

PID controls car speed

PID controls car speed

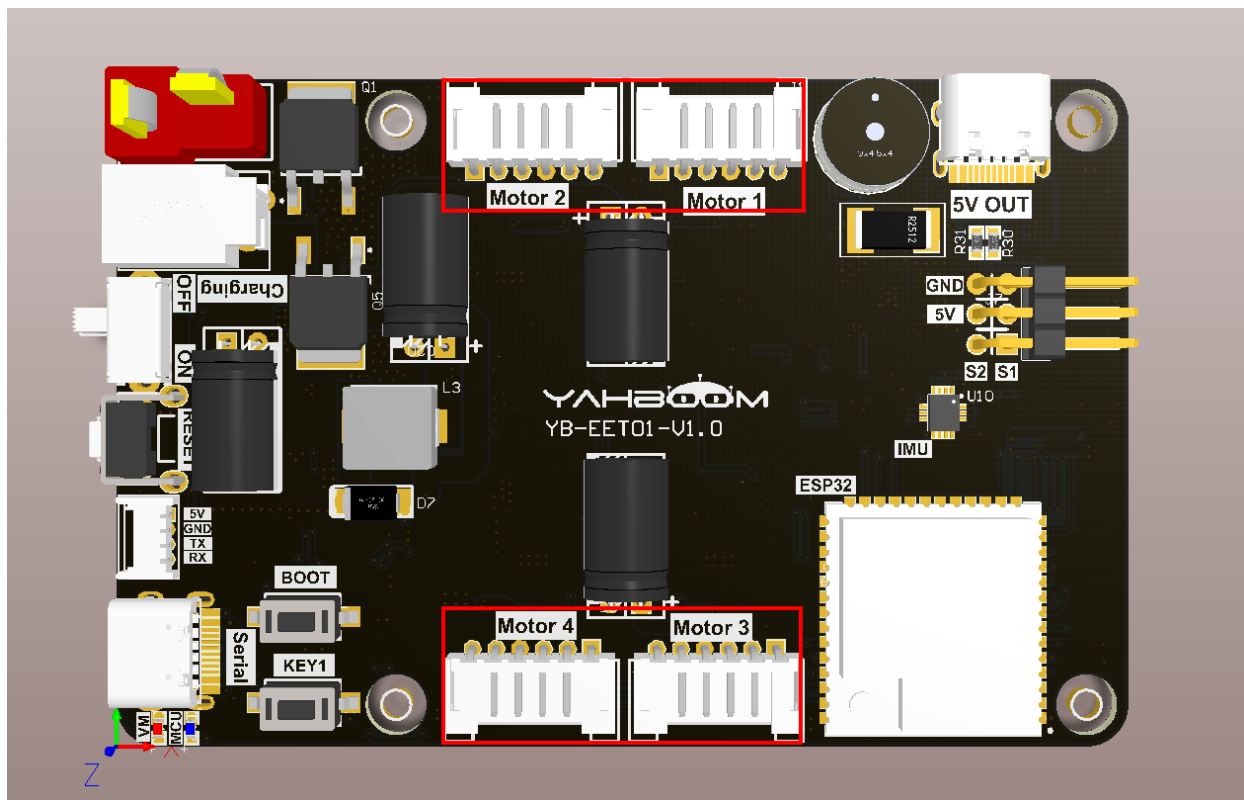
1. Experimental purpose
2. Hardware connection
3. Core code analysis
4. Compile and download the burning firmware
5. Experimental results

1. Experimental purpose

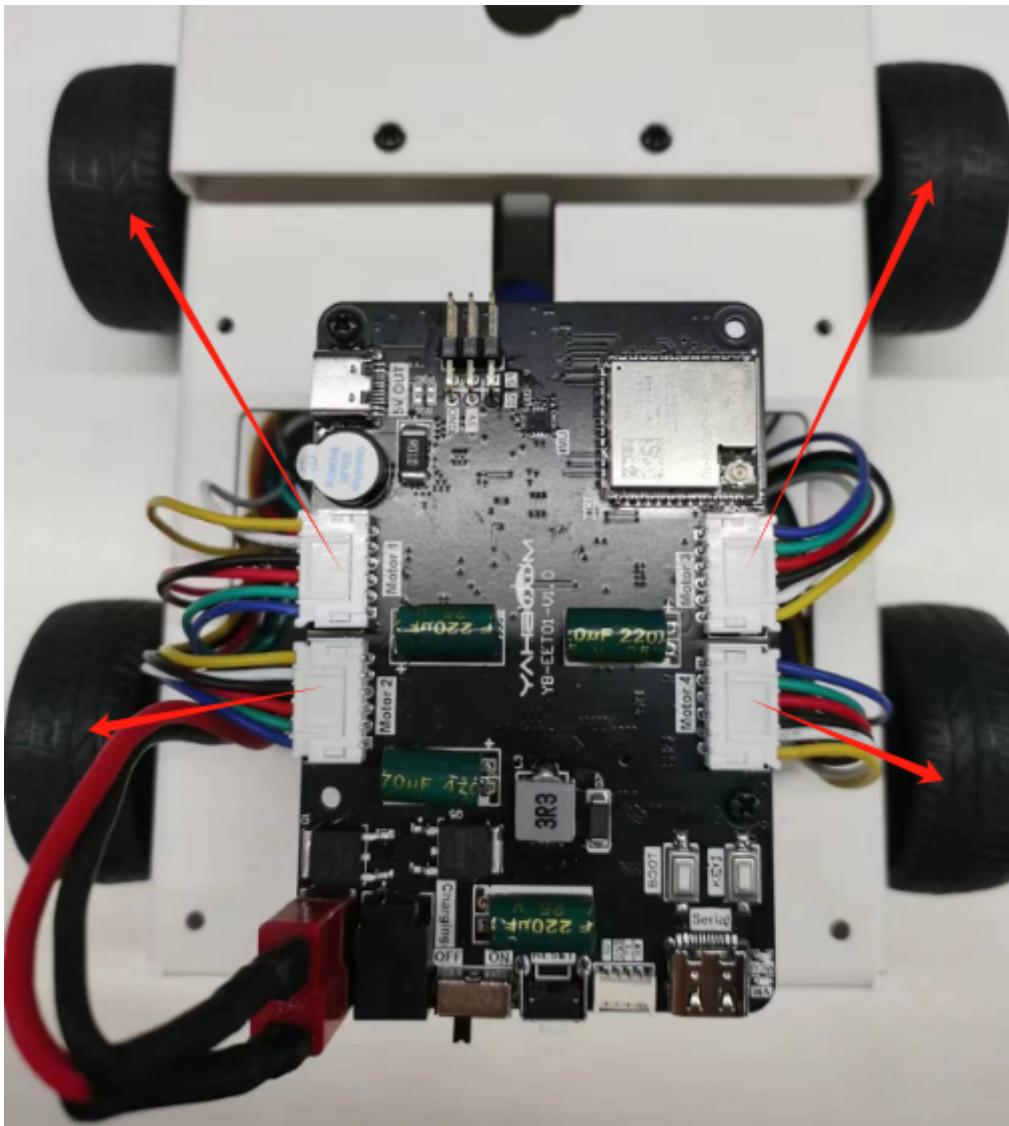
Use the encoded motor interface of the microROS control board to learn how ESP32 uses the number of motor encoder pulses combined with the PID algorithm to control the speed of motor rotation.

2. Hardware connection

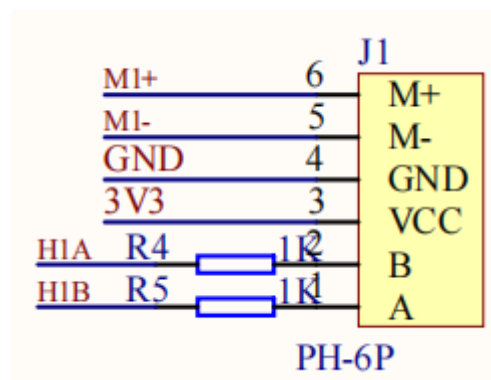
As shown in the figure below, the microROS control board integrates four encoder motor control interfaces. An additional encoder motor needs to be connected. The motor control interface supports 310 motors. A type-C data cable also needs to be connected to the computer and the microROS control board to burn firmware. Function.



The corresponding names of the four motor interfaces are: left front wheel->Motor1, left rear wheel->Motor2, right front wheel->Motor3, right rear wheel->Motor4.



Motor interface line sequence, there is a detailed line sequence silk screen on the back of the microROS control board. Here we take Motor1 as an example. M1+ and M1- are the interfaces for controlling the rotation of the motor. GND and VCC are the power supply circuits of the encoder. H1A and H1B are the encoder pulses. Detection pin.



Note: If you are using the 310 motor and motor cable provided by Yabo Intelligence, connect the white wire shell end to the interface on the microROS control board, and the black wire shell end to the 310 motor interface.

3. Core code analysis

The virtual machine path corresponding to the program source code is as follows

```
~/esp/samples/esp32_samples/pid_speed
```

Since the PID algorithm component needs to be used, the `pid_ctrl` component is imported into the motor.

```
dependencies:
  pid_ctrl: "^0.1.1"
```

Since the PID algorithm requires an encoder and a PWM motor, the components of the encoder and PWM motor need to be copied to the components directory. Then initialize the encoder and PWM motor while initializing the motor, and open the motor task.

```
void Motor_Init(void)
{
    Encoder_Init();
    PwmMotor_Init();

    xTaskCreatePinnedToCore(Motor_Task, "Motor_Task", 10*1024, NULL, 10, NULL, 1);
}
```

After the motor task starts, the PID related parameters are saved, and the incremental PID algorithm is used to control the motor. Among them, `pid_runtime_param.kp = 1.0`; `pid_runtime_param.ki = 0.2`; `pid_runtime_param.kd = 0.2`; are the default PID parameters.

```
static void Motor_Task(void *arg)
{
    ESP_LOGI(TAG, "Start Motor_Task with core:%d", xPortGetCoreID());
    pid_runtime_param.kp = 1.0;
    pid_runtime_param.ki = 0.2;
    pid_runtime_param.kd = 0.2;
    pid_runtime_param.cal_type = PID_CAL_TYPE_INCREMENTAL;
    pid_runtime_param.max_output = PWM_MOTOR_MAX_VALUE;
    pid_runtime_param.min_output = -PWM_MOTOR_MAX_VALUE;
    pid_runtime_param.max_integral = 1000;
    pid_runtime_param.min_integral = -1000;

    pid_ctrl_config_t pid_config = {
        .init_param = pid_runtime_param,
    };
    for (int i = 0; i < MOTOR_MAX_NUM; i++)
    {
        ESP_ERROR_CHECK(pid_new_control_block(&pid_config, &pid_motor[i]));
    }

    vTaskDelay(pdMS_TO_TICKS(100));
}
```

```

    TickType_t lastwakeTime = xTaskGetTickCount();
    while (1)
    {
        Motor_PID_Ctrl();
        vTaskDelayUntil(&lastwakeTime, MOTOR_PID_PERIOD);
    }

    Motor_Stop(STOP_BRAKE);
    vTaskDelete(NULL);
}

```

If you need to update the PID parameters, you can call the Motor_Update_PID_Parm function to fine-tune the PID parameters.

```

void Motor_Update_PID_Parm(float pid_p, float pid_i, float pid_d)
{
    pid_runtime_param.kp = pid_p;
    pid_runtime_param.ki = pid_i;
    pid_runtime_param.kd = pid_d;
    for (int i = 0; i < MOTOR_MAX_NUM; i++)
    {
        pid_update_parameters(pid_motor[i], &pid_runtime_param);
    }
}

```

The current PID parameters can be read through the Motor_Read_PID_Parm function.

```

void Motor_Read_PID_Parm(float* out_p, float* out_i, float* out_d)
{
    *out_p = pid_runtime_param.kp;
    *out_i = pid_runtime_param.ki;
    *out_d = pid_runtime_param.kd;
}

```

Set the speed of the motor in m/s. A positive number means forward rotation and a negative number means backward rotation. Due to the motor speed limit, the input range of speed is -1.0~1.0m/s.

```

void Motor_Set_Speed(float speed_m1, float speed_m2, float speed_m3, float speed_m4)
{
    static float speed_m[MOTOR_MAX_NUM] = {0};
    speed_m[0] = Motor_Limit_Speed(speed_m1);
    speed_m[1] = Motor_Limit_Speed(speed_m2);
    speed_m[2] = Motor_Limit_Speed(speed_m3);
    speed_m[3] = Motor_Limit_Speed(speed_m4);

    for (int i = 0; i < MOTOR_MAX_NUM; i++)
    {
        // 速度转化成10毫秒编码器目标数量
        // The speed is converted to the number of encoder targets in 10
        milliseconds
    }
}

```

```

        speed_count[i] = speed_m[i] /
(MOTOR_WHEEL_CIRCLE/MOTOR_ENCODER_CIRCLE/MOTOR_PID_PERIOD);
        pid_target[i] = (float)speed_count[i];
    }
    pid_enable = 1;
}

```

The PID algorithm calculates the deviation based on the current encoder data and the previous encoder data and outputs the PWM pulse value of the motor, thereby controlling the motor to rotate at the set speed.

```

static void Motor_PID_Ctrl(void)
{
    static int last_count[MOTOR_MAX_NUM] = {0};
    static int cur_count[MOTOR_MAX_NUM] = {0};
    static float real_pulse[MOTOR_MAX_NUM] = {0};
    static float new_speed[MOTOR_MAX_NUM] = {0};

    for (int i = 0; i < MOTOR_MAX_NUM; i++)
    {
        cur_count[i] = Encoder_Get_Count(ENCODER_ID_M1 + i);
        real_pulse[i] = cur_count[i] - last_count[i];
        last_count[i] = cur_count[i];
        read_speed[i] = real_pulse[i] *
(MOTOR_WHEEL_CIRCLE/MOTOR_ENCODER_CIRCLE/MOTOR_PID_PERIOD);
        if (pid_enable)
        {
            pid_compute(pid_motor[i], pid_target[i] - real_pulse[i], &new_speed[i]);
            PwmMotor_Set_Speed(MOTOR_ID_M1 + i, (int)new_speed[i]);
            new_pid_output[i] = new_speed[i];
        }
    }
}

```

Call the Motor_Init function in app_main to initialize the encoder motor, then set the speed of the four wheels of the car, print the current speed values of the four motors every 100 milliseconds, and the car stops after 5 seconds.

```

void app_main(void)
{
    printf("hello yahboom\n");
    ESP_LOGI(TAG, "Nice to meet you!");
    float speed_m1, speed_m2, speed_m3, speed_m4;
    int count = 0;
    vTaskDelay(pdMS_TO_TICKS(1000));

    Motor_Init();
    Motor_Set_Speed(0.5, 0.5, -0.30, -0.30);

    while (1)

```

```

{
    Motor_Get_Speed(&speed_m1, &speed_m2, &speed_m3, &speed_m4);
    ESP_LOGI(TAG, "Read_Speed:%.2f, %.2f, %.2f, %.2f", speed_m1, speed_m2,
speed_m3, speed_m4);
    vTaskDelay(pdMS_TO_TICKS(100));
    count++;
    if(count >= 50)
    {
        Motor_Stop(STOP_COAST);
    }
}
}

```

4. Compile and download the burning firmware

Note: Since the connected motor will rotate after the firmware is burned, please lift the car into the air first to prevent the car from moving around on the table.

Use a Type-C data cable to connect the virtual machine/computer and the microROS control board. If the system pops up, choose to connect to the virtual machine.

Activate the ESP-IDF development environment. Note that every time you open a new terminal, you need to activate the ESP-IDF development environment before compiling the firmware.

```
source ~/esp/esp-idf/export.sh
```

Enter the project directory

```
cd ~/esp/Samples/esp32_samples/pid_speed
```

Compile, flash, and open the serial port simulator

```
idf.py build flash monitor
```

If you need to exit the serial port simulator, press **Ctrl+]**.

5. Experimental results

The serial port simulator prints the "hello yahboom" greeting. At this time, the four-way motor speed will be printed every 100 milliseconds. And the motor starts to rotate at the preset speed.

Motor_Set_Speed(0.5, 0.5, -0.30, -0.30) is a function to adjust the speed. The motor speed can be adjusted according to actual needs. Theoretically, there will be a certain error between the motor rotation speed and the actual setting. If there is not much difference between the read speed and the set speed, it means the motor is normal. The motor will automatically stop after 5 seconds of rotation. If you need to let the motor move again, please press the reset button on the microROS control board.

```
I (1530) gpio: GPIO[13]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (1539) gpio: GPIO[14]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (1548) MOTOR: Start Motor_Task with core:1
I (1548) MAIN: Read_Speed:0.00, 0.00, 0.00, 0.00
I (1658) MAIN: Read_Speed:0.00, 0.00, 0.00, 0.00
I (1758) MAIN: Read_Speed:0.20, 0.20, -0.09, -0.09
I (1858) MAIN: Read_Speed:0.39, 0.39, -0.19, -0.20
I (1958) MAIN: Read_Speed:0.48, 0.46, -0.28, -0.29
I (2058) MAIN: Read_Speed:0.51, 0.51, -0.29, -0.29
I (2158) MAIN: Read_Speed:0.51, 0.49, -0.30, -0.30
I (2258) MAIN: Read_Speed:0.51, 0.51, -0.30, -0.30
I (2358) MAIN: Read_Speed:0.51, 0.49, -0.30, -0.29
I (2458) MAIN: Read_Speed:0.49, 0.51, -0.29, -0.30
I (2558) MAIN: Read_Speed:0.49, 0.52, -0.30, -0.30
I (2658) MAIN: Read_Speed:0.51, 0.49, -0.30, -0.30
I (2758) MAIN: Read_Speed:0.51, 0.51, -0.30, -0.30
I (2858) MAIN: Read_Speed:0.51, 0.49, -0.30, -0.30
I (2958) MAIN: Read_Speed:0.49, 0.49, -0.30, -0.29
I (3058) MAIN: Read_Speed:0.52, 0.51, -0.29, -0.29
I (3158) MAIN: Read_Speed:0.49, 0.49, -0.29, -0.30
```