任务一技术报告

目录

1.成果展示

1.1成果介绍

本次成果展示所用到的车体结构是恩智浦智能车竞赛中的 C 型车模，该车模的二轴车机械结构决定了后轴两轮与转向中心在一条直线上。单片机主控芯片型号为STM32F103RCT6，，该芯片有64引脚，72MHz主频，256kFlash，64kSRAM。

串口助手用的野火多功能调试助手，山外调试助手（用来显示波形）。蓝牙控制时，用到的软件为蓝牙调试器。

我们将运动控制时分成两种，模式0为非阿克曼运动模式，此时我们可以通过上位机发送不同信息，控制小车的转向角度和左右轮速度，模式1为阿克曼运动模式，此时我们只需要控制舵机的转向和中心线速度的大小，就能固定出小车的运动状态。发送信息为0~4共五位数据，其中位4为终止位，该位发送终止符'$'，对应位4置'\0'，其他位内容如下表。

	位0	**位1**	**位2，3**
切换运动模式	m	o	de
非阿克曼运动模式			
左轮速度	C	+/-	左轮加减速度大小，
		=	左轮更改后速度大小
右轮速度	B	+/-	左轮加减速度大小，
		=	右轮更改后速度大小
转弯角度	T	+/-	+增加右转角度，-减少左转角度
		L/R	L设置左转角度，R设置右转角度
	R	L/R	左转/右转半径
阿克曼运动模式			
	位0	位1	位2，3
中心线速度	V	+/-	加减速度大小，
		=	更改后速度大小
转弯角度	T	+/-	+增加右转角度，-减少左转角度
		L/R	L设置左转角度，R设置右转角度

1.2 串口控制下的阿克曼运动模型

1.2.1 基础任务

串口助手发送位置信息，小车做基于阿克曼转向的左转右转时驱动电机的差速控制。

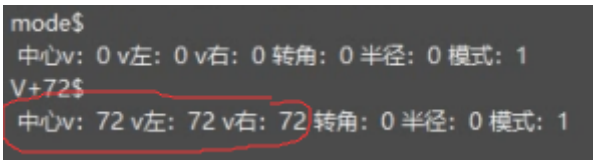
1.2.2 过程介绍

1.2.2.1 模式切换

小车的初始模式为非阿克曼运动模式，我们可以通过发送"mode$"，切换成为阿克曼模式，阿克曼运动模式下，回显多了中心速度V。

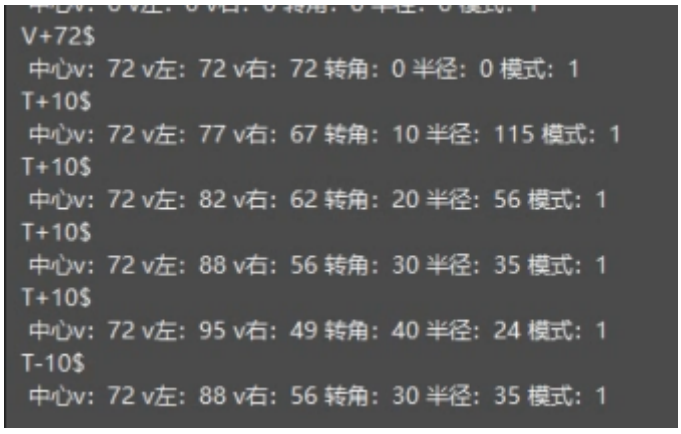
1.2.2.2 速度控制

我们通过发送“V+72$”，控制小车中心速度增加72cm/s。



1.2.2.3 转向控制

我们通过发送“T+10$”或者“T-10$”,控制小车右转或者左转10°，在此过程中，小车运动的半径，左轮右轮的速度，会按照阿克曼运动模型进行改变。



1.3 蓝牙控制下阿克曼运动模型

此为创新部分，并不包含在基础任务中。

1.3.1 蓝牙基本设置

设置如下表

按键名称	按下时发送	松开时发送
左转10	T-10$	
左转30	T-30$	
左转60	T-60$	
右转10	T+10$	
右转30	T+30$	
右转60	T+60$	
速度加72CM/S	V+72$	
速度减72CM/s	V-72$	
停	V=00$	
模式转换	mode$	
非阿克曼下加速	C+72$	B+72$

蓝牙设置页面如下

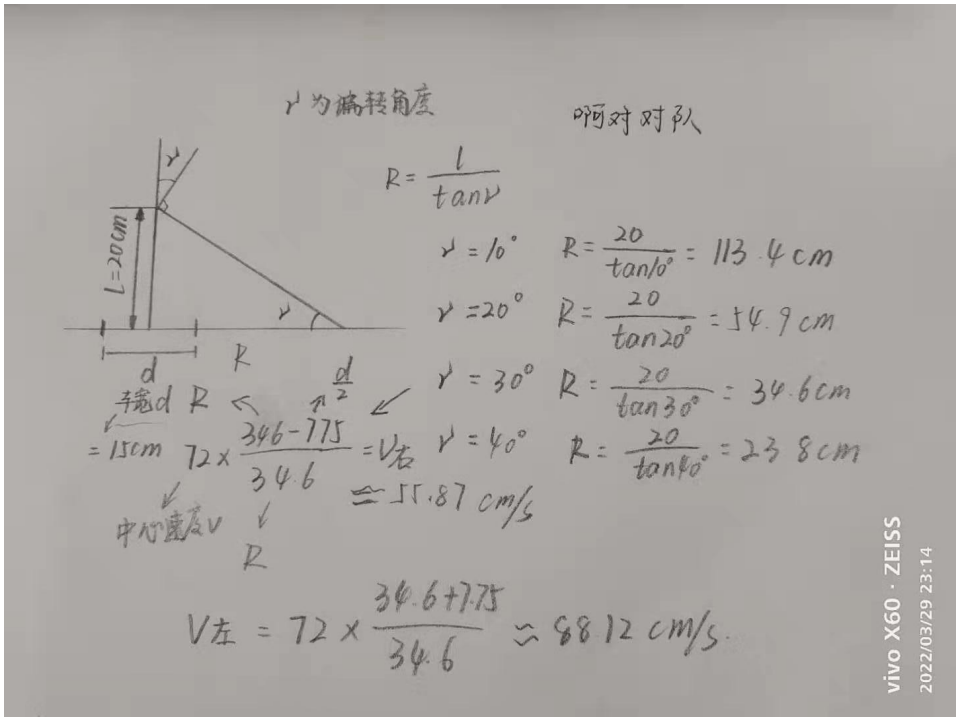

18:46
未连接设备
Tx: 0B/s
Rx: 0B/s
Err: 0B/s
未收到消息
左转10
左转30
左转60
右转10
右转30
右转60
速度加72CM/S
模式转换
停
非阿克曼下的加速
速度减72CM/S
编辑模式
设备连接
对话模式
专业调试
按钮控制
设置

其中单个按键展示

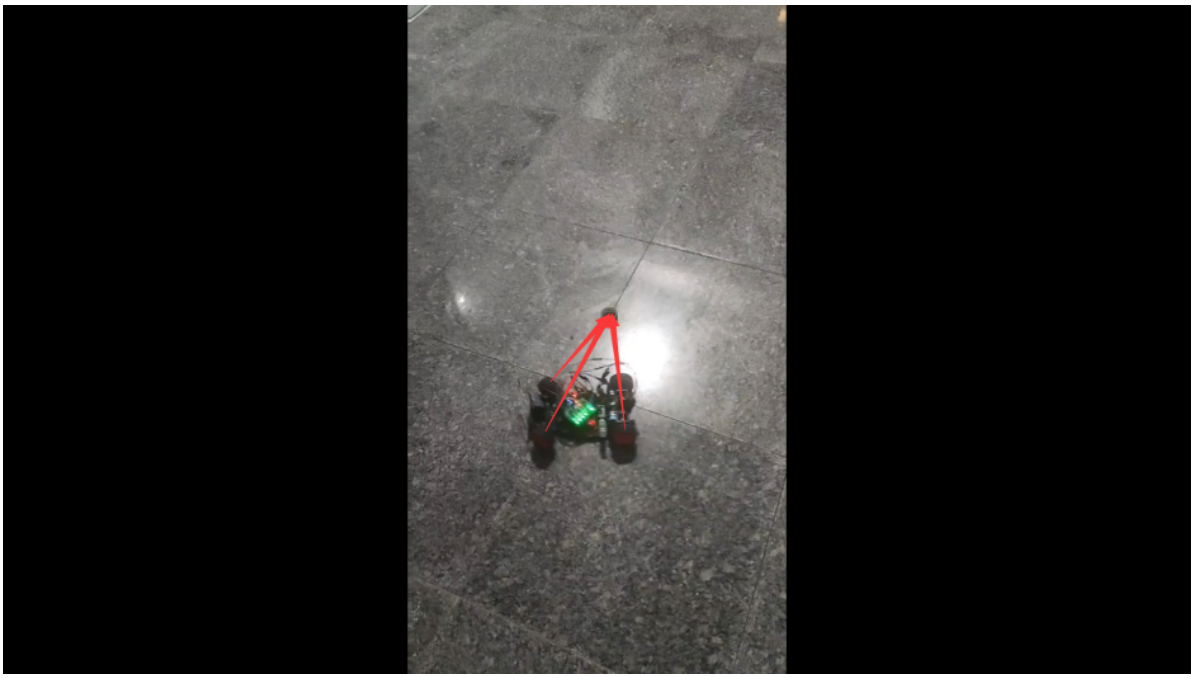
1.3.2 阿克曼转圈

视频中我们以72cm/s的中心速度，以及右转30°的实例演示了阿克曼运动，上面的串口控制阿克曼时，我们可以看到，此情况时

这与我们视频中的测量结果是相同的，半径约为35，直径约为70。同时我们通过数据的计算得出的理想化结果也是与此相同的，下图为各个数据的计算情况。



并且我们可以从视频中清楚的看到，车轮是满足阿克曼运动模型的要求，及前轮转圈的圆心集合于后轴的延长线上。



1.3.3 阿克曼转弯以及比较非阿克曼转弯

我们通过对比中心速度72cm/s,转弯30°的阿克曼转弯和左右轮都为72cm/s的非阿克曼转弯，可以明显看出非阿克曼在处理转弯时明显笨拙。

1.3 非阿克曼运动下的控制

1.3.1 基础任务

通过串口助手发送位置信息来驱动电机的正转反转，舵机左右转控制。

1.3.2 进阶扩展

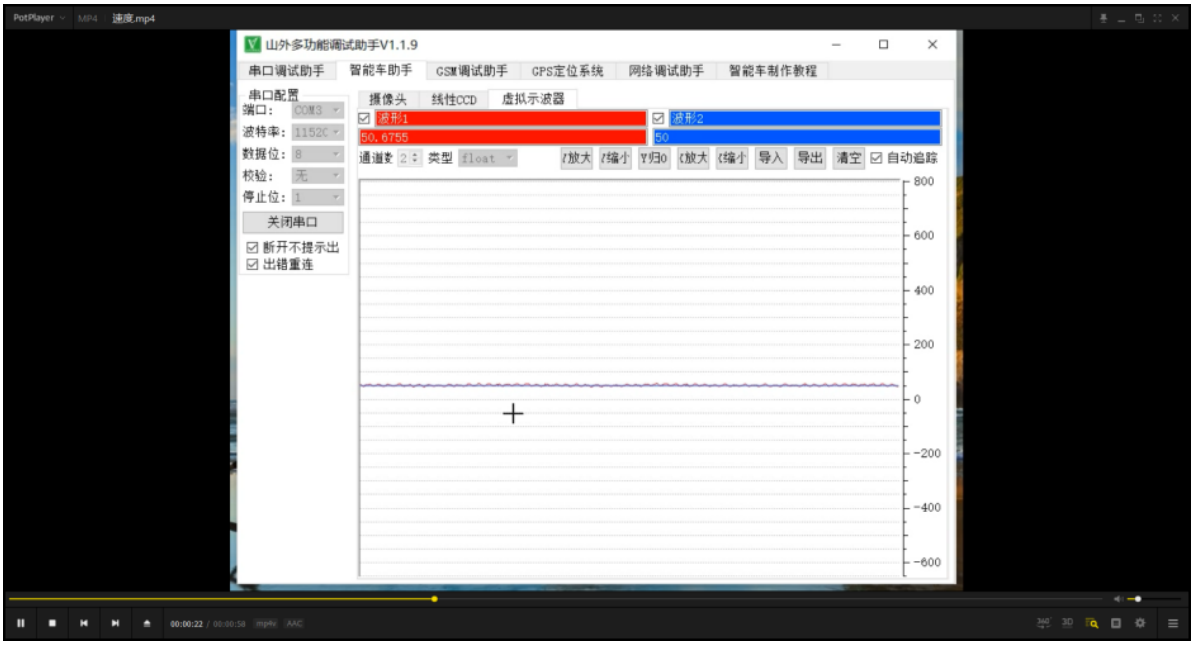
根据山外的通讯协议书写代码实现对车轮速度的检测，便于理论分析以及数据的调参。
通讯协议部分

```
void Usart_SendArray( USART_TypeDef * pUSARTx, uint8_t *array,uint8_t num)
{
  uint8_t i;
  for(i=0;i < num;i++)
  {
    Usart_SendByte( pUSARTx, array[i]);
  }

  // 等待发送数据寄存器为空
  while (USART_GetFlagStatus(pUSARTx, USART_FLAG_TC) == RESET);
}
void shanwai_sendware(uint8_t *wareaddr1,uint8_t *wareaddr2, uint32_t waresize)
{
    uint8_t cmdf[2] = {0x03, 0xfc};    //串口调试 使用的前命令
    uint8_t cmdr[2] = {0xfc, 0x03};    //串口调试 使用的后命令

    Usart_SendArray(UART4,cmdf, sizeof(cmdf));    //先发送前命令
    Usart_SendArray(UART4,wareaddr1, waresize);
    Usart_SendArray(UART4,wareaddr2, waresize);    //发送数据
    Usart_SendArray(UART4,cmdr, sizeof(cmdr));    //发送后命令
}
```

调参后的速度效果（以左轮50cm/s为例，红线为实际速度，蓝线为目标速度）



2. 创新点介绍

2.1 成果展示部分

我们为了更好的展示成果，对呈现方式做出了几项创新。
1.视频左半部分为小车的运行情况，右半部分为同时间串口助手的运行情况。
2.用白色的标记物粘贴在轮子上，以明显看出左右轮子的运动情况。
3.我们在程序中添加回显代码，回显代码包含两个部分，一个是发送信息的回显，一个是此时运动状态的显示，我们们可以通过回显的情况判断此次发送信息的情况，具体判断标准在4.5.2.2中做出详细讲解，同时也可以通过数据明确此时小车的运动状态。
4.通过蓝牙控制的方式展示了小车阿克曼转弯的效果，并比较了阿克曼与非阿克曼运动的差异。
5.借助山外调试助手，对速度控制做出呈现。

2.2 代码部分

1.系统框架部分采用模式切换的方式，自由切换阿克曼与非阿克曼模式。
2.通讯方面自主创新通讯协议，实现对小车各种运动情况的控制。
3.根据山外通讯协议，编写传输数据代码，实现速度显示的效果。
4.设置标志位，采用沿次发送运动状态的方式，确保发送数据的准确性。
5.算法上创新模式切换下的测速，PID等函数，以配合框架运行。
6.函数相互独立，便于以后代码的移植。

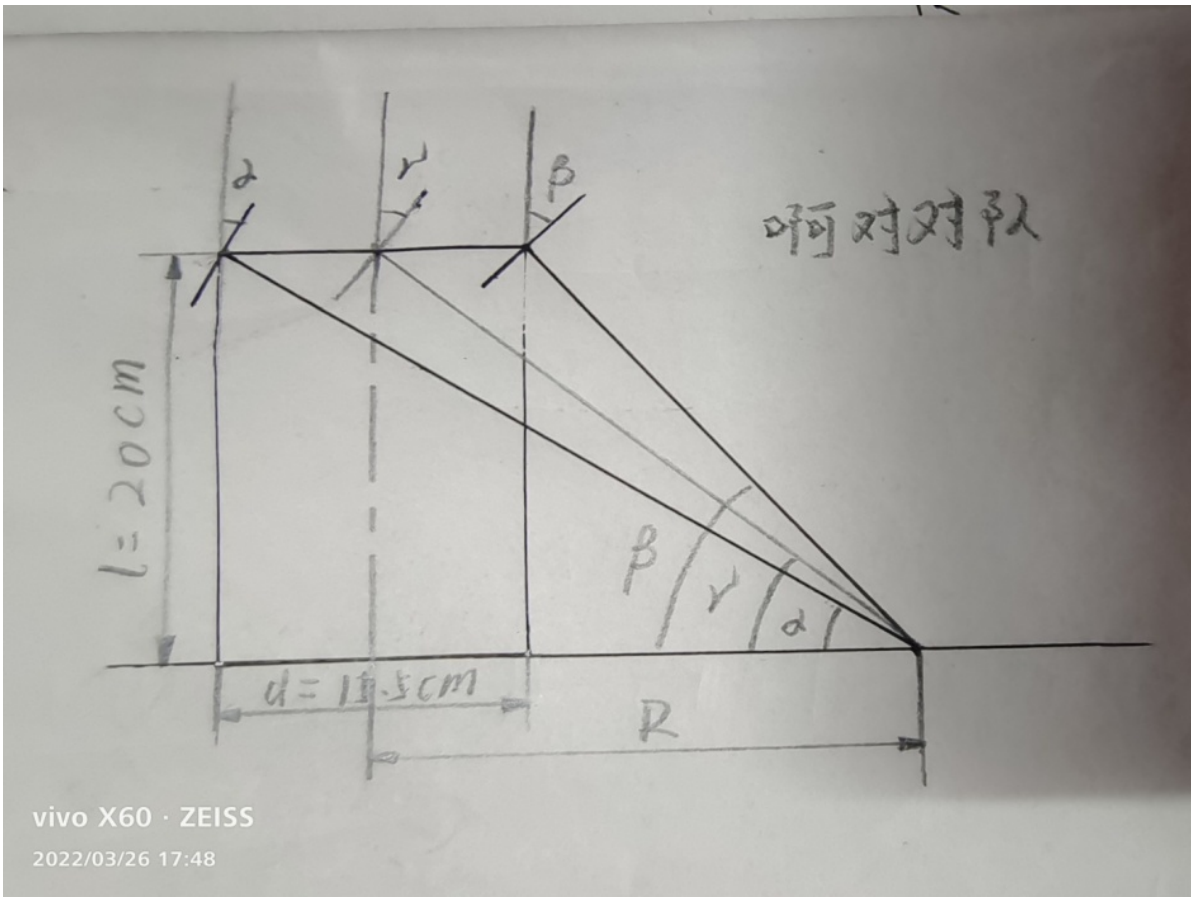
3. 阿克曼运动模型的理解，应用及未来优化

3.1 基本认识

阿克曼运动模型是理想情况下，即在不考虑汽车质心侧偏、汽车行驶过程中的侧向力、横摆角和极端恶劣的路况下，前方两个轮子控制转向，后方两个轮子控制运动速度，实现每个车轮的运动轨迹完全符合它的自然运动轨迹，保证轮胎和地面是纯滚动无滑移现象的一种运动。

阿克曼运动模型满足前方的左右车轮的垂线交于后两轮的延长线上，四个轮子的路径圆心大致交于后轴延长线上瞬时转向中心，从而让车辆顺畅转弯。

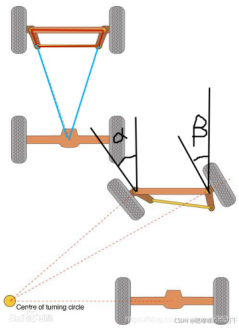
3.2 理论分析



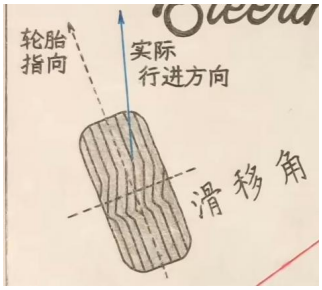
3.2.1 前轮偏转角度

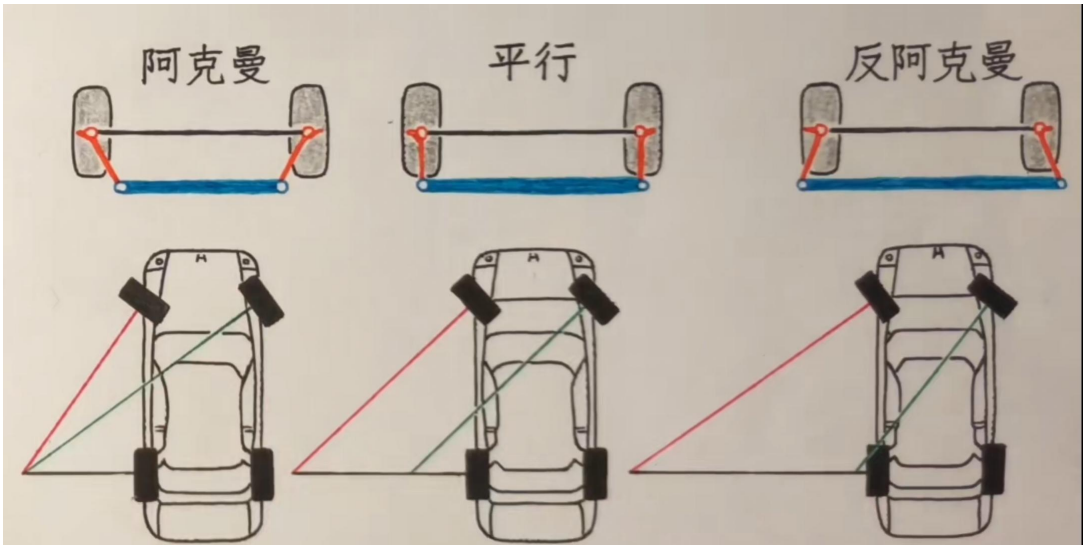
如上图，改图下文称为前图，设车长为l，车宽为d，前方左轮和右轮的偏角分别为α和β,则cotα=(R+d/2)/l，cotβ=(R-d/2)/l，得cotα-cotβ=d/l，也就是当车轮在运动过程中，转弯外侧车轮偏角的余切值-转弯内侧的余切值=车宽/车长。

因次阿克曼运动模型下的运动前两轮的偏角是有关的，一旦确定了其中的某一个，就能确定了另一个的偏转角度。



实现这种角度关系常用的机械结构为阿克曼梯形，如上图，红线标注处是阿克曼梯形，蓝线标出的便是标准阿克曼运动下梯形满足的性质，梯形边延长线相交于后轴的中心位置，此时阿克曼运动满足理想运动，内轮转角α比外轮转角β大大约2~4度。





然而在实际的智能车运动过程中，如上图，转弯时会有摩擦，滑移角等现象的存在，造成轮胎转向与轮胎的实际行进方向发生偏移，因而也会有将延长线交于后轴之上的百分比阿克曼模型，甚至会有反阿克曼运动模型的结构，如上图。

3.2.2 后轮的差速控制

前图中，阿克曼运动模型下的差速控制，左右轮的速度是相对绑定的，一方面可以通过设置转弯的半径R，或者前方中心线的偏转角度γ控制前轮的偏转幅度，另一方面通过设置中心线上车的运动线速度或者车的转弯角速度控制车的运动快慢。

首先根据cotγ=R/l，可以相互推导出R与γ的关系，左右轮速度测算，中心速度v=w*R，可以推导出v与w的关系。

左转情况下：v左=v（*1-d/2/R*）,*v*右=*v*（1+d/2/R）。右转情况下：v左=v（*1+d/2/R*）,*v*右=*v*（1-d/2/R）。由于R=l/tanγ，当γ的正负能够改变半径的正负，因此设γ左转时为正，那么就可以根据tanγ的正负变化以及1加减d/2/R的变化相抵消，总结成为一个公式，v左=v（*1-d*tanγ/2/l），v右=v（*1+d*tanγ/2/l）。同时我们也能够推算出关于左右轮差速的一个公式，Δv=w*d*=*v*d/R。下图为这些变量的相对转化的代码图片。

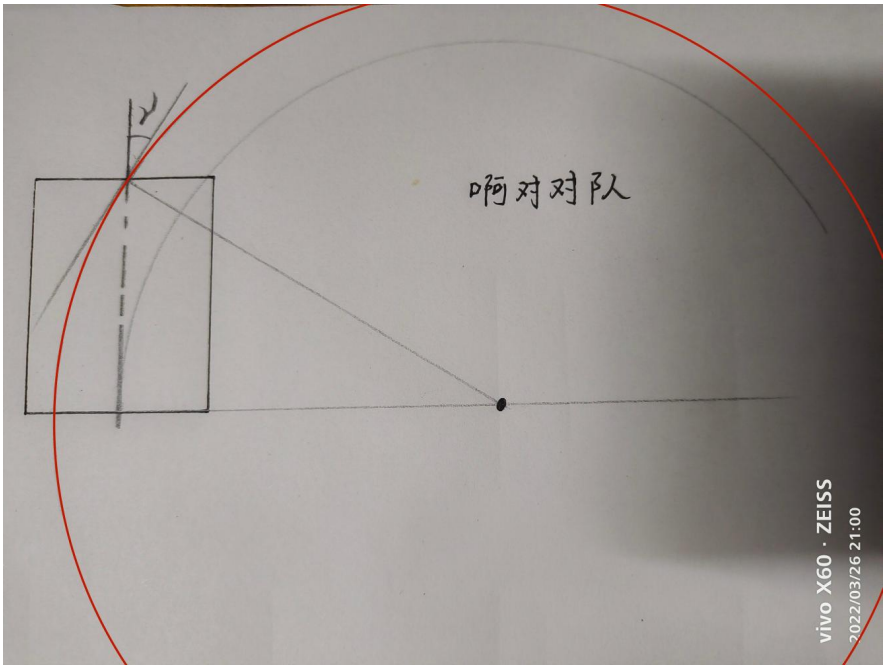
```
float PID_realize(float actual_val)
{

    if(mode==1)//阿克曼运动模式
      {
    /*title_angle是中心线偏移目标角度值，首先用（angle-1210）/2000
      得出此时舵机偏转角度占180度的比例，再*3.1415926535得出舵机偏
      转角度的弧度制，再乘前方车轮偏转角度和舵机偏转角度的比例系数
      2，换算成前方车轮中心线的偏转角度title_angle          */
    title_angle=2.0*(((float)angle-1210.0)*3.1415926535/2000.0);
    double Tand = tan(title_angle);//求解正切值
    pid.target_val= v*(1-15.5*Tand/2.0/20.0);   //根据偏转角度正切值换算出

  }
    else{}
```

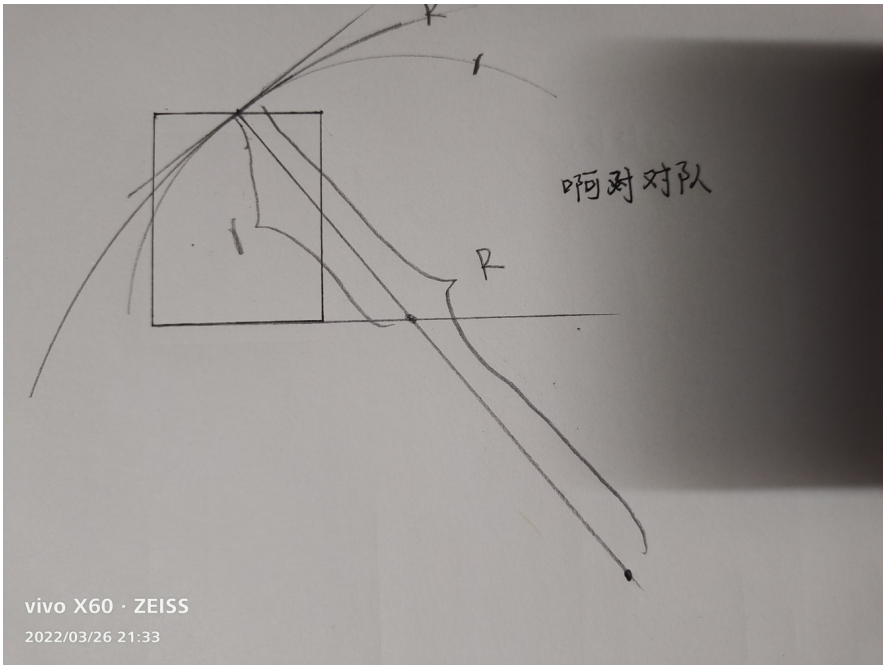
3.3 阿克曼运动的应用

3.3.1 阿克曼运动在线下赛的应用思考



将阿克曼运动模型应用于实际时，如线下赛车道线检测，并不是直接考虑左右轮此时的转向角度，而是考虑前轴中点的偏移角度如上图。这个γ对应前图中的γ，而运动的路径也是用红线标出的路径而不是测算速度时所利用的后轴中心的半径。

如前图中cotγ=R/l,cotα=(R+d/2)/l，cotβ=(R-d/2)/l,得cotγ-cotβ=d/(2*l*)，*cotα-cotγ=d/(2*l），同样的，得到γ的大小也同样得到了α和β的值。



对问题进一步思考发现，真正在沿某个弧度进行运动时并不能准确的保证车体的偏转角度正好满足后轴的延长线过圆周的圆心，如上图，理想的运动路径为R，而根据阿克曼运动模型无法直接进行按照R运动，但是可以根据转角γ找出可以运动圆r的途径，从而将一个大圆弧转换成为无数个小的圆弧，实现一个理想的圆弧转换，当让可能会产生误差，但可以通过累计误差或者及时的纠正进行对路径的优化。

3.3.2 生活中的阿克曼

阿克曼运动已经广泛应用于日常生活中，无论是对于智能车还是对于家用汽车，甚至赛车，阿克曼运动模型都是其不可缺少的运动模型，只不过对于不同的车型会有着不同的运行要求，因此形成阿克曼的运动机械结构，以及阿克曼的程度也各不相同。

3.4 未来优化

1.根据线下赛车模的数据信息，对参数进行修改和优化。
2.结合车道线，红绿灯检测等任务进行完整的闭环控制。
3.进行多线程操作，优化小车运动效果。
4.结合具体线下赛要求做出功能优化和创新。

4. 代码讲解

代码讲解会按照阿克曼运动模型建立的先后顺序进行讲解，重点部分会进行细致讲解。

4.1 主函数

主函数中包含对各个初始化函数的调用，以及对主控制函数的一个循环调用。

```
int main()
{

  angle_last=angle;
  ADVANCE_TIM_Init();//TIM1及其外设端口初始化（舵机）
  TIM_MOTOR_Init();//TIM3及其外设端口初始化（电机）
  Delay_Init();//延时函数初始化
  LED_GPIO_Config();//LED灯GPIO初始化，用于检测效果
  TIM4_Init();//左轮编码器模式
  TIM8_Init();//右轮编码器模式
  USART_Config();//蓝牙串口
  PID_param_init();//PID参数初始化

  while(1){
  motor_pid_control();//主控制函数


  }
}
```

4.2 主控制函数motor_pid_control();

4.2.1 文件安排

pid_motor.c文件，及其头文件pid_motor.h

4.2.3 函数介绍

首先获取左轮当前速度speed_L = Get_Speed_L();

而后将获取的左轮速度作为变量给到PID_realize（）函数进行计算，通过PID对于误差的分析将本次的速度值转化成为可以利用成为控制电机的参数cont_val = PID_realize(speed_L);

接下来会对cont_val也就是电机定时器TIM3的CCR进行阈值设限和正负判断，阈值为+500和-500同ARR值，如过为正则电机正转，如果为负，则电机反转。

最后判断标志位情况，进行小车运动状态的数据发送。

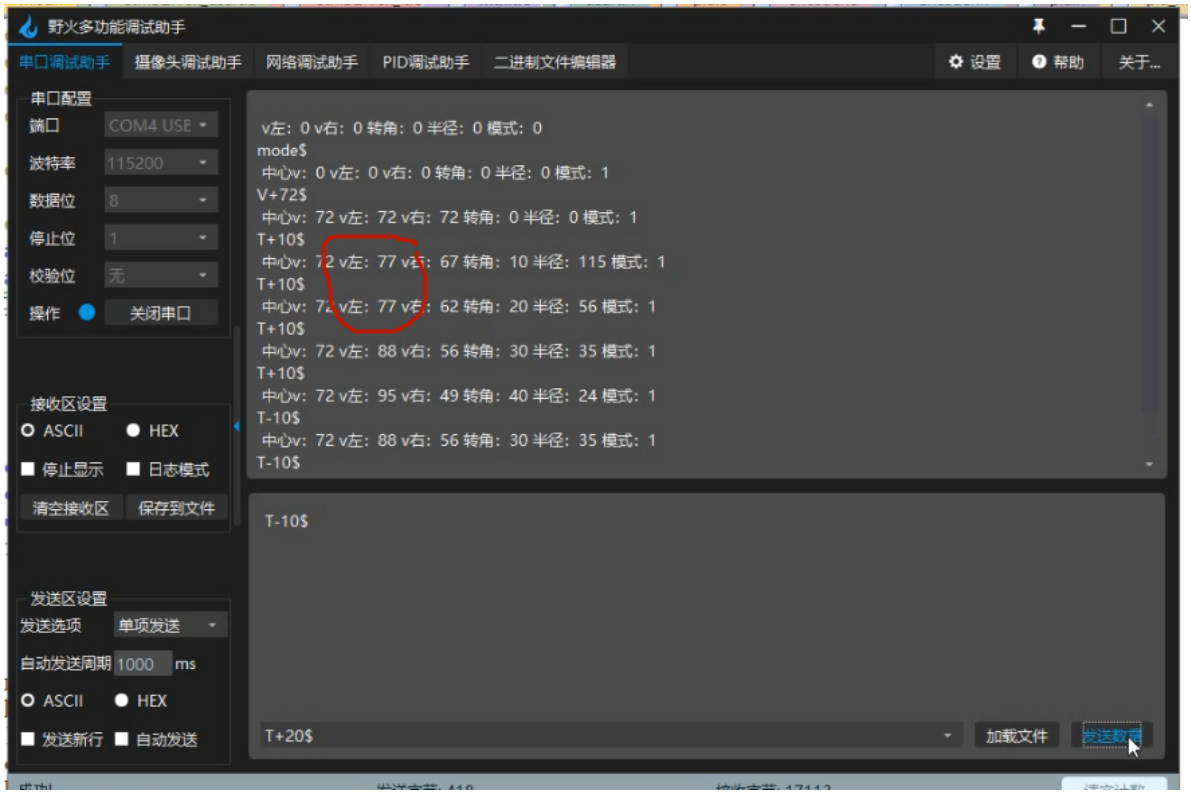
4.2.4 优化介绍

关于串口传输小车当前的运动状态的技术优化。

更改前：

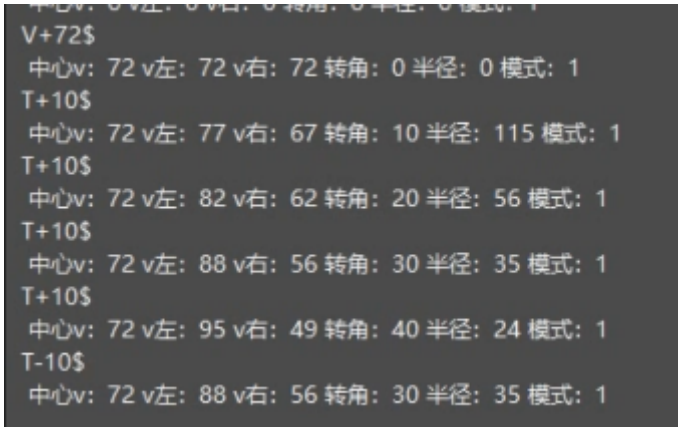
每经历一次主函数控制就会做一次串口接受信息的判断，判断标准为mode_usart的数值，当该数数值为1时，则是接受数据正常，会通过print函数（print函数已经重定向）向上位机发送此时车的运动信息，包含左右轮运动速度，转弯角度（左角度为负数，右角度为正数），以及转弯半径。

产生的问题，如下图，如果串口接收中断在函数的中间发生，就会引起左轮速度并没有计算成为理想的数据，导致数据错误。



更改后：

我们增加了沿次发送的标志位，使得接收数据后等到下一次控制函数的执行，再进行数据的显示，这样就能有效避免该问题的发生。



4.2.5 代码部分

```
/**
  * 函数功能：主控制函数
  * 输入参数：无
  * 返 回 值：左轮速度  L.speed
```

```
  * 作    者：啊对对队
  * 创    新：
  主控制函数中进行了左右轮的速度数据获取，计算，控制过程
  并对小车运动状态的信息传输做出控制
  */
void motor_pid_control(void)
{

    /*左轮数据处理*/
        float cont_val_L = 0;           // 当前左轮控制值（控制电机的占空比）
        float speed_L = 0;       // 左轮当前的实际速度
    speed_L = Get_Speed_L();//返回值单位cm/s
    cont_val_L = PID_realize(speed_L); // 进行 PID 计算
    //printf("%.2f\n",speed_L);
    /* 创新   调用山外发送函数--助于数据分析及结果呈现*/
      //shanwai_sendware((u8*)&speed_L,(u8*)&pid.target_val,4);

    /*右轮数据处理*/
    float cont_val_R = 0;           // 当前控制值（控制电机的占空比）
        float actual_speed_R = 0;       // 右轮当前的实际速度
        actual_speed_R = Get_Speed_R();//返回值单位cm/s
    cont_val_R = PID_realize2(actual_speed_R); // 进行 PID 计算
    //printf("%.2f\n",actual_speed_R);
    /* 创新   调用山外发送函数--助于数据分析及结果呈现*/
      //shanwai_sendware((u8*)&actual_speed_R,(u8*)&pid2.target_val,4);


        /* 左轮速度上限及方向处理*/
        if (cont_val_L > 500) // 判断电机方向
        {
            cont_val_L=500;
            TIM_SetCompare2(TIM3,0);
            TIM_SetCompare1(TIM3,(int)cont_val_L);
        }
        else if(cont_val_L<-500)
        {
            cont_val_L = -500;
            cont_val_L=-cont_val_L;
            TIM_SetCompare2(TIM3,(int)cont_val_L);
            TIM_SetCompare1(TIM3,0);
        }
        else if(cont_val_L>0)
        {
            TIM_SetCompare2(TIM3,0);
            TIM_SetCompare1(TIM3,(int)cont_val_L);
        }
        else
        {
            cont_val_L=-cont_val_L;
            TIM_SetCompare2(TIM3,(int)cont_val_L);
            TIM_SetCompare1(TIM3,0);
        }



        /* 右轮速度上限及方向处理*/
        if (cont_val_R > 500)
```

```
        {
            cont_val_R=500;
            TIM_SetCompare4(TIM3,0);
            TIM_SetCompare3(TIM3,(int)cont_val_R);
        }
        else if(cont_val_R<-500)
        {
            cont_val_R = -500;
            cont_val_R=-cont_val_R;
            TIM_SetCompare4(TIM3,(int)cont_val_R);
            TIM_SetCompare3(TIM3,0);
        }
        else if(cont_val_R>0)
        {
            TIM_SetCompare4(TIM3,0);
            TIM_SetCompare3(TIM3,(int)cont_val_R);
        }
        else
        {
            cont_val_R=-cont_val_R;
            TIM_SetCompare4(TIM3,(int)cont_val_R);
            TIM_SetCompare3(TIM3,0);
        }

         /*判断角度信息是否发生变化*/
    if(angle_last!=angle){

      TIM_SetCompare1(TIM1, angle);
      angle_last=angle;

    }
    /* 创新：沿次数据发送程序--解决串口中断发生引起的左轮速度数据错误问题*/
    if(print==1) //判断沿次发送标志位是否置1
      {
      extern float title_angle; //弧度制转角
      float R; //旋转半径
      title_angle=2.0*(((float)angle-1210.0)*3.1415926535/2000.0);

      if(title_angle>0){
      R=(float)20.0/tan(title_angle);
      }
      else{
        R=(float)20.0/tan(-title_angle);
      }
      /*创新：根据不同模式，进行不同的数据发送形式*/
      if(mode==1)
        {
        printf("\n 中心v：%.f v左：%.f v右：%.f 转角：%.f 半径：%.f 模式：%d\n",v,
pid.target_val, pid2.target_val,(float)-(angle-1210)*2.0*180/2000.0,R,mode);  //
打印实际值和目标值
      }
      else if(mode==0){
        printf("\n v左：%.f v右：%.f 转角：%.f 半径：%.f 模式：%d\n", pid.target_val,
pid2.target_val,(float)-(angle-1210)*2.0*180/2000.0,R,mode);  // 打印实际值和目标值
      }
        print=0;
    }
```

```
    else if(print==2){
       printf("输入错误");
       print=0;
    }
    /*首次判断到串口数据标志位时，将标志传递给沿次发送标志位*/
    if(mode_usart==1){
      print=1;
      mode_usart=0;
    }
    else if(mode_usart==2){
      print=2;
      mode_usart=0;
    }
    else{
  }
}
```

4.3 速度测算函数 Get_Speed();

4.3.1文件安排

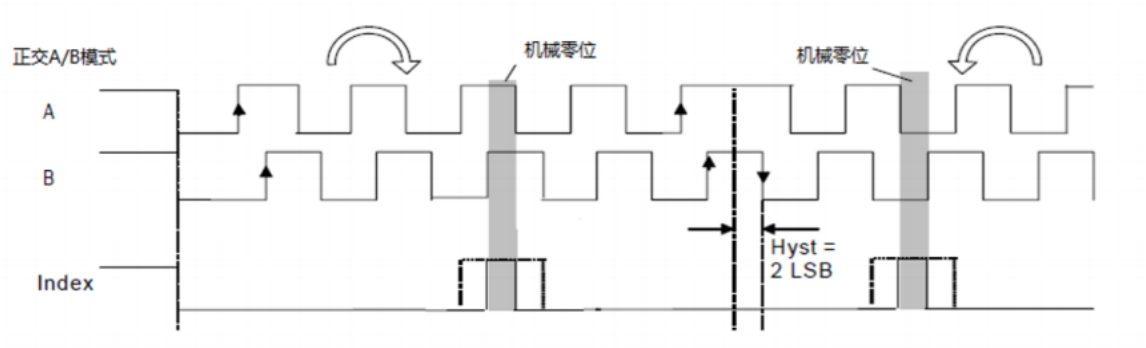
速度测算函数是编码器文件encoder.c中的核心函数，其头文件为encoder.h

4.3.2 函数介绍

首先为了更好的理解速度测算函数，先介绍编码器中断服务函数void TIM4_IRQHandler(void);

4.3.2.1 中断服务函数void TIM4_IRQHandler(void);

编码器的单个通道每个周期（编码器轮子转一圈）会发送512个方形波，由于采用上下沿同时捕获每个通道一个周期会捕获512×2次，同时采用TI1，2同时计数，因而是512×4，又从0开始计数，中断源为定时器的更新中断，也就是计数为CCR=512×4-1时触发中断，发生中断后变量title（编码器1也就是左轮编码器是tiltle1，右轮编码器是tiltle2）进行计数。




```
/**
  * 函数功能: 左编码器中断服务函数
  * 输入参数: 无
  * 返 回 值: 无
  * 作    者：啊对对队
  * 说    明: 通过定时器CR1位判断轮子转向
  */
void TIM4_IRQHandler(void)
{
```

```
    if (TIM_GetITStatus(TIM4, TIM_IT_Update)) // 检查 TIM4 更新中断发生与否
    {
      LED_B_TOGGLE;

        if((TIM4->CR1>>4 & 0x01)==0) //DIR==0
            L.integer++;
        else if((TIM4->CR1>>4 & 0x01)==1)//DIR==1
            L.integer--;
        TIM_ClearITPendingBit(TIM4, TIM_IT_Update); // 清除 TIMx 更新中断标志
    }

}
/**
  * 函数功能: 右编码器中断服务函数
  * 输入参数: 无
  * 返 回 值: 无
  * 作     者: 啊对对队
  * 说     明: 通过定时器CR1位判断轮子转向
  */
void TIM8_UP_IRQHandler(void)
{

    if (TIM_GetITStatus(TIM8, TIM_IT_Update)) // 检查 TIM4 更新中断发生与否
    {
      LED_2_TOGGLE;

        if((TIM8->CR1>>4 & 0x01)==0) //DIR==0
            R.integer++;
        else if((TIM8->CR1>>4 & 0x01)==1)//DIR==1
            R.integer--;
        TIM_ClearITPendingBit(TIM8, TIM_IT_Update); // 清除 TIMx 更新中断标志
    }

}
```

4.3.2.2 速度测算函数Get_Speed();

4.3.2.2.1常见测速方式

M法:

又叫做频率测量法。这种方法是在一个固定的定时时间内（以秒为单位），统计这段时间的编码器脉冲数，计算速度值。设编码器单圈总脉冲数为 C，在时间 T0 内，统计到的编码器脉冲数为 M0，则转速 n 的计算公式为:

$$n = \frac{M_0}{CT_0}$$ (单位：r/s)

在高速测量时 M0 很大，可以获得较好的测量精度和平稳性。

T法:

又叫做周期测量法。这种方法是建立一个已知频率的高频脉冲并对其计数，计数时间由捕获到的编码器相邻两个脉冲的间隔时间 TE 决定，计数值为 M1。设编码器单圈总脉冲数为 C，高频脉冲的频率为 F0，则转速 n 的计算公式为:

$$n = \frac{1}{CT_E} = \frac{F_0}{CM_1}$$ (单位：r/s)

编码器单圈总脉冲数 C 是常数，高频脉冲频率 F0 已设定好，所以转速 n 跟 M1 成反比。在电机高转速的时候，编码器脉冲间隔时间 TE 很小，使得测量周期内的高频脉冲计数值 M1 也变得很少，导致测量误差变大，而在低转速时，TE 足够大，测量周期内的 M1 也足够多，所以 T 法和 M 法刚好相反，更适合测量低速。

表73　计数方向与编码器信号的关系

有效边沿	相对信号的电平 (TI1FP1对应TI2, TI2FP2对应TI1)	TI1FP1信号		TI2FP2信号	
		上升	下降	上升	下降
仅在TI1计数	高	向下计数	向上计数	不计数	不计数
	低	向上计数	向下计数	不计数	不计数
仅在TI2计数	高	不计数	不计数	向上计数	向下计数
	低	不计数	不计数	向下计数	向上计数
在TI1和TI2上计数	高	向下计数	向上计数	向上计数	向下计数
	低	向上计数	向下计数	向下计数	向上计数

由于测算数据为两个通道的上升及下降沿情况，数据变化极大，并且测算的速度较快，因此我们采用M法。

4.3.2.2.2 代码分析

左右轮子测速方式相同，以下分析以TIM4左轮为例

首先读取此时的定时器CNT值，并直接对CNT的数值进行转换L.fraction =(float) (TIM4->CNT)/(512.0×4.0);得出现在所转圈数的小数部分。

接着计算出此时圈数的准确数值L.title=L.fraction + L.integer;//当前转数。

分析：
当圈数为正数时，自然是所转的圈数加上小数的部分就是实际的圈数。
对于圈数为负数的情况下，例如从圈数为0的情况下开始向下计数，那么 L.integer会变成-1，而CNT会从512.0×4.0-1开始向下计数直到变成0，因而实际的L.title就应该是L.integer加上L.fraction就是加上此时CNT占512.0×4.0的比例。

接着延时10ms后再次获取圈数信息title_last。

将两次的圈数做差，得出10ms的时间延时内编码器所转圈数。

而后先×100得到1s编码器所转圈数，再根据齿轮数的关系×30/68得出车轮所转圈数，再×6.4×3.1415926，其中6.4为车轮直径，3.1415926为Π的约值，便能计算出1s内车轮行进的距离也就是速度。

```
/**
  * 函数功能：计算左轮速度
  * 输入参数：无
  * 返 回 值：左轮速度   L.speed
  * 作      者：啊对对队
  * 创      新：速度求解算法
  采取延时求解速度的方式，最大程度防止因中间函数的延时导致的速度求解误差。
  */
float Get_Speed_L(void)
{
```

```
      L.fraction =(float) (TIM4->CNT)/(512.0*4.0); //读取此时编码器转数的小数部分
        L.title=L.fraction + L.integer; // 当前编码器精确转数
        //printf("%d\n\n\n",circle_count);
    Delay_MS(10);//延时0.01s
        L.fraction =(float) (TIM4->CNT)/(512.0*4.0); //读取此时编码器转数的小数部分
        L.title_last=L.fraction + L.integer; // 当前编码器精确转数
  /*
    先*100得到1s编码器所转圈数，再根据齿轮数的关系*30/68得出车轮
    所转圈数，再*6.4*3.1415926，其中6.4为车轮直径，3.1415926为π
    的约值，便能计算出1s内车轮行进的距离也就是速度(cm/s)。
  */
      L.speed=(float)(L.title_last-L.title)*100.0*30*6.4*3.1415926/68;
        return L.speed;
}



/**
  * 函数功能: 计算右轮速度
  * 输入参数: 无
  * 返 回 值: 右轮速度   R.speed
  * 作     者：啊对对队
  * 创     新: 速度求解算法
  采取延时求解速度的方式，最大程度防止因中间函数的延时导致的速度求解误差。
  */

float Get_Speed_R(void)
{
      R.fraction =(float) (TIM8->CNT)/(512.0*4.0); //读取此时编码器转数的小数部分
        R.title=R.fraction + R.integer; // 当前编码器精确转数
        //printf("%d\n\n\n",circle_count);
    Delay_MS(10);
        R.fraction =(float) (TIM8->CNT)/(512.0*4.0); //读取此时编码器转数的小数部分
        R.title_last=R.fraction + R.integer;  // 当前编码器精确转数
  /*
    先*100得到1s编码器所转圈数，再根据齿轮数的关系*30/68得出车轮
    所转圈数，再*6.4*3.1415926，其中6.4为车轮直径，3.1415926为π
    的约值，便能计算出1s内车轮行进的距离也就是速度(cm/s)。
  */
      R.speed=(float)(R.title_last-R.title)*100.0*30*6.4*3.1415926/68;
        return R.speed;
}
```

4.3.2.3 编码器其他代码部分

宏定义及数据结构体date：

```
#define ENCODER_TIM                    TIM4
#define ENCODER_TIM_APBxClock_FUN      RCC_APB1PeriphClockCmd
#define ENCODER_TIM_CLK                RCC_APB1Periph_TIM4

// 输入捕获能捕获到的最小的频率为 72M/{ (ARR+1)*(PSC+1) }
#define ENCODER_TIM_Period           512*4-1
#define ENCODER_TIM_Prescaler        0
// 中断相关宏定义
#define ENCODER_TIM_IRQ             TIM4_IRQn
#define ENCODER_TIM_IRQHandler      TIM4_IRQHandler
```

```
// TIM3 输入捕获通道
#define ENCODER_TIM_CH1_GPIO_CLK    RCC_APB2Periph_GPIOB
#define ENCODER_TIM_CH1_PORT        GPIOB
#define ENCODER_TIM_CH1_PIN         GPIO_Pin_6
#define ENCODER_TIM_CH2_GPIO_CLK    RCC_APB2Periph_GPIOB
#define ENCODER_TIM_CH2_PORT        GPIOB
#define ENCODER_TIM_CH2_PIN         GPIO_Pin_7


//编码器接口倍频数
#define ENCODER_MODE                TIM_ENCODER_TI12

#define ENCODER_TIM2                    TIM8
#define ENCODER_TIM_APBxClock_FUN2      RCC_APB2PeriphClockCmd
#define ENCODER_TIM_CLK2                RCC_APB2Periph_TIM8

// 输入捕获能捕获到的最小的频率为 72M/{ (ARR+1)*(PSC+1) }
#define ENCODER_TIM_Period2           512*4-1
#define ENCODER_TIM_Prescaler2        0
// 中断相关宏定义
#define ENCODER_TIM_IRQ2              TIM8_UP_IRQn
#define ENCODER_TIM_IRQHandler2       TIM8_UP_IRQHandler

// TIM3 输入捕获通道
#define ENCODER_TIM_CH1_GPIO_CLK2    RCC_APB2Periph_GPIOC
#define ENCODER_TIM_CH1_PORT2        GPIOC
#define ENCODER_TIM_CH1_PIN2         GPIO_Pin_6
#define ENCODER_TIM_CH2_GPIO_CLK2    RCC_APB2Periph_GPIOC
#define ENCODER_TIM_CH2_PORT2        GPIOC
#define ENCODER_TIM_CH2_PIN2         GPIO_Pin_7


//编码器接口倍频数
#define ENCODER_MODE2                TIM_ENCODER_TI12


typedef struct
{
     float integer;
     float fraction;
     float title;
     float title_last;
     float speed;
}date;
```

编码器结构体初始化：

编码器：

编码器共两个，分别是左，右轮测速，

左轮：TIM4，捕获引脚PB6，7，计数方式为TI1，TI2同时计数

右轮：TIM8，捕获引脚PC6，7，计数方式为TI1，TI2同时计数

```
void TIM4_Init(void)
{
    GPIO_InitTypeDef              GPIO_InitStruct;
    TIM_TimeBaseInitTypeDef       TIM_TimeBaseInitStructure;
  TIM_ICInitTypeDef             ECD_TIM_ICInitStructure;
  NVIC_InitTypeDef              NVIC_InitStructure;


    //使能TIM4时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4 ,ENABLE);
    //使能引脚时钟，复用时钟
    RCC_APB2PeriphClockCmd(ENCODER_TIM_CH1_GPIO_CLK|ENCODER_TIM_CH2_GPIO_CLK ,
ENABLE);



    //GPIO初始化
    // 设置输入类型
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  // 设置引脚速率
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
  // 选择要控制的 GPIO 引脚
  GPIO_InitStruct.GPIO_Pin = ENCODER_TIM_CH1_PIN |ENCODER_TIM_CH2_PIN;

    GPIO_Init(GPIOB,&GPIO_InitStruct);


    //定时器设置
    //重装载值 = ( 编码器线数 * 4 ) -1
  TIM_TimeBaseInitStructure.TIM_Period = ENCODER_TIM_Period;
    //预分频
  TIM_TimeBaseInitStructure.TIM_Prescaler=ENCODER_TIM_Prescaler;
    //向上计数
  TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up;
    //时钟分割
  TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;
  TIM_TimeBaseInitStructure.TIM_RepetitionCounter=0;
  TIM_TimeBaseInit(ENCODER_TIM,&TIM_TimeBaseInitStructure);//初始化TIM4

    //编码器设置
    //编码器配置标准外设库

TIM_EncoderInterfaceConfig(ENCODER_TIM,TIM_EncoderMode_TI12,TIM_ICPolarity_Risin
g, TIM_ICPolarity_Rising);//计数模式3



    //溢出中断设置
    //使能TIM4溢出中断
  TIM_ITConfig(ENCODER_TIM,TIM_IT_Update,ENABLE);

  NVIC_InitStructure.NVIC_IRQChannel=ENCODER_TIM_IRQ;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x00;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x01;
  NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
```

```
  NVIC_Init(&NVIC_InitStructure);
    TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
    //清空CNT
    TIM_SetCounter(ENCODER_TIM,0);
    TIM_Cmd(ENCODER_TIM,ENABLE);


}
void TIM8_Init(void)
{
    GPIO_InitTypeDef              GPIO_InitStruct2;
    TIM_TimeBaseInitTypeDef       TIM_TimeBaseInitStructure2;
  TIM_ICInitTypeDef             ECD_TIM_ICInitStructure2;//? ? ? ?
  NVIC_InitTypeDef              NVIC_InitStructure2;


    //使能TIM8时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM8 ,ENABLE);
  //RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);
    //使能引脚时钟，复用时钟
    RCC_APB2PeriphClockCmd(ENCODER_TIM_CH1_GPIO_CLK2|ENCODER_TIM_CH2_GPIO_CLK2 ,
ENABLE);



    //GPIO初始化
    // 设置输入类型
  GPIO_InitStruct2.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  // 设置引脚速率
  GPIO_InitStruct2.GPIO_Speed = GPIO_Speed_50MHz;
  // 选择要控制的 GPIO 引脚
  GPIO_InitStruct2.GPIO_Pin = ENCODER_TIM_CH1_PIN2 |ENCODER_TIM_CH2_PIN2;

    GPIO_Init(GPIOC,&GPIO_InitStruct2);


    //定时器设置
    //重装载值 = ( 编码器线数 * 4 ) -1
  TIM_TimeBaseInitStructure2.TIM_Period = ENCODER_TIM_Period2;
    //预分频
  TIM_TimeBaseInitStructure2.TIM_Prescaler=ENCODER_TIM_Prescaler2;
    //向上计数
  TIM_TimeBaseInitStructure2.TIM_CounterMode=TIM_CounterMode_Up;
    //时钟分割
  TIM_TimeBaseInitStructure2.TIM_ClockDivision=TIM_CKD_DIV1;
  TIM_TimeBaseInitStructure2.TIM_RepetitionCounter=0;
  TIM_TimeBaseInit(ENCODER_TIM2,&TIM_TimeBaseInitStructure2);//初始化TIM4

    //编码器设置
    //编码器配置标准外设库

TIM_EncoderInterfaceConfig(ENCODER_TIM2,TIM_EncoderMode_TI12,TIM_ICPolarity_Risi
ng, TIM_ICPolarity_Rising);//计数模式3



    //溢出中断设置
    //使能TIM8溢出中断
```

```
  TIM_ITConfig(ENCODER_TIM2,TIM_IT_Update,ENABLE);

  NVIC_InitStructure2.NVIC_IRQChannel= ENCODER_TIM_IRQ2;
  NVIC_InitStructure2.NVIC_IRQChannelPreemptionPriority=0x00;
  NVIC_InitStructure2.NVIC_IRQChannelSubPriority=0x01;
  NVIC_InitStructure2.NVIC_IRQChannelCmd=ENABLE;
  NVIC_Init(&NVIC_InitStructure2);
    TIM_ClearITPendingBit(TIM8, TIM_IT_Update);
    //清空CNT
    TIM_SetCounter(ENCODER_TIM2,0);
    TIM_Cmd(ENCODER_TIM2,ENABLE);


}
```

4.4 PID速度控制函数PID_realize（）；

4.4.1文件安排

pid.c文件，及其头文件pid.h。

4.4.2函数介绍

首先通过对比多种PID算法，增量式PID算法具有原理简单，易于实现，适用面广，控制参数相互独立，参数的选定比较简单等优点，因而选用增量式PID。

在程序中，PID速度控制函数中分为两种数据分析情况：

一种是阿克曼模式下的PID运算，一种是非阿克曼运动模式下的运算。两者最大的区别就是左右轮速度的来源问题。

在阿克曼运动模式下，左右轮的速度是有中心线的速度和前方转向的角度所共同控制的，而在非阿克曼运动模式下，左右轮的速度是单纯考设定的数值进行操作的。

4.4.2.1阿克曼运动模式：

首先计算出前方转弯的角度。

当变量angle（TIM1（舵机控制对于定时器）的CCR）为1210时，前车轮摆正。
当angle>1210,则舵机右转，对应前方车轮左转
当angle<1210,则舵机右转，对应前方车轮左转

title_angle=2.0*(((float)angle-1210.0)×3.1415926535/2000.0);
说明：title_angle是中心线偏移目标角度值，首先用（angle-1210）/2000得出此时舵机偏转角度占180度的比例，再×3.1415926535得出舵机偏转角度的弧度制，再乘前方车轮偏转角度和舵机偏转角度的比例系数2，换算成前方车轮中心线的偏转角度title_angle

接着计算出偏转角度对应的tan值。

double Tand = tan(title_angle);

接着同阿克曼运动分析中后轮的差速控制，根据cotγ=R/l，可以相互推导出R与γ的关系，左右轮速度测算，中心速度v=w*R，可以推导出v与w的关系。

左转情况下：v左=v（*1-d/2/R*）,*v*右=*v*（1+d/2/R）。右转情况下：v左=v（*1+d/2/R*）,*v*右=*v*（1-d/2/R）。由于R=l/tanγ，当γ的正负能够改变半径的正负，因此设γ左转时为正，那么就可以根据tanγ的正负变化以及1加减d/2/R的变化相抵消，总结成为一个公式，v左=v（*1-d*tanγ/2/l），v右=v（*1+d*tanγ/2/l）。

```
/**
  * 函数功能: 左轮PID算法（此处只用PI）
  * 输入参数: 左轮速度  L.speed
  * 返 回 值: 控制参数  cont_val_L
  * 作    者：啊对对队
  * 创    新: 模式控制下的PID控制算法
  */
float PID_realize(float actual_val)
{

    if(mode==1)//阿克曼运动模式
      {
    /*title_angle是中心线偏移目标角度值，首先用（angle-1210）/2000
      得出此时舵机偏转角度占180度的比例，再*3.1415926535得出舵机偏
      转角度的弧度制，再乘前方车轮偏转角度和舵机偏转角度的比例系数
      2，换算成前方车轮中心线的偏转角度title_angle          */
    title_angle=2.0*(((float)angle-1210.0)*3.1415926535/2000.0);
    double Tand = tan(title_angle);//求解正切值
    pid.target_val= v*(1-15.5*Tand/2.0/20.0);  //根据偏转角度正切值换算出

  }
    else{}
        //计算目标值与实际值的误差
    pid.err=pid.target_val-actual_val;
        //PID算法实现
    pid.actual_val+=pid.Kp*(pid.err-pid.err_last)+pid.Ki*pid.err;
        //误差传递
    pid.err_last=pid.err;
    //printf("target=%d\n",angle);
        //返回当前实际值
    return pid.actual_val;
}
/**
  * 函数功能: 右轮PID算法（此处只用PI）
  * 输入参数: 右轮速度  R.speed
  * 返 回 值: 控制参数  cont_val_R
  * 作    者：啊对对队
  * 创    新: 模式控制下的PID控制算法
  */
float PID_realize2(float actual_val2)
{
  if(mode==1){

    title_angle=2.0*(((float)angle-1210.0)*3.1415926535/2000.0);
    double Tand = tan(title_angle);
    pid2.target_val= v*(1+15.5*Tand/2.0/20.0);

  }
  else{}
        //计算目标值与实际值的误差
    pid2.err=pid2.target_val-actual_val2;
        //PID算法实现
```

```
    pid2.actual_val+=pid2.Kp*(pid2.err-pid2.err_last)+pid2.Ki*pid2.err;
        //误差传递
    pid2.err_last=pid2.err;
        //返回当前实际值
    return pid2.actual_val;
}
```

4.4.2.2 非阿克曼运动模式

在非阿克曼运动模式下，由于左右速度都是分别设置，而且与舵机转向无关，因此直接进行PID测算即可。

4.5 串口通讯：

4.5.1 文件安排：

串口代码对应文件usart.c以及相应头文件usart.h

4.5.2函数介绍

4.5.2.1 UART4中断服务函数及串口数据解析函数

发送数据一共5位，位0：改变内容，位1：改变方式，位2，3：数据大小，位4：为'\0'，发送时以$为结束位。

在接收中断时进行阿克曼运动的调控。

一共两种运动模式，一种是阿克曼运动模式下的运动，一种非阿克曼运动模式。

模式切换为"mode$"

阿克曼运动模式：

	位0	位1	位2，3
中心线速度	V	+/-	加减速度大小，
=	更改后速度大小		
转弯角度	T	+/-	+增加右转角度，-减少左转角度
L/R	L设置左转角度，R设置右转角度		
R	L/R	左转/右转半径	

非阿克曼模式：

	位0	位1	位2，3
左轮速度	C	+/-	左轮加减速度大小，
=	左轮更改后速度大小		
右轮速度	B	+/-	左轮加减速度大小，
=	右轮更改后速度大小		
转弯角度	T	+/-	+增加右转角度，-减少左转角度
L/R	L设置左转角度，R设置右转角度		
R	L/R	左转/右转半径	

```
/**
  * 函数功能：分析串口发送信息
  * 输入参数：串口接收的五位的数组数据
  * 返 回 值：无
  * 作      者：啊对对队
  * 创      新：模式切换控制
  ------------------------------------------------------------------------
   阿克曼运动模式：      |          |
                            |  位0     |     位1      |      位2，3
           中心线速度   |   V     |     +/-     |      加减速度大小，
                          |          |      =      |      更改后速度大小
           转弯角度      |   T      |     +/-     |      +增加右转角度，-减少左转角度
                          |          |     L/R      |      L设置左转角度，R设置右转角度
                          |   R      |     L/R      |      左转/右转半径
   非阿克曼模式：
                            |  位0     |     位1      |      位2，3
           左轮速度      |   C      |     +/-     |      左轮加减速度大小，
                          |          |      =      |      左轮更改后速度大小
           右轮速度      |   B      |     +/-     |      左轮加减速度大小，
                          |          |      =      |      右轮更改后速度大小
           转弯角度      |   T      |     +/-     |      +增加右转角度，-减少左转角度
                          |          |     L/R      |      L设置左转角度，R设置右转角度
                          |   R      |     L/R     |      左转/右转半径
  ------------------------------------------------------------------------
  */

static void DataHandler(char a[5]){
if(mode==1)//阿克曼控制模式
  {
 if(a[0]=='V')//主速度控制
   {
   PID_param_init_L();
   PID_param_init_R();
   if(a[1]=='+')//加速
     {
        v+= (a[2] - '0')*10 + (a[3] - '0');
```

```
    }
     else if(a[1]=='-')//减速
      {
         v-= (a[2] - '0')*10 + (a[3] - '0');
    }
     else if(a[1]=='=')//设定速度
       {
         v=(a[2] - '0')*10 + (a[3] - '0');
     }
     else
       {
         mode_usart=2;
     }
  }
  else if(a[0]=='T')//中心偏角式转向控制
    {
      PID_param_init_L();
      PID_param_init_R();
      if(a[1]=='-')//舵机右转，轮子左转
      {
         angle+= (int)(((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
    }
     else if(a[1]=='+')//减速
      {
         angle-= (int)(((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
    }
     else if(a[1]=='L')//左转设定角度
       {
         angle= (int)(1210+((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
     }
       else if(a[1]=='R')//右转设定角度
       {
         angle= (int)(1210-((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
     }
     else
       {
         mode_usart=2;
     }
  }
  else if(a[0]=='R')//半径式转向控制
    {
      PID_param_init_L();
      PID_param_init_R();
      if(a[1]=='L'){
      angle= (int)(1210+atan(20/((a[2] - '0')*10 + (a[3] -
'0')))*2000/(2*3.1415926));
      }
      else if(a[1]=='R'){
      angle= (int)(1210+atan(20/((a[2] - '0')*10 + (a[3] -
'0')))*2000/(2*3.1415926));
      }
      else{
        mode_usart=2;
      }
  }
    /*模式切换*/
    else if(a[0]=='m'){
      if(a[1]=='o'){
```

```
      if(a[2]=='d'){
      if(a[3]=='e'){
      mode=0;
      }
      }
      }
    }
  else//数据第一位错误
    {
    mode_usart=2;
  }
  }
else//非阿克曼模式
       {
     if(a[0]=='C')//左轮速度控制控制
    {
      PID_param_init_L();
      if(a[1]=='+')//左轮加速
      {
         pid.target_val+= (a[2] - '0')*10 + (a[3] - '0');
    }
     else if(a[1]=='-')//左轮减速
      {
         pid.target_val-= (a[2] - '0')*10 + (a[3] - '0');
    }
     else if(a[1]=='=')//左轮设定速度
       {
         pid.target_val=(a[2] - '0')*10 + (a[3] - '0');
     }
     else
       {
         mode_usart=2;
     }
   }
    else if(a[0]=='B')//右轮速度控制
      {
      PID_param_init_R();
    if(a[1]=='+')//右轮加速
      {
         pid2.target_val+= (a[2] - '0')*10 + (a[3] - '0');
    }
     else if(a[1]=='-')//右轮减速
      {
         pid2.target_val-= (a[2] - '0')*10 + (a[3] - '0');
    }
     else if(a[1]=='=')//右轮设定速度
       {
         pid2.target_val=(a[2] - '0')*10 + (a[3] - '0');
     }
     else
       {
         mode_usart=2;
     }
    }
      else if(a[0]=='T')//中心偏角式转向控制
    {
      if(a[1]=='+')//舵机右转，轮子左转
      {
```

```
        angle-= (int)(((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
   }
    else if(a[1]=='-')//舵机左转，轮子右转
     {
        angle+= (int)(((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
   }
    else if(a[1]=='L')//设定左转中心偏转角度
      {
        angle= (int)(1210+((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
    }
      else if(a[1]=='R')//设定右转中心偏转角度
      {
        angle= (int)(1210-((a[2] - '0')*10 + (a[3] - '0'))*2000/(2*180));
    }
    else
      {
        mode_usart=2;
    }
 }
 else if(a[0]=='R')//半径式转向控制
   {
     if(a[1]=='L')//左转
       {
     angle= (int)(1210+atan(20/((a[2] - '0')*10 + (a[3] -
'0')))*2000/(2*3.1415926));
     }
     else if(a[1]=='R')//右转
       {
     angle= (int)(1210+atan(20/((a[2] - '0')*10 + (a[3] -
'0')))*2000/(2*3.1415926));
     }
     else{
       mode_usart=2;
     }
 }
   /*模式切换*/
   else if(a[0]=='m'){
     if(a[1]=='o'){
     if(a[2]=='d'){
     if(a[3]=='e'){
     mode=1;
     }
     }
     }
   }

else//错误
   {
        mode_usart=2;
 }

 }
}

/**
  * 函数功能：串口中断服务函数
  * 输入参数：无
  * 返 回 值：无
```

```
  * 创     新：接受串口数据，并且回显和调用串口数据函数
  */

void USART_IRQHandler(void)
{
    uint8_t ucTemp;
  if (USART_GetITStatus(USARTx, USART_IT_RXNE) != RESET)//如果接收数据寄存器不为空
    {
        ucTemp = USART_ReceiveData(USARTx);//保留寄存器中的 数据
        USART_SendData(USARTx, ucTemp);//将数据返回至发送方
      if(ucTemp!='$'){
        a[i]=ucTemp;
        i++;
    }
      else{
         a[4] = '\0';
         mode_usart=1;
         DataHandler(a);
         i = 0;                          //初始化i，准备接收下一次的数据
      }
}
}
```

4.5.2.2 关于上位机接收信息解析

此部分在创新点处提到，在这里做出解释，单片机在每接收到一组数据信息后，mode_usart变量作为标志位会置1，接着会在数据处理函数中做出判断，如果是错误的数据那么 mode_usart会置2，最终会在结果会在主控制函数中进行沿次的发送，因此当完成一次发送时会产生四种情况。第一种是没有返回值，那么这种情况只会是串口发送数据失败，单片机没有接收到任何数据。第二种是只有发送数据的回显，而没有小车运动状况的显示，那么这种情况是发送数据位不足。第三种就是显示发送数据以及输入错误的显示，那么说明发送数据并不符合通讯规则。最后一种便是发送信息的回显以及小车的运动状况，这种情况就是证明这是一次成功的数据发送。

4.5.2.3 其他基本代码部分

宏定义

```
 //串口宏定义，不同的串口挂载的总线不一样，移植时需要修改这几个宏
#define USARTx                     UART4
#define USART_CLK                  RCC_APB1Periph_UART4
#define USART_APBxClkCmd           RCC_APB1PeriphClockCmd
#define USART_BAUDRATE             115200

// USART GPIO 引脚宏定义
#define USART_GPIO_CLK             (RCC_APB2Periph_GPIOC )
#define USART_GPIO_APBxClkCmd      RCC_APB2PeriphClockCmd

#define USART_TX_GPIO_PORT         GPIOC
#define USART_TX_GPIO_PIN          GPIO_Pin_10
#define USART_RX_GPIO_PORT         GPIOC
#define USART_RX_GPIO_PIN          GPIO_Pin_11

#define USART_IRQ                  UART4_IRQn
#define USART_IRQHandler           UART4_IRQHandler
```

串口初始化函数：

引脚：串口使用的是串口4，对应TX，RX引脚分别为PC10以及PC11，PC10设置为推挽复用模式，PC11设置为浮空输入模式

串口4：波特率为115200，字长8，停止位1，校验位无

中断类型为USART_IT_RXNE（接收中断）

```
/**
  * 函数功能: 串口硬件初始化配置
  * 输入参数: 无
  * 返 回 值: 无
  * 说    明: 使用标准库
  */

void USART_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;

    // 打开串口GPIO的时钟
    USART_GPIO_APBxClkCmd(USART_GPIO_CLK, ENABLE);

    // 打开串口外设的时钟
    USART_APBxClkCmd(USART_CLK, ENABLE);

    // 将UART4 Tx的GPIO配置为推挽复用模式
    GPIO_InitStructure.GPIO_Pin = USART_TX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(USART_TX_GPIO_PORT, &GPIO_InitStructure);

    // 将UART4 Rx的GPIO配置为浮空输入模式
    GPIO_InitStructure.GPIO_Pin = USART_RX_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(USART_RX_GPIO_PORT, &GPIO_InitStructure);

    // 配置串口的工作参数
    // 配置波特率
    USART_InitStructure.USART_BaudRate = USART_BAUDRATE;
    // 配置帧数据字长
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    // 配置停止位
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    // 配置校验位
    USART_InitStructure.USART_Parity = USART_Parity_No;
    // 配置硬件流控制
    USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
    // 配置工作模式，收发一起
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    // 完成串口的初始化配置
    USART_Init(USARTx, &USART_InitStructure);

    // 串口中断优先级配置
    NVIC_Configuration();

    // 使能串口接收中断
    USART_ITConfig(USARTx, USART_IT_RXNE, ENABLE);
```

```
    // 使能串口
    USART_Cmd(USARTx, ENABLE);
}
```

中断初始化函数：

中断源为UART4， 中断组为2，抢断优先级以及子优先级都为0

```
/**
  * 函数功能: 串口对应的嵌套向量中断控制器 NVIC 初始化函数
  * 输入参数: 无
  * 返 回 值: 无
  * 说    明: 使用标准库
  */

static void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* 嵌套向量中断控制器组选择 */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

    /* 配置 USART 为中断源 */
    NVIC_InitStructure.NVIC_IRQChannel = USART_IRQ;
    /* 抢断优先级*/
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    /* 子优先级 */
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    /* 使能中断 */
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    /* 初始化配置NVIC */
    NVIC_Init(&NVIC_InitStructure);
}
```

4.6 电机

电机的主要控制部分已经在主控制函数实现，这里主要是一些其他的电机函数

4.6.1 文件安排：

电机设置是motor.c以及对应的头文件motor.h

4.6.2 函数介绍：

电机初始化函数：

```
void TIM_MOTOR_Init(void)
{
    /* --------------输出比较通道1、2、3、4的GPIO初始化------------- */
    GPIO_InitTypeDef GPIO_InitStructure; //声明GPIO初始化结构体

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
    //使能AFIO复用时钟    //使能重映射IO时钟
```

```
    GPIO_PinRemapConfig(GPIO_PartialRemap_TIM3, ENABLE); //开启重映射
    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);
    //shut JTAG
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_0 |
GPIO_Pin_1; //设置GPIO管脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
     //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
     //设置输出频率
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    //初始化GPIO

    /*----------------------时基结构体初始化-----------------------*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure; //声明定时器时基结构体

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //使能重映射定时器时钟

    /* 配置周期，这里配置为0.0005s。提示：Tout= ((arr+1)*(psc+1))/Tclk */
    TIM_TimeBaseStructure.TIM_Period = 500 - 1;                  //定时器周期arr，即
自动重装载寄存器的值
                                                                 //累计
TIM_Period+1个频率后产生一个更新或者中断
    TIM_TimeBaseStructure.TIM_Prescaler = 71;                    //定时器预分频器设置
psc，时钟源经该预分频器才是定时器时钟
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;      //时钟分频因子，输出
互补脉冲配置死区时间需要用到
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //计数器计数模式，设
置为向上计数
    TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;             //设置重复计数器的值
为0，不设置重复计数
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);              //初始化定时器

    /*--------------------输出比较结构体初始化---------------------*/
    TIM_OCInitTypeDef TIM_OCInitStructure; //声明定时器输出比较结构体

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;              //选择定时器模
式:TIM脉冲宽度调制模式1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //使能输出比较
    TIM_OCInitStructure.TIM_Pulse = 0;                             //设置初始占空比
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;      //输出极性:TIM输
出比较极性高(高电平有效)


    TIM_OC1Init(TIM3, &TIM_OCInitStructure); //初始化输出比较通道1
    TIM_OC2Init(TIM3, &TIM_OCInitStructure); //初始化输出比较通道2
    TIM_OC3Init(TIM3, &TIM_OCInitStructure); //初始化输出比较通道3
    TIM_OC4Init(TIM3, &TIM_OCInitStructure); //初始化输出比较通道4

    TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能通道1的CCR1上的预装载寄存
器
    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能通道2的CCR2上的预装载寄存
器
    TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能通道3的CCR3上的预装载寄存
器
    TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能通道4的CCR4上的预装载寄存
器
```

```
    TIM_Cmd(TIM3, ENABLE);                //使能TIM
    TIM_ARRPreloadConfig(TIM3, ENABLE); //使能TIM1在ARR上的预装载寄存器
}
```