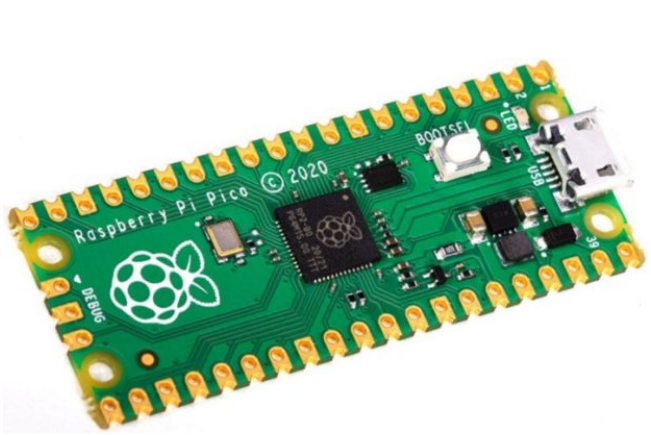
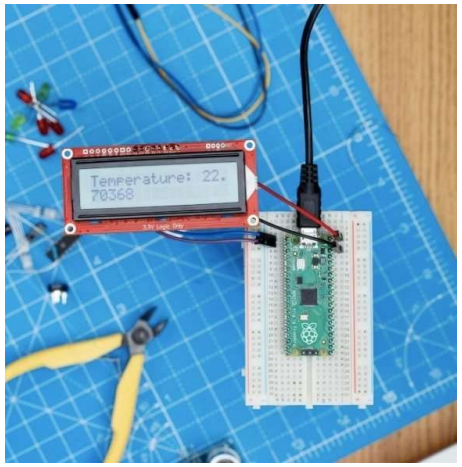
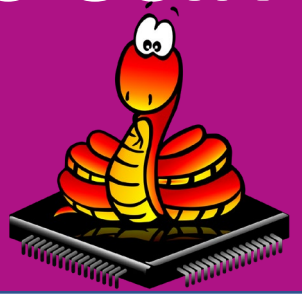


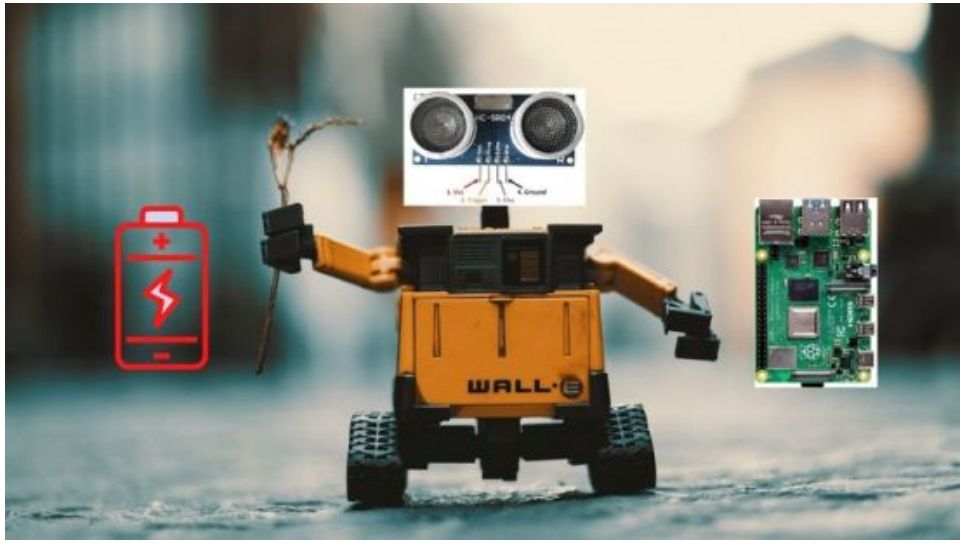


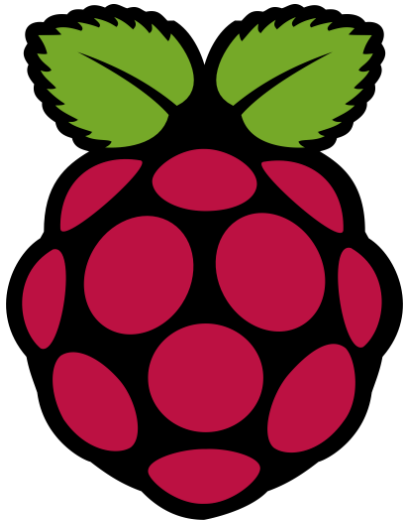
INTERNET OF THINGS (IOT) PROJECTS USING PYTHON

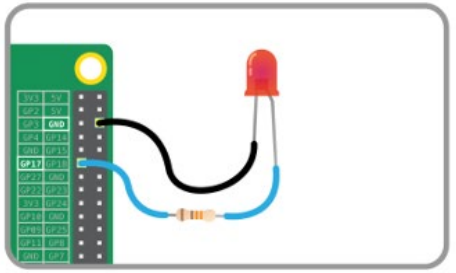
(CSE 4110)

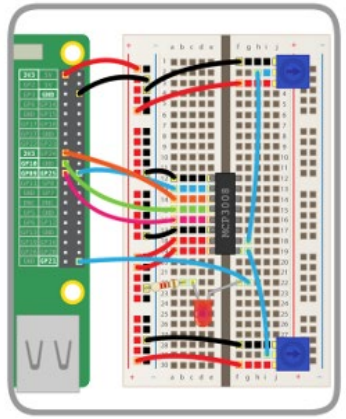
(LECTURE – 6)











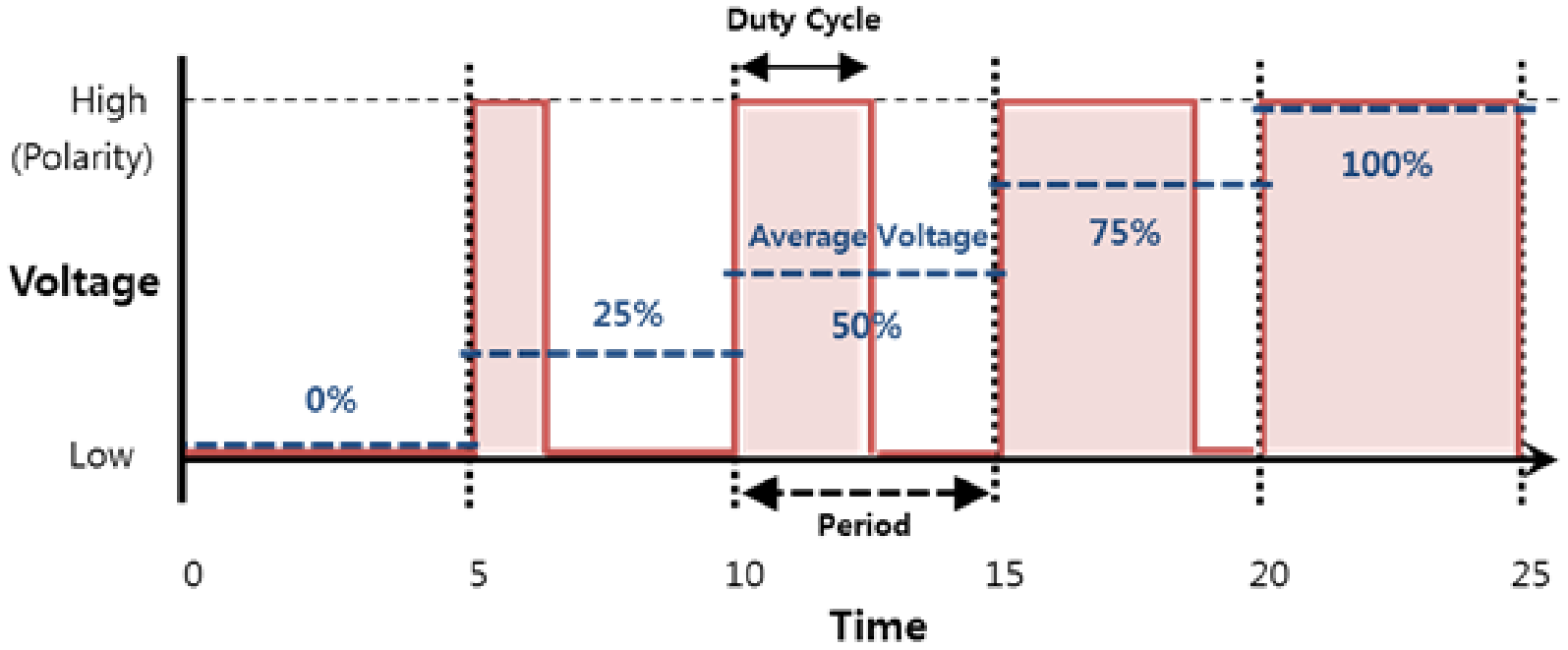
Pulse With Modulation (PWM)

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.

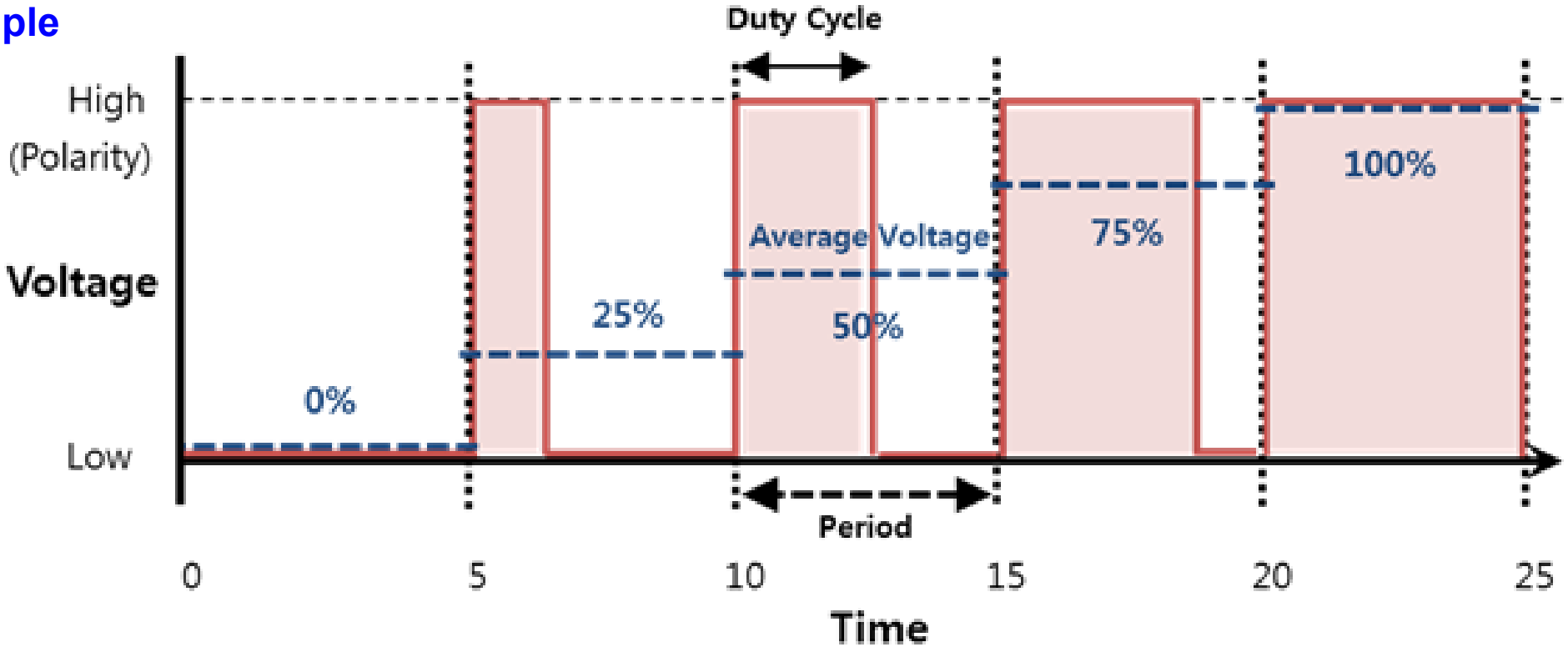
The PWM consists in varying the width of an electrical pulse.

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



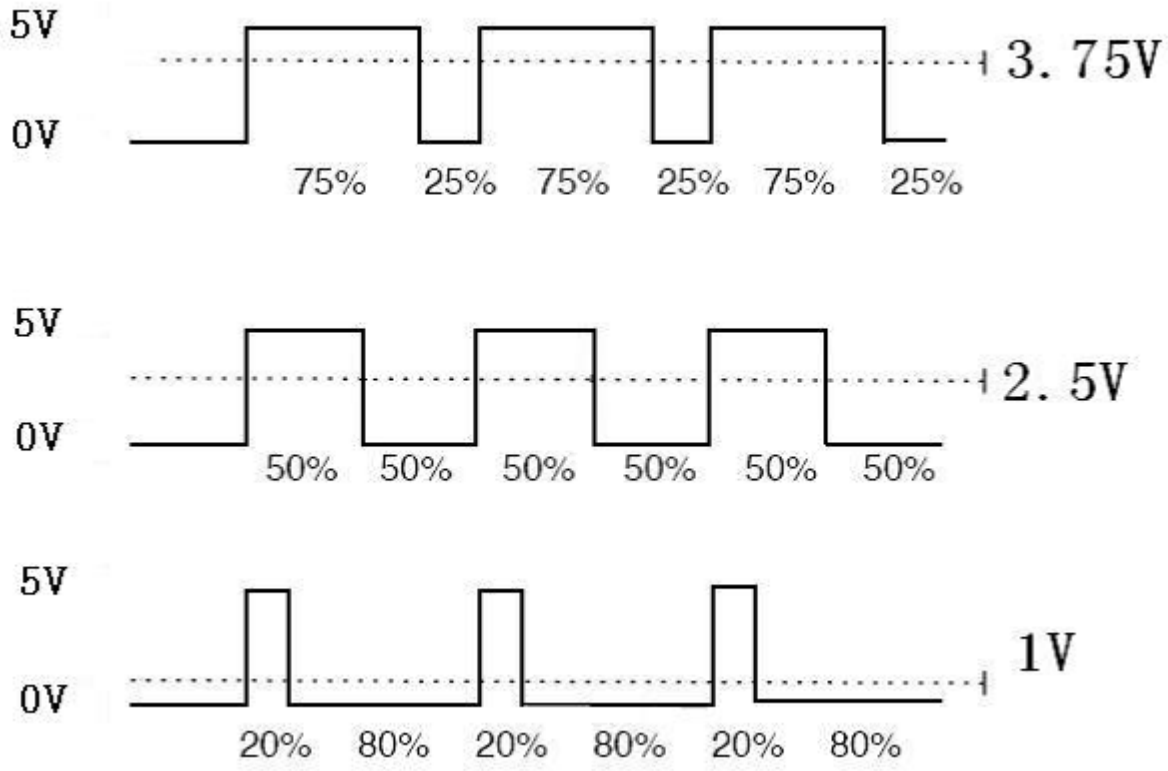
The succession of pulses with a given width is seen on average as a constant voltage between 0V and 3.3V, whose value is determined by :

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

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

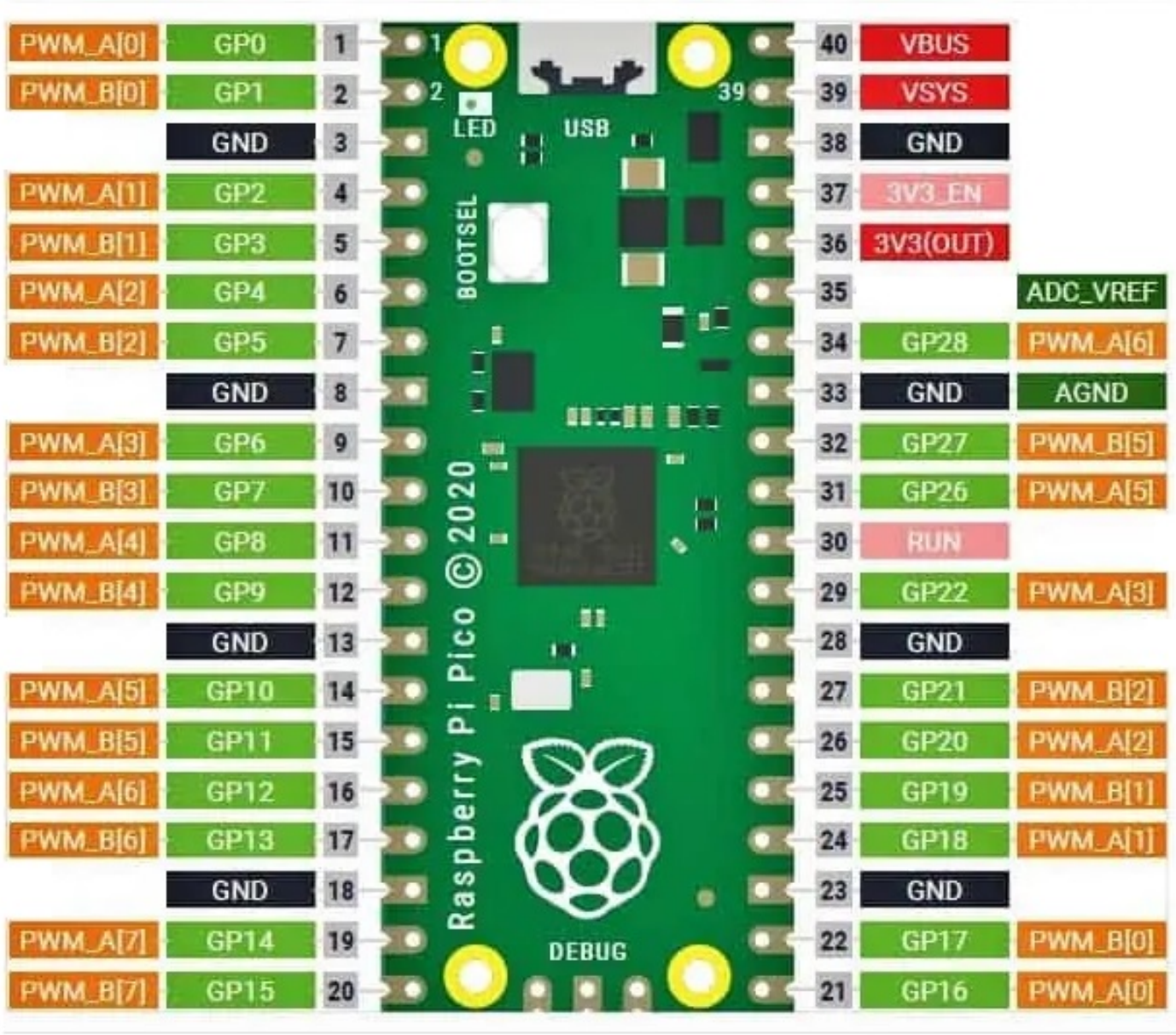
All 30 GPIO pins on RP2040 can be used for PWM:

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



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

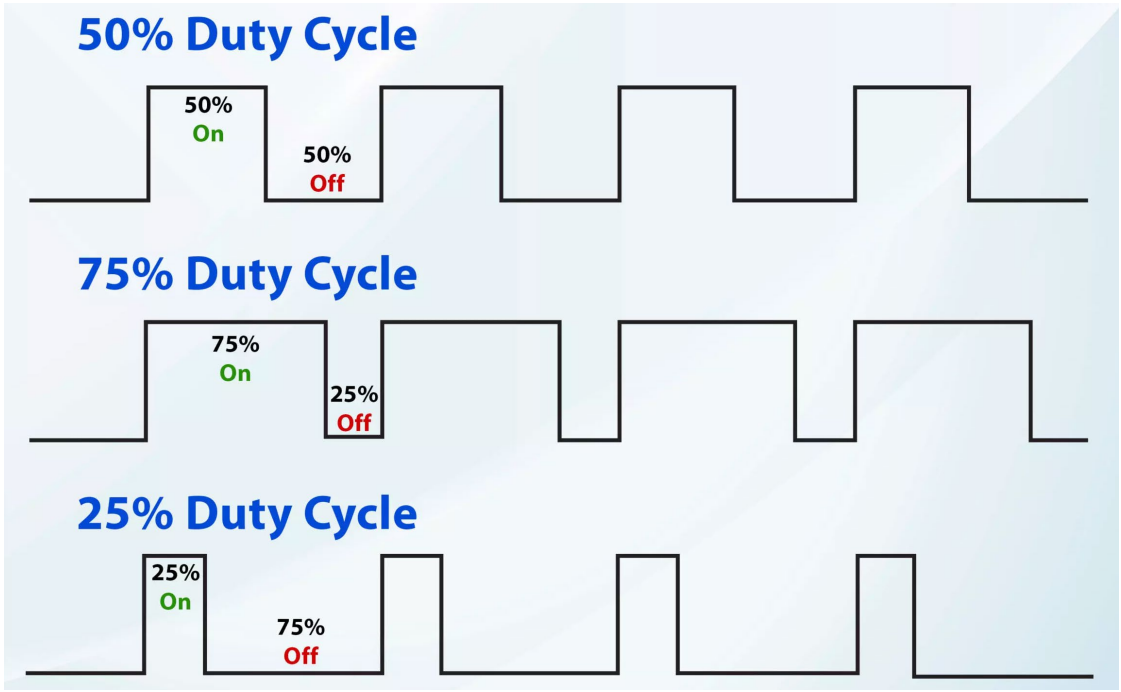
Pulse With Modulation (PWM)

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



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

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 α=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** .

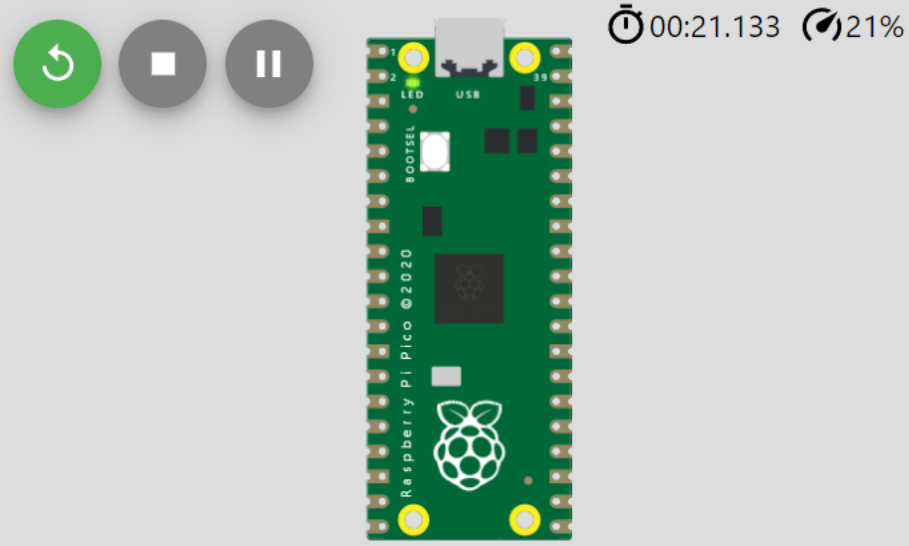
We use the function **.duty_u16()** to select the duty cycle with a value between 0 and $2^{16} - 1$ (**0-65535**). The voltage is set directly on the output of the previously selected pin

WOKWi SAVE SHARE Docs B

main.py ● diagram.json ● PIO

```
1   from machine import Pin, PWM
2
3   LED_BUITLTIN = 25                          # For Raspberry Pi Pico
4
5   pwm_led = PWM(Pin(LED_BUITLTIN, 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):
12          pwm_led.duty_u16(duty)                 # For Pi Pico
```

Simulation



LED Breathing : Duty Cycle Percentage Specification

PWM on MicroPython with Pico

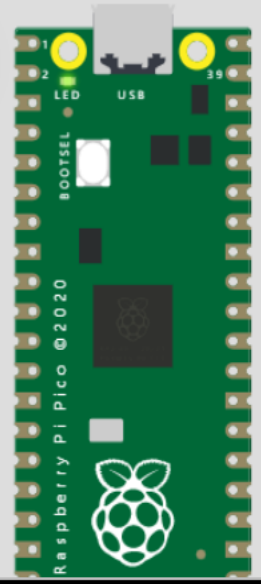
WOKWI SAVE SHARE Docs B

main.py ● diagram.json ● PIO

```
1  from machine import Pin, PWM
2
3  LED_BUITLTIN = 25                                  # For Raspberry Pi Pico
4
5  pwm_led = PWM(Pin(LED_BUITLTIN, 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

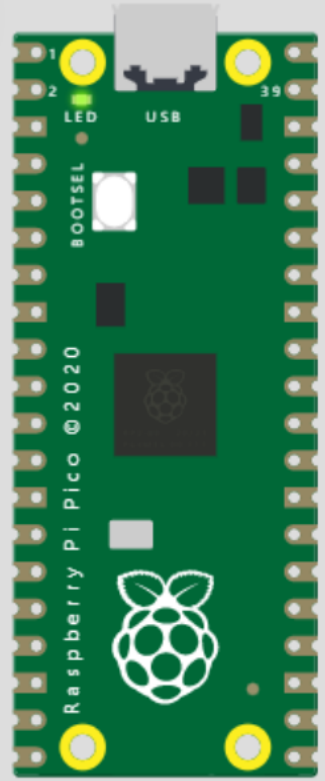
WOKWI SAVE SHARE Docs B

main.py ● diagram.json ● PIO

```
1   from machine import Pin, PWM
2   import time
3
4   LED_BUITLTIN = 25                                # For Raspberry Pi Pico
5
6   pwm_led = PWM(Pin(LED_BUITLTIN, 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

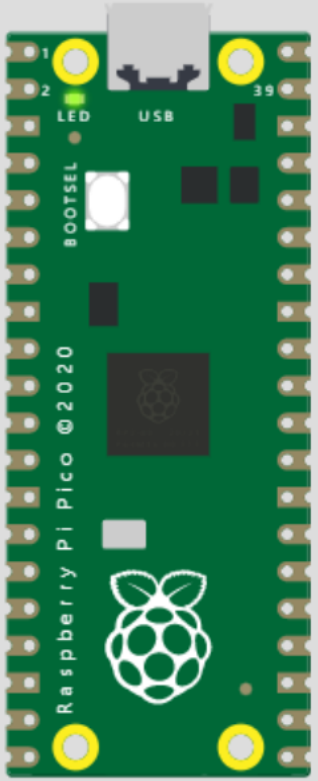
WOKWi SAVE SHARE Docs B

main.py ● diagram.json ● PIO

```
1  from machine import Pin, PWM
2
3  LED_BUITLTIN = 25                          # For Raspberry Pi Pico
4
5  pwm_led = PWM(Pin(LED_BUITLTIN, 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

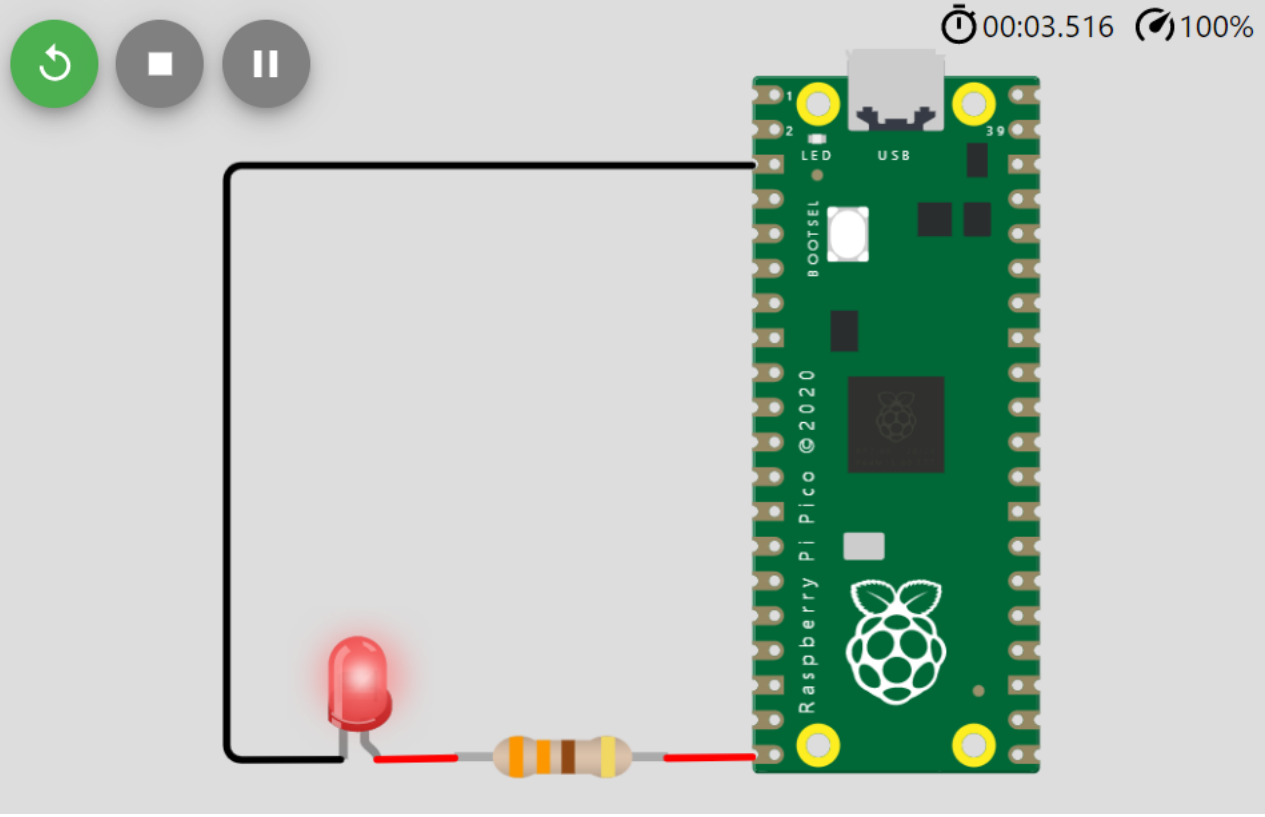
WOKWI SAVE SHARE Docs B

main.py ● diagram.json ● PIO

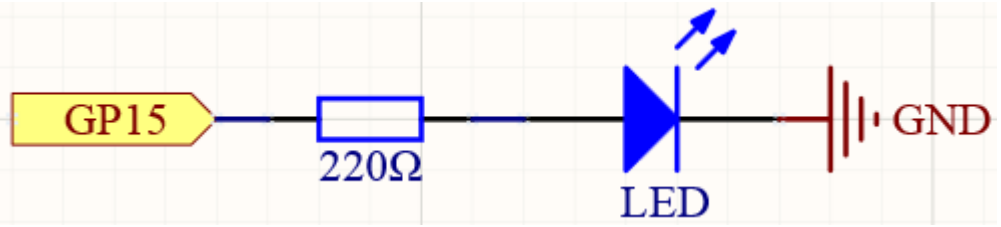
```
1  from machine import Pin, PWM
2  import utime
3
4  led = machine.PWM(machine.Pin(15))
5  led.freq(1000)
6
7  for brightness in range(0,65535,50):
8      led.duty_u16(brightness)
9      utime.sleep_ms(10)
10 led.duty_u16(0)
11
```

Simulation

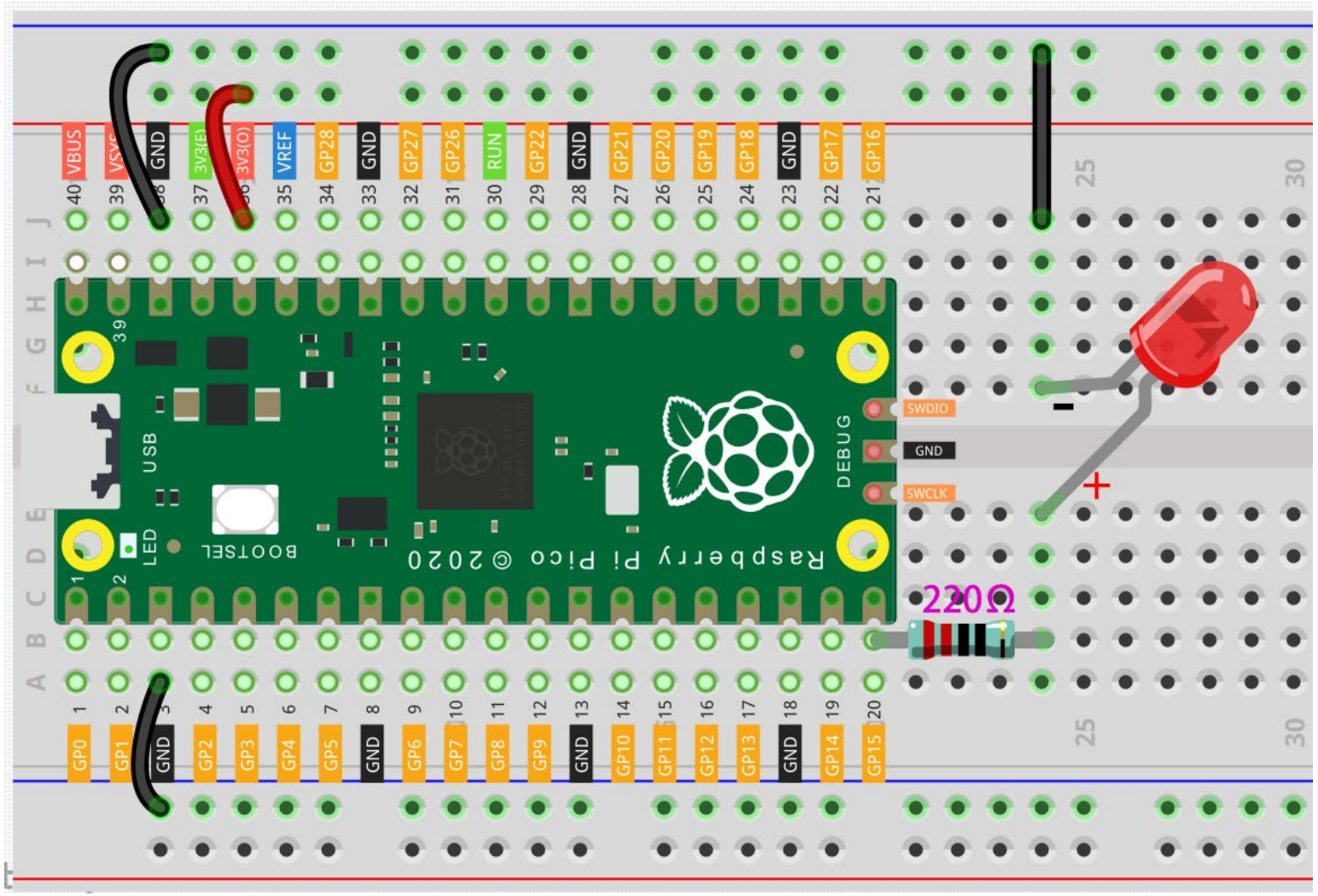
00:03.516 100%



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%

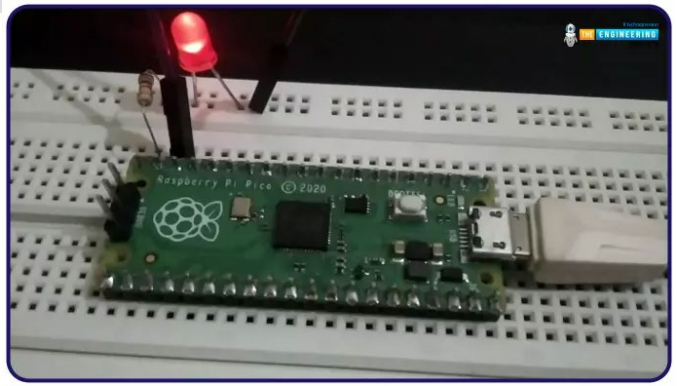
1) Change the duty cycle to 100%

2) Change the duty cycle to 50%

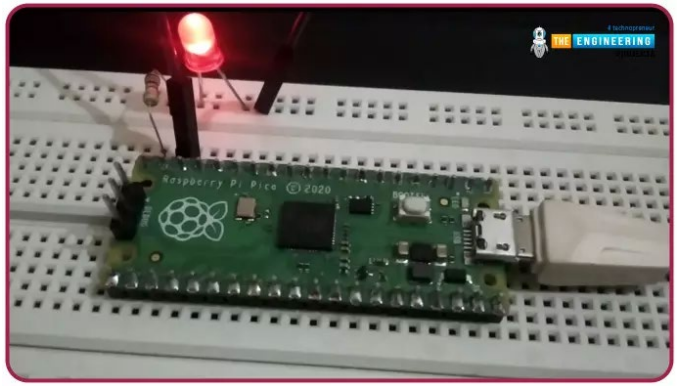
LED Breathing : Up and Down Fading the External LED

```
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)
```

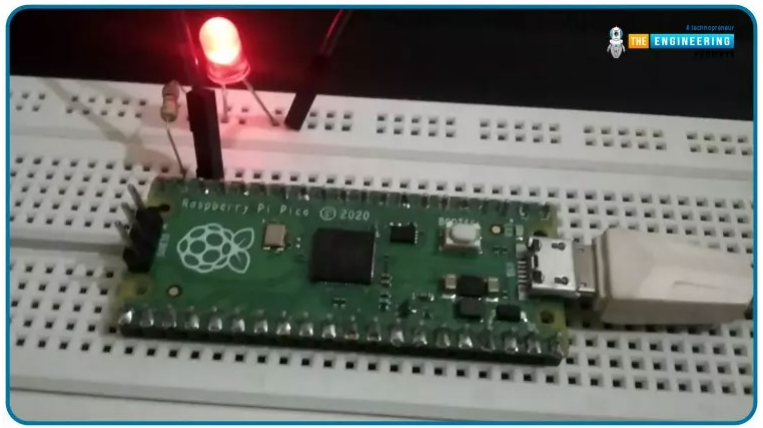
Apply on "Thonny"



Brightness level 1

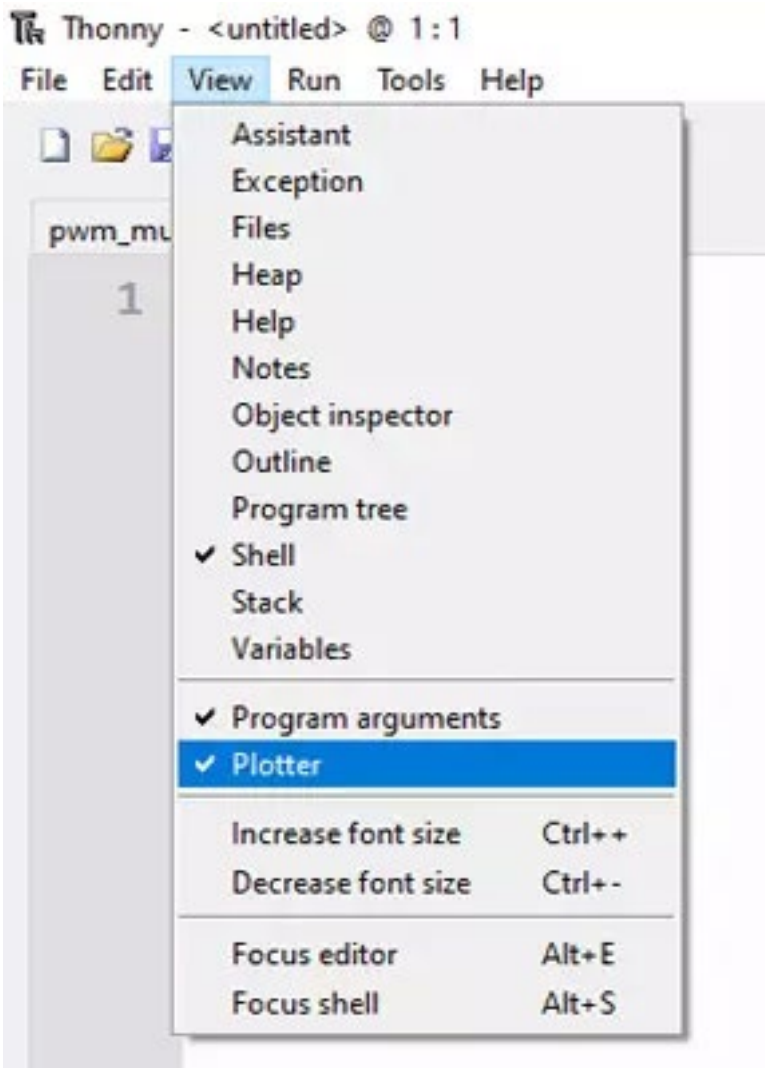


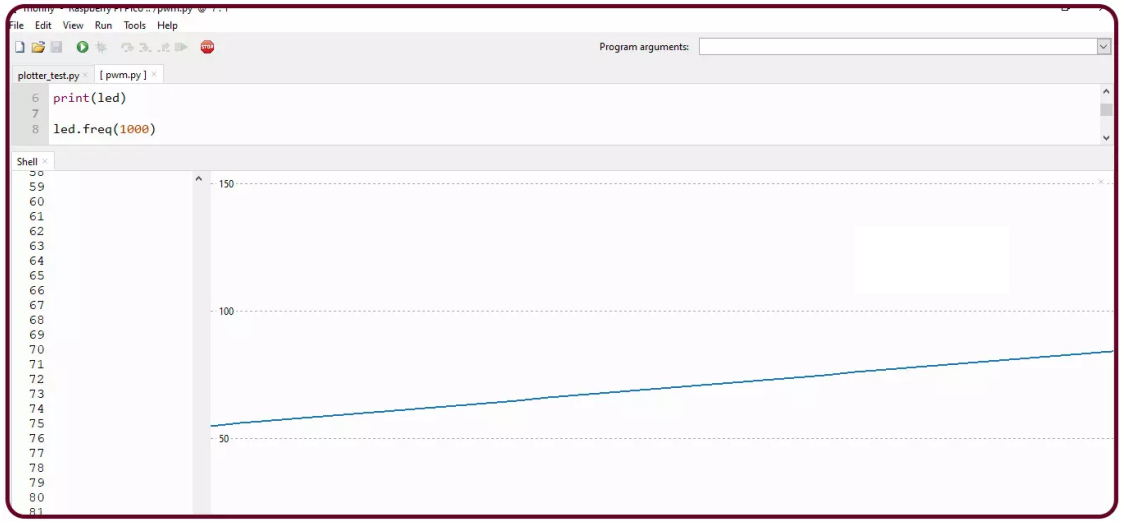
Brightness level 2



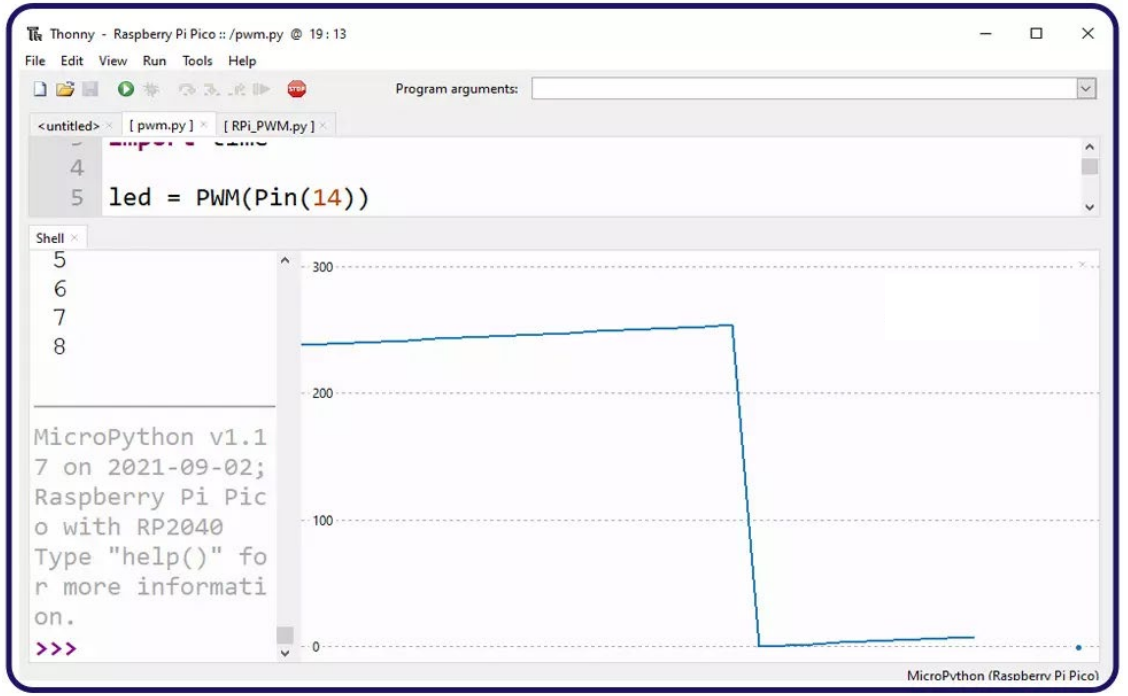
Brightness level 3

LED Breathing : Plotter in Thonny



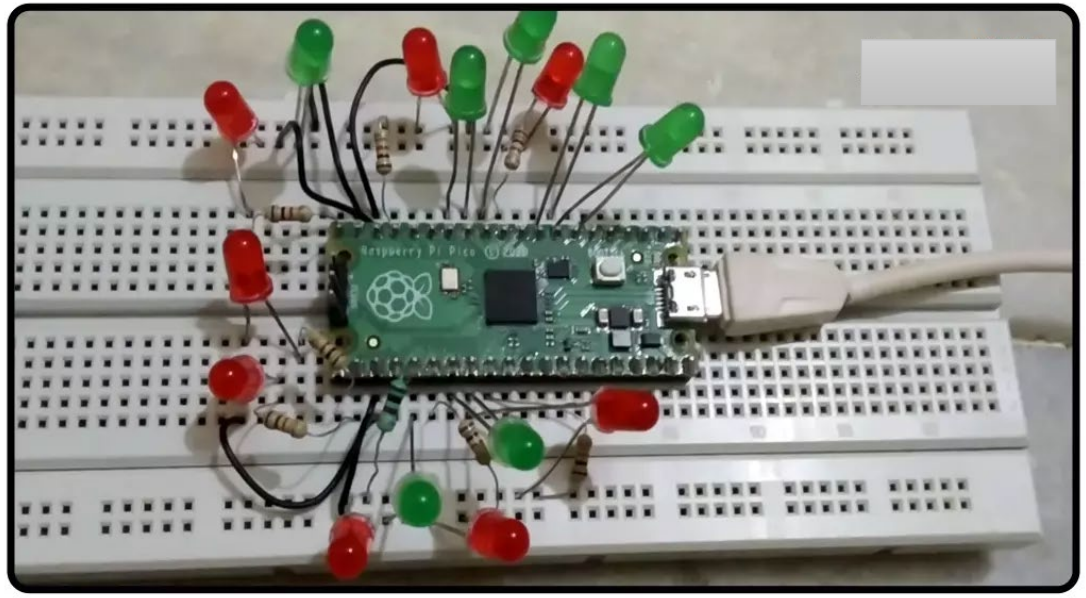


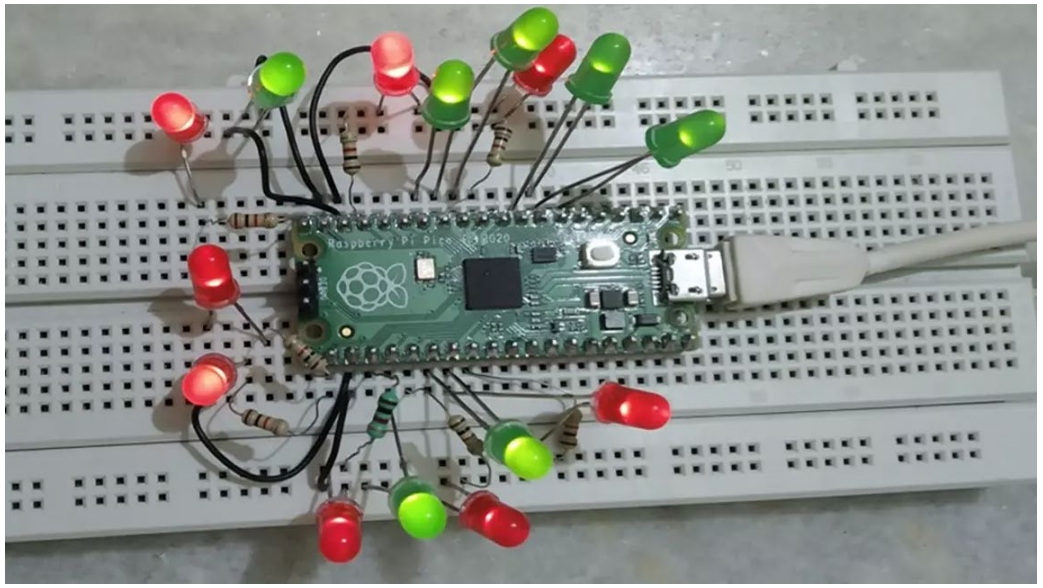
Rising PWM output 0 to 65535
(brightness increasing)



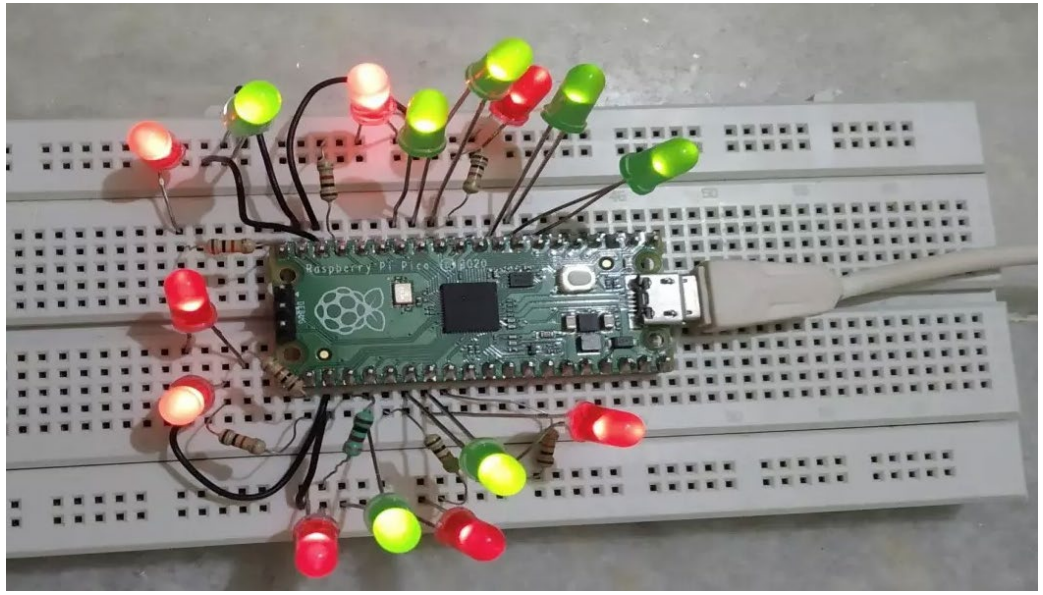
PWM output maximum to 0

LED Breathing : Complete the Challenge

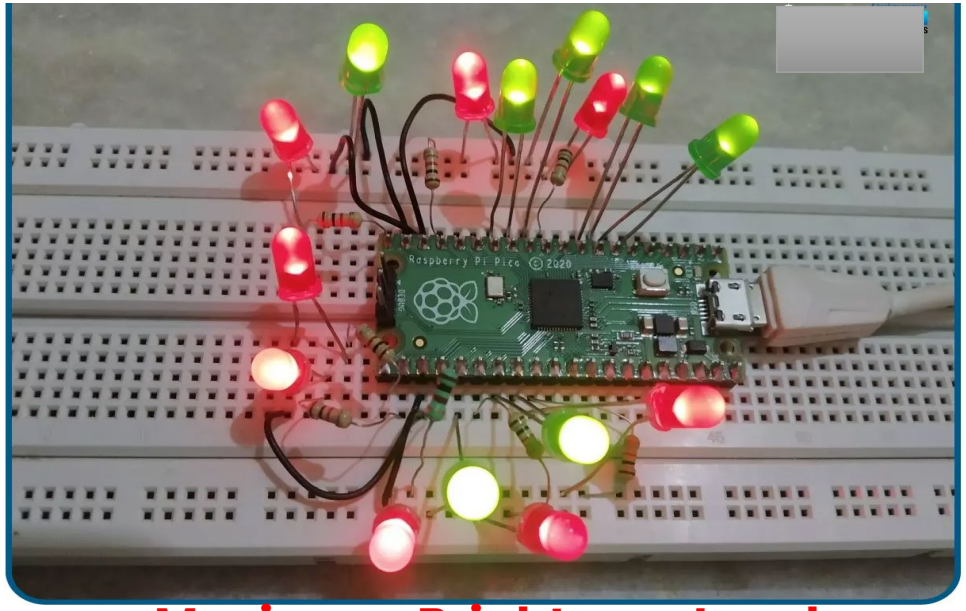




Minimum Brightness Level



Intermediate 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 pir
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 foe 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_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 prequency
```

Try to complete the circuit connection with the help of code

LED Breathing : Complete the Challenge

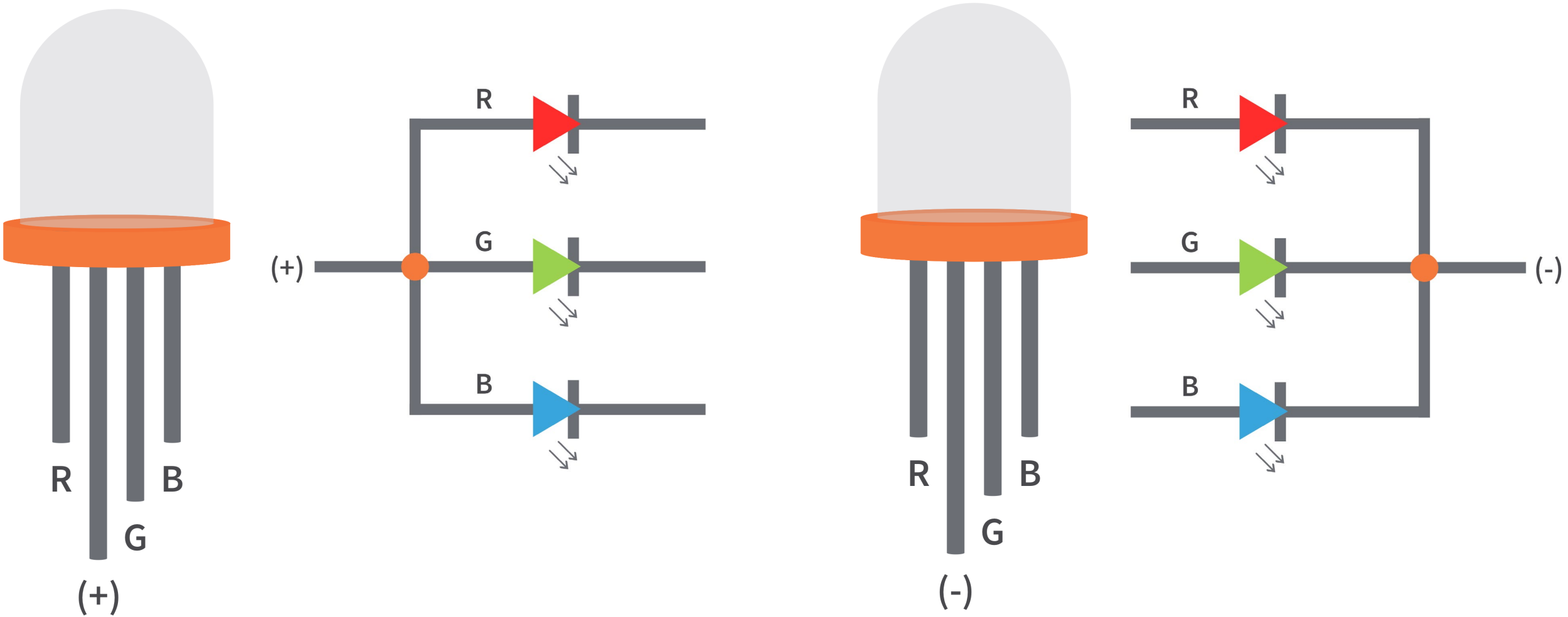
```
while True:
    for duty in range(0, 65535):    # Increasing LED broghtness
    led_1.duty_u16(duty)
    led_2.duty_u16(duty)
    led_3.duty_u16(duty)
    led_4.duty_u16(duty)
    led_5.duty_u16(duty)
    led_6.duty_u16(duty)
    led_7.duty_u16(duty)
    led_8.duty_u16(duty)
    led_9.duty_u16(duty)
    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 RGB LED

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

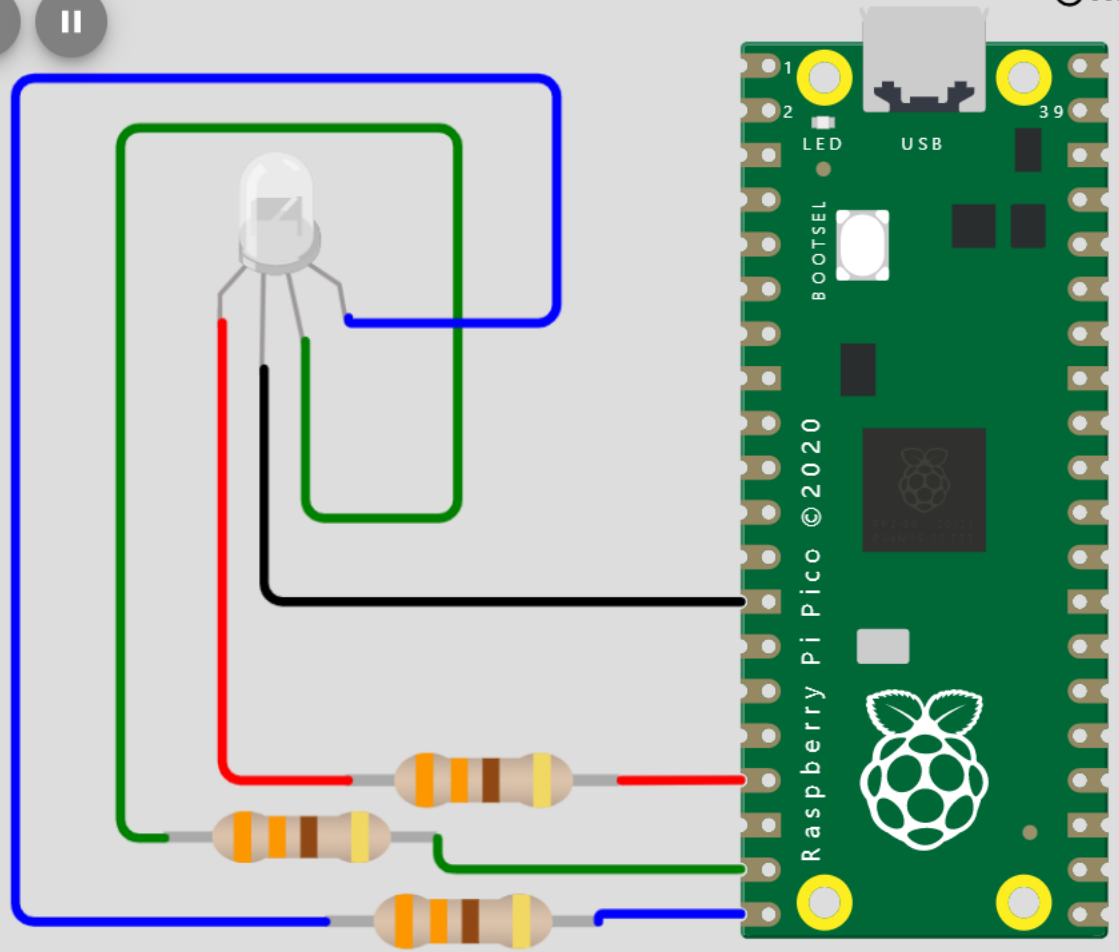
WOKWi SAVE SHARE Docs B

main.py ● diagram.json ● PIO

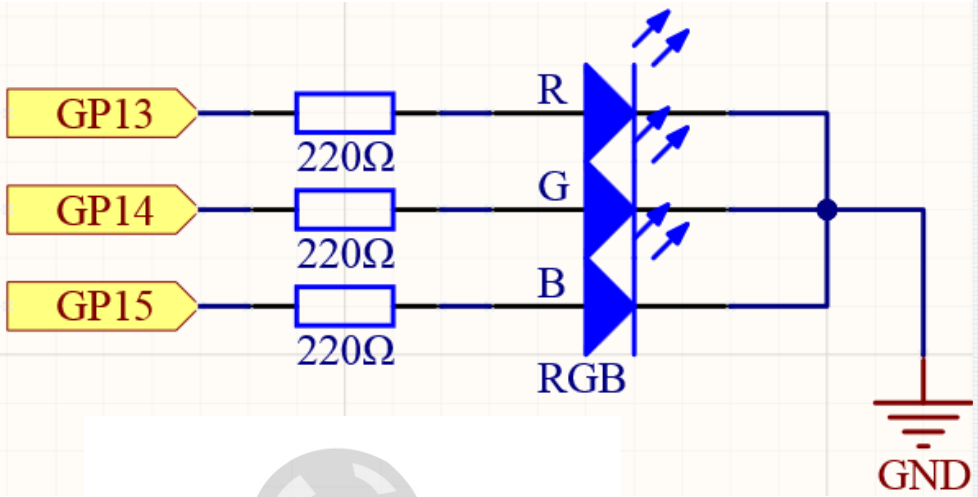
```
1  from machine import Pin
2  from utime import sleep
3
4  red = Pin(13, Pin.OUT)
5  green = Pin(14, Pin.OUT)
6  blue = Pin(15, Pin.OUT)
7
8  #Clear common cathode RGB LED
9
10 red.value(0)
11 green.value(0)
12 blue.value(0)
13
14 while True:
15     red.toggle()
16     sleep(1)
17     red.toggle()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

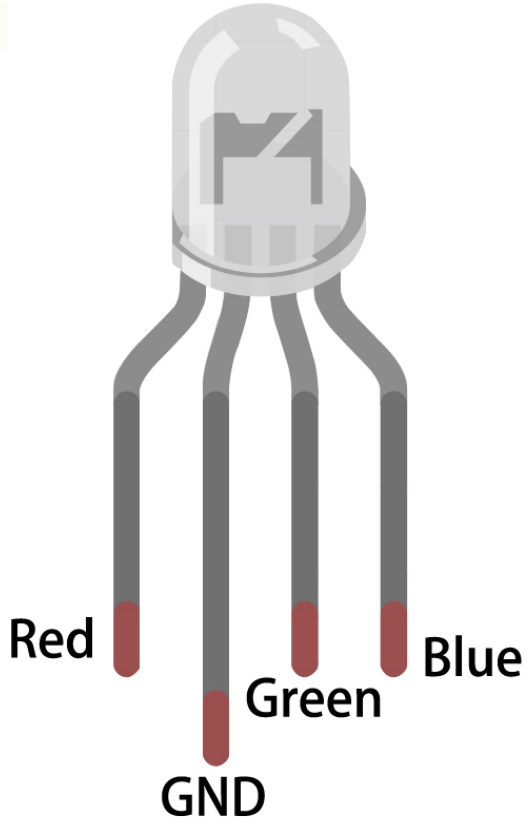
Simulation

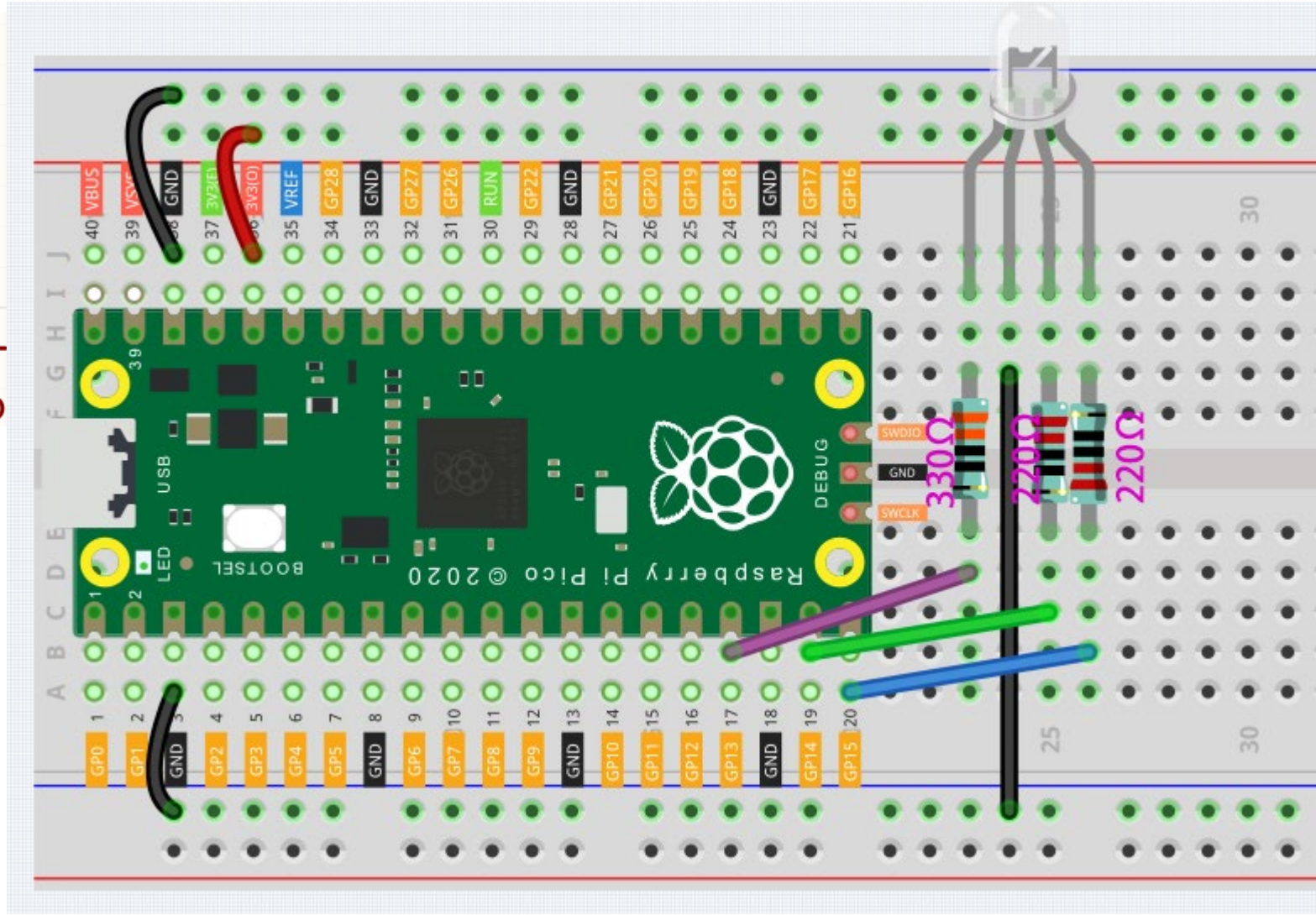
00:26.763 100%



Colorful Light: Light can be superimposed







Colorful Light: Mystery of additive color mixing

WOKWI SAVE SHARE Docs B

main.py | diagram.json | rgb.py ● | PIO

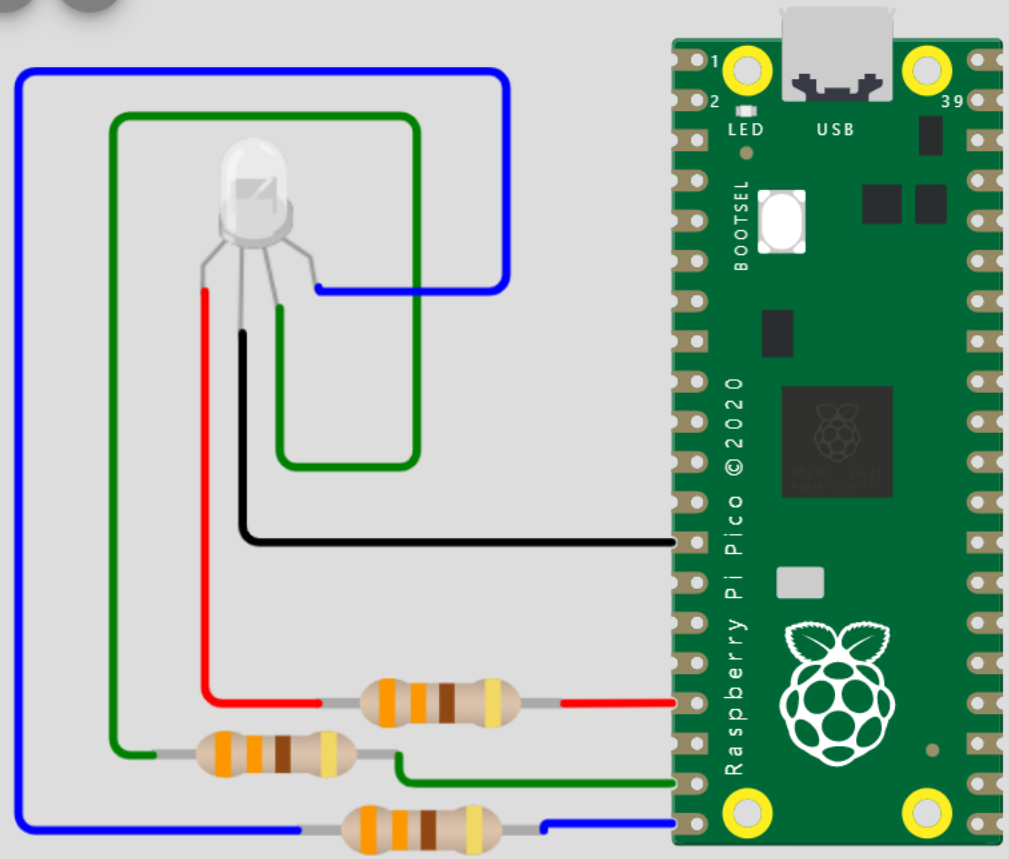
```
1  from machine import Pin, PWM
2  from utime import sleep
3
4  red = PWM(Pin(13))
5  green = PWM(Pin(14))
6  blue = PWM(Pin(15))
7
8  red.freq(1000)
9  green.freq(1000)
10 blue.freq(1000)
11
12 def interval_mapping(x, in_min, in_max, out_min, out_max):
13     return (x - in_min) * (out_max - out_min) / (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)
25
```

Change these values in

THONNY and see the magic

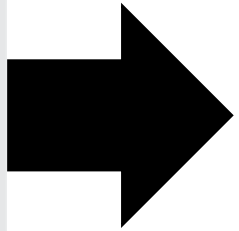
Simulation

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)

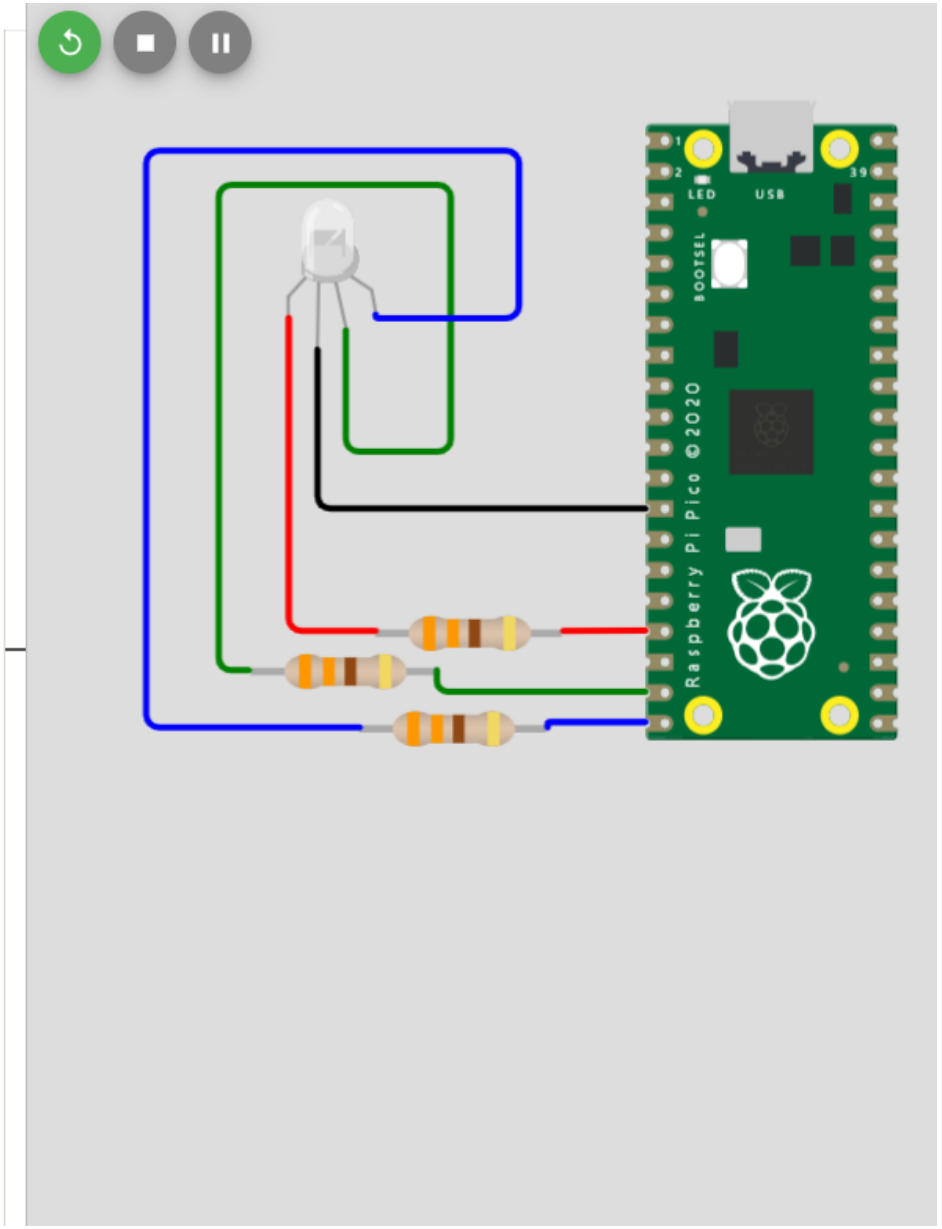


Check these colours in your

RGB LED

Colorful Light: Mystery of additive color mixing

```python
from machine import Pin, PWM
from utime import sleep

red = PWM(Pin(13))
green = PWM(Pin(14))
blue = PWM(Pin(15))

red.freq(1000)
green.freq(1000)
blue.freq(1000)

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 color_to_duty(rgb_value):
    rgb_value = int(interval_mapping(rgb_value,0,255,0,65535))
    return rgb_value

def color_set(red_value,green_value,blue_value):   #let the three primary colors work together.
    red.duty_u16(color_to_duty(red_value))
    green.duty_u16(color_to_duty(green_value))
    blue.duty_u16(color_to_duty(blue_value))

while True:
    color_set(255,0,0)
    sleep(1)
    color_set(0,255,0)
    sleep(1)
    color_set(0,0,255)
    sleep(1)
    color_set(0,255,255)
    sleep(1)
    color_set(255,255,0)
    sleep(1)
    color_set(255,255,255)
    sleep(1)
    color_set(128,0,0)
    sleep(1)
    color_set(128,128,0)
    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

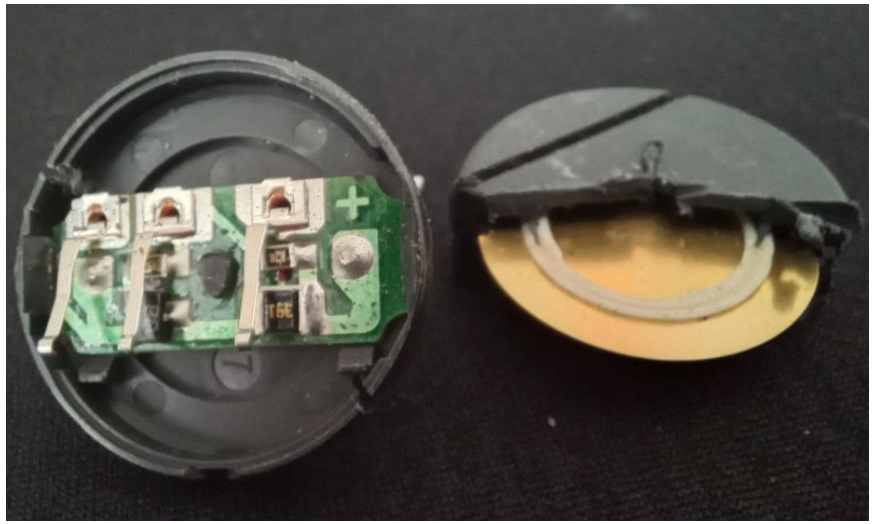
ACTIVE BUZZERS

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.

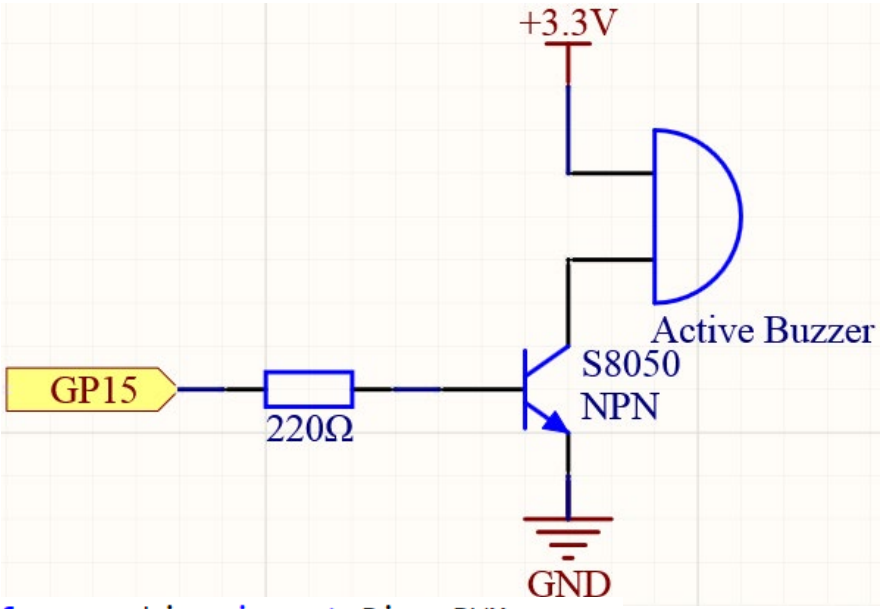
Custom Tone: Create a Melody

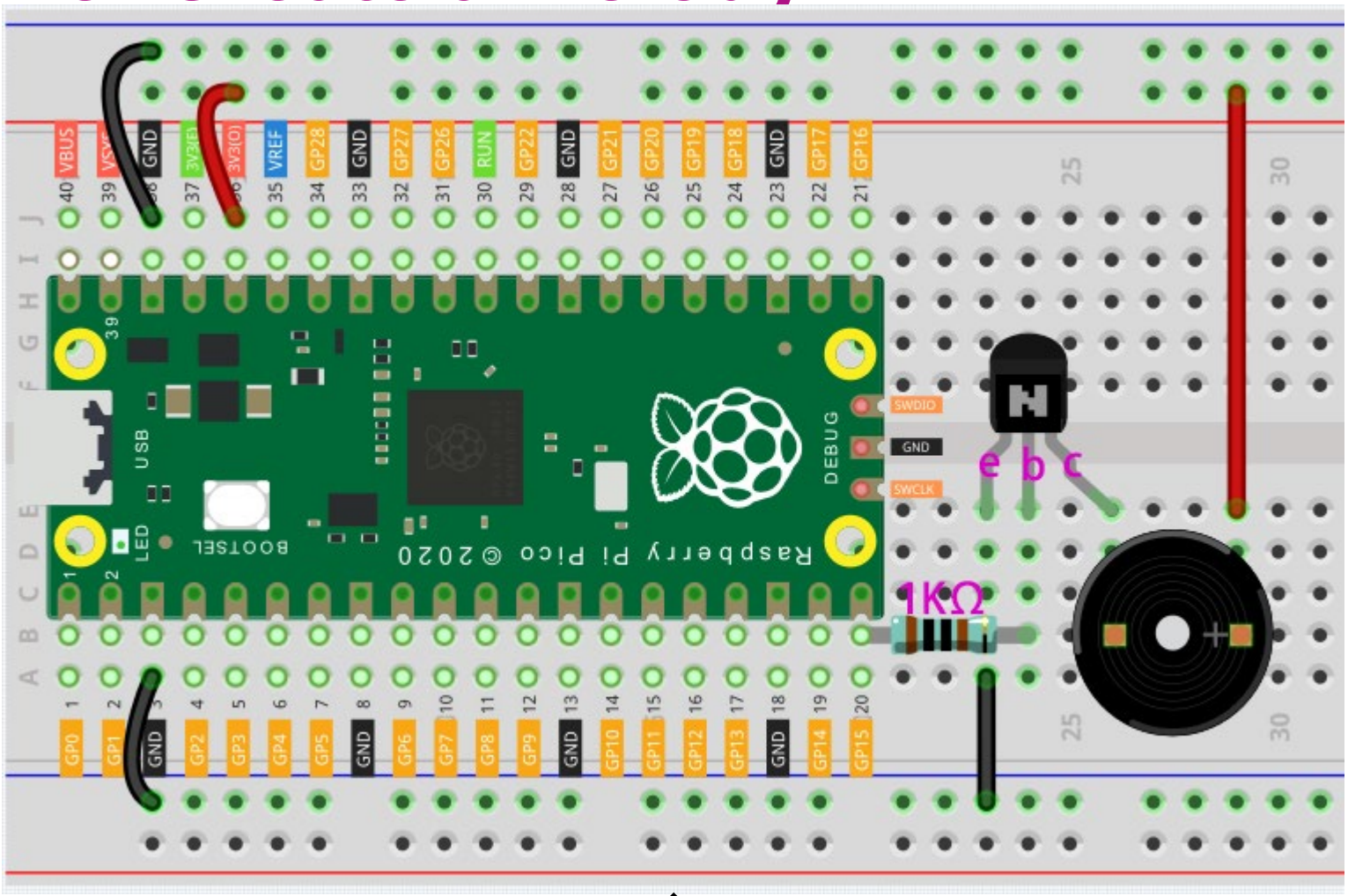
Inside an PASSIVE BUZZERS

.

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





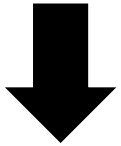

```
from machine import Pin, PWM
import utime

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.

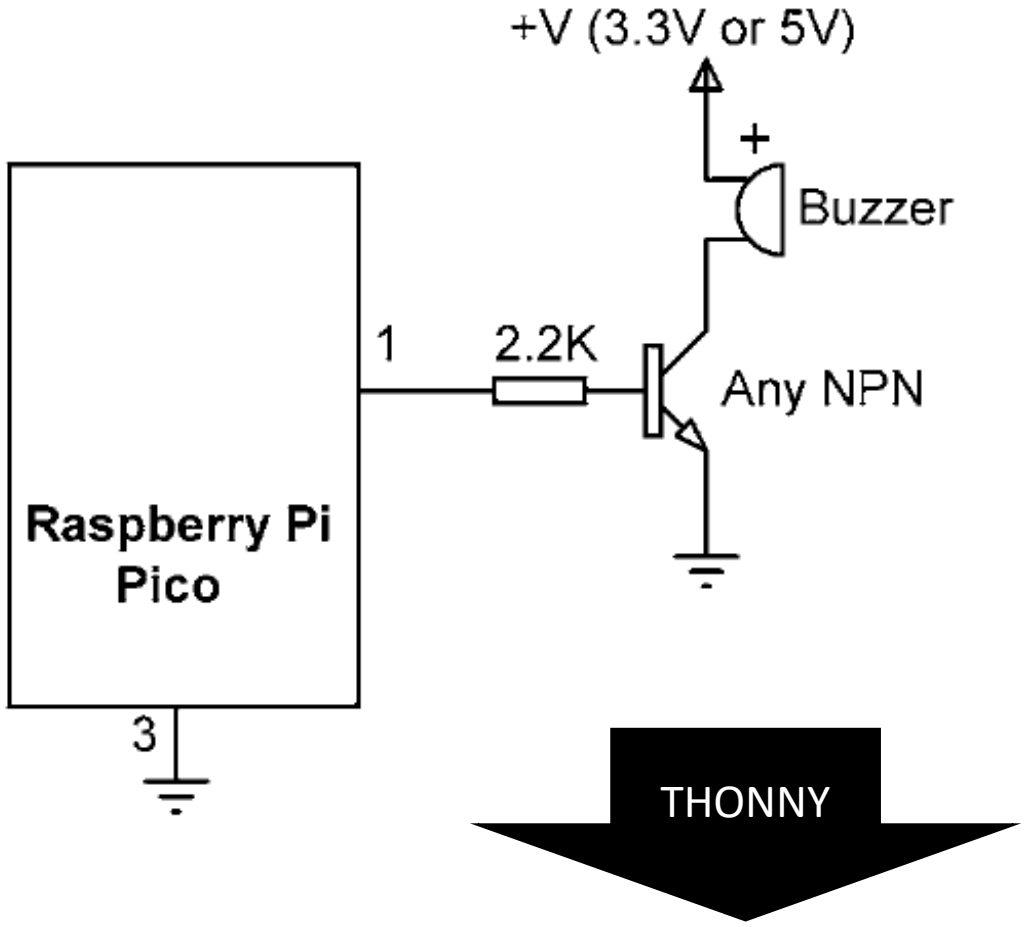
Notes	C_4	C_4#	D_4	D_4#	E_4	F_4	F_4#	G_4	G_4#	A_4	A_4#	B_4
Hz	261.63	277.18	293.66	311.13	329.63	349.23	370	392	415.3	440	466.16	493.88



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

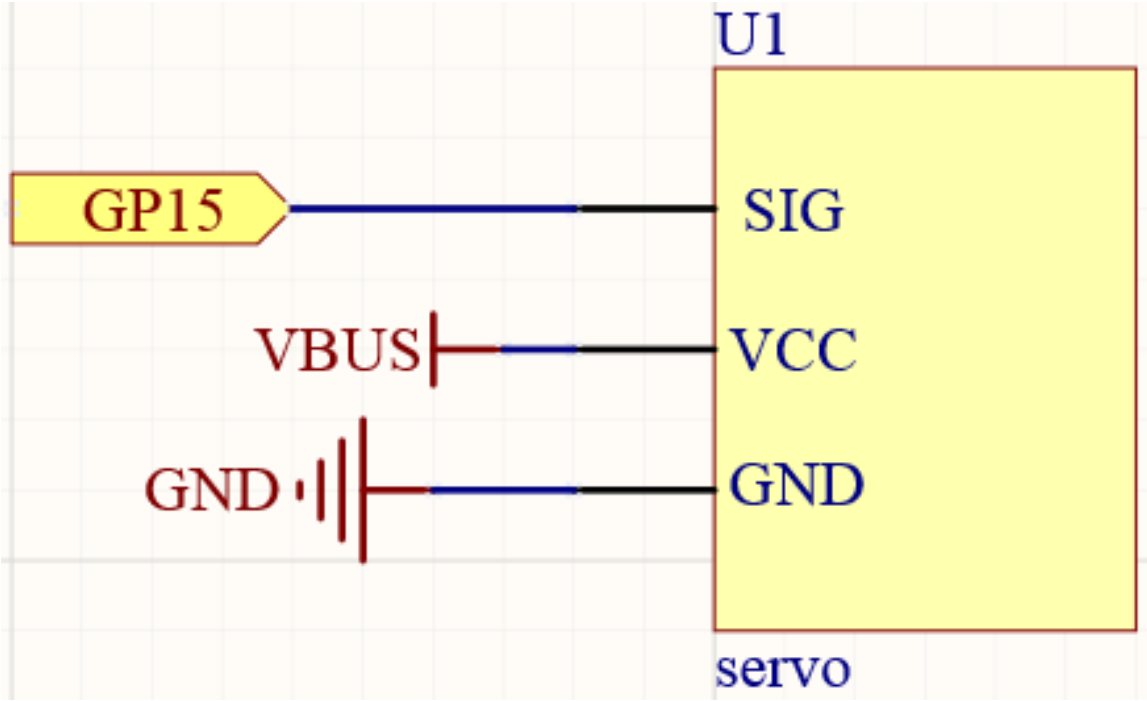


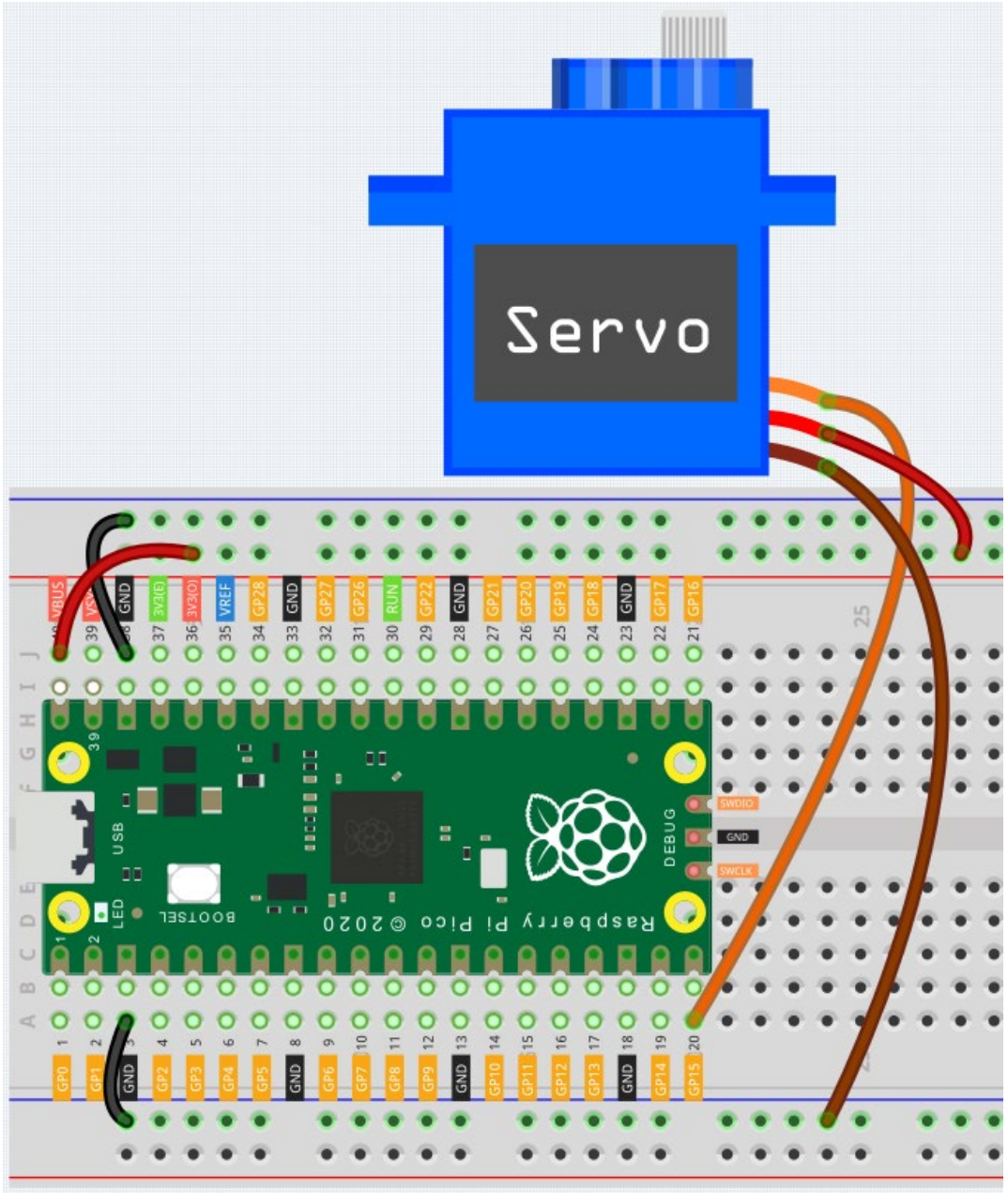

```
from machine import Pin, PWM
import utime

ch = PWM(Pin(0))            # PWM output at GP0
MaxNotes = 25
Durations = [0]*MaxNotes
#
# Melody frequencis
#
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
```

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

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.





Swinging Servo

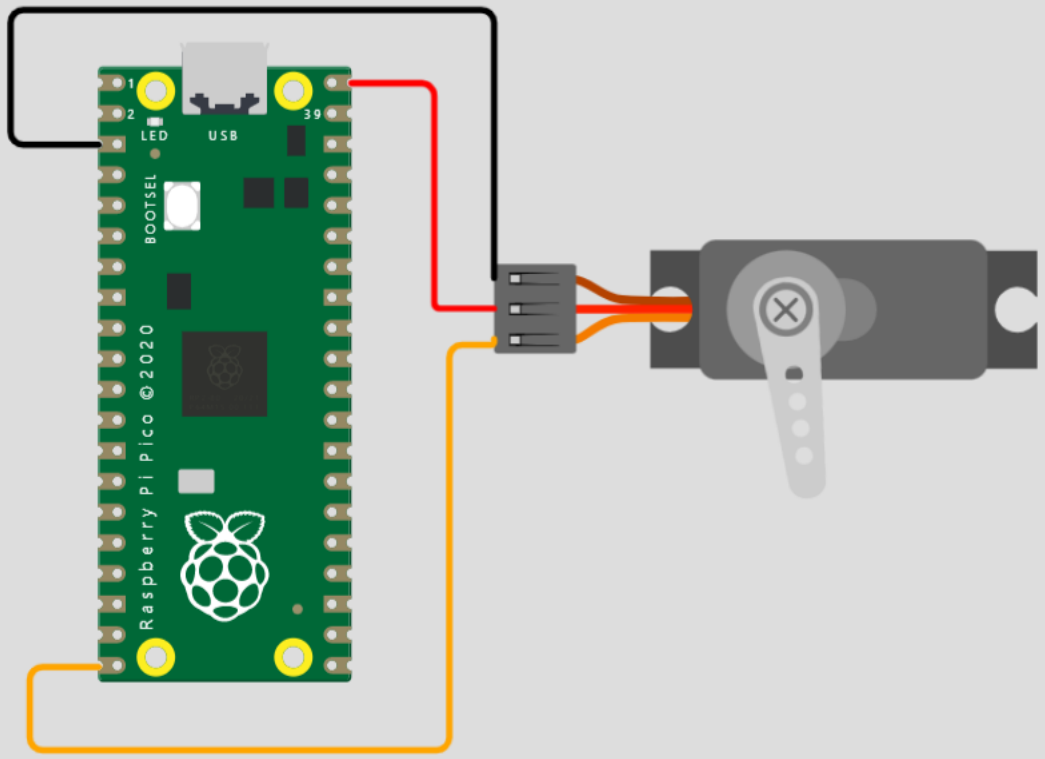
WOKWi SAVE SHARE Docs B

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

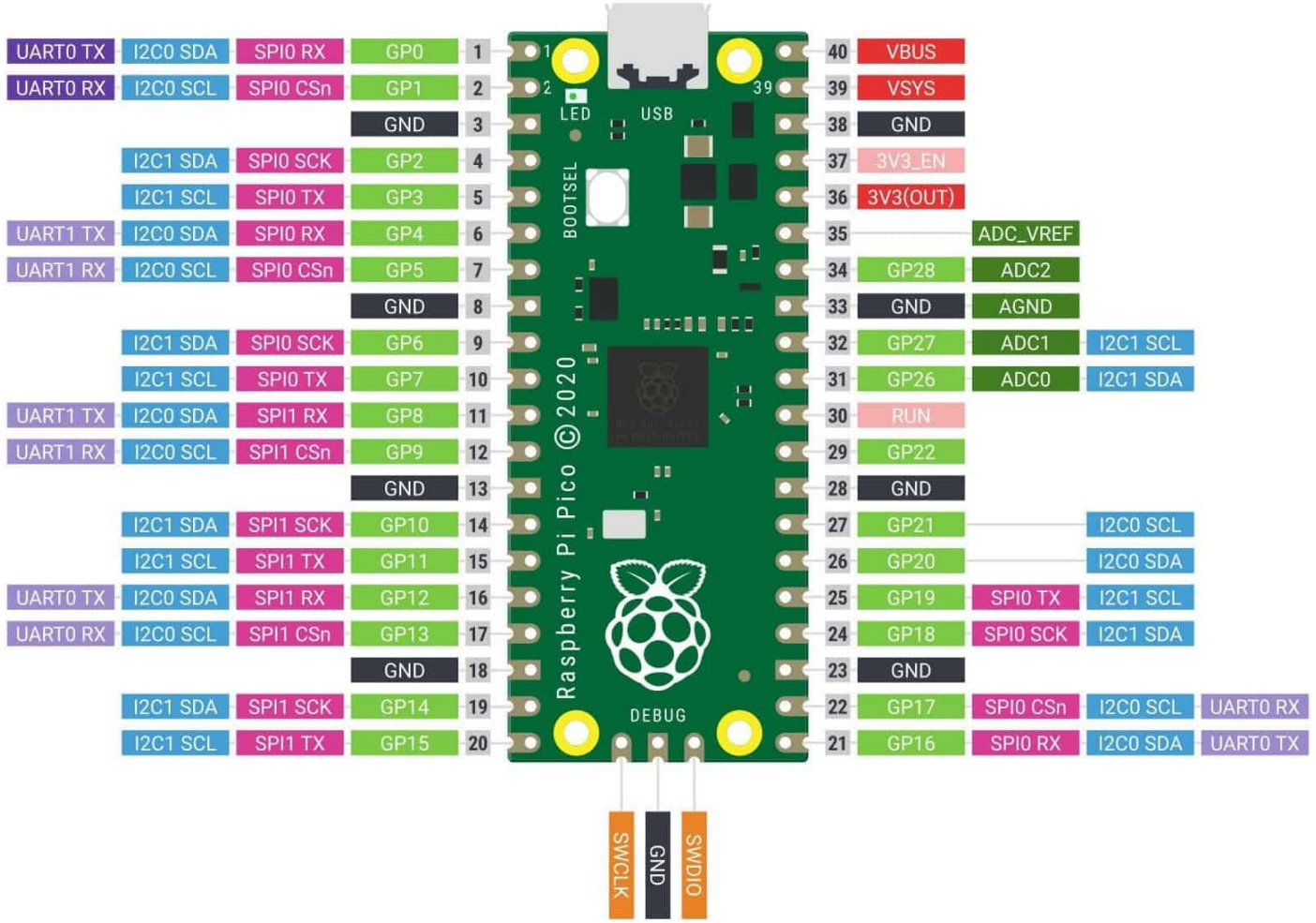
00:25.778 99%



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.

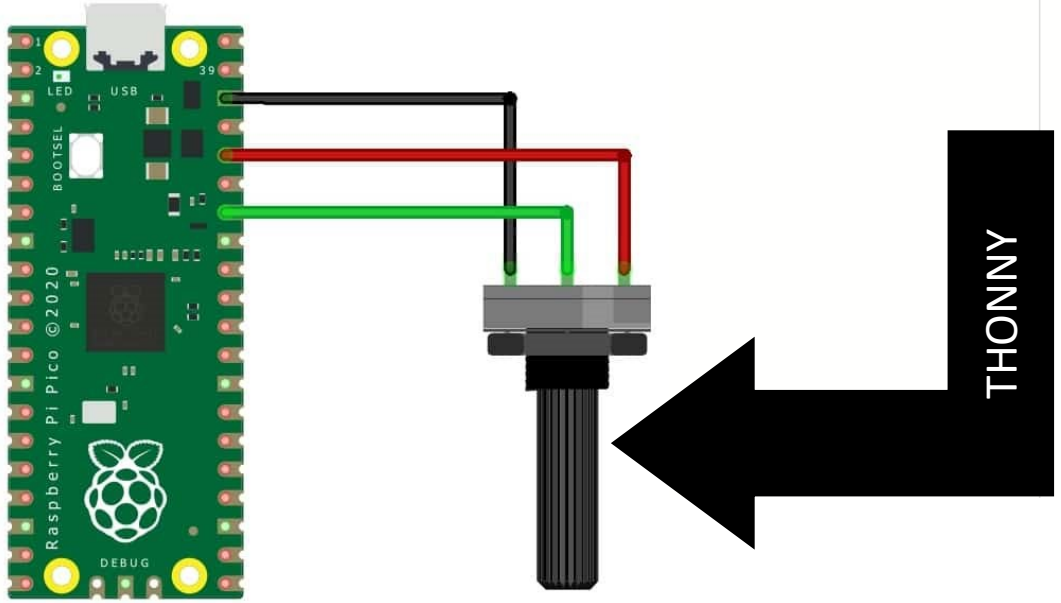
- The ADC conversion speed per sample is **2µs** that is **500kS/s**. The RP2040 microcontroller operates on a 48MHZ clock frequency which comes from USB PLL. So, its ADC takes a 96 CPU clock cycle to perform one conversion. Therefore, the sampling frequency is (96 x 1 / 48MHz) = 2 µs per sample (500kS/s).

Turn the Knob: Welcome to Analog World

WOKWI SAVE SHARE Docs B

main.py ● diagram.json ● PIO

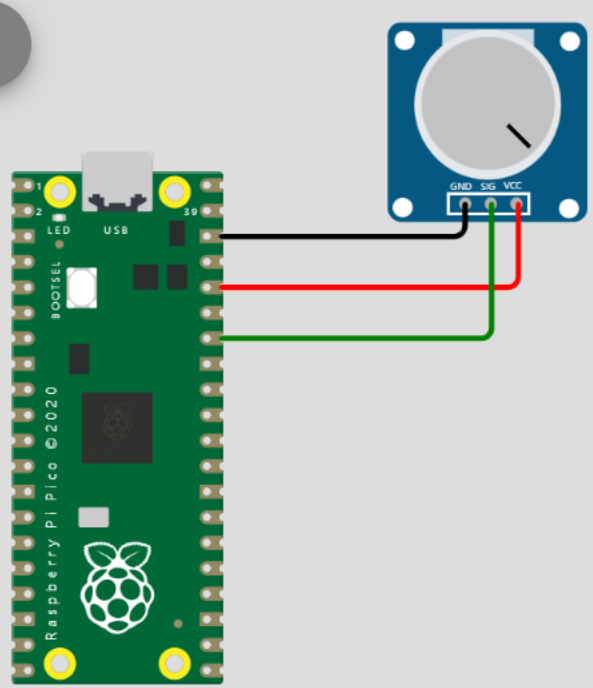
```
1  import machine
2  import utime
3
4  analog_value = machine.ADC(28)
5
6  while True:
7      reading = analog_value.read_u16()
8      print("ADC: ",reading)
9      utime.sleep(0.2)
10
```



THONNY

Simulation

00:52.243 99%




```
ADC:  65535
ADC:  65535
ADC:  65535
ADC:  65535
ADC:  65535
```

Turn the Knob: Control an LED using POT

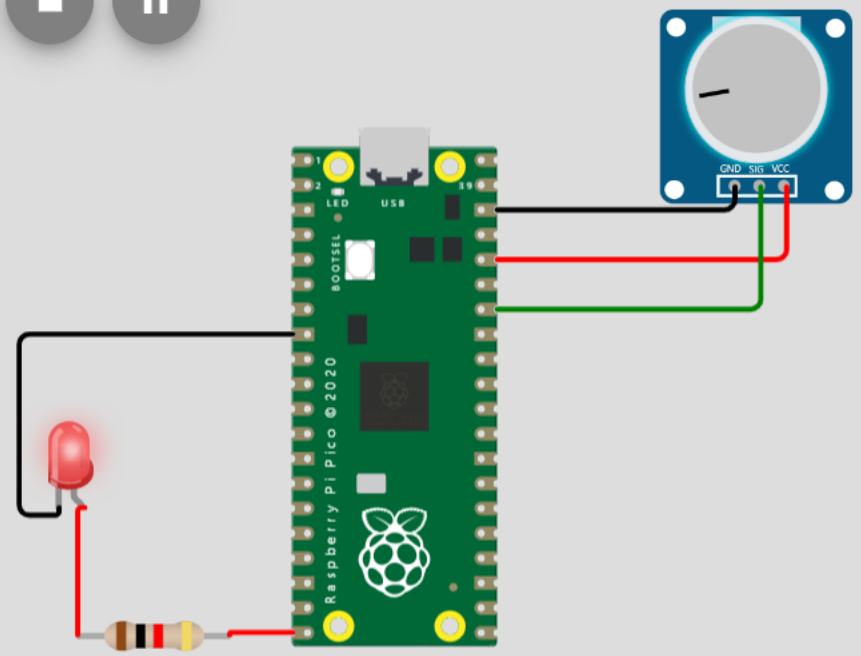
WOKWI SAVE SHARE Docs B

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)
```

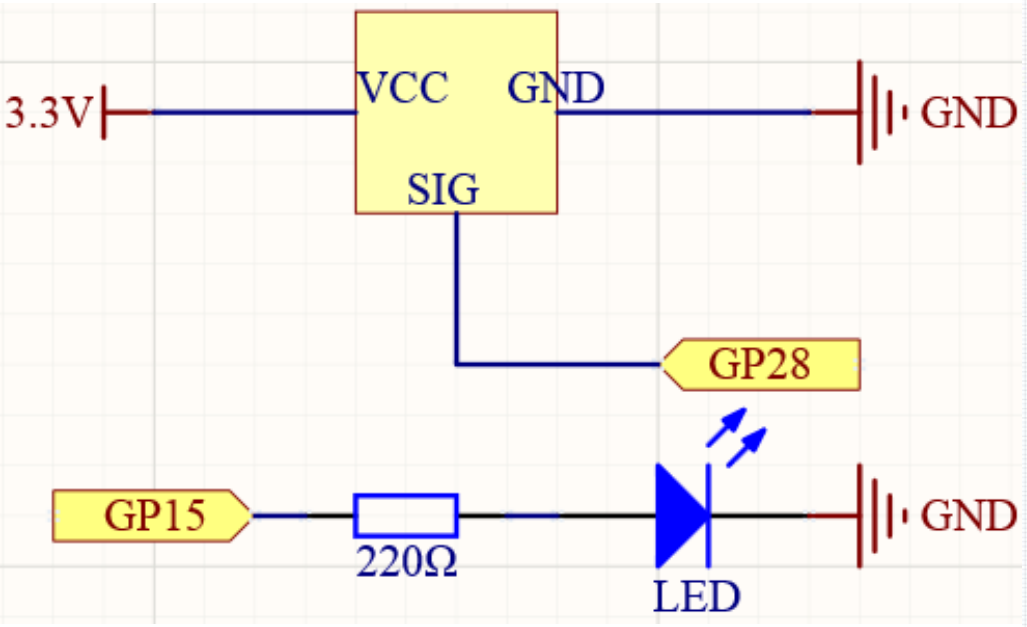
Simulation

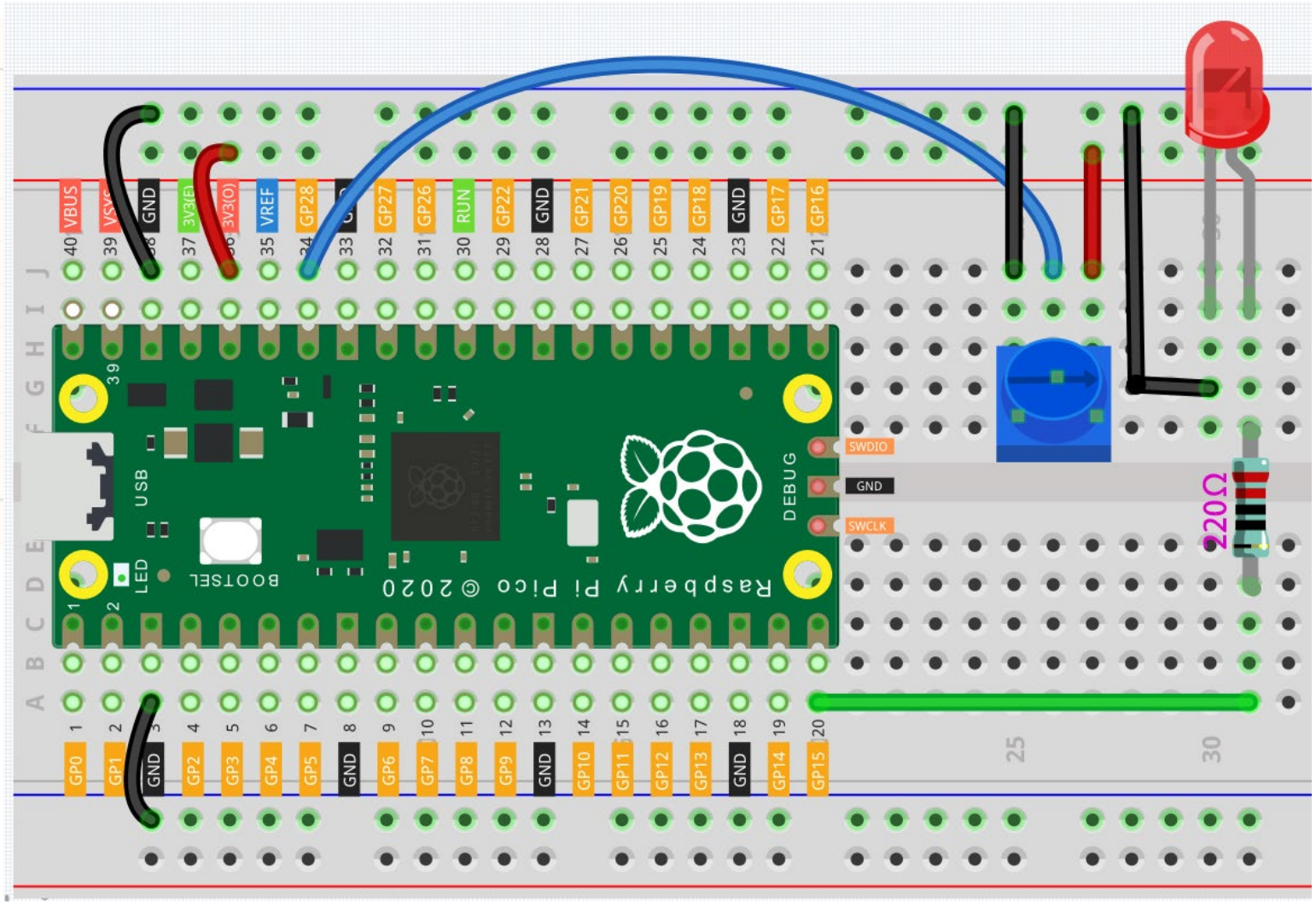
00:12.432 99%




```
8514
8514
8514
8514
```

Turn the Knob: Control an LED using POT





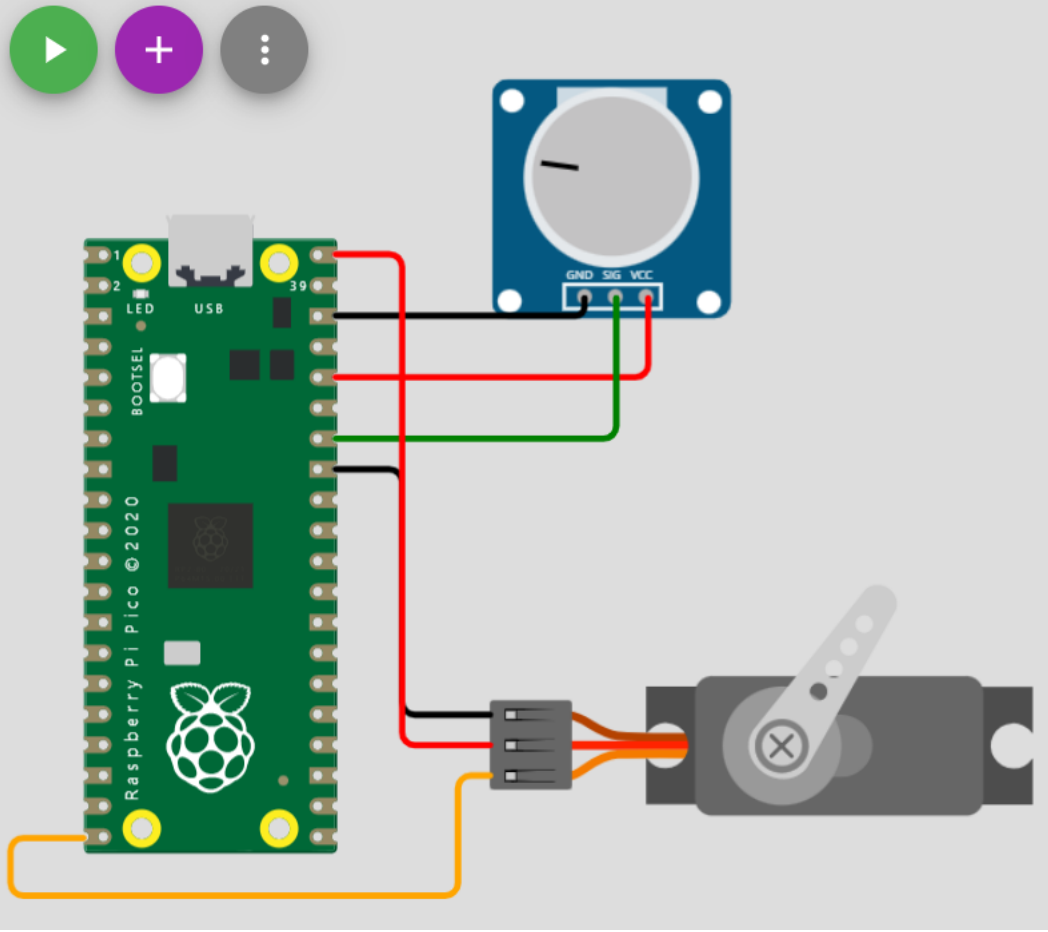
Swing the servo from left to right using POT

WOKWI SAVE SHARE Docs B

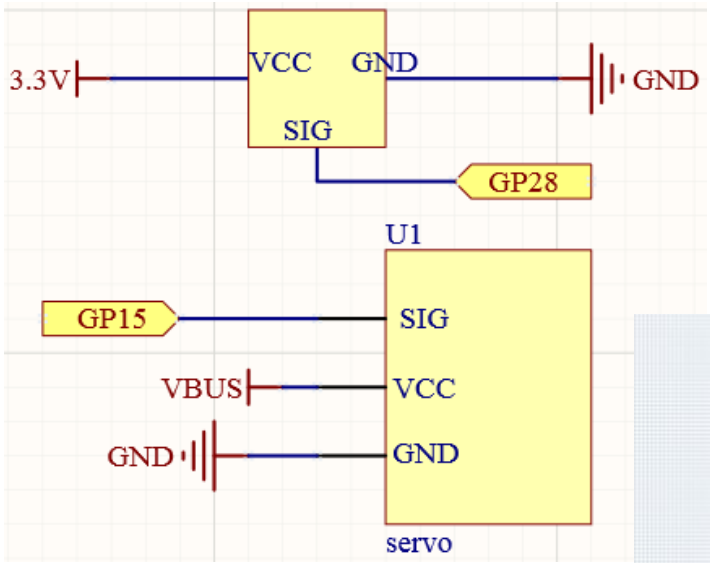
main.py ● diagram.json ●

```
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



.