Line Following Car

1. Quick Start

This tutorial explains how to use the PICO2 motherboard to start line following on a track after obtaining data from the eight-channel grayscale line following module, and how to modify some parameters in the code to improve the car's performance on your own track.

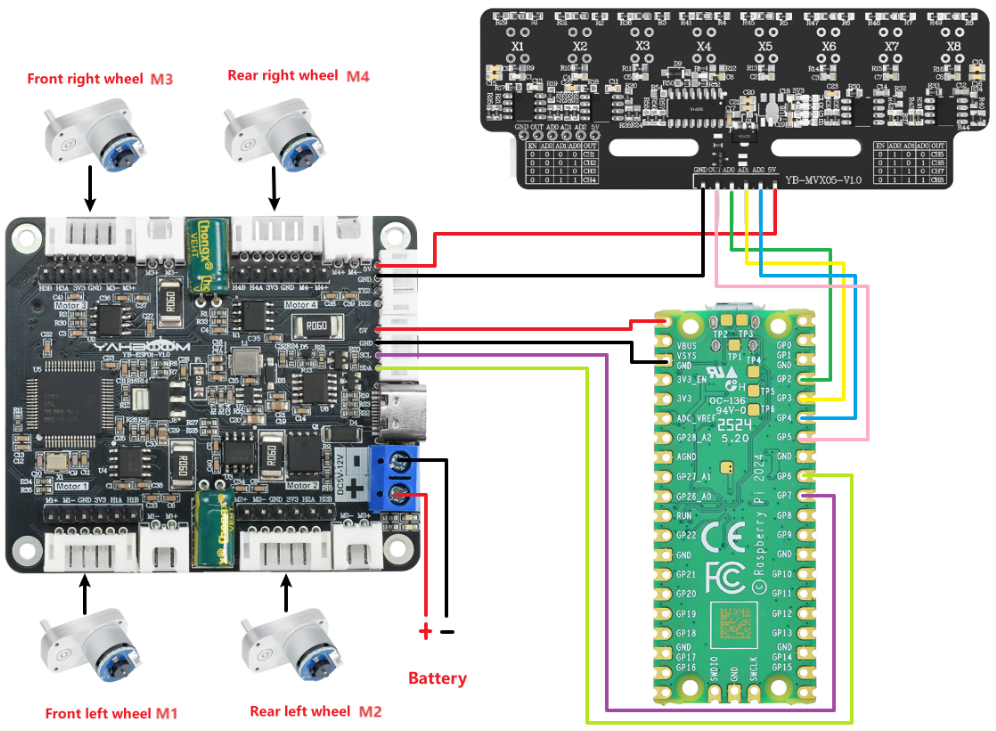
The hardware used in this tutorial consists of a PICO2 motherboard sold by Yabo, an eight-channel grayscale line following module, a V2 wheeled car chassis, an L-type 520 motor, a four-channel motor driver board, and a 12V battery pack. After connecting the wires according to the wiring diagram below, flash the program to the PICO2 motherboard, and you can power it on to start line following. If the effect is not satisfactory, you can jump to the code explanation section to view the description of the LineFollowing class and modify the line following parameters to optimize the line following effect.

In addition, you need to modify the appropriate motor PID parameters. This example uses an L-type 520 motor with the following PID values: P: 1.9, I: 0.2, D: 0.8. Use a PC serial port assistant to communicate with the four-channel motor driver board and send commands to modify the PID values. If using other motors, you need to test and determine suitable PID parameters for each motor.

This optimization process requires some experience in debugging small vehicles. Non-Yabo modules are for reference only.

2. Hardware Wiring

Since the 5V pins on the PICO2 motherboard are insufficient, the 5V and GND pins of the eight-channel grayscale line-following module can be connected to the four-channel motor driver board.

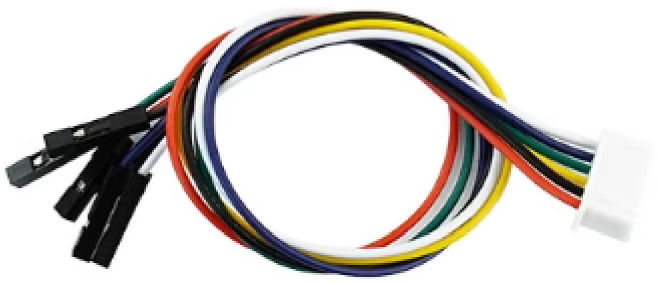


Eight-channel grayscale line-following module	PICO2
AD0	GP2
AD1	GP3
AD2	GP4
OUT	GP5

Eight-channel grayscale line-following module	Four-channel motor driver board
5V	5V
GND	GND

Four-channel motor driver board	PICO2
5V	VBUS
GND	GND
SCL	GP7
SDA	GP6

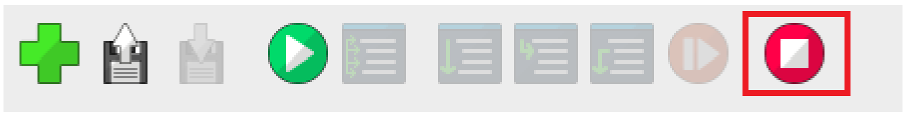
The standard cable for the eight-channel grayscale line-following module sold by Yabo is an XH2.54 to 6-pin DuPont cable. One end of the XH2.54 ribbon cable should be connected to the eight-channel grayscale module, and the other end of the DuPont cable can be connected normally as shown in the picture above:



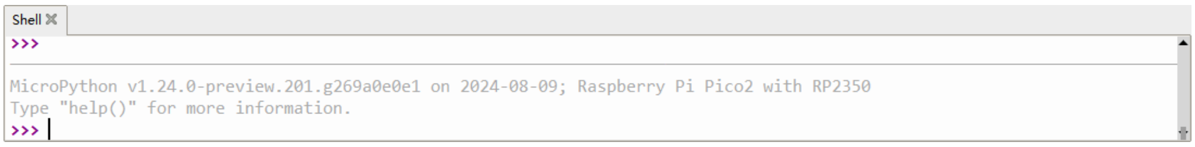
The connection method between the four-channel motor driver board sold by Yabo and the L-type 520 motor can be found in the tutorial "Introduction and Usage of Common Motors" in the four-channel motor driver board datasheet. Due to its length, it will not be elaborated here.

3. Usage

Locate the code Grayscale_Trace, open it with Thonny software, connect the Raspberry Pi pico2, and click the stop button on the far right of the top toolbar:



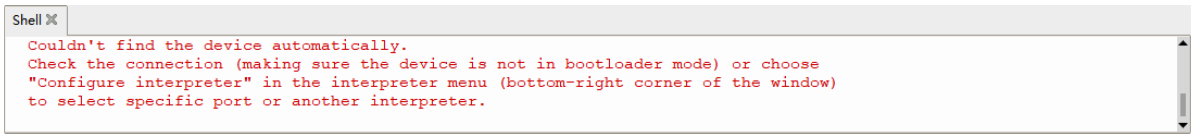
Then you will see the current firmware information of pico2 pop up in the message bar at the bottom, indicating that the software has recognized the serial port:



```
Shell
>>>

MicroPython v1.24.0-preview.201.g269a0e0e1 on 2024-08-09; Raspberry Pi Pico2 with RP2350
Type "help()" for more information.
>>>
```

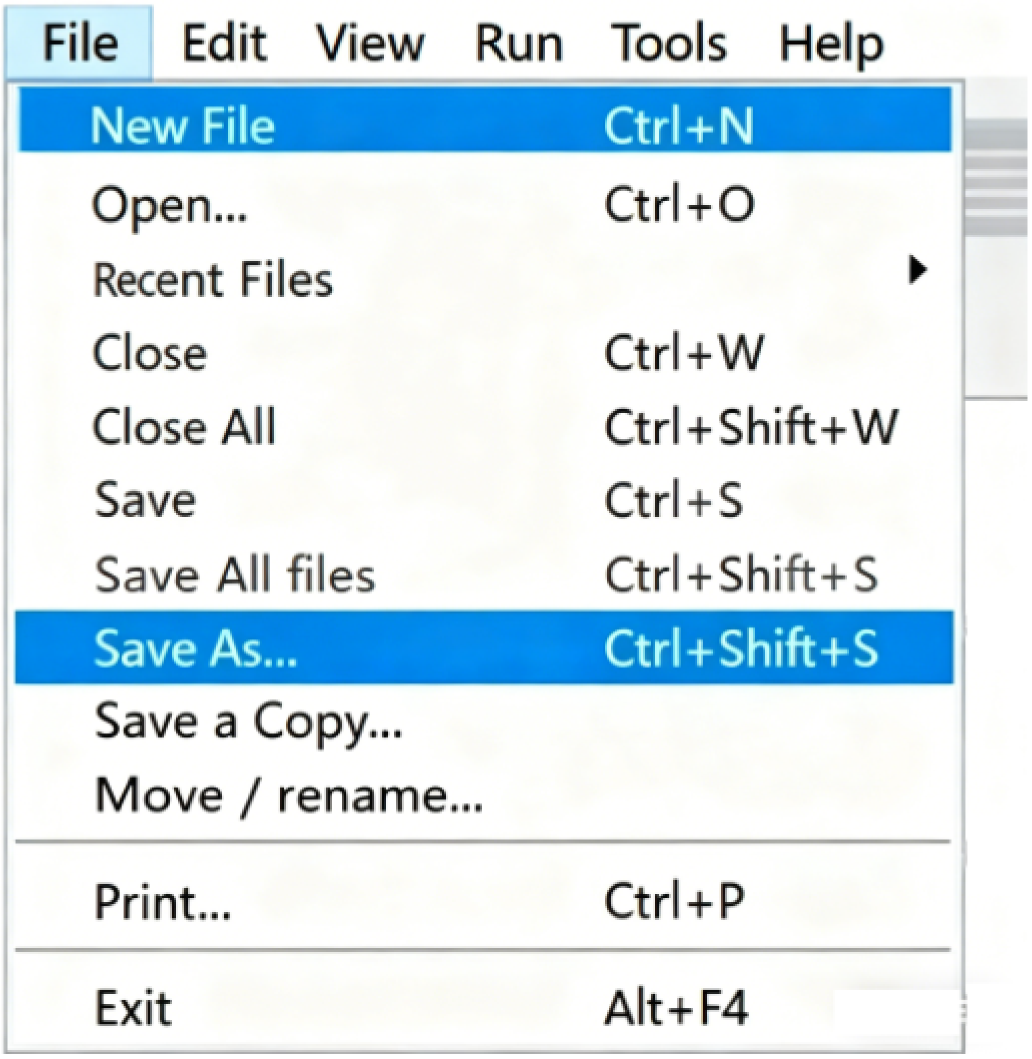
If the message bar here shows that the serial port cannot be recognized, or if you are not familiar with basic operations, you need to consult the PICO2 motherboard documentation, find the tutorial on setting up the development environment (Python), and learn the relevant basic usage and firmware flashing. Flashing the firmware to the motherboard will enable serial port recognition:



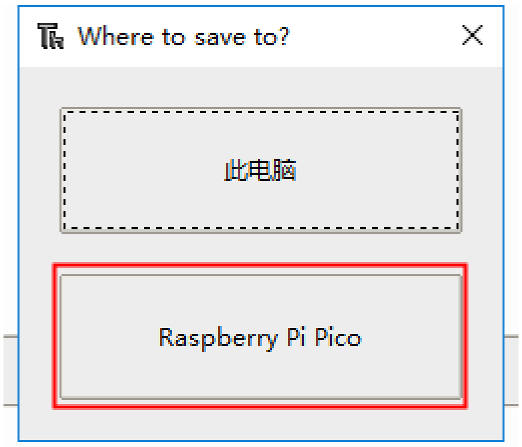
```
Shell
  Couldn't find the device automatically.
  Check the connection (making sure the device is not in bootloader mode) or choose
  "Configure interpreter" in the interpreter menu (bottom-right corner of the window)
  to select specific port or another interpreter.
```

Since the car line tracking cannot keep the PICO2 connected to the data cable continuously, the program needs to be written to the PICO motherboard for offline operation.

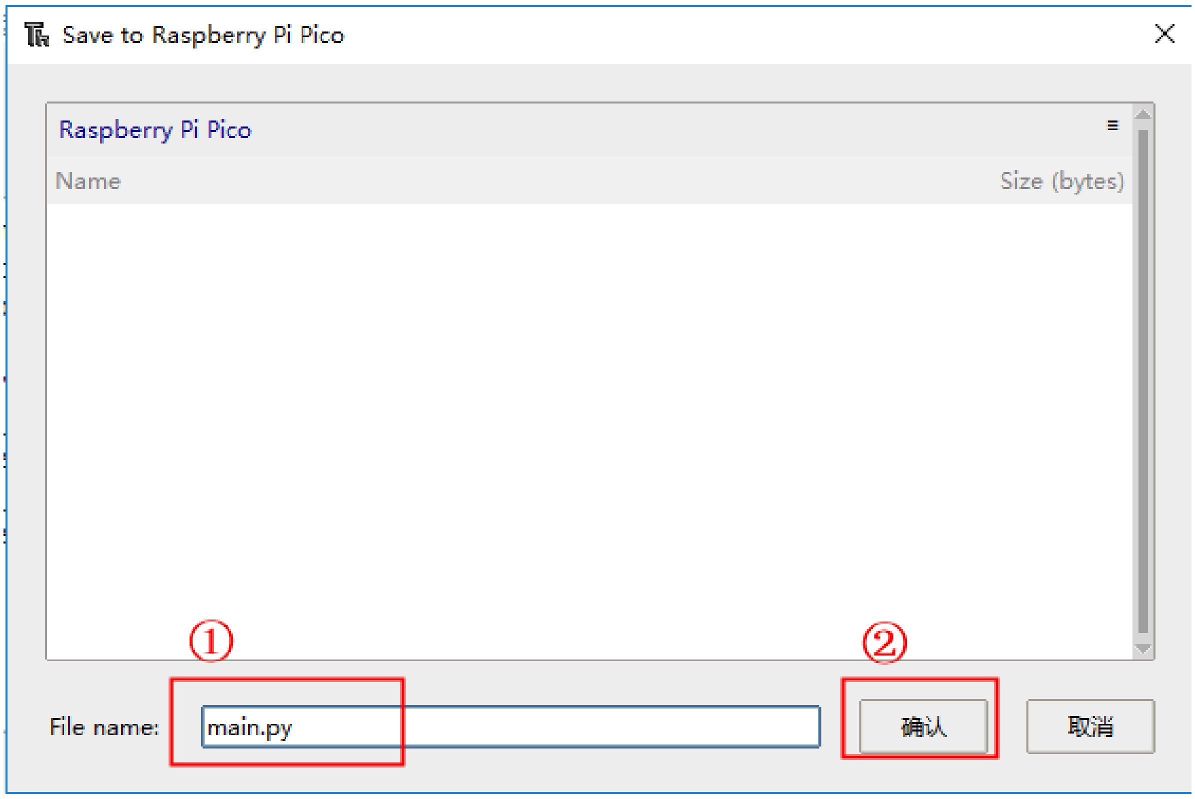
1. Select File -- Save As.



2. Select Raspberry Pi Pico.



3. Enter "main.py" in the "File name" field. Note that the filename must be "main.py" to start the program. Then click "OK".



4. After the program burning progress bar is complete, you can disconnect the connection and power on the car to make it run.
5. To remove this program from automatic execution, click "Save As" again, enter PICO, and then right-click and delete the saved `main.py` file.

4. Phenomenon and Results

After power-on, the PICO2's onboard LEDs will flash three times, indicating successful power-on. After a few seconds, the onboard LEDs will remain constantly lit, indicating that the hardware interface initialization is complete and line-following has begun. If all eight LEDs on the eight-channel grayscale line-following module are either all lit or all off, the car will not move to prevent it from running randomly on the wrong map. When the LEDs on the line-following module are inconsistent, the car begins line-following.

The LEDs on the line-following module need to be opposite to those on the track outside the line. That is, if the LED on the line is off, the corresponding LED on the track outside the line needs to be lit for normal line-following.

5. Code Explanation

```
##########################################################
# Important Configuration Parameters
#########################################################
# Select the corresponding motor parameter configuration according to the type of
motor you use
MOTOR_TYPE = 5
#1: 520 motor
#2: 310 motor
#3: Speed code disc TT motor
#4: TT DC reduction motor
#5: L-type 520 motor
#1: 520 motor
```

```
#2: 310 motor
#3: Speed code disc TT motor
#4: TT DC reduction motor 5:L type 520 motor

########################################################
# Place the grayscale line-following module on the track and observe the light
facing the line. If the light is on, the value is 1, LINE_RAW_VALUE=1; if the
light is off, the value is 0, LINE_RAW_VALUE=0
LINE_RAW_VALUE = 1
##########################################################
```

The code begins with two macro definitions: `MOTOR_TYPE` and `LINE_RAW_VALUE`.

- `MOTOR_TYPE`: Modify `MOTOR_TYPE` according to the motor you purchased. The motors adapted in the code are all currently sold by Yabo.
- `LINE_RAW_VALUE`: Place the grayscale line-following module on the track and observe the light directly facing the line. If the light is on, set `LINE_RAW_VALUE=1`; if the light is off, set the value to 0, `LINE_RAW_VALUE=0`.

```
class MotorDriver:
    def __init__(self):
        # ...
    #  Configure motor type
    def set_motor_type(self,data):
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.MOTOR_TYPE_REG, [data])
    #   Configuring Dead Zone
    def set_motor_deadzone(self,data):
        buf = [(data >> 8) & 0xFF, data & 0xFF]
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.MOTOR_DEADZONE_REG, buf)
    # Configuring magnetic loop
    def set_pluse_line(self,data):
        buf = [(data >> 8) & 0xFF, data & 0xFF]
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.MOTOR_PLUSELINE_REG, buf)
    #  Configure the reduction ratio
    def set_pluse_phase(self,data):
        buf = [(data >> 8) & 0xFF, data & 0xFF]
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.MOTOR_PLUSEPHASE_REG, buf)
    # Configuration Diameter
    def set_wheel_dis(self,data):
        bytes_data = self.float_to_bytes(data)
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.WHEEL_DIA_REG,
list(bytes_data))
    #  Controlling Speed
    def control_speed(self, m1, m2, m3, m4):
        speeds = [
            (m1 >> 8) & 0xFF, m1 & 0xFF,
            (m2 >> 8) & 0xFF, m2 & 0xFF,
            (m3 >> 8) & 0xFF, m3 & 0xFF,
            (m4 >> 8) & 0xFF, m4 & 0xFF
        ]
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.SPEED_CONTROL_REG, speeds)
    # Control PWM (for motors without encoder)
    def control_pwm(self, m1, m2, m3, m4):
        pwms = [
            (m1 >> 8) & 0xFF, m1 & 0xFF,
            (m2 >> 8) & 0xFF, m2 & 0xFF,
            (m3 >> 8) & 0xFF, m3 & 0xFF,
            (m4 >> 8) & 0xFF, m4 & 0xFF
        ]
        self.i2c_write(self.MOTOR_MODEL_ADDR, self.PWM_CONTROL_REG, pwms)
    def set_motor_parameter(self):
        # ...
        pass
```

`MotorDriver` Class

- `set_motor_type` : Configures the motor type on the motor driver board.
- `set_motor_deadzone` : Configures the dead zone parameter of the motor on the motor driver board.
- `set_pluse_line` : Configures the number of magnetic ring lines of the photoelectric encoder on the motor driver board.
- `set_pluse_phase` : Configures the reduction ratio of the motor on the motor driver board.
- `set_wheel_dis` : Configures the diameter of the wheels on the small wheels on the motor driver board.
- `control_speed` : Sends commands to the motor driver board via I2C to control the speed of the four motors in encoder closed-loop control.
- `control_pwm` : Sends commands to the motor driver board via I2C to control the speed of the four motors in PWM open-loop control.
- `set_motor_parameter`: Configures the relevant parameters of the motor (type, reduction ratio, magnetic core length, wheel diameter, dead zone) based on the preset motor type (MOTOR_TYPE).

```
# Grayscale Sensor Class
class GrayscaleSensor:
    def __init__(self, ad0_pin=2, ad1_pin=3, ad2_pin=4, out_pin=5):
        # ...
    def _select_channel(self, channel):
        """
        Select the sensor channel
        """
        self.ad0.value((channel >> 0) & 0x01)
        self.ad1.value((channel >> 1) & 0x01)
        self.ad2.value((channel >> 2) & 0x01)

    def _read_out_value(self):
        """
        Read the value from the OUT pin
        """
        return self.sensor_out.value()

    def read_all(self):
        """
        Read all channels
        """
        values = []
        for i in range(self.channels):
            self._select_channel(i)
            time.sleep_us(50)
            values.append(self._read_out_value())
        return values
```

`GrayscaleSensor` Class

- `_select_channel`: Selects the sensor channel of the multiplexer by outputting high and low levels of the GPIO based on the channel number.
- `_read_out_value`: Reads the current digital value of the sensor data output pin.
- `read_all`: Selects all sensor channels in sequence, reads and returns a list of output values for each channel.

```python
# Line Following Class
class LineFollowing:
    def __init__(self, motor, sensor):
        """
        Initialize the line following controller
        """
        ...

        #  PID Parameters
        self.kp = 160.0    # Proportional coefficient -Improve response speed
        self.ki = 0.5     # Integral coefficient - Enhance stability
        self.kd = 20.0    #  Derivative coefficient - Improve predictive ability

        ...

        #  Control Parameters
        self.base_speed = 330    #  Base speed
        self.max_speed = 400     # Max speed

        #  Sensor weights (Position weights of 8 sensors)
        self.sensor_weights = [-5.0, -4.0, -2.0, -1.0, 1.0, 2.0, 4.0, 5.0]

        ...

    def check_sensors_safe(self, sensor_values):
        # ...

    def calculate_error(self, sensor_values):
        # ... (code for calculate_error)
    def pid_control(self, error):
        # ... (code for pid_control)
    def differential_speed_control(self, pid_output):
        # ... (code for differential_speed_control)
    def follow_line(self):
        # ... (code for follow_line)
```

`LineFollowing` Class

Key Modifications for Optimizing Line Following Performance:

PID Parameters

kp: Increasing `kp` results in a faster response but is prone to oscillation; decreasing `kp` reduces oscillation but slows down the response and may cause deviation from the path.

ki: Increasing `ki` eliminates steady-state errors faster but easily causes overshoot and oscillations; decreasing `ki` reduces the risk of overshoot but slows down error elimination or may not completely eliminate it.

kd: Increasing `kd` improves system stability and reduces oscillations and overshoot, but excessive `kd` makes the system more sensitive to noise and causes jitter; decreasing `kd` reduces sensitivity to noise but weakens oscillation suppression, and the response may be delayed or overshooted.

Speed Parameters

base_speed: Determines the vehicle's forward speed during smooth line following, affecting overall efficiency and cornering response.

max_speed: Limits the maximum speed of the vehicle's left and right wheels to prevent excessive acceleration deviations.

Grayscale Line Following Module Weight Parameters

sensor_weights:Modifies the weights on the grayscale line following module. The first value refers to the X1 light at the outermost edge of the module, and the second value is for the X2 light. Increasing the first value results in a more dramatic response when the X1 light is on, while decreasing it produces a smoother response.

- `check_sensors_safe`: Checks if all grayscale sensors display the same value, used to determine if the vehicle is in a safe state during startup.
- `calculate_error`: Calculates the error value of the vehicle's deviation from the line center based on the values read from the grayscale sensors and preset weights.
- `pid_control`: Executes the PID control algorithm, calculating the control output based on the input error value, including dead-zone handling, integral zeroing, and dynamic integral limiting.
- `differential_speed_control`: Calculates the differential speed between the left and right wheels based on the PID control output, thereby adjusting the vehicle's steering and limiting its speed.
- `follow_line`: The main line-following function, responsible for reading sensor data, performing safety checks, calculating errors, and implementing PID control and differential control to drive the motors.