THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

ISDN 2602

Final Project

Code Explanation

User Manual

Ver. 1.0

Jason Chan

3rd May 2024

Table of Contents

Introduction

Since there are many tasks for the car perform at the same time, FreeRTOS is used instead of super loop.

FreeRTOS

FreeRTOS, an open-source and popular real-time operating system, is now available for the ESP32 platform, bringing robust multitasking capabilities and efficient resource management.

Designed to harness the full potential of ESP32's dual-core architecture, FreeRTOS offers a flexible and reliable solution for developing applications that require precise timing, task prioritization, and synchronization. With its small footprint and efficient kernel, FreeRTOS optimizes the use of system resources, ensuring optimal performance even in resource-constrained environments.

Skeleton Code:

```
8   /*Include the FreeRTOS library*/
9   #include "freertos/FreeRTOS.h"
10  #include "freertos/task.h"
11  #include "freertos/semphr.h"
12  #include "freertos/queue.h"
```

Libraries are added to enable FreeRTOS in the program.

Creating User Task

In FreeRTOS, “tasks” are used instead of putting the code inside the loop(). To create a task, StackType, TaskTCB and TaskHandler need to be initialized.

Take a simple LED blink function as an example.

```
/*------------------------------------------------------*/
/*------------LED Blinking Task-------------
  --------Stack and Handle Settings---------
To ensure there is visualization that the program is running*/
StackType_t uxBlinkTaskStack[configMINIMAL_STACK_SIZE];
StaticTask_t xBlinkTaskTCB;
TaskHandle_t BlinkTaskTCB;
```

Task Function

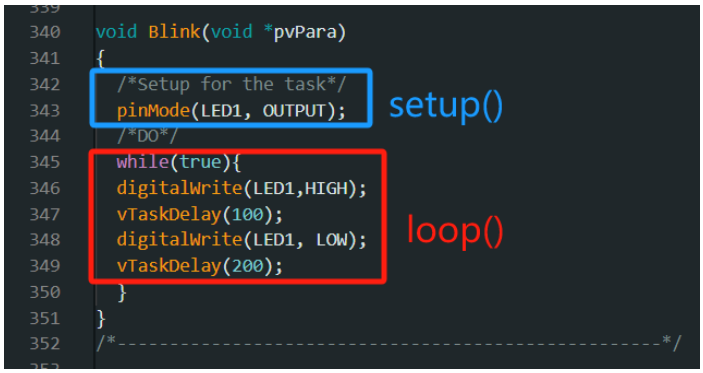
```
void Blink(void *pvPara)
{
  /*Setup for the task*/
  pinMode(LED1, OUTPUT);
  /*DO*/
  while(true){
  digitalWrite(LED1,HIGH);
  vTaskDelay(100);
  digitalWrite(LED1, LOW);
  vTaskDelay(200);
  }
}
/*------------------------------------------------------*/
```

The template for creating a task function: void TaskName(void *pvPara)

**Where void *pvPara is a must, even it is not used in the function. **

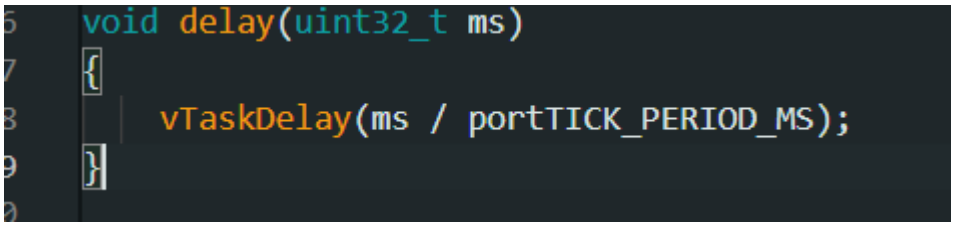
There are two parts inside the function:

1. The Setup Part (Only run once)
2. Keep Running Part (looping without giving specific instructions such as semphr)



****IMPORTANT****

In FreeRTOS, vTaskDelay( ) is used but not delay( ), although delay( ) is actually calling vTaskDelay( )



vTaskDelay( ) must be added inside the while loop otherwise the task will not run in MCU.

After creating the user task function, the next step is to tell the MCU to run the task.

ESP32S3 MCU Cores

ESP32S3 is a dual-core MCU, and we can assign different tasks to different cores (Core 0 and Core 1). Typically, Core 0 is used for Wifi or Bluetooth related tasks and Core 1 for the remain tasks.

In the skeleton code:

Inside void setup(),

```
void setup() {
// put your setup code here, to run once:
Serial.begin(115200);
Serial.println("---------Initializing...---------");
//Set up PWM Channel for Motor
Motor::Init();
Serial.println("Wheel Motors Initialized");

//Set up PWM Channel for Servo Motor
Servo::Init();
Serial.println("Servo Motor Initialized");

//Initialize IR Sensor
IR::Init();
Serial.println("IR Sensor Initialized");

//Initialize RFID Reader
Wire.begin(RFID_SDA, RFID_SCL);
mfrc522.PCD_Init();

Serial.println("RFID Initialized");
//Initialize IMU
```

After initializing all the firmware...

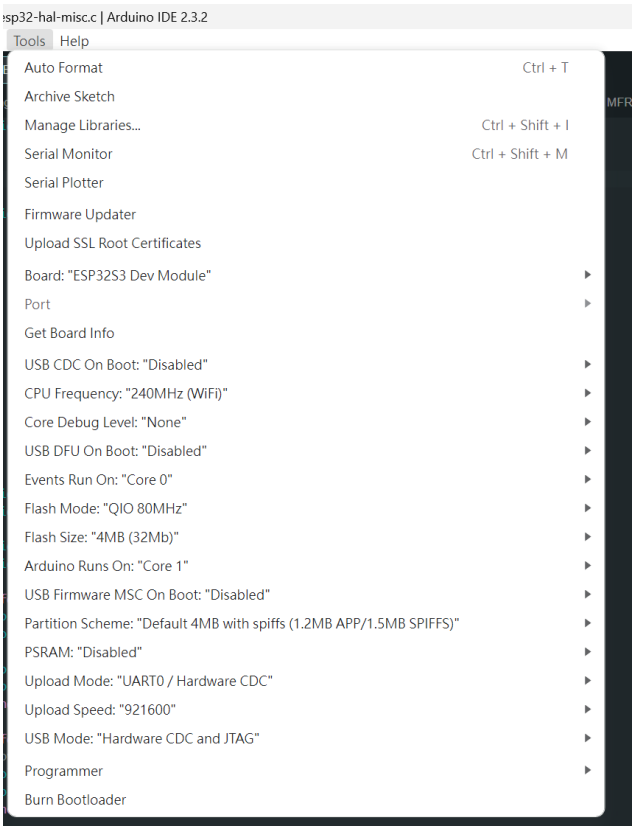
```
/*FreeRTOS Task Pinned to core*/
/*Do not change the config of the core
  Events Run on Core:   Core 0 (For FreeRTOS)
  Arduino Runs on Core: Core 1 (As Default)

  Run Core Tasks Config:
  Core 0: local  task  (Control)
  Core 1: online task (Firebase)*/
/*xTaskCreatePinnedtoCore: pin the specific task to desired core (esp32 is a dual cores MCU)
  xTaskCreatePinnedToCore(  void((void *pvPara)), Text for the task, Stack (Min. is 1024), const para. , &TaskTCB, uxPriority, Core )*/
xTaskCreatePinnedToCore(FireBaseTask, "FireBase", 10000, NULL, 3 , &FireBaseTaskTCB, 0 );
xTaskCreatePinnedToCore(Blink, "Blink", 2048, NULL, 1 , &BlinkTaskTCB, 1 );
xTaskCreatePinnedToCore(RFIDTagReader, "RFIDReader", 2048, NULL, 2 , &RFIDTagReaderTCB, 1 );
xTaskCreatePinnedToCore(LineTrackingTask, "LineTracking", 10000, NULL, 2 , &LineTrackingTaskTCB, 1 );
xTaskCreatePinnedToCore(calculateRPMTask, "calcula  teRPM", 10000, NULL, 3 , &calculateRPMTaskTCB, 1 );
```

The tasks created above are assigned to a specific core according to the logic above.

****xTaskCreatePinnedtoCore: pin the specific task to desired core (esp32 is a dual core MCU)**

xTaskCreatePinnedToCore(void((void *pvPara)), Text for the task, Stack (Min. is 1024), const para. , &TaskTCB, uxPriority, Core)**

Upload Setting



Explanation for skeleton code

RFID Reader

The model of the RFID Reader used on the car is MFRC522, and most of the codes in this function are borrowed from MFRC522_I2C driver.

```
/*Creating the Class for RFID Reader*/
MFRC522 mfrc522(0x28, RFID_RST);

/*Function for getting the RFID Tag ID Number*/
String getTagUID(){
    String tagUID = "";
  for (byte i = 0; i < mfrc522.uid.size; i++) {
    tagUID += mfrc522.uid.uidByte[i] < 0x10 ? "0" : "";
    tagUID += String(mfrc522.uid.uidByte[i], HEX);
  }
  return tagUID;
```

Creating the Class for the RFID Reader with the address and the Reset Pin.

String getTagUID() is the function that returns the RFIDTagUID in String.

The result is shown below:

```
20:34:09.957 ->  Motors Initialized.
20:34:09.957 -> Wheel Motors Initialized
20:34:09.957 -> Servo Motor Initialized.
20:34:09.957 -> Servo Motor Initialized
20:34:09.957 -> IR Sensor Initialized
20:34:09.957 -> RFID Initialized
20:34:09.957 -> Interrupt Pins Initialized
20:34:09.957 -> ---------Initialized---------
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:09.957 -> RFID Tag:
20:34:10.049 -> RFID Tag:
20:34:10.128 -> RFID Tag:
```

```
20:34:20.316 -> RFID Tag: b3c35214
20:34:20.397 -> RFID Tag: b3c35214
20:34:20.478 -> RFID Tag: b3c35214
20:34:20.567 -> RFID Tag: b3c35214
20:34:20.694 -> RFID Tag: b3c35214
20:34:20.787 -> RFID Tag: b3c35214
20:34:20.878 -> RFID Tag: b3c35214
20:34:20.971 -> RFID Tag: b3c35214
20:34:21.016 -> RFID Tag: b3c35214
20:34:21.106 -> RFID Tag: b3c35214
20:34:21.236 -> RFID Tag: b3c35214
20:34:21.329 -> RFID Tag: b3c35214
20:34:21.374 -> RFID Tag: b3c35214
20:34:21.489 -> RFID Tag: b3c35214
20:34:21.582 -> RFID Tag: b3c35214
```

LineTracking Task

```
/*-------------------------------------------------------------------------------------------------------------------------------*/
/*------------Line Tracking Task------------*/
/*--------Stack and Handle Settings---------*/

StackType_t uxLineTrackingTaskStack[configMINIMAL_STACK_SIZE];
StaticTask_t xLineTrackingTaskTCB;
TaskHandle_t LineTrackingTaskTCB;


/*User Task for Line Tracking*/
void LineTrackingTask(void *pvPara)
{
  /*Setup for the task*/

  /*DO*/
  while(true){



    LineTracking::FollowingLine(IR::Tracking(), LeftSpeed, RightSpeed);

    /*A delay must be added inside each User Task*/
    vTaskDelay(10);
  }
}
```

Similar to the code in Lab 6, the function LineTracking::FollowingLine() is changed to have LeftSpeed and RightSpeed to achieve independent speed control for left and right wheel.

Reminder: the line vTaskDelay(10); is very essential, do not delete it.

PID Control

A general PID function is shown below:

```
/*DO NOT Chnage the code below*/
/*-------------------------------------------------------------------------------------------------------------------*/
/*PID Control for Motor
  PID Function and Settings
  Creating PID_t for Mult. PID Setting
  PID 1 for LeftWheel
  PID 2 for RightWheel */
struct PID_t
{
    /*Creating the parameters for PID*/
    volatile float Kp ;
    volatile float Ki ;
    volatile float Kd ;

    volatile float target_val;  // The target RPM
    float actual_val;           // Actual RPM Reading
    float err;                  // Error
    float err_last;
    float integral;

    /*General PID Function*/
    float PID_realize(float temp_val)
    {
        this->err = this-> target_val - temp_val;

        this->integral += this->err;

        this->actual_val = this->Kp * this->err + this->Ki * this->integral + this->Kd * (this->err - this->err_last);

        this->err_last = this->err;

        return this->actual_val;
    }

}PID;

/*-------------------------------------------------------------------------------------------------------------------*/
```

Introduction of Struct and Class

For C++ beginner:

In order to store and call the PID function in different settings, i.e. Left Wheel and Right Wheel, a struct is created. A “PID_t Variable” contains {Kp, Ki, Kd … target_val … integral} with a function PID_realize().

```
/*Define 2 sets PID for 2 Motors*/
RPMCounter_t TargetRPM;
/*Change the PID Para. here
  LeftMotor PID*/
  PID_t pid1 = {10.0f   //Kp
              , 0.3f    //Ki
              , 0.0f}; //Kd

/*RightMotor PID*/
  PID_t pid2 = {8.0f    //Kp
              , 0.5f    //Ki
              , 0.0f}; //Kd
```

To define a new PID_t Variable pid1 and pid2 for 2 wheels.

```
/*------------------------------------------------------*/
    /*Setting the actual value to PID function*/
    pid1.actual_val = LeftMotor.RPMCounter();
    pid2.actual_val = RightMotor.RPMCounter();

    /*Compute the PID and Write the Result to Speed of the Wheel*/
    LeftSpeed  = Motor::RPMtoPWM(pid1.PID_realize(LeftMotor.RPMCounter()), LeftWheel);
    RightSpeed = Motor::RPMtoPWM(pid1.PID_realize(RightMotor.RPMCounter()), RightWheel);

/*------------------------------------------------------*/
    /*FOR DEBUG USAGE*/
```

When we want to call the PID with a specific setting such as pid1, we can directly call pid1.PID_realize() When the function is called, all the parameters used is under pid1 (such as pid1.Ki, pid1.Kd ....)

Creating struct can simply the coding and reduce the complexity for defining variables.

RPM Counter

```
180    */
181   typedef struct RPMCounter_t{
182
183   volatile int encoderPulses;
184   unsigned long previousMillis;
185   volatile float rpm;
186
187
188     float RPMCounter(){
189     unsigned long currentMillis = millis();
190
191     // Check if the time interval has elapsed
192     if (currentMillis - previousMillis >= interval) {
193       // Calculate RPM
194       float rotations = float(encoderPulses) / ((float) encoderResolution);
195       float time = (currentMillis - previousMillis) / 100.0f; // Convert to seconds
196       float rpm = (rotations / time) * 60.0f;
197
198       // Reset encoder pulse count and update previousMillis
199       encoderPulses = 0;
200       previousMillis = currentMillis;
201
202       // Print RPM
203       // Serial.println(rpm);
204       vTaskDelay(100/ portTICK_PERIOD_MS);  // Delay for 0.1 second
205       return rpm;
206     }
207
208   }
209   } RPM;
210
```

Similar to the approach in PID_t, RPMCounter_t is created for 2 encoders (Left & Right Motor encoder).

The function counts the Encoder Pulses and find the RPM of the motor. In RPMCounter(), the period of counting the pulses is 100ms (0.1s)

The equation of finding the RPM is shown below:

```
float rotations = float(encoderPulses) / ((float) encoderResolution);
float time = (currentMillis - previousMillis) / 1000.0f; // Convert to seconds
float rpm = (rotations / time) * 60.0f;
```

```
/*Constants for Encoder
  Find out the encoder resolution by yourself */
const int encoderResolution =320; // Number of pulses per revolution
const unsigned long interval = 1000; // Time interval in milliseconds 1000ms

/*Encoder to RPM Function and Settings
```

The encoderResolution added to test the resolution of the encoder. If we turn the Speed of Motor to Maximum (1024), the rpm should be around 600 rev/min.

To find the encoder pulse, interrupt routine function is required.

```
/*
/*Interrupt Service Routine Function
  Since attachInterrupt() cannot using non Static function
  Below are 2 IRAM_ATTR function for handle the interrupts for the encoder*/
void IRAM_ATTR handleLeftEncoderInterrupt() {
  //init the local variable
  int change = 0;

  // Read the current state of the encoder pins
  EncoderLeft.pinAState = digitalRead(EncoderLeft.Encoder_A);
  EncoderLeft.pinBState = digitalRead(EncoderLeft.Encoder_B);

  // Determine the direction of rotation based on the phase change
  if (EncoderLeft.pinAState != EncoderLeft.pinBState) {
    change = (EncoderLeft.pinAState == HIGH) ? 1 : 0;
  } else {
    change = (EncoderLeft.pinAState == HIGH) ? 0 : 1;
  }

  // Update the encoder count
  LeftMotor.encoderPulses += change;
}

void IRAM_ATTR handleRightEncoderInterrupt() {
  //init the local variable
  int change = 0;

  // Read the current state of the encoder pins
  EncoderRight.pinAState = digitalRead(EncoderRight.Encoder_A);
  EncoderRight.pinBState = digitalRead(EncoderRight.Encoder_B);

  // Determine the direction of rotation based on the phase change
  if (EncoderRight.pinAState != EncoderRight.pinBState) {
    change = (EncoderRight.pinAState == HIGH) ? 1 : 0;
  } else {
    change = (EncoderRight.pinAState == HIGH) ? 0 : 1;
  }

  // Update the encoder count
  RightMotor.encoderPulses += change;
}
```

****Since Interrupt requires the function to be Static, creating an interrupt function in a struct is impossible****

Finally, setup the Pin Mode and attach the interrupt to the pin, so that the function will be called when the pin signal is changed.

```
// Init the PinMode for the Encoder Pins
pinMode(Motor_L_Encoder_A, INPUT_PULLUP);
pinMode(Motor_L_Encoder_B, INPUT_PULLUP);

pinMode(Motor_R_Encoder_A, INPUT_PULLUP);
pinMode(Motor_R_Encoder_B, INPUT_PULLUP);

// Attach the interrupt service routine to the encoder pins
attachInterrupt(digitalPinToInterrupt(Motor_L_Encoder_A), handleLeftEncoderInterrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(Motor_R_Encoder_A), handleRightEncoderInterrupt, CHANGE);
Serial.println("Interrupt Pins Initialized");
```

Speed Control

```
void calculateRPMTask(void *pvPara) {
  /*Setup for the Task*/
  /*-------------------------------------------------------*/
  /*Define 2 sets PID for 2 Motors*/
  RPMCounter_t TargetRPM;
  /*Change the PID Para. here
    LeftMotor PID*/
    PID_t pid1 = {10.0f   //Kp
                , 0.3f   //Ki
                , 0.0f}; //Kd

  /*RightMotor PID*/
    PID_t pid2 = {8.0f   //Kp
                , 0.5f   //Ki
                , 0.0f}; //Kd

  /*Set the initial Target RPM Here*/
  pid1.target_val = 150.0f;
  pid2.target_val = 150.0f;
/*-------------------------------------------------------*/
  while (true) {
/*-------------------------------------------------------*/
    /*FOR DEBUG USAGE*/
    // Serial.print("RPM Left: ");
    // LeftMotor.RPMCounter();

    // Serial.print("RPM Right: ");
    // RightMotor.RPMCounter();
/*-------------------------------------------------------*/
    /*Setting the actual value to PID function*/
    pid1.actual_val = LeftMotor.rpm;
    pid2.actual_val = RightMotor.rpm;

    /*Compute the PID and Write the Result to Speed of the Wheel*/
    LeftSpeed  = Motor::RPMtoPWM(pid1.PID_realize(LeftMotor.rpm), LeftWheel);
    RightSpeed = Motor::RPMtoPWM(pid1.PID_realize(RightMotor.rpm), RightWheel);
```

After initializing the PID value of each PID (pid1 and pid2), set the pid.actual_val to be the reading of the RPMCounter.

Then, using the function PID_realize() to have PID control on the RPM value. Since the speed of the motor is controlled by the PWM duty cycle of the IN1 and IN2 Pin of the motor driver, a conversion between the RPM value and the PWM value is needed.

In MotorControl.cpp:

```
/*To Find the Relationship between RPM and PWM to adjust the PWM using Target RPM*/
float Motor::RPMtoPWM(float TargetRPM, uint8_t Wheel){
    float TargetPWM = 0.0f;
    /*Be Awared of 2 Motor may have a different PWM and RPM ratio*/
    switch (Wheel)
    {
    case LeftWheel:
    /*Find the math relationship
      it's not a linear relationship
      But can make the estimate value by 2 - 3 range and apply linear estimation*/
    TargetPWM = ((TargetRPM - 400.0f)/20.0f) * 60.0f ;

    if(TargetPWM > 1024.0f)
    TargetPWM = 1024.0f;

    return TargetPWM;


    case RightWheel:
    TargetPWM = TargetRPM;

    if(TargetPWM > 1024.0f)
    TargetPWM = 1024.0f;

    return TargetPWM;
    }

}
```

To find the relationship between RPM and PWM, fine testing is required. Then, a “linear relationship” can be estimated. (You may need to find different linear relationships for different ranges of Speed and RPM)

Before return the value, a limit is enabled so that the value returned to the main program will not exceed 1024.

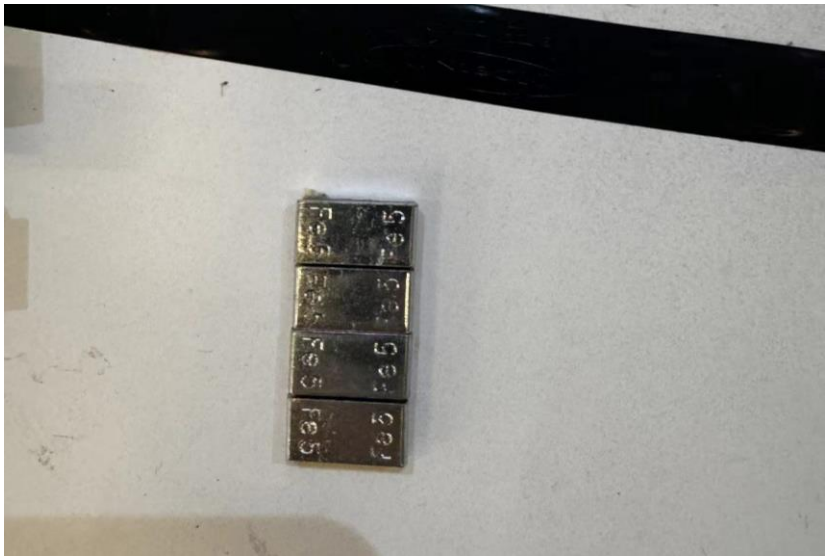
To Do List:

PID Speed Control (RPM based)

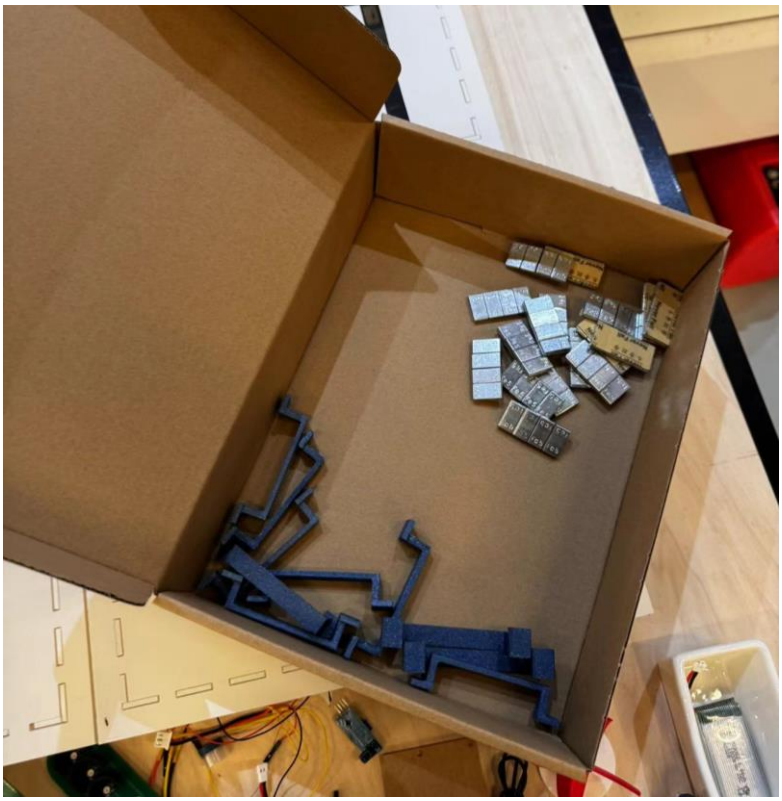
1. Tune the encoder resolution, if needed.
2. Find the relationship between RPM and PWM value for the Motor.
3. Adjust the PID value of the PID controller.
4. Save the PID parameters.
5. Adjust the Speed whenever you want to.

Additional Modification

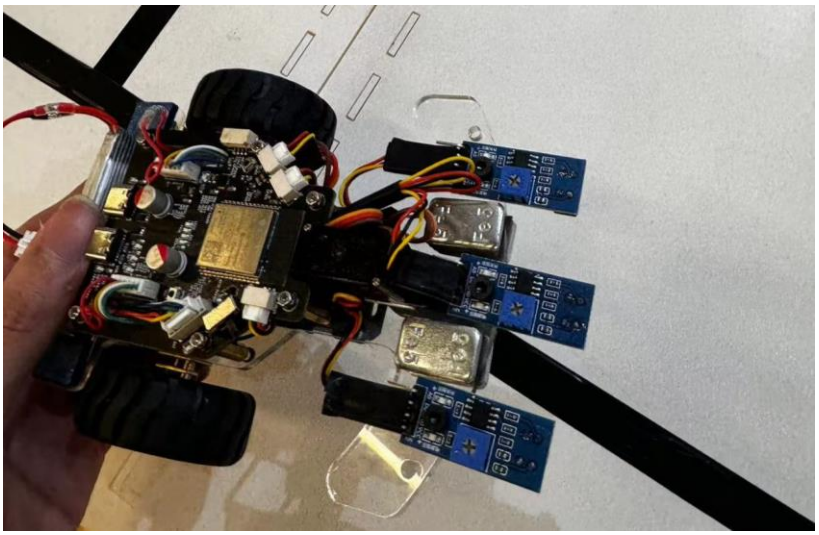
Counter Weight

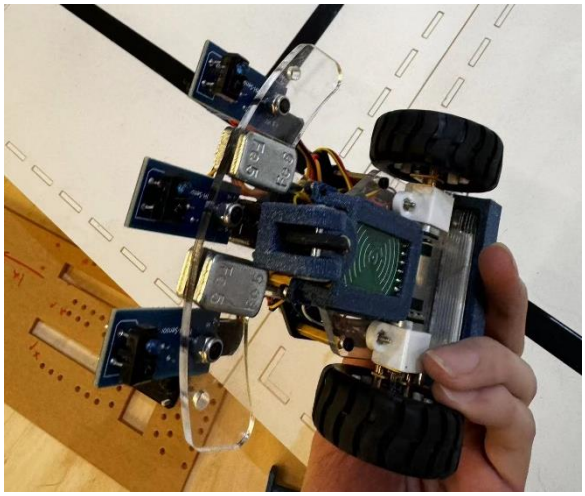


Inside the this box (same as the one used to store the new battery holder)



How to add counter Balance?





3D printed part for IR checking (for Traffic Light)

TBA