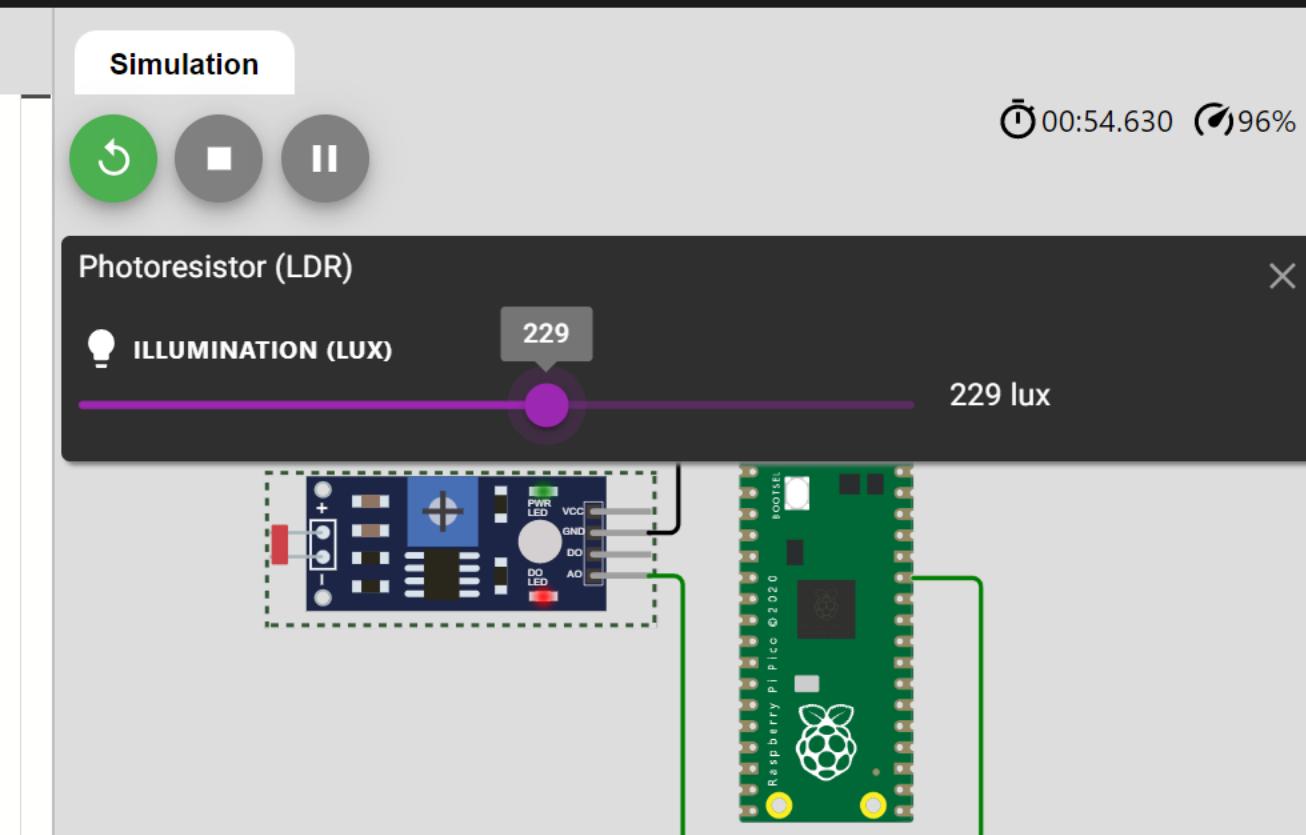
Photoresistor (LDR)

ILLUMINATION (LUX) 229

229 lux

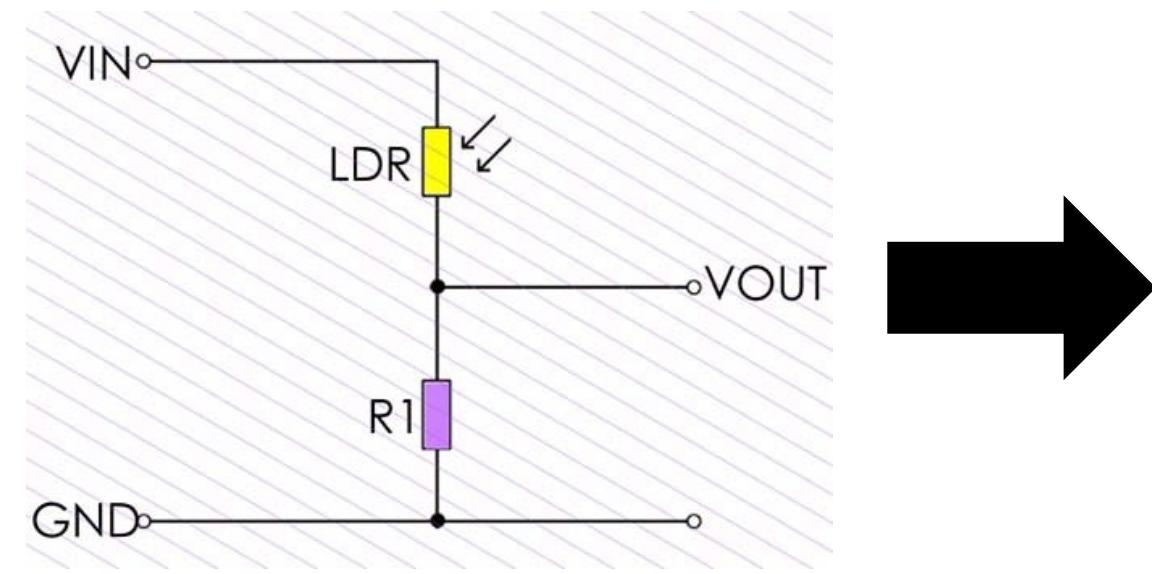
Raspberry Pi Pico © 2020

52044
52044
23477
23477



Light Dependent Resistor (LDR)

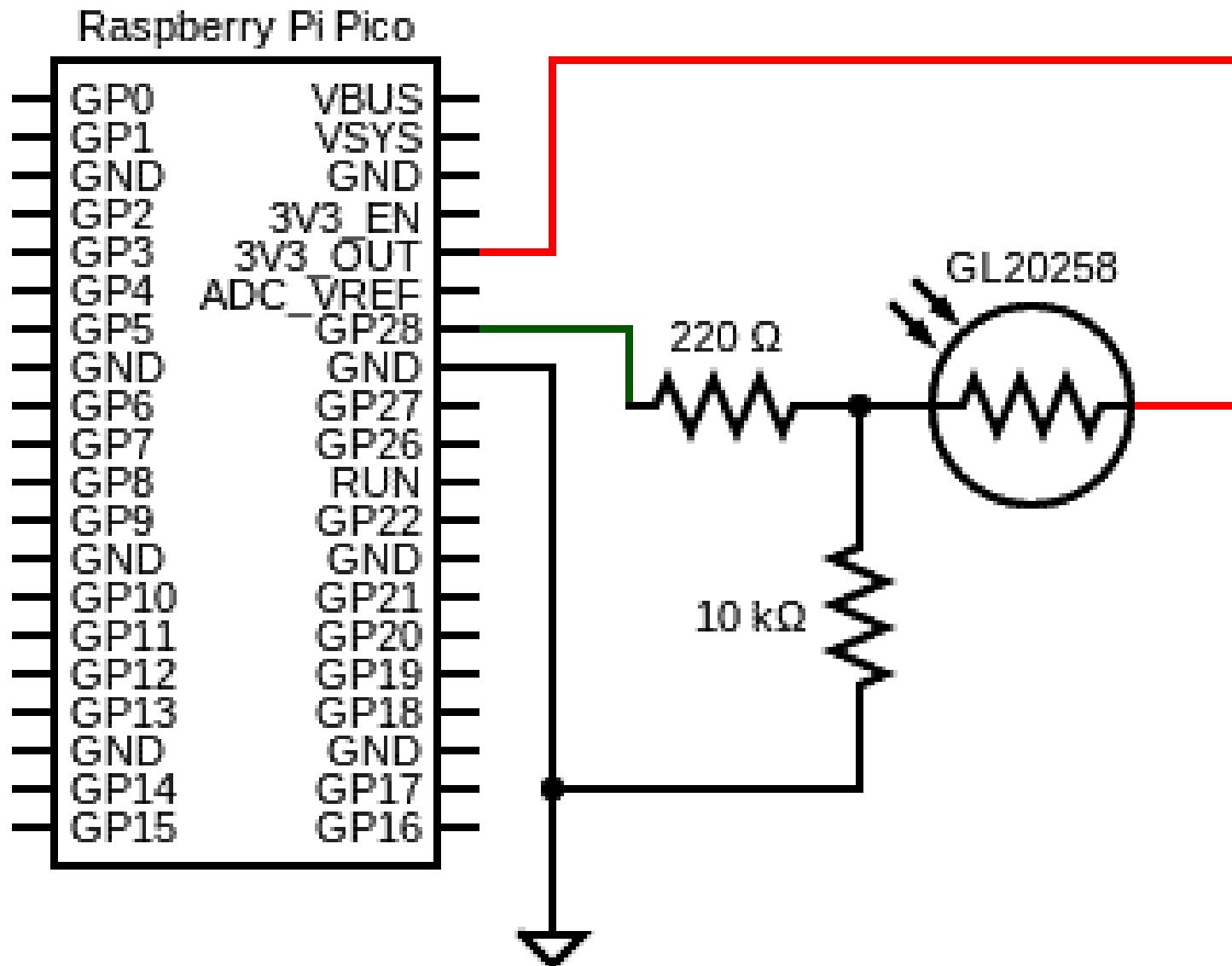
A photoresistor is in fact a light-dependent resistor (LDR). As the name suggests, it acts just like a variable resistor, changing its resistance in response to how much light is falling on it. Usually, a photoresistor has a very high resistance in the absence of light and very low resistance when stepped into the light.



Can you calculate Vout?

If you feed this VOUT to an ADC pin of Raspberry Pi Pico (**GP28/ADC2** in this experiment), compared to the reference voltage (3.3V) provided by Raspberry Pi Pico, you will get a value proportional to the light exposed to the photoresistor. With ADC module from MicroPython, this value will range between 0 and 65535, and we'll transform it roughly into a percentage scale (0% and 100%).

Light Dependent Resistor (LDR)



Light Dependent Resistor (LDR)

WOKWI

 SAVE

 SHARE

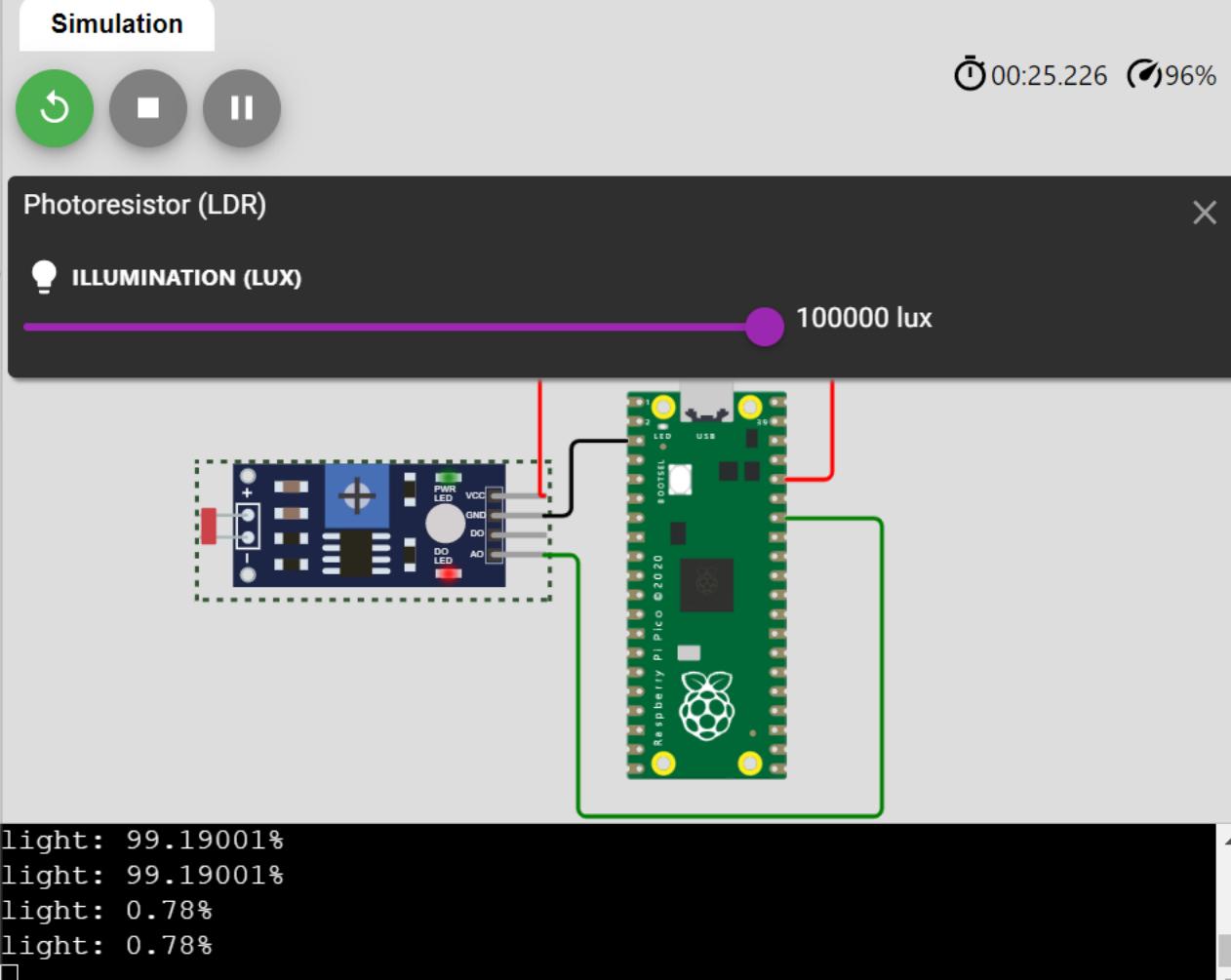
Docs

B

```
main.py ● diagram.json ● PIO 🐍
```

```
1  from machine import ADC, Pin
2  from time import sleep
3
4  photoPIN = 28
5
6  def readLight(photoGP):
7      photoRes = ADC(Pin(28))
8      light = photoRes.read_u16()
9      light = round(light/65535*100)
10     return light
11
12 while True:
13     print("light: " + str(readLight()))
14     sleep(1)
```

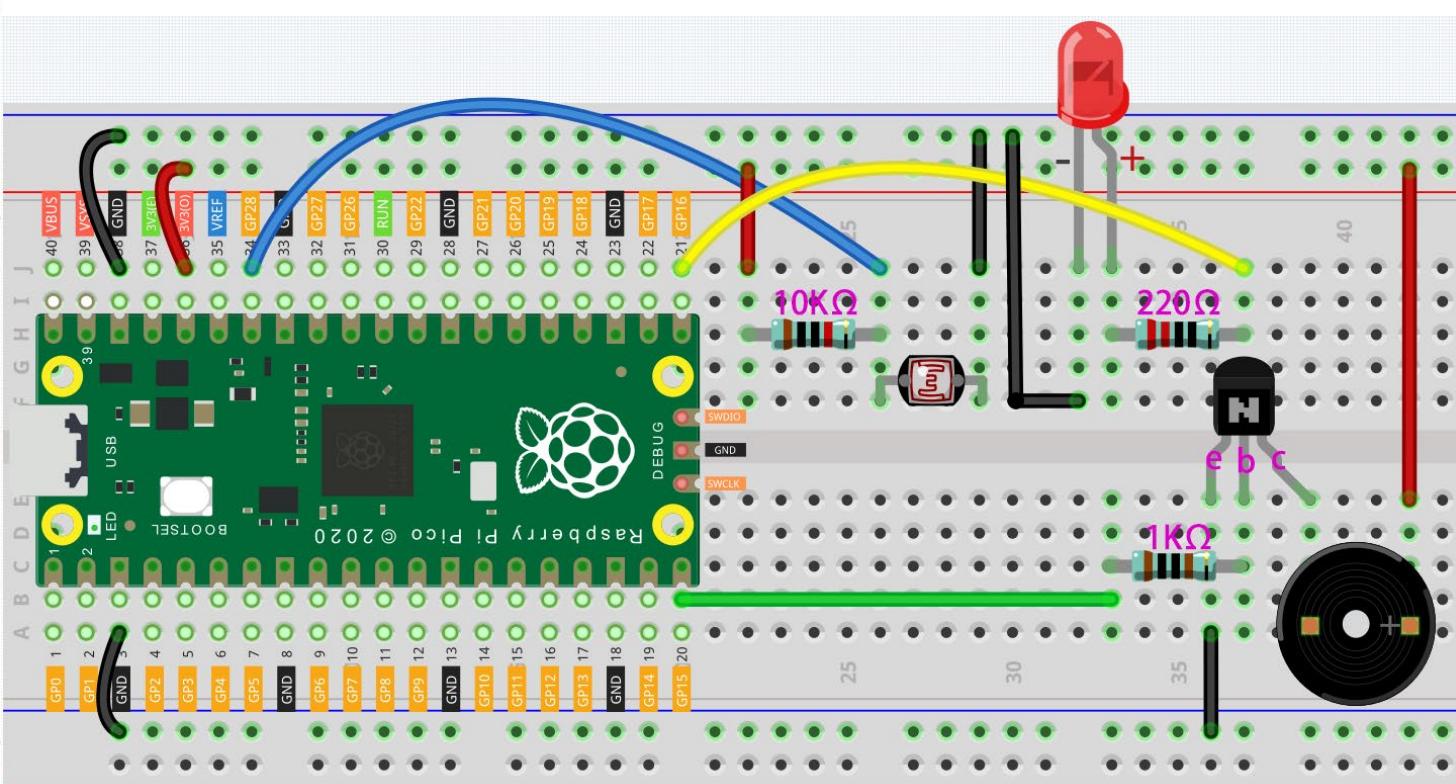
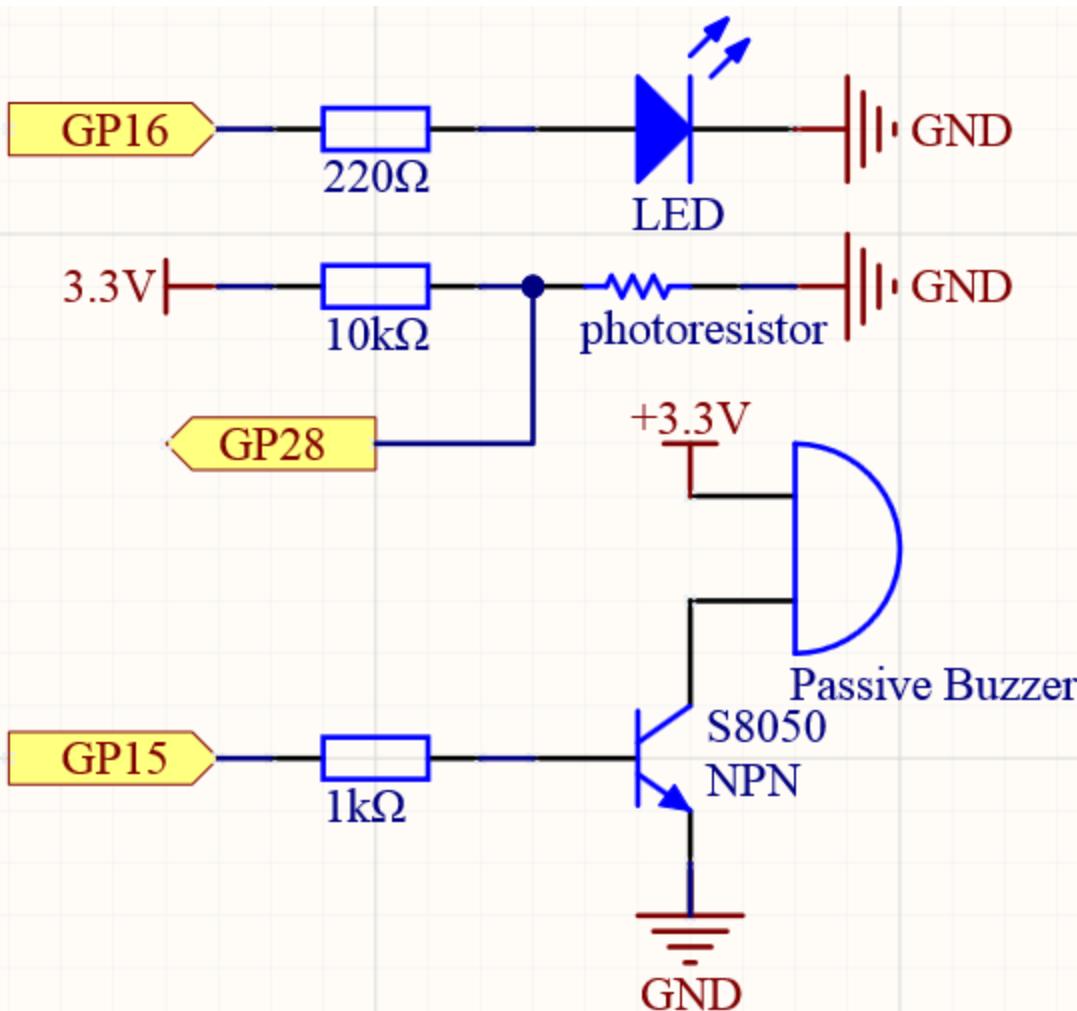
**Complete it
in THONNY**



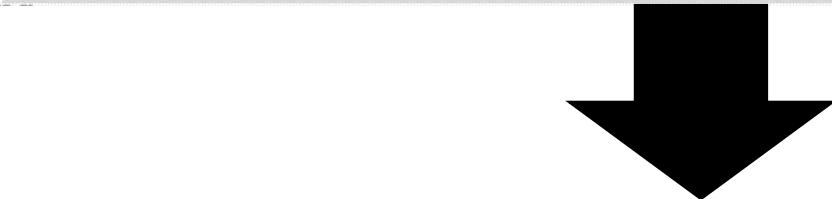
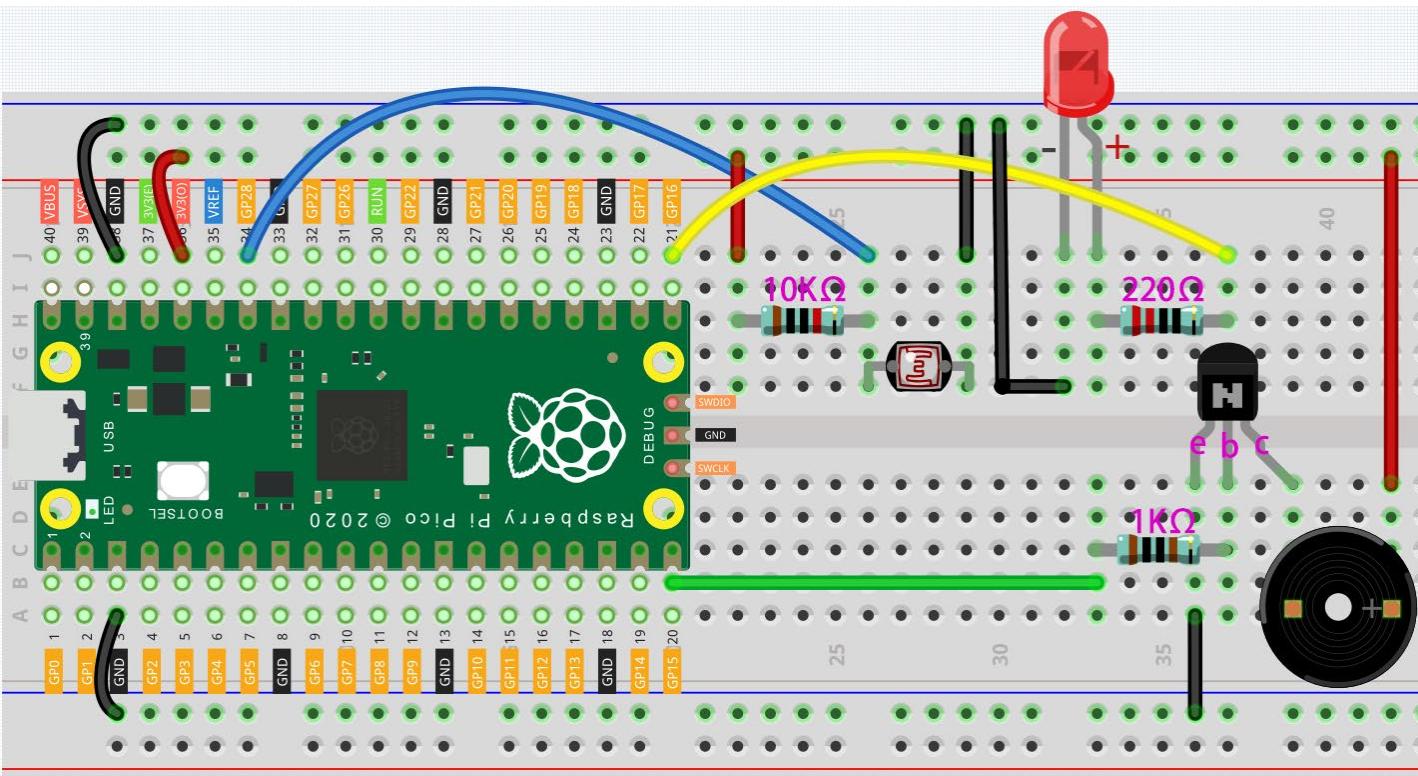
Light Theremin: Electronic musical instrument

- Theremin is an electronic musical instrument that does not require physical contact. It produces different tones by sensing the position of the player's hand.
- The instrument's controlling section usually consists of two metal antennas that sense the relative position of the thereminist's hands and control oscillators for frequency with one hand, and amplitude (volume) with the other. The electric signals from the theremin are amplified and sent to a loudspeaker.
- We cannot reproduce the same instrument through Pico, but we can use photoresistor and passive buzzer to achieve similar gameplay.

Light Theremin: Electronic musical instrument



Light Theremin: Electronic musical instrument



Five seconds to calibrate the detection range of the photoresistor

```
from machine import Pin, PWM, ADC
import utime

led = Pin(16, Pin.OUT)
photoresistor = ADC(28)
buzzer = PWM(Pin(15))

light_low=65535
light_high=0

def interval_mapping(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def tone(pin,frequency,duration):
    pin.freq(frequency)
    pin.duty_u16(30000)
    utime.sleep_ms(duration)
    pin.duty_u16(0)

# calibrate the photoresistor max & min values.
timer_init_start = utime.ticks_ms()
led.value(1)
while utime.ticks_diff(utime.ticks_ms(), timer_init_start)<5000:
    light_value = photoresistor.read_u16()
    if light_value > light_high:
        light_high = light_value
    if light_value < light_low:
        light_low = light_value
led.value(0)

# play
while True:
    light_value = photoresistor.read_u16()
    pitch = int(interval_mapping(light_value,light_low,light_high,50,6000))
    if pitch > 50 :
        tone(buzzer,pitch,20)
    utime.sleep_ms(10)
```