PROJECT NUMBER: 2023-1-PL01-KA220-SCH-000154043

Bringing the Internet of Things in school education as a tool to address 21st century challenges

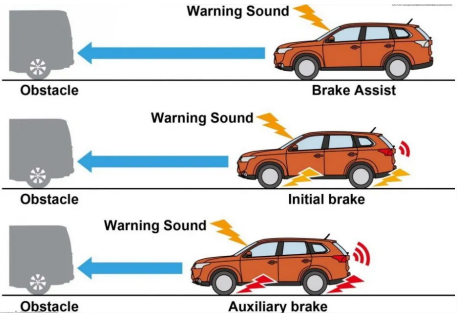
"Smart car: how to reduce the number of car accidents?"

Co-funded by the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Education and Culture Executive Agency (EACEA). Neither the European Union nor EACEA can be held responsible for them.

Project objective

- The aim of this project is to familiarize students with the issue of road safety.
- The scenario will involve developing a line-follower vehicle that will adjust its speed depending on the distance from an obstacle, such as another vehicle traveling in front of it or a pedestrian who suddenly enters the road.
- This is particularly important in preventing pedestrians from being hit. Statistics show that approximately 17% of all people killed in road accidents in the EU are pedestrians.
- The WHO, in turn, indicates that approximately 20-50 million injuries occur annually as a result of various road accidents.



Source of image: https://miro.medium.com/v2/resize:fit:1400/format:webp/1*nRxRkYqTTMbZeoSUknzDlg.jpeg

- Furthermore, the scenario will develop a system that measures traffic volume on a given road by counting the number of passing cars.

Educational goals

Through to this project, students will be able to:

- Construct a robot using available components (chassis, controllers, motors).
- Create a advanced program in the MicroPython programming language.
- Use external libraries from the GitHub repository.
- Create a program to control a line follower robot and familiarize yourself with robot control algorithms, including the Proportional Integral Derivative (PID) controller.
- Use the *Adafruit IO* cloud platform and send data to it.

In addition, students:

- Identify the benefits of using systems to prevent car accidents and ways to prevent head-on collisions.
- Identify ways to measure traffic flow.
- Improve cooperation skills and break down cross-cultural barriers.

Project Content

- Two warm-up exercises:
  - using DC motors (controlling the speed and direction of rotation of the wheels),
  - using a the tracker sensor (reading data from 3 sensors).
- Level 1: creating a line follower based on an external library from the github repository.
- Level 2: avoiding head-on collisions with other vehicles.
- Level 3: measuring the number of vehicles passing over the bridge.

Educational path

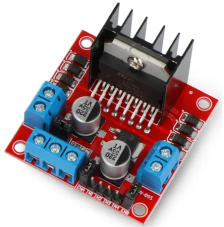
1. **Group formation:** Divide your students into teams of two or three.
2. **Brainstorming:** Encourage students to search online for information about car accident statistics and their main causes.
3. **Discussion and assignment of the activity:** Encourage each team to present their data. Together, explore ways to reduce car accidents using technology.
4. **Planning:** Encourage each team to think about how they will construct the car (where the line sensor and the ultrasonic distance sensor should be placed).
5. **Creation:** Using the student worksheets, encourage each team to create their own smart car.
6. **Testing - optimization:** After completing the project, encourage your students to test their smart car. Based on the test results, you can encourage each team to optimize their project.
7. **Presentation - sharing:** Encourage your students to present their projects in plenary and ask them to reflect on the whole experience. Encourage all teams to consider the impact of such smart car on the road safety, the functioning of cities and the environment.

Necessary equipment

- Raspberry Pi Pico W board,
- breadboard (contact plate) with 400 holes (no more),
- Female-Male and Male-Male connection wires,
- ultrasonic distance sensor HC-SR04,
- Velleman VMA500 2WD - 2-Wheel Robot Chassis with DC Motor Drive. Now also available under a new name: Whadda WPK500. *You can also use a different chassis, but then you'll need to plan the layout of the components yourself.*
- L298N - two-channel motor controller - 12V/2A,
- mounting bracket for distance sensor HC-SR04,
- tracker Sensor, Infrared Line Tracking, Waveshare 12223,

Source: https://kamami.pl/.

Source: https://www.velleman.eu.



Source: https://botland.com.pl/.

Necessary equipment

- External power source: Powerbank with 5V output,
- IR beam interruption sensor - LED 5mm - 0-100cm or smaller range.
- Screw M3, 8 mm - 8 pieces
- Screw M3, at least 14mm max 30mm - 2 pieces
- M3 nut - 4 pieces
- Spacer sleeve M3, 30 mm - 4 pieces
- Screwdriver
- Double-sided tape
- Line follower route mat (black line on white background)

Software:

- Thonny editor
- Adafruit IO cloud

Source: https://kamami.pl/.

Source: https://botland.store.

Time plan

Estimated time of project implementation in class:

- 45 minutes to introduce into the project (including brainstorming and discussion) and planning,
- 135 minutes to complete warm-up activities and level 1,
- 45 minutes for level 2,
- 90 minutes for level 3,
- 45 minutes for wrap-up and discussion,

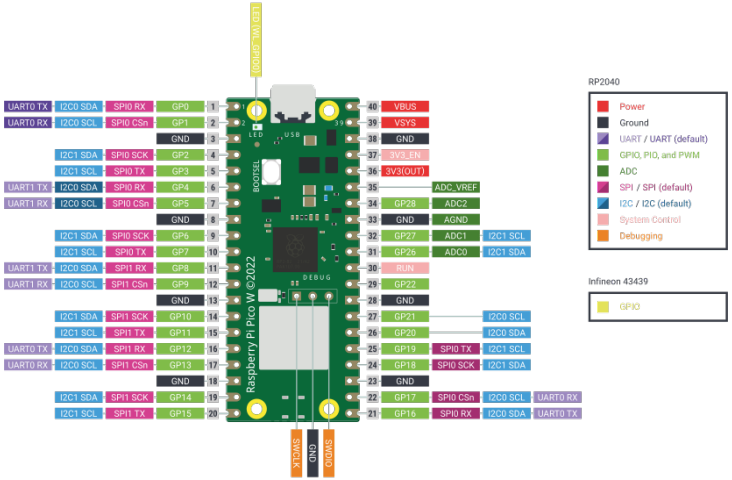
Connection

Follow the steps as instructed in the Teacher's Guide.

Warm-up exercises

Warm-up exercise no. 1 - board layout

- *General Purpose Input/Output* (GPIO) are pins that are used to generate or read digital signals, meaning that they have only two possible values: 0 or 1.
- Pins that perform the GPIO function are abbreviated GPx (green rectangles), where x is the GPIO pin number, starting from 0 to 28.

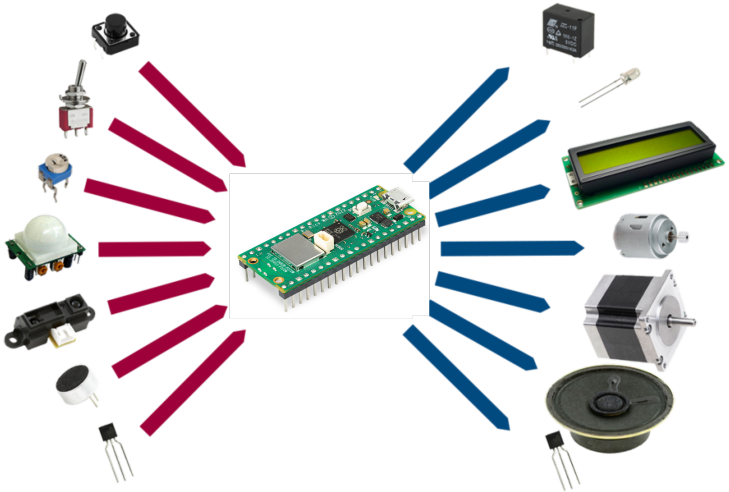


Source: https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html

Warm-up exercise no. 1 - pin configuration

The pins to which the components are connected can be:

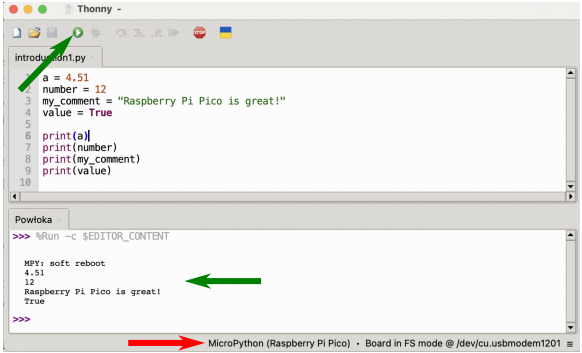
- *input*,
- *output*.



Source of original image: https://blog.codebender.cc/2014/03/07/lesson-1-inputs-and-outputs/

Warm-up exercise no. 1 - Thonny editor

- The goal of the first exercise is to understand how to control DC motors to make the wheels rotate in a specified direction and at a selected speed.
- Exercise 1 is divided into versions for older and younger school students.
- Below we present the solution for older school students.
- Versions for younger students can be found in the teacher's guide for the project.
- Before we begin, open the Thonny editor. Then, select the MicroPython language in the lower right corner of the screen.



Digression: The general structure of each program

- The *while* loop is a type of loop that will repeat anything we put in it until a condition is met. The while loop is created as follows:

```
while condition:
        commands
```

- In the case of microcontroller programming, an infinite loop is created, i.e. a loop that will execute all the time until the board's power is turned off. Hence the value *True* is given as the condition.

```
#Adding necessary libraries that contain useful functions
import library_name

#Creating variables
#Execution of commands that are to be executed only once

#Infinite loop
while True:
        #Commands that are to be executed continuously
```

Warm-up exercise no. 1

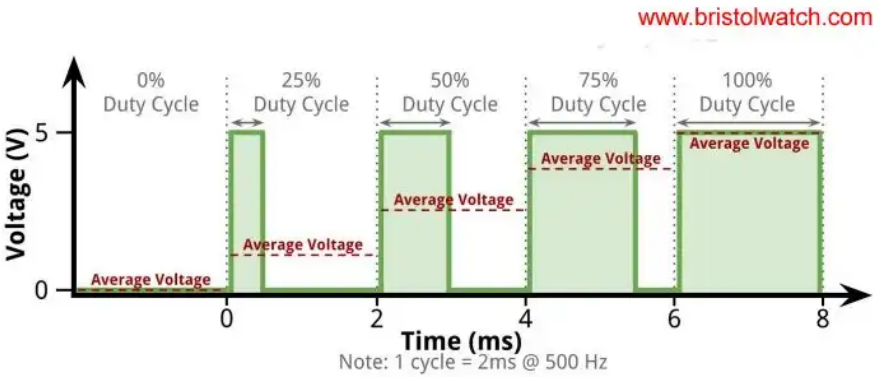
1. First, let's add the necessary libraries, i.e. *machine* and *utime*:

```
import machine
import utime
```

2. Next, let's configure the pins for the right motor. The controller has two types of outputs: *Input 1* and *Input 2* for direction control, and *Enable* for speed control. *Input 1* and 2 are connected to digital outputs GP19 and GP20, and the *Enable* pin is connected to output GP21, which we will control using PWM:

```
import machine
import utime

#right motor
ENA = machine.PWM(machine.Pin(21))
IN1 = machine.Pin(20, machine.Pin.OUT)
IN2 = machine.Pin(19, machine.Pin.OUT)
```

PWM



Average output voltage is proportional to duty cycle ON time.

Warm-up exercise no. 1

3. Let's configure the pins for the left motor similarly. In this case, *Input 3* and *Input 4* are connected to digital outputs GP17 and GP18, and the *Enable* pin is connected to digital output GP16:

```
import machine
import utime

ENA = machine.PWM(machine.Pin(21))  #right motor
IN1 = machine.Pin(20, machine.Pin.OUT)
IN2 = machine.Pin(19, machine.Pin.OUT)

ENB = machine.PWM(machine.Pin(16))  #left motor
IN3 = machine.Pin(18, machine.Pin.OUT)
IN4 = machine.Pin(17, machine.Pin.OUT)
```

Warm-up exercise no. 1

4. Now let's set the signal frequency on the PWM pins to 1kHz.

```
1  import machine
2  import utime
3
4  ENA = machine.PWM(machine.Pin(21))  #right motor
5  IN1 = machine.Pin(20, machine.Pin.OUT)
6  IN2 = machine.Pin(19, machine.Pin.OUT)
7
8  ENB = machine.PWM(machine.Pin(16))  #left motor
9  IN3 = machine.Pin(18, machine.Pin.OUT)
10 IN4 = machine.Pin(17, machine.Pin.OUT)
11
12 ENA.freq(1000)
13 ENB.freq(1000)
```

Warm-up exercise no. 1

5. To make the wheel move forward or backward, set the values according to the table (Note: *The "..." sign means that part of the code was omitted because it wouldn't fit on the slide. Do not delete the omitted code.*):

```
1  import machine
2  ...
3  ENB.freq(1000)
4
5  def drive(left, right, direction, t=1.0):
6      if right >= 0:
7          IN1.value(1) #forward
8          IN2.value(0)
9      else:
10         IN1.value(0) #backwards
11         IN2.value(1)
```

Right motor			
Direction	Input 1	Input 2	Enable
Forward	1	0	1
Backwards	0	1	1
Stop	0	0	0
Left motor			
Direction	Input 3	Input 4	Enable
Forward	1	0	1
Backwards	0	1	1
Stop	0	0	0

Warm-up exercise no. 1

```
    ...
    else:
        IN1.value(0) #backwards
        IN2.value(1)

     if left >= 0:
         IN3.value(1) #forward
         IN4.value(0)
    else:
        IN3.value(0)  #backwards
        IN4.value(1)

    #Sets the DC motor speed. Value ranges from 0 to 65535
    ENA.duty_u16(min(65535, abs(int(right)))) #The min function was used to
        ↪ensure that the value sent to the motor was in the range.
    ENB.duty_u16(min(65535, abs(int(left))))
    utime.sleep(t) #the time the motor will move in the indicated direction
```

Warm-up exercise no. 1

6. It's possible that a student connected the wires backwards, meaning the red wire was connected to *Input 2* instead of *Input 1*. In this case, the motor would spin backward instead of forward. Therefore, let's add a *direction* variable to the code, which will programmatically change the direction:

```
...
ENB.freq(1000)

direction = 1 #Change to -1 if the vehicle is moving backward instead of
    ↪forward. Remember that the front is where the line sensors and
    ↪ultrasonic sensor are.

def drive(left, right, direction, t=1.0):
    left=direction*left
    right=direction*right
    if right >= 0:
        IN1.value(1) #forward
...
```

Warm-up exercise no. 1

7. Now let's add functions to stop the motors:

```
import machine
...

def drive(left, right, direction, t=1.0):
    ...

def stop():
    ENA.duty_u16(0)
    ENB.duty_u16(0)
```

Warm-up exercise no. 1

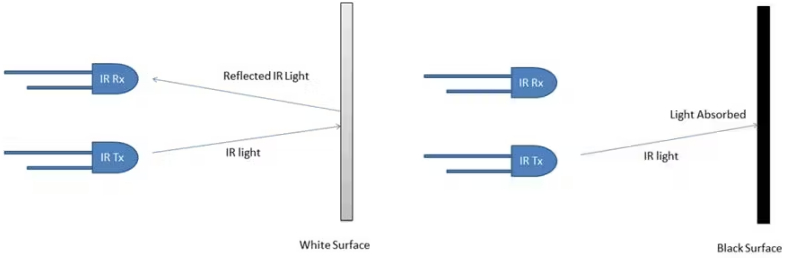
8. The last step is to call the drive function with various parameters to check whether the vehicle moves forward, turns left, turns right and moves backward:

```
1  import machine
2  ...
3
4  def stop():
5      ENA.duty_u16(0)
6      ENB.duty_u16(0)
7
8  drive(40000, 0, 2)          #only left motor - forward
9  drive(0, 40000, 2)          #only right motor - forward
10 drive(50000, 50000, 2)      #both motors - forward
11 drive(-35000, -35000, 2)    #both motors - backwards
12 stop()
```

Run the program and test its operation. Remember that the front of the vehicle is where the ultrasonic distance sensor is mounted. If the direction doesn't match, change the *direction* variable to -1.

Warm-up exercise no. 2

- The next exercise is to test the operation of the tracker sensor, which returns a voltage proportional to the reflected light intensity.
- First, light is emitted by an infrared diode and hits the surface. If the surface is white, almost all of the light is reflected and returned to the sensor. This will result in a near-maximum reading on the analog pin, close to 65535.
- When the light hits the black line, it is partially absorbed, and the reflected beam is significantly weaker. This reading is significantly lower than 65535.



Source: https://www.hackster.io.

Warm-up exercise no. 2

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

```
import machine
import utime
```

2. Next, let's configure the analog pins to which the IR2, IR3, and IR4 sensors are connected. The tracker sensor has five sensors: IR1, IR2, IR3, IR4, and IR5, but we won't be using the two outermost ones:

```
import machine
import utime

IR_LEFT   = machine.ADC(26)   # Analog pin configuration for IR2
IR_CENTER = machine.ADC(27)   # Analog pin configuration for IR3
IR_RIGHT  = machine.ADC(28)   # Analog pin configuration for IR4
```

Warm-up exercise no. 2

3. In the main loop, read the values from the analog pins and display the values in the terminal:

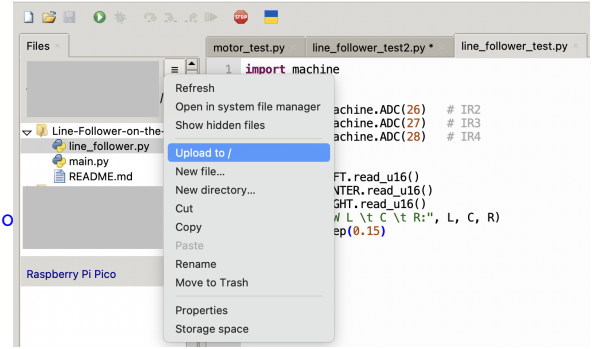
```
import machine
import utime

IR_LEFT   = machine.ADC(26)
IR_CENTER = machine.ADC(27)
IR_RIGHT  = machine.ADC(28)

while True:
    L = IR_LEFT.read_u16() #Reading the value from sensor
    C = IR_CENTER.read_u16()
    R = IR_RIGHT.read_u16()
    print("RAW L \t C \t R:", L, C, R)
    utime.sleep(0.15)
```

Test the program by moving the vehicle over the black line, alternating between the sensors being over the line and the white background. You should see a high value over the white background and a low value over the black background. If any of the sensors have unchanged values, it most likely means they are connected incorrectly.

Level 1

Level 1

- In the first step, we'll create a line follower vehicle. To do this, we'll need an external library that contains a number of useful functions for controlling the vehicle.
- Let's start by downloading it. Visit the project's GitHub: https://github.com/angsam/Line-Follower-on-the-Raspberry-Pi-Pico select the green *Code* button, and then click *Download Zip*.
- After unzipping the library, go to the Thonny editor and select *View*→*Files* from the top menu.



- Then, locate the downloaded library on the right, select the three-line icon, and then *Upload to*.
- Upload the *line_follower.py* and *main.py* files to the Raspberry Pi Pico W board in this way.

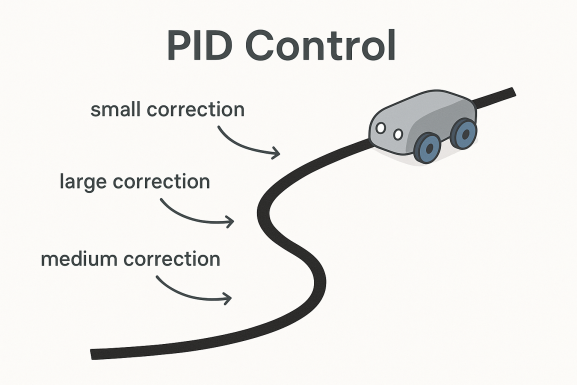
Level 1

Next, open the *main.py* and analyze its operation. Generally, the line follower algorithm consists of the following steps:

- Configuring the motors and tracker sensor.
- Loading the default parameters.
- Calibrating the tracker sensor, i.e., setting the minimum and maximum values the sensor reads to recognize the black line. Calibration time can be extended by changing the value of the ms variable.
- In the main loop, we read the values from the tracker sensor.
- Next, we use the PID procedure (see next slide).
- Slowing down in curves and accelerating on a straight road.
- Finding your way when lost.
- Setting the direction and speed

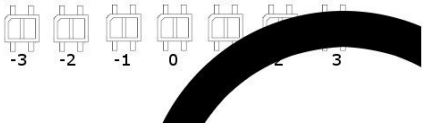
PID procedure

- PID (*Proportional-Integral-Derivative*) is a control method that calculates how much the robot's current position should be corrected to reach the desired location in the next step.
- In the case of a line follower robot, this means calculating the appropriate steering angle to keep the vehicle on the line.
- If the line is turning left, the PID controller calculates how much the wheels should be turned to keep the robot on the track. Depending on the line's curvature, the turn can be gentle or sharp - it is the PID controller that allows for precise selection of the appropriate value.



PID procedure

- The PID controller consists of three parts (terms):
  1. Proportional Term (P):
     - Responsible for the current position error relative to the line's center.
     - You can imagine it like this: the robot has an array of sensors (e.g., 6), each of which has a weight assigned to it, depending on its distance from the center.
     - Example: If the line is under the two rightmost sensors, we add their weights, e.g., 2+3=5, and then divide by the number of active sensors: 5/2=2.5. The resulting result is the position error.



Source: https://forbot.pl

- Based on this, we calculate the proportional correction:

$$correction = P \cdot error \tag{1}$$

- The larger the error, the stronger the correction.

2. Integral term (I):
   - This takes into account the sum of past errors - that is, the accumulation of situations where the robot deviated from the route in one direction.
   - If the robot hasn't turned sharply enough for several steps, the integral term will force a larger correction to restore its position:

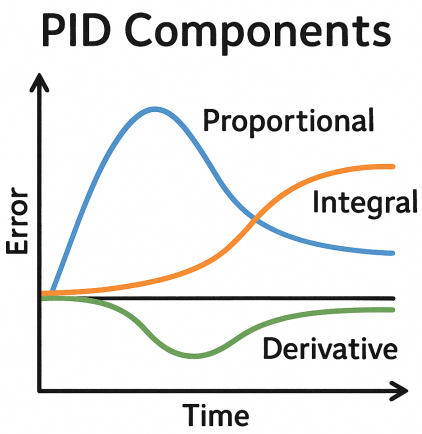
$$Correction = P \cdot error + I \cdot \sum error \tag{2}$$

   - This allows the controller to also respond to long-term deviations.
3. Derivative Term (D):
   - Calculates the difference between the current and previous error.
   - Its function is to dampen oscillations - it stabilizes the robot's motion as it approaches the center of the line (i.e., as the error decreases).
   - Final formula:

$$Correction = P \cdot error + I \cdot \sum error + D \cdot (error - last\ error) \tag{3}$$

PID procedure

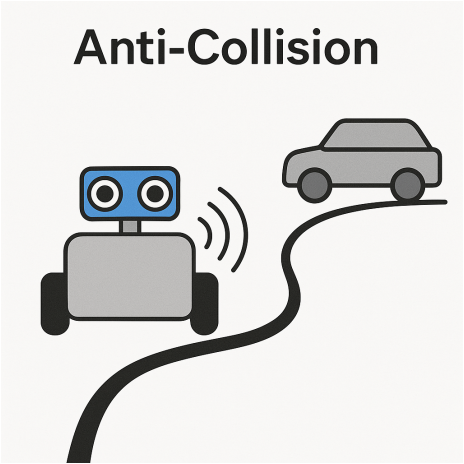
- Why is PID necessary?
- Without a PID controller, the robot would react too quickly or too slowly, causing it to oscillate around or deviate from the line.
- Thanks to properly selected P, I, and D coefficients, the line follower can follow the route smoothly and stably, even with sharp curves or uneven lighting.
- Too complicated? Don't worry, there's a ready-made function in the library that implements the PID procedure.



Time to test!

Level 2

- Now let's move on to frontal collision avoidance.
- For this purpose, we'll use an ultrasonic distance sensor.
- The distance sensor will measure the distance to an obstacle, and if the obstacle is too close, the vehicle will slow down.



Level2

1. First, add the ultrasonic distance sensor configuration:

```
import time
import line_follower as lf

#Motors
motor = lf.motors_init(ena_pin=21, in1_pin=20, in2_pin=19,
    ↪enb_pin=16, in3_pin=18, in4_pin=17, drive_dir=1)
line_sensors = lf.sensors_init(left=26, center=27, right=28,
    ↪alpha=0.45, flip_lr=False)

ultrasonic_sensor = lf.ultrasound_init(trig_pin=14, echo_pin=15)

#Parameters
p = lf.params_default()
state = {'last_seen_ms': time.ticks_ms(), 'last_dist_ts': 0, '
    ↪last_dist': None}
```

Level2

2. Create auxiliary variables: *last_measure* (time of the last measurement), *stop_cm* (distance at which the vehicle should stop), *slow_cm* (distance at which the vehicle should start slowing), and *speed_factor*:

```
import time
import line_follower as lf
...
pid = lf.pid_init(Kp=p['Kp'], Ki=p['Ki'], Kd=p['Kd'], I_max=p['I_MAX'])

last_measure = 0
stop_cm = 12
slow_cm = 30
speed_factor=1

while True:
    now = time.ticks_ms()
    ...
```

3. If 60 seconds have passed since the last measurement, we will take a new measurement using the ultrasonic distance sensor. To do this, call the *ultrasound_measure_cm* function, which returns the distance in cm from the obstacle, or *None* if no obstacle was detected. We also store the current time in the *last_measure* variable.

```
import time
...
while True:
    ...
    # 3) Speed profile (curve or straight line)
    base_now = lf.base_speed_for(err, p['BASE_SPEED'], p['TURN_SLOWDOWN'],
        ↪ p['TURN_BETA'])

    # 4) Anti-collision + turbo (separate step)
    if time.ticks_diff(now, last_measure) >= 60:
        dist_cm = lf.ultrasound_measure_cm(ultrasonic_sensor)
        last_measure = now
```

Level2

4. Next, set the *speed_factor* value based on the distance from the obstacle.

```
import time
...
while True:
    ...
    # 4) Anti-collision + turbo (separate step)
    if time.ticks_diff(now, last_measure) >= 60:
        dist_cm = lf.ultrasound_measure_cm(ultrasonic_sensor)
        last_measure = now
    if dist_cm is None:
        speed\_factor = 1.0
    elif dist_cm <= stop_cm:
        speed_factor = 0
    elif dist_cm < slow_cm:
        speed_factor = (dist_cm - stop_cm) / (slow_cm - stop_cm)
    else:
        speed_factor = 1.0
```

Level2

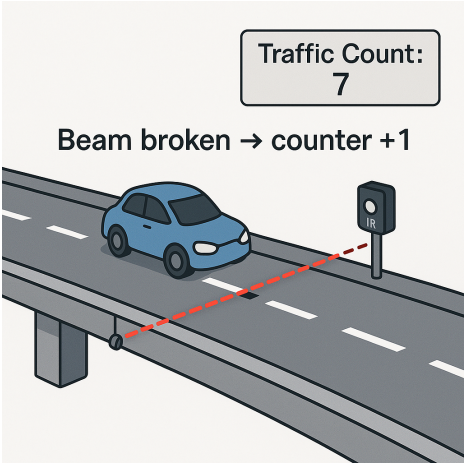
5. In the next step, scale the current speed using *speed_factor* and set the new speed.

```
import time
...
while True:
    ...
    if dist_cm is None:
        speed\_factor = 1.0
    elif dist_cm <= stop_cm:
        speed_factor = 0
    elif dist_cm < slow_cm:
        speed_factor = (dist_cm - stop_cm) / (slow_cm - stop_cm)
    else:
        speed_factor = 1.0
    base_now = int(base_now * speed_factor)
    base_now = lf.straight_turbo(base_now, err, p['STRAIGHT_EPS'], p['
        ↪STRAIGHT_BOOST'], speed_factor)
```

Test the vehicle by placing various obstacles in front of it.

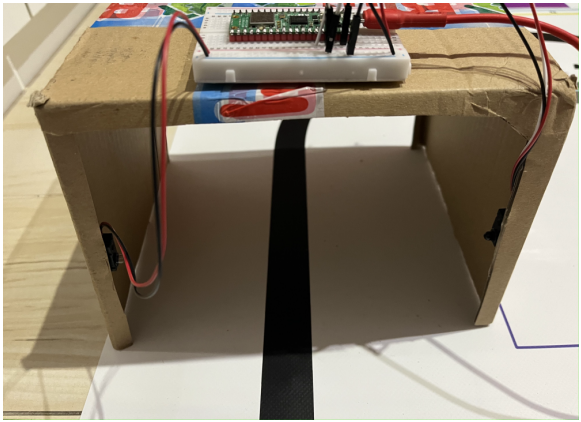
Level 3

- In the third level, we'll use an infrared transmitter and receiver circuit that will count passing vehicles, which will interrupt the beam between the transmitter and receiver.
- To do this, connect the circuit:
  - the red wires from the transmitter and receiver to 3.3V,
  - the black wires from the transmitter and receiver to GND,
  - the white/yellow wire from the receiver to a selected digital pin, e.g., GP28.



Level 3 - time for crafting

Prepare the bridge and install an IR sensor on it. Remember to make the bridge large enough so that vehicles won't stop in front of it, considering it an obstacle.



Level 3

Once you have your bridge ready, open the Thonny editor and follow these steps:

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

```
import machine
import utime
```

2. Configure the pin to which the IR receiver is connected as an input and pull up the *PULL_UP* resistors.

```
import machine
import utime

sensor = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_UP)
```

Level 3

3. Create auxiliary variables: *sensorState* (current state), *lastState* (previous state), and *counter* (the number of vehicles that have crossed the bridge).

```
1 import machine
2 import utime
3
4 sensor = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_UP)
5
6 sensorState = 0
7 lastState=1
8 counter = 0
```

Level 3

4. In the main loop, read the value returned by the IR receiver. If it doesn't detect a vehicle, we read 1, and if a vehicle crosses and interrupts the beam, we read 0.

```
import machine
import utime

sensor = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_UP)

sensorState = 0
lastState=1
counter = 0

while True:
    sensorState = sensor.value()
    print(sensorState)
```

Level 3

5. If the current state is 1 and the previous state is 0, it means the car has crossed the bridge, and you can increment the counter by one. Finally, add a delay, which you can adjust to your needs.

```
import machine
import utime
...
while True:
    sensorState = sensor.value()
    print(sensorState)

    if sensorState==1 and lastState==0:
        counter+=1
        print("Counter="+str(counter))

    lastState=sensorState
    utime.sleep(1)
```

Level 3

6. Test the system. If everything works correctly, proceed to the next step, which involves sending data about the number of vehicles to the Adafruit IO cloud.
7. First, you need to create a free account at `https://io.adafruit.com` . Next, you will want to send counter to the cloud. To do this, you need to create feed. Feeds are objects that store data. To create a feed, go to the "Feed" tab and select the "New Feed" button.

Then a window will appear where you need to enter the feed name, e.g. *Counter*.

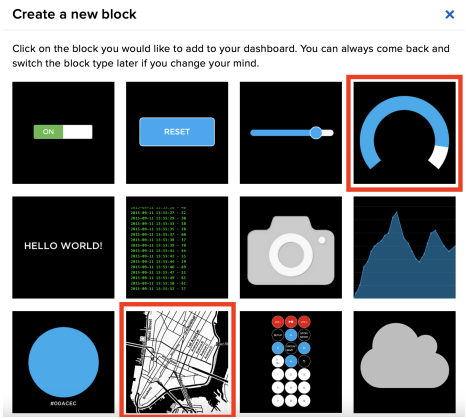
Level 3

8. The next step is to create a dashboard. To do this, select the "Dashboards" tab and then "New Dashboard":

9. A window will pop up where you should enter the selected dashboard name. Then click on the name of your dashboard to enter it. Then on the right side, select the settings symbol and choose "Create New Block".

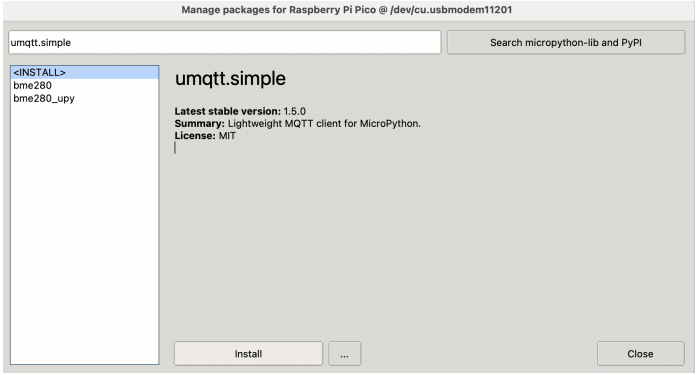
Level 3

- Adafruit IO has various blocks available for displaying data (text boxes, gauges, charts, maps, etc.). In our case, let's choose a gauge:



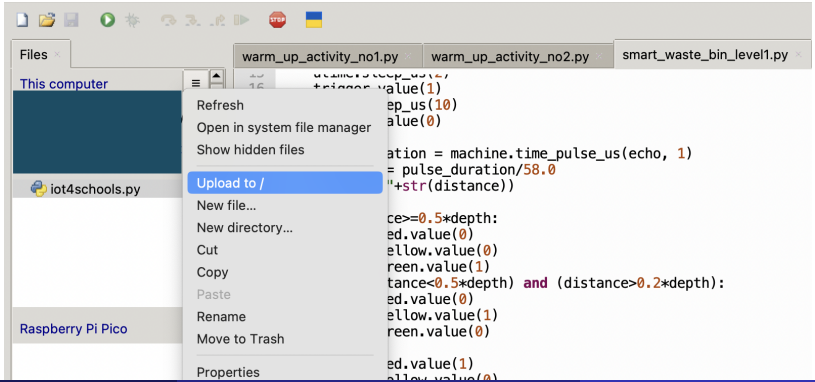
Level 3

1. Then a window will pop up asking which feed we want to connect the block to. Select the feed defining the counter to the gauges.
2. Now let's go to the Thonny editor and install the *umqtt.simple* library. To do this, select "Tools" and then "Manage packages". A window will pop up where you should enter the name of the library you want to install, in this case *umqtt.simple*:



Level 3

13. Now let's install the *iot4schools.py* library, which can be downloaded from GitHub (`https://github.com/angsam/iot4schools`) or the project page.
14. To do this, in the Thonny editor, select *View* in the top menu and then *Files*. Then find the library and select the three-line symbol. Select the *Upload to* option, which will upload the library to the Raspberry Pi Pico board.



Level 3

15. Now we can go back to our code and add the pieces necessary to send data to the cloud. First, add the *iot4schools* library:

```
import machine
import utime
import iot4schools
...
while True:
    sensorState = sensor.value()
    print(sensorState)

    if sensorState==1 and lastState==0:
        counter+=1
        print("Counter="+str(counter))

        lastState=sensorState
        utime.sleep(1)
```

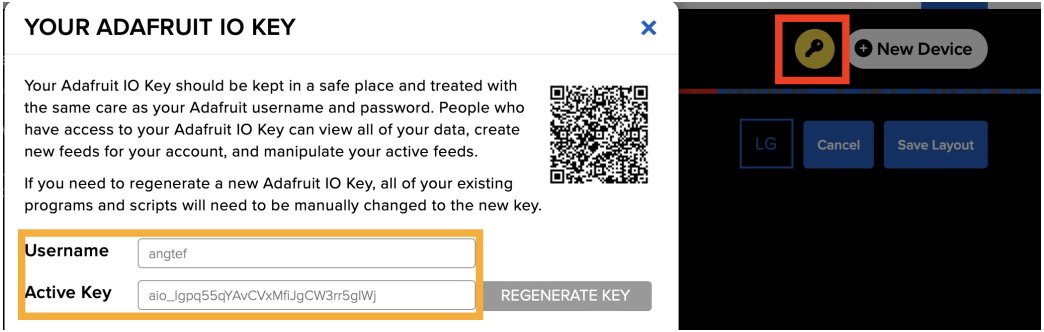
Level 3

16. Add a piece of code that will allow you to connect to your WIFI. Here you need to fill in lines 7-8. First, enter the name of your WIFI and then the password.

```
1  import machine
2  import utime
3  import iot4schools
4  ...
5  counter = 0
6
7  WIFI_SSID = "name of your wifi"
8  WIFI_PASSWORD = "password to your wifi"
9  ADAFRUIT_IO_USERNAME = "your username"
10 ADAFRUIT_IO_KEY = "key"
11
12 iot4schools.connect_wifi(WIFI_SSID, WIFI_PASSWORD)
13
14 while True:
```

Level 3

- The last two parameters are the data from Adafruit IO. Go back to the Adafruit website and click the key symbol. When you do this, your login and key will appear. Copy this data to the program to lines 9-10:



A key from Adafruit IO platform.

Level 3

- Now we will use MQTT (Message Queuing Telemetry Transport), a lightweight communication protocol based on the publish/subscribe model. First, you need to create an MQTT client and connect to it:

```
1  import machine
2  ...
3  ADAFRUIT_IO_KEY = "key"
4
5  iot4schools.connect_wifi(WIFI_SSID, WIFI_PASSWORD)
6  client = iot4schools.create_mqtt_client(ADAFRUIT_IO_USERNAME,
       ↪ADAFRUIT_IO_KEY)
7  iot4schools.connect_mqtt(client)
8
9  while True:
10 ...
```

Level 3

Then you need to create variable that store path to feed and create a function that sends data to feed. To do this, use the code below, changing only the feed name and you username in line 20.

```
1  import machine
2  ...
3  iot4schools.connect_wifi(WIFI_SSID, WIFI_PASSWORD)
4  client = iot4schools.create_mqtt_client(ADAFRUIT_IO_USERNAME,
       ↪ADAFRUIT_IO_KEY)
5  iot4schools.connect_mqtt(client)
6
7  FEED_COUNTER = "your_user_name/feeds/Counter"
8
9  def send_data(Counter):
10     client.publish(FEED_COUNTER, str(Counter))
11     print("Counter sent: "+str(Counter))
12
13 while True:
```

Level 3

20. The last step is to execute the data sending function in the main loop and pass the number of vehicles to it.

```
import machine
...
def send_data(Counter):
    client.publish(FEED_COUNTER, str(Counter))
    print("Counter sent: "+str(Counter))

while True:
    sensorState = sensor.value()
    print(sensorState)

    if sensorState==1 and lastState==0:
        counter+=1
        print("Counter="+str(counter))
        send_data(counter)
```