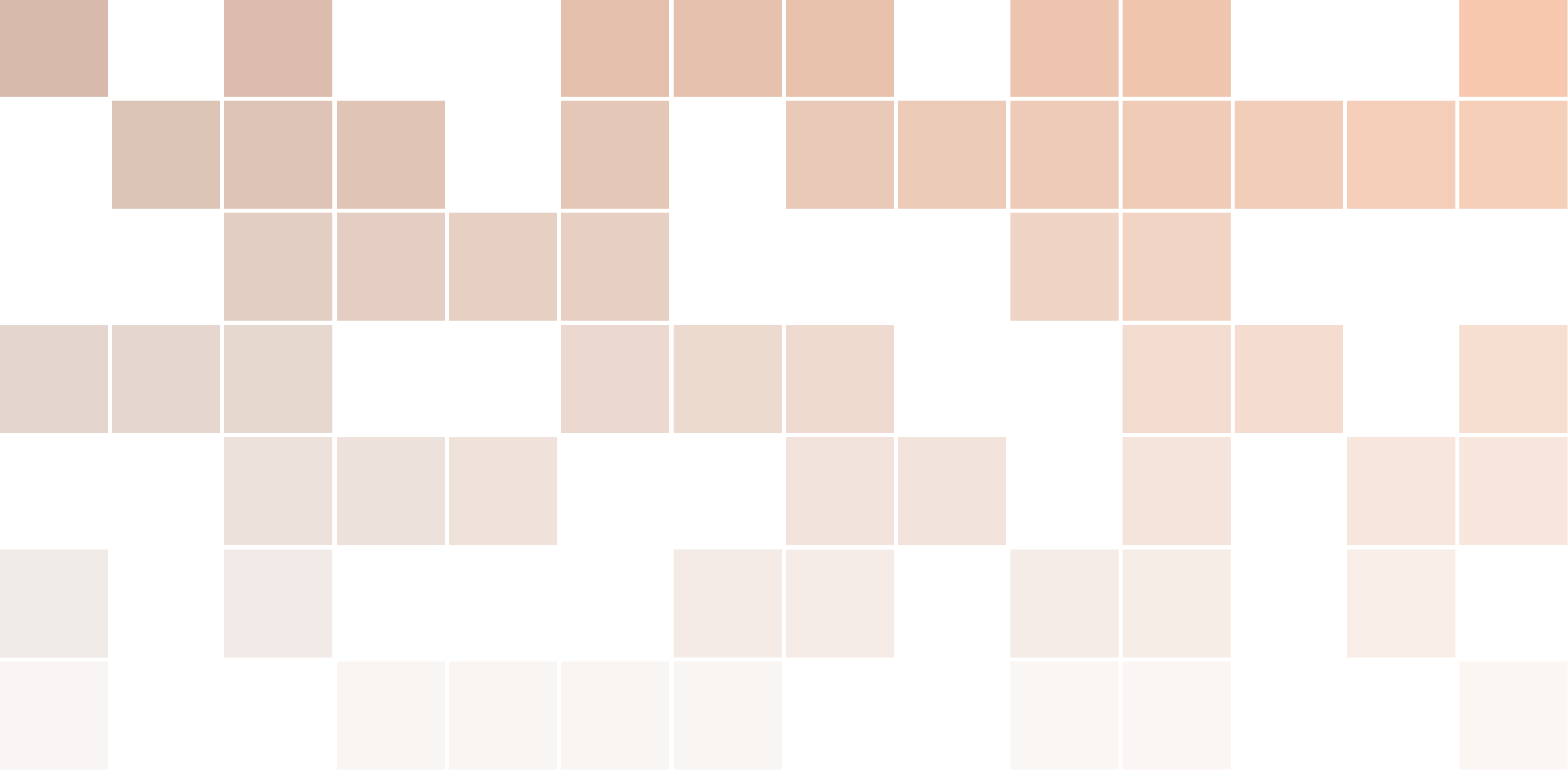
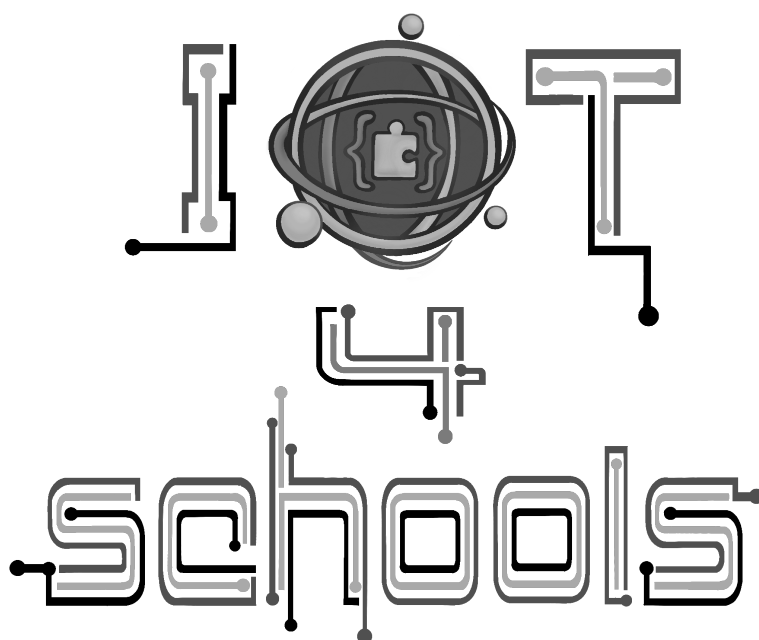


Technical guide about Raspberry Pi Pico and microPython



Authors: [1]Angelika Tefelska, [1]Dariusz Tefelski
Contributors: [2]Chrissa Papasarantou, [2]Rene Alimisi

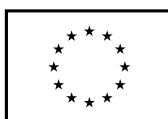
IoT4schools Consorcium:
[1]Warsaw University of Technology,
[2]EDUMOTIVA,

Project title: “Bringing the Internet of Things in school education as a tool to address 21st century challenges”, project number: 2023-1-PL01-KA220-SCH-000154043, Erasmus+ KA220-SCH.

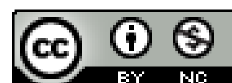
The European Commission’s support to produce this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

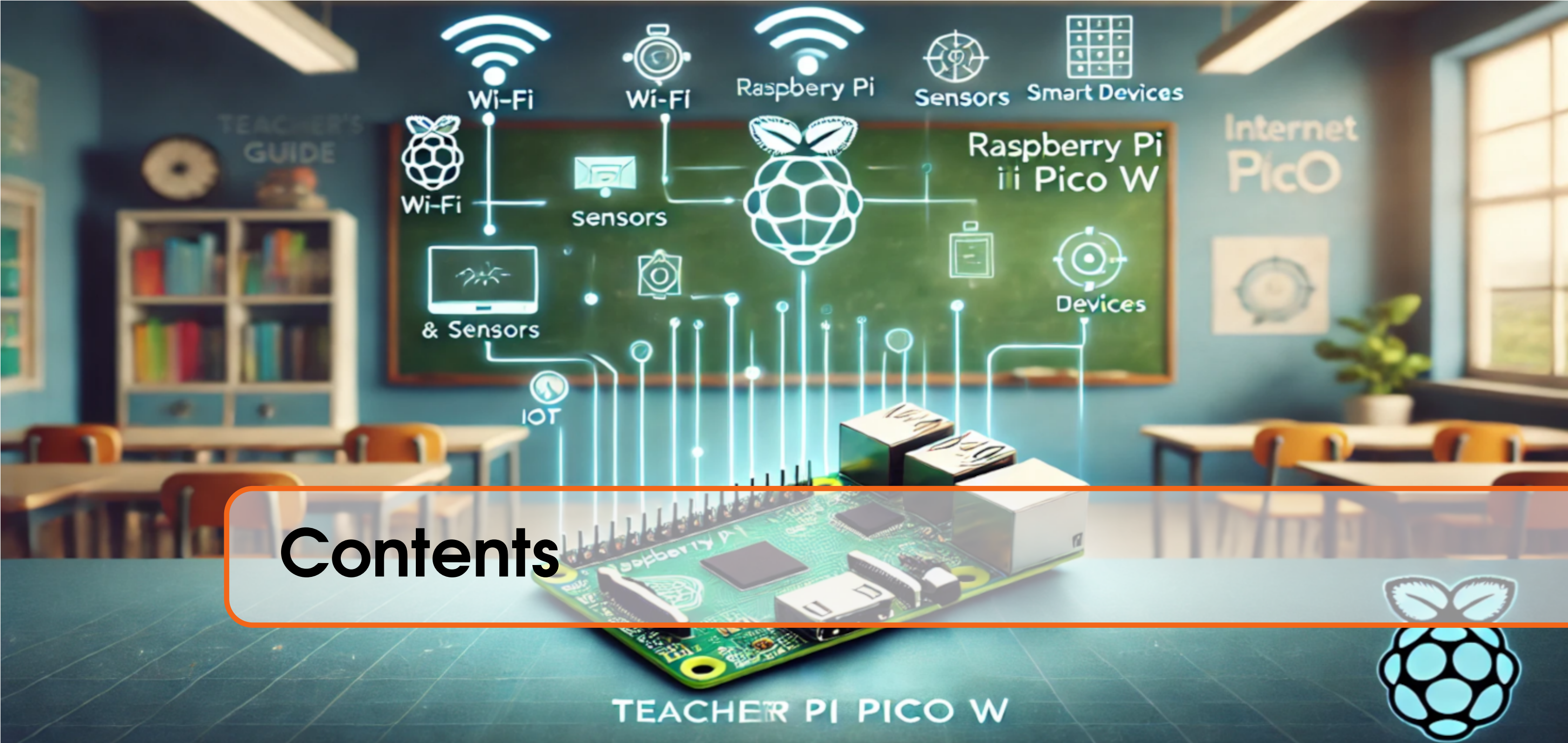
License: CC BY-NC 4.0 LEGAL CODE, Attribution-NonCommercial 4.0 International

LaTeX template was taken from: https://www.latextemplates.com/template/legrand-orange-book
The chapter images generated with DALL-E, OpenAI.



Co-funded by
the European Union





Contents

1 Introduction 5
1.1 Raspberry Pi Pico board 5
1.2 Installation 7

2 Basic electronic components 8

3 Introduction to MicroPython language 13

4 The digital signals 19
4.1 Example 1: blinking LED project 20
4.2 Example 2: LED turned on/off with push button 22
4.3 Example 3: Light switched on by a motion sensor 26
4.4 Pulse Width Modulation (PWM) 28

5 The analog signals 34
5.1 Example 5: temperature measurement 34

6 Interrupts 38
6.1 Example 6: Sound signals at traffic lights 39

7 Sensors 46
7.1 Example 7: parking sensor 46
7.2 Example 8: GPS 51
7.3 Example 9: Weather station 54
7.4 Example 10: Indoor air quality measurement 57
7.5 Example 11: Temperature measurement using external A/D converter 61

7.6 Example 12: Temperature measurement using 1-wire bus 71

8 Actuators 78

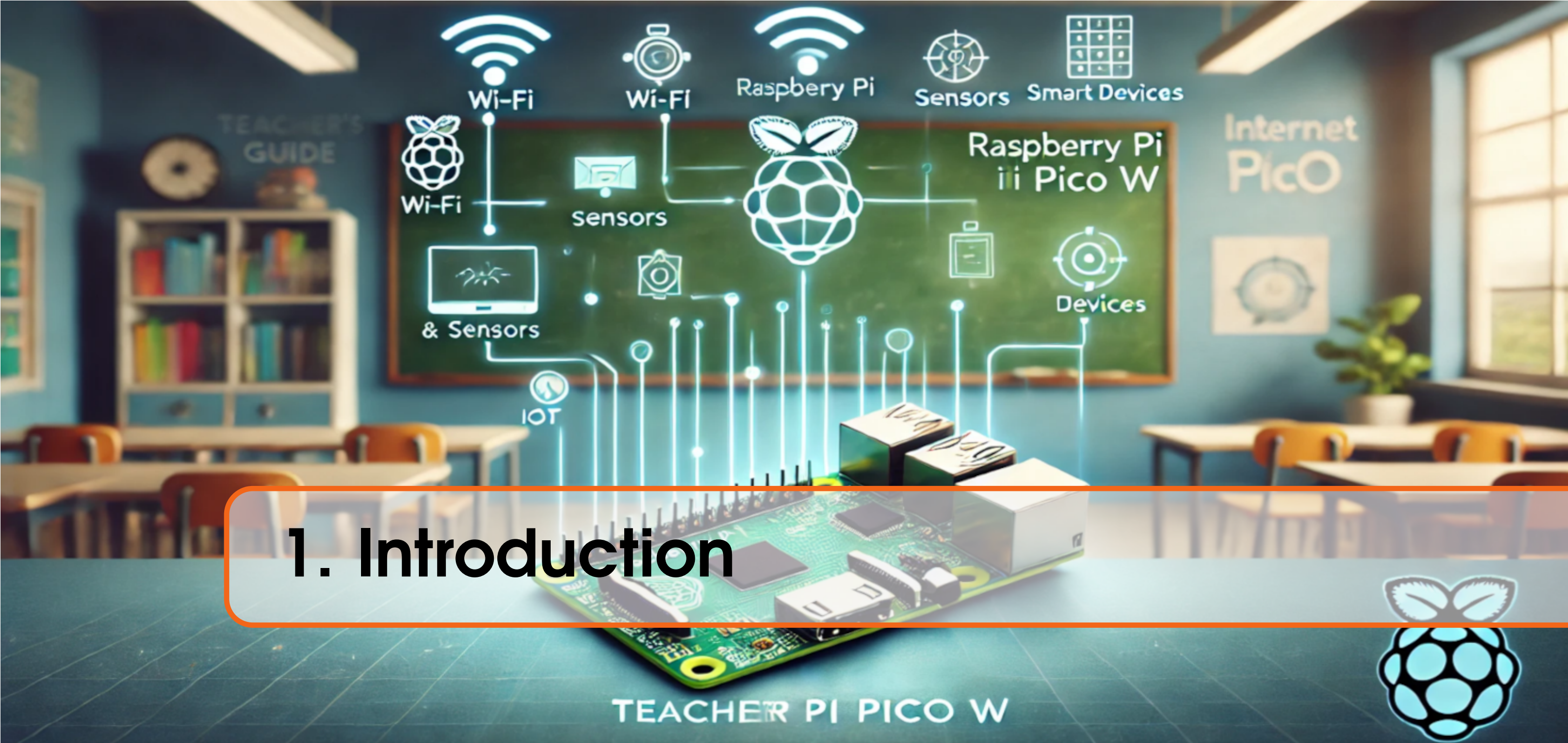
8.1 Example 13: Servo motor 78

8.2 Example 14: Smart ventilation 81

9 Wireless communication 85

9.1 Example 15: Bluetooth module 85

9.2 Example 16: Adafruit IO 85



1. Introduction

This guide is an introduction to the Raspberry Pi Pico W board, which can be used in Internet of Things (IoT) projects. The development of digitalization shows that we are increasingly using smart solutions that improve our quality of life and allow us to save resources such as electricity or water. Familiarizing yourself with the subject of IoT is extremely important in the modern world. One of the boards that allows you to enter the world of IoT technology is the Raspberry Pi Pico W, which is compact, easy to use and cheap. Hence, this guide will focus on presenting the Raspberry Pi Pico W board and the MicroPython language used to program it. The guide is dedicated to pupils, students, teachers, educators and hobbyists who have not previously dealt with the Raspberry Pi Pico W board. We encourage you to learn through practice, i.e., by performing sample projects with a guide in parallel, which will allow you to quickly gain fluency in using the board. Enjoy learning and have fun!

1.1 Raspberry Pi Pico board

In 2021, the **Raspberry Pi Pico** series boards were launched, which were a breakthrough product offered by the Raspberry Pi Foundation. Previous versions of Raspberry Pi boards were System-on-Chip (SoC), which were miniature computers. On the other hand, the Raspberry Pi Pico series are a completely different type of boards - they are based on microcontrollers and are a great alternative to Arduino boards. The heart of the Raspberry Pi Pico board is the proprietary **RP2040** system - a dual-core ARM Cortex M0+ microcontroller operating at 133 MHz, equipped with 264 KB of SRAM and 2 MB of Flash memory. The goal was to provide an efficient microcontroller with significant computing power that would meet the expectations of hobbyists from around the world. In addition to the excellent RP2040 system, it is worth noting that the board was equipped with 26 *General-Purpose Input/Output* (GPIO) pins, including 16 PWM channels and communication interfaces such as I2C, SPI, UART. If you are not familiar with the board equipment mentioned, it does not matter, we will discuss all the elements later in the guide. The layout of the mentioned elements on the Raspberry Pi Pico board is shown in Figure 1.1.

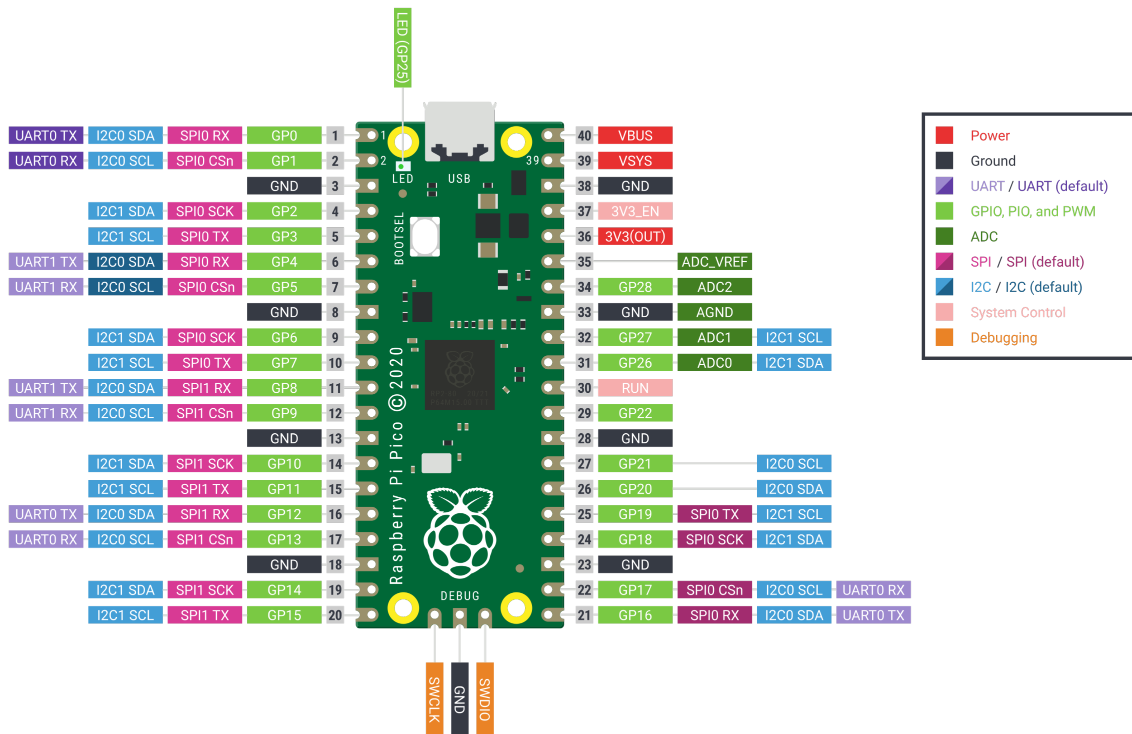


Figure 1.1: The pinout of Raspberry Pi Pico. Source of image: https://www.raspberrypi.com.

The Raspberry Pi Pico series consists of 4 small boards (not much larger than a pendrive) shown in Figure 1.2.

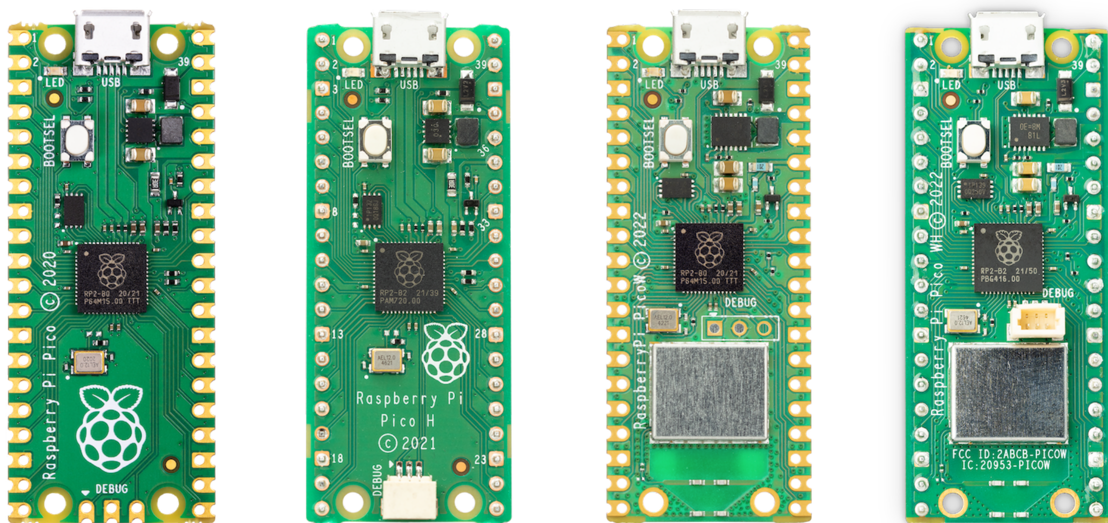


Figure 1.2: The 4 version of Raspberry Pi Pico boards. Source of image: https://www.raspberrypi.com.

The first board on the left is the *Raspberry Pi Pico*, the next board is the *Raspberry Pi Pico H*, which differs only in that the pins are already soldered and dedicated connector

for debugging purposes is present. If you order the standard version of the Raspberry Pi Pico, you will have to solder the pins yourself. If you have never soldered, do not worry. On the Internet, you will find many example videos on how to solder pins, e.g. https://www.youtube.com/watch?v=zbhOyCA_4lg. If, after all, you would prefer not to solder pins, a better choice would be to choose boards marked with letter *H*. However keep in mind, that you will not be able to solder pins for debug purposes to this board and need to use dedicated connector (cable and connectors available with Raspberry Pi Debug probe device). The third board from the left is the *Raspberry Pi Pico W*, which additionally has a built-in Wi-Fi and Bluetooth module, which is extremely important in Internet of Things (IoT) projects. In this guide, we will mainly focus on the Raspberry Pi Pico W version, which will be necessary for IoT projects. The fourth board is the *Raspberry Pi Pico WH*, which differs from the Raspberry Pi Pico W only in the pins that are already soldered (and presence of debug probe connector). Therefore, if you do not intend to solder pins, we recommend the Raspberry Pi Pico WH version for IoT projects.

All Raspberry Pi Pico boards are equipped with a microUSB connector, which is used for programming and powering the board. The board can be programmed in C/C++ or MicroPython. In this guide, we will focus on MicroPython as the easiest option for beginners. For those willing to learn Raspberry Pi Pico programming in C/C++, we recommend the guide on this topic available here: https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf.

There are also third-party boards available on the market, which usually contain additional elements and connectors, e.g. allowing for easy battery connection, or USB-C connector instead of microUSB.

News 1.1.1

Recently, a new version of the Raspberry Pi Pico 2W has been released, which differs from its predecessor mainly in: the microcontroller (the RP2040 based on the ARM Cortex-M0+ Dual-Core 133 MHz has been replaced with the RP2350 based on the ARM Cortex-M33 Dual-Core 150 MHz), the amount of SRAM and Flash memory increased (2x more) and the number of PWM channels increased from 16 to 24. As a bonus new RISC-V architecture cores appeared for more advanced users as an altarnative choice: either ARM or RISC-V cores, although greater capabilities remain on ARM cores.

However, the new version of the RP2350 microcontroller has one problem with pull-down resistors (see the chapter on digital signals). Despite the aforementioned problem, which can be mitigated, the new version is an interesting choice if you intend to create more extensive programs and need more memory. This guide uses the Raspberry Pi Pico W, which is enough for most IoT projects.

1.2 Installation

Before we start programming the Raspberry Pi Pico, you need to install:

- Thonny editor (`https://projects.raspberrypi.org/en/projects/getting-started-with-the-pico/2`)
- Raspberry Pi Pico firmware (`https://projects.raspberrypi.org/en/projects/getting-started-with-the-pico/3`).

2. Basic electronic components

Before we begin our adventure with programming the Raspberry Pi Pico W, let's get to know the basic electronic components we will be using:

- **Resistor** - is an element used to reduce the current flowing through it. It dissipates energy as heat energy. A resistor is a linear element. The larger the resistor we use, the less current will flow through the circuit, according to Ohm's law:

$$I = \frac{V}{R} \tag{2.1}$$

  where: V - voltage on the element (in some countries marked with the letter U), I - current intensity, R - resistance.
  The resistance value of a resistor can be read from the color code located on the resistor as shown in the example in Fig. 2.1. For each resistor, we will read the resistance value from left to right. For example, in Fig. 2.1, the top resistor has the following band colors: yellow (value 4), pink (value 7), red (x100), and silver (tolerance=10%). So its value is $47x100\Omega = 4700\Omega = 4.7k\Omega$ and its tolerance is 10%. The tolerance shows how much the real resistance can differ from the nominal value (eg. for a 10% $4.7k\Omega$ resistor, its actual resistance will be between $4.23k\Omega$ ($0.9 \cdot 4.7k\Omega$) and $5.17k\Omega$ ($1.1 \cdot 4.7k\Omega$).)
- **LED diode** - is a semiconductor element that converts current into light. More precisely, electrons passing from a higher energy level to a lower one in a semiconductor emit a photon and depending on the energy of the photon, we obtain different colors of LEDs. LEDs are polarized elements, which means that current will flow through them in only one direction. Fig. 2.2 shows a LED with a cathode and anode lead.

  The typical LED diode needs voltage near 1.7 V (V_{LED}) and current between 1 to 15 mA. The voltage on the Raspberry Pi Pico pins is **3.3V** (V_{RP}), so you need to use a resistor to limit the current flow:

$$R = \frac{V_{RP} - V_{LED}}{I} = (107; 1600)\Omega \tag{2.2}$$

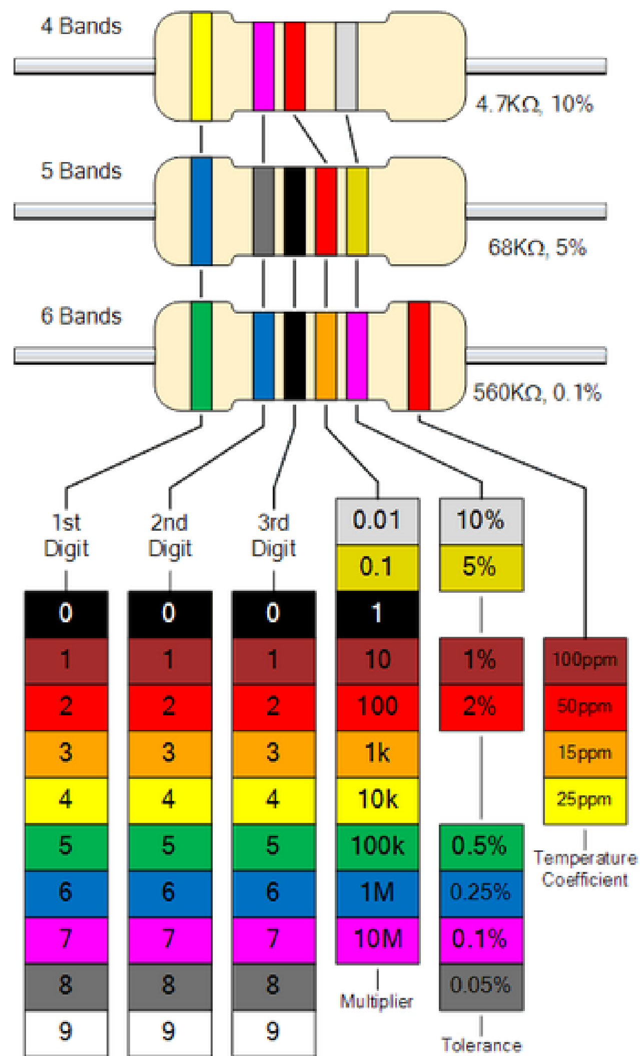


Figure 2.1: The color code of resistors. Source: https://electronicspost.com/resistor-color-code/

Figure 2.2: A LED with anode and cathode marked (left image) and a diode symbol (right image). Source: https://www.build-electronic-circuits.com/what-is-an-led/ .

- **Potentiometer** - is a variable resistor that allows you to adjust the resistance between two legs and thus divide the voltage. It consists of three legs, where two legs are connected to the resistance path (see Fig. 2.3) and the third leg to the slider. By adjusting the position of the slider, either with a knob or a screwdriver depending on the type of potentiometer, we divide the resistance path into two resistors connected in series. In this way, if we connect the first leg to 3.3V and the third leg to 0V, we will get a voltage between 0V and 3.3V on the middle leg, depending on the position of the knob.

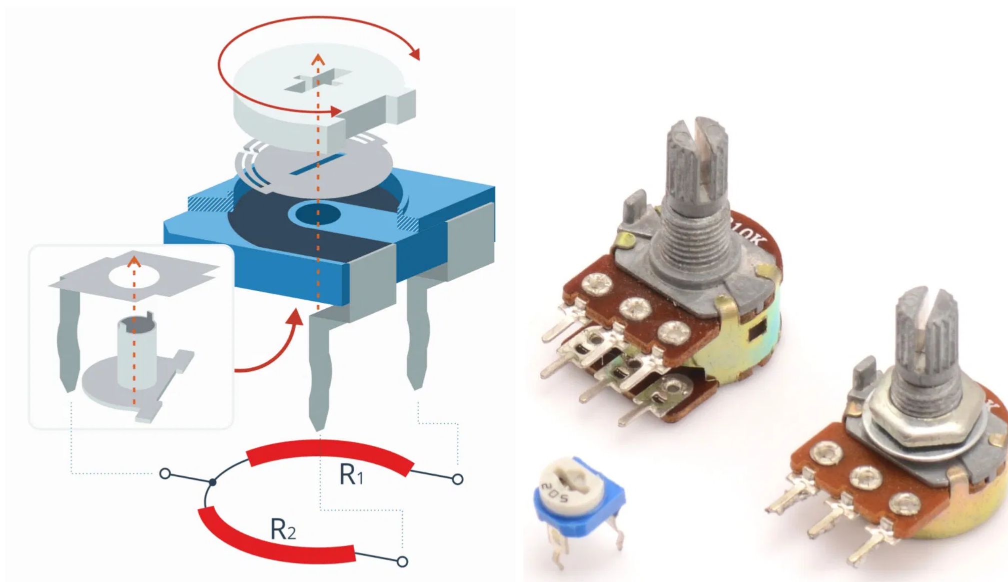


Figure 2.3: Principle of operation of a potentiometer. Source: https://forbot.pl/blog/leksykon/potencjometr .

- **Capacitor** - is an element that accumulates an electric charge. It consists of two plates and a dielectric between the plates (see left Fig. 2.4). Capacitors are used, among other elements, to e.g. filter a signal (the capacitor does not pass the DC voltage, and pass the ripples, hence it can smooth out the voltage, if connected to ground) or to create resonant circuits (extracting a signal with a specific frequency).

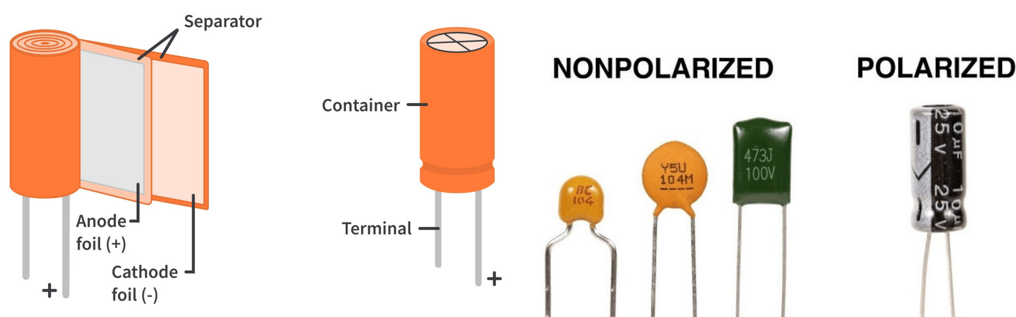


Figure 2.4: Left image: internal structure of a capacitor. Source of image: https://www.circuitbread.com. Right image: examples of polarized and non-polarized capacitors. Source of image: https://www.ariat-tech.pl

Capacitors can be divided into: capacitors which require proper polarization (e.g. electrolytic capacitors) and those which don't (e.g. ceramic or foil capacitors). These which need proper polarization should only be connected in one specified direction (see markings on the capacitor housing). The electrolytic capacitors are characterized by high capacity but they are not efficient with high frequency signals due to the dissipation factor. Ceramic capacitors do not dry out but they are not efficient with filtration of the low frequencies, because they have smaller capacitance.

- **Button** - is an element that closes the circuit when the button is pressed. When the button is not pressed, the circuit is open. The construction of the button is shown in Fig. 2.5.

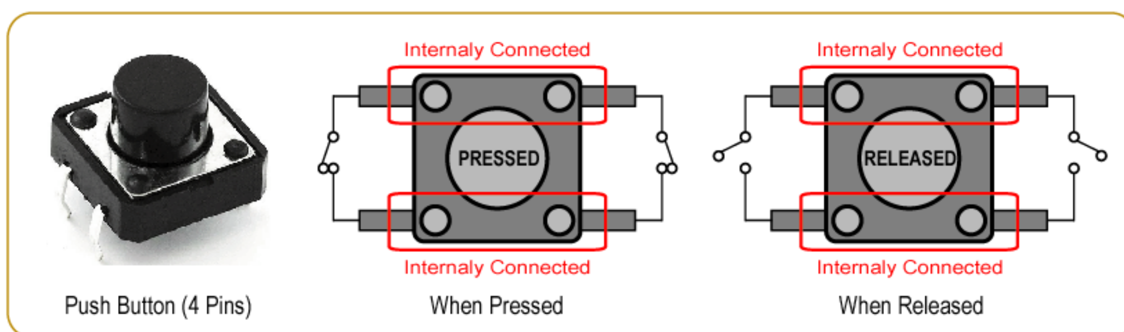


Figure 2.5: Push button operating principle. Source: https://gmostofabd.github.io/8051-Push-Button/.

- **Breadboard** - is a type of board used to connect electronic components, i.e. to build electronic circuits. The board consists of many holes that are connected vertically as shown in Fig. 2.6 (black line).

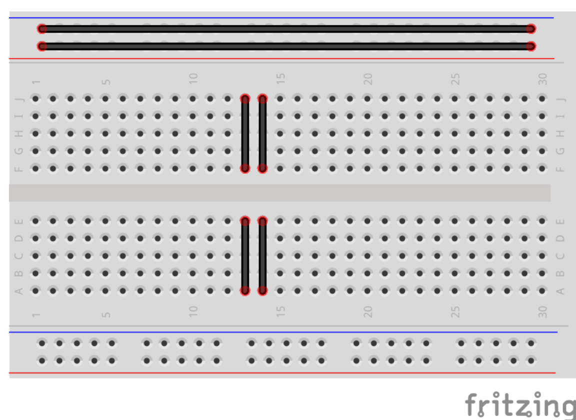


Figure 2.6: Connected holes are marked with black lines on the breadboard.

So the holes are connected in columns but not in rows. The board consists of two parts separated by a large gap. These two parts are not connected to each other. On the edges of the board there is a section for connecting power and ground. It is usually marked with two lines: red and blue. In this case, the holes are connected horizontally and not vertically. That is, along the red line, all the holes are connected and we usually connect the power supply here, i.e. 3.3V in the case of the Raspberry Pi Pico. Similarly, all the holes along the blue line are connected and we usually

connect ground there. Fig. 2.7 shows the correct placement of sample electronic components so that their pins are not shorted.

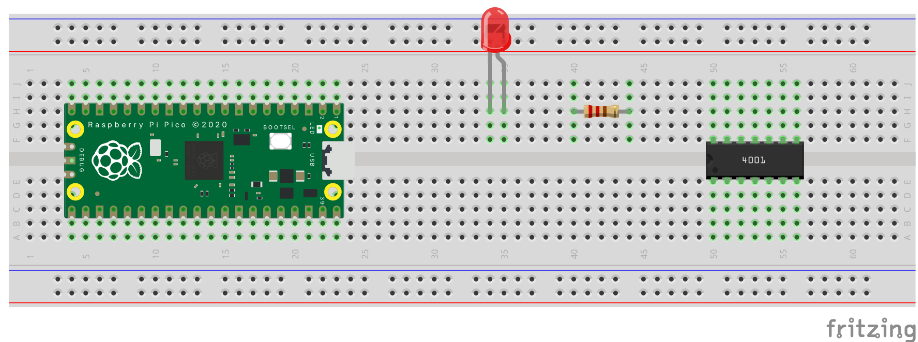


Figure 2.7: Example of placing elements on a breadboard so that the legs are not shorted.

- **Sensors** - are elements that allow for the measurement of various physical quantities, e.g. temperature, pressure, humidity, distance, movement, heart rate, etc. Sensors return measured values as: analog signals or digital signals usually transmitted using data communication interfaces (bus).
- **Actuators** - are elements that perform movement, e.g. servomechanisms, DC motors.
- **Shields** - are boards that extend the functionality of the basic board. There are many types of shields. At the beginning, a very useful shield is the GPIO Expander For Raspberry Pi Pico (e.g. from Waveshare *Pico to hat* - see Fig. 2.8), which allows you to connect components to the Raspberry Pi Pico without having to place the Raspberry Pi Pico on the breadboard. This solution reduces the risk of damaging the Raspberry Pi Pico board by placing it incorrectly on the breadboard.

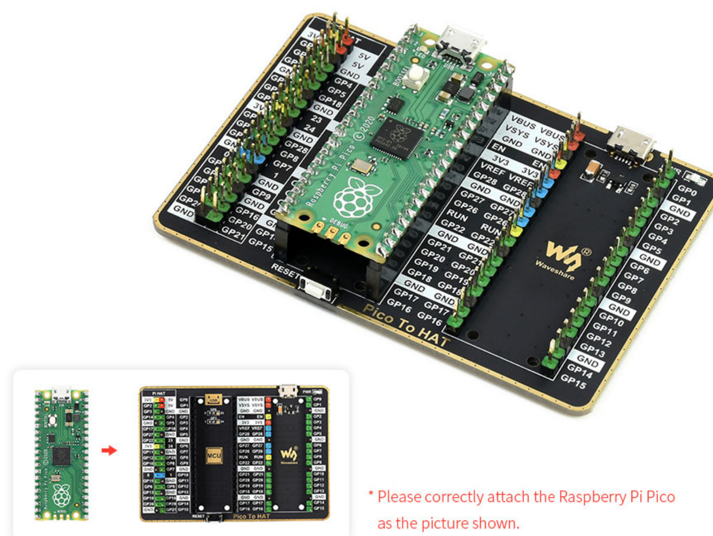
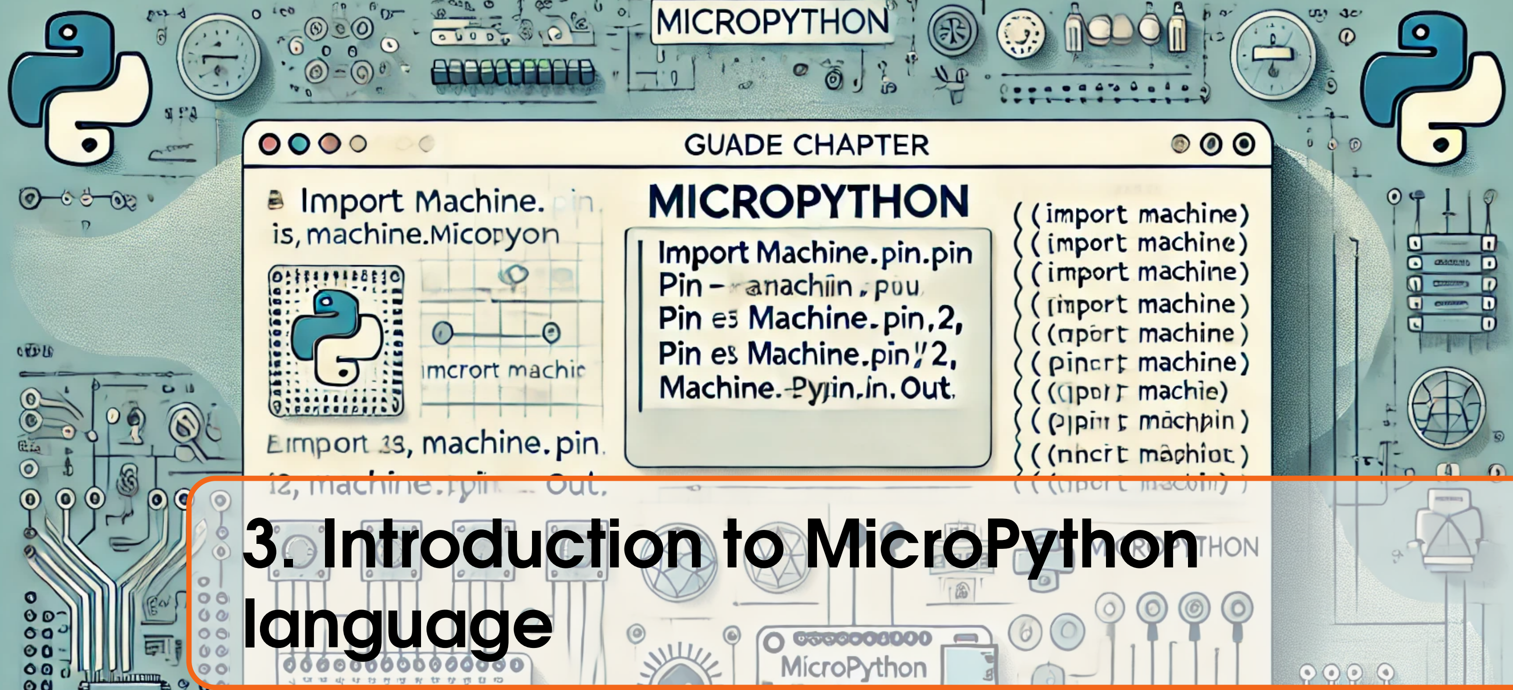


Figure 2.8: Waveshare Pico to hat expander. Source: https://www.waveshare.com/pico-to-hat.htm



3. Introduction to MicroPython language

This chapter will present basic, selected elements of the MicroPython language, necessary to create simple projects based on the Raspberry Pi Pico. Here they are:

- **Variable** - is a programming element that allows you to store a given value under a chosen name, e.g. when creating a mathematical program to count the areas of various figures, we would often use the number π equal to, let's assume: 3.14. Instead of entering it manually in each equation, you can create a variable: pi=3.14, then an example program to count, e.g. the area of a circle would look like this:

```
pi = 3.14
r = 10
area_circle = pi*r*r
```

  See that here we have created three numeric variables. One storing the value of the number pi (pi), the second storing the value of the radius of the circle (r) and the third storing the result, i.e. the area of the circle (area_circle). Remember that in Python and MicroPython, variables are created by first entering the variable name of your choice and after the sign = its value. There are many types of variables. In the above example, we presented *double* type variables (floating point numbers, e.g. 3.42, -5.68, etc.). In addition, the basic types of variables include:
  - *integer* (e.g. 0, -4, 12, etc.),
  - *boolean* (logical variables with two possible values: *True* or *False*),
  - *string* (strings of characters, e.g. "Hello", "Raspberry Pi Pico").

  Fig. 3.1 shows an example program in which several variables of different types were created. To display them on the screen, the *print()* function can be used. What we put as an argument of print function will be displayed in the terminal. Notice that in order to start the program, you must press the *Run* button (green button with an arrow). If it is grayed out, check if the Raspberry Pi Pico board is connected to the computer via USB and then select again: *Micropython (Raspberry Pi Pico)...* in the lower right corner as shown by the arrow in figure. If you save the program to the Raspberry Pi Pico with name "main.py" it will start automatically when the Raspberry Pi Pico board will be powered regardless of whether it is connected to the computer or powered from e.g. Powerbank.

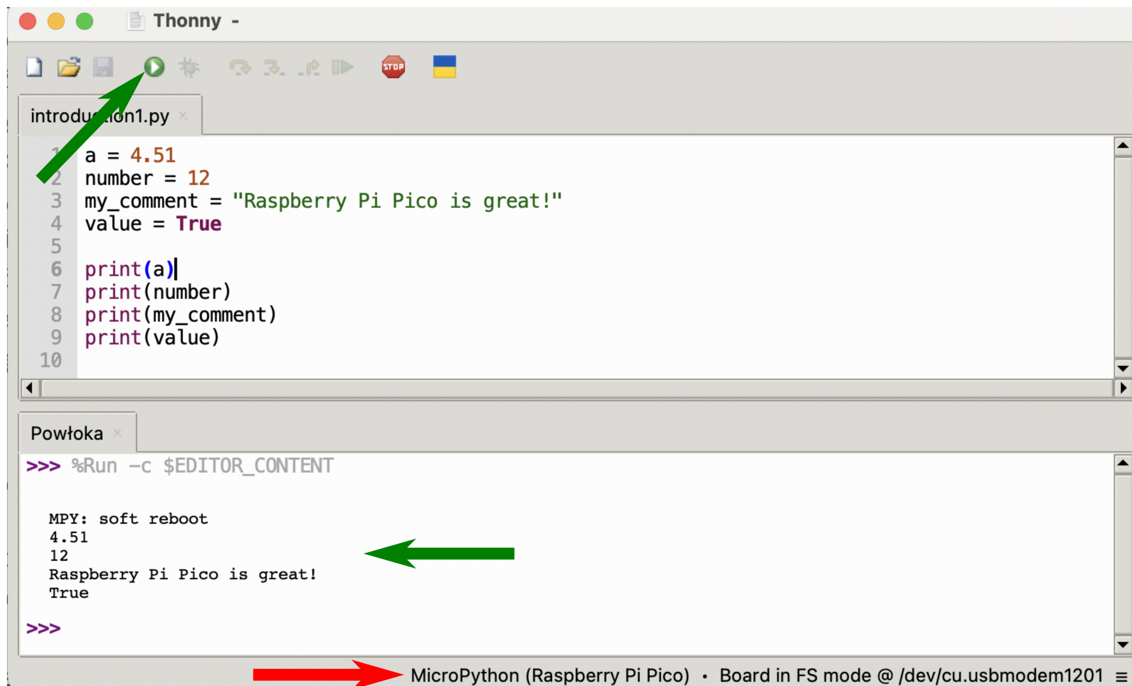


Figure 3.1: The example program with different type of variables.

- **Lists** - allow you to create a set of values. An example list looks like *values=[1, 35, 6, 14]* which means it consists of 4 elements separated by commas. The elements are numbered with indexes similarly to houses along the street, only with the difference that we number them from zero and not one. So to extract value 6 from the list, you need to call the command: *values[2]* because the 0th element is value 1, the 1st element is value 35, the 2nd element is value 6 and the 3rd element is value 14. The element indexes are given in square brackets next to the list name.
- **Functions** - are a subroutine, or a separate part of a program that performs a specific operation. A function can accept input data called function arguments and can return a result. For example, you can write a function that calculates the area of a circle. To declare a function, use the word *def* and then put the function name, e.g. *circle_area*. After the function name, we place brackets and inside we provide the function arguments. In the case of calculating the area of a circle, we need the user to provide the radius of the circle and this will be our function argument. After the brackets, we place a colon. Inside the function, we place the code that is to be executed when the function is called, i.e. calculating the area of a circle using the formula: $A = \pi r^2$. In order for the function to return a result, we use the word *return* and after it the value that the function is to return. So the code would look like this:

```
def circle_area(radius):
    return 3.14*radius**2
```

Notice that the word return is placed 4 spaces after the word def, which is on the line above. In MicroPython, indentation indicates which lines are inside a function/loop/structure and which are outside. It is very important that you make sure to indent, otherwise the program will execute the lines outside the function/loop/struc-

ture.

The next issue is the exponentiation operation. In MicroPython, exponentiation operations are performed by placing two stars, i.e. x^y will be written as $x**y$.

To call a given function in a program, you simply need to provide its name and put arguments in brackets as shown in Fig. 3.2. Note that two alternative possibilities of calling the *print* function are shown. In the case of calculating the area of the first circle, the comment and the value are displayed separately. So these two pieces of information are on separate lines after calling the program. You can also display several pieces of information in the print function at once, separating them with commas. Then the information displayed will be on one line.

introduction2.py

```
1  def circle_area(radius):
2      return 3.14*radius**2
3
4  print("Area of circle 1 is")
5  a = circle_area(3)
6  print(a)
7  b = circle_area(4)
8  print("Area of circle 2 is",b)
```

Powłoka

```
>>> %Run -c $EDITOR_CONTENT

  MPY: soft reboot
  Area of circle 1 is
  28.26
  Area of circle 2 is 50.24

>>>
```

MicroPython (Raspberry Pi Pico) • Board in FS mode @ /dev/cu.usbmodem1201

Figure 3.2: The example program with definition of function.

- **if...else structure** - allows for the execution of different code fragments depending on the condition being met. At the beginning, we write the word *if* and after it the condition that is to be met. At the end, we place a colon character. All subsequent lines that we place inside *if* (after indentation) will be executed only if the condition is met. Then we can, but do not have to, add an *else* block and after it a colon character. All subsequent lines inside the *else* block will be executed only if the condition in the *if* has not been met. An example program in Fig. 3.3.
  The example program uses two new elements. The first is the division operation with remainder (%), which returns the remainder of division by a given number, e.g. 12%2 will return 0 because there is no remainder from dividing 12 by 2, but 13%2 will return 1. Another new thing is the equalization operation. If we want to check if a number equals another, we use the double equal sign (==). The remaining equalization operations look like this:
  - a>b (checking if a is greater than b)
  - a>=b (checking if a is greater than or equal to b)

- a<b (checking if a is less than b)
- a<=b (checking if a is less than or equal to b).

```
introduction3.py
1 a = 12
2 if a%2 == 0:
3     print("This number is even")
4 else:
5     print("This number is odd")

Powłoka
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
This number is even

>>>
```

```
introduction3.py
1 a = 13
2 if a%2 == 0:
3     print("This number is even")
4 else:
5     print("This number is odd")

Powłoka
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
This number is odd

>>>
```

Figure 3.3: An example code showing how to use *if...else* structure.

In the above program, you can display a message about whether the number is even or odd in a nicer way. The print function allows you to combine several strings by adding them with the "+" sign. To display the variable *a*, which is an integer type, you must first convert it to a string using the *str()* function. An example program is shown in Fig. 3.4

```
introduction3.py
1 a = 12
2 if a%2 == 0:
3     print(str(a)+" is even")
4 else:
5     print(str(a)+" is odd")
6 
7

Powłoka
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
12 is even

>>>
MicroPython (Raspberry Pi Pico) • Board in FS mode @ /dev/cu.usbmodem1201
```

Figure 3.4: Example program showing concatenating strings to display them in one message using the print function.

- **For loop** - is a loop that allows you to repeat a given piece of code several times. An example program with a for loop is shown in Fig. 3.5.
  At the beginning, the word *for* is placed, then the iterating variable (in this case called *i*), then the word *in* and then the range. In this case, the range was numeric from 0 to 5, which means that the iterating variable *i* will increase its value by 1 each time the for loop is executed, starting from zero, up to a value one less than the upper range, i.e. up to and including 4. In this case, the for loop will execute 5 times and in each iteration the value of the iterator *i* will be added to the variable *s*.

```
introduction3.py *
1  s = 0
2  for i in range(0,5):
3      s=s+i
4      print("i="+str(i)+" s="+str(s))

Powłoka

MPY: soft reboot
i=0 s=0
i=1 s=1
i=2 s=3
i=3 s=6
i=4 s=10
>>>
MicroPython (Raspberry Pi Pico) • Board in FS mode @ /dev/cu.usbmodem1201
```

Figure 3.5: Example program showing for loop.

- **While loop** - executes a given piece of code as long as the condition is met. An example program showing the operation of the while loop is placed in Fig. 3.6.
  At the beginning, the word *while* is placed, then the condition that must be met for the loop to execute, and at the very end a colon character. In this case, the loop will execute until the variable s is less than 10. In the middle of the loop, the value of the variable s is displayed and then its value is increased by 2. This operation can be written in several ways. One of them is: $s = s+2$ or, in short, $s+ = 2$, which will also add the value 2 to the current value of the variable s.

```
introduction3.py
1  s = 0
2  while s<10:
3      print("s="+str(s))
4      s+=2 #This is equivalent to: s=s+2

Powłoka

MPY: soft reboot
s=0
s=2
s=4
s=6
s=8
>>>
MicroPython (Raspberry Pi Pico) • Board in FS mode @ /dev/cu.usbmodem1201
```

Figure 3.6: Example program showing while loop.

The example program uses the # sign. Everything that is placed on the same line after the # sign is a comment that is not executed by the program.

In the case of microcontroller programming, an infinite loop is usually used, which repeats a given fragment of code all the time until the power supply to the microcontroller board is turned off. Most often, an infinite loop is executed using a *while* loop

by specifying a logical value of *True* as the condition:

```
while True:
    print("It is infinite loop")
```

The introduced, selected elements of the MicroPython language are, according to the authors, the most useful for starting work with the Raspberry Pi Pico W board. If you need more information about the MicroPython language, it is best to familiarize yourself with the documentation for this language.

4. The digital signals

Digital signals are signals that have one of two possible values 0 or 1, which in case of Raspberry Pi Pico microcontroller corresponds to voltage values 0 V or 3.3 V. They can be used to control elements such as LEDs (turning on or off) or to obtain information from elements such as a button (pressed or not pressed) or a motion sensor (motion detected or no motion detected). Reading or generating digital signals is done using ***general-purpose input/output (GPIO)*** pins. GPIO are pins that can be used both as input pins, from which we can read information, e.g. about pressing a button, or as output pins, on which we can generate digital signals, e.g. to turn on an LED. Fig. 4.1 shows the 40 pins from the Raspberry Pi Pico board, which are numbered from the top left. Pins serving as GPIO are abbreviated *GPx* (green rectangles), where *x* is the ordinal number of the GPIO pin, starting from 0 to 28.

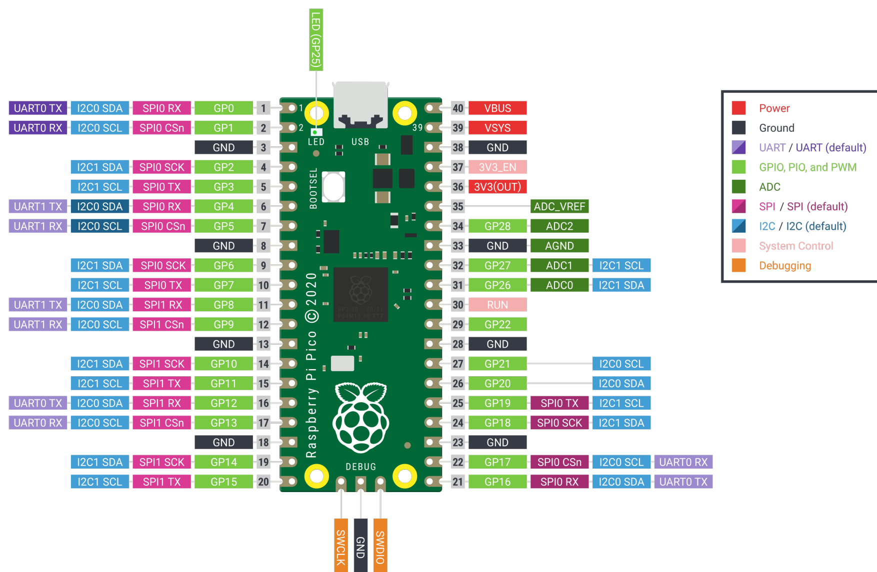


Figure 4.1: The pinout of Raspberry Pi Pico. Source of image: https://www.raspberrypi.com.

4.1 Example 1: blinking LED project

Let's start the Raspberry Pi Pico adventure with a blinking LED project. First, connect the LED to the selected GPIO pin as shown in Fig. 4.2.

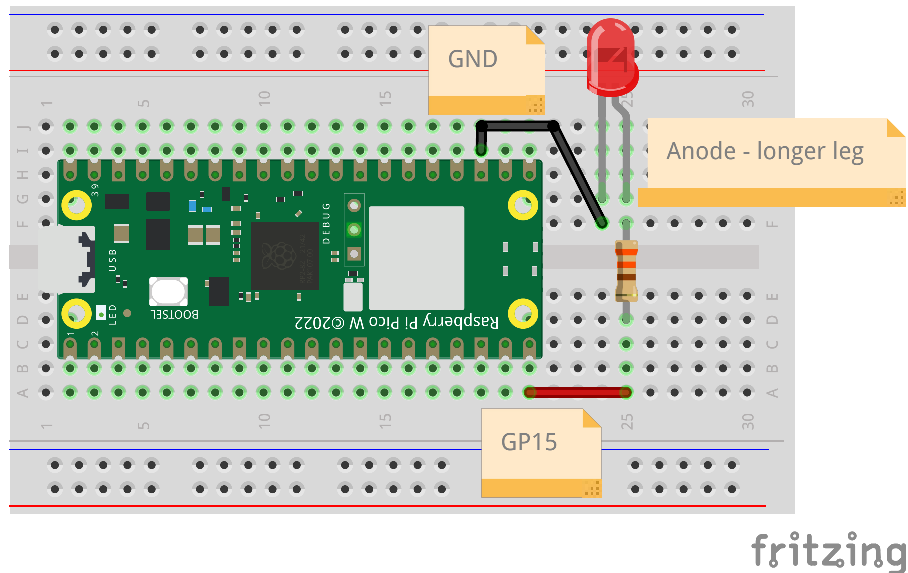


Figure 4.2: Connecting the electronic circuit from example 1.

Now let's use Thonny to write a program in MicroPython that will turn the diode on and off alternately every 1s. To do this, follow these steps:

1. Let's start by including the library in MicroPython:

```
import machine
```

The *machine* library contains all the necessary instructions for communicating with the Raspberry Pi Pico. The command *import* includes the specified library in the project.

2. Next, you need to configure the GP15 pin to work as an output pin because we control the LED by sending a value of 1 or 0 to the GP15 pin. The ***Pin*** function from the *machine* library is used for this purpose. To call functions from library in the MicroPython language, first we provide the name of the library and after the dot the name of the function from this library, i.e. *machine.Pin()*. The Pin function takes two arguments. The first one is the GPIO pin number. The second one is the specification of whether the GPIO pin will work as an input (***machine.Pin.IN***) or output (***machine.Pin.OUT***). Hence, in this case the function call would look like *machine.Pin(15, machine.Pin.OUT)*. The object returned by the *Pin* function was assigned to the created variable *led*.

```
#Including machine library
import machine

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)
```

3. In the next step we place an infinite loop, which will contain the main part of the program:

```
#Including machine library
import machine

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)

while True:
```

4. In the next step we light up the LED by sending the value 1 to the GP15 pin. The *value()* function is used to set the value on the pin, which takes the value 0 or 1 as an argument.

```
#Including machine library
import machine

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)

while True:
    led.value(1)
```

5. Now the program should wait 1s so that we can see the effect of lighting up the diode. For this purpose we will use the *utime* library, which contains functions for delays. First, we need to add the library:

```
#Including libraries
import machine
import utime

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)

while True:
    led.value(1)
```

6. The *sleep* function from the *utime* library allows you to delay the program for a selected number of seconds. Let's set a delay of 1s after the diode lights up:

```
#Including libraries
import machine
import utime

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)

while True:
    led.value(1)
    utime.sleep(1)
```

7. The last step is to turn off the LED by sending value 0 to pin GP15 and wait 1s to see the effect.

```python
#Including libraries
import machine
import utime

#Configuring GPIO pin 15 as output
led = machine.Pin(15, machine.Pin.OUT)

while True:
    led.value(1)
    utime.sleep(1)
    led.value(0)
    utime.sleep(1)
```

Run the above code in the Thonny editor and see the effect. If the diode blinks every 1s, you have done everything correctly. If not, first check the connection of the LED diode on the board and then check if you have not made a mistake in the program when rewriting the code.

Tip 4.1.1

In the above example, we set the value of the GP15 pin to 1 or 0 to turn the diode on or off. The value() function was used for this purpose. However, the program can be written even simplier. The toggle() function changes the value on a given pin to the opposite, i.e. if the value was set to 1, it changes to 0, and if the value was set to 0, it changes to 1. So in the above example, the interior of the infinite loop would change to:

```python
while True:
    led.toggle()
    utime.sleep(1)
```

Tip 4.1.2

If you save the program to the Raspberry Pi Pico (not on the computer) with name "main.py" it will start automatically when the Raspberry Pi Pico board will be powered regardless of whether it is connected to the computer or powered from e.g. Powerbank.

4.2 Example 2: LED turned on/off with push button

In this example, the LED will turn on or off when the button is pressed. To do this, first connect the circuit according to Fig. 4.3.

Push button has been connected to GP14 pin and LED to GP15. Now open Thonny and let's write some MicroPython code that will control LED depending on the button press. To do this, follow the steps below:

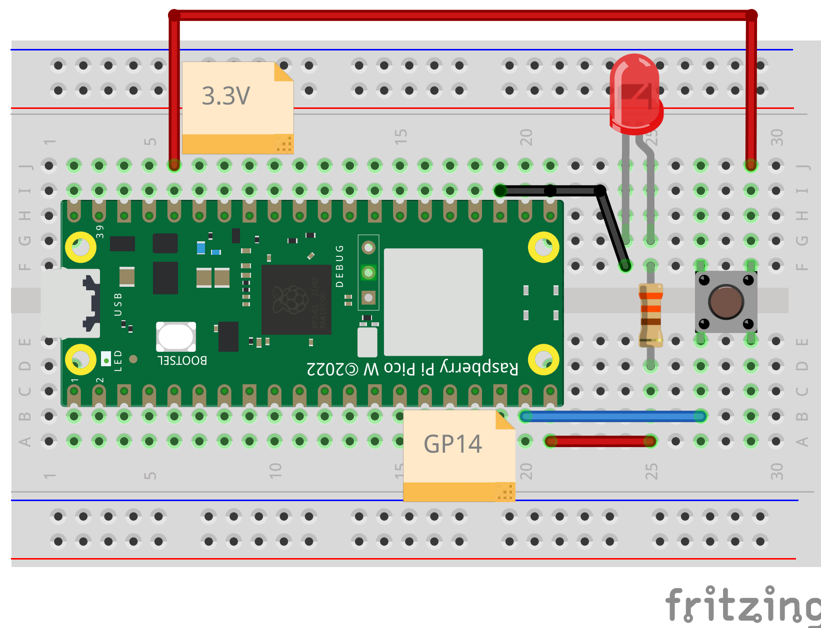


Figure 4.3: Connecting the electronic circuit from example 2.

1. First, you need to add the machine library and configure the GP15 pin, to which the LED is connected, as an output as it was done in the previous example.

```
import machine

led = machine.Pin(15, machine.Pin.OUT)
```

2. Then you need to configure the GP14 pin, to which the button is connected, as an input element (option **machine.Pin.IN**) because we will read the value from the button (whether it has been pressed). The button should also be connected to resistors called *pull-up* or *pull-down*, which are already mounted inside the Raspberry Pi Pico board and connected to all GPIO pins. These resistors are necessary to avoid floating input, i.e. a situation in which information would be read from the GPIO pin that the button is pressed when we have not pressed it at all. This can be considered as malfunction and serious design mistake. Pull-down resistors connect the button to ground, which means that if the button is not pressed, the value 0 will be read. Pull-up resistors connect the button to 3.3V, which means that if the button is not pressed, the value 1 will be read. In this case, we will use pull-down resistors. For this purpose, **machine.Pin.PULL_DOWN** should be provided as the third argument of the *Pin* function and opposite button lead should be connected to +3.3 V. If we want to use pull-up resistors, we would have to pass **machine.Pin.PULL_UP** as the third argument and opposite button lead should be connected to ground (GND), which has 0 V potential.

```
import machine

led = machine.Pin(15, machine.Pin.OUT)
but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_DOWN)
```

3. In the next step, let's first turn off the LED by setting the value 0:

```
import machine

led = machine.Pin(15, machine.Pin.OUT)
but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_DOWN)

led.value(0)
```

4. Now let's create the main loop of the program and in it let's check the button's state. If the value returned by the button is equal to 1, it means that the button was pressed:

```
import machine

led = machine.Pin(15, machine.Pin.OUT)
but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_DOWN)

led.value(0)

while True:
    if but.value()==1:
```

5. Next, let's change the LED state to the opposite when the button is pressed:

```
import machine

led = machine.Pin(15, machine.Pin.OUT)
but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_DOWN)

led.value(0)

while True:
    if but.value()==1:
        led.toggle()
```

6. The last step is to add a 1s delay. This delay can be shorter but is necessary because when we press the button it is on for a few milliseconds before we release the button. During this time the program will execute the main loop several times and the command to change the LED state to the opposite will be executed several times. To prevent this, add a delay. Remember to add the *utime* library:

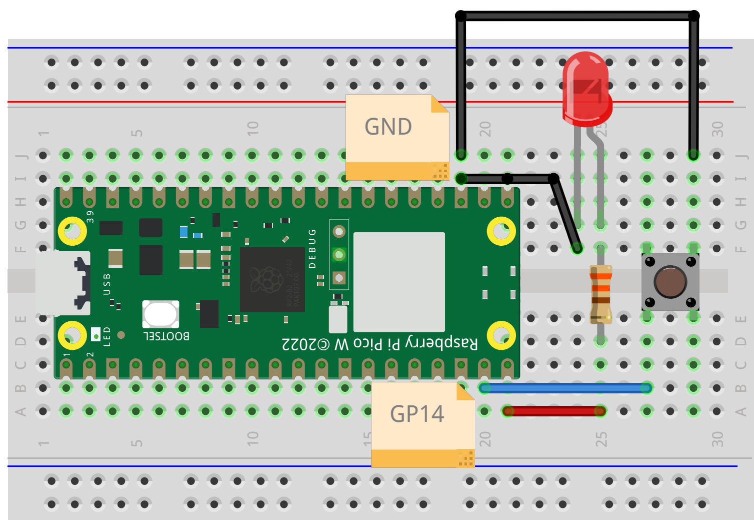
```
import machine
import utime

led = machine.Pin(15, machine.Pin.OUT)
but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_DOWN)

led.value(0)

while True:
    if but.value()==1:
        led.toggle()
```

```
12          utime.sleep(1)
```

Run the above code in the Thonny editor and press the button connected to the Raspberry Pi. Does the LED light up? If not, check the button connection.

Warning 4.2.1

If you are using Raspberry Pi Pico 2W, the RP2350 microcontroller has a problem with reducing the voltage to 0V when using pull-down resistors. In reality, the voltage drops to about 2.2V. For this reason, it is better to use pull-up resistors that connect the button to 3.3V, which means that if the button is not pressed, the value 1 will be read. To do this, connect the button leg to 0V instead of 3.3V as shown in figure:



However, two changes must be made in the program. The first is to change the constant *machine.Pin.PULL_DOWN* to *machine.Pin.PULL_UP* when configuring the button. The second change is that the button will be pressed when we read the value 0 and not 1 on the pin to which the button is connected:

```
1  import machine
2  import utime
3
4  led = machine.Pin(15, machine.Pin.OUT)
5  but=machine.Pin(14,machine.Pin.IN,machine.Pin.PULL_UP)
6
7  led.value(0)
8
9  while True:
10     if but.value()==0:
11         led.toggle()
12     utime.sleep(1)
```

4.3 Example 3: Light switched on by a motion sensor

In this example, we will create automatic light switch that will switch the light on when the motion is detected by PIR motion sensor. For this purpose, we will use a LED diode and a HC-SR501 PIR motion sensor shown in Fig. 4.4 According to the documentation for the HC-SR501 PIR sensor, this sensor should be connected to a power supply from 5V to 20V. On the Raspberry Pi Pico W board, a 5V power supply is supplied on the *VBUS* pin. When using sensors powered by a higher voltage than 3.3V, make sure that the returned signal has a voltage of no more than 3.3V, otherwise we can damage the Raspberry Pi Pico board. Where can I find such information? The easiest way is to find a datasheet for a given sensor on the Internet and read what voltage the high signal generated by the sensor has. In this case, the manufacturer states that the HC-SR501 PIR sensor returns signals with a value of 3.3V or 0V.

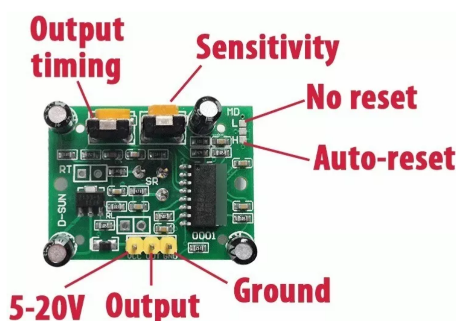


Figure 4.4: PIR Motion Sensor HC-SR501. Source: https://www.unoelectro.com.ar.

So connect the PIR motion sensor and LED diode to the Raspberry Pi Pico W board as shown in Fig. 4.5.

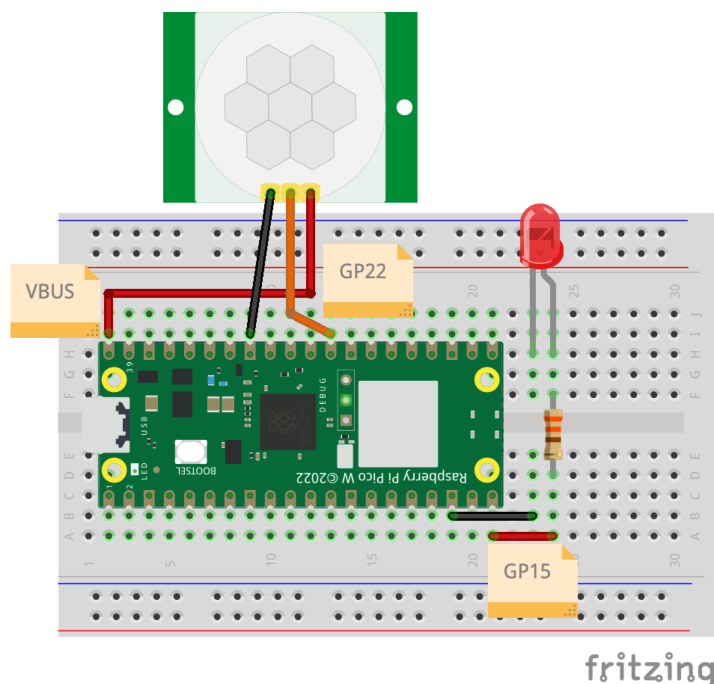


Figure 4.5: Connecting the electronic circuit from example 3.

Now let's write a program in the Thonny editor that will light up the LED for 5 seconds when motion is detected. To do this, follow these steps:

1. First you need to import the *machine* and *utime* libraries:

```
import machine
import utime
```

2. Then you need to configure the GP15 pin to which the LED is connected as output and turn off the LED:

```
import machine
import utime

LED = machine.Pin(15, machine.Pin.OUT)
LED.value(0)
```

3. The PIR motion sensor is an input element that returns 0 or 1 depending on whether it has detected movement. Therefore, the GP22 pin, to which the PIR motion sensor is connected, should be configured as an input:

```
import machine
import utime

LED = machine.Pin(15, machine.Pin.OUT)
LED.value(0)

PIR = machine.Pin(22, machine.Pin.IN)
```

4. Now in the main loop you need to read the value returned by the motion sensor and display it in the terminal:

```
import machine
import utime

LED = machine.Pin(15, machine.Pin.OUT)
LED.value(0)

PIR = machine.Pin(22, machine.Pin.IN)

while True:
    motion = PIR.value()
    print(motion)
```

5. Now run the program and check how the motion sensor behaves. Fig. 4.4 shows two potentiometers marked as "Output timing" and "Sensitivity". You can use them to adjust the motion sensor. We recommend setting the "Output timing" potentiometer to the lowest value possible and adjusting the sensitivity to your own taste. Don't worry if the sensor returns the value 1 several times after detecting movement. This is normal because the shortest signal duration is longer than the execution time of the main program loop. Once you have adjusted the motion sensor, continue writing the program.
6. In the next step, the LED should light up when movement is detected for 5 seconds, otherwise the LED should be off:

```
import machine
import utime

LED = machine.Pin(15, machine.Pin.OUT)
LED.value(0)

PIR = machine.Pin(22, machine.Pin.IN)

while True:
    motion = PIR.value()
    print(motion)
    if(motion==1):
        LED.value(1)
        utime.sleep(5)
    else:
        LED.value(0)
```

The program is ready. Test its operation.

4.4 Pulse Width Modulation (PWM)

Some components such as RGB LEDs or servos are controlled using Pulse Width Modulation (PWM). This technique allows for generating rectangular signals with a specified duty cycle from 0 to 100%. For example, if the duty cycle is set to 20%, then for 20% of the pulse duration we have a high signal and for 80% of the pulse duration we have a low signal. Example signal waveforms generated using the PWM technique are shown in Fig. 4.6.

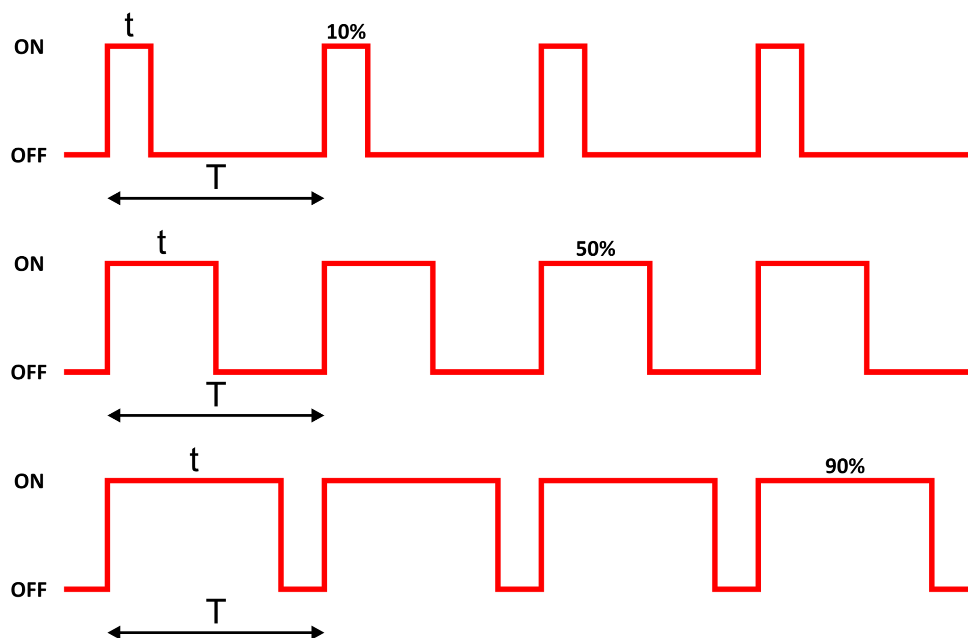


Figure 4.6: Example PWM signals with duty cycles of 10% (top plot), 50% (middle plot), and 90% (bottom plot). Source: https://www.robotyka.net.pl/pwm-modulacja-szerokosci-impulsu/.

4.4.0.1 Example 4: RGB LED

An RGB diode consists of three diodes in the colors red, green, and blue. To generate a selected color, you control the fill of each color with a PWM. For example, olive is a mixture of red and green. In the RGB code, to generate olive, you need to set 50% red fill and 50% green fill. PWM signals make this possible, hence they are used to control RGB diodes. The RGB LED has 4 pins as shown in Fig. 4.7.

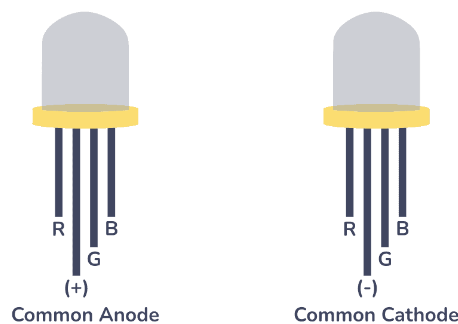


Figure 4.7: Types of RGB LED diodes. Source: https://www.build-electronic-circuits.com/rgb-led/ .

The longest lead is the common lead. If we are using a common cathode RGB diode, the longest lead should be connected to GND. And lighting a specific color is done by setting the high state on a red, green or blue lead. If the RGB diode has a common anode, then the longest lead should be connected to 3.3V and lighting a specific color is done by setting the low state on a given lead. In this example, we will use a common cathode RGB diode, which should be connected as shown in Fig. 4.8. The remaining 3 legs are the leads for the individual colors: red, green and blue. Don't forget to add current limiting resistors to each color control pin (3 x 330 Ω should be optimal).

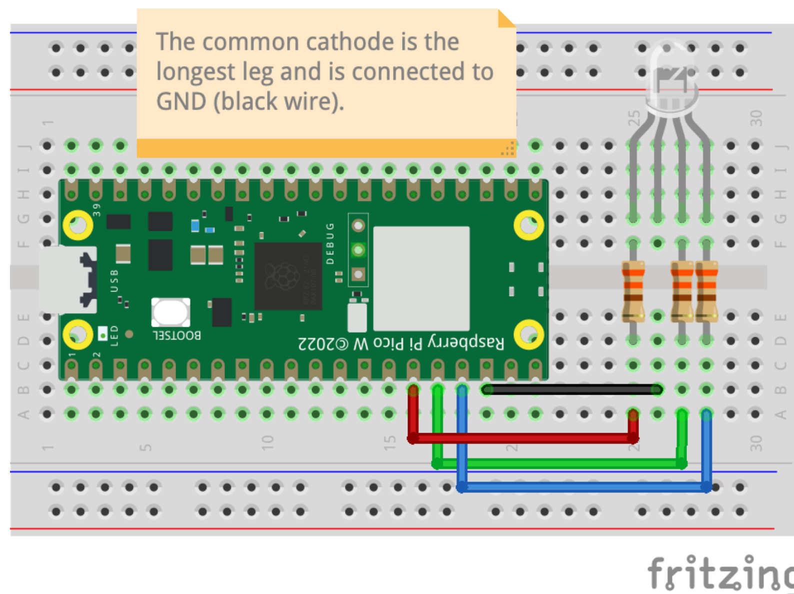


Figure 4.8: Example connection of a common cathode RGB LED. We used 3 x 330 Ω resistors for current limiting of each LED.

After connecting the RGB diode, open Thonny and start writing the code that will display the color sequences: red, green, blue, pink, white and yellow. To do this, you need to:

1. First, add the *machine* and *utime* libraries:

```
import machine
import utime
```

2. Next, you need to define which pins will be used to generate PWM signals. This is done using the *PWM* function from the *machine* library, which takes a *Pin* object as an argument. As mentioned, the RGB LED consists of three diodes (red, green, and blue), so you need to configure three pins as PWM.

```
import machine
import utime

red = machine.PWM(machine.Pin(11))
green = machine.PWM(machine.Pin(12))
blue = machine.PWM(machine.Pin(13))
```

3. Next, you need to set the PWM signal frequency. This is what the *freq* function is for:

```
import machine
import utime

red = machine.PWM(machine.Pin(11))
green = machine.PWM(machine.Pin(12))
blue = machine.PWM(machine.Pin(13))

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

4. In Raspberry Pi Pico W, the resolution of PWM signals is 16 bits. This means that the PWM signal fill is set from 0 to 65535, where 65535 is 100% fill. However, colors are given in RGB code, where values are from 0 to 255. Hence, it is convenient to create functions for setting the fill on a given diode, which will convert the value given in the range (0; 255) to the range (0; 65535) so that the user does not have to manually recalculate these values. To convert values given in the range (0; 255) to the range (0; 65535), the given values should be multiplied by 65535/255=257:

```
import machine
import utime

red = machine.PWM(machine.Pin(11))
green = machine.PWM(machine.Pin(12))
blue = machine.PWM(machine.Pin(13))

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

```

```
def set_color(r, g, b):
    red.duty_u16(r*257)
    green.duty_u16(g*257)
    blue.duty_u16(b*257)
```

5. Now we need to create the main while loop, in which the colors red, green and blue will be set and there will be a 1s delay between them:

```
import machine
import utime

red = machine.PWM(machine.Pin(11))
green = machine.PWM(machine.Pin(12))
blue = machine.PWM(machine.Pin(13))

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

def set_color(r, g, b):
    red.duty_u16(r*257)
    green.duty_u16(g*257)
    blue.duty_u16(b*257)

while True:
    set_color(255,0,0) #red
    utime.sleep(1)

    set_color(0,255,0) #green
    utime.sleep(1)

    set_color(0,0,255) #blue
    utime.sleep(1)
```

6. Run the program and see if these three colors are displayed on the RGB diode in order. If not, check the connection and code. If they are displayed, add the remaining colors to display, i.e.: pink, white and yellow.

```
import machine
import utime

red = machine.PWM(machine.Pin(11))
green = machine.PWM(machine.Pin(12))
blue = machine.PWM(machine.Pin(13))

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

def set_color(r, g, b):
    red.duty_u16(r*257)
    green.duty_u16(g*257)
```

```
15     blue.duty_u16(b*257)
16 
17 while True:
18     set_color(255,0,0) #red
19     utime.sleep(1)
20     set_color(0,255,0) #green
21     utime.sleep(1)
22     set_color(0,0,255) #blue
23     utime.sleep(1)
24     set_color(255,20,147) #pink
25     utime.sleep(1)
26     set_color(255,255,255) #white
27     utime.sleep(1)
28     set_color(255, 255, 0) #yellow
29     utime.sleep(1)
```

Test the finished program.

Tip 4.4.1

What should be changed in the above example if we have a common anode RGB LED? Two things should be changed:

1. The connection of the RGB LED. The black wire from Fig. 4.8 should be connected to 3.3V and not GND.
2. The individual colors in the common anode RGB LED are lit with a low state and not a high state. So to set 100% fill, you should set the value 0 and not 65535 on a given pin. Hence, you should modify the *set_color*() function as follows:

```
1 def set_color(r, g, b):
2     red.duty_u16(65535-r*257)
3     green.duty_u16(65535-g*257)
4     blue.duty_u16(65535-b*257)
```

Warning 4.4.1

When using more pins configured to generate PWM signals, be careful. In Raspberry Pi Pico W we have 8 PWM channels, each with two outputs A and B (see Fig. 4.9). Each output with the same number, e.g. A[5] and B[5] are not completely independent. We can use these two outputs in one program and set different PWM duty cycles, but the frequency of these signals will be the same.

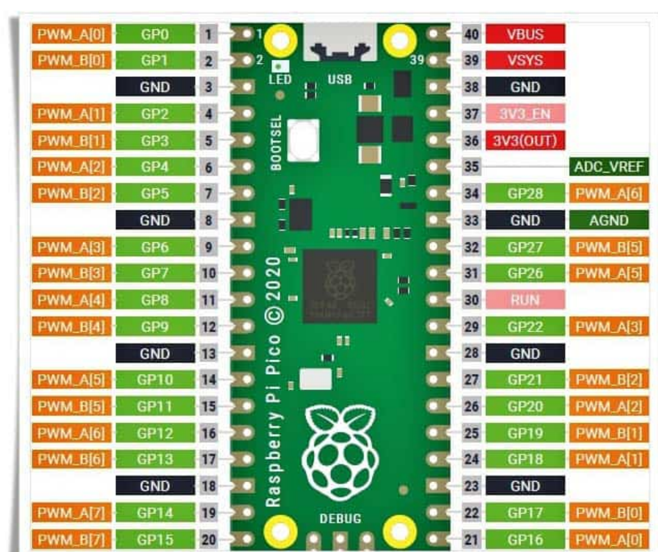
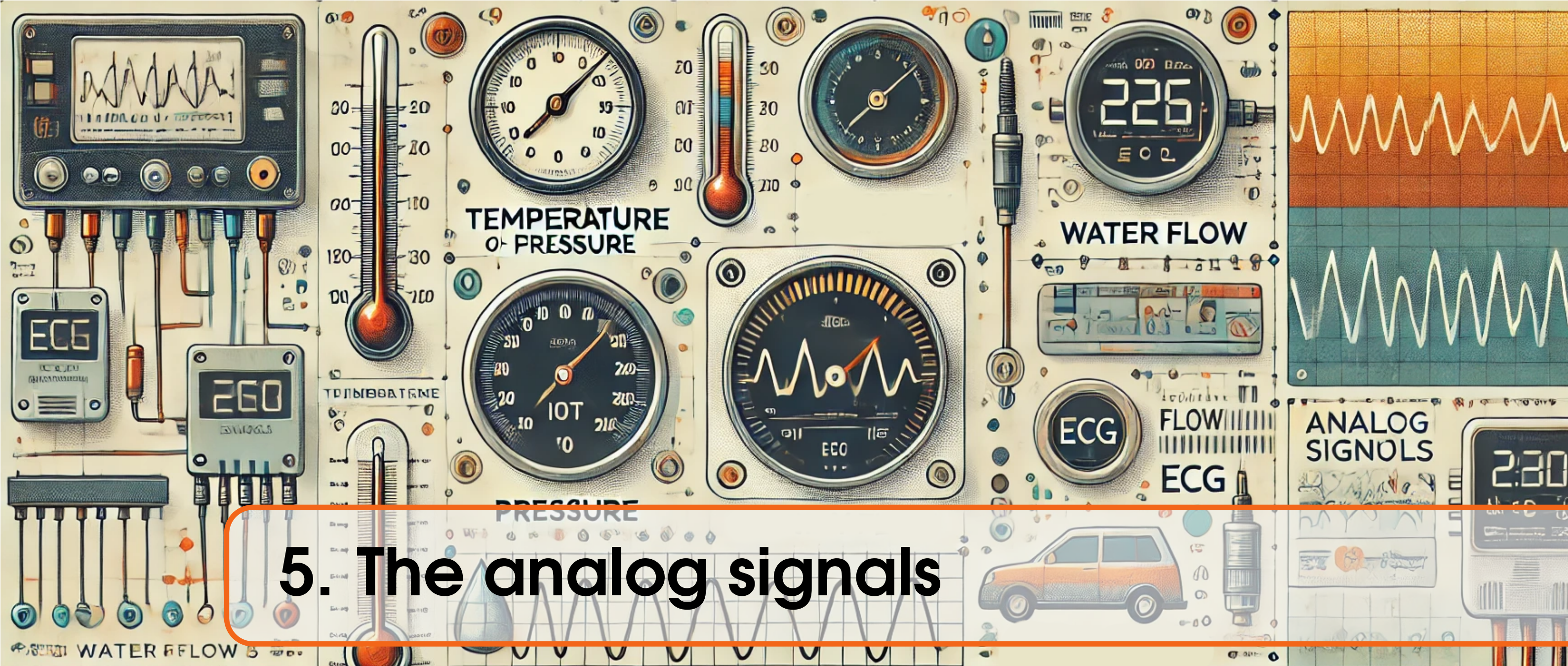


Figure 4.9: The PWM channels. Source: https://www.codrey.com/raspberry-pi/raspberry-pi-pico-pwm-primer/.



5. The analog signals

In addition to digital signals that have two possible states 0 or 1 (0 V or 3.3 V accordingly in Low Voltage TTL standard), analog signals carry information in terms of voltage or current. These provide continuous values in a given range, e.g. a sensor measuring temperature will return voltage proportional to temperature (e.g. 235 mV will correspond to the temperature of $23.5^{o}C$. We can say that sensor gain equals to 10 $mV/^{o}C$). To read the voltage of analog signals, an analog-to-digital converter (ADC) is necessary. Such a device is built into the Raspberry Pi Pico microcontroller already. The built-in converter is 12-bit. If we want more accurate measurements, we can connect an external converter to the Raspberry Pi Pico (more on this later in the guide).

5.1 Example 5: temperature measurement

Temperature measurement can be done in two ways: by using the built-in temperature sensor (biased bipolar diode) or an external temperature sensor, e.g. LM35. In this case, both methods will be implemented and the results will be compared. Let's start with the built-in temperature sensor. Open the Thonny editor and perform the following steps:

1. First you need to add the *machine* and *utime* libraries:

```
import machine
import utime
```

2. Then you need to configure the pin to work as an ADC, for this purpose the *ADC(pin number)* function is used. The built-in temperature sensor is connected to the fourth ADC channel:

```
import machine
import utime

built_in_temp = machine.ADC(4)
```

3. Then in the main while loop, read the value from the ADC using the *read_u16()* function. Although the ADC is 12-bit, the *read_u16* function immediately converts the value from 12bits to 16bits:

```
import machine
import utime

built_in_temp = machine.ADC(4)

while True:
    built_in_temp.read_u16()
```

4. To obtain the voltage value, convert the values returned by the *read_u16* function in the range (0;65535) to the voltage (0;3.3)V:

```
import machine
import utime

built_in_temp = machine.ADC(4)

while True:
    voltage1 = built_in_temp.read_u16()*3.3/65535
```

5. Then you need to convert the voltage to temperature according to the formula in the RP2040 documentation. The result will be displayed in the terminal every 1s:

```
import machine
import utime

built_in_temp = machine.ADC(4)

while True:
    voltage1 = built_in_temp.read_u16()*3.3/65535
    temp1 = 27 - (voltage1-0.706)/0.001721
    print(temp1)
    utime.sleep(1)
```

Test the program.

Measurement using the built-in temperature sensor is burdened with two fundamental problems. The first is the high inaccuracy of temperature measurement resulting from the sensitivity of the read values depending on the reference voltage. A change in the reference voltage by 1% already causes a temperature reading error of about 4^oC. Another problem is the fact that the built-in temperature sensor actually measures the temperature of the RP2040 and not the environment. Therefore, during intensive operation of the RP2040, the read temperature may be much higher than the ambient temperature. If you want a more accurate temperature measurement, it is better to use an external temperature sensor.

Now let's extend the program by adding temperature reading from the analog LM35 sensor. To do this, connect the LM35 sensor to the Raspberry Pi Pico W as shown in Fig. 5.1. The pins on which the ADC channels are led out are: GP26, GP27 and GP28 (see Fig. 1.1 - dark green markings).

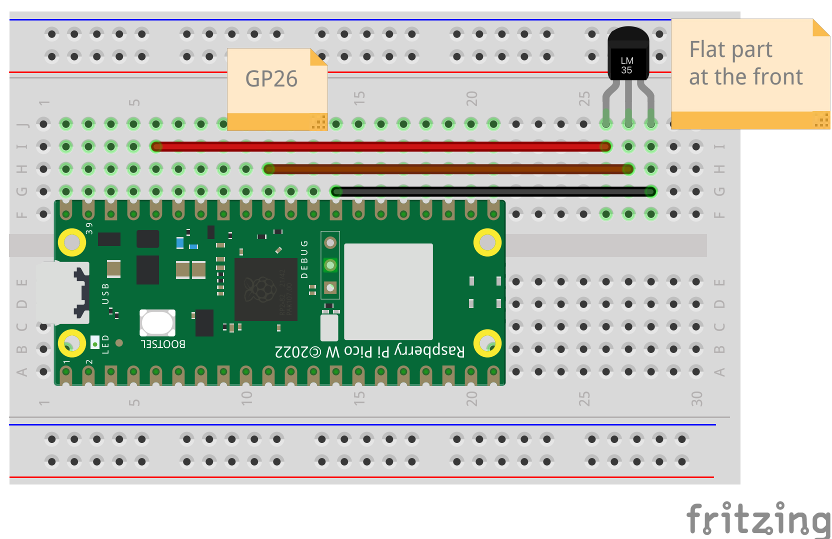


Figure 5.1: Example connection of a LM35 sensor.

6. Now let's configure the GP26 pin to work as an ADC:

```
import machine
import utime

built_in_temp = machine.ADC(4)
external_temp = machine.ADC(26)

while True:
    voltage1 = built_in_temp.read_u16()*3.3/65535
    temp1 = 27 - (voltage1-0.706)/0.001721
    print(temp1)
    utime.sleep(1)
```

7. In the next step, read the voltage returned by the LM35 sensor and multiply the result by 100 to obtain the temperature in degrees Celsius according to the LM35 sensor documentation (sensor gain is 10 $mV/^{o}C$, so we need voltage in mV thus we multiply by 1000 and divide by sensor gain: 10, to get temperature in ^{o}C. Hence multiply value after ADC conversion by 100):

```
import machine
import utime

built_in_temp = machine.ADC(4)
external_temp = machine.ADC(26)

while True:
        voltage1 = built_in_temp.read_u16()*3.3/65535
        temp1 = 27 - (voltage1-0.706)/0.001721

        voltage2 = external_temp.read_u16()*3.3/65535
        temp2 = voltage2*100
```

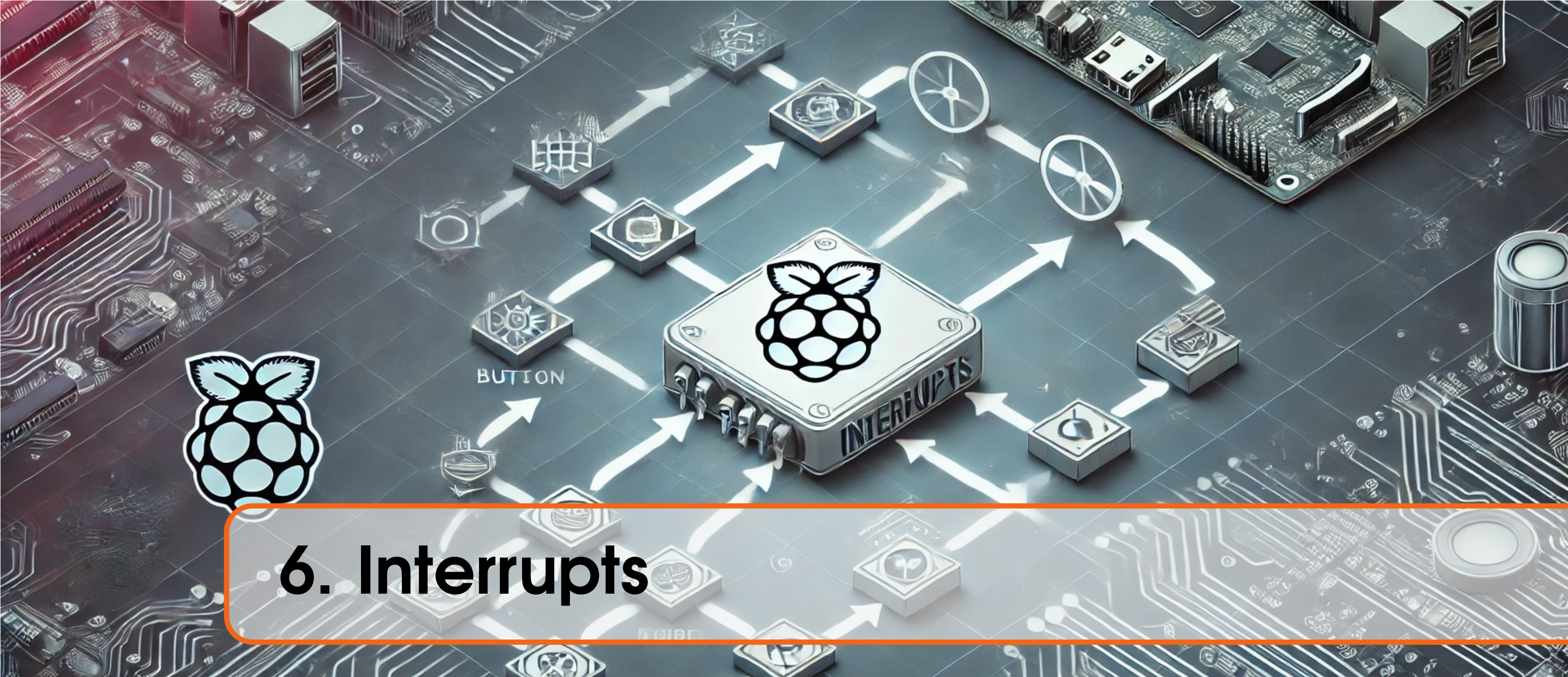
```
13
14          utime.sleep(1)
```

8. The final step is to display both results in the terminal in single line:

```
1  import machine
2  import utime
3
4  built_in_temp = machine.ADC(4)
5  external_temp = machine.ADC(26)
6
7  while True:
8          voltage1 = built_in_temp.read_u16()*3.3/65535
9          temp1 = 27 - (voltage1-0.706)/0.001721
10
11         voltage2 = external_temp.read_u16()*3.3/65535
12         temp2 = voltage2*100
13         print("Temp1="+str(temp1)+" Temp2="+str(temp2))
14         utime.sleep(1)
```

Test the program and compare the results.

The use of external ADCs to obtain more accurate voltage measurements will be discussed later in this tutorial.



6. Interrupts

Interrupts give us a mechanism that allows to stop the execution of a program for the time of a special procedure and then return to the moment where the program was stopped and resume it (see Fig. 6.1). Interrupts can be triggered by external or internal signals.

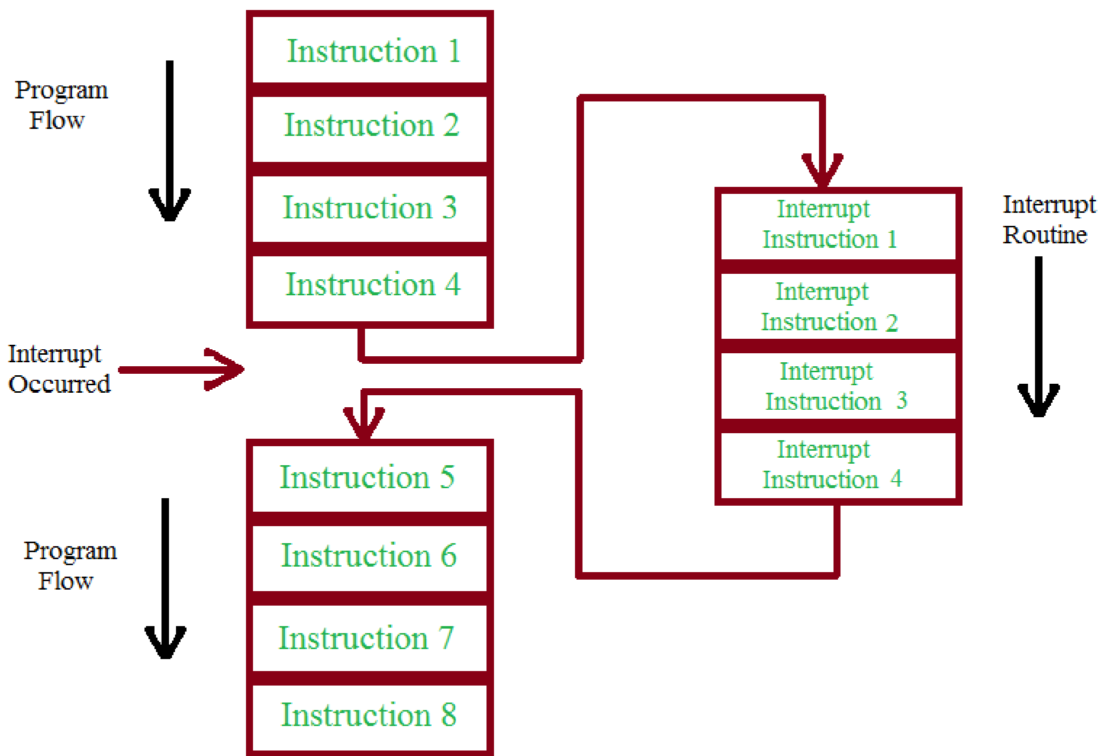


Figure 6.1: How interrupts work. Source: https://makergram.com/blog/what-is-interrupts/

To better understand the idea, interrupts are performed automatically at many moments in life, e.g. when we are reading a book and the phone rings, we stop reading the book

to answer the phone and after the conversation ends, we go back to reading the book. Exactly the same idea occurs in microcontrollers. External interrupts can be triggered by the following signals on the selected GPIO pin:

- Pin.IRQ_RISING - when the signal changes from low to high.
- Pin.IRQ_FALLING - when the signal changes from high to low.

6.1 Example 6: Sound signals at traffic lights

To better understand the idea of interruption, we will make a pedestrian traffic light with an audible signal for blind or visually impaired people when a button is pressed. For this purpose, we will use 3 LEDs to signal the traffic light, a buzzer with a tone generator and a button that will turn on the audible signal for at least one full sequence of lights.

First, connect the system according to Fig. 6.2.

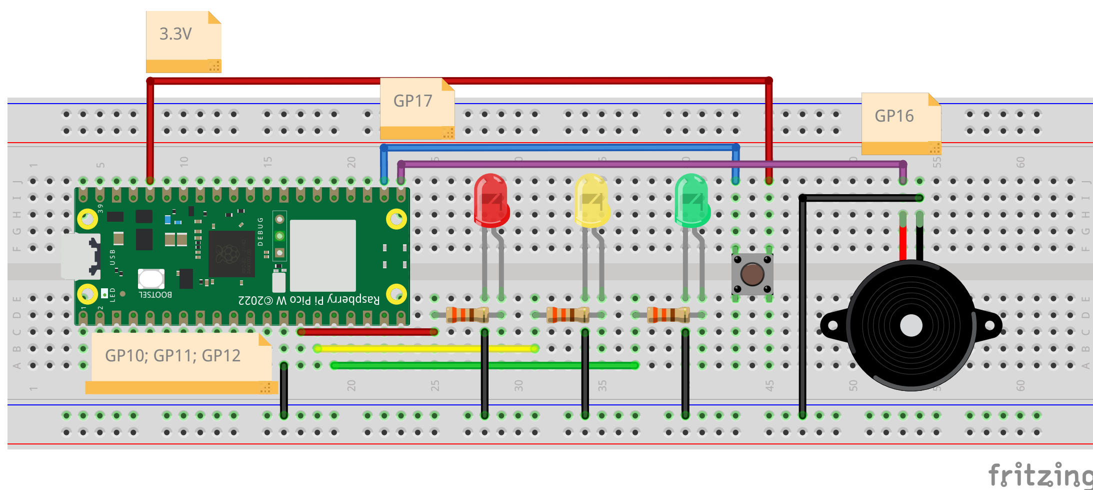


Figure 6.2: Connecting the electronic circuit from example 6.

Then open the Thonny editor and follow these steps:

1. First, add the necessary libraries: *machine* and *utime* and configure the pins to which the LEDs are connected as outputs:

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
led_green = machine.Pin(12, machine.Pin.OUT)
```

2. Then configure the GP16 pin to which the buzzer is connected as a pin to generate a PWM signal and set the PWM signal frequency to 1000 Hz.

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
```

```
led_green = machine.Pin(12, machine.Pin.OUT)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)
```

3. In the main loop of the program, the LEDs should be lit according to the sequence:
   - red LED on, other LEDs off
   - red and yellow LEDs on, green off
   - green LED on and other LEDs off.

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
led_green = machine.Pin(12, machine.Pin.OUT)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

while True:
    led_red.value(1)
    led_yellow.value(0)
    led_green.value(0)

    led_red.value(1)
    led_yellow.value(1)
    led_green.value(0)

    led_red.value(0)
    led_yellow.value(0)
    led_green.value(1)
```

4. Next, let's create a logical variable *sound_active*, which will have one of two possible values: *True* or *False*. When it has the value *True*, we will start the buzzer and generate an audible signal informing whether the lights are red or green. In addition, let's create a function *make_sound(duration)*, in which we will generate a sound signal. For this purpose, we will use the function *duty_u16()*, which will set the duty of the PWM signal. This function accepts values from 0 to 65535. The duty change will affect the volume of the sound generated by the buzzer. Sound signals in different countries may differ, but as a rule it is assumed that during a red or yellow light, the sound signal is generated at longer intervals than during a green light. To control the intervals between the generated sound signals, we will create a variable *duration* given as an argument to the function:

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
led_green = machine.Pin(12, machine.Pin.OUT)
```

```
7
8  buzzer = machine.PWM(machine.Pin(16))
9  buzzer.freq(1000)
10
11 sound_active=False
12
13 def make_sound(duration):
14     if sound_active:
15         buzzer.duty_u16(16383) %turn on buzzer
16         utime.sleep(1)
17         buzzer.duty_u16(0) %turn off buzzer
18         utime.sleep(duration)
19     else:
20         buzzer.duty_u16(0)
21
22         #Because when the sound signal is turned on,
23         #the total delay is 1 + duration. So here we
24         #add 1 to maintain the same delay.
25
26         utime.sleep(duration+1)
27
28 while True:
29     led_red.value(1)
30         ...
```

5. Now let's add sound generation after each traffic lights. The device should generate a warning sound signal 5 times. For this purpose, a for loop is created, which will execute 5 times and call the *make_sound* function 5 times. In the case of red or yellow, the interval between sound signals should be 2 s and in the case of green 1 s. Hence these parameters were given to the *make_sound* function:

```
1  import machine
2  import utime
3
4  led_red = machine.Pin(10, machine.Pin.OUT)
5  led_yellow = machine.Pin(11, machine.Pin.OUT)
6  led_green = machine.Pin(12, machine.Pin.OUT)
7
8  buzzer = machine.PWM(machine.Pin(16))
9  buzzer.freq(1000)
10
11 sound_active=False
12
13 def make_sound(duration):
14     if sound_active:
15         buzzer.duty_u16(16383) %turn on buzzer
16         utime.sleep(1)
17         buzzer.duty_u16(0) %turn off buzzer
18         utime.sleep(duration)
19     else:
20         buzzer.duty_u16(0)
```

```
            utime.sleep(duration+1)

while True:
    led_red.value(1)
    led_yellow.value(0)
    led_green.value(0)
    for i in range(0,5):
        make_sound(2)

    led_red.value(1)
    led_yellow.value(1)
    led_green.value(0)
    for i in range(0,5):
        make_sound(2)

    led_red.value(0)
    led_yellow.value(0)
    led_green.value(1)
    for i in range(0,5):
        make_sound(1)
```

6. Now let's configure the GP17 pin, to which the button is connected, as an input. In addition, let's configure the interrupt. To do this, use the function: ***irq(trigger, handler)***. The first argument specifies what type of signal the interrupt is sensitive to, e.g. a falling or rising edge. In the case of a rising edge, the interrupt will be triggered when the button is pressed, and in the case of a falling edge, when the button is released. In this case, we will use a rising edge. The second argument is the name of the function we created, which will be executed when an interrupt occurs. We will create a function called *sound_for_blind* in the next step:

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
led_green = machine.Pin(12, machine.Pin.OUT)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

sound_active=False

def make_sound(duration):
    if sound_active:
        buzzer.duty_u16(16383) %turn on buzzer
        utime.sleep(1)
        buzzer.duty_u16(0) %turn off buzzer
        utime.sleep(duration)
    else:
        buzzer.duty_u16(0)
        utime.sleep(duration+1)
```

```
22
23 button = machine.Pin(17, machine.Pin.IN, machine.Pin.PULL_DOWN)
24 button.irq(trigger=machine.Pin.IRQ_RISING, handler=sound_for_blind)
25
26 while True:
27     led_red.value(1)
28     ...
```

7. Now let's create a function that will be called when an interrupt occurs. Inside this function we will change the value of the variable *sound_active* to True. Since this is a variable defined outside the function we should call the command global *variable_name* which informs that this variable is created outside the function:

```
1  import machine
2  import utime
3
4  led_red = machine.Pin(10, machine.Pin.OUT)
5  led_yellow = machine.Pin(11, machine.Pin.OUT)
6  led_green = machine.Pin(12, machine.Pin.OUT)
7
8  buzzer = machine.PWM(machine.Pin(16))
9  buzzer.freq(1000)
10
11 sound_active=False
12
13 def make_sound(duration):
14     if sound_active:
15         buzzer.duty_u16(16383) %turn on buzzer
16         utime.sleep(1)
17         buzzer.duty_u16(0) %turn off buzzer
18         utime.sleep(duration)
19     else:
20         buzzer.duty_u16(0)
21         utime.sleep(duration+1)
22
23 def sound_for_blind(pin):
24     global sound_active
25     sound_active=True
26     print("Sound active: "+str(sound_active))
27
28 button = machine.Pin(17, machine.Pin.IN, machine.Pin.
       ↪ PULL_DOWN)
29 button.irq(trigger=machine.Pin.IRQ_RISING, handler=
       ↪ sound_for_blind)
30
31 while True:
32     led_red.value(1)
33     ...
```

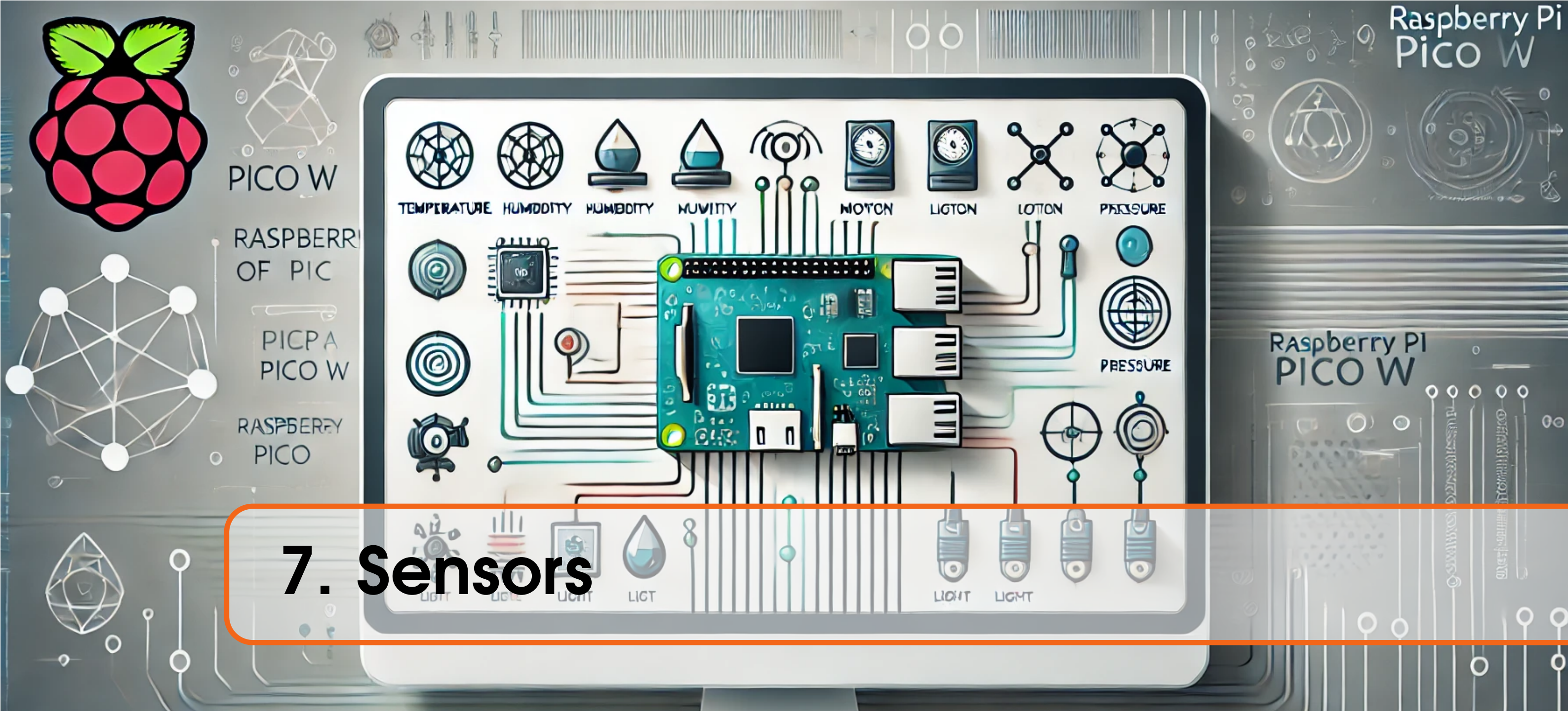
Now when the user presses the button, the *sound_for_blind* function will be called, regardless of which line of code in the while loop the program will execute.

8. The last step is to turn off the sound signal after the at least one full light switching sequence. To do this, let's create a *counts* variable that will count how many times the full sequence has been executed after the interruption. Then, in the main program, we will check if *counts* is greater than or equal to 1, if so, we turn off the sound signal by setting the *sound_active* variable to *False*. If the sound signal is on, then with each iteration of the while loop we will increase the value of the *counts* variable by 1:

```
import machine
import utime

led_red = machine.Pin(10, machine.Pin.OUT)
led_yellow = machine.Pin(11, machine.Pin.OUT)
led_green = machine.Pin(12, machine.Pin.OUT)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

sound_active=False

def make_sound(duration):
    if sound_active:
        buzzer.duty_u16(16383)
        utime.sleep(1)
        buzzer.duty_u16(0)
        utime.sleep(duration)
    else:
        buzzer.duty_u16(0)
        utime.sleep(duration+1)

def sound_for_blind(pin):
    global sound_active
    sound_active=True
    print("Sound active: "+str(sound_active))

button = machine.Pin(17, machine.Pin.IN, machine.Pin.
  ↪ PULL_DOWN)
button.irq(trigger=machine.Pin.IRQ_RISING, handler=
  ↪ sound_for_blind)

counts=0

while True:
#How many times, the full sequence has been executed
    if counts>=1:
        sound_active = False
        counts=0
    if sound_active:
        counts=+1

```

```
41      led_red.value(1)
42      led_yellow.value(0)
43      led_green.value(0)
44      for i in range(0,5):
45          make_sound(2)
46
47      led_red.value(1)
48      led_yellow.value(1)
49      led_green.value(0)
50      for i in range(0,5):
51          make_sound(2)
52
53      led_red.value(0)
54      led_yellow.value(0)
55      led_green.value(1)
56      for i in range(0,5):
57          make_sound(1)
```

The program is now ready, you can test it.



7. Sensors

Sensors are used to measure various physical quantities , such as temperature, pressure, UV radiation intensity, acceleration, sound detection, gas concentration, etc. Sensors can return analog values (e.g. temperature sensor discussed in chapter 5) or digital values. Digital values can be either 0-1 (e.g. motion sensor, sound detection sensor) or they can be values from a given range, and then serial communication interfaces (buses) such as I^2C, SPI, $UART$ or $1-wire$ are used to transfer data. The digital transmission has an advantage over analog transmission in that is much less sensitive to external interference and noise.

The serial communication interface consists of a group of lines, which are used to send data between connected devices. Serial interfaces can be divided in two groups: synchronous and asynchronous. The first group use additional serial clock line, which synchronize all devices connected to communication interface. The most popular synchronous buses are SPI (Serial Peripheral Interface) and I^2C (Inter-Integrated Circuit), which is also marked as $I2C$, IIC or TWI (Two Wire Interface). The most popular asynchronous serial communication interfaces are $UART$ (Universal asynchronous receiver-transmitter) and $1-wire$.

In this chapter, we will discuss one sensor per type of digital communication. Other sensors are used in most ways similarly to those discussed below.

7.1 Example 7: parking sensor

Parking sensors emit sound when the distance between car and obstacle gets small, e.g. less than 1 meter when reverse parking. The frequency of the emitted sound gets higher the closer the distance to the obstacle is. To create a parking sensor, we will use the HC-SR04 ultrasonic distance sensor and a buzzer. First, connect the system according to Fig. 7.1.

Now go to Thonny editor and start creating a program that will first read the distance from an obstacle using an ultrasonic distance sensor and then generate a warning sound if the obstacle is closer than 1m. To do this, follow these steps:

1. First, let’s add the necessary libraries, i.e. *machine* and *utime*, and configure the pins: GP16 (buzzer) as PWM, GP18 (trigger) as output, and GP19 (echo) as input. In addition, let’s set the PWM signal generation frequency to 1000 Hz.

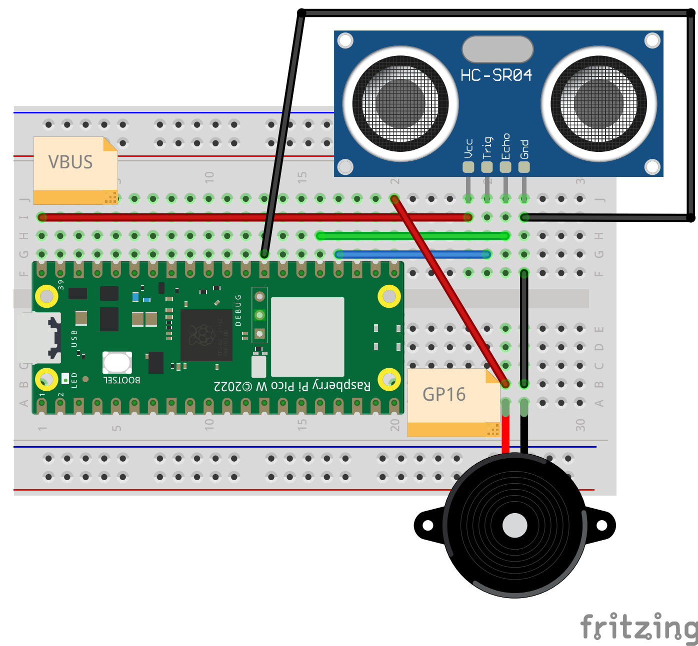


Figure 7.1: Connecting the electronic circuit from example 7.

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)
```

2. Next, in the main loop, let's add a code fragment to measure the distance from the ultrasonic distance sensor. According to the documentation for the HC-SR04 sensor (see Fig. 7.2), first set the low signal on the trigger for a short time, e.g. $2\mu s$. Then set the high signal for $10\mu s$. To generate delays in microseconds, use the function: *utime.sleep_us()*. In the next step, set the low signal on the trigger.

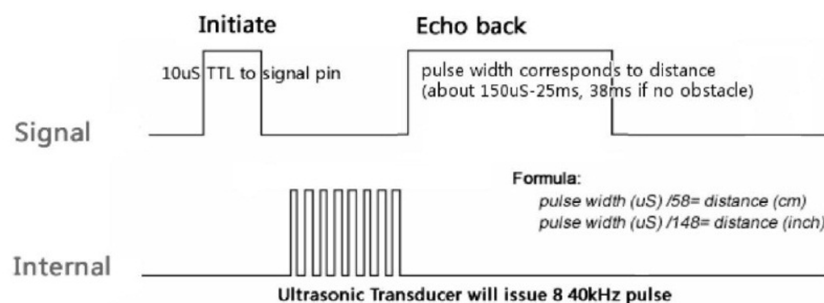


Figure 7.2: Principle of measuring distance on the HC-SR04 ultrasonic distance sensor. Source: https://www.electronicoscaldas.com/datasheet/HC-SR04.pdf.

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

while True:
    trigger.value(0)
    utime.sleep_us(2)
    trigger.value(1)
    utime.sleep_us(10)
    trigger.value(0)
```

3. Now we need to measure how long the high signal on the echo pin lasted, because the duration of the signal on the echo pin is related to distance. To do this, we will use the *utime.ticks_us()* function, which measures how much time has passed in μs since the program was started. First, we will create a while loop that will execute as long as the signal is low. Inside, we will place the *tick_us()* function. This way, we will get information about when the signal was last low.

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

while True:
    trigger.value(0)
    utime.sleep_us(2)
    trigger.value(1)
    utime.sleep_us(10)
    trigger.value(0)

    while echo.value()==0:
        signal_off = utime.ticks_us()
```

4. Similarly, we will measure when the signal was last high on the echo pin:

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
```

```
buzzer.freq(1000)

while True:
    trigger.value(0)
    utime.sleep_us(2)
    trigger.value(1)
    utime.sleep_us(10)
    trigger.value(0)

    while echo.value()==0:
        signal_off = utime.ticks_us()

    while echo.value()==1:
        signal_on = utime.ticks_us()
```

5. The difference between the time of the last occurrence of the high and low signal is the duration of the high pulse on the echo pin. How do we translate the pulse duration into distance? Look at Fig. 7.3 showing the idea of measuring distance with an ultrasonic distance sensor.

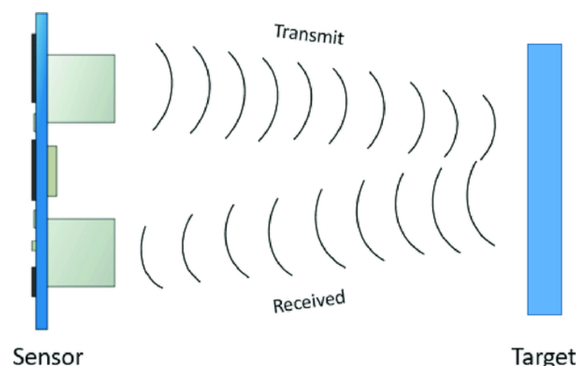


Figure 7.3: The principle of measuring distance using an ultrasonic sensor.. Source: https://www.researchgate.net/figure/A-block-diagram-of-Ultrasonic-sensor-working-principles_fig5_344385811.

At the beginning, a sound wave is emitted, which reflects from the object and returns to the sensor. Therefore, in the measured time t, the wave travels twice the distance between the sensor and the object and moves at a speed of about 340 m/s (the speed of sound in air). Therefore, we can write the equation for the speed:

$$v = \frac{2d}{t} \tag{7.1}$$

$$d = v \cdot \frac{t}{2} = 0.034 \frac{cm}{\mu s} \cdot \frac{t}{2} \approx \frac{t}{58} \tag{7.2}$$

Hence, the obtained pulse duration should be divided by 58 to get the distance in centimeters:

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

while True:
    trigger.value(0)
    utime.sleep_us(2)
    trigger.value(1)
    utime.sleep_us(10)
    trigger.value(0)

    while echo.value()==0:
        signal_off = utime.ticks_us()

    while echo.value()==1:
        signal_on = utime.ticks_us()

    diff = signal_on-signal_off
    distance = diff/58.0
    print("Distance="+str(distance))
```

6. The last step is to generate a warning sound on the buzzer. To do this, we check if the distance is less than 100 cm. If so, we generate a PWM signal with the selected duty cycle on the buzzer, thus emitting a sound wave. The entered value will translate into the volume of the emitted sound. Then we set a delay proportional to the distance. In this case, the distance was divided by 50 so that the warning sound was not too long. The value of 50 was selected empirically, it does not make much physical sense. Then we turn off the buzzer by setting the fill to 0 and also wait for a time proportional to the distance.

```
import machine
import utime

trigger = machine.Pin(18, machine.Pin.OUT)
echo = machine.Pin(19, machine.Pin.IN)

buzzer = machine.PWM(machine.Pin(16))
buzzer.freq(1000)

while True:
    trigger.value(0)
    utime.sleep_us(2)
    trigger.value(1)
    utime.sleep_us(10)
    trigger.value(0)
```

```

    while echo.value()==0:
        signal_off = utime.ticks_us()

    while echo.value()==1:
        signal_on = utime.ticks_us()

    diff = signal_on-signal_off
    distance = diff/58.0
    print("Distance="+str(distance))

    if distance<100:
        buzzer.duty_u16(16383)
        utime.sleep(distance/50)
        buzzer.duty_u16(0)
        utime.sleep(distance/50)
```

Now the program is ready and you can test it.

7.2 Example 8: GPS module

In this example, we will focus on the asynchronous UART communication bus, which will be used to receive data from the Waveshare Neo-6m/7m GPS module. The Raspberry Pi Pico has 2 UART ports: UART0 and UART1 (see Fig: 1.1). We will connect the GPS module to UART1 (GP4 and GP5 pins). Just remember that the lines to Rx (Receiver) and Tx (Transmitter) should be connected crosswise, i.e. the Rx line from the GPS module to the Tx of the Raspberry Pi Pico (GP4) and the Tx line from the GPS to the Rx of the Raspberry Pi Pico (GP5). We connect the power supply of the GPS module to 3.3V. So connect the system as shown in Fig. 7.4.

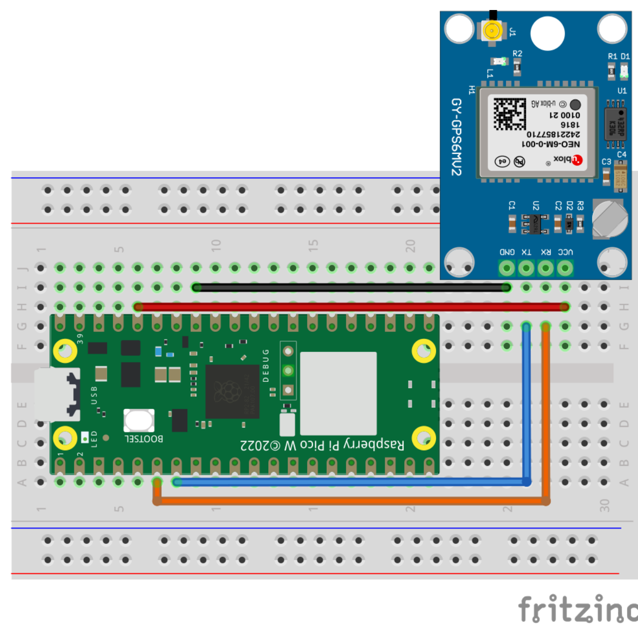


Figure 7.4: Connecting the electronic circuit from example 8.

Now open the Thonny editor and let's try to write code that will receive data frames from the GPS module:

1. First, you need to add the *machine* and *utime* libraries and configure the UART. For this purpose, use the *machine.UART()* function, which takes the UART port number as its first argument, i.e. the value 0 or 1. We connected the GPS module to UART1, so we enter 1. Then you need to provide the baund rate and the pins to which the GPS module is connected. If you look at Fig. 1.1, UART1 is available on two sets of pins: GP4 and GP5 or GP8 and GP9. So you need to indicate which pins are used to connect the GPS module:

```
1  import machine
2  import utime
3
4  uart = machine.UART(1, baudrate=9600, tx=machine.Pin(4),
     ↪ rx=machine.Pin(5))
```

2. Then in the main loop we check if there is any data available in the UART buffer with the *any()* function and if there is, we read the line with the *readline()* function. We display the read line in the terminal:

```
1  import machine
2  import utime
3
4  uart = machine.UART(1, baudrate=9600, tx=machine.Pin(4),
     ↪ rx=machine.Pin(5))
5
6  while True:
7      if uart.any():
8          line = uart.readline()
9          print(line)
10     utime.sleep(1)
```

When you run the program, at the beginning you will get the result as shown in Fig. 7.5.

```
b'$GPVTG,,,,,,,,,N*30\r$GPRMC,,V,,,,,,,,,$GPRMC,,V,,,,,,,,,,N$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPRMC,,V,,,,,,,,,,N*$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,,,,,,,,N*30\r$GPRMC,,V,,,,,,,,,$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,,,,,,,,N*30\r\n'
b'$GPGGA,,,$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,,,,,,,,N*30\r\n'
b'$GPGGA,,,,,,0,00,99.99,,,,,,*48\r\n'
b'$GPG$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,,,,,,,,N*30\r\n'
b'$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,$GPRMC,,V,,,,,,,,,,N*53\r\n'
b'$GPVTG,,,,,,,,,N*30\r\n'
MicroPython (Raspberry Pi Pico) • Board in FS mode @ /dev/cu.usbmodem11201
```

Figure 7.5: Example output from GPS module without getting fix to the satellites.

This result means that the GPS module has not yet get fix to the satellites and is sending empty characters between the commas. When it gets fix, data will appear between the commas, which we will be able to interpret (e.g. as shown in Fig. 7.6). How to interpret data from GPS? Data received from GPS is recorded in accordance with the NMEA (National Marine Electronics Association) protocol, in which each

```
$GPGLL,2232.73995,N,11404.60273,E,030427.00,A,A*6B
$GPRMC,030428.00,A,2232.73995,N,11404.60275,E,0.037,,070314,,,A*7E
$GPVTG,,T,,M,0.037,N,0.069,K,A*28
$GPGGA,030428.00,2232.73995,N,11404.60275,E,1,07,1.17,122.5,M,-2.3,M,,*4F
$GPGSA,A,3,29,21,18,05,14,22,26,,,,,,2.12,1.17,1.77*00
$GPGSV,3,1,10,05,18,096,31,12,07,154,15,14,12,248,29,15,44,025,*7B
$GPGSV,3,2,10,18,38,325,43,21,61,296,41,22,09,304,31,24,70,114,*7D
$GPGSV,3,3,10,26,10,045,16,29,16,208,35*7B
$GPGLL,2232.73995,N,11404.60275,E,030428.00,A,A*62
$GPRMC,030429.00,A,2232.73994,N,11404.60277,E,0.017,,070314,,,A*7E
$GPVTG,,T,,M,0.017,N,0.031,K,A*27
$GPGGA,030429.00,2232.73994,N,11404.60277,E,1,07,1.17,122.7,M,-2.3,M,,*4F
$GPGSA,A,3,29,21,18,05,14,22,26,,,,,,2.12,1.17,1.77*00
$GPGSV,3,1,10,05,18,096,31,12,07,154,14,14,12,248,29,15,44,025,*7A
$GPGSV,3,2,10,18,38,325,43,21,61,296,41,22,09,304,31,24,70,114,21*7E
$GPGSV,3,3,10,26,10,045,14,29,16,208,35*79
$GPGLL,2232.73994,N,11404.60277,E,030429.00,A,A*60
```

Figure 7.6: Example output from GPS, which got fix to the satellites. Source: https://www.waveshare.com/wiki/File:UART-GPS-NEO-6M-User-Manual-2.png.

sequence begins with an identifier, e.g. GPGGA - Global Positioning System Fix Data. So to read the interesting data, you need to find the appropriate sequence identifier, in which the information you are looking for is located. In this case, we want to read the position, so we are only interested in the GPGGA frame. The rest of the frames are discussed here: https://aprs.gids.nl/nmea/ . The GPPGA frame looks like shown in Fig. 7.7.

Name	Example Data	Description
Sentence Identifier	$GPGGA	Global Positioning System Fix Data
Time	170834	17:08:34 Z
Latitude	4124.8963, N	41d 24.8963' N or 41d 24' 54" N
Longitude	08151.6838, W	81d 51.6838' W or 81d 51' 41" W
Fix Quality: - 0 = Invalid - 1 = GPS fix - 2 = DGPS fix	1	Data is from a GPS fix
Number of Satellites	05	5 Satellites are in view
Horizontal Dilution of Precision (HDOP)	1.5	Relative accuracy of horizontal position
Altitude	280.2, M	280.2 meters above mean sea level
Height of geoid above WGS84 ellipsoid	-34.0, M	-34.0 meters
Time since last DGPS update	blank	No last update
DGPS reference station id	blank	No station id
Checksum	*75	Used by program to check for transmission errors

Figure 7.7: GPGGA sequence. Source: https://aprs.gids.nl/nmea/

3. To extract a GPGGA frame, first convert the read text in the form of bytes to a string with utf-8 encoding. For this purpose, the *decode()* function is used. Then, we display only those lines that start with $GPGGA. For this purpose, we use the *startswith()* function, which returns *True* if the string starts with the selected word given in brackets:

```
import machine
import utime

uart = machine.UART(1, baudrate=9600, tx=machine.Pin(4),
    ↪ rx=machine.Pin(5))

while True:
    if uart.any():
        line = uart.readline()
        line = line.decode('utf-8')
        if line.startswith('$GPGGA'):
            print(line)
     utime.sleep(1)
```

4. Next, we need to split the string into fragments. Each piece of information in the string is separated by a comma, so we will use the *split()* function, which splits the string into fragments according to the separation character given in the brackets.

```
import machine
import utime

uart = machine.UART(1, baudrate=9600, tx=machine.Pin(4),
    ↪ rx=machine.Pin(5))

while True:
    if uart.any():
        line = uart.readline()
        line = line.decode('utf-8')
        if line.startswith('$GPGGA'):
            parts = line.split(',')
            latitude = parts[2]
            longitude = parts[4]
            print(latitude+","+longitude)
    utime.sleep(1)
```

Now the program is ready and can be tested. To check if we have read the correct location, we can use google maps. Then in the search field we should enter the latitude and longitude in the form e.g.: 52 13.30093, 21 00.41831.

7.3 Example 9: Weather station

The Serial Peripheral Interface consists of four lines:

- SPI SCK - serial clock, which allows to synchronize devices,
- SPI TX (formerly called MOSI - *Master Output Slave Input*) - line, which transmits bits from master device to all slave devices,
- SPI RX (formerly called MISO - *Master Input Slave Output*) - line, which transmits bits from slave devices to master device,
- SPI CS (Chip select) - line, which activate communication with chosen slave device. The slave device starts to listen and respond when a low value is set on its CS line. This line is sometimes called *Slave select* and then marked as SS.

To point that active state is low in line name frequently a line above the text is placed, like $\overline{SS}$ or $\overline{CS}$.

The schematic connection between devices (block diagram) is shown in Fig. 7.8. The master is only one and this device receives information from slave devices on demand. The master device generates the clock signal, which will synchronize all devices. It is possible to connect many slave devices (e.g. sensors) to master (e.g. Raspberry Pi Pico board) but all slave devices need to have separate SS/CS lines. It is worth to mention that SPI is full-duplex bus. It means that data can be transmitted and received by device in the same time.

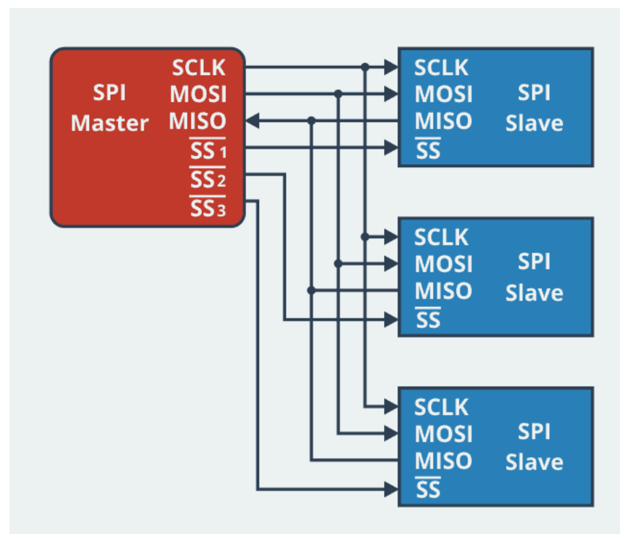


Figure 7.8: The schematic connection between devices using SPI bus. Source: https://forbot.pl/blog/kurs-stm32-f4-10-obsluga-spi-wyswietlacz-oled-id13475.

The dedicated library to use SPI bus is called *SPI* but most sensors have ready libraries for MicroPython, which can be found in Thonny editor. This example will show how to use ready libraries for sensors, which send data using SPI interface.

As an example, we will make a weather station based on the BME280 sensor, which allows measuring of temperature, pressure and humidity. First, you need to install the library. In Thonny, we can search for ready-made libraries by selecting *Tools* → *Manage Packages* from the menu. Then, you need to enter the name of the sensor for which you are looking library for. Since the BME280 sends data via both SPI and I2C, you can additionally add the bus name when searching for a library, e.g. as shown in Fig. 7.9.
Then, select one of the ready-made libraries. In this example, we will use the ***bme280-upy*** library. Click on it and then press the *Install* button.
Next, you need to connect the sensor to the Raspberry Pi Pico, which has 2 independent SPI buses: SPI0 and SPI1 (see Fig. 1.1). We will use SPI0 available on pins GP16-GP19. Connect the sensor according to Fig. 7.11.
When installing the *bme280-upy* library (see Fig. 7.10), basic information about the library is provided along with links to the repository and the PyPI page.

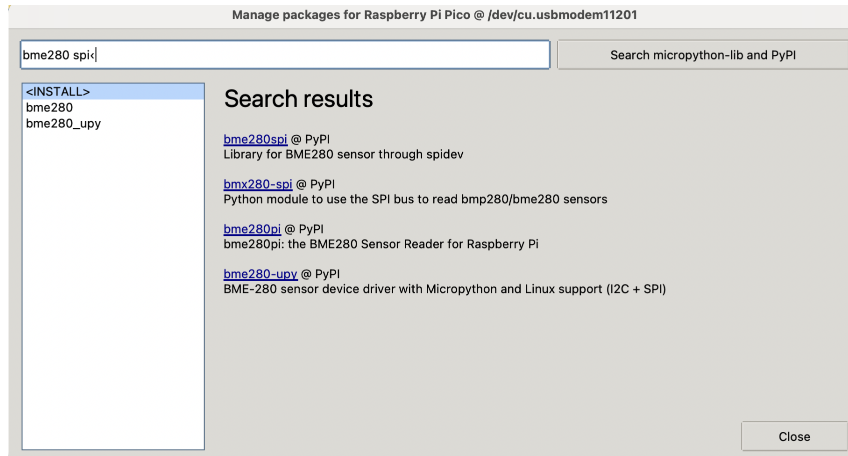


Figure 7.9: Searching a library for BME280 sensor.

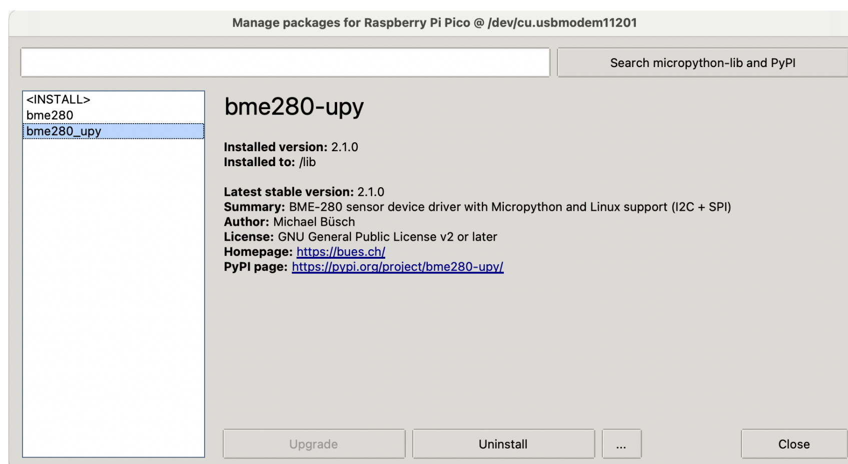


Figure 7.10: Information about *bme280-upy* library.

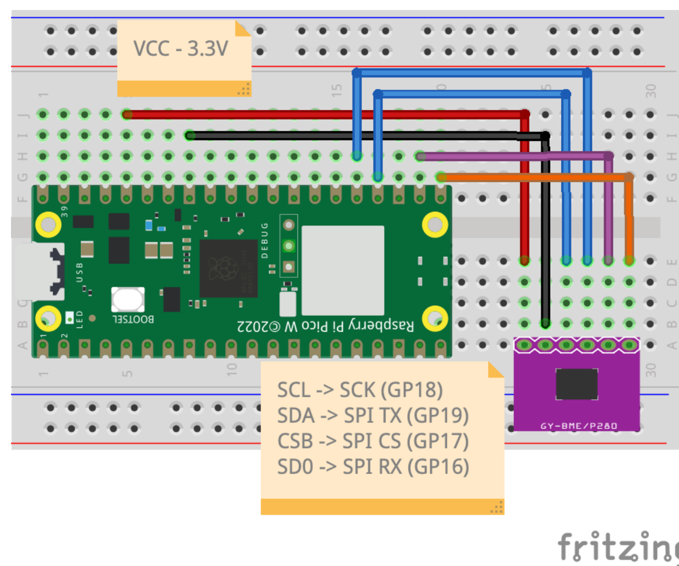


Figure 7.11: Connecting the electronic circuit from example 9.

Open the PyPI page and note that usually the library creator provides an example of how to use the library along with other necessary information.

Let's create a program that reads temperature, pressure and humidity from sensor, based on the example from the PyPI page of bme280-upy library. To do this, follow these steps:

a. First, add the machine and bme280 libraries:

```
import machine
import bme280
```

b. Then you need to configure the sensor by specifying which bus you will use for communication and if it is SPI, you should also specify the CS pin that was used:

```
import machine
import bme280

bme = bme280.BME280(spiBus=0, spiCS=17)
```

c. Then you need to read the data from the sensor, which returns the temperature in degrees Celsius. In the case of humidity, you need to multiply the result by 100 to get a result in percent and the pressure should be divided by 100 to get a result in hPa:

```
import machine
import bme280

bme = bme280.BME280(spiBus=0, spiCS=17)

while True:
    temperature, humidity, pressure = bme.readForced(
       ↪ filter=bme280.FILTER_2, tempOversampling=
       ↪ bme280.OVSMPL_4, humidityOversampling=bme280
       ↪ .OVSMPL_4, pressureOversampling=bme280.
       ↪ OVSMPL_4)
    humidity = humidity*100
    pressure = pressure/100
    print("T="+str(temperature)+", humidity="+str(
       ↪ humidity)+", pressure ="+str(pressure))
```

Run the program and test how it works.

7.4 Example 10: Indoor air quality measurement

The Inter-Integrated Circuit bus consists on two lines:

- SDA (Serial Data Line) is a data line, which is used to send data between master and slave devices.
- SCL (Serial Clock Line).

Both lines are connected to VCC by pull-up resistors.

In this example, we will not delve into the I2C transmission protocol, but we will use a ready-made sensor library. Note in Fig. 1.1 that we have two I2C buses (marked as *I2C0* and *I2C1*) led out on several pins.
As an example, we will create a program to read air quality parameters in a room such as: Air Quality Index, CO2 concentration (eC02) and total volatile organic compounds (TVOC) concentration. For this purpose, we will use the DFROBOT ENS160 Air Quality Sensor. The sensor should be connected to the Raspberry Pi according to table no. 7.1.

ENS160 sensor	Raspberry Pi Pico W
3V3	3V3
GND	GND
SCL	GP15
SDA	GP14

Table 7.1: Connection of DFROBOT ENS160 sensor to Raspberry Pi Pico W.

This time, we will not find a ready-made library in the Thonny editor package manager. What to do in such a case? You can search for a ready-made library on the Internet and download it, e.g. from github. In this example, we will use the repository `https://github.com/TimHanewich/Air-Quality-IoT/tree/master` . To use the ready library shared on github, you need to download its sources by clicking on the *Code* button and selecting *Download ZIP* as shown in Fig. 7.12.

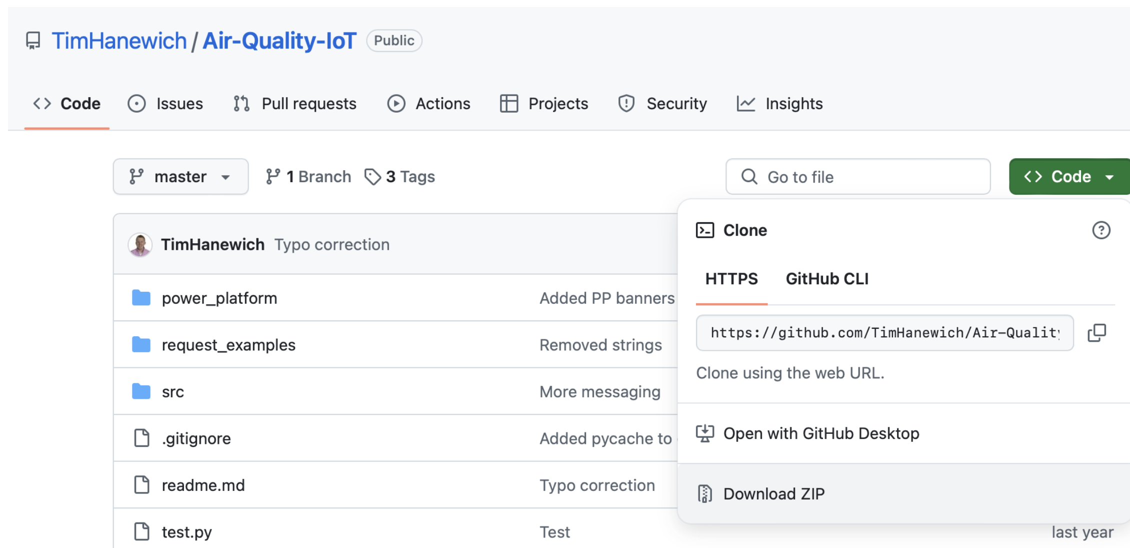


Figure 7.12: Downloading the library from the github repository.

Then you need to unpack the zip file to the folder. Go to Thonny and connect to the Raspberry Pi Pico board and then select *VIEW* → *Files* in the top panel. Then on the left side we will have a preview of the directories on the computer and below the content of the Raspberry Pi Pico board. Copy the ENS160 library (src/ENS160.py) to the Raspberry Pi Pico as shown in Fig. 7.13.
Usually, in the downloaded repository there is an example of how to use the library. In this case, we will also find an example called *main.py* but it will contain much

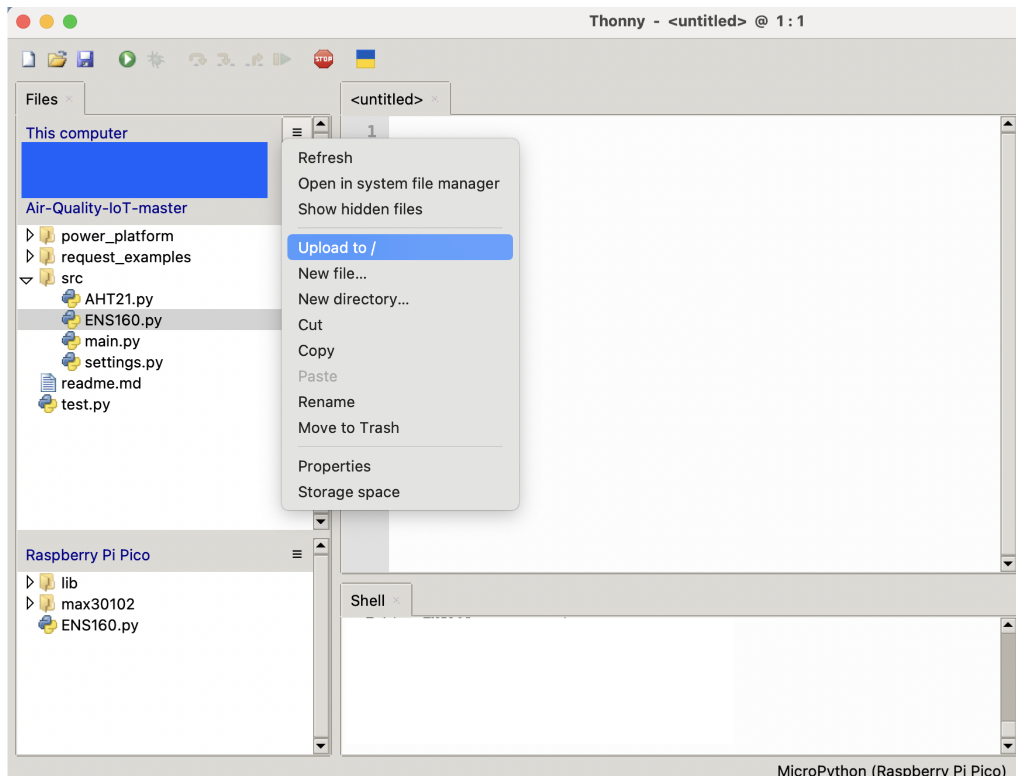


Figure 7.13: Installing the downloaded library onto the Raspberry Pi Pico board.

more information than we need (connection to the AHT21 sensor and connection to Wi-Fi). Open the *main.py* file and remove unnecessary elements, the program should then looks like this:

```
import machine
import utime
import ENS160

# set up
print("Setting up ENS160")
i2c = machine.I2C(1, sda=machine.Pin(14), scl=machine.Pin
    ↪ (15))
ens = ENS160.ENS160(i2c)
ens.reset()
utime.sleep(0.5)
ens.operating_mode = 2
utime.sleep(2.0)

while True:
    # take reading from ENS160
    print("Taking ENS160 measurements... ")
    aqi = ens.AQI
    eco2 = ens.CO2
    tvoc = ens.TVOC
    print("AQI: " + str(aqi) + ", ECO2: " + str(eco2) + ",
        ↪ TVOC: " + str(tvoc))
```

```
21        utime.sleep(1)
```

Run the program and test it. How to interpret the read data? Just familiarize yourself with the tables in the sensor documentation (see Fig. 7.14, 7.15, 7.16).

AQI Reference

Level	Description	Suggestion	Recommended Stay Time
5	Extremely bad	In unavoidable circumstances, please strengthen ventilation	Not recommended to stay
4	Bad	Strengthen ventilation, find sources of pollution	Less than one month
3	Moderate	Strengthen ventilation, close to the water source	Less than 12 months
2	Good	Maintain adequate ventilation	Suitable for long-term living
1	Excellent	No suggestion	Suitable for long-term living

Figure 7.14: AQI Reference.

eCO2/CO2 Concentration Reference

eCO2/CO2	Level	Result/Suggestion
21500	Terrible	Indoor air pollution is serious/Ventilation is required
1000-1500	Bad	Indoor air is polluted/ Ventilation is recommended
800-1000	Moderate	It is OK to ventilate
600-800	Good	Keep normal
400-600	Eexcellent	No suggestion

Figure 7.15: eC02 Concentration Reference.

TVOC Concentration Reference

TOVC (ppb)	Effects on Human Health
>6000	Headaches and nerve problem
750-6000	Restless and headache
50-750	Restless and uncomfortable
<50	No effect

Figure 7.16: TVOC Reference.

7.5 Example 11: Temperature measurement using external A/D converter

In this example we will learn how to read the voltage value from an analog-to-digital converter (ADC). Up to now we have been using the built-in analog-to-digital converter, which is 12 bit (resolution of about 800 μV). If we want better accuracy, we can use an external converter, e.g. ADS1115 (16 bit - resolution of about 76 μV). In this example we will connect the LM35 temperature sensor to the ADS1115 converter according to Fig. 7.17.

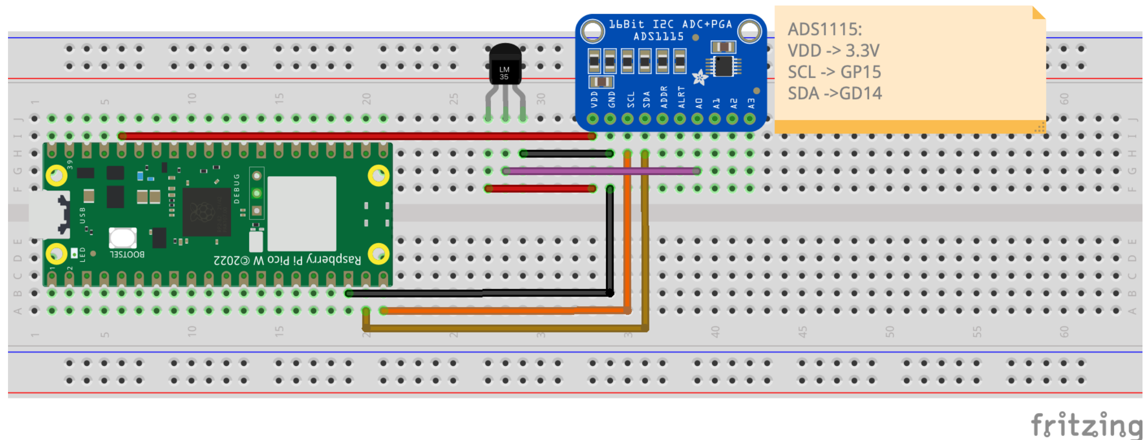


Figure 7.17: Connecting the electronic circuit from example 11.

In the case of the ADS1115 converter, we will not use a ready-made library that could be found on Github, but we will write our own library. Why? The last two examples showed how to use the SPI and I2C buses with ready-made libraries and we did not delve into the data transmission protocol. Now we will not use ready-made libraries to show how to communicate with a converter or another sensor with generic communication interface library (which is available out of the box in MicroPython for Raspberry Pi Pico). First, you should start by finding a datasheet for the ADS1115 converter on the Internet (https://www.ti.com/lit/ds/symlink/ads1115.pdf).

Then you need to find three things in the documentation:

- writing protocol
- reading protocol
- register addresses along with the values that need to be sent to them to set the given parameters.

This is a general diagram of what you need to do, and in more detail:

a. Scan what devices are connected to the I2C bus to find the A/D converter address. To do this, first configure the I2C buses using the machine.I2C() function, which takes the I2C bus number as the first argument (0 for I2C0 or 1 for I2C1 - the numbers are in Fig. 1.1. In our case, it will be I2C1). The second argument is the I2C bus clock frequency. The third and fourth arguments are the pins that were used to connect SDA and SCL.

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
```

b. Next, we use the scan() function, which returns the addresses of devices connected to the I2C bus in the form of a list. Then we read the values from the list using a for loop. To display the data in hexadecimal code, which is how addresses are usually written, we use the hex function, which converts decimal values to hexadecimal:

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
devices = i2c.scan()
for device in devices:
        print(hex(device))
```

c. Run the code and then save the read address to the variable ADS1115_I2C_ADDRESS:

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
devices = i2c.scan()
for device in devices:
    print(hex(device))

ADS1115_I2C_ADDRESS = 0x48
```

Tip 7.5.1

You can check obtained address of the ADS1115 A/D converter with table 7.2 of the ADS1115 datasheet. The hexadecimal value 0x48 is 0b1001000 in binary format. You can use: print(bin(device)) in your program.

d. Then you need to find the addresses of the registers inside the ADC, where the configuration and the values measured by the ADC are stored. The addresses can be found in the sensor documentation in Table 6 as shown in Fig. 7.18. Notice, that in this table, address pointer register is described with individual bits. Single byte has 8 bits [7:0]. Bits from 7 (oldest) to 2 are reserved and you should always write 0 to them. Bits from 1 to 0 are important and we have 4 addresses, where we only use the '00' and '01' addresses.

Pass the read addresses to variables in the program.

Table 6. Address Pointer Register Field Descriptions

Bit	Field	Type	Reset	Description
7:2	Reserved	W	0h	Always write 0h
1:0	P[1:0]	W	0h	**Register address pointer** 00 : Conversion register 01 : Config register 10 : Lo_thresh register 11 : Hi_thresh register

Figure 7.18: Table 6 from datasheet. Source: https://www.ti.com/lit/ds/symlink/ads1115.pdf

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
devices = i2c.scan()
for device in devices:
        print(hex(device))

ADS1115_I2C_ADDRESS = 0x48
ADS1115_CONVERSION_REG = 0x00
ADS1115_CONFIG_REG = 0x01
```

e. Then, you need to look for information on what parameters of the ADC can be set and what values should be sent to the config register for this purpose. All this data is in Table 8 (see Fig. 7.19 and 7.20)) and the letter *h* designations indicate values written in hexadecimal code, e.g. 1h=0x01, which must be set on individual bits of the 16-bit configuration register. In this example, we will perform a single measurement from channel 0 in the range of $\pm$4.096V. We will not use a comparator. The necessary values have been written down for the variables:

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
devices = i2c.scan()
for device in devices:
    print(hex(device))

ADS1115_I2C_ADDRESS = 0x48
ADS1115_CONVERSION_REG = 0x00
ADS1115_CONFIG_REG = 0x01

ADS1115_CONFIG_OS_SINGLE = 0x8000  # Start a single
    ↪ conversion
ADS1115_CONFIG_MUX_AIN0 = 0x4000  #AIN0 Chain (Single-
    ↪ ended channel)
ADS1115_CONFIG_GAIN = 0x0200  #Gain amplifier +-4.096V
```

```
ADS1115_CONFIG_MODE_SINGLE = 0x0100  #Single-shot mode
ADS1115_CONFIG_DR_128SPS = 0x0080  #Data rate = 128
    ↪ SPS
ADS1115_CONFIG_COMP_DISABLED = 0x0003  #disable
    ↪ comparator
```

Table 8. Config Register Field Descriptions

Bit	Field	Type	Reset	Description
15	OS	R/W	1h	**Operational status or single-shot conversion start** This bit determines the operational status of the device. OS can only be written when in power-down state and has no effect when a conversion is ongoing. When writing: 0 : No effect 1 : Start a single conversion (when in power-down state) When reading: 0 : Device is currently performing a conversion 1 : Device is not currently performing a conversion
14:12	MUX[2:0]	R/W	0h	**Input multiplexer configuration (ADS1115 only)** These bits configure the input multiplexer. These bits serve no function on the ADS1113 and ADS1114. 000 : AIN_P = AIN0 and AIN_N = AIN1 (default) 001 : AIN_P = AIN0 and AIN_N = AIN3 010 : AIN_P = AIN1 and AIN_N = AIN3 011 : AIN_P = AIN2 and AIN_N = AIN3 100 : AIN_P = AIN0 and AIN_N = GND 101 : AIN_P = AIN1 and AIN_N = GND 110 : AIN_P = AIN2 and AIN_N = GND 111 : AIN_P = AIN3 and AIN_N = GND
11:9	PGA[2:0]	R/W	2h	**Programmable gain amplifier configuration** These bits set the FSR of the programmable gain amplifier. These bits serve no function on the ADS1113. 000 : FSR = ±6.144 V[1] 001 : FSR = ±4.096 V[1] 010 : FSR = ±2.048 V (default) 011 : FSR = ±1.024 V 100 : FSR = ±0.512 V 101 : FSR = ±0.256 V 110 : FSR = ±0.256 V 111 : FSR = ±0.256 V
8	MODE	R/W	1h	**Device operating mode** This bit controls the operating mode. 0 : Continuous-conversion mode 1 : Single-shot mode or power-down state (default)
7:5	DR[2:0]	R/W	4h	**Data rate** These bits control the data rate setting. 000 : 8 SPS 001 : 16 SPS 010 : 32 SPS 011 : 64 SPS 100 : 128 SPS (default) 101 : 250 SPS 110 : 475 SPS 111 : 860 SPS

Figure 7.19: Config register field description - part 1. Source: https://www.ti.com/lit/ds/symlink/ads1115.pdf

Table 8. Config Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
4	COMP_MODE	R/W	0h	**Comparator mode (ADS1114 and ADS1115 only)** This bit configures the comparator operating mode. This bit serves no function on the ADS1113. 0 : Traditional comparator (default) 1 : Window comparator
3	COMP_POL	R/W	0h	**Comparator polarity (ADS1114 and ADS1115 only)** This bit controls the polarity of the ALERT/RDY pin. This bit serves no function on the ADS1113. 0 : Active low (default) 1 : Active high
2	COMP_LAT	R/W	0h	**Latching comparator (ADS1114 and ADS1115 only)** This bit controls whether the ALERT/RDY pin latches after being asserted or clears after conversions are within the margin of the upper and lower threshold values. This bit serves no function on the ADS1113. 0 : Nonlatching comparator . The ALERT/RDY pin does not latch when asserted (default). 1 : Latching comparator. The asserted ALERT/RDY pin remains latched until conversion data are read by the master or an appropriate SMBus alert response is sent by the master. The device responds with its address, and it is the lowest address currently asserting the ALERT/RDY bus line.
1:0	COMP_QUE[1:0]	R/W	3h	**Comparator queue and disable (ADS1114 and ADS1115 only)** These bits perform two functions. When set to 11, the comparator is disabled and the ALERT/RDY pin is set to a high-impedance state. When set to any other value, the ALERT/RDY pin and the comparator function are enabled, and the set value determines the number of successive conversions exceeding the upper or lower threshold required before asserting the ALERT/RDY pin. These bits serve no function on the ADS1113. 00 : Assert after one conversion 01 : Assert after two conversions 10 : Assert after four conversions 11 : Disable comparator and set ALERT/RDY pin to high-impedance (default)

Figure 7.20: Config register field description - part 2. Source: https://www.ti.com/lit/ds/symlink/ads1115.pdf

f. In the next step we will combine all set bits into one 16 bit value to send it to the configuration register:

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
    ↪ sda=machine.Pin(14))
devices = i2c.scan()
for device in devices:
    print(hex(device))

ADS1115_I2C_ADDRESS = 0x48
ADS1115_CONVERSION_REG = 0x00
ADS1115_CONFIG_REG = 0x01

ADS1115_CONFIG_OS_SINGLE = 0x8000
ADS1115_CONFIG_MUX_AIN0 = 0x4000
ADS1115_CONFIG_GAIN = 0x0200
ADS1115_CONFIG_MODE_SINGLE = 0x0100
ADS1115_CONFIG_DR_128SPS = 0x0080
ADS1115_CONFIG_COMP_DISABLED = 0x0003

ADS1115_CONFIG = (
ADS1115_CONFIG_OS_SINGLE
| ADS1115_CONFIG_MUX_AIN0
```

```
| ADS1115_CONFIG_GAIN
| ADS1115_CONFIG_MODE_SINGLE
| ADS1115_CONFIG_DR_128SPS
| ADS1115_CONFIG_COMP_DISABLED
)
```

g. In the next step, we will create the function for sending our value to the register. To do this, we need to see what the communication protocol looks like. According to the documentation (see Fig. 7.21), first you need to send the device address (ADC I2C address), then the register address (e.g. configuration) and then the data bytes. The ADS1115 converter is 16 bits device, and first you need to send the 8 oldest bits (designation: D15-D8) and then the 8 youngest bits (designation: D7-D0). This is called Big-endian. Usually in our computer memory values are stored in Little-endian configuration which is opposite to this. We will have to deal with this. It will be most convenient to make function for sending data and inside a 4-element array with the data that needs to be sent (device address, register address, oldest bits, youngest bits). In the array, we will store data in the form of bytes, hence we will use the bytearray() function, which creates an array with data stored in the form of bytes.

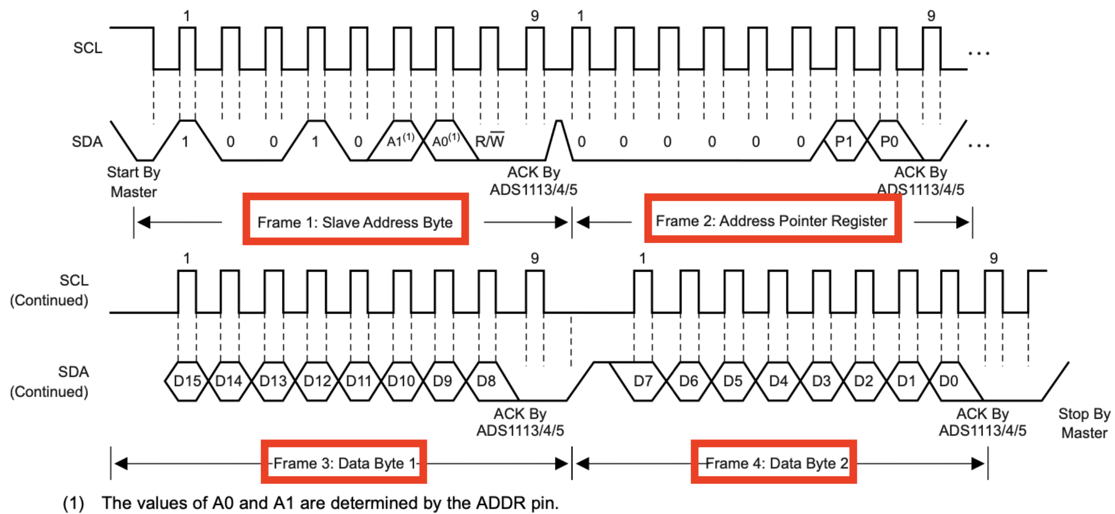


Figure 7.21: Timing diagram from writing. Source: https://www.ti.com/lit/ds/symlink/ads1115.pdf

```
import machine
...

ADS1115_CONFIG = (
        ADS1115_CONFIG_OS_SINGLE
        | ADS1115_CONFIG_MUX_AIN0
        | ADS1115_CONFIG_GAIN
        | ADS1115_CONFIG_MODE_SINGLE
        | ADS1115_CONFIG_DR_128SPS
        | ADS1115_CONFIG_COMP_DISABLED
```

```
)

def write_ads1115_register(register, value):
    data = bytearray([register, (value >> 8) & 0xFF,
      ↪ value & 0xFF])
```

In this snippet above, when we create a bytearray, you may notice two unknown entries:

- (value >> 8) - this is an operation of shifting 8 bits to the right, i.e. when the data is 16 bits, e.g. 1010 1010 0011 0011, the result will be 0000 0000 1010 1010. So the 8 older bits will jump to the place of the 8 younger bits, and instead of them there should be only zeros.
- value & 0xFF - this is a logical AND operation, the result of which will be an unchanged 16-bit value. Why such an operation? If the *value* variable contained something more than just the 16 bits that interest us, then we get rid of the rest of the information and pass only 16 bits. In Python, which is dynamic language, we cannot be sure, what value type was passed. This could be 32-bit value with some higher bits set and shift operation could bring bits to 16-bit field. So to be on the safe side it is better to properly mask the value with AND boolean operation.

h. The next step is to send data from the table to the ADC via the I2C bus:

```
import machine
...

ADS1115_CONFIG = (
        ADS1115_CONFIG_OS_SINGLE
        | ADS1115_CONFIG_MUX_AIN0
        | ADS1115_CONFIG_GAIN
        | ADS1115_CONFIG_MODE_SINGLE
        | ADS1115_CONFIG_DR_128SPS
        | ADS1115_CONFIG_COMP_DISABLED
)

def write_ads1115_register(register, value):
    data = bytearray([register, (value >> 8) & 0xFF,
      ↪ value & 0xFF])
    i2c.writeto(ADS1115_I2C_ADDRESS, data)
```

i. We already know how to send data. It's time to create function for reading data. To do this, we need to find information about the data reading protocol in the datasheet (see Fig. 7.22). Reading data is done in two steps. The first step is to send the ADC I2C address and the register address (by setting the Address Pointer Register) from which we want to read data (red rectangles in Fig. 7.22). The second step is to read two bytes of data from the ADC address (dark blue rectangles in Fig. 7.22). So let's create a function that will read data from the register:

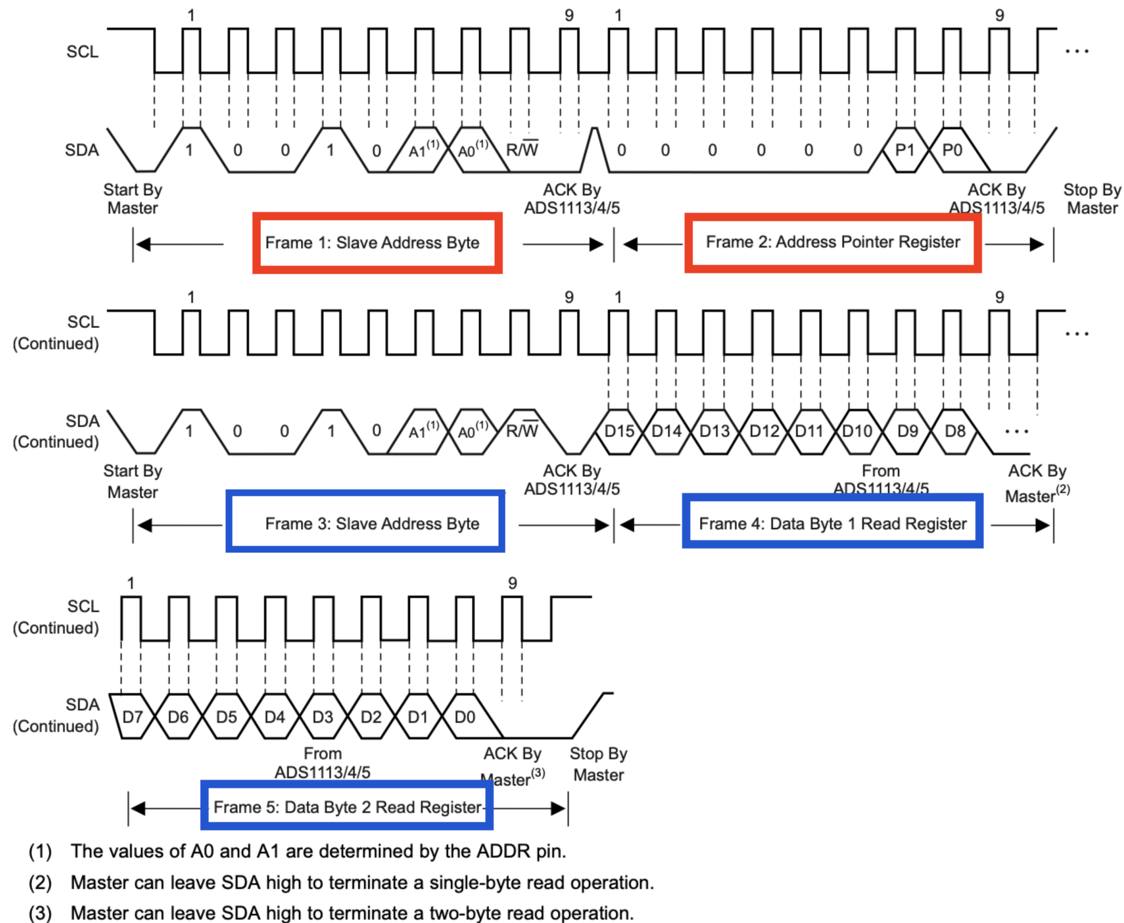


Figure 7.22: Timing diagram from reading. Source: https://www.ti.com/lit/ds/symlink/ads1115.pdf

```python
import machine
...

ADS1115_CONFIG = (
        ADS1115_CONFIG_OS_SINGLE
        | ADS1115_CONFIG_MUX_AIN0
        | ADS1115_CONFIG_GAIN
        | ADS1115_CONFIG_MODE_SINGLE
        | ADS1115_CONFIG_DR_128SPS
        | ADS1115_CONFIG_COMP_DISABLED
)

def write_ads1115_register(register, value):
    data = bytearray([register, (value >> 8) & 0xFF,
    ↪ value & 0xFF])
    i2c.writeto(ADS1115_I2C_ADDRESS, data)

def read_ads1115_register(register, num_bytes):
```

```
    i2c.writeto(ADS1115_I2C_ADDRESS, bytearray([
        register]))
    return i2c.readfrom(ADS1115_I2C_ADDRESS, num_bytes
        )
```

j. We already have functions for reading and writing data via the I2C bus. Now let's create a function in which we will configure the AD converter and read data from it and then convert it to voltage. Here we will use the *from_bytes()* function, which converts values from individual bytes form to decimal value. Notice, that we need to specify how bytes are arranged in memory in *from_bytes()* function. Because these were read as Big-endian, we use **'big'** specifier.

```
import machine
...

def read_ads1115_register(register, num_bytes):
    i2c.writeto(ADS1115_I2C_ADDRESS, bytearray([
        register]))
    return i2c.readfrom(ADS1115_I2C_ADDRESS, num_bytes
        )

def read_adc():
    #Sending configuration to configuration register
    write_ads1115_register(ADS1115_CONFIG_REG,
        ADS1115_CONFIG)

    #Wait for measurement to finish
    utime.sleep(0.01)

    #Reading conversion value
    result = read_ads1115_register(
        ADS1115_CONVERSION_REG, 2)
    raw_value = int.from_bytes(result, 'big')
    if raw_value >= 0x8000:  #Correction for negative
        numbers
        raw_value -= 0x10000

     #Convert value to voltage
     voltage = raw_value * (4.096 / 32768)  #Range
         +-4.096V/max value
     return voltage
```

k. Now let's add the main loop in which we will read the voltage value and convert it to temperature (multiplication by 100 is according to the documentation for the LM35 temperature sensor):

```
import machine
import utime

i2c = machine.I2C(1, freq=400000, scl=machine.Pin(15),
     sda=machine.Pin(14))
devices = i2c.scan()
```

```
for device in devices:
    print(hex(device))

ADS1115_I2C_ADDRESS = 0x48
ADS1115_CONVERSION_REG = 0x00
ADS1115_CONFIG_REG = 0x01

ADS1115_CONFIG_OS_SINGLE = 0x8000
ADS1115_CONFIG_MUX_AIN0 = 0x4000
ADS1115_CONFIG_GAIN = 0x0200
ADS1115_CONFIG_MODE_SINGLE = 0x0100
ADS1115_CONFIG_DR_128SPS = 0x0080
ADS1115_CONFIG_COMP_DISABLED = 0x0003

ADS1115_CONFIG = (
        ADS1115_CONFIG_OS_SINGLE
        | ADS1115_CONFIG_MUX_AIN0
        | ADS1115_CONFIG_GAIN
        | ADS1115_CONFIG_MODE_SINGLE
        | ADS1115_CONFIG_DR_128SPS
        | ADS1115_CONFIG_COMP_DISABLED
)

def write_ads1115_register(register, value):
    data = bytearray([register, (value >> 8) & 0xFF,
      ↪ value & 0xFF])
    i2c.writeto(ADS1115_I2C_ADDRESS, data)

def read_ads1115_register(register, num_bytes):
    i2c.writeto(ADS1115_I2C_ADDRESS, bytearray([
      ↪ register]))
    return i2c.readfrom(ADS1115_I2C_ADDRESS, num_bytes
      ↪ )

def read_adc():
    #Sending configuration to configuration register
    write_ads1115_register(ADS1115_CONFIG_REG,
      ↪ ADS1115_CONFIG)

    #Wait for measurement to finish
    utime.sleep(0.01)

    #Reading conversion value
    result = read_ads1115_register(
      ↪ ADS1115_CONVERSION_REG, 2)
    raw_value = int.from_bytes(result, 'big')
    if raw_value >= 0x8000:  #Correction for negative
      ↪ numbers
        raw_value -= 0x10000

```

```
50      #Convert value to voltage
51      voltage = raw_value * (4.096 / 32768)  #Range
          ↪ +-4.096V/max value
52      return voltage
53
54  while True:
55      voltage = read_adc()
56      temp = voltage*100
57      print("T="+str(temp))
58      utime.sleep(1)
```

The code is ready. Test it out.

7.6 Example 12: Temperature measurement using 1-wire bus

The 1-wire interface was developed by Dallas Semiconductor company. This bus consists of only one line. This interface works similar to I2C bus but the 0 and 1 bits are defined by time duration of low signal. The 1-wire bus works slower than I2C. Maximal speed is 16 kbit/s.
In this example, we will read the temperature from the DS18B20 temperature sensor via the 1-wire bus and display the read temperature on an LCD display with an I2C converter (sometimes the converter is purchased separately for the display). To do this, connect the system as in Fig. 7.23.

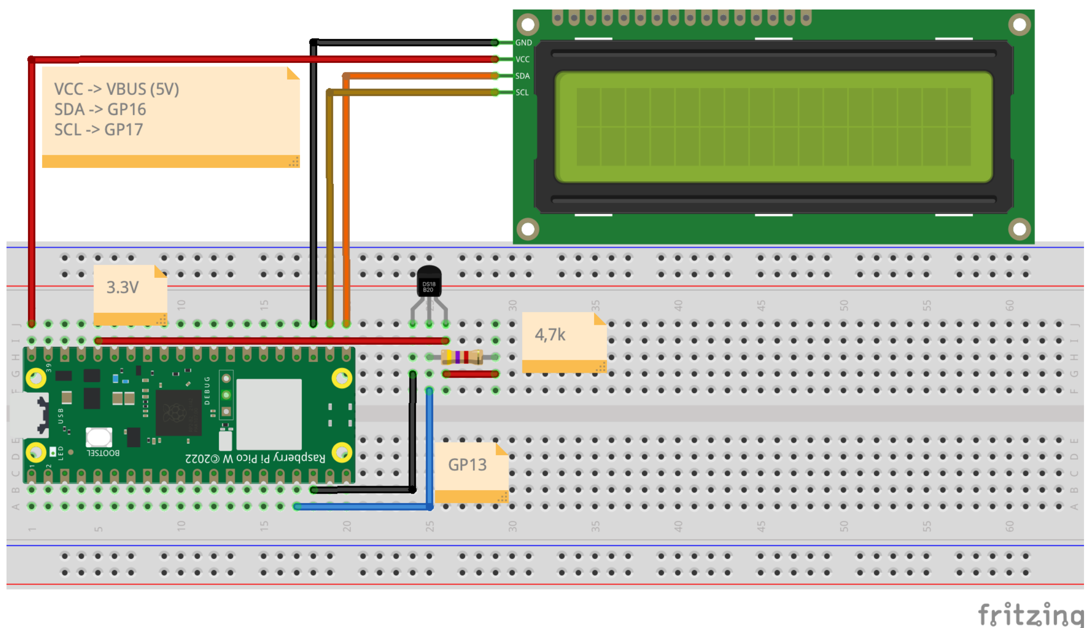


Figure 7.23: Connecting the electronic circuit from example 12.

Now let's write a program that will read data from the DS18B20 sensor and display it on the display. To do this, follow these steps:

a. Install the onwire library and ds18x20 using the package manager in the Thonny editor (see Fig. 7.24 and 7.25).

Figure 7.24: Installation of onewire library.

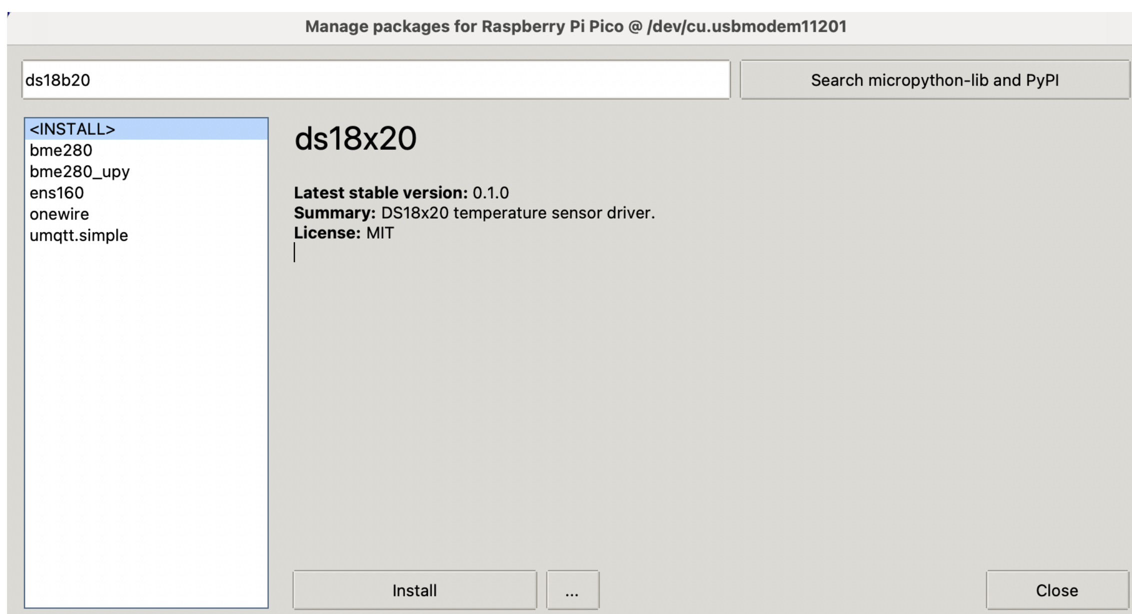


Figure 7.25: Installation of DS18x20 library.

b. Now let's add the necessary libraries:

```
import machine
import onewire
import ds18x20
import utime
```

c. Next, you need to specify which GPIO pin will serve as the 1-wire bus. To do this, use the *onewire.OneWire(pin number)* function. The next step is to configure the DS18B20 sensor, i.e. call the function: *ds18x20.DS18X20()*, which takes a onewire class object as an argument:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
```

d. In the next step, you need to determine the address of the DS18B20 sensor that we connected to the 1-wire bus. This can be done automatically using the scan() function:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()
```

e. Then you should give the sensor a command to measure the temperature (*convert_temp()* function) and then wait about 750 ms until it measures according to the documentation for the DS18B20 sensor. Then you should read the measured temperature value in Celsius using the *read_temp()* function, which takes the sensor address as an argument. Since the *scan()* function returns an array with addresses, you should extract the first element of the array. Arrays in Python are numbered from 0 and to extract a given fragment of the array, you should enter *array_name[index]*:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)

    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

f. You can now test the program. If you read the temperature correctly, we can move on to the LCD display. First, download two files: *lcd_api.py* (https://github.com/T-622/RPI-PICO-I2C-LCD/blob/main/lcd_api.py) and *i2c_lcd.py* (https://github.com/T-622/RPI-PICO-I2C-LCD/blob/main/pico_i2c_lcd.py) and upload them to Raspbery Pi Pico as shown in

Fig. 7.26.

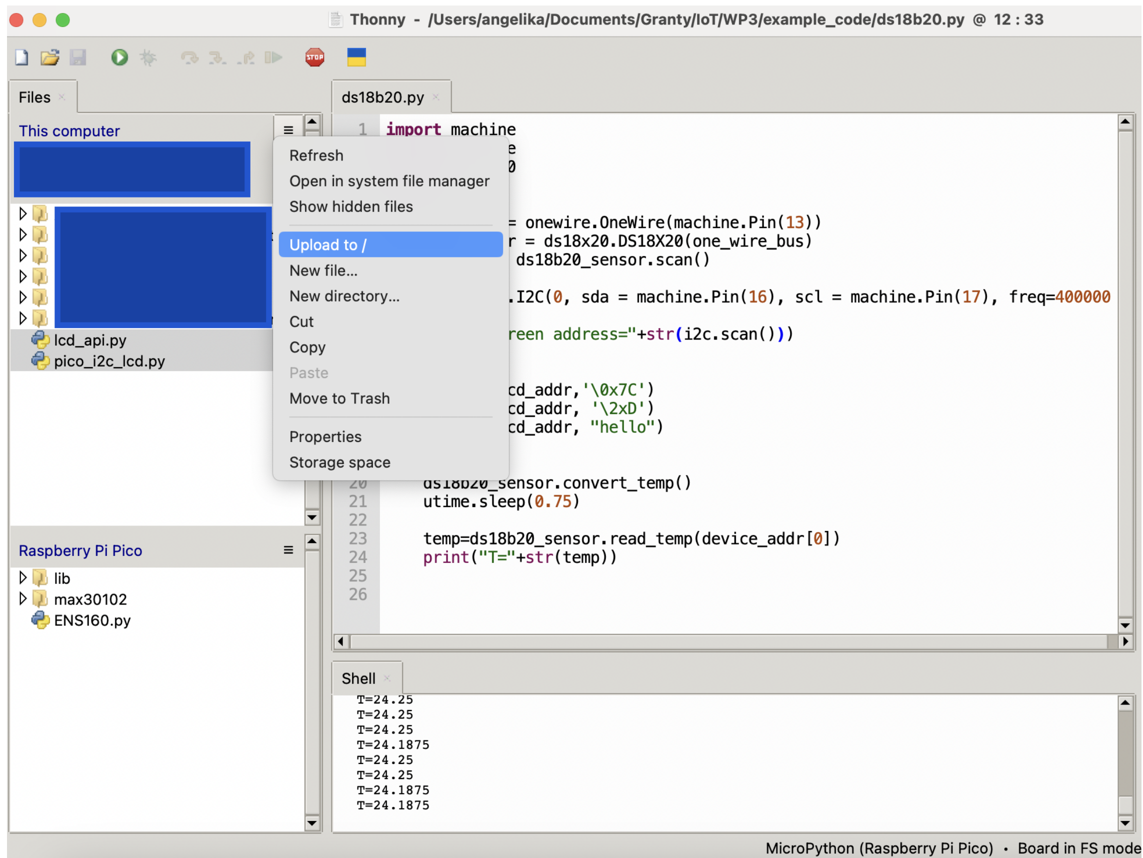


Figure 7.26: Installing libraries for the LCD display

g. Now let's add the necessary libraries:

```python
import machine
import onewire
import ds18x20
import utime
from lcd_api import LcdApi
from pico_i2c_lcd import I2cLcd

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()
...
```

h. Next, let's configure the I2C bus and scan it to find the LCD display address. Run the program and read the value.

```python
import machine
import onewire
import ds18x20
import utime
from lcd_api import LcdApi
from pico_i2c_lcd import I2cLcd
```

```
7
8  one_wire_bus = onewire.OneWire(machine.Pin(13))
9  ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
10 device_addr = ds18b20_sensor.scan()
11
12 i2c = machine.I2C(0, sda = machine.Pin(16), scl =
     ↪ machine.Pin(17), freq=400000)
13 print("LCD screen address="+str(i2c.scan()))
14 ...
```

i. Let's save the read LCD display address to a variable and create variables to store the display dimensions (number of rows and columns):

```
1  import machine
2  import onewire
3  import ds18x20
4  import utime
5  from lcd_api import LcdApi
6  from pico_i2c_lcd import I2cLcd
7
8  one_wire_bus = onewire.OneWire(machine.Pin(13))
9  ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
10 device_addr = ds18b20_sensor.scan()
11
12 i2c = machine.I2C(0, sda = machine.Pin(16), scl =
     ↪ machine.Pin(17), freq=400000)
13 print("LCD screen address="+str(i2c.scan()))
14
15 I2C_ADDR=63
16 I2C_NUM_ROWS = 2
17 I2C_NUM_COLS = 16
18 ...
```

j. Let's initialize the LCD display and then wait 1s. Then let's clear the display, position the cursor at the beginning and display the welcome text:

```
1  import machine
2  import onewire
3  import ds18x20
4  import utime
5  from lcd_api import LcdApi
6  from pico_i2c_lcd import I2cLcd
7
8  one_wire_bus = onewire.OneWire(machine.Pin(13))
9  ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
10 device_addr = ds18b20_sensor.scan()
11
12 i2c = machine.I2C(0, sda = machine.Pin(16), scl =
     ↪ machine.Pin(17), freq=400000)
13 print("LCD screen address="+str(i2c.scan()))
14
15 I2C_ADDR=63
```

```
I2C_NUM_ROWS = 2
I2C_NUM_COLS = 16

lcd = I2cLcd(i2c, I2C_ADDR, I2C_NUM_ROWS, I2C_NUM_COLS
    ↪ )
utime.sleep(1)

lcd.clear()
lcd.move_to(0,0)
lcd.putstr("Hello")
...
```

k. The last step is to display the temperature on the next row of the LCD display after measuring the temperature:

```
import machine
import onewire
import ds18x20
import utime
from lcd_api import LcdApi
from pico_i2c_lcd import I2cLcd

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

i2c = machine.I2C(0, sda = machine.Pin(16), scl =
    ↪ machine.Pin(17), freq=400000)
print("LCD screen address="+str(i2c.scan()))

I2C_ADDR=63
I2C_NUM_ROWS = 2
I2C_NUM_COLS = 16

lcd = I2cLcd(i2c, I2C_ADDR, I2C_NUM_ROWS, I2C_NUM_COLS
    ↪ )
utime.sleep(1)

lcd.clear()
lcd.move_to(0,0)
lcd.putstr("Hello")

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)

    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))

    lcd.move_to(0,1)
    lcd.putstr("T="+str(temp)+" C")
```

The program is ready and can be tested.

Tip 7.6.1

If you would like to use the LCD display without an I2C converter, you can use the library available here: `https://github.com/gusandrioli/liquid-crystal-pico`.

8. Actuators

Actuators are elements making a movement (executive elements). The basic actuators are: servo, stepper motor and DC motor.

8.1 Example 13: Servo motor

The servo is a motor with a gear that rotates by a given angle, usually in the range of $(0;180)^{o}$ or $(-90;90)^{o}$. The construction of the servo is shown in Fig. 8.1. The servo can be connected directly to the Raspberry Pi Pico board. The three wires are brought out of the servo. The middle one (usually red) should be connected to the power supply (+5V = VBUS). The black one is GND and the last wire (usually light colored: white/yellow) is the control that should be connected to the PWM pin.

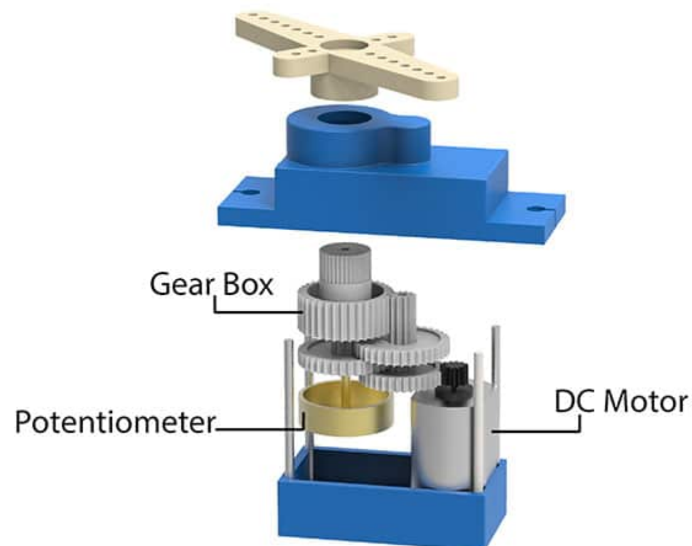


Figure 8.1: Principle of operation of a servomechanism. Source: https://ai.thestempedia.com/docs/quarky/quarky-technical-specifications/servo-motor-with-quarky/.

The servo control is based on the PWM signal. The controller, located in the servo, reads the PWM signal and on its basis determines the angle by which the gear is to rotate. The idea of the servo operation is shown in Fig. 8.2.

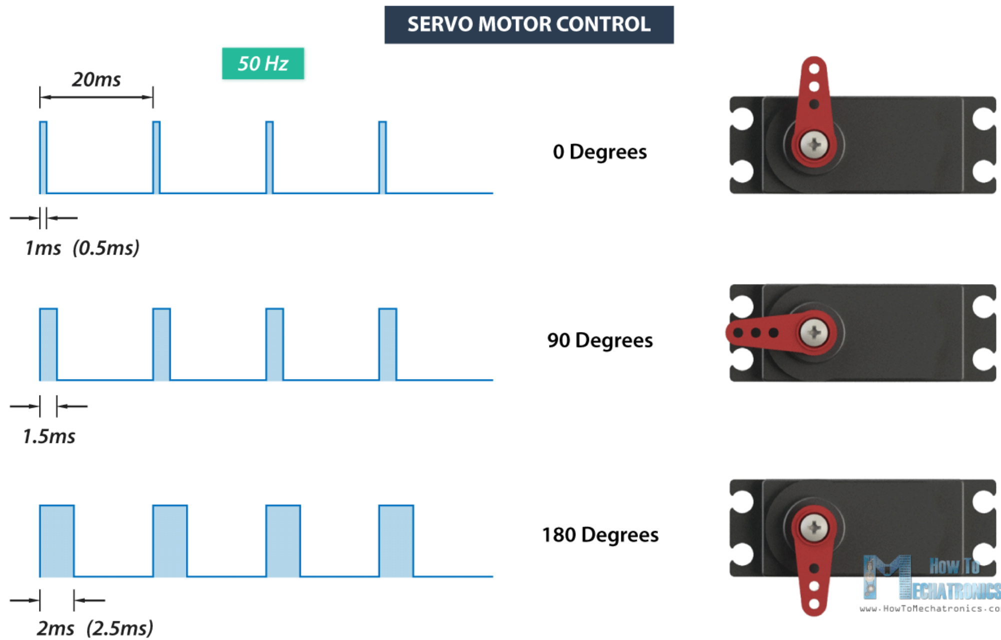


Figure 8.2: Principle of servo control. Source: https://howtomechatronics.com/wp-content/uploads/2018/03/RC-Servo-Motor-Control-Signal.png.

Let's create a program that will rotate the servo from minimum to middle position and then to maximum and back. To do this, let's first connect the system as shown in Fig. 8.3 (red wire- VBUS, yellow wire- GP18, black wire- GND).

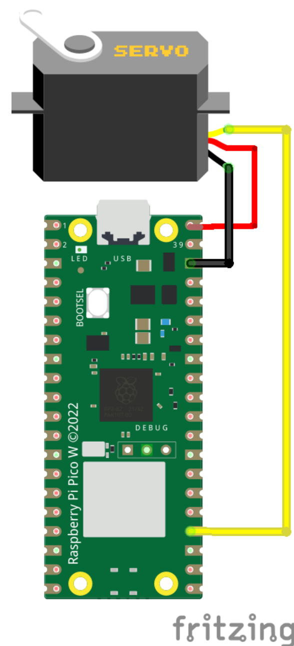


Figure 8.3: Connecting the electronic circuit from example 13.

Open the Thonny editor and follow these steps:

1. Add the necessary libraries:

```
import machine
import utime
```

2. According to the documentation for the servo, the minimum position is when the high signal lasts 0.9 ms (sometimes 0.5 ms). The middle position is when the high signal lasts 1.5 ms and the maximum when 2.1 ms (sometimes 2.5 ms). The PWM signal period should be 20 ms (50 Hz). Let's create variables that will store these values in ns:

```
import machine
import utime

MIN = 900000
MID = 1500000
MAX = 2100000
```

3. Configure the GP18 pin as PWM and set the PWM signal frequency to 50 Hz.

```
import machine
import utime

MIN = 900000
MID = 1500000
MAX = 2100000

pwm = machine.PWM(machine.Pin(18))
pwm.freq(50)
```

4. We will define the servo positions by setting the duty cycle. In the previous chapter, the *duty_u16()* function was presented, which set the duty cycle given in the range from 0 to 65535 (100%). It will be more convenient here to use the *duty_ns()* function, which sets the duty cycle for a specified time in nanoseconds. Let's set the initial deflection to the middle:

```
import machine
import utime

MIN = 900000
MID = 1500000
MAX = 2100000

pwm = machine.PWM(machine.Pin(18))
pwm.freq(50)
pwm.duty_ns(MIN)
```

5. In the main loop, let's make that the servo rotates from the minimum position through the middle to the maximum and back every 1s:

```
import machine
import utime

```

```
MIN = 900000
MID = 1500000
MAX = 2100000

pwm = machine.PWM(machine.Pin(18))
pwm.freq(50)
pwm.duty_ns(MIN)

while True:
    pwm.duty_ns(MIN)
    utime.sleep(1)
    pwm.duty_ns(MID)
    utime.sleep(1)
    pwm.duty_ns(MAX)
    utime.sleep(1)
    pwm.duty_ns(MID)
    utime.sleep(1)
```

The program is ready, test its operation.

8.2 Example 14: Smart ventilation

In this example, we will create a smart ventilation system that will consist of a Pololu DS18B20 temperature sensor (from chapter no. 7.6) and a DC motor connected via a DRV8835 controller to the Raspberry Pi Pico. Depending on the temperature, we will regulate the speed of the DC motor, to which the fan blades can be connected. Let's start by connecting the electronic system according to the table 8.1.

DRV8835	Raspberry Pi Pico	DC motor
GND	GND	-
VIN, VCC and MD	VBUS	-
O1 and O2	-	two legs
VM	-	-
IN1 PH	GP16	-
IN2 EN	GP17	-

Table 8.1: Example connection of DRV8835 driver to Raspberry Pi Pico.

Now open the Thonny editor and follow these steps:

1. Let's start by copying the program for the DS18B20 temperature sensor discussed in chapter 7.6.

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()
```

```
while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

2. DC motor speed is controlled by Pulse Width Modulation (PWM) when using the DRV8835 controller. To do this, let's configure pin 17 as PWM and pin 16 as GPIO output because it will be used to change the direction of rotation:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

DCmotor = machine.PWM(machine.Pin(17))
direction = machine.Pin(16, machine.Pin.OUT)

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

3. Now let's set the PWM signal frequency to 1 kHz and the direction of rotation (value 0 or 1):

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

DCmotor = machine.PWM(machine.Pin(17))
direction = machine.Pin(16, machine.Pin.OUT)

DCmotor.freq(1000)
direction.value(0)

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

4. Now let's create variables in which we will store the PWM signal duty cycle for minimum and maximum speed (RPM - rotations per minute). Remember that the *duty_u16()* function accepts values from 0 to 65535, so the variables should be in these ranges. In addition, let's create variables to store the minimum and maximum temperature for which the ventilation should work. Let's also create a mapping function that will allow us to convert the temperature to the appropriate PWM duty cycle (somewhat proportional to RPM):

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

DCmotor = machine.PWM(machine.Pin(17))
direction = machine.Pin(16, machine.Pin.OUT)

DCmotor.freq(1000)
direction.value(0)

min_speed = 20000
max_speed = 65535
Tmin=22
Tmax=50

def map_value(x, in_min, in_max, out_min, out_max):
    return int((x - in_min) * (out_max - out_min) / (
        ↪ in_max - in_min) + out_min)


while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

5. The last step is to convert the temperature read from the DS18B20 sensor into the PWM signal duty cycle that controls the engine speed so that the higher the temperature, the faster the engine rotates the fan blades. Then we set the designated duty cycle on PWM and correct PWM signal should be generated on PWM pin:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
```

```
device_addr = ds18b20_sensor.scan()

DCmotor = machine.PWM(machine.Pin(17))
direction = machine.Pin(16, machine.Pin.OUT)

DCmotor.freq(1000)
direction.value(0)

min_speed = 20000
max_speed = 65535
Tmin=22
Tmax=50

def map_value(x, in_min, in_max, out_min, out_max):
    return int((x - in_min) * (out_max - out_min) / (
      ↪ in_max - in_min) + out_min)

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))

    speed = map_value(temp, Tmin, Tmax, min_speed,
      ↪ max_speed)
    DCmotor.duty_u16(speed)
```

The program is now ready and you can test it.

9. Wireless communication

Wireless communication can be done in several ways via radio techniques, the most popular are: via Bluetooth, or via Wi-Fi. In this chapter, we will discuss them both. We will also use Internet connection to the cloud service - using the Adafruit IO cloud.

9.1 Example 15: Bluetooth module

In the case of the Raspberry Pi Pico W board, we have a built-in Bluetooth module, which is very well documented in the official datasheet: https://datasheets.raspberrypi.com/picow/connecting-to-the-internet-with-pico-w.pdf . Go to the "Working with Bluetooth in MicroPython" section and there you will find an example "Creating a temperature service peripheral". You can download the finished code from the repository: https://github.com/raspberrypi/pico-micropython-examples/tree/master/bluetooth(picow_ble_temp_sensor.py) and the necessary *ble_advertising.py* library. The library should be loaded onto the Raspberry Pi Pico board and simply run the *picow_ble_temp_sensor.py* example. To read the data, you can use the LightBlue application on a smartphone with Android or iOS.

9.2 Example 16: Adafruit IO

In this example, we will send the temperature reading from the DS18B20 sensor to the Adafruit IO cloud. There are several different clouds, but in this guide we will focus on the cloud that we feel is the simplest and offers a lot of possibilities for creating IoT projects.

The Adafruit IO cloud uses the MQTT (Message Queuing Telemetry Transport) protocol, which is based on the publish/subscribe model. The idea is that client "A" publishes information under a given topic and any other client which subscribed for this topic will read the data published by client "A". This idea is schematically presented in Fig. 9.1.

To create a program that sends temperature to the Adafruit IO cloud, follow these steps:

1. First, you need to create a free account at https://io.adafruit.com.
2. Next, you will need to send temperature value to the cloud. To do this, you need to create one feed. Feed is an object that stores data. To create a feed, go to the "Feed" tab and select the "New Feed" button (see Fig. 9.2).

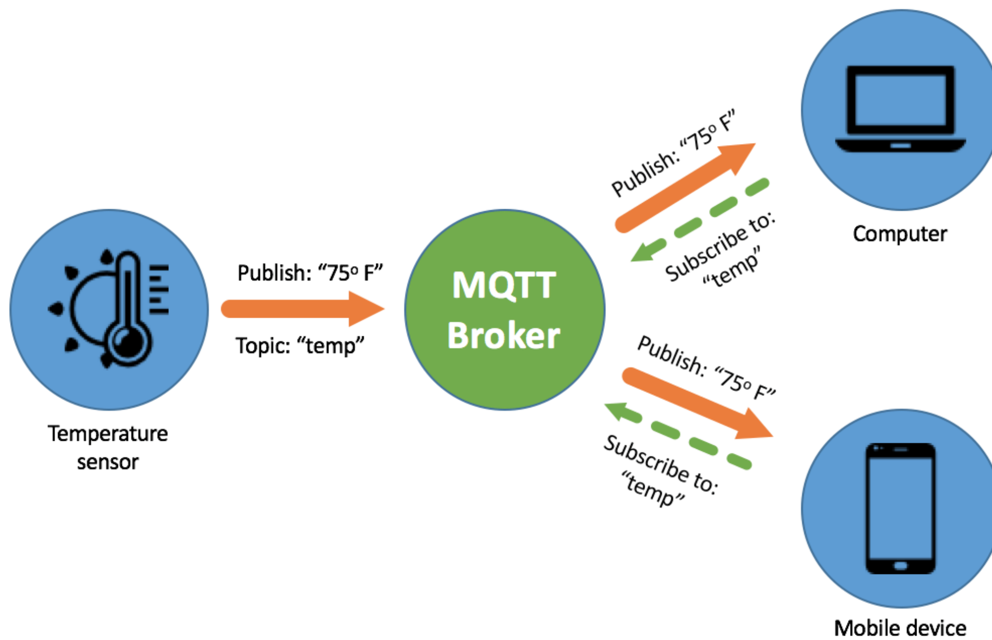


Figure 9.1: How the MQTT protocol works. Source: https://www.akcp.com/blog/scaling-mqtt-network-for-better-operational-output/

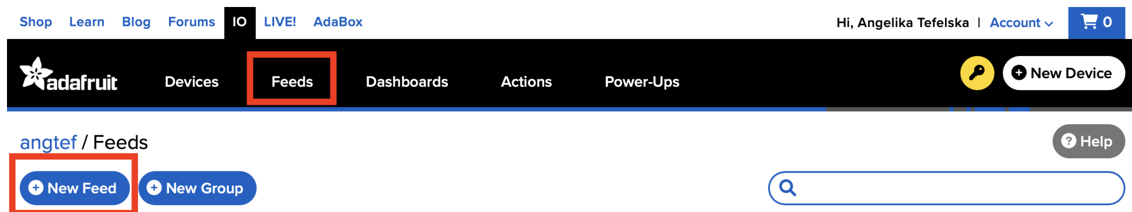


Figure 9.2: Creating a feed in Adafruit IO.

3. Then a window will appear where you need to enter the feed name, e.g. *Temp*.
4. The next step is to create a dashboard. To do this, select the "Dashboards" tab and then "New Dashboard" (see Fig. 9.3).

Figure 9.3: Creating a dashboard in Adafruit IO.

5. A window will pop up where you should enter the selected dashboard name. Then on the right side, select the settings symbol and choose "Create New Block" (see Fig. 9.4).
6. Adafruit IO has various blocks available for displaying data (text boxes, gauges, charts, maps, etc.). In our case, let’s choose a gauge (see Fig. 9.5).

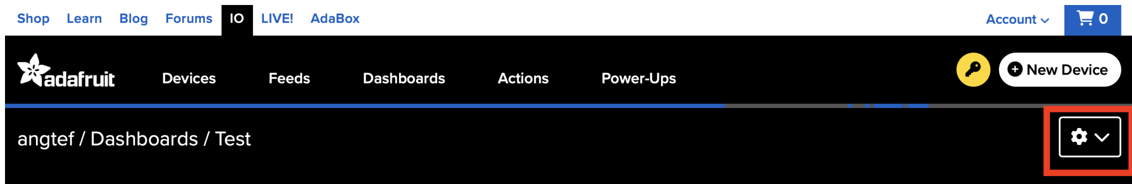


Figure 9.4: Editing a dashboard in Adafruit IO.

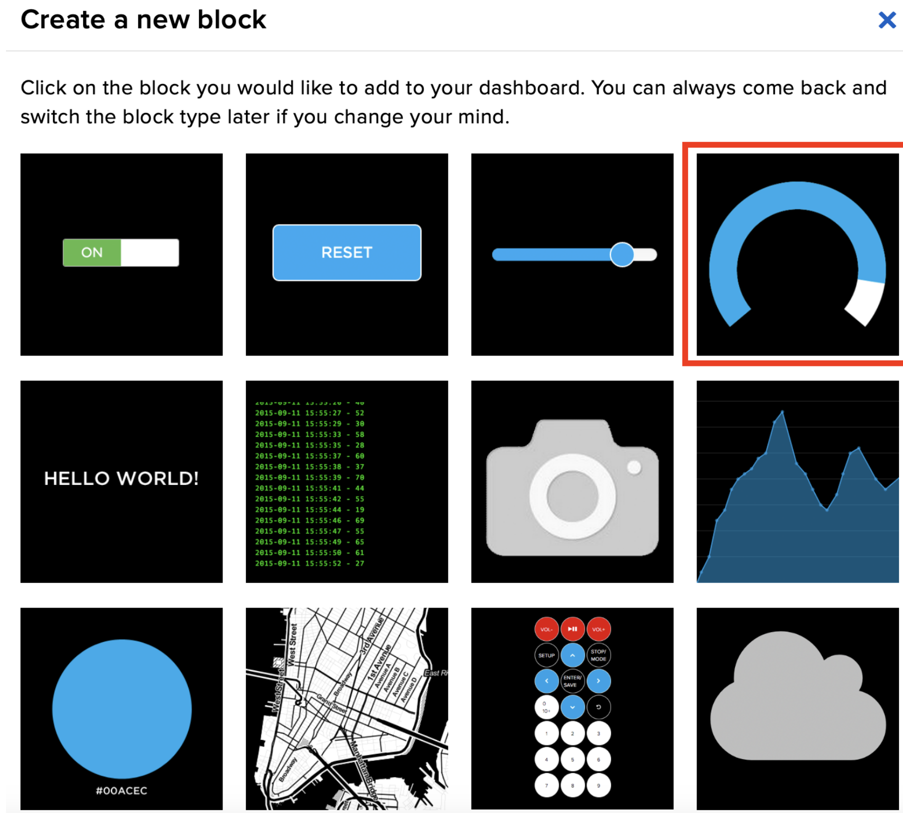


Figure 9.5: Available blocks in Adafruit IO.

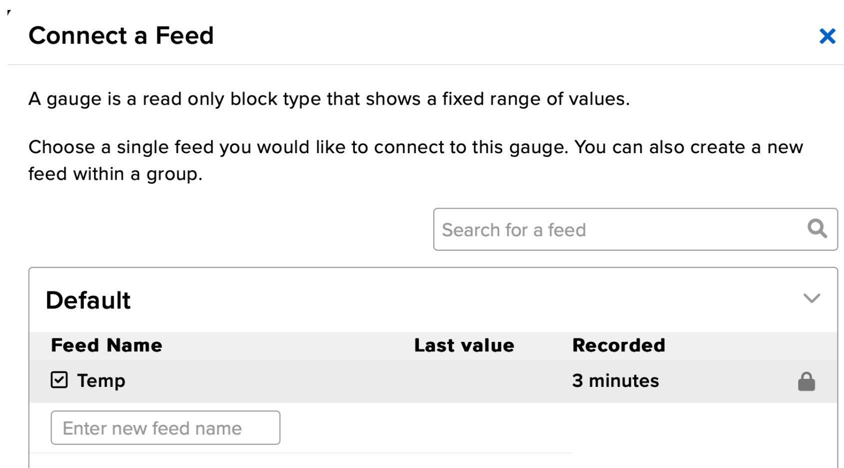


Figure 9.6: Connecting the block to the feed. in Adafruit IO.

7. Then a window will pop up asking which feed we want to connect the block to. Select the feed you created to store the temperature values (see Fig. 9.6).
8. Now let's go to the Thonny editor and install the *umqtt.simple* library from the packages manager in the Thonny editor.
9. Now let's return to the example of reading temperature from the DS18B20 sensor:

```
import machine
import onewire
import ds18x20
import utime

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

10. Next, add the necessary libraries:

```
import machine
import onewire
import ds18x20
import utime
import network
from umqtt.simple import MQTTClient

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

11. Add a piece of code that will allow you to connect to your Wi-Fi. Here you need to fill sensitive data in lines 12-13. First, enter the name of your Wi-Fi (SSID - Service Set Identifier) and then the password, so the Raspberry Pi Pico W can connect to your Wi-Fi network. This Wi-Fi should allow connection to the Internet, because we want to connect with Adafruit IO Cloud service:

```
import machine
import onewire
import ds18x20
import utime
import network
from umqtt.simple import MQTTClient
```

```

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

WIFI_SSID = "your_wifi_name"
WIFI_PASSWORD = "your_wifi_password"
ADAFRUIT_IO_USERNAME = "username"
ADAFRUIT_IO_KEY = "key"

def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(WIFI_SSID, WIFI_PASSWORD)
    while not wlan.isconnected():
        print("Connecting to Wi-Fi...")
        utime.sleep(1)
     print("Connected to Wi-Fi:", wlan.ifconfig())

connect_wifi()

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

12. The last two parameters are credentials for access to the Adafruit IO cloud. Go back to the Adafruit website and click the key symbol. When you do this, your Username and Key will appear (see Fig. 9.7). Copy this data to the program to lines 14-15.

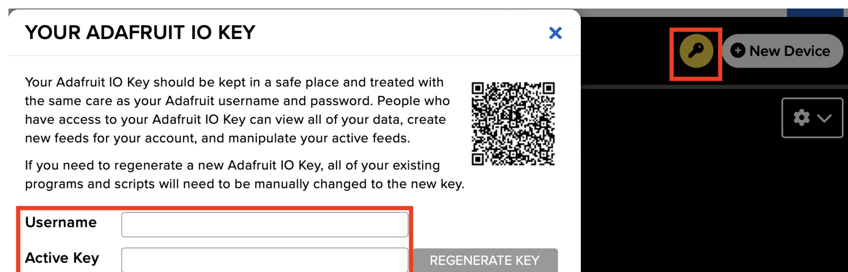


Figure 9.7: Username and key.

13. Now we will use MQTT protocol. Use the code below, changing only the feed name in lines 30:

```
import machine
import onewire
import ds18x20
import utime
import network
from umqtt.simple import MQTTClient

```

```
one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

WIFI_SSID = "your_wifi_name"
WIFI_PASSWORD = "your_wifi_password"
ADAFRUIT_IO_USERNAME = "username"
ADAFRUIT_IO_KEY = "key"

def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(WIFI_SSID, WIFI_PASSWORD)
    while not wlan.isconnected():
        print("Connecting to Wi-Fi...")
        utime.sleep(1)
    print("Connected to Wi-Fi:", wlan.ifconfig())

ADAFRUIT_IO_SERVER = "io.adafruit.com"
ADAFRUIT_IO_PORT = 1883  # Port MQTT
CLIENT_ID = "raspberry_pi_pico"

FEED_TEMP_LEVEL = "username/feeds/Temp"

client = MQTTClient(CLIENT_ID, ADAFRUIT_IO_SERVER,
    ↪ ADAFRUIT_IO_PORT, ADAFRUIT_IO_USERNAME,
    ↪ ADAFRUIT_IO_KEY)

def connect_mqtt():
    client.connect()
    print("Connected to Adafruit IO")

connect_wifi()
connect_mqtt()

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

14. Now let's add function to send data to the cloud:

```
import machine
import onewire
import ds18x20
import utime
import network
from umqtt.simple import MQTTClient

one_wire_bus = onewire.OneWire(machine.Pin(13))
```

```
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

WIFI_SSID = "your_wifi_name"
WIFI_PASSWORD = "your_wifi_password"
ADAFRUIT_IO_USERNAME = "username"
ADAFRUIT_IO_KEY = "key"

def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(WIFI_SSID, WIFI_PASSWORD)
    while not wlan.isconnected():
        print("Connecting to Wi-Fi...")
        utime.sleep(1)
    print("Connected to Wi-Fi:", wlan.ifconfig())

ADAFRUIT_IO_SERVER = "io.adafruit.com"
ADAFRUIT_IO_PORT = 1883  # Port MQTT
CLIENT_ID = "raspberry_pi_pico"

FEED_TEMP_LEVEL = "username/feeds/Temp"

client = MQTTClient(CLIENT_ID, ADAFRUIT_IO_SERVER,
    ADAFRUIT_IO_PORT, ADAFRUIT_IO_USERNAME,
    ADAFRUIT_IO_KEY)

def connect_mqtt():
    client.connect()
    print("Connected to Adafruit IO")

connect_wifi()
connect_mqtt()

def send_data(temp):
    client.publish(FEED_TEMP_LEVEL, str(temp))
    print("Temp. sent:"+str(temp))

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
```

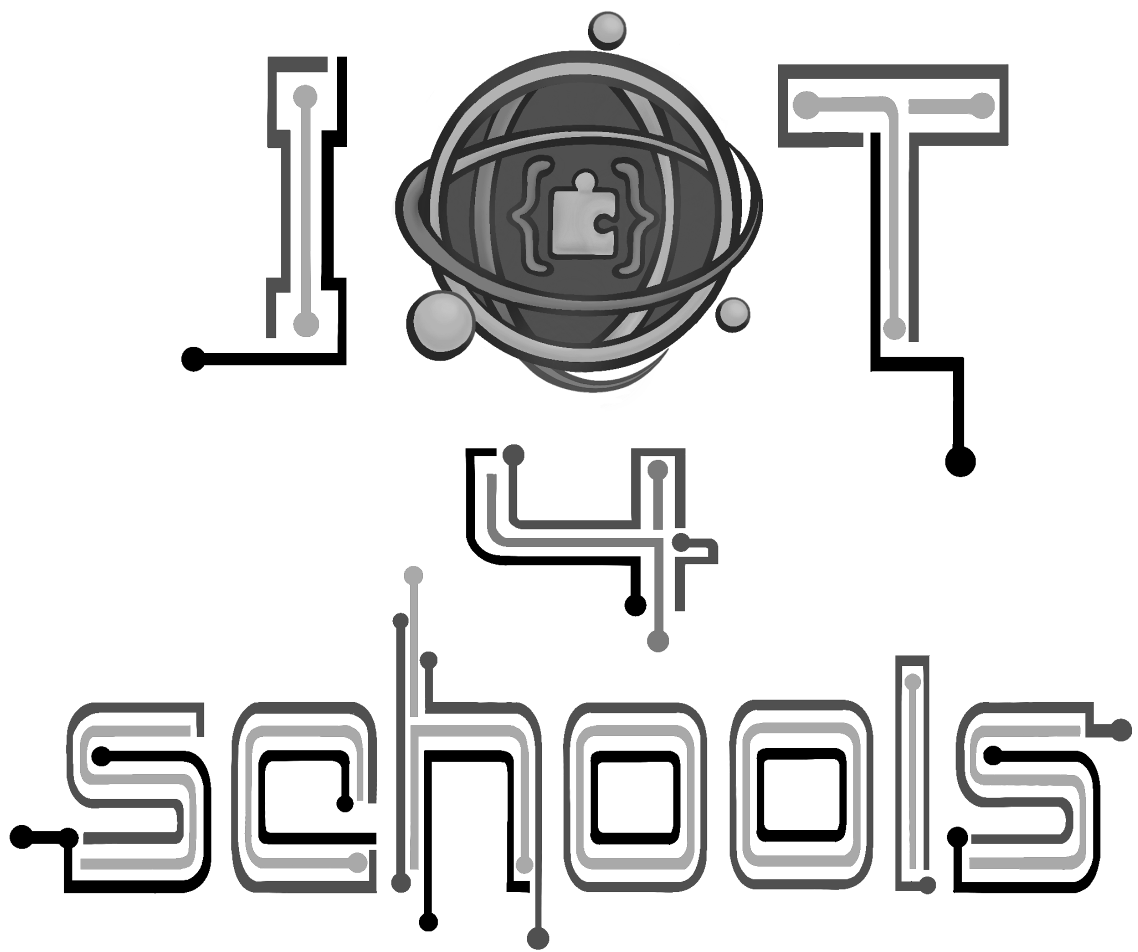
15. The last step is to call the function to send data after measuring the temperature and sending its value to the cloud:

```
import machine
import onewire
import ds18x20
import utime
```

```
import network
from umqtt.simple import MQTTClient

one_wire_bus = onewire.OneWire(machine.Pin(13))
ds18b20_sensor = ds18x20.DS18X20(one_wire_bus)
device_addr = ds18b20_sensor.scan()

WIFI_SSID = "your_wifi_name"
WIFI_PASSWORD = "your_wifi_password"
ADAFRUIT_IO_USERNAME = "username"
ADAFRUIT_IO_KEY = "key"

def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(WIFI_SSID, WIFI_PASSWORD)
    while not wlan.isconnected():
        print("Connecting to Wi-Fi...")
        utime.sleep(1)
    print("Connected to Wi-Fi:", wlan.ifconfig())

ADAFRUIT_IO_SERVER = "io.adafruit.com"
ADAFRUIT_IO_PORT = 1883  # Port MQTT
CLIENT_ID = "raspberry_pi_pico"
FEED_TEMP_LEVEL = "username/feeds/Temp"

client = MQTTClient(CLIENT_ID, ADAFRUIT_IO_SERVER,
    ↪ ADAFRUIT_IO_PORT, ADAFRUIT_IO_USERNAME,
    ↪ ADAFRUIT_IO_KEY)

def connect_mqtt():
    client.connect()
    print("Connected to Adafruit IO")

connect_wifi()
connect_mqtt()

def send_data(temp):
    client.publish(FEED_TEMP_LEVEL, str(temp))
    print("Temp. sent:"+str(temp))

while True:
    ds18b20_sensor.convert_temp()
    utime.sleep(0.75)
    temp=ds18b20_sensor.read_temp(device_addr[0])
    print("T="+str(temp))
    send_data(temp)
```

The program is ready. Run it and go to Adafruit IO to the created dashboard. You should see the measured temperature on the indicator.

This was the last example in this guide. We hope that the presented examples allowed you to get to know the Raspberry Pi Pico W board. We encourage you to continue developing and looking for interesting examples of using the Raspberry Pi Pico W board both within the IoT4schools project (https://www.iot.fizyka.pw.edu.pl) and elsewhere.


IoT
4
schools