Line Following Car

1. Quick Start

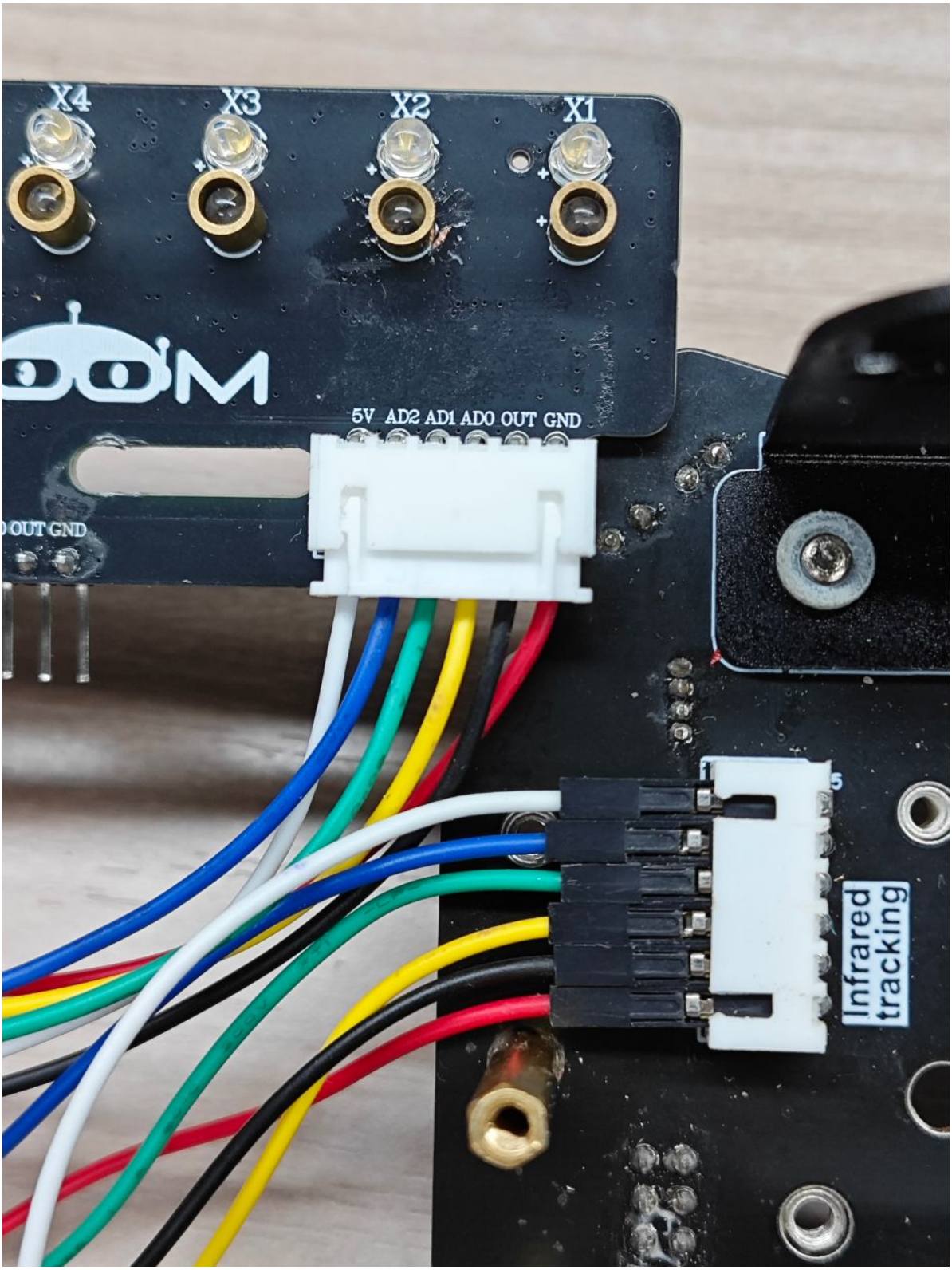
This tutorial explains how to use an STM32 motherboard to acquire data from eight infrared tracking sensors and how to modify some parameters in the code to improve the car's performance on your own track.

The hardware used in this tutorial is an STM32 development board car (sold by Yabo), eight infrared tracking sensors, a 310 motor, and a 7.4V battery pack. Refer to the wiring diagram below to connect the wires. After burning the program, place the car on the tracking track. When the sensor detects a black line, the car will automatically unlock and start following the line. If the line following effect is not good, you can skip to the code explanation section to modify the PID parameters and optimize the line following effect.

This optimization step requires the user to have some experience debugging cars. Modules not from Yabo are for reference only.

2. Hardware Wiring

Flip the STM32 development board car to the bottom. You will see a silkscreened `Infrared tracking` port. Connect this port to our eight-channel grayscale module. Since there are no specific pin markings next to the interface, you can follow the wiring instructions based on the different colored DuPont wires in the picture.

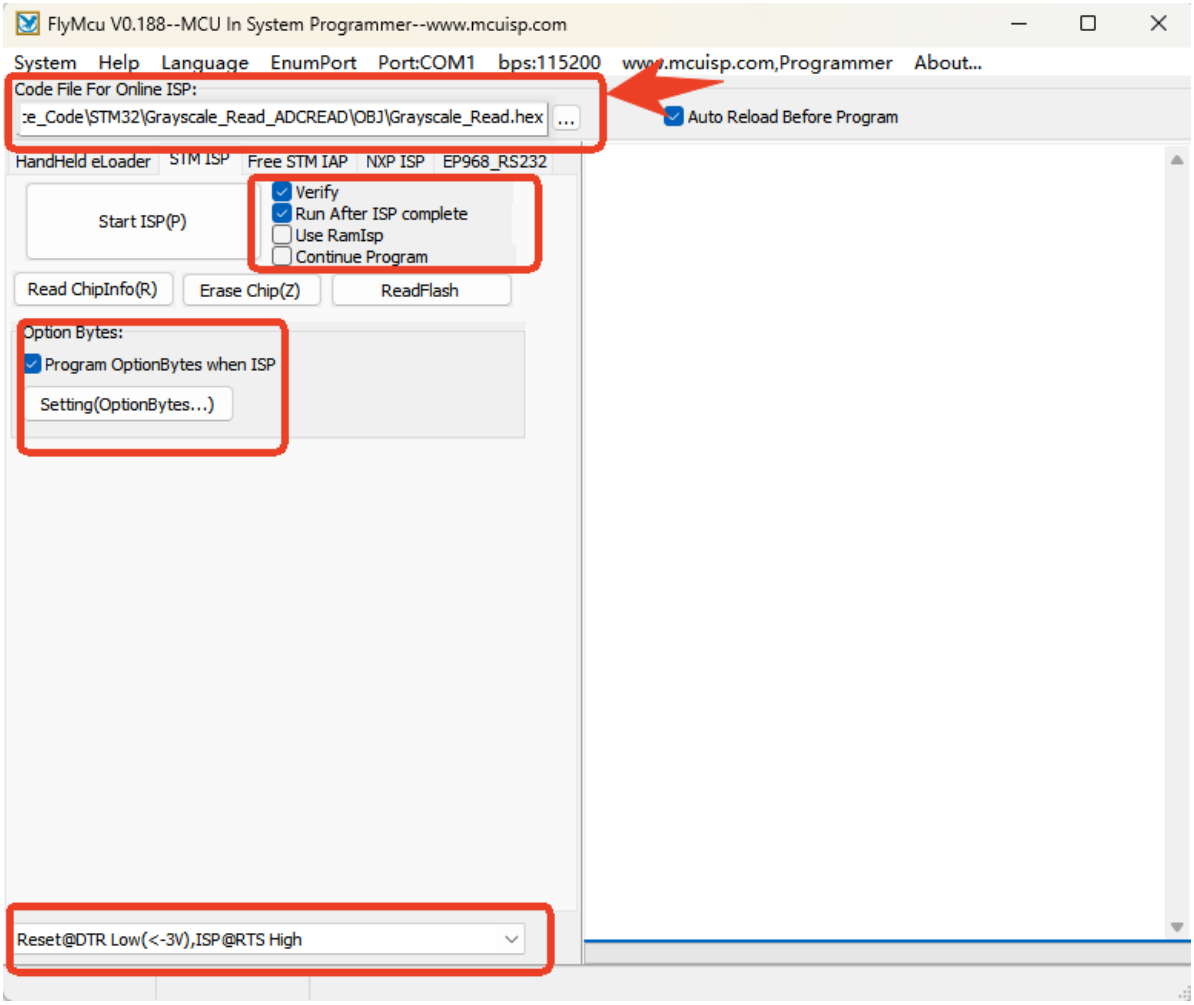


Eight-channel grayscale line-following module	STM32 development board car
5V	5V
GND	GND
AD0	PF15
AD1	PF14
AD2	PF13
OUT	PG0

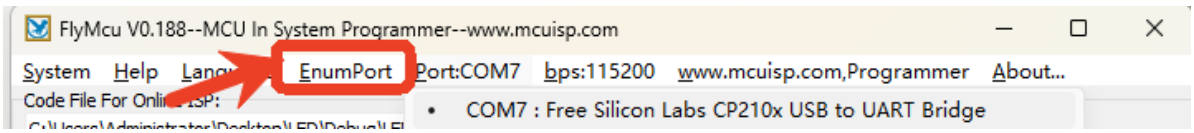
The eight-channel grayscale module sold by Yabo uses XH2.54 to 6-pin DuPont cables for connection:

3. Instructions for Use

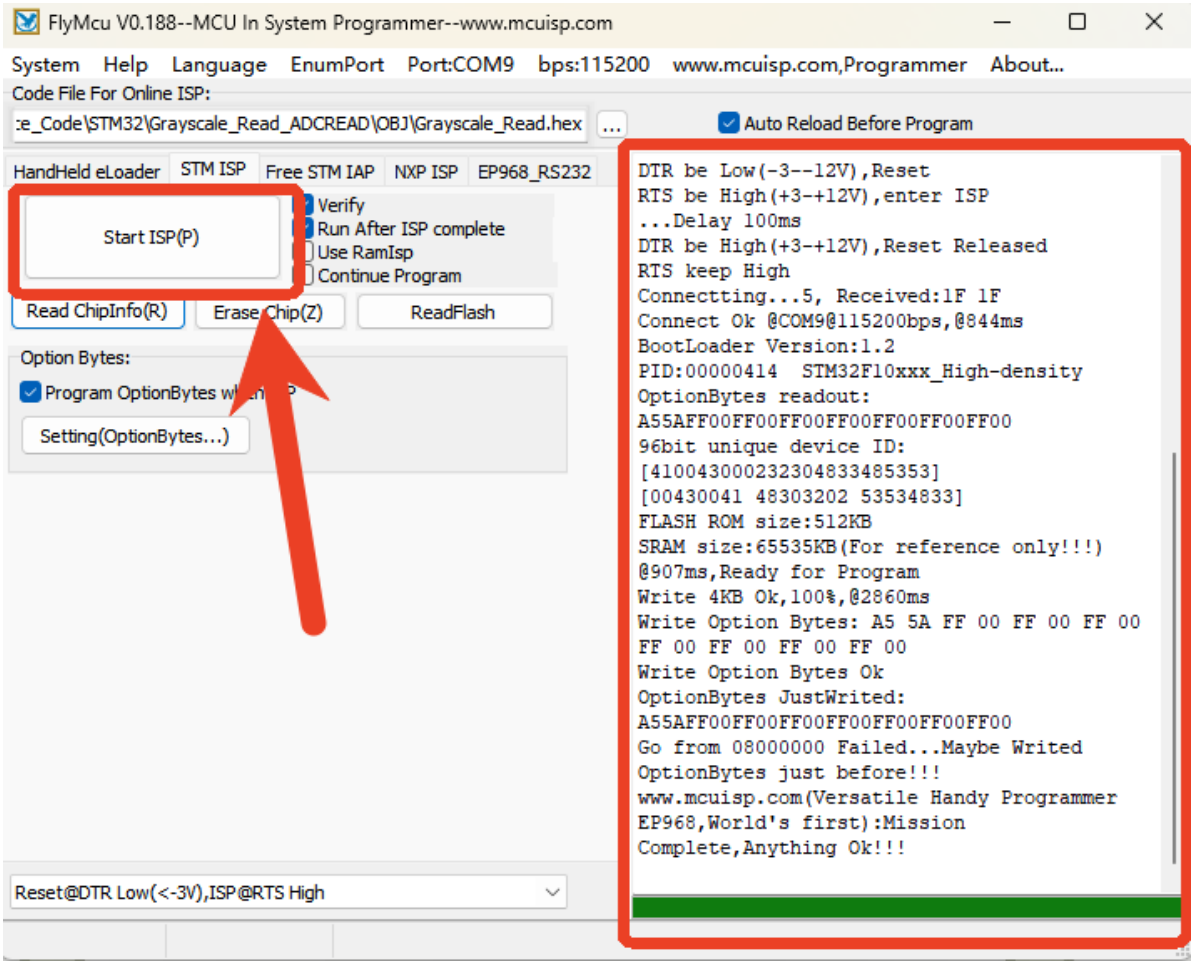
1. Open the FlyMcu programming tool. Select the hex file from the code project. Ensure the areas highlighted in red match the image.



2. Connect the STM32 to the computer. Click "Search Serial Ports" and select the COM port corresponding to the STM32.



3. Then click "Start Programming." The program will begin to be burned into the motherboard. Wait for the progress bar on the right to complete, and you will see a message indicating successful burning.



4. After burning is complete, place the car on the track, turn on the switch, and start line following.

4. Phenomenon and Results

After powering on, the program starts running. If all eight LEDs on the eight-channel grayscale line following module are either fully lit or all off, the car will not move to prevent it from running around on the wrong map. When the LEDs on the line following module are inconsistent, the car begins line following.

The LED lights on the patrol module along the designated patrol line need to be opposite to those on the track outside the line. In other words, if the LED lights on the patrol line are off, the corresponding LED lights on the track outside the line must be on for the patrol to function correctly.

5. Code Explanation

```
//////////////////////////////////////////////////////////
// Important Configuration Parameters
//////////////////////////////////////////////////////////
// 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

#define LINE_RAW_VALUE 1

//////////////////////////////////////////////////////////
```

The `LINE_RAW_VALUE` macro is set at the beginning of the main.c file.

- `LINE_RAW_VALUE`: This is used to 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.

```
// USER/main.c
void line_following_init(line_following_t* controller) {
    controller->kp = 270.0f;    //  proportional gain
    controller->ki = 0.5f;      //  integral gain
    controller->kd = 50.0f;     //  derivative gain

    controller->base_speed = 450;   //  base speed
    controller->max_speed = 800;    //  maximum speed

    controller->sensor_weights[0] = -5.0f;
    controller->sensor_weights[1] = -4.0f;
    controller->sensor_weights[2] = -2.0f;
    controller->sensor_weights[3] = -1.0f;
    controller->sensor_weights[4] = 1.0f;
    controller->sensor_weights[5] = 2.0f;
    controller->sensor_weights[6] = 4.0f;
    controller->sensor_weights[7] = 5.0f;
}
```

`line_following_init` function

Key modifications to optimize line following effect:

PID parameters

- `kp`: Increasing `kp` makes the response faster, but it is prone to oscillation; decreasing `kp` reduces oscillation, but the response will be slower and may deviate from the line.
- `ki`: Increasing `ki` can eliminate steady-state error faster, but it is prone to overshoot and oscillation; decreasing `ki` can reduce the risk of overshoot, but the error elimination will be slower or even impossible to completely eliminate. - `kd`: Increasing `kd` improves system stability and reduces wobbling and overshoot, but excessive `kd` increases noise sensitivity and causes jitter; decreasing `kd` reduces noise sensitivity but weakens wobbling suppression, potentially leading to lag or overshoot.

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 deviation.

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 module's outermost edge, and the second value is for the X2 light. Increasing the value results in a more dramatic response when the X1 light is on, while decreasing it results in a smoother response.

```
// APP/trace_task.c
```

```
bool check_sensors_safe(line_following_t* controller, uint16_t* sensor_values) {
    uint16_t first_value = sensor_values[0];
    //...
    return false;
}

float calculate_error(line_following_t* controller, uint16_t* sensor_values,
uint16_t line_raw_value) {
    float weighted_sum = 0.0f;
    int active_sensors = 0;
    //...
    return error;
}

float pid_control(line_following_t* controller, float error) {
    if (fabsf(error) < 0.6f) {
        error = 0.0f;
    }
    //...
    return output;
}

void differential_speed_control(line_following_t* controller, float pid_output,
                                int16_t* left_speed, int16_t* right_speed) {
    //...
}

void follow_line(line_following_t* controller, uint16_t* sensor_values,
                 uint16_t line_raw_value) {
    if (controller->motor_locked) {
        if (check_sensors_safe(controller, sensor_values)) {
            controller->motor_locked = false;
        } else {
            Contrl_Speed(0, 0, 0, 0);
            return;
        }
    }

    float error = calculate_error(controller, sensor_values, line_raw_value);
    float pid_output = pid_control(controller, error);
    int16_t left_speed, right_speed;
    differential_speed_control(controller, pid_output, &left_speed,
&right_speed);

    Contrl_Speed(left_speed, left_speed, right_speed, right_speed);
}
```

`trace_task` Module

- `check_sensors_safe`: Checks if all grayscale sensors display the same value, used to determine if the vehicle is in a safe state at startup.
- `calculate_error`: Calculates the error value of the vehicle's deviation from the line center based on the values read by the grayscale sensors and preset weights.
- `pid_control`: Executes the PID control algorithm and calculates the control output based on the input error value.

- `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.
- `follow_line`: The main function for line following, responsible for reading sensor data, performing safety checks, calculating errors, and performing PID control and differential control to drive the motor.

```
// APP/app_motor.c
void Motion_Handle(void)
{
    Motion_Get_Speed(&car_data);

    if (g_start_ctrl)
    {
        PID_Calc_Motor(&motor_data);
        Motion_Set_Pwm(motor_data.speed_pwm[0], motor_data.speed_pwm[1],
motor_data.speed_pwm[2], motor_data.speed_pwm[3]);
    }
}

void Motion_Set_Pwm(int16_t Motor_1, int16_t Motor_2, int16_t Motor_3, int16_t
Motor_4)
{
    if (Motor_1 >= -MOTOR_MAX_PULSE && Motor_1 <= MOTOR_MAX_PULSE)
    {
        Motor_Set_Pwm(Motor_M1, Motor_1);
    }
    //...
    if (Motor_4 >= -MOTOR_MAX_PULSE && Motor_4 <= MOTOR_MAX_PULSE)
    {
        Motor_Set_Pwm(Motor_M4, Motor_4);
    }
}
```

`app_motor` Module

- `Motion_Handle`: The core task of motor speed closed-loop control, executed periodically in a 10ms timer interrupt (TIM6). Call flow:

1. Call `Motion_Get_Speed` to read the encoder feedback speed, internally performing PID calculations.
2. Call `Motion_Set_Pwm` to output the PWM control quantity to the motor driver.

- `Motion_Set_Speed`: Saves the calculated target speeds of the four wheels to the `motor_data` structure for use by the PID controller.

```
// APP/PID_Motor.h
#define PID_MOTOR_KP  (1.5f)
#define PID_MOTOR_KI  (0.08f)
#define PID_MOTOR_KD  (0.5f)
```

If you are not using the 310 motor sold by Yabo, you can modify the motor's PID parameters here. Specific values need to be tested by yourself.