机械与能源工程系
SUSTech Department of Mechanical and Energy Engineering

项 目 报 告

项目名称：智能循迹避障小车

课程名称：嵌入式系统与机器人

课程编号：ME432

学　　号：11911022

姓　　名：沈浩瑜 江轶豪 叶玮

专　　业：机器人工程

指导教师：柯文德

2021 年 12 月 31 日

（内容编排要求：单栏排版，凡是中文均采用宋体，凡是英文均采用 Times New Roman 字体，行间距均为 1.25 倍；标题采用四号宋体，不加粗；正文采用小四号字体，不加粗；图表居中，其文字及标题采用五号字体，不加粗，图的标题在图的正下方，表的标题在表的正上方；每组撰写一份项目报告书，总页数不少于 25 页）

目　录

一、 研究目标

项目内容为利用给定组件、传感器以及开发板组装的小车完成循迹任务以及避障任务。循迹判断需要从指定点开始，使用循迹传感器形式，并停止在指定区域内，并计时。避障任务需要从指定点开始，在白色墙面范围内使用超声波传感器进行避障行驶，到达红黄蓝色板区域时使用颜色传感器进行判断并停止在红色色板区域内（三块色板颜色随机），并计时。

二、 研究背景

嵌入式系统与通用计算机系统相对应，有软件与硬件组成，且相互紧密集成。随着信息时代的高速发展，嵌入式系统已经走入我们的生活中。在嵌入式系统与机器人课程中，我们系统学习了嵌入式系统的总体概念，并以 stm32 系统为典型进行了嵌入式系统编程的实例训练。项目在此基础上，以智能小车为主题，契合了当下蓬勃发展的无人车主题，综合锻炼了嵌入式系统编程能力。

三、基本原理及方法

（一）循迹传感器以及电机控制

（1）循迹传感器调试

五路循迹传感器存在 5 个输出 pin 口，根据检测到黑线与否输出高低电平。当检测到黑线时输出低电平，检测到白线时输出高电平。为不与其他 IO 口冲突，设置 PB13 读取左侧第一个传感器的电平，PC6 读取左边第二个传感器的电平，PB14 读取中间传感器的电平，PC7 读取右侧第二个传感器的电平，PB15 读取右侧第一个传感器的电平，并在每次循环中更新。初始化中设置为 input 模式。

```
/*Configure GPIO pins : PB13 PB14 PB15 循迹传感器*/
  GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

```
7.
8.
9.  /*Configure GPIO pins : PC6 PC7 循迹传感器*/
10.   GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
11.   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
12.   GPIO_InitStruct.Pull = GPIO_PULLDOWN;
13.   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
14.   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

3.1.1 循迹传感器初始化

```
1. while (1)
2.   {
3.     /* USER CODE END WHILE */
4.         flag = 0;
5.         count = 0;
6. //       printf("while loop is running!\r\n");
7.         HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_1);
8.
9.         for(int i=0; i<3; i++)
10.         {
11.             if(i==0)
12.                 printf("RGB = (");
13.             if(i==2)
14.                 printf("%d)\r\n",(int) (cnt[i]*RGB_Scale[i]));
15.             else
16.                 printf("%d, ", (int) (cnt[i]*RGB_Scale[i]));
17.         }
18.         KEY1 = HAL_GPIO_ReadPin(GPIOB,GPIO_PIN_13);//左 1
19.         KEY2 = HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_6);//左 2
20.         KEY3 = HAL_GPIO_ReadPin(GPIOB,GPIO_PIN_14);//中
21.         KEY4 = HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_7);//右 2
22.         KEY5 = HAL_GPIO_ReadPin(GPIOB,GPIO_PIN_15);//右 1
23.         printf("%d, %d, %d, %d, %d\r\n",KEY1, KEY2, KEY3, KEY4, KEY5);

24.         Sonic_Trig(20);
```

3.1.2 反复读取电平

（2）循迹策略

```
1. if(KEY1 !=0 & KEY2 !=0 & KEY3 == 0 & KEY4 != 0 & KEY5 !=0){//中间灯检测
   到黑线,其余检测到白线，最正中情况
2.             CarGo();
3.         }
```

```
4.          else if(KEY1 !=0 & KEY2 ==0 & KEY3 == 0 & KEY4 != 0 & KEY5 !=0)
    {//左二和中间灯检测到黑线,其余检测到白线
5.              CarGo();
6.          }
7.          else if(KEY1 !=0 & KEY2 !=0 & KEY3 == 0 & KEY4 == 0 & KEY5 !=0)
    {//右二和中间灯检测到黑线,其余检测到白线
8.              CarGo();
9.          }
10.          else if(KEY1 !=0 & KEY2 !=0 & KEY3 == 0 & KEY4 == 0 & KEY5 ==
    0){//左边两灯检测到白线，中间，右 2 和右 1 检测到黑线
11.              TurnRightB();//大右转
12.          }
13.          else if(KEY1 !=0 & KEY2 !=0 & KEY3 != 0 & KEY4 == 0 & KEY5 ==
    0){//左边两灯以及中间检测到白线，右 2 和右 1 检测到黑线
14.              TurnRightB();//大右转
15.          }
16.          else if(KEY1 != 0 & KEY2!=0 & KEY3 !=0 & KEY4 ==0 & KEY5 !=0)
    {//左侧三灯以及右 1 检测到白线，右 2 灯检测到黑线
17.              TurnRight();//小右转
18.          }
19.          else if(KEY1 != 0 & KEY2!=0 & KEY3 !=0 & KEY4 !=0 & KEY5 ==0)
    {//左侧四灯检测到白线，右侧两灯检测到黑线
20.              TurnRightB();//大右转
21.          }
22.          else if (KEY1==0 & KEY2 ==0 & KEY3 == 0 & KEY4 != 0 & KEY5 !=
    0){//右边两灯检测到白线，中间以及左 2，左 1 检测到黑线
23.              TurnLeftB();//大左转
24.          }
25.          else if(KEY1==0 & KEY2 ==0 & KEY3 != 0 & KEY4 != 0 & KEY5 !=0)
    {//左 1 左 2 检测到黑线，其余检测到白线
26.              TurnLeftB();//大左转
27.          }
28.          else if (KEY1 !=0 & KEY2 ==0 & KEY3 != 0 & KEY4 != 0 & KEY5 !
    =0){//左二检测到黑线，其余检测到白线
29.              TurnLeft();//小左转
30.          }
31.          else if (KEY1 ==0 & KEY2 !=0 & KEY3 != 0 & KEY4 != 0 & KEY5 !
    =0){//左 1 检测到黑线，其余检测到白线
32.              TurnLeftB();//大左转
33.          }
34.          else if(KEY1 !=0 & KEY2 !=0 & KEY3 != 0 & KEY4 != 0 & KEY5 !=
    0){//全部检测为白色
35.              GoBack();//倒车
36.          }
```

```
        else if(KEY1 ==0 & KEY2 ==0 & KEY3 == 0 & KEY4 == 0 & KEY5 == 0){//检测到全为黑线
            black_cnt += 1;
            if(black_cnt >= 16){//通过黑线计数大于临界值停车
                Stop();
            }
            else{
                CarGo();
            }
        }
        else{
            Stop();
        }
```

3.1.3 情况分类

在循迹任务中，我们需要小车循着黑线到达终点。实际测得，黑线宽度最多容纳两个传感器检测到黑线。故设置在左耳与中间，仅中间以及中间与右二传感器检测到黑线这三种情况直行。直行速度 pwm 为 2250。

整个循线场地有小弧度以及大弧度的弯交错，为应对不同情况的转弯需求，设置了小车不同速度的转弯。

左侧三灯以及右 1 检测到白线，右 2 灯检测到黑线时，进行小右转；左边两灯检测到白线，中间，右 2 和右 1 检测到黑线，左边两灯以及中间检测到白线，右 2 和右 1 检测到黑线以及左侧四灯检测到白线，右侧两灯检测到黑线这三种情况时小车进行大右转。

左二检测到黑线，其余检测到白线时，小车进行小左转。右边两灯检测到白线，中间以及左 2，左 1 检测到黑线，左 1 左 2 检测到黑线，其余检测到白线以及左 1 检测到黑线，其余检测到白线这三种情况时进行大左转。

```
void TurnLeft(){//循迹用左转
    Motor_Rotate(1,1950,600); //右前轮快转
    Motor_Rotate(2,1550,600); //左前轮慢转
    Motor_Rotate(4,1550,600); //左后轮慢转
    Motor_Rotate(5,1950,600); //右后轮快转
}
void TurnLeftB(){//循迹用大左转
    Motor_Rotate(1,2200,600); //右前轮快转
    Motor_Rotate(2,1550,600); //左前轮慢转
     Motor_Rotate(4,1550,600); //左后轮慢转
     Motor_Rotate(5,2200,600); //右后轮快转
}

void TurnRight() {//循迹用右转
    Motor_Rotate(1, 1550, 500); //右前轮慢转
    Motor_Rotate(2, 1950, 500); //左前轮快转
    Motor_Rotate(4, 1950, 500); //左后轮快转
    Motor_Rotate(5, 1550, 500); //右后轮慢转
```

```
}

void TurnRightB() {//循迹用大右转
    Motor_Rotate(1, 1550, 500); //右前轮慢转
    Motor_Rotate(2, 2200, 500); //左前轮快转
    Motor_Rotate(4, 2200, 500); //左后轮快转
    Motor_Rotate(5, 1550, 500); //右后轮慢转
}
```

3.1.4 转弯参数

由于在车速过快时小车可能来不及转弯冲出赛道导致循迹失败，故设置当五个传感器都检测到白色时倒车。

循迹最后需要停在黑色区块中，由于赛道中存在黑线交叉，为防止提前停止，设置了全局变量 black_cnt 在五个传感器检测到黑线时计数，达到一定计数后停止，否则直行。

```
else if(KEY1 ==0 & KEY2 ==0 & KEY3 == 0 & KEY4 == 0 & KEY5 ==0){//检测到全为黑线
            black_cnt += 1;
            if(black_cnt >= 16){//通过黑线计数大于临界值停车
                Stop();
            }
            else{
                CarGo();
            }
        }
         else{
             Stop();
         }
```

3.1.5 停车策略

（3）电机控制

整车电机 ID 为右前轮为 1，左前轮为 2，左后轮为 4，右后轮为 5。为方便控制以及运动学分析，通过修改接线使四个电机转向一致。同时基于参考代码的 motor_rotate()函数编写了让任意 ID 电机以一定 pwm 速度一直转动的函数，方便了电机的连续控制。

根据驱动板原理图可知，通过串口发送#idPpwmTtime!形式的字符串即可实现对各个 ID 电机的转速以及转动时间的控制。在该处我们使用串口二发送命令，初始化串口二波特率为 115200 以实现通信。

```
void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */
```

```
9.
10.   /* USER CODE END USART1_Init 1 */
11.   huart2.Instance = USART2;
12.   huart2.Init.BaudRate = 115200;
13.   huart2.Init.WordLength = UART_WORDLENGTH_8B;
14.   huart2.Init.StopBits = UART_STOPBITS_1;
15.   huart2.Init.Parity = UART_PARITY_NONE;
16.   huart2.Init.Mode = UART_MODE_TX_RX;
17.   huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
18.   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
19.   huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
20.   huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
21.   if (HAL_UART_Init(&huart2) != HAL_OK)
22.   {
23.     Error_Handler();
24.   }
25.   /* USER CODE BEGIN USART2_Init 2 */
26.
27.   /* USER CODE END USART2_Init 2 */
28.
29. }
```

3.1.6 串口二初始化

3.1.7 电机串口控制原理图

在 Motor_Rotate(int id, int pwm, int time)中，我们依次对电机 id, 转速 pwm 和转动时间 time 进行拼接，并完善补零逻辑。同时通过串口一发送控制信息到 PC 端供调试。

```
1. void Motor_Rotate(int id, int pwm, int time)
2. {
```

```
3.          char ID_str[10] = {};
4.          char PWM_str[10] = {};
5.          char TIME_str[10] = {};
6.
7.          char data[50] = {"#"};      //data 数组，用于发送控制指令,按照协议，
    开头是"#"
8.
9.          //将整形数的 ID 号转换为字符串
10.          myitoa(id, ID_str, 10);
11.
12.          //在 ID 号的字符串的前方补零，如果有必要的话
13.          if( (3 - strlen(ID_str)) == 2) {
14.              strcat(data,"00");
15.          }
16.          else if( (3 - strlen(ID_str)) == 1) {
17.              strcat(data,"0");
18.          }
19.          //字符串拼接
20.          strcat(data, ID_str);
21.          //将整形数的 PWM 脉宽值转换为字符串
22.          myitoa(pwm,PWM_str,10);
23.
24.          //在 PWM 脉宽的字符串的前方补零，如果有必要的话
25.          if( (4 - strlen(PWM_str)) == 1) {
26.              strcat(data,"P0");
27.          }
28.          else if( (4 - strlen(PWM_str)) == 0) {
29.              strcat(data,"P");
30.          }
31.          //字符串拼接
32.          strcat(data, PWM_str);
33.          //将整形数的转动时间转换为字符串
34.          myitoa(time, TIME_str, 10);
35.
36.          //在转动时间的字符串的前方补零，如果有必要的话
37.          if( (4 - strlen(TIME_str)) == 2) {
38.              strcat(data,"T00");
39.          }
40.          else if( (4 - strlen(TIME_str)) == 1) {
41.              strcat(data,"T0");
42.          }
43.          else if( (4 - strlen(TIME_str)) == 0) {
44.              strcat(data,"T");
45.          }
```

```
46.
47.         //字符串拼接
48.         strcat(data, TIME_str);
49.         strcat(data,"!\r\n");        //末端添加"\r\n"，使得指令输出到 PC 端时，每一条指令仅占 1 行，单纯为了方便观察和调试。
50.
51.
52.         //串口 2 发送电机控制指令
53.         UART2_Send((uint8_t *)data, sizeof(data));
54.         //将同样的控制指令通过串口 1，输出给 PC 端，以便调试
55.         printf("%s", data);
56.
57.
58. }
```

3.1.8 motor_rotate

由于之前完善的补零逻辑无法发送 time 为“0000”的字符串，故在 Motor_Rotate()的基础上编写专用于发送 time 为“0000”的命令实现持续转动。

```
1. void Motor_GO(int id, int pwm)
2. {
3.     char ID_str[10] = {};
4.     char PWM_str[10] = {};
5.     char TIME_str[10] = {};
6.     char data[50] = {"#"};       //data 数组，用于发送控制指令,按照协议，开头是"#"
7.     //将整形数的 ID 号转换为字符串
8.     myitoa(id, ID_str, 10);
9.     //在 ID 号的字符串的前方补零，如果有必要的话
10.     if( (3 - strlen(ID_str)) == 2) {
11.         strcat(data,"00");
12.     }
13.     else if( (3 - strlen(ID_str)) == 1) {
14.         strcat(data,"0");
15.     }
16.     //字符串拼接
17.     strcat(data, ID_str);
18.     //将整形数的 PWM 脉宽值转换为字符串
19.     myitoa(pwm,PWM_str,10);
20.     //在 PWM 脉宽的字符串的前方补零，如果有必要的话
21.     if( (4 - strlen(PWM_str)) == 1) {
22.         strcat(data,"P0");
23.     }
24.     else if( (4 - strlen(PWM_str)) == 0) {
25.         strcat(data,"P");
26.     }
```

```
27.     //字符串拼接
28.     strcat(data, PWM_str);
29.     //在转动时间的字符串的前方补零，如果有必要的话
30.     strcat(data,"T0000");
31.     //字符串拼接
32.     strcat(data, TIME_str);
33.     strcat(data,"!\r\n");        //末端添加"\r\n"，使得指令输出到 PC 端时，
    每一条指令仅占 1 行，单纯为了方便观察和调试。
34.     //串口 2 发送电机控制指令
35.     UART2_Send((uint8_t *)data, sizeof(data));
36.     //将同样的控制指令通过串口 1，输出给 PC 端，以便调试
37.     printf("%s", data);
38.     HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_1);
39. }
```

3.1.9 motor_go

（二） 颜色传感器的使用及其颜色识别策略

1．颜色传感器的配置与使用

（1）颜色传感器 TCS3200 的基本介绍

本次项目我们的四轮小车颜色识别采用的是 TCS3200 颜色传感器，其中，该类传感器包括一块 TAOS-TCS3200RGB 感应芯片和 4 个白光 LED 灯。该类传感器能在一定的范围内检测和测量几乎所有频段的可见光。

TCS3200D 有可编程的彩色光到电信号频率的转换器，当被测物体反射光的红、绿、蓝三色光线分别透过相应滤波器到达 TAOS-TCS3200RGB 感应芯片时，其内置的振荡器会输出方波，方波频率与所感应的光强成比例关系，光线越强，内置的振荡器方波频率越高。TCS3200 传感器有一个 OUT 引脚，它输出信号的频率与内置振荡器的频率也成比例关系，它们的比率因子可以靠其引脚 S0 和 S1 的高低电平来选择，如下图：

S0	S1	OUTPUT FREQUENCY SCALING(f0)
L	L	Power down
L	H	2%
H	L	20%
H	H	100%

3.2.1 S0 与 S1 和输出频率比例

当 S0 处于低电平，S1 处于高电平，输出信号的频率与内置振荡器的频率的比例为 2%；当 S0 处于高电平，S1 处于低电平，输出信号的频率与内置振荡器

的频率的比例为 20%；当 S0 处于高电平，S1 处于高电平，输出信号的频率与内置振荡器的频率的比例为 100%；当 S0 处于低电平，S1 处于低电平，则颜色传感器停止工作。

通常所看到的物体颜色，实际上是物体表面反射出的部分有色光在人眼中的反应。白色是由各种频率的可见光混合在一起构成的，也就是说白光中包含着各种颜色的色光(如红 R、黄 Y、绿 G、青 V、蓝 B、紫 P)。根据德国物理学家赫姆霍兹(Helinholtz)的三原色理论可知，各种颜色是由不同比例的三原色(红、绿、蓝)混合而成的。由三原色感应原理可知，如果知道构成各种颜色的三原色的值，就能够知道所测试物体的颜色。对于 TCS3200 来说，当选定一个颜色滤波器时，它只允许某种特定的原色通过，阻止其它原色的通过。例如：当选择红色滤波器时，入射光中只有红色可以通过，蓝色和绿色都被阻止，这样就可以得到红色光的光强；同理，选择其它的滤波器，就可以得到蓝色光和绿色光的光强。通过这三个光强值，就可以分析出反射到 TCS3200 传感器上的光的颜色。TCS3200 传感器有红绿蓝和清除 4 种滤光器，可以通过其引脚 S2 和 S3 的高低电平来选择滤波器模式，如下图：

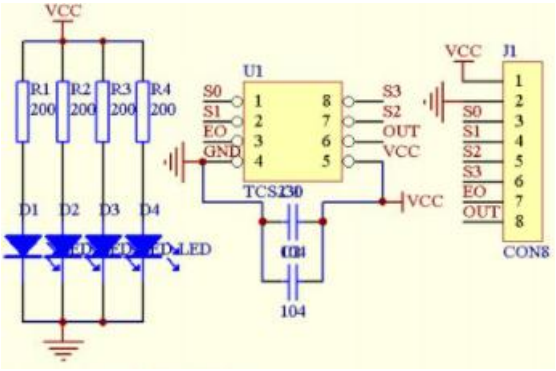
S2	S3	PHOTODIODE TYPE
L	L	Red
L	H	Blue
H	L	Clear (no filter)
H	H	Green

3.2.2 S2 和 S3 与颜色传感器滤波器模式的关系

当 S2 处于低电平，S3 处于低电平，此时的滤波器为红色滤光器；当 S2 处于低电平，S3 处于高电平，此时的滤波器为蓝色滤光器；当 S2 处于高电平，S3 处于高电平，此时的滤波器为绿色滤光器；当 S2 处于高电平，S3 处于低电平，此时的滤波器不工作。

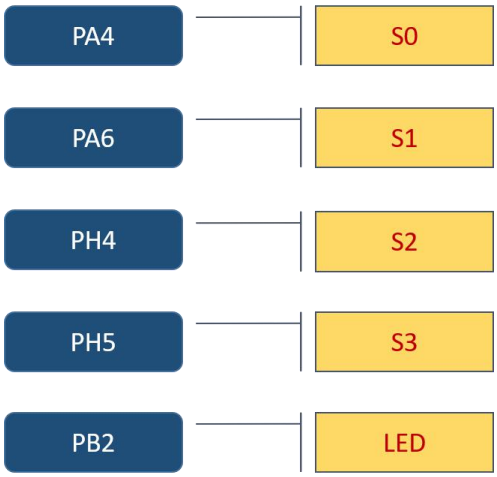
（2）颜色传感器 TCS3200 开发板上的配置

我们项目开发板的 MCU 采用的是 STM32F767IGTx 意法半导体 MCU。为了能让颜色传感器 TCS3200 在开发板上正常工作，我们需要配置相关引脚。颜色传感器有 6 个引脚（除去 GND 和 5V），包括 S0，S1，S2，S3 与 OUT 输出引脚。下图为颜色传感器 TCS3200 的原理图：



3.2.3 颜色传感器 TCS3200 原理图

根据工作基本原理，我们需要向 S0，S1，S2，S3 与 LED 输出高或低电平，从而控制我们所需的工作模式，故关于这五个引脚，我们选择的是 GPIO_MODE_OUTPUT_PP 的工作模式，用于输出电平。考虑到其他传感器在开发板上的配置和空间的布局分配，我们选择了 PA4，PA6，PH4，PH5，PB2 作为 S0，S1，S2，S3 与 LED 的接线引脚，具体如下图所示：



3.2.4 颜色传感器引脚布置

首先，S0，S1 具体配置代码如下：

```
/*Configure GPIO pins : PA4 PA6 颜色传感器*/
GPIO_InitStruct.Pin = GPIO_PIN_4|GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

其中，在上拉/下拉电阻的选择中，由于这两个引脚既要输出高电平和低电平，作为推挽输出模式，故电阻选择 NOPULL 模式，保证输出正常。为保证节能等其他相关要求，我们选择 FREQ_LOW 的输出频率模式。

接着，S2 和 S3 具体配置代码如下：

```
/*Configure GPIO pins : PH4 PH5 颜色传感器*/
```

```
  GPIO_InitStruct.Pin = GPIO_PIN_4|GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);
```

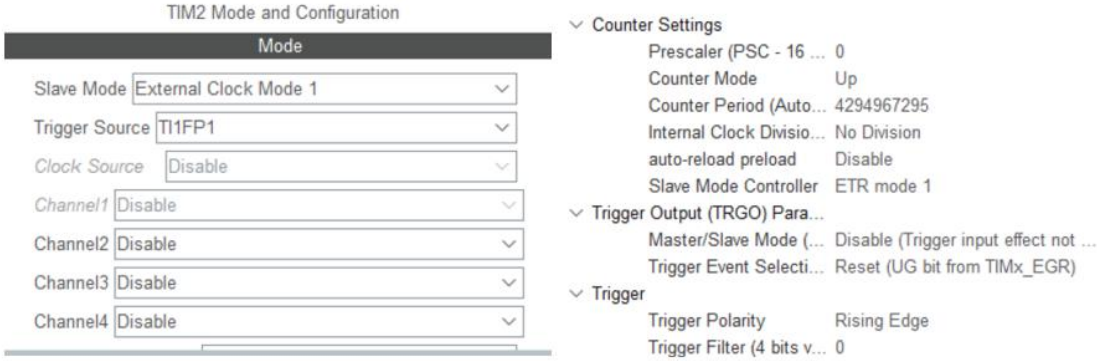
其中，在上拉/下拉电阻的选择中，由于这两个引脚既要输出高电平和低电平，作为推挽输出模式，故电阻选择 NOPULL 模式，保证输出正常。为保证节能等其他相关要求，我们选择 FREQ_LOW 的输出频率模式。

最后，LED 具体配置代码如下：

```
    GPIO_InitStruct.Pin = GPIO_PIN_2;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

其中，在上拉/下拉电阻的选择中，由于这两个引脚既要输出高电平和低电平，作为推挽输出模式，故电阻选择 NOPULL 模式，保证输出正常。为保证节能等其他相关要求，我们选择 FREQ_LOW 的输出频率模式。

基本的引脚配置完毕，最后还剩下 OUT 输出引脚的配置。OUT 的作用在于发送有确定频率的方波脉冲给开发板，因此，我们需要一个外部时钟源的定时器，来统计 TCS3200 的 OUT 发送给开发板的脉冲数，从而确定各滤波器接收下的原色光的光强度。通过查阅开发板原理图以及相关 MCU 的 datasheet，我们选择了 TIM2 作为外部时钟源的定时器，具体配置情况如下图：



3.2.5 定时器 TIM2 在 STM32CUBEMX 配置情况

代码配置如下图：

```
/* TIM2 init function */
void MX_TIM2_Init(void)
{
  TIM_SlaveConfigTypeDef sSlaveConfig = {0};
```

```
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 0xffffffff;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
  sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
  sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
  sSlaveConfig.TriggerFilter = 0;
  if (HAL_TIM_SlaveConfigSynchro(&htim2, &sSlaveConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK)
  {
    Error_Handler();
  }
}
```

根据开发板原理图，TIM2 外部时钟源所属的引脚为 PA5，故引脚配置更新为如下图所示：

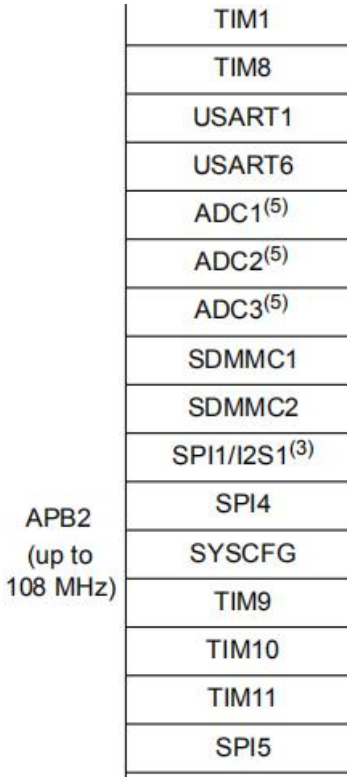
3.2.6 颜色传感器所有引脚（除 GND&5V）配置

最后，我们还需要每隔一段时间来统计 TIM2 接收的脉冲数，这就需要通过 TIM1 来实现，其中，TIM1 代码配置如下图所示：

```
/* TIM1 init function */
void MX_TIM1_Init(void)
{
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 499;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 49999;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if   (HAL_TIM_ConfigClockSource(&htim1,   &sClockSourceConfig)   !=
HAL_OK)
  {
    Error_Handler();
```

```
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) !=
HAL_OK)
    {
      Error_Handler();
    }
  }
```

通过查阅数据手册 (datasheet)资料，可以知道 TIM1 的时钟源来自 APB2 总线，此外也可以在程序中的_HAL_RCC_TIM1_CLK_ENABLE 定义里面找到，可以看到代码中的 APB2，说明 TIM1 挂载在 APB2 上。



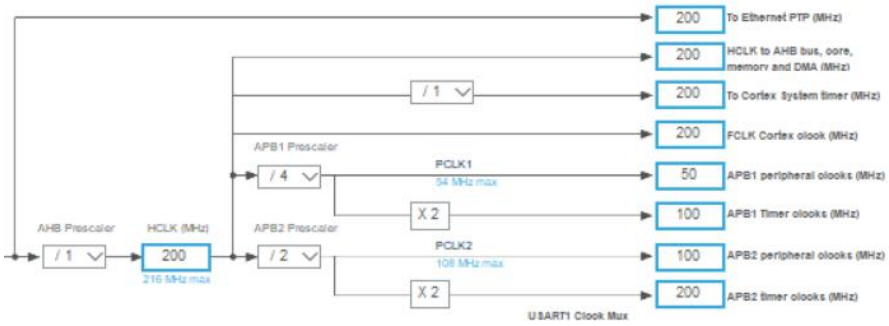
3.2.7 查阅 DataSheet 得到 TIM1 的时钟源来自 APB2 总线

相关佐证代码如下：

```
#define __HAL_RCC_TIM1_CLK_ENABLE()
do { \                                                __IO     uint32_t
tmpreg; \
                          SET_BIT(RCC->APB2ENR, RCC_APB2ENR_TIM1EN);\
```

```
                                        /* Delay after an RCC peripheral clock
enabling */ \
                                tmpreg = READ_BIT(RCC->APB2ENR, RCC_APB2ENR_TIM1EN);\
                                                UNUSED(tmpreg); \} while(0)
```

注意到在时钟树配置页面下的 APB2 Timer clocks（MHz）为 200MHz，这意味着提供给 TIM1 预分频寄存器的频率就是 200MHz。



3.2.8 CUBEMX 时钟树

根据定时器触发频率的公式：

$$f = \frac{f_0}{(Prescaler+1)(Counter\ Period+1)} = \frac{200\times10^6}{(499+1)(49999+1)}Hz$$

$$= 8Hz$$

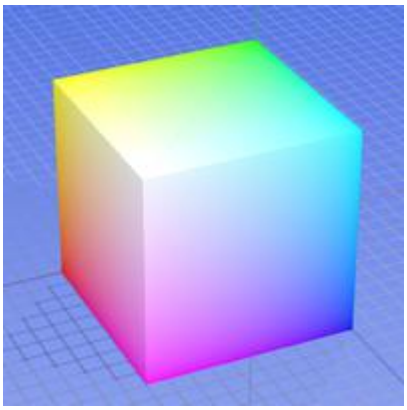
也就是说，TIM1 会以 8Hz 的频率，也就是每 0.125s 作出相应的响应，即周期中断回调，以实现统计 TIM2 中脉冲数目的目标。

2. 颜色识别策略分析

（1）颜色模型基本介绍

在当下的传统视觉领域中，颜色识别及阈值分类常常是作为视觉识别的的第一道“关卡”。在 OpenCV 中，我们常常要用到 RGB 和 HSV 两个常见的颜色模型。

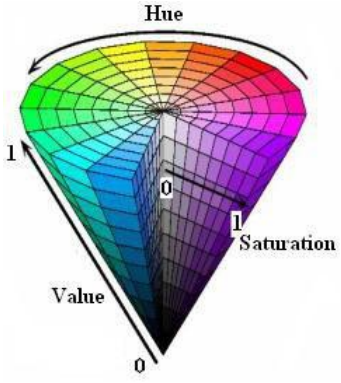
RGB 是从颜色发光的原理来设计定的，通俗点说它的颜色混合方式就好像有红、绿、蓝三盏灯，当它们的光相互叠合的时候，色彩相混，而亮度却等于两者亮度之总和，越混合亮度越高，即加法混合。红、绿、蓝三个颜色通道每种色各分为 256 阶亮度。当三色灰度数值相同时，产生不同灰度值的灰色调，即三色灰度都为 0 时，是最暗的黑色调；三色灰度都为 255 时，是最亮的白色调



3.2.9 RGB 颜色三维模型

而对于 HSV，则是更偏向于针对人眼观感的一种颜色模型，这个模型中颜色的参数分别是：色调（H, Hue），饱和度（S,Saturation），明度（V, Value）。通俗的来讲，H 是色彩；S 是深浅， S = 0 时，只有灰度；V 是明暗，表示色彩的明亮程度，但与光强无直接联系。

色调 H 用角度度量，取值范围为 0°～360°，从红色开始按逆时针方向计算，红色为 0°，绿色为 120°,蓝色为 240°。它们的补色是：黄色为 60°，青色为 180°,品红为 300°；饱和度 S 表示颜色接近光谱色的程度。一种颜色，可以看成是某种光谱色与白色混合的结果。其中光谱色所占的比例愈大，颜色接近光谱色的程度就愈高，颜色的饱和度也就愈高。饱和度高，颜色则深而艳。光谱色的白光成分为 0，饱和度达到最高。通常取值范围为 0%～100%，值越大，颜色越饱和；明度表示颜色明亮的程度，对于光源色，明度值与发光体的光亮度有关；对于物体色，此值和物体的透射比或反射比有关。通常取值范围为 0%（黑）到 100%（白）。



3.2.10 HSV 倒锥形模型

（2）方法一：利用 RGB 颜色模型转化为 HSV 颜色模型识别红色

由于 RGB 颜色系统对亮度等光照条件鲁棒性不足，在 OpenCV 中我们一般会将图像转换到 HSV 颜色系统下进行处理。为此，在本次嵌入式项目中，我们首次是尝试将得到的 RGB 值转换为 HSV 值来识别颜色。

首先是 RGB 转换为 HSV 的转换公式，定义：$\boldsymbol{R}^{'}=\frac{\boldsymbol{R}}{\boldsymbol{255}}, \boldsymbol{G}^{'}=\frac{\boldsymbol{G}}{\boldsymbol{255}}, \boldsymbol{B}^{'}=\frac{\boldsymbol{B}}{\boldsymbol{255}}, \boldsymbol{C_{max}}=\max\left(\boldsymbol{R}^{'},\boldsymbol{G}^{'},\boldsymbol{B}^{'}\right), \boldsymbol{C_{min}}=\boldsymbol{min}\left(R^{'},G^{'},B^{'}\right), \Delta=C_{max}-C_{min}$，则：

$$H=\begin{cases}0^{\circ} & ,\Delta=0\\ 60^{\circ}*\left(\dfrac{G'-B'}{\Delta}+0\right) & ,C_{\max}=R'\\ 60^{\circ}*\left(\dfrac{B'-R'}{\Delta}+2\right) & ,C_{\max}=G'\\ 60^{\circ}*\left(\dfrac{R'-G'}{\Delta}+4\right) & ,C_{\max}=B'\end{cases}$$

$$S=\begin{cases}0 & ,C_{\max}=0\\ \dfrac{\Delta}{C_{\max}} & ,C_{\max}\neq 0\end{cases}$$

$$V=C_{\max}$$

s

相关转换代码如下：

```
void rgb2hsv(uint16_t *rgb, uint16_t *hsv)
{
    float max, min, delta=0;
    float r = (rgb[0]/255);
    float g = (rgb[1]/255);
    float b = (rgb[2]/255);
    max = max3(r, g, b);
    min = min3(r, g, b);
    delta = (max - min);
    //printf("r:%f, g:%f, b:%f\n", r, g, b);
    if (delta == 0) {
        hsv[0] = 0;
    } else {
        if (r == max) {
            hsv[0] = ((g-b)/delta)*60;
```

```
            } else if (g == max) {
                hsv[0] = 120+(((b-r)/delta)*60);
            } else if (b == max) {
                hsv[0] = 240 + (((r-g)/delta)*60);
            }

            if (hsv[0] < 0) {
                hsv[0] += 360;
            }
        }
        if (max == 0) {
            hsv[1] = 0;
        } else {
           hsv[1] = (float)(delta/max);
        }
        hsv[2] = max;
        hsv[2] = (unsigned int)(((rgb[0]&0xff)<<16) | ((rgb[1]&0xff)<<8) |
(rgb[2]&0xff));
    }
```

其中，我们定义了：

```
    #define max(a,b) ((a) > (b) ? (a) : (b))
    #define min(a,b) ((a) < (b) ? (a) : (b))
    #define max3(a,b,c) (((a) > (b) ? (a) : (b)) > (c) ? ((a) > (b) ? (a) :
(b)) : (c))
    #define min3(a,b,c) (((a) < (b) ? (a) : (b)) < (c) ? ((a) < (b) ? (a) :
(b)) : (c))
```

其中，红色在 HSV 颜色空间分布的颜色阈值范围如下图：

	黑	灰	白	红		橙	黄	绿	青	蓝	紫
hmin	0	0	0	0	156	11	26	35	78	100	125
hmax	180	180	180	10	180	25	34；	77	99	124	155
smin	0	0	0	43		43	43	43	43	43	43
smax	255	43	30	255		255	255	255	255	255	255
vmin	0	46	221	46		46	46	46	46	46	46
vmax	46	220	255	255		255	255	255	255	255	255

3.2.11 HSV 颜色空间分布的颜色阈值范围

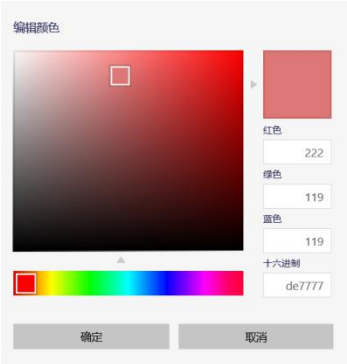
然而，在实际运用中，利用 RGB 转换 HSV 来实现颜色识别效果并不理想，经常出现颜色识别失效，小车反应滞后等问题，目前我们分析了以下几点可能的原因：

a. RGB 转换至 HSV 过程复杂，影响 while 循环和定时器的数据更新同步。
b. 颜色传感器经过白平衡之后误差依然较大，尤其是在不同光照条件下，RGB 经常会出现大于 255 的情况，影响后续的转换。
c. 转换代码本身存在某些问题，存在失真的现象。
d. HSV 阈值划分红色边界模糊，需要通过不断地调试，相比视觉来说缺少实时性，不太理想。

为此，我们选择了方法二，即直接通过 RGB 划分阈值来识别颜色。

（3）方法二：利用 RGB 颜色模型划分阈值识别红色

RGB 颜色模型其实并没有明确的特定阈值范围，但原理也更相对简单。以 Windows 自带的画图软件为例，如下图所示：



3.2.12 Win10 自带的画图 3D 软件

我们不难发现，只要 RGB 中的 R 值，大于 B 值和 G 值，就可以粗略地判断其颜色所属的区间。对于差异性特别明显的红色，经过白平衡之后的颜色传感器产生的误差也并不会明显地影响到颜色大致的判断，因此，基本的颜色逻辑判断如下：

```
if(rgb[0] > rgb[1] && rgb[0] > rgb[2] && turn_cnt >= 4){ // 当 R>B, R>G, 且小车已经转了四个弯时
            red_count = red_count+1;
            if (red_count >= 1){
                Stop();
                stop_flag = 1;
            }
        }
```

在该代码中，联系上文内容，Stop()是控制电机使得小车停下的命令函数，stop_flag 是一个标识符，当 stop_flag = 1 时，代表小车停下并且会一直停下。其中 red_count 代表的是识别到红色的次数。

因为考虑到小车从一个给定的速度到停下需要时间，并且要满足小车整个车身都在红色区域内，我们将停下的条件设置为 red_count >= 1，也就是说，根据定时器 TIM1 的中断回调函数的周期，即 0.125s 内，当位于小车中间的颜色传感器初次检测到红色（此时小车已有半个车身在红色区域），小车必须停下。这些参数包括定时器 TIM1 的配置也是通过多次测试得到的结果。

（4）白平衡

由于不同颜色的滤波器灵敏度不一样，即相同输出频率下，在相同时间内对于所用的标准白色物体，红绿蓝三种颜色滤波器输出的脉冲值不一致，会直接导致测出来的颜色 RGB 值出现较大的系统性误差。为此，先通过白平衡确定一个白色阈值，利用比例系数强行使得所有颜色滤波器提供给开发板的白色值相同是非常有必要的。

把一个白色物体放置在 TCS3200 颜色传感器之下，两者相距 1cm 左右，点亮传感器上的 4 个白光 LED 灯，然后选通三原色的滤波器，让被测物体反射光中红、绿、蓝三色光分别通过滤波器，得到 OUT 引脚输出信号的脉冲数。用白色的 RGB 标准值 255 分别除以对应颜色的脉冲值，就得到了三种颜色的比例因子。有了白平衡校正得到的 RGB 比例因子，就可以测得其他颜色的脉冲信号所换算成的 RGB 值。具体相关代码如下：

```
//选择 2%的输出比例因子
S0_L;
S1_H;
LED_ON;     //打开四个白色 LED，进行白平衡
HAL_Delay(3000);       //延时三秒，等待识别
//通过白平衡测试,计算得到白色物的 RGB 值 255 与 0.5 秒内三色光脉冲数的 RGB 比例因子
for(int i=0;i<3;i++)
{
    RGB_Scale[i] = 255.0/cnt[i];
    printf("Scaler: %5lf  \r\n", RGB_Scale[i]);
}
//红绿蓝三色光分别对应的 0.5s 内 TCS3200 输出脉冲数，乘以相应的比例因子就是我们所谓的 RGB 标准值
//打印被测物体的 RGB 值
for(int i=0; i<3; i++)
{
```

```
            printf("%d \r\n", (int) (cnt[i]*RGB_Scale[i]));
        }
        printf("White Balance Done!\r\n");
        //白平衡结束
```

其中，RGB_Scale[i] = 255.0/cnt[i]代表的就是比例因子，当我们通过传感器得到颜色 RGB 的原始值（即 cnt[i]）后，还需要根据式子 cnt[i]*RGB_Scale[i]来得到我们需要的真实测量值。

（5）颜色识别策略基本流程分析

通过上文所列出来的内容，我们制定出一套基于 TCS3200 颜色传感器的一套颜色识别方案，其中，我们根据标识符来表示颜色传感器中滤波器所属的状态，从而来执行相关操作，并在颜色识别完毕之后，更改滤波器的模式，其中，我们定义了 TCS_Next(int s2, int s3)和 filter(int s2, int s3)这两个函数，前者用于更新滤波器模式，后者用于执行滤波器操作：

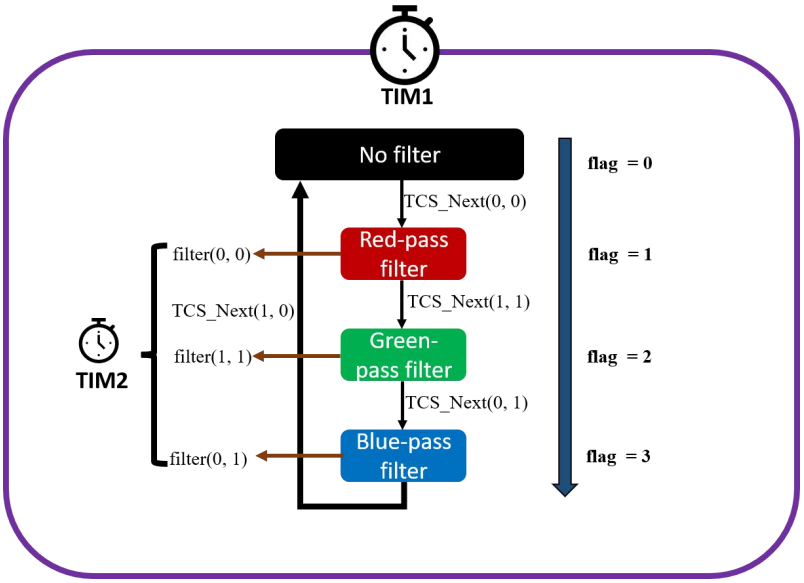
```
    void TCS_Next(int s2, int s3)
    {
        count = 0; //统计脉冲数清零
        flag++;    //输出信号计数标志+1，进行下一个颜色的脉冲统计
        filter(s2,s3);     //选择颜色滤波器模式
}
    //滤波器模式选择函数，根据 S2 和 S3 的电位来选择红、绿、蓝三种颜色的滤波器
    void filter(int s2, int s3)
    {
        if(s2 == 0 && s3 == 0){
                S2_L;S3_L;
            }
        if(s2 == 0 && s3 == 1){
                S2_L;S3_H;
            }
        if(s2 == 1 && s3 == 0){
                S2_H;S3_L;
            }
        if(s2 == 1 && s3 == 1){
                S2_H;S3_H;
            }
```

```
}
```

以及所有的#define：

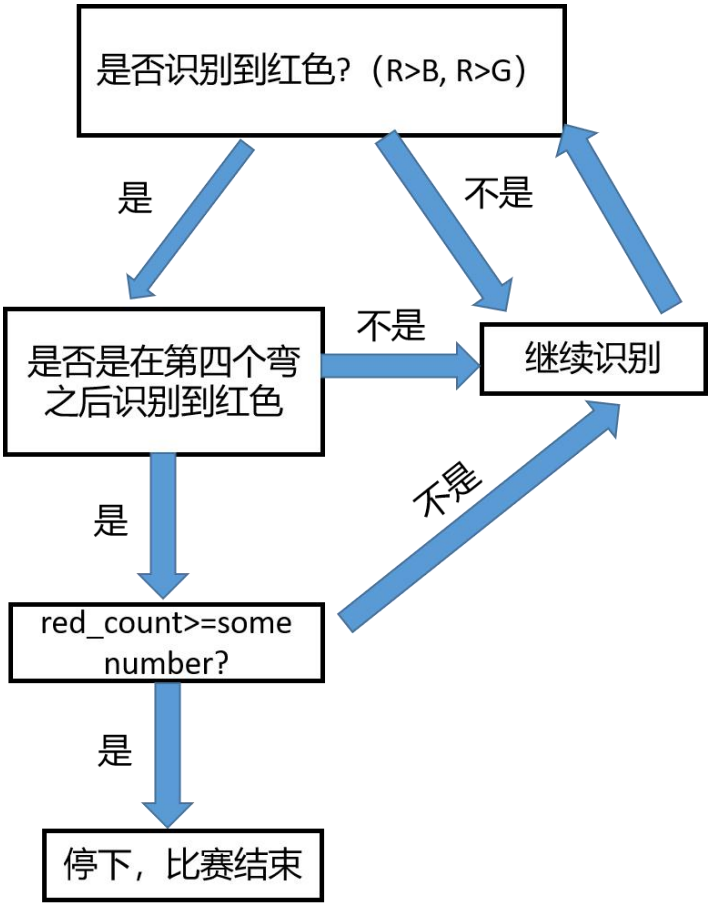
```
#define S0_L HAL_GPIO_WritePin(GPIOA,GPIO_PIN_4,0)
#define S0_H HAL_GPIO_WritePin(GPIOA,GPIO_PIN_4,1)

#define S1_L HAL_GPIO_WritePin(GPIOA,GPIO_PIN_6,0)
#define S1_H HAL_GPIO_WritePin(GPIOA,GPIO_PIN_6,1)

#define S2_L HAL_GPIO_WritePin(GPIOH,GPIO_PIN_4,0)
#define S2_H HAL_GPIO_WritePin(GPIOH,GPIO_PIN_4,1)

#define S3_L HAL_GPIO_WritePin(GPIOH,GPIO_PIN_5,0)
#define S3_H HAL_GPIO_WritePin(GPIOH,GPIO_PIN_5,1)

#define LED_OFF HAL_GPIO_WritePin(GPIOB,GPIO_PIN_2,0)
#define LED_ON  HAL_GPIO_WritePin(GPIOB,GPIO_PIN_2,1)
```

其中下图为整个颜色传感器的运作模式：



3.2.13 颜色传感器运行流程图

下图为颜色识别的逻辑导图：



颜色识别逻辑导图流程图

相比于方法一，方法二虽然也经常有比较明显的滞后现象，但方法二更加稳定，具有较大优势，故选择方法二。自此，颜色识别模块结束。

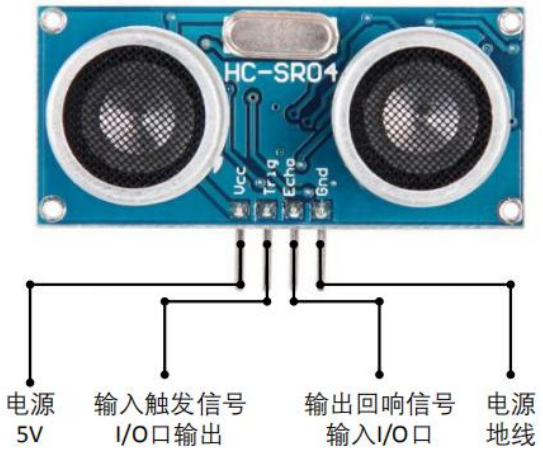
（三） 超声波测距模块的使用及避障策略

1．超声波测距模块的配置与使用

（1）超声波测距模块 HC-SR04 的基本介绍

HC-SR04 超声波测距模块可提供 2cm-400cm 的非接触式距离感测功能，测距精度可达高到 3mm；模块包括超声波发射器、接收器与控制电路。

该模块上有 VCC 电源，GND 地线，TRIG 触发控制信号输入和 ECHO 回响信号输出四个 IO 口。



3.3.1 超声波测距模块实物图

该模块的基本工作原理为以下三个步骤：

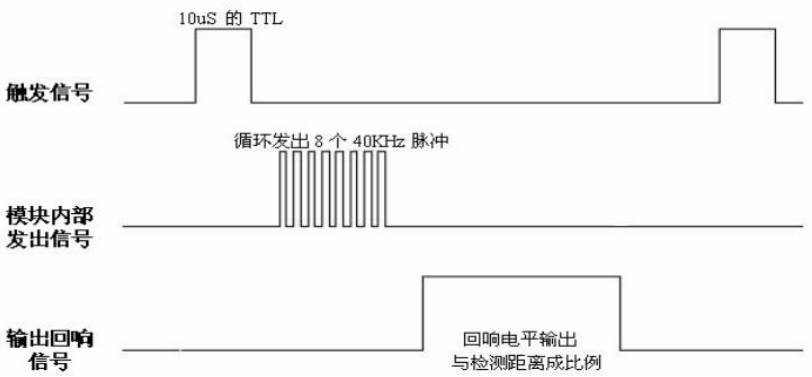
a) 单片机通过 IO 口 TRIG 向模块输入至少 10us 的高电平信号，触发测距；

b) 模块自动发送 8 个 40khz 的方波，并检测是否有信号返回；

c) 模块检测到有信号返回，则通过 IO 口 ECHO 输出一个高电平，高电平持续的时间就是超声波从发射到返回的时间。

因此，我们可以通过高电平持续时间计算出所测的距离：

$$测试距离 = \frac{高电平时间 \times 声速(340m/s)}{2}$$



3.3.2 超声波时序图

（2）超声波测距模块 HC-SR04 相关定时器的配置

在此次项目中，为使超声波测距模块正常工作，我们配置了定时器 TIM5 和 TIM6。其中，TIM5 用于输入回响信号捕获；TIM6 用于实现毫秒级延时，以便发送合适时长的触发信号。

a) 输入捕获时基参数初始化函数和输入捕获通道参数初始化函数

```
/* TIM5 init function */
void MX_TIM5_Init(void)
{
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_IC_InitTypeDef sConfigIC = {0};
  htim5.Instance = TIM5;
  htim5.Init.Prescaler = 99;
  htim5.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim5.Init.Period = 0xffffffff;
  htim5.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim5.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  HAL_TIM_Base_Init(&htim5);
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  HAL_TIM_ConfigClockSource(&htim5, &sClockSourceConfig);
  HAL_TIM_IC_Init(&htim5);

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim5, &sMasterConfig);
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  HAL_TIM_IC_ConfigChannel(&htim5, &sConfigIC, TIM_CHANNEL_1);
  HAL_TIM_IC_ConfigChannel(&htim5, &sConfigIC, TIM_CHANNEL_2);
}
```

因为使用了两个测距模块，因此我们配置了 TIM5 的两个通道，即通道 1 和通道 2。

```
/* TIM6 init function */
void MX_TIM6_Init(void)
{
  TIM_MasterConfigTypeDef sMasterConfig = {0};
```

```
    htim6.Instance = TIM6;
    htim6.Init.Prescaler = 99;
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim6.Init.Period = 65535;
    htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    HAL_TIM_Base_Init(&htim6);
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_ENABLE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig);

}
```

我们将 TIM6 的自动装载值设置为 65535，这会影响后面延时函数的延时上限。

b) 输入捕获时基参数初始化回调函数

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* tim_baseHandle)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    if(tim_baseHandle->Instance==TIM5)
  {
    /* TIM5 clock enable */
    __HAL_RCC_TIM5_CLK_ENABLE();

    /* GPIO clock enable */
    __HAL_RCC_GPIOH_CLK_ENABLE();

    /**TIM5 GPIO Configuration
    PH10      ------> TIM5_CH1
    PH11      ------> TIM5_CH2
    */

    GPIO_InitStruct.Pin = GPIO_PIN_10|GPIO_PIN_11;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF2_TIM5;
    HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);

    /* TIM5 interrupt Init */
```

```
    HAL_NVIC_SetPriority(TIM5_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(TIM5_IRQn);

  }
    else if(tim_baseHandle->Instance==TIM6)
  {
    /* TIM6 clock enable */
    __HAL_RCC_TIM6_CLK_ENABLE();

    /* TIM6 interrupt Init */
    HAL_NVIC_SetPriority(TIM6_DAC_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
  }

}
```

通过查阅 STM32F767 数据手册和考虑其他传感器在开发板上的配置和空间的布局分配，我们选择将 PH10 和 PH11 复用为 TIM5_CH1 和 TIM5_CH2。

PH10	I/O	FT	-	TIM5_CH1, I2C4_SMBA, FMC_D18, DCMI_D1, LCD_R4, EVENTOUT
PH11	I/O	FT	-	TIM5_CH2, I2C4_SCL, FMC_D19, DCMI_D2, LCD_R5, EVENTOUT

3.3.3 STM32F767 部分引脚定义

c) 定时器中断服务函数，捕获中断发生时执行

```
void TIM5_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim5);
}

void TIM6_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim6);
}
```

d) 定时器输入捕获中断处理回调函数，在 HAL_TIM_IRQHandler 中会被调用

```
uint8_t    TIM5CH1_CAPTURE_STA=0;  //通道 1 输入捕获状态
uint32_t   TIM5CH1_CAPTURE_Date1;  //通道 1 数据 1 (输入捕获值 TIM5 是 32 位)
uint32_t   TIM5CH1_CAPTURE_Date2;  //通道 1 数据 2
uint8_t    TIM5CH2_CAPTURE_STA=0;  //通道 2 输入捕获状态
uint32_t   TIM5CH2_CAPTURE_Date1;  //通道 2 数据 1
uint32_t   TIM5CH2_CAPTURE_Date2;  //通道 2 数据 2

uint16_t Ultrasonic_us1 = 0;
```

```
uint16_t Ultrasonic_us2 = 0;

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
  if(htim->Instance == TIM5){
    /* TIM_CH1 */
    if(htim->Channel==HAL_TIM_ACTIVE_CHANNEL_1) {
      if((TIM5CH1_CAPTURE_STA&0X80)==0) //还未成功捕获
      {
        if(TIM5CH1_CAPTURE_STA&0X40)      //捕获到一个下降沿
        {
          TIM5CH1_CAPTURE_STA|=0X80;      //标记成功捕获到一次高电平脉宽
          TIM5CH1_CAPTURE_Date2=
          HAL_TIM_ReadCapturedValue(&htim5,TIM_CHANNEL_1);//获取当前捕获值

          Ultrasonic_us1=TIM5CH1_CAPTURE_STA&0X3F;
          Ultrasonic_us1*=0XFFFFFFFF;                    //溢出时间总和
          Ultrasonic_us1+=TIM5CH1_CAPTURE_Date2;
          Ultrasonic_us1-=TIM5CH1_CAPTURE_Date1;         //得到总的高电平时间
          TIM5CH1_CAPTURE_Date1=0;
          TIM5CH1_CAPTURE_STA=0;              //开启下一次捕获

          TIM_RESET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_1);//清除原来的设
置
          TIM_SET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_1,
          TIM_ICPOLARITY_RISING);           //配置 TIM5 通道 1 上升沿捕获
        }else                                //还未开始,第一次捕获上升沿
        {
          TIM5CH1_CAPTURE_STA=0;            //清空
          TIM5CH1_CAPTURE_Date2=0;
          TIM5CH1_CAPTURE_STA|=0X40;      //标记捕获到了上升沿
          TIM5CH1_CAPTURE_Date1=
          HAL_TIM_ReadCapturedValue(&htim5,TIM_CHANNEL_1);//获取当前捕获值
          TIM_RESET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_1);//清除原来的设
置
          TIM_SET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_1,
          TIM_ICPOLARITY_FALLING);          //配置 TIM5 通道 1 下降沿捕获
        }
      }
    }
    /* TIM_CH2 */
    else if(htim->Channel==HAL_TIM_ACTIVE_CHANNEL_2) {
      if((TIM5CH2_CAPTURE_STA&0X80)==0)
      {
```

```
            if(TIM5CH2_CAPTURE_STA&0X40)
            {
              TIM5CH2_CAPTURE_STA|=0X80;
   TIM5CH2_CAPTURE_Date2=HAL_TIM_ReadCapturedValue(&htim5,TIM_CHANNEL_2);
              Ultrasonic_us2=TIM5CH2_CAPTURE_STA&0X3F;
              Ultrasonic_us2*=0XFFFFFFFF;
              Ultrasonic_us2+=TIM5CH2_CAPTURE_Date2;
              Ultrasonic_us2-=TIM5CH2_CAPTURE_Date1;
              TIM5CH2_CAPTURE_Date1=0;
              TIM5CH2_CAPTURE_STA=0;
              TIM_RESET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_2);
TIM_SET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_2,TIM_ICPOLARITY_RISING);
            }else
            {
              TIM5CH2_CAPTURE_STA=0;
              TIM5CH2_CAPTURE_Date2=0;
              TIM5CH2_CAPTURE_STA|=0X40;
   TIM5CH2_CAPTURE_Date1=HAL_TIM_ReadCapturedValue(&htim5,TIM_CHANNEL_2);
              TIM_RESET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_2);
TIM_SET_CAPTUREPOLARITY(&htim5,TIM_CHANNEL_2,TIM_ICPOLARITY_FALLING)
;
            }
          }
      }
      }
}
```

此部分运用了定时器作为输入捕获。具体流程在定时器输入捕获的实验报告中已有详细分析，因此在此不再赘述。

e) 使用 TIM6 实现的毫秒级延时函数

```
//定时器毫秒级延时，最大延时时间为 65535us。如果大于最大延时时间，则可能异常。
void Delay_us(uint16_t us)
{
    HAL_TIM_Base_Start(&htim6);
    __HAL_TIM_SET_COUNTER(&htim6,0);
    while(us > __HAL_TIM_GET_COUNTER(&htim6) ) {};
    HAL_TIM_Base_Stop(&htim6);

}
```

当定时器 6 计数大于需要延时的毫秒数时，关闭定时器 6。

（2）超声波测距模块 HC-SR04 相关 IO 口的配置

a) IO 初始化函数

```
void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  __HAL_RCC_GPIOH_CLK_ENABLE();

    /*Configure GPIO pins : PH3 PH2  超声波*/
    GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_2;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);

    HAL_GPIO_WritePin(GPIOH,GPIO_PIN_3,0);
    HAL_GPIO_WritePin(GPIOH,GPIO_PIN_2,0);
}
```

我们选择 PH2 和 PH3 分别作为两个测距模块输入触发信号的引脚。

b) 发送触发信号

```
void Sonic_Trig(uint16_t us)
{
        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_3,1);
        Delay_us(us);
        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_3,0);

}

void Sonic_Trig1(uint16_t us)
{
        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_2,1);
        Delay_us(us);
        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_2,0);
}
```

通过使用毫秒延时函数来发送一定时间的高电平信号。

2. 避障策略分析

（1）PID 控制介绍

在实地测试中我们发现，小车在直线行驶的过程中车身会发生一定的偏移，从而影响转弯的效果，为了解决这个问题，我们选择引入 pid 控制。

PID，即“比例、积分、微分”，是一种常见的控制算法。在工程实际中，应

用最为广泛的调节器控制规律为比例、积分、微分控制，简称 PID 控制。它以其结构简单、稳定性好、工作可靠、调整方便而成为工业控制的主要技术之一。

数字 PID 算法又分为位置式 PID 控制算法和增量式 PID 控制算法。该项目使用的是增量式 PID 控制算法。

增量式 PID 控制算法公式：

$$\Delta u = K_p(e_n - e_{n-1}) + K_i e_n + K_d((e_n - e_{n-1}) - (e_{n-1} - e_{n-2}))$$

K_p：比例系数，反应调节速度。K_p越大，系统响应速度越快，K_p越小，响应速度越慢。K_p过大会产生震荡，使系统不稳定。

K_i：积分控制，消除系统余差。

K_d：微分控制，减小系统震荡。

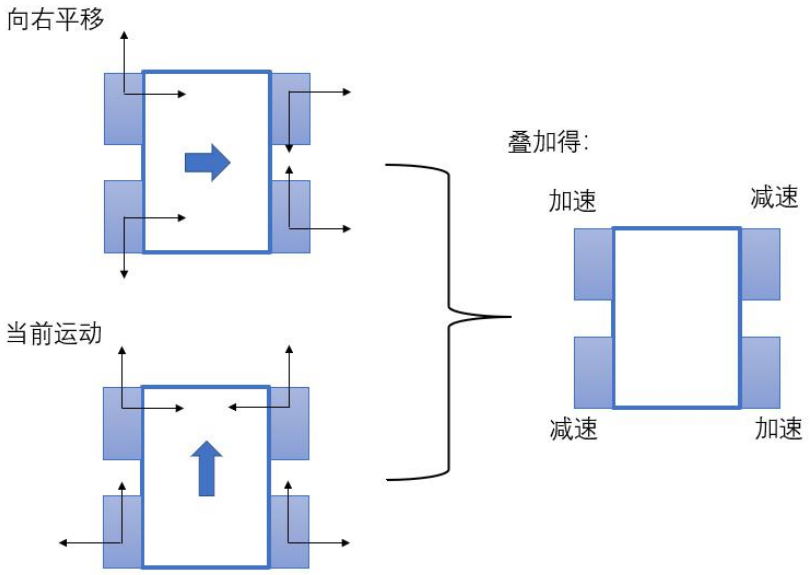
e_n：输入量。此处为车身右侧与挡板的距离和设定距离之间的误差。

e_{n-1}：上一次的输入量。

e_{n-2}：上上次的输入量。

获得Δu 之后，我们需要使用该增量改变电机转速。

假定小车向前运动，当小车偏离到中心线的左边，此时小车的速度变化应该往右平动。

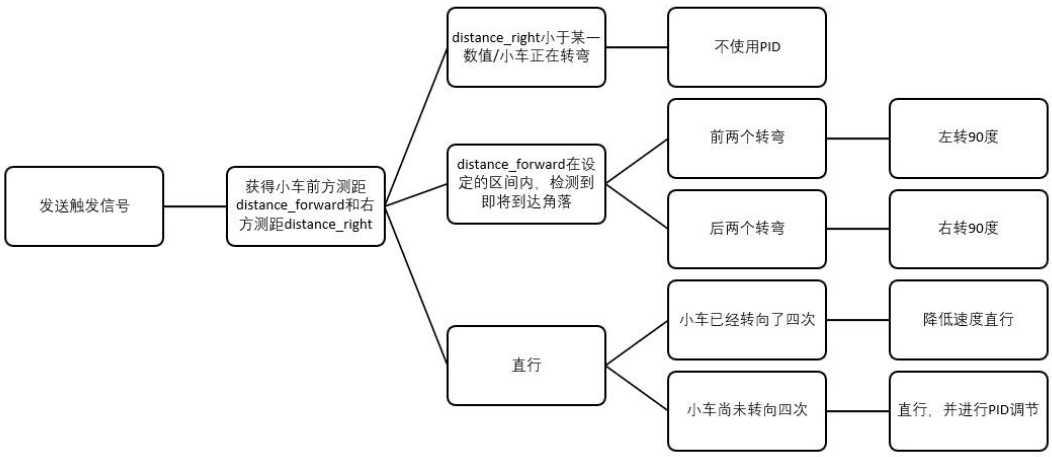


往右平动修正的运动分析

（2）PID 控制代码实现

```
float Position_PID (float Encoder,float Target)
{
    float Position_KP = 52;
    float Position_KI = 0.5;
    float Position_KD = 11;
    static float Bias,Pwm,Integral_bias,Last_Bias;
    Bias=Encoder-7.2; //计算偏差 en，期望的距离为 7.2cm
    Integral_bias+=Bias; //求出偏差的积分
    Pwm=Position_KP*Bias+Position_KI*Integral_bias+Position_KD*(Bias-Last_Bias);
    Last_Bias=Bias; //保存上一次偏差
    return Pwm; //增量输出
}
/*  在直线行驶中引入 pid */
void CarGo_PID(float Pwm){
    Motor_GO(1,Go_Vel-Pwm/5-5);//右前
    Motor_GO(2,Go_Vel+Pwm/5);   //左前
    Motor_GO(4,Go_Vel-Pwm/5);   //左后
    Motor_GO(5,Go_Vel+Pwm/5-5);//右后
}
```

（3）避障基本流程



3.3.4 避障流程图

（3）避障代码实现

相关变量：

```
float distance_forward = 0;//前方测距
float distance_right = 0;   //右方测距
int turn_flag = 0;                  // 直行为 0，左转为 1
int turn_flag1 = 0;             // 直行为 0，右转为 1
int turn_dir = 0;
int turn_cnt = 0;                   // turn counter
float PWM = 0;
int stop_flag = 0;                  //小车是否停下
```

```
/*    发送触发信号    */
Sonic_Trig(20);
Sonic_Trig1(20);

//获得测距
distance_forward = (float)(Ultrasonic_us1 * 17.0 / 1000);
distance_right = (float)(Ultrasonic_us2 * 17.0 / 1000);

//右侧距离小于 27.5cm 并且小车不在转向，计算 PID 参数 PWM
if(distance_right < 27.5 && turn_flag == 0 && turn_dir <= 3){
        PWM = Position_PID(distance_right,target);
}else{
//距离右侧挡板较远，不使用 PID
        PWM = 0;
}
if(stop_flag == 0){
    //检测到即将到达角落（前两个转向）
    if(distance_forward    <= 27.4 && distance_forward    >= 7) {
        turn_flag = 1;
    }
    //检测到即将到达角落（后两个转向）
    if(distance_forward    <= 24 && distance_forward    >= 7) {
        turn_flag1 = 1;
    }
    //前两个转向
    if(turn_flag == 1 && (turn_dir/2)%2 == 0) {
        CircleL();//原地转 90 度左转
        turn_dir = turn_dir + 1;
        turn_cnt +=1;
        Ultrasonic_us1 = 50;
        HAL_Delay(585);
        turn_flag = 0;
```

```
            turn_flag1 = 0;
        }
        //第三个转向
        else if(turn_flag1 == 1 && turn_dir == 2) {
            HAL_Delay(200);
            if(turn_dir%2==0){
            CircleR1();//原地转 90 度右转
            }
            turn_dir = turn_dir + 1;
            turn_cnt +=1;
            Ultrasonic_us1 = 50;
            Ultrasonic_us2 = 50;
            HAL_Delay(600);
            turn_flag = 0;
            turn_flag1 = 0;
        }
        //第四个转向
        else if(turn_flag1 == 1 && turn_dir == 3) {
            HAL_Delay(200);
            CircleR();//原地转 90 度右转
            turn_dir = turn_dir + 1;
            turn_cnt +=1;
            Ultrasonic_us1 = 50;
            HAL_Delay(600);
            turn_flag = 0;
            turn_flag1 = 0;
        }
        //转向了四次之后直行
        else if(turn_flag == 0 && turn_dir == 4){
            CarGo();//直线前进，速度较慢
            HAL_Delay(450);
        }
        else{
            CarGo_PID(PWM);//直线前进
            HAL_Delay(450);
        }
```

四、实验数据

最后比赛过程中在两次尝试中循迹任务获得了 18 秒的成绩，三次尝试中避障任务获得了 25 的成绩并最终识别成功，较好完成了所有项目目标。

五、实验结果分析

最终测试中循迹任务表现稳定。避障任务中受底盘一致性影响整体小车在长距离直行后整体姿态可能会产生一定的 z 轴偏移，影响后续运动，由于未加入陀

螺仪等传感器，难以及时准确地对姿态进行修正。颜色传感器存在一定延时但识别效果满足目标要求。

六、参考文献

附件：程序核心代码（此项是必须的！）

PS：建议加入思维导图或程序框图，以便解释程序逻辑、实验原理及实现方法。提交格式：pdf 格式文档，以“成员 1-成员 2-成员 3-嵌入式系统与机器人-项目报告书”命名。