自己动手做一台 SLAM 导航机器人

第四章：差分底盘设计

作者：

知乎@小虎哥哥爱学习

目录

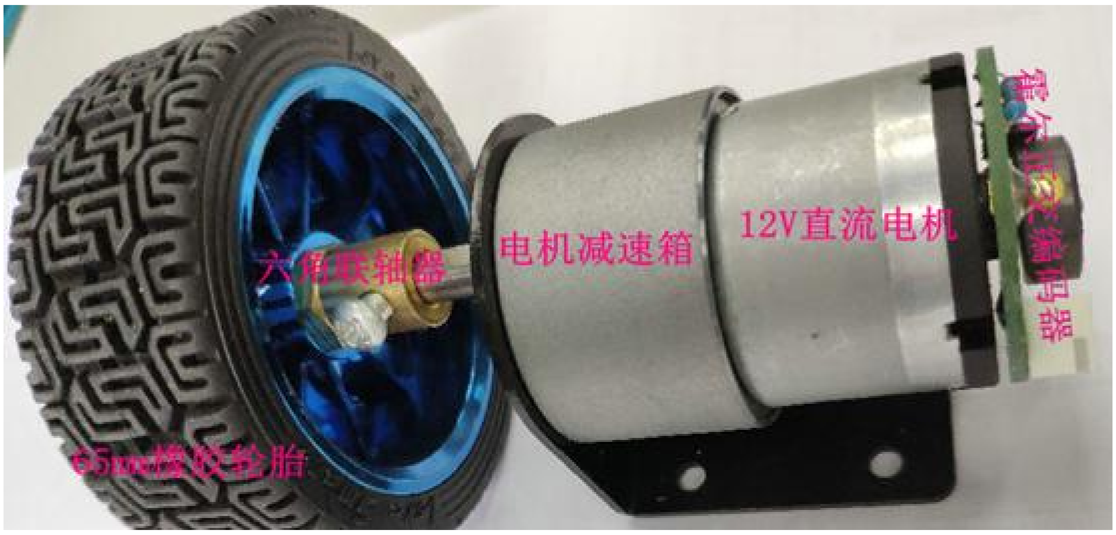
运动底盘是移动机器人的重要组成部分，不像激光雷达、IMU、麦克风、音响、摄像头这些通用部件可以直接买到，很难买到通用的底盘。一方面是因为底盘的尺寸结构和参数是要与具体机器人匹配的；另一方面是因为底盘包含软硬件整套解决方案，是很多机器人公司的核心技术，一般不会随便公开。出于强烈的求知欲与学习热情，我想自己 DIY 一整套两轮差分底盘，并且将完整的设计过程公开出去供大家学习。说干就干，本章节主要内容：

1.stm32 主控硬件设计
2.stm32 主控软件设计
3.底盘通信协议
4.底盘 ROS 驱动开发
5.底盘 PID 控制参数整定
6.底盘里程计标定

1.stm32 主控硬件设计

完整的 stm32 主控硬件包括：带霍尔编码器的直流减速电机、电机驱动、stm32 单片机开发板等配件。

1.1.带霍尔编码器的直流减速电机



（图 1）带霍尔编码器的直流减速电机

要制作一台机器人底盘，需要一套完整的电机部件，就如图 1 中看到的一样，需要有轮胎、联轴器、减速箱、电机和编码器，具体选型可以参考这几个方面的因素：

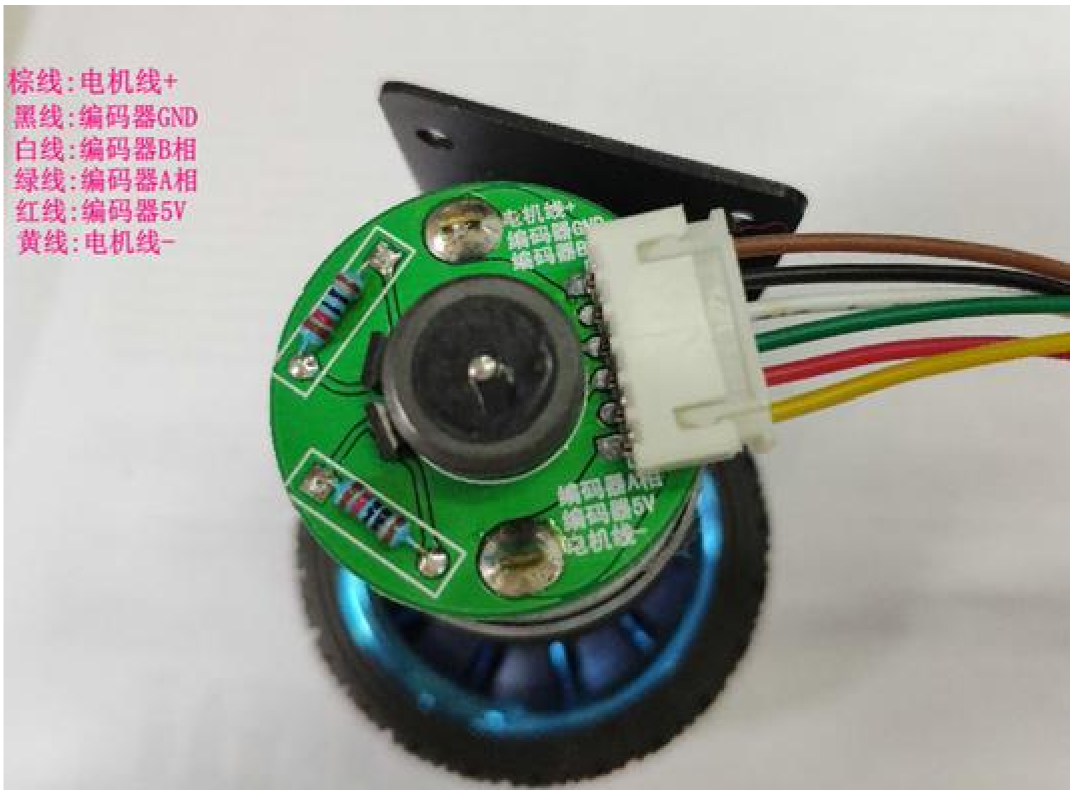
轮胎：直径越大，小车的越障能力越好，但会降低小车爬坡的马力；

联轴器：选择跟轮胎与电机输出轴尺寸相匹配的型号；

减速箱：减速比决定电机输出轴的扭矩，减速比越大，输出轴扭矩越大，但输出轴转速越慢；

电机：一般是 12V 的电机，直流有刷简单易控制；

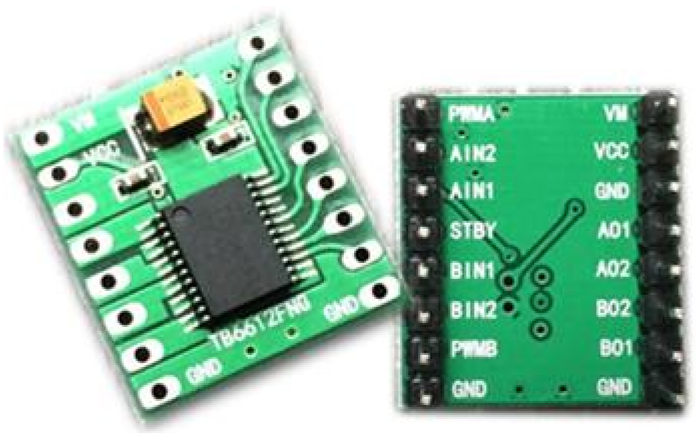
编码器：一般为增量式正交编码器，编码线数根据实际需要精度进行选择。



（图 2）电机接线端口

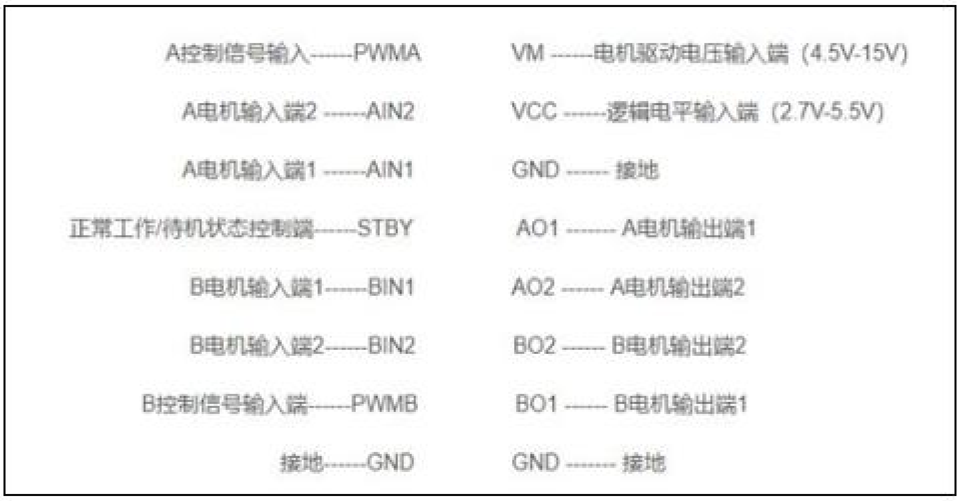
如图 2，可以清楚的看到电机的接线端口，其实电路板上也是有丝印标注的。接线分为两类，一类是电机控制（电机线+、电机线-），另一类是编码器（编码器 5V、编码器 A 相、编码器 B 相、编码器 GND）。

1.2.电机驱动电路



（图 3）TB6612FNG 电机驱动

了解了电机的构造知识后，就来介绍一下如何将电机驱动起来。如图 3 所示，TB6612FNG 是很流行的一款电机驱动芯片，相比于传统的 L298N 效率上提升很多，而且体积大幅减小。TB6612FNG 是双驱动，也就是可以驱动两个电机；TB6612FNG 每通道输出最高 1.2 A 的连续驱动电流，启动峰值电流达 2A/3.2 A（连续脉冲/单脉冲）；4 种电机控制模式：正转/反转/制动/停止；PWM 支持频率高达 100 kHz。



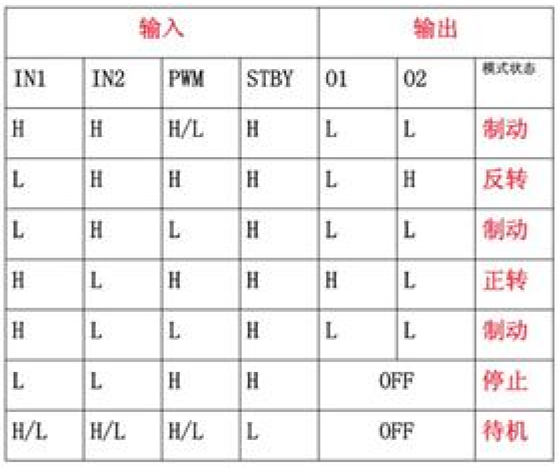
A控制信号输入------PWMA	VM ------电机驱动电压输入端（4.5V-15V）
A电机输入端2 ------AIN2	VCC ------逻辑电平输入端（2.7V-5.5V）
A电机输入端1 ------AIN1	GND ------ 接地
正常工作/待机状态控制端------STBY	AO1 -------- A电机输出端1
B电机输入端1------BIN1	AO2 ------ A电机输出端2
B电机输入端2------BIN2	BO2 ------ B电机输出端2
B控制信号输入端------PWMB	BO1 ------ B电机输出端1
接地------GND	GND -------- 接地

（图 4）TB6612FNG 引脚定义

TB6612FNG 的引脚定义，如图 4 所示，引脚分为电源脚、控制输入脚、控制输出脚。
VM：为电机驱动电压，根据实际电机额定电压选择，推荐使用 12V 供电；
VCC：逻辑电源供电，推荐使用 5V 供电；
STBY：待机/工作状态切换，低电平待机，高电平工作；

PWMA：A 端口电机 PWM 调速信号输入；
AIN1 和 AIN2：A 电机转向控制信号输入；
PWMB：B 端口电机 PWM 调速信号输入；
BIN1 和 BIN2：B 电机转向控制信号输入；

AO1 和 AO2：A 端口电机驱动信号输出；
BO1 和 BO2：B 端口电机驱动信号输出。

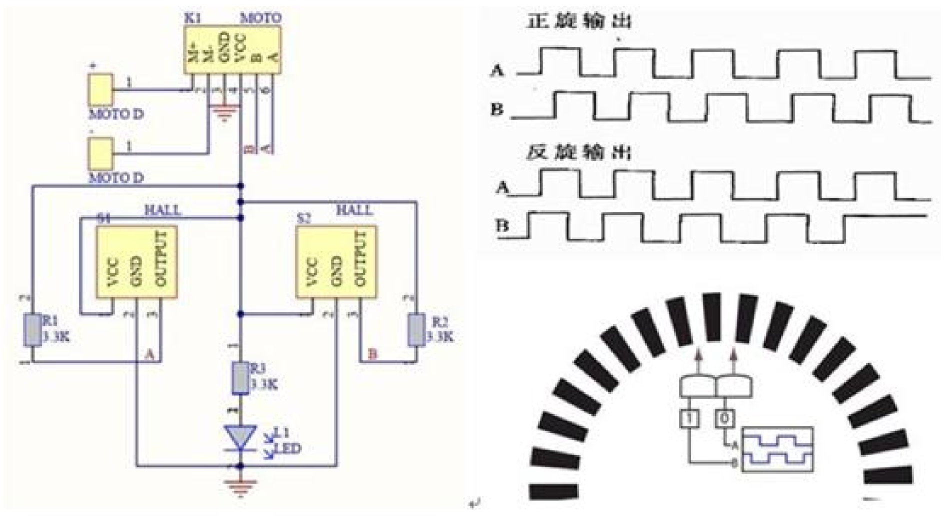


输入				输出		
IN1	IN2	PWM	STBY	01	02	模式状态
H	H	H/L	H	L	L	制动
L	H	H	H	L	H	反转
L	H	L	H	L	L	制动
H	L	H	H	H	L	正转
H	L	L	H	L	L	制动
L	L	H	H	OFF		停止
H/L	H/L	H/L	L	OFF		待机

（图 5）TB6612FNG 控制信号真值表

最后，我们来看一下控制信号的逻辑真值表，如图 5，输入由单片机 IO 口给定，再结合 PWM 信号，便可以实现对电机的正/反转和调速控制了。由于两路电机控制是一模一样的，所以另一路控制信号的逻辑真值表就不重复赘述了。

1.3.霍尔正交编码器原理

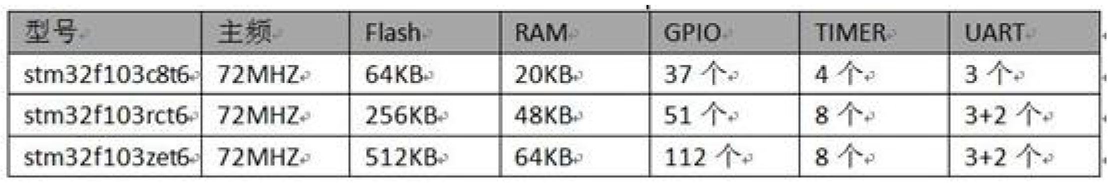


（图 6）霍尔正交编码器原理

如果两个信号相位相差 90 度，则这两个信号称为正交。由于两个信号相位相差 90 度，因此可以根据两个信号哪个先哪个后来判断方向。利用单片机的 IO 口对编码器的 A、B 相进行捕获，很容易得到电机的转速和转向。霍尔正交编码器原理，如图 6。

1.4.stm32 单片机最小系统

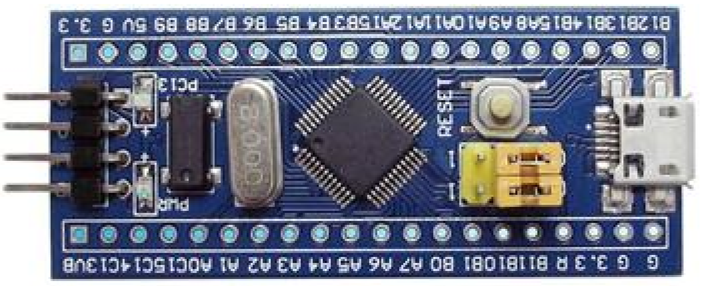
stm32 单片机常用的型号是 stm32f103，根据具体需求的 Flash 容量、RAM 容量、IO 口数量进行选择，下面是常用的一些型号参数对比，如图 7。



型号	主频	Flash	RAM	GPIO	TIMER	UART
stm32f103c8t6	72MHZ	64KB	20KB	37 个	4 个	3 个
stm32f103rct6	72MHZ	256KB	48KB	51 个	8 个	3+2 个
stm32f103zet6	72MHZ	512KB	64KB	112 个	8 个	3+2 个

（图 7）stm32f103 系列单片机参数对比

考虑到 stm32 主控只是用于两个电机的控制，资源开销不算大，需要用到的 IO 口也不是很多，定时器资源也不多，出于性价比考虑推荐 stm32f103c8t6 这个型号。



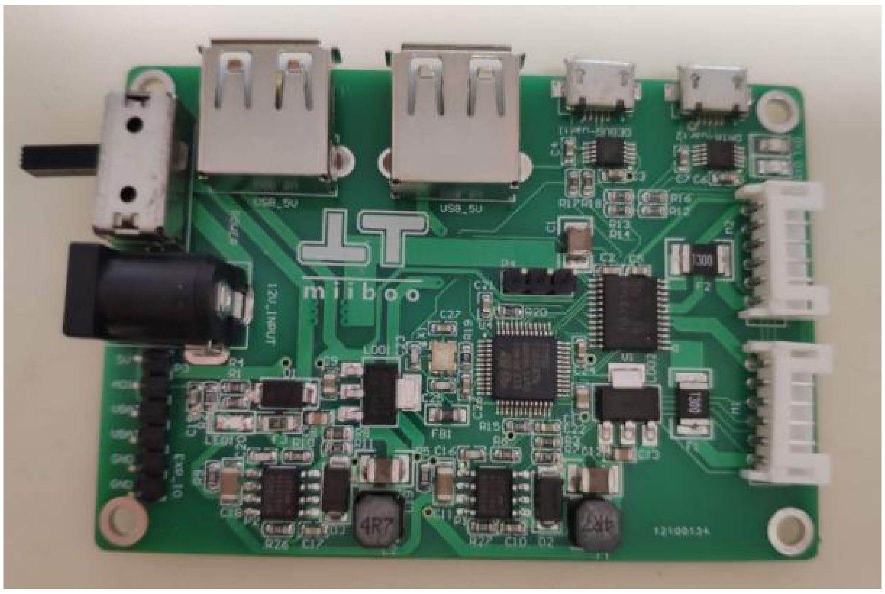
（图 8）stm32f103f103c8t6 最小系统板

如图 8 所示，stm32f103c8t6 最小系统板比较简洁，控制两个电动机，只需要用两个 IO 口输出 2 路 PWM 分别给两个电机调速，用 4 个 IO 口分别控制两个电机的方向，另外 4 个 IO 口分别接两个电机的正交编码器输入，UART1 与 UART2 跟上位机连接分别用于程序 debug

与上层指令控制。

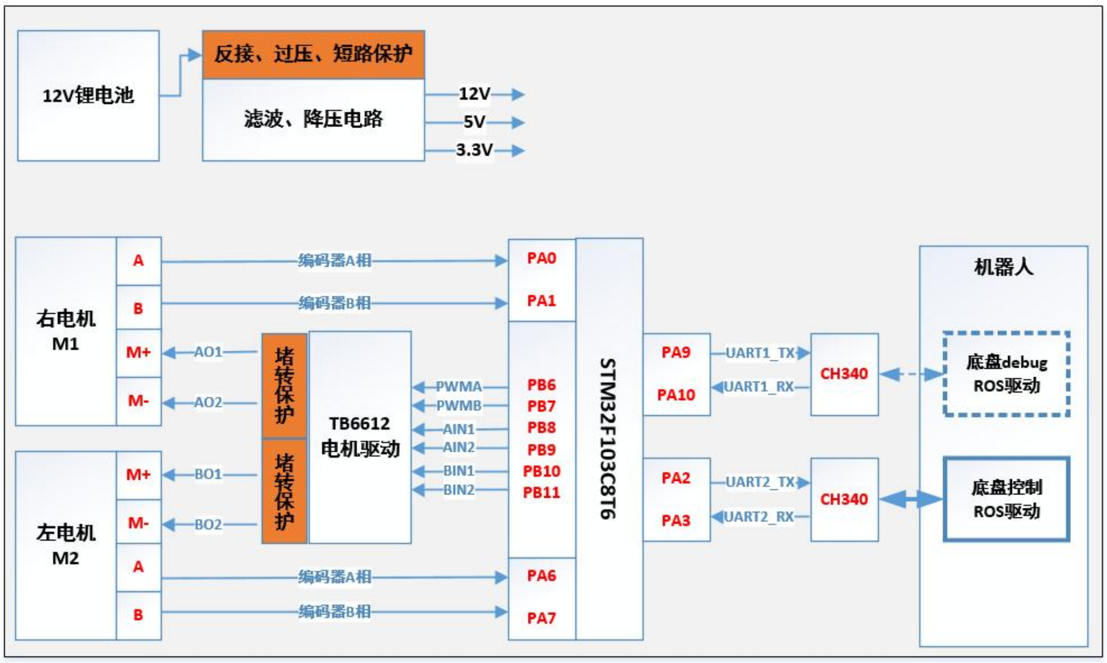
1.5.stm32 主控硬件整体框图

第一个版本的硬件电路是用飞线连接的各个模块，电路稳定性很差，而且外观极其丑陋。痛定思痛，决心老老实实设计电路板，把各模块集成到一个板子上，经过两次改板打样，终于成功了。如图 9，板子简洁美观，而且接插端子布局合理，符合我一向严苛的标准。



（图 9）stm32 主控电路板

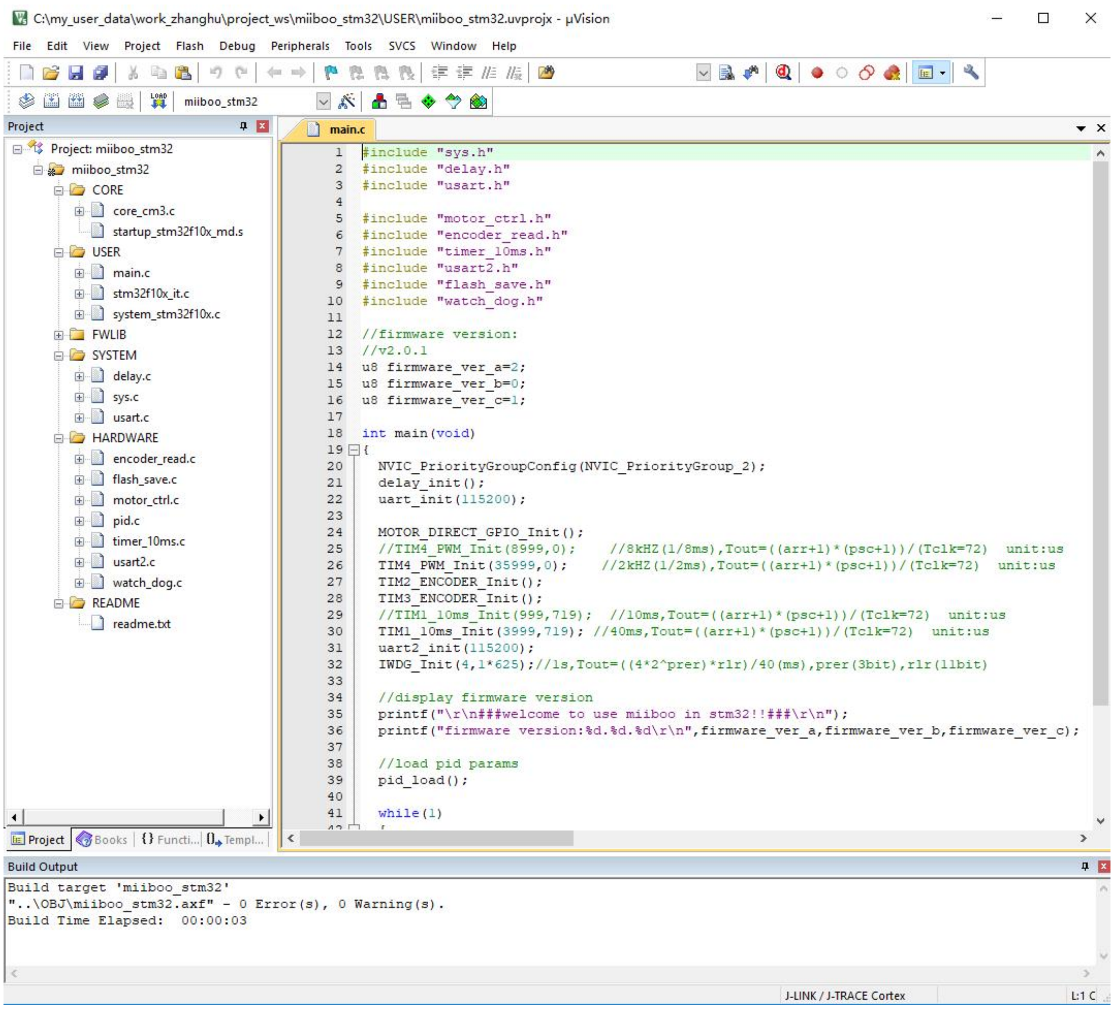
好了，有了这个电路板就好办多了。针对这个电路板，讲讲我的设计思路吧。首先需要设计一个电源系统，用于单片机供电、电机供电、外部设备供电，同时还要考虑电源反接、过压、短路等保护；然后需要设计一个 stm32 单片机最小系统电路；最后围绕 stm32 最小系统，需要设计电机驱动、UART 转 USB、编码器信号捕获这些外围电路，同时还要考虑电机堵转保护、电机对系统电源干扰等问题。逐一采坑之后，差不多就完成设计了。stm32 主控硬件整体框图，如图 10。



（图 10）stm32 主控硬件整体框图

2. stm32 主控软件设计

上一节搭建好了底盘的 stm32 主控硬件，现在就来说说怎么开发配套的 stm32 软件。关于建立 stm32 工程、使用 stm32 开发库、stm32 软件调试方法等基础知识就不多说了，有需要的可以查阅相关资料学习，我觉得 http://www.openedv.com《正点原子》的开发资料写的还可以。我就直接从底盘控制的项目入手，直接进行项目中各个功能需求开始分析讲解，如图 11，是我的底盘控制 stm32 工程项目。



```c
#include "sys.h"
#include "delay.h"
#include "usart.h"

#include "motor_ctrl.h"
#include "encoder_read.h"
#include "timer_10ms.h"
#include "usart2.h"
#include "flash_save.h"
#include "watch_dog.h"

//firmware version:
//v2.0.1
u8 firmware_ver_a=2;
u8 firmware_ver_b=0;
u8 firmware_ver_c=1;

int main(void)
{
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  delay_init();
  uart_init(115200);

  MOTOR_DIRECT_GPIO_Init();
  //TIM4_PWM_Init(8999,0);    //8kHZ(1/8ms),Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM4_PWM_Init(35999,0);    //2kHZ(1/2ms),Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM2_ENCODER_Init();
  TIM3_ENCODER_Init();
  //TIM1_10ms_Init(999,719);  //10ms,Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM1_10ms_Init(3999,719); //40ms,Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  uart2_init(115200);
  IWDG_Init(4,1*625);//1s,Tout=((4*2^prer)*rlr)/40(ms),prer(3bit),rlr(11bit)

  //display firmware version
  printf("\r\n###welcome to use miiboo in stm32!!###\r\n");
  printf("firmware version:%d.%d.%d\r\n",firmware_ver_a,firmware_ver_b,firmware_ver_c);

  //load pid params
  pid_load();

  while(1)
```

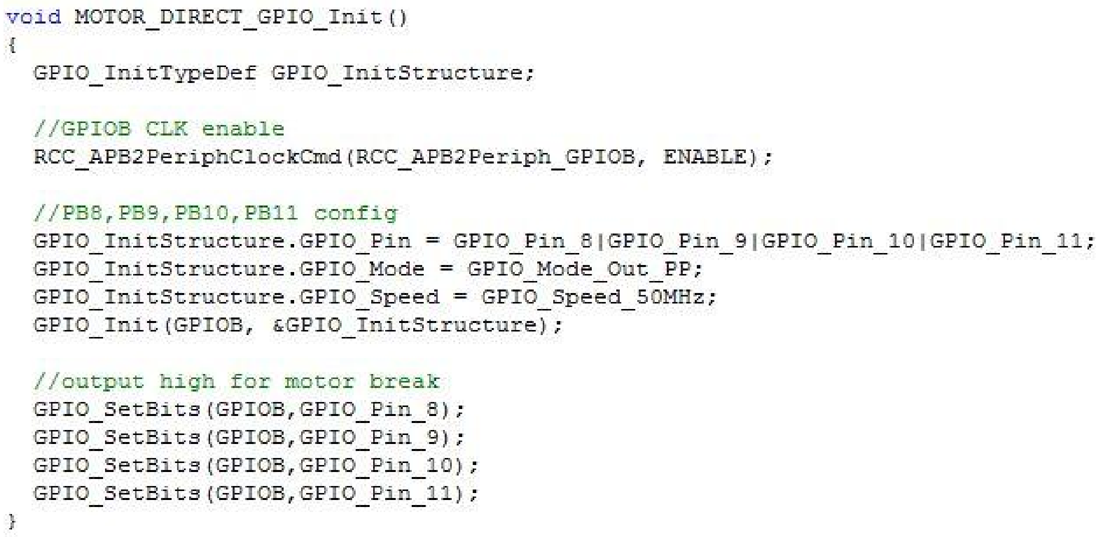
```
Build target 'miiboo_stm32'
"..\OBJ\miiboo_stm32.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:03
```

（图 11）底盘控制 stm32 工程项目

2.1.电机控制

电机控制分为两个部分（电机转向控制、电机转速控制），这些都集成在了电机驱动芯片 TB6612FNG 里面，所以只需要用单片机的 IO 口产生控制转向的高低电平和控制转速的 PWM 波就能实现。

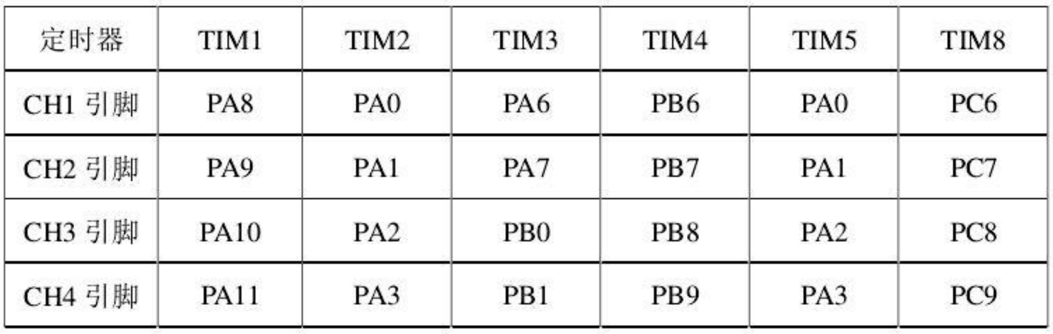
首先，初始化 IO 口作为输出脚，用于产生高低电平输出来控制转向，实例代码如图 12。



```
void MOTOR_DIRECT_GPIO_Init()
{
  GPIO_InitTypeDef GPIO_InitStructure;

  //GPIOB CLK enable
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

  //PB8,PB9,PB10,PB11 config
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  //output high for motor break
  GPIO_SetBits(GPIOB,GPIO_Pin_8);
  GPIO_SetBits(GPIOB,GPIO_Pin_9);
  GPIO_SetBits(GPIOB,GPIO_Pin_10);
  GPIO_SetBits(GPIOB,GPIO_Pin_11);
}
```

（图 12）电机转向控制 IO 口初始化

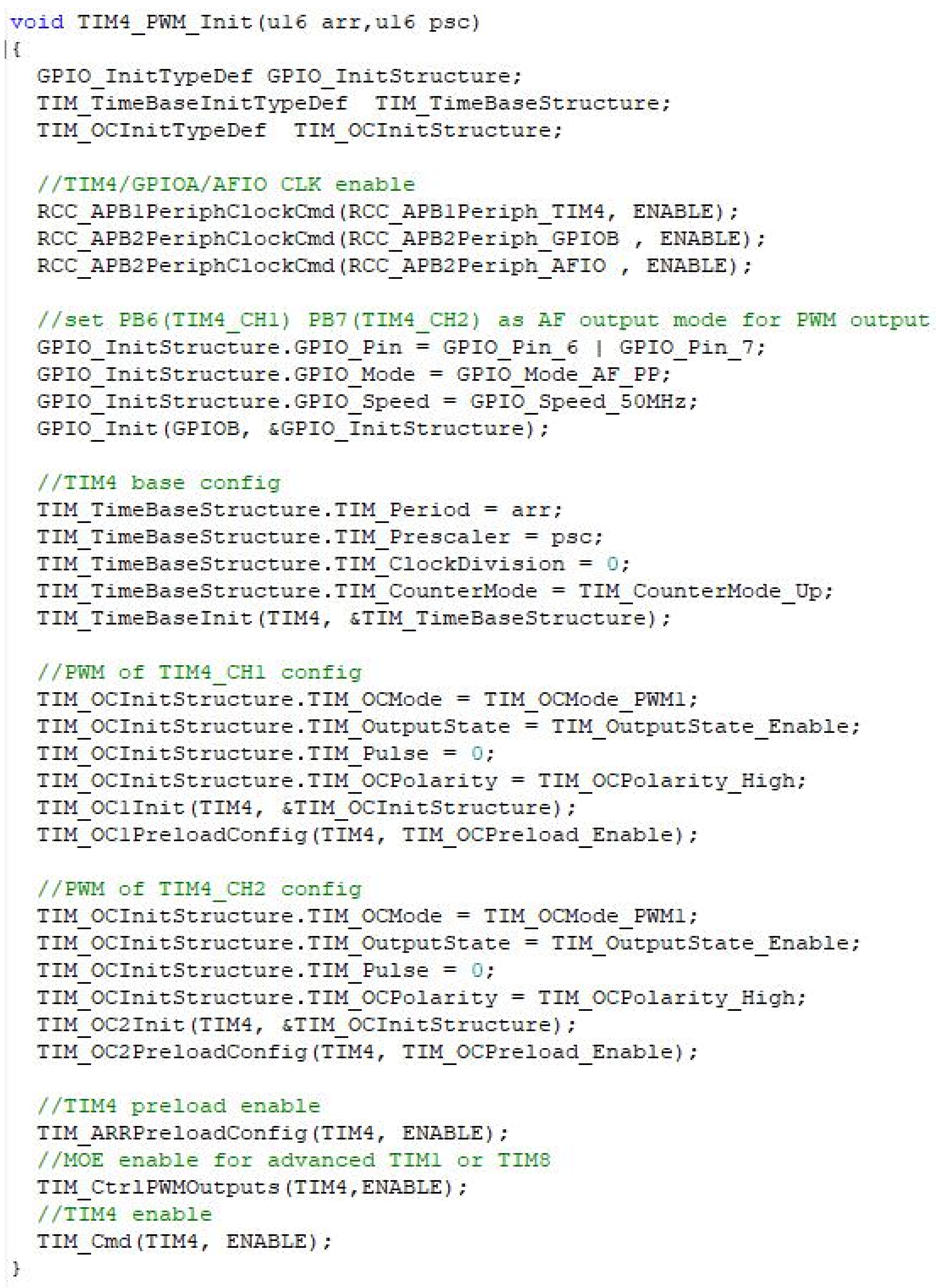
然后，用通用定时器 TIM4 的通道 CH1 和 CH2 分别产生两路 PWM 输出用于两个电机的转速控制，定时器默认引脚分配如图 13。



定时器	TIM1	TIM2	TIM3	TIM4	TIM5	TIM8
CH1 引脚	PA8	PA0	PA6	PB6	PA0	PC6
CH2 引脚	PA9	PA1	PA7	PB7	PA1	PC7
CH3 引脚	PA10	PA2	PB0	PB8	PA2	PC8
CH4 引脚	PA11	PA3	PB1	PB9	PA3	PC9

（图 13）stm32 定时器通道默认引脚分配

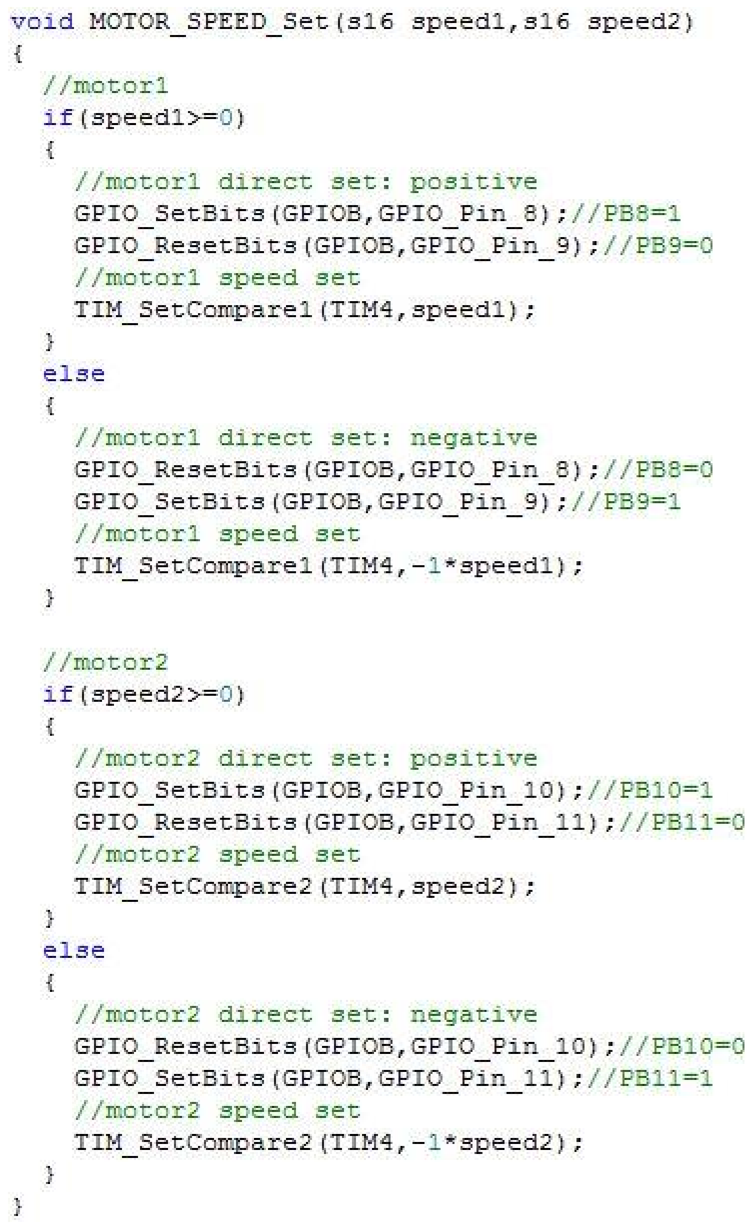
初始化通用定时器 TIM4 的通道 CH1 和 CH2 为 PWM 输出，实例代码如 14。



```
void TIM4_PWM_Init(u16 arr,u16 psc)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_OCInitTypeDef  TIM_OCInitStructure;

  //TIM4/GPIOA/AFIO CLK enable
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB , ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO , ENABLE);

  //set PB6(TIM4_CH1) PB7(TIM4_CH2) as AF output mode for PWM output
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  //TIM4 base config
  TIM_TimeBaseStructure.TIM_Period = arr;
  TIM_TimeBaseStructure.TIM_Prescaler = psc;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);

  //PWM of TIM4_CH1 config
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 0;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC1Init(TIM4, &TIM_OCInitStructure);
  TIM_OC1PreloadConfig(TIM4, TIM_OCPreload_Enable);

  //PWM of TIM4_CH2 config
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 0;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC2Init(TIM4, &TIM_OCInitStructure);
  TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable);

  //TIM4 preload enable
  TIM_ARRPreloadConfig(TIM4, ENABLE);
  //MOE enable for advanced TIM1 or TIM8
  TIM_CtrlPWMOutputs(TIM4,ENABLE);
  //TIM4 enable
  TIM_Cmd(TIM4, ENABLE);
}
```

（图 14）电机转速控制 IO 口初始化

最后，将电机转向和速度控制的操作封装在一个函数中，便于其它地方调用，实例代码如图 15。



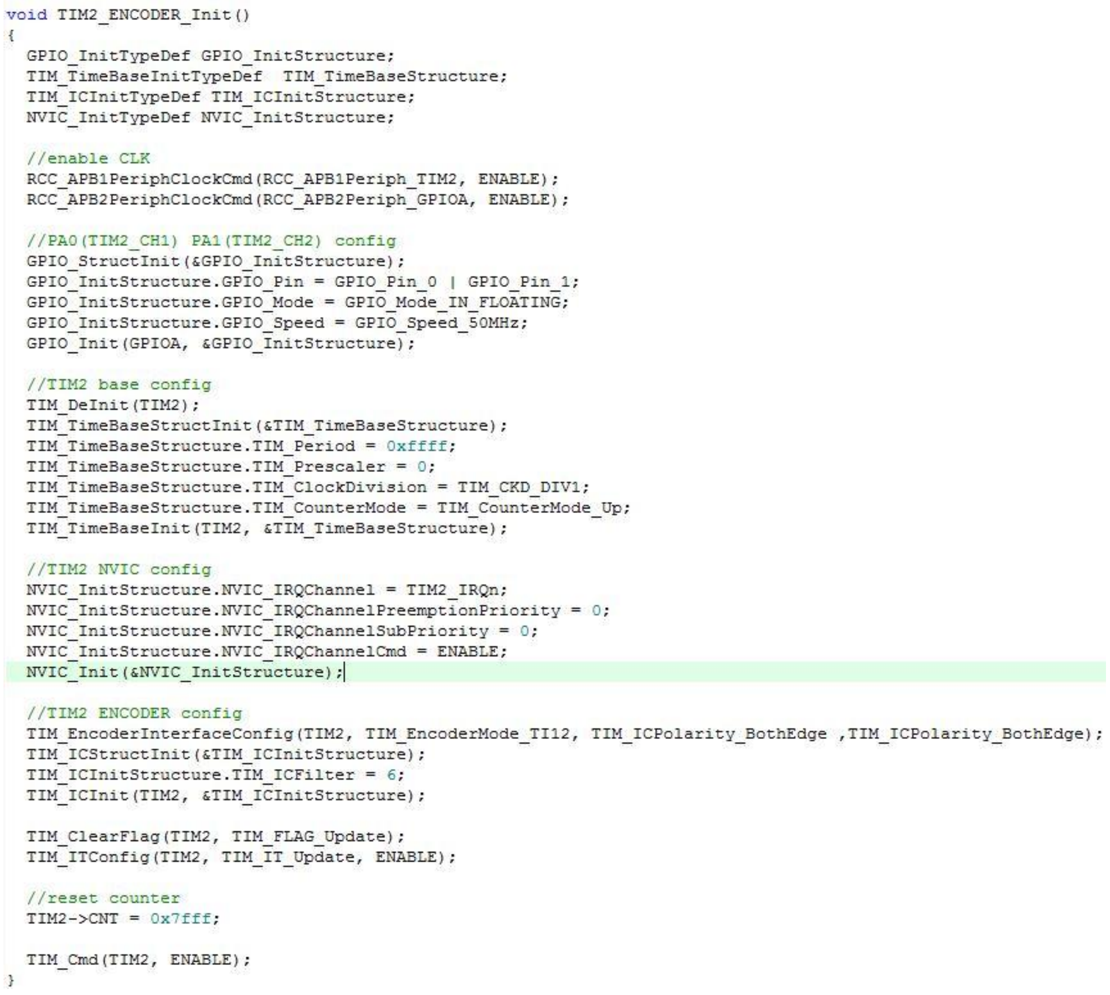
```
void MOTOR_SPEED_Set(s16 speed1,s16 speed2)
{
  //motor1
  if(speed1>=0)
  {
    //motor1 direct set: positive
    GPIO_SetBits(GPIOB,GPIO_Pin_8);//PB8=1
    GPIO_ResetBits(GPIOB,GPIO_Pin_9);//PB9=0
    //motor1 speed set
    TIM_SetCompare1(TIM4,speed1);
  }
  else
  {
    //motor1 direct set: negative
    GPIO_ResetBits(GPIOB,GPIO_Pin_8);//PB8=0
    GPIO_SetBits(GPIOB,GPIO_Pin_9);//PB9=1
    //motor1 speed set
    TIM_SetCompare1(TIM4,-1*speed1);
  }

  //motor2
  if(speed2>=0)
  {
    //motor2 direct set: positive
    GPIO_SetBits(GPIOB,GPIO_Pin_10);//PB10=1
    GPIO_ResetBits(GPIOB,GPIO_Pin_11);//PB11=0
    //motor2 speed set
    TIM_SetCompare2(TIM4,speed2);
  }
  else
  {
    //motor2 direct set: negative
    GPIO_ResetBits(GPIOB,GPIO_Pin_10);//PB10=0
    GPIO_SetBits(GPIOB,GPIO_Pin_11);//PB11=1
    //motor2 speed set
    TIM_SetCompare2(TIM4,-1*speed2);
  }
}
```

（图 15）电机转向和速度控制封装

2.2.编码器数据读取

编码器对底盘来说至关重要，一方面底盘通过编码器的反馈进行 PID 闭环速度控制，另一方面底盘通过编码器进行航迹推演得到里程计用于后续的定位与导航等高级算法中。这里用到的编码器是正交编码器，所以直接使用通用定时器的输入捕获中的编码器模式来读取编码器。采用通用定时器 TIM2 的通道 CH1 和 CH2 捕获 encoder1 的 A 相和 B 相脉冲，采用通用定时器 TIM3 的通道 CH1 和 CH2 捕获 encoder2 的 A 相和 B 相脉冲。

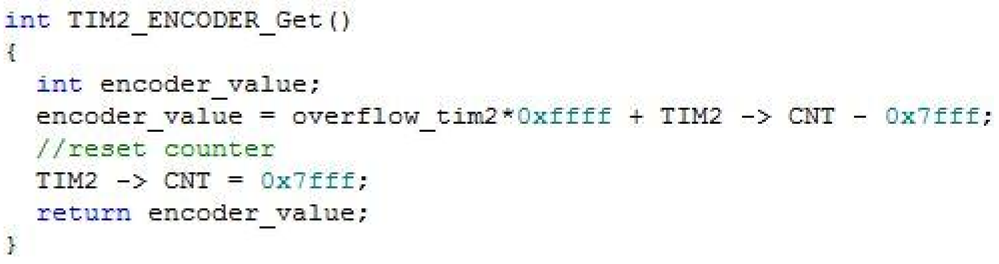
先初始化 TIM2 作为编码器 encoder1 的捕获，实例代码如图 16。



```
void TIM2_ENCODER_Init()
{
  GPIO_InitTypeDef GPIO_InitStructure;
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_ICInitTypeDef TIM_ICInitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  //enable CLK
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  //PA0(TIM2_CH1) PA1(TIM2_CH2) config
  GPIO_StructInit(&GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  //TIM2 base config
  TIM_DeInit(TIM2);
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  TIM_TimeBaseStructure.TIM_Period = 0xffff;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

  //TIM2 NVIC config
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  //TIM2 ENCODER config
  TIM_EncoderInterfaceConfig(TIM2, TIM_EncoderMode_TI12, TIM_ICPolarity_BothEdge ,TIM_ICPolarity_BothEdge);
  TIM_ICStructInit(&TIM_ICInitStructure);
  TIM_ICInitStructure.TIM_ICFilter = 6;
  TIM_ICInit(TIM2, &TIM_ICInitStructure);

  TIM_ClearFlag(TIM2, TIM_FLAG_Update);
  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

  //reset counter
  TIM2->CNT = 0x7fff;

  TIM_Cmd(TIM2, ENABLE);
}
```

（图 16）初始化 TIM2 作为编码器 encoder1 的捕获

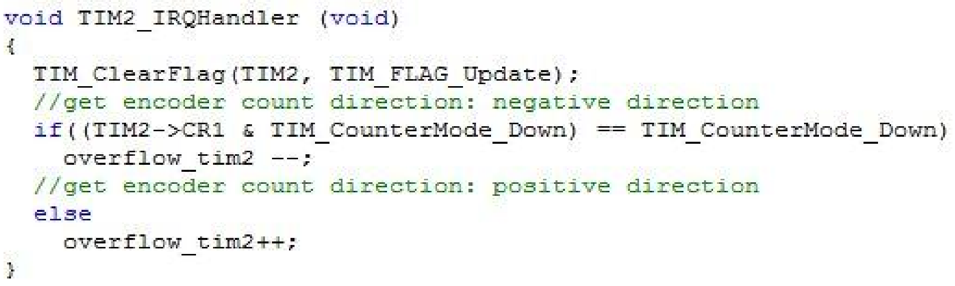
然后，将读取编码器计数值的操作封装在一个函数中，便于其它地方调用，实例代码如图 17。



```
int TIM2_ENCODER_Get()
{
  int encoder_value;
  encoder_value = overflow_tim2*0xffff + TIM2 -> CNT - 0x7fff;
  //reset counter
  TIM2 -> CNT = 0x7fff;
  return encoder_value;
}
```

（图 17）读取编码器 encoder1 计数值封装

最后，编写 TIM2 计数溢出时的中断处理函数，实例代码如图 18。



```
void TIM2_IRQHandler (void)
{
  TIM_ClearFlag(TIM2, TIM_FLAG_Update);
  //get encoder count direction: negative direction
  if((TIM2->CR1 & TIM_CounterMode_Down) == TIM_CounterMode_Down)
    overflow_tim2 --;
  //get encoder count direction: positive direction
  else
    overflow_tim2++;
}
```

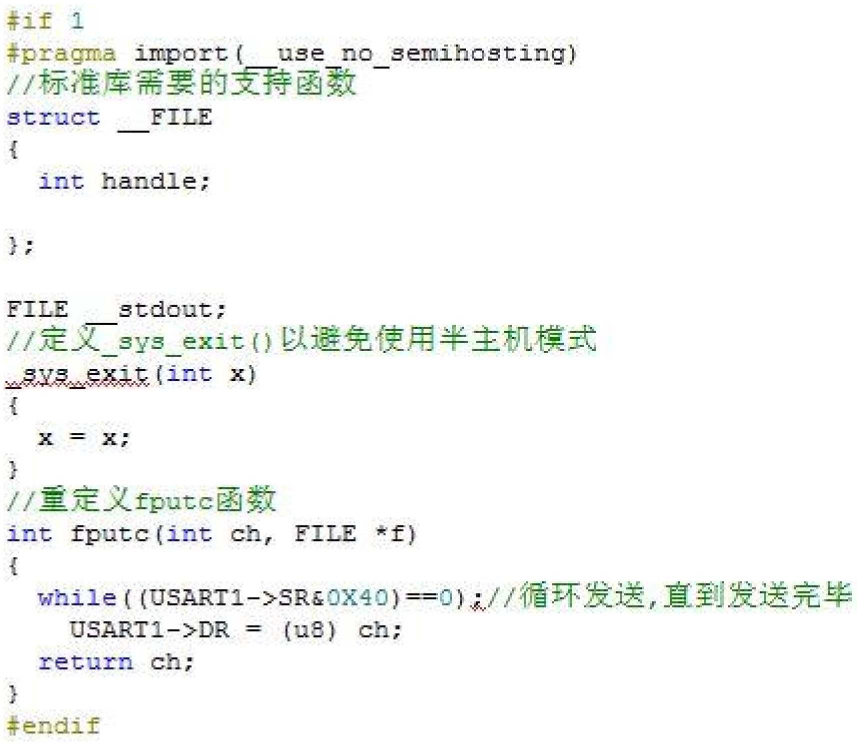
（图 18）TIM2 计数溢出中断处理函数

同理可得 TIM3 捕获 encoder2 的代码实现，这里就不在赘述了。

2.3.串口数据收发

串口 2 是数据接口，负责接收上位机发送过来的控制指令，同时将编码器值返回给上位机；串口 1 是 debug 接口，负责接收上位机发送过来的版本信息请求、PIDm 默认值恢复、PID 值设定等调试指令，同时将程序中的 debug 打印信息返回给上位机。但是在底盘正常工作时，只需要连接串口 2；串口 1 是预留出来给有需要自己动手修改 PID 参数使用的。

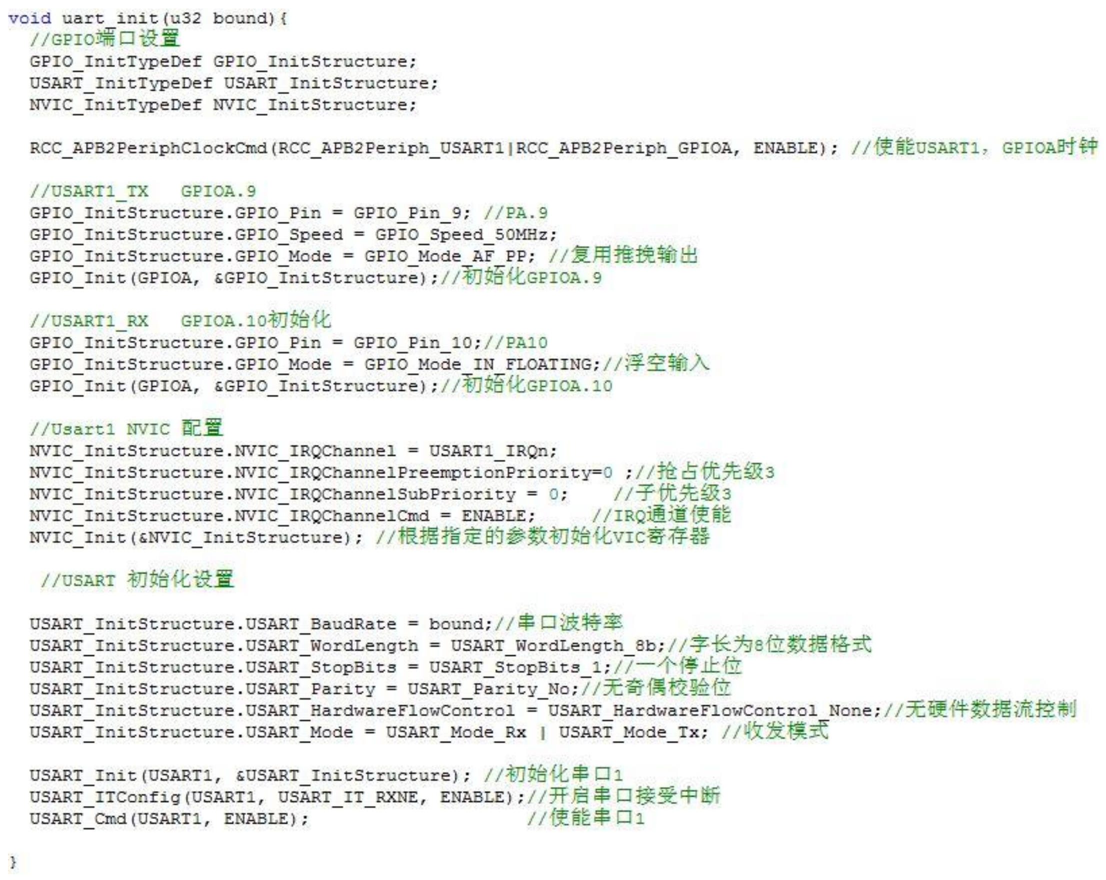
首先，配置串口 1，先对串口 1 的输出进行 printf 函数打印支持，实例代码如图 19。



```c
#if 1
#pragma import(__use_no_semihosting)
//标准库需要的支持函数
struct __FILE
{
  int handle;

};

FILE __stdout;
//定义_sys_exit()以避免使用半主机模式
_sys_exit(int x)
{
  x = x;
}
//重定义fputc函数
int fputc(int ch, FILE *f)
{
  while((USART1->SR&0X40)==0);//循环发送,直到发送完毕
    USART1->DR = (u8) ch;
  return ch;
}
#endif
```

（图 19）串口 1 的输出进行 printf 函数打印支持

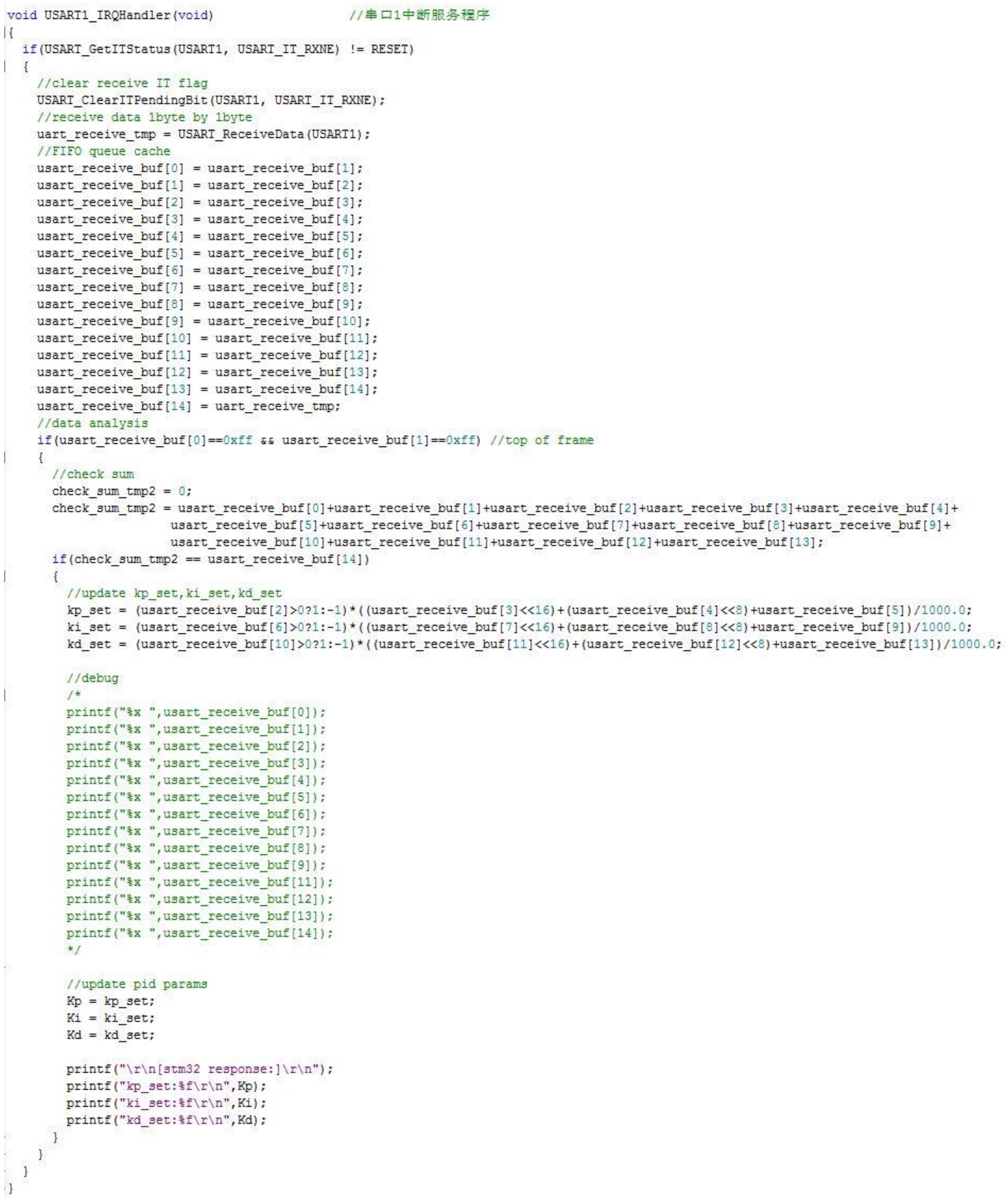
然后，初始化串口 1，实例代码如图 20。



```
void uart_init(u32 bound){
  //GPIO端口设置
  GPIO_InitTypeDef GPIO_InitStructure;
  USART_InitTypeDef USART_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|RCC_APB2Periph_GPIOA, ENABLE); //使能USART1，GPIOA时钟

  //USART1_TX   GPIOA.9
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
  GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化GPIOA.9

  //USART1_RX   GPIOA.10初始化
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;//PA10
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//浮空输入
  GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化GPIOA.10

  //Usart1 NVIC 配置
  NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0 ;//抢占优先级3
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;     //子优先级3
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;      //IRQ通道使能
  NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化VIC寄存器

   //USART 初始化设置

  USART_InitStructure.USART_BaudRate = bound;//串口波特率
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为8位数据格式
  USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
  USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无硬件数据流控制
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式

  USART_Init(USART1, &USART_InitStructure); //初始化串口1
  USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);//开启串口接受中断
  USART_Cmd(USART1, ENABLE);                    //使能串口1

}
```

（图 20）初始化串口 1

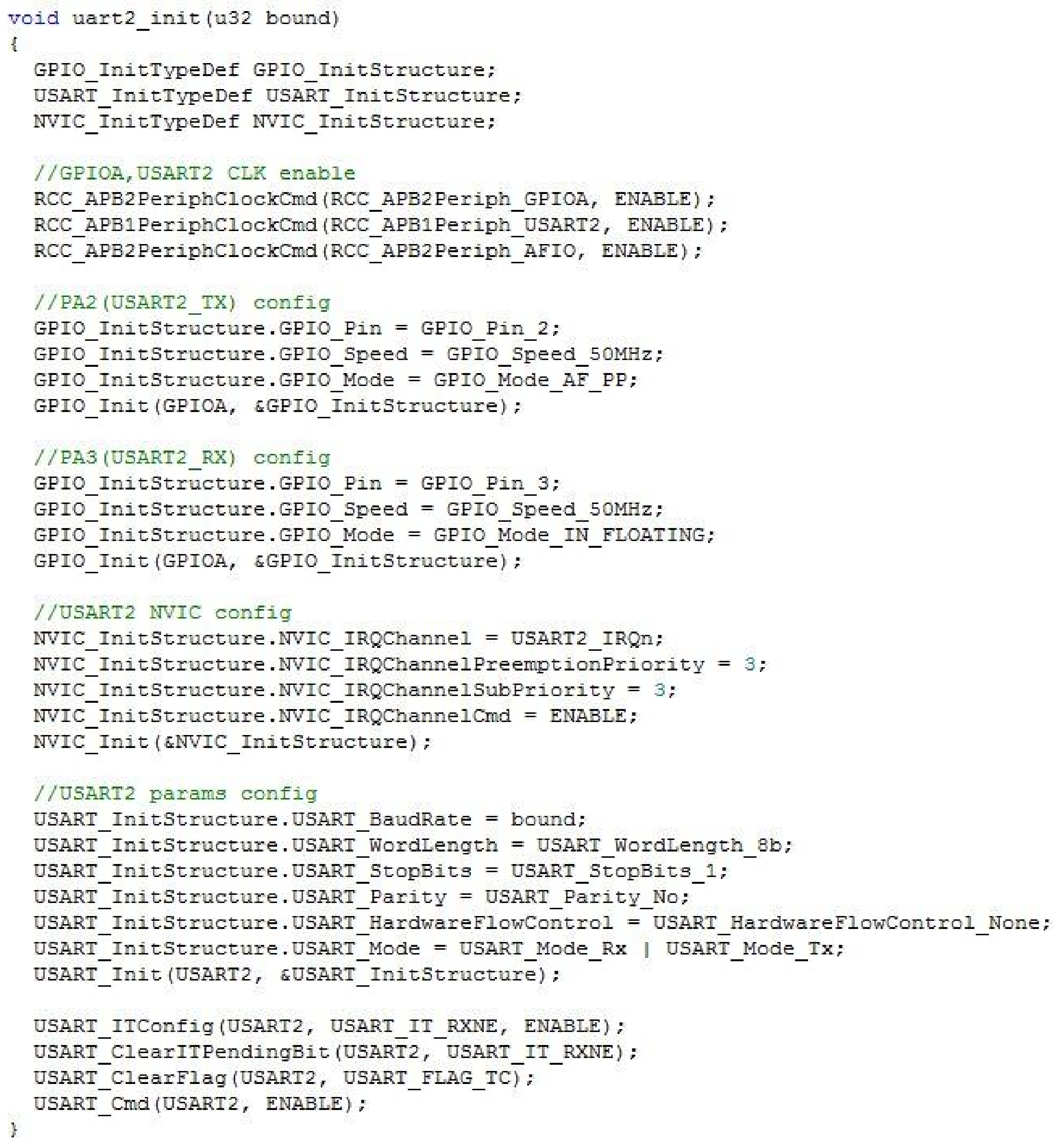
最后，编写串口 1 接收中断处理函数，此函数主要进行对上位机发过来的数据进行协议解析，实例代码如图 21。



```
void USART1_IRQHandler(void)                //串口1中断服务程序
{
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
  {
    //clear receive IT flag
    USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    //receive data 1byte by 1byte
    uart_receive_tmp = USART_ReceiveData(USART1);
    //FIFO queue cache
    usart_receive_buf[0] = usart_receive_buf[1];
    usart_receive_buf[1] = usart_receive_buf[2];
    usart_receive_buf[2] = usart_receive_buf[3];
    usart_receive_buf[3] = usart_receive_buf[4];
    usart_receive_buf[4] = usart_receive_buf[5];
    usart_receive_buf[5] = usart_receive_buf[6];
    usart_receive_buf[6] = usart_receive_buf[7];
    usart_receive_buf[7] = usart_receive_buf[8];
    usart_receive_buf[8] = usart_receive_buf[9];
    usart_receive_buf[9] = usart_receive_buf[10];
    usart_receive_buf[10] = usart_receive_buf[11];
    usart_receive_buf[11] = usart_receive_buf[12];
    usart_receive_buf[12] = usart_receive_buf[13];
    usart_receive_buf[13] = usart_receive_buf[14];
    usart_receive_buf[14] = uart_receive_tmp;
    //data analysis
    if(usart_receive_buf[0]==0xff && usart_receive_buf[1]==0xff) //top of frame
    {
      //check sum
      check_sum_tmp2 = 0;
      check_sum_tmp2 = usart_receive_buf[0]+usart_receive_buf[1]+usart_receive_buf[2]+usart_receive_buf[3]+usart_receive_buf[4]+
                       usart_receive_buf[5]+usart_receive_buf[6]+usart_receive_buf[7]+usart_receive_buf[8]+usart_receive_buf[9]+
                       usart_receive_buf[10]+usart_receive_buf[11]+usart_receive_buf[12]+usart_receive_buf[13];
      if(check_sum_tmp2 == usart_receive_buf[14])
      {
        //update kp_set,ki_set,kd_set
        kp_set = (usart_receive_buf[2]>0?1:-1)*((usart_receive_buf[3]<<16)+(usart_receive_buf[4]<<8)+usart_receive_buf[5])/1000.0;
        ki_set = (usart_receive_buf[6]>0?1:-1)*((usart_receive_buf[7]<<16)+(usart_receive_buf[8]<<8)+usart_receive_buf[9])/1000.0;
        kd_set = (usart_receive_buf[10]>0?1:-1)*((usart_receive_buf[11]<<16)+(usart_receive_buf[12]<<8)+usart_receive_buf[13])/1000.0;

        //debug
        /*
        printf("%x ",usart_receive_buf[0]);
        printf("%x ",usart_receive_buf[1]);
        printf("%x ",usart_receive_buf[2]);
        printf("%x ",usart_receive_buf[3]);
        printf("%x ",usart_receive_buf[4]);
        printf("%x ",usart_receive_buf[5]);
        printf("%x ",usart_receive_buf[6]);
        printf("%x ",usart_receive_buf[7]);
        printf("%x ",usart_receive_buf[8]);
        printf("%x ",usart_receive_buf[9]);
        printf("%x ",usart_receive_buf[11]);
        printf("%x ",usart_receive_buf[12]);
        printf("%x ",usart_receive_buf[13]);
        printf("%x ",usart_receive_buf[14]);
        */

        //update pid params
        Kp = kp_set;
        Ki = ki_set;
        Kd = kd_set;

        printf("\r\n[stm32 response:]\r\n");
        printf("kp_set:%f\r\n",Kp);
        printf("ki_set:%f\r\n",Ki);
        printf("kd_set:%f\r\n",Kd);
      }
    }
  }
}
```

（图 21）串口 1 接收中断处理函数

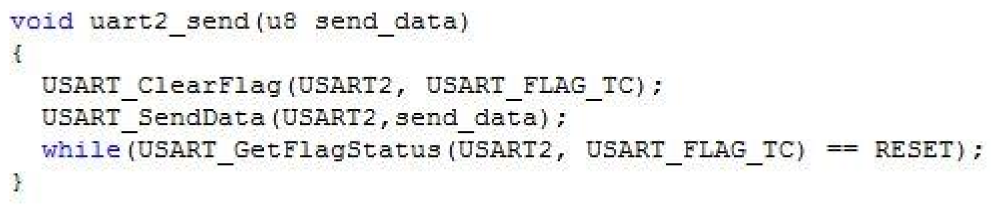
接下来，介绍串口 2，初始化串口 2，实例代码如图 22。



```
void uart2_init(u32 bound)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  USART_InitTypeDef USART_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  //GPIOA,USART2 CLK enable
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

  //PA2(USART2_TX) config
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  //PA3(USART2_RX) config
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  //USART2 NVIC config
  NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  //USART2 params config
  USART_InitStructure.USART_BaudRate = bound;
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure.USART_StopBits = USART_StopBits_1;
  USART_InitStructure.USART_Parity = USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
  USART_Init(USART2, &USART_InitStructure);

  USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
  USART_ClearITPendingBit(USART2, USART_IT_RXNE);
  USART_ClearFlag(USART2, USART_FLAG_TC);
  USART_Cmd(USART2, ENABLE);
}
```

（图 22）初始化串口 2

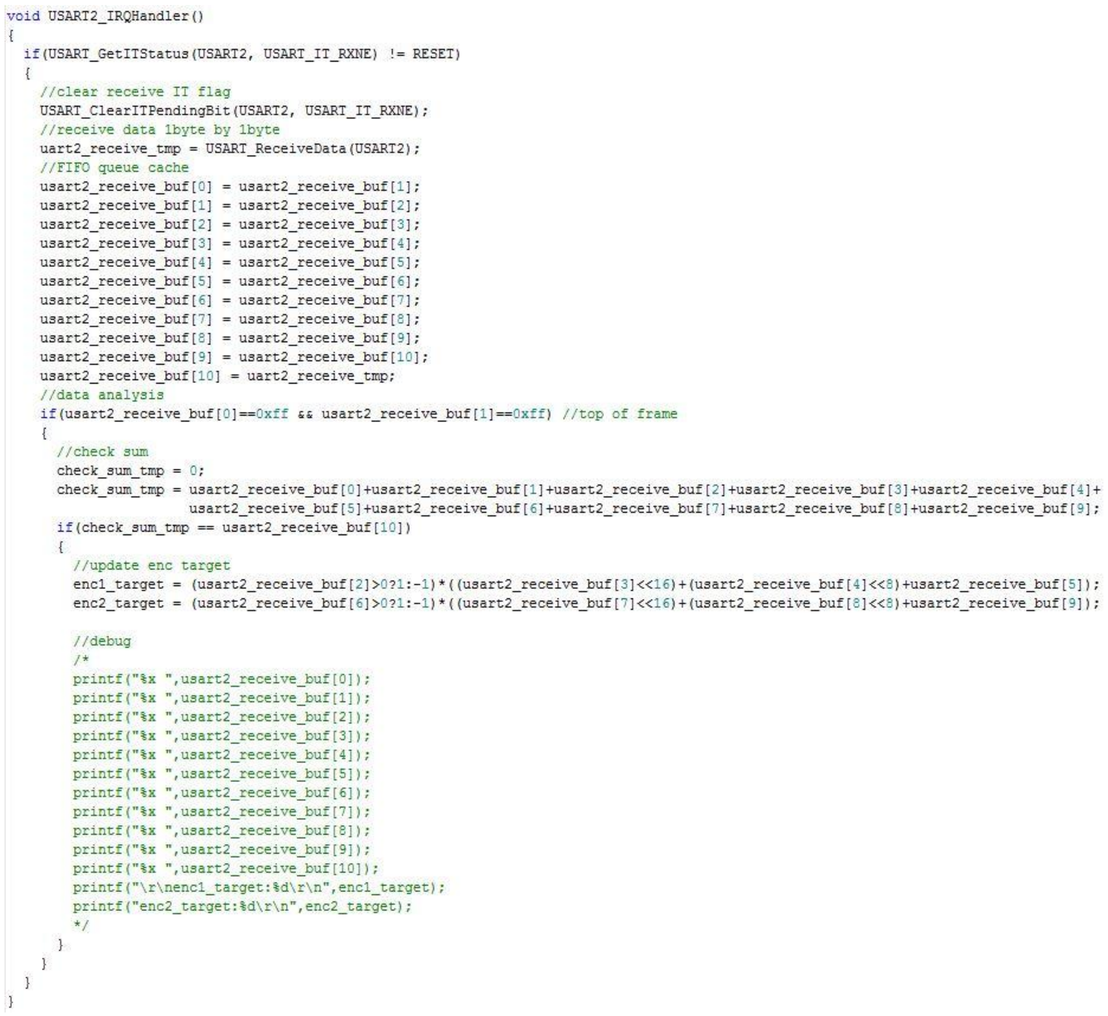
然后，将串口 2 发送数据的操作封装到函数中，便于其它地方调用，实例代码如图 23。



```
void uart2_send(u8 send_data)
{
  USART_ClearFlag(USART2, USART_FLAG_TC);
  USART_SendData(USART2,send_data);
  while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
}
```

（图 23）串口 2 发送数据封装

最后，编写串口 2 接收中断处理函数，此函数主要进行对上位机发过来的数据进行协议解析，实例代码如图 24。



```
void USART2_IRQHandler()
{
  if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
  {
    //clear receive IT flag
    USART_ClearITPendingBit(USART2, USART_IT_RXNE);
    //receive data 1byte by 1byte
    uart2_receive_tmp = USART_ReceiveData(USART2);
    //FIFO queue cache
    usart2_receive_buf[0] = usart2_receive_buf[1];
    usart2_receive_buf[1] = usart2_receive_buf[2];
    usart2_receive_buf[2] = usart2_receive_buf[3];
    usart2_receive_buf[3] = usart2_receive_buf[4];
    usart2_receive_buf[4] = usart2_receive_buf[5];
    usart2_receive_buf[5] = usart2_receive_buf[6];
    usart2_receive_buf[6] = usart2_receive_buf[7];
    usart2_receive_buf[7] = usart2_receive_buf[8];
    usart2_receive_buf[8] = usart2_receive_buf[9];
    usart2_receive_buf[9] = usart2_receive_buf[10];
    usart2_receive_buf[10] = uart2_receive_tmp;
    //data analysis
    if(usart2_receive_buf[0]==0xff && usart2_receive_buf[1]==0xff) //top of frame
    {
      //check sum
      check_sum_tmp = 0;
      check_sum_tmp = usart2_receive_buf[0]+usart2_receive_buf[1]+usart2_receive_buf[2]+usart2_receive_buf[3]+usart2_receive_buf[4]+
                      usart2_receive_buf[5]+usart2_receive_buf[6]+usart2_receive_buf[7]+usart2_receive_buf[8]+usart2_receive_buf[9];
      if(check_sum_tmp == usart2_receive_buf[10])
      {
        //update enc target
        enc1_target = (usart2_receive_buf[2]>0?1:-1)*((usart2_receive_buf[3]<<16)+(usart2_receive_buf[4]<<8)+usart2_receive_buf[5]);
        enc2_target = (usart2_receive_buf[6]>0?1:-1)*((usart2_receive_buf[7]<<16)+(usart2_receive_buf[8]<<8)+usart2_receive_buf[9]);

        //debug
        /*
        printf("%x ",usart2_receive_buf[0]);
        printf("%x ",usart2_receive_buf[1]);
        printf("%x ",usart2_receive_buf[2]);
        printf("%x ",usart2_receive_buf[3]);
        printf("%x ",usart2_receive_buf[4]);
        printf("%x ",usart2_receive_buf[5]);
        printf("%x ",usart2_receive_buf[6]);
        printf("%x ",usart2_receive_buf[7]);
        printf("%x ",usart2_receive_buf[8]);
        printf("%x ",usart2_receive_buf[9]);
        printf("%x ",usart2_receive_buf[10]);
        printf("\r\nenc1_target:%d\r\n",enc1_target);
        printf("enc2_target:%d\r\n",enc2_target);
        */
      }
    }
  }
}
```

（图 24）串口 2 接收中断处理函数

到这里，串口有 1 和串口 2 的数据发送与接收都编写好了，依据我们定义的 usart2 数据通信协议和 usart1 调试通信协议，上位机就可以编写对应的程序来跟底盘的串口 2 和串口 1 进行通信了。关于通信协议的具体内容，将在后续做展开。

2.4.电机速度 PID 控制

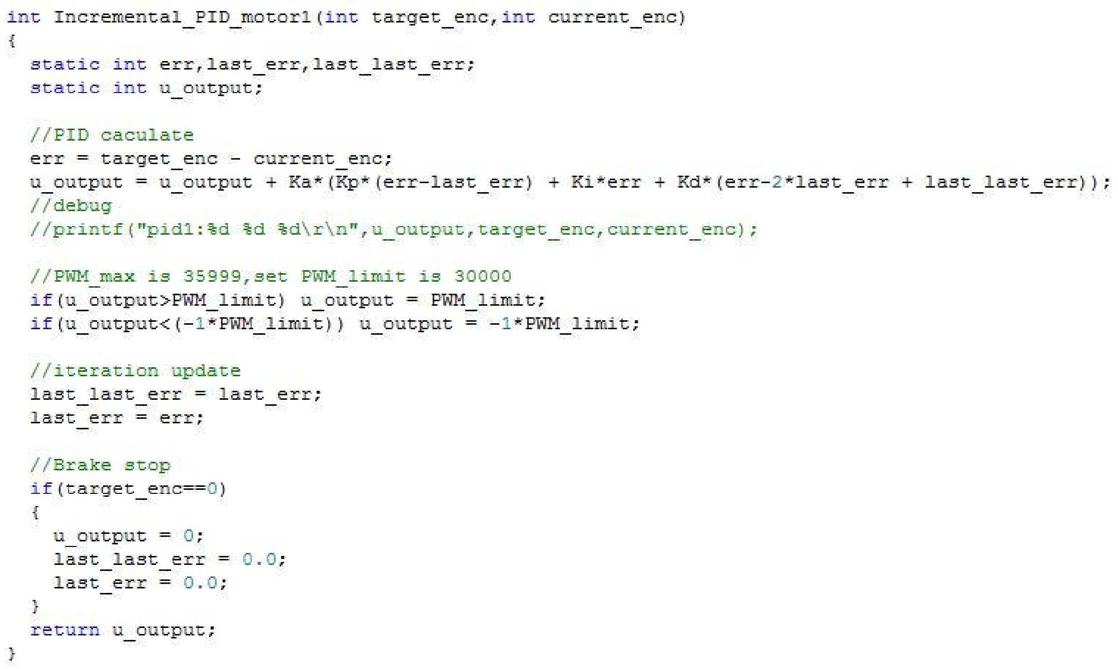
我在底盘中采用的是增量型 PID 算法，编程涉及到的数学表达式有 3 个，分别是：

e(k) = target_value - current_value

delta_u(k) = Kp*[e(k)-e(k-1)] + Ki*e(k) + Kd*[e(k)-2*e(k-1)+e(k-2)]

u(k) = u(k-1) + delta_u(k)

将这 3 个数学表达式封装到函数中，便于其它地方调用，实例代码如图 25。



```
int Incremental_PID_motor1(int target_enc,int current_enc)
{
  static int err,last_err,last_last_err;
  static int u_output;

  //PID caculate
  err = target_enc - current_enc;
  u_output = u_output + Ka*(Kp*(err-last_err) + Ki*err + Kd*(err-2*last_err + last_last_err));
  //debug
  //printf("pid1:%d %d %d\r\n",u_output,target_enc,current_enc);

  //PWM_max is 35999,set PWM_limit is 30000
  if(u_output>PWM_limit) u_output = PWM_limit;
  if(u_output<(-1*PWM_limit)) u_output = -1*PWM_limit;

  //iteration update
  last_last_err = last_err;
  last_err = err;

  //Brake stop
  if(target_enc==0)
  {
    u_output = 0;
    last_last_err = 0.0;
    last_err = 0.0;
  }
  return u_output;
}
```

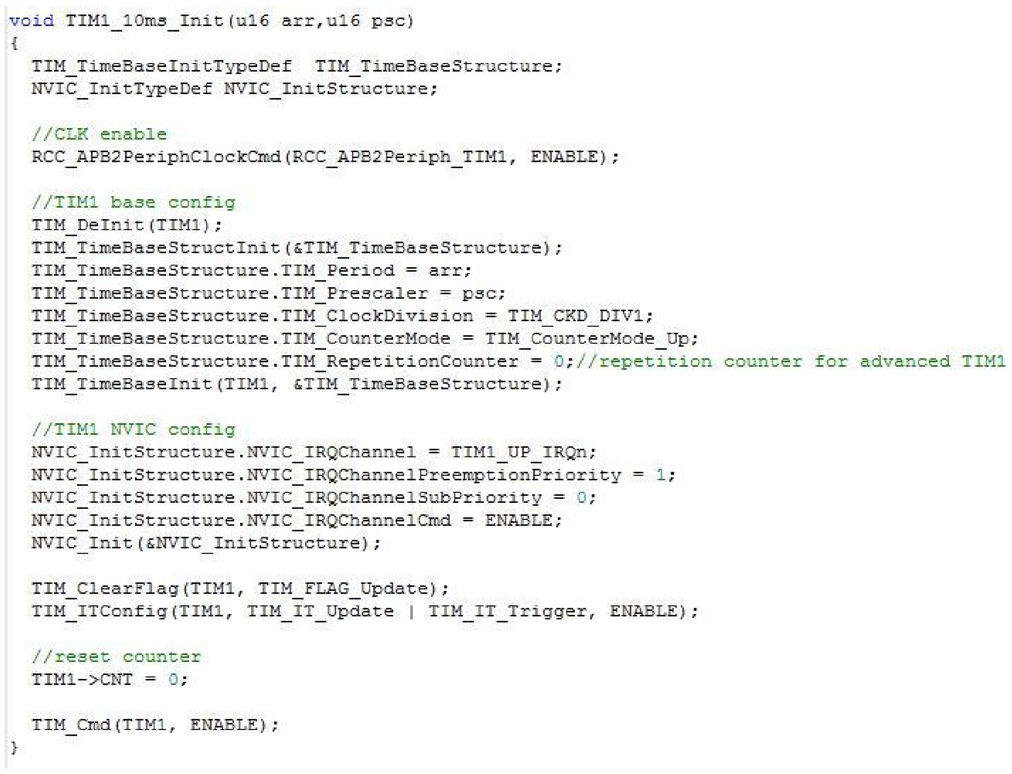
（图 25）串口 2 接收中断处理函数

电机 1 与电机 2 采用同样的 PID 算法，所以电机 2 的 PID 算法代码实现就不赘述了。关于 PID 参数的整定方法，将在后续做展开。

2.5.周期性控制

通过上面的讲解，各个模块的驱动代码都准备就绪了，现在需要产生一个周期性的过程，在里面实现编码器计数值采样、PID 控制等具体实现。这里采用定时器 TIM1 产生一个周期性的中断，在中断处理函数中实现各模块的具体操作。

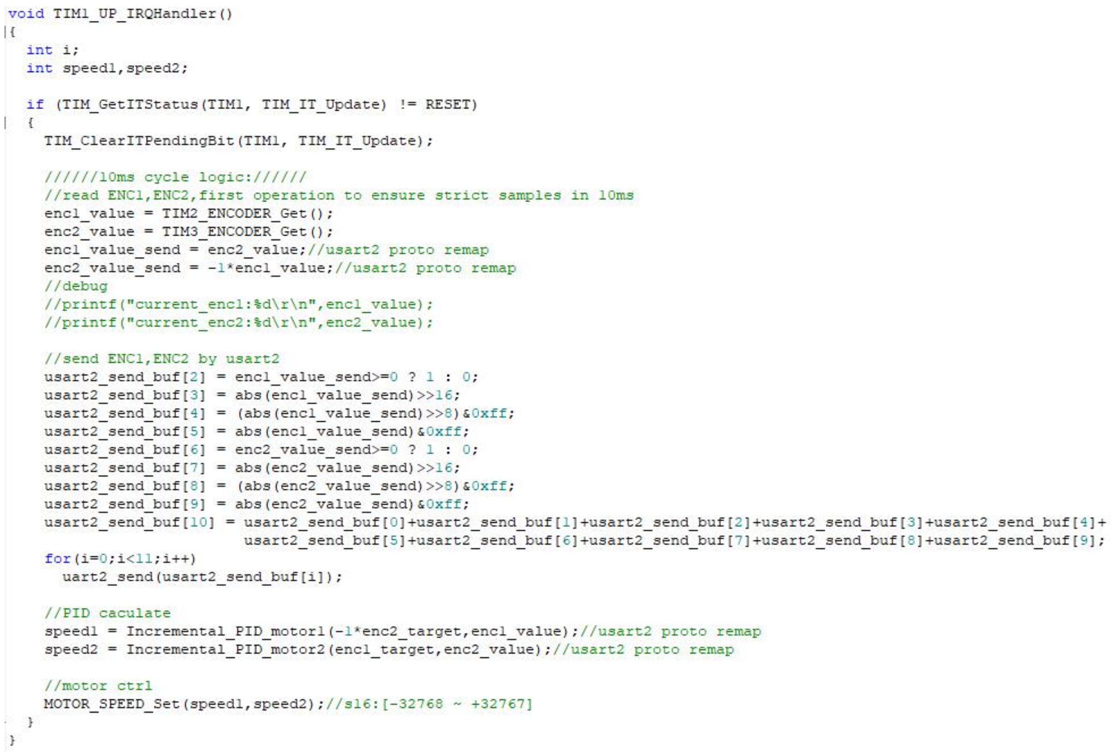
首先，配置定时器 TIM1，实例代码如图 26。



```c
void TIM1_10ms_Init(u16 arr,u16 psc)
{
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  //CLK enable
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);

  //TIM1 base config
  TIM_DeInit(TIM1);
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  TIM_TimeBaseStructure.TIM_Period = arr;
  TIM_TimeBaseStructure.TIM_Prescaler = psc;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;//repetition counter for advanced TIM1
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

  //TIM1 NVIC config
  NVIC_InitStructure.NVIC_IRQChannel = TIM1_UP_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  TIM_ClearFlag(TIM1, TIM_FLAG_Update);
  TIM_ITConfig(TIM1, TIM_IT_Update | TIM_IT_Trigger, ENABLE);

  //reset counter
  TIM1->CNT = 0;

  TIM_Cmd(TIM1, ENABLE);
}
```

（图 26）配置定时器 TIM1

然后，编写中断处理函数，实例代码如图 27。



```c
void TIM1_UP_IRQHandler()
{
  int i;
  int speed1,speed2;

  if (TIM_GetITStatus(TIM1, TIM_IT_Update) != RESET)
  {
    TIM_ClearITPendingBit(TIM1, TIM_IT_Update);

    //////10ms cycle logic://////
    //read ENC1,ENC2,first operation to ensure strict samples in 10ms
    enc1_value = TIM2_ENCODER_Get();
    enc2_value = TIM3_ENCODER_Get();
    enc1_value_send = enc2_value;//usart2 proto remap
    enc2_value_send = -1*enc1_value;//usart2 proto remap
    //debug
    //printf("current_enc1:%d\r\n",enc1_value);
    //printf("current_enc2:%d\r\n",enc2_value);

    //send ENC1,ENC2 by usart2
    usart2_send_buf[2] = enc1_value_send>=0 ? 1 : 0;
    usart2_send_buf[3] = abs(enc1_value_send)>>16;
    usart2_send_buf[4] = (abs(enc1_value_send)>>8)&0xff;
    usart2_send_buf[5] = abs(enc1_value_send)&0xff;
    usart2_send_buf[6] = enc2_value_send>=0 ? 1 : 0;
    usart2_send_buf[7] = abs(enc2_value_send)>>16;
    usart2_send_buf[8] = (abs(enc2_value_send)>>8)&0xff;
    usart2_send_buf[9] = abs(enc2_value_send)&0xff;
    usart2_send_buf[10] = usart2_send_buf[0]+usart2_send_buf[1]+usart2_send_buf[2]+usart2_send_buf[3]+usart2_send_buf[4]+
                          usart2_send_buf[5]+usart2_send_buf[6]+usart2_send_buf[7]+usart2_send_buf[8]+usart2_send_buf[9];
    for(i=0;i<11;i++)
      uart2_send(usart2_send_buf[i]);

    //PID caculate
    speed1 = Incremental_PID_motor1(-1*enc2_target,enc1_value);//usart2 proto remap
    speed2 = Incremental_PID_motor2(enc1_target,enc2_value);//usart2 proto remap

    //motor ctrl
    MOTOR_SPEED_Set(speed1,speed2);//s16:[-32768 ~ +32767]
  }
}
```

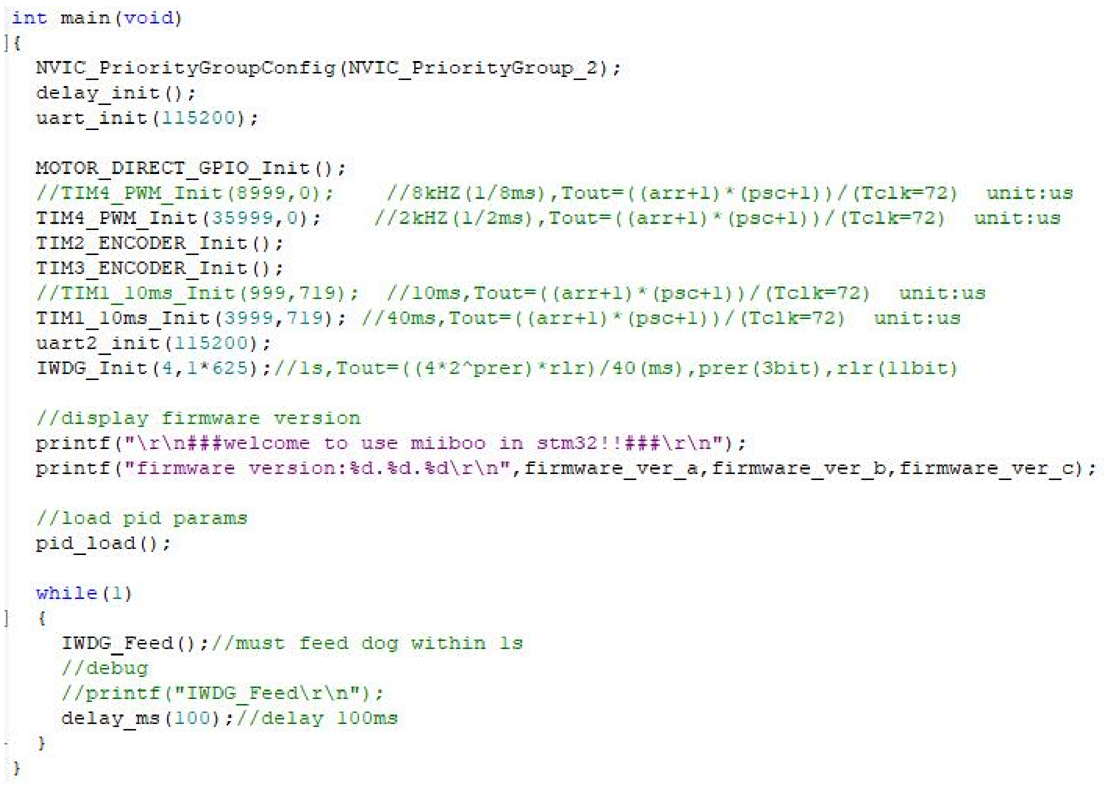
（图 27）TIM1 中断处理函数

2.6.stm32 主控软件整体框图

通过上面的讲解，对底盘控制的 stm32 程序实现有了一定的了解，接下来就来做一个总

结。

先来看看 main()函数实现，如图 28。

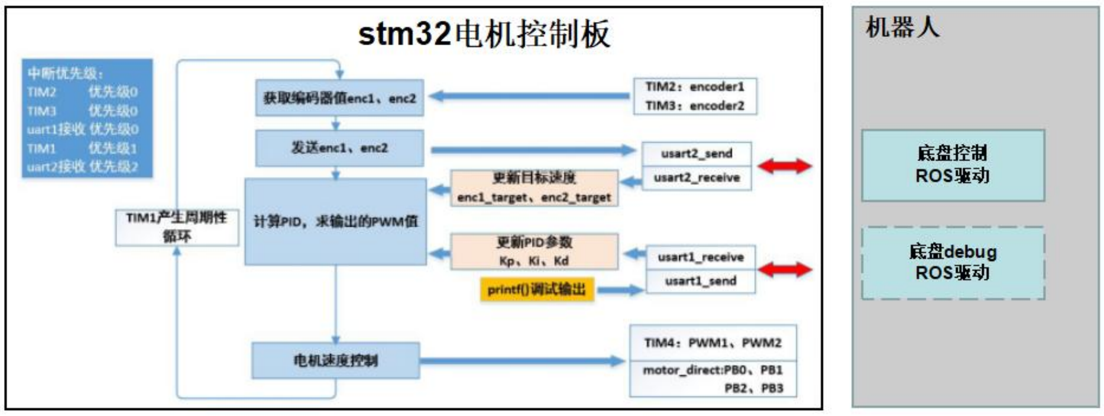


```c
int main(void)
{
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  delay_init();
  uart_init(115200);

  MOTOR_DIRECT_GPIO_Init();
  //TIM4_PWM_Init(8999,0);    //8kHZ(1/8ms),Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM4_PWM_Init(35999,0);    //2kHZ(1/2ms),Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM2_ENCODER_Init();
  TIM3_ENCODER_Init();
  //TIM1_10ms_Init(999,719);  //10ms,Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  TIM1_10ms_Init(3999,719); //40ms,Tout=((arr+1)*(psc+1))/(Tclk=72)  unit:us
  uart2_init(115200);
  IWDG_Init(4,1*625);//1s,Tout=((4*2^prer)*rlr)/40(ms),prer(3bit),rlr(11bit)

  //display firmware version
  printf("\r\n###welcome to use miiboo in stm32!!###\r\n");
  printf("firmware version:%d.%d.%d\r\n",firmware_ver_a,firmware_ver_b,firmware_ver_c);

  //load pid params
  pid_load();

  while(1)
  {
    IWDG_Feed();//must feed dog within 1s
    //debug
    //printf("IWDG_Feed\r\n");
    delay_ms(100);//delay 100ms
  }
}
```

（图 28）main()函数实现

结合上面 TIM1 中断处理函数，不难发现，整个 stm32 程序的执行过程：
a.在 main()函数中初始化各个模块；
b.TIM1 中断处理函数周期性的读取编码器值、反馈获取的编码值、PID 控制；
c.剩下的就是串口 1 和串口 2 的通信交互。

具体 stm32 主控软件整体框图如图 29。



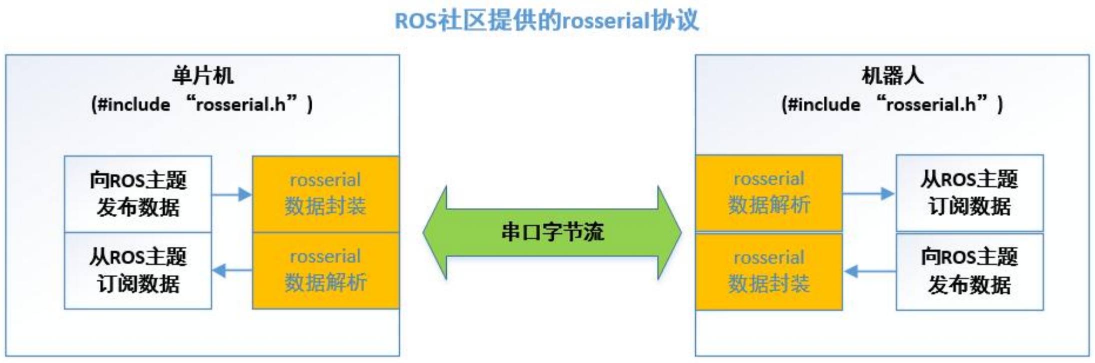
（图 29）stm32 主控软件整体框图

需要说明的是，在周期性循环体中，要首先读取编码器的值，来保证严格的等间隔采样。

3. 底盘通信协议

对于做纯 SLAM 算法、机器人导航避障、或者别的需要用到移动底盘的应用，其实不需要搞明白底盘的底层硬件原理和软件实现等繁琐的细节，只需要根据底盘通信协议，在上层应用程序中利用串口以收发数据的方式来完成对底盘的操作。也就是说底盘的底层操作细节被封装到基于串口通信的 API 中了。

先来说说 ROS 社区提供的 rosserial 库，rosserial 库是为了解决单片机与机器人之间的通信问题，使用 rosserial 库可以实现单片机与机器人之间透明的 ROS 主题发布与订阅通信。原理其实很简单，如图 30。

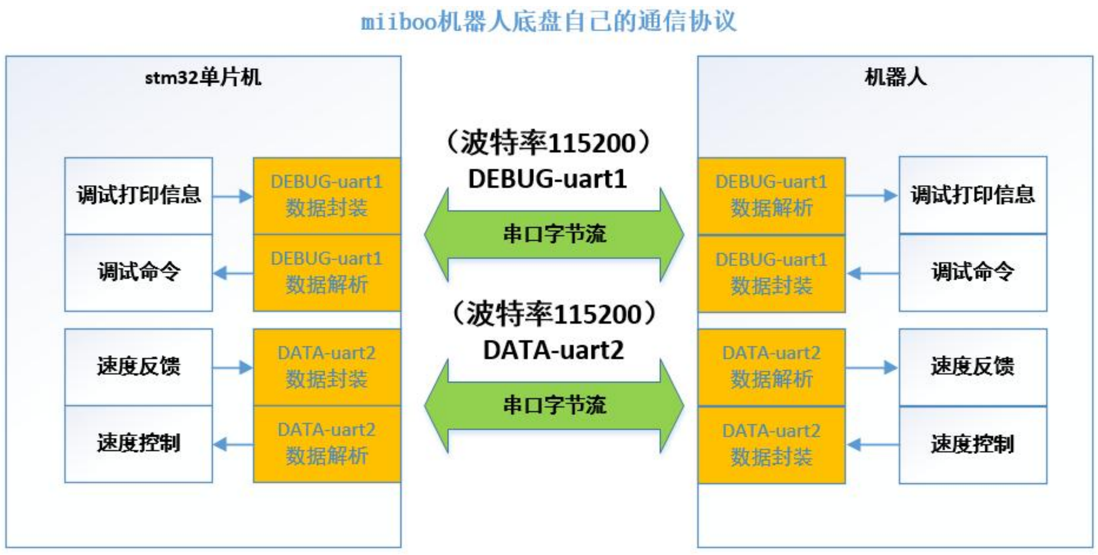


（图 30）ROS 社区提供的 rosserial 协议

单片机中通过包含 rosserial.h 头文件来引用 rosserial 库中的数据封装与数据解析方法，这样在单片机上可以直接按照 ROS 中发布和订阅数据的语法来编写程序，rosserial 库会自动完成封装和解析；被 rosserial 库封装成串口字节流后可以在串口数据线上传输；在机器人上同样通过包含 rosserial.h 头文件来引用 rosserial 库中的数据封装与数据解析方法，这样在机器人上直接按照 ROS 中发布和订阅数据的语法来编写程序，rosserial 库会自动完成封装和解析。rosserial 协议建立了单片机与机器人之间的透明 ROS 通信，这个 ROS 机器人开发这带来了很大的方便。

但是，rosserial 协议虽然好，目前 rosserial 对很多单片机的支持还不是很好，只对少数型号的单片机（比如 Arduino 系列单片机）有支持，像应用广泛的 stm32 单片机就没有官方 rosserial 库的支持；另一个缺点，rosserial 协议比较臃肿，这样对通信的资源消耗大并且影响数据实时性。

其实解决 rosserial 协议这几个缺点很简单，我们借鉴 rosserial 协议的思想，对 rosserial 协议中的冗余进行裁剪，我们 miiboo 机器人底盘自己的通信协议也就应运而生了。miiboo 机器人底盘自己的通信协议，如图 31。

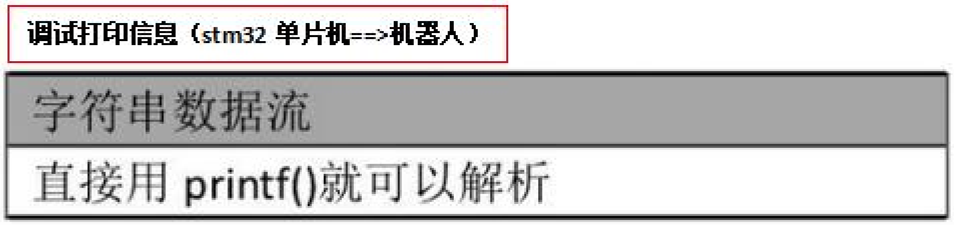


（图 31）miiboo 机器人底盘自己的通信协议

其实很好理解，miiboo 机器人底盘自己的通信协议包含两个部分：DEBUG-uart1 和 DATA-uart2。DEBUG-uart1 用于 stam32 与机器人之间传输调试打印信息、调试命令；DATA-uart2 用于 stam32 与机器人之间传输速度反馈、速度控制。并且 DEBUG-uart1 和 DATA-uart2 两个串口都采用波特率 115200 进行数据传输。下面就针对 DEBUG-uart1 和 DATA-uart2 这两部分的协议进行详细的讲解。

3.1.DEBUG-uart1 协议内容

DEBUG-uart1 协议内容分为：调试打印信息（stm32 单片机==>机器人）、调试命令（stm32 单片机<==机器人）。调试打印信息是 stm32 单片机向机器人发送数据，调试命令是机器人向 stm32 单片机发送数据。

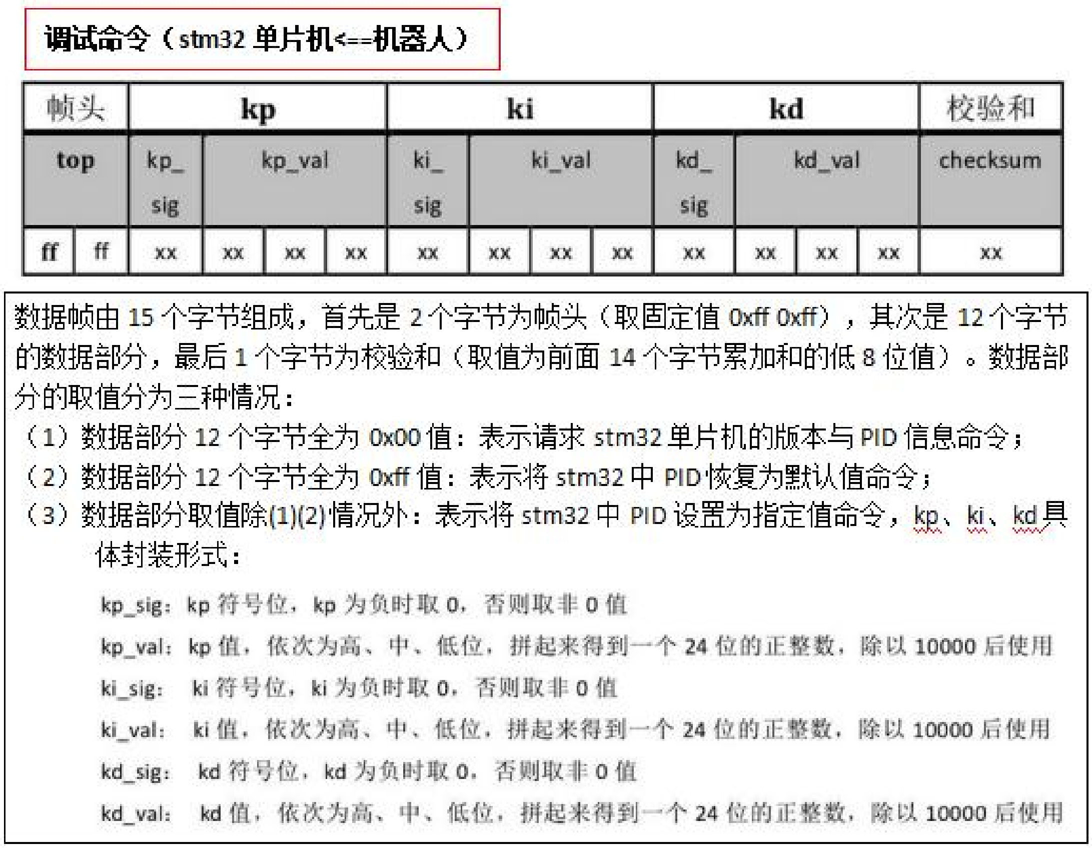

调试打印信息（stm32 单片机==>机器人）

字符串数据流

直接用 printf()就可以解析

（图 32）调试打印信息（stm32 单片机==>机器人）

在机器人端，对从串口获取的字符串数据流，直接用 printf()函数就可以解析。



调试命令（stm32 单片机<==机器人）

帧头		kp				ki				kd				校验和
top		kp_sig	kp_val			ki_sig	ki_val			kd_sig	kd_val			checksum
ff	ff	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

数据帧由 15 个字节组成，首先是 2 个字节为帧头（取固定值 0xff 0xff），其次是 12 个字节的数据部分，最后 1 个字节为校验和（取值为前面 14 个字节累加和的低 8 位值）。数据部分的取值分为三种情况：

（1）数据部分 12 个字节全为 0x00 值：表示请求 stm32 单片机的版本与 PID 信息命令；

（2）数据部分 12 个字节全为 0xff 值：表示将 stm32 中 PID 恢复为默认值命令；

（3）数据部分取值除(1)(2)情况外：表示将 stm32 中 PID 设置为指定值命令，kp、ki、kd 具体封装形式：

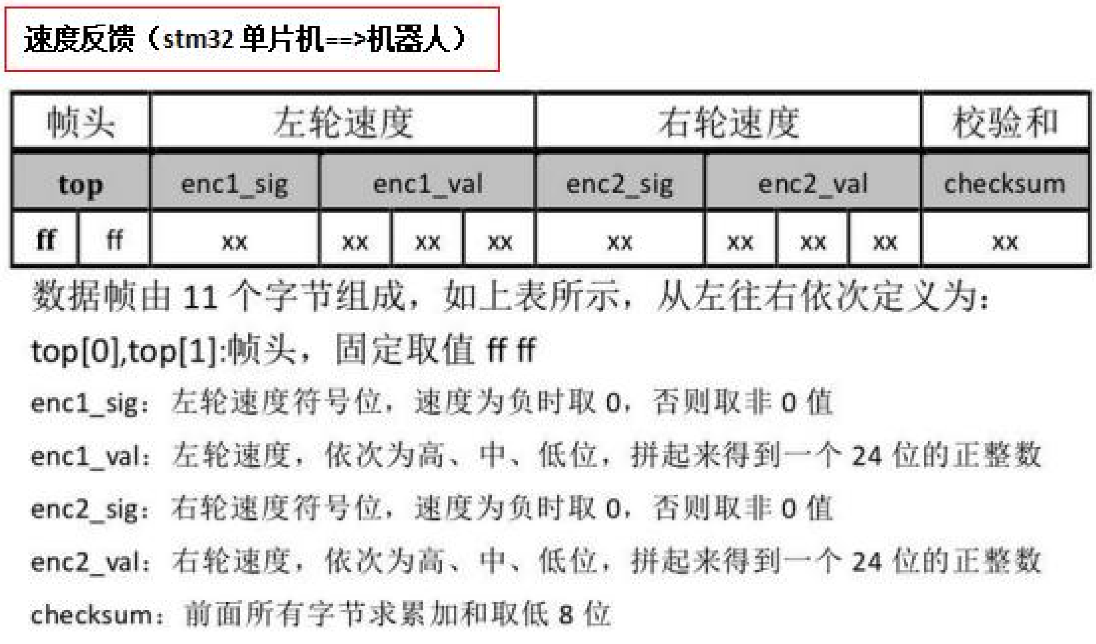
kp_sig：	kp 符号位，kp 为负时取 0，否则取非 0 值
kp_val：	kp 值，依次为高、中、低位，拼起来得到一个 24 位的正整数，除以 10000 后使用
ki_sig：	ki 符号位，ki 为负时取 0，否则取非 0 值
ki_val：	ki 值，依次为高、中、低位，拼起来得到一个 24 位的正整数，除以 10000 后使用
kd_sig：	kd 符号位，kd 为负时取 0，否则取非 0 值
kd_val：	kd 值，依次为高、中、低位，拼起来得到一个 24 位的正整数，除以 10000 后使用

（图 33）调试命令（stm32 单片机<==机器人）

在机器人端，将要下发的调试命令（其实就是请求 stm32 单片机的版本信息与 PID 信息命令、请求将 stm32 中 PID 恢复为默认值命令、请求将 stm32 中 PID 设置为指定值命令）封装成对应的数据帧，然后让串口下发由这 15 个字节组成的数据帧就行了。

3.2.DATA-uart2 协议内容

DATA-uart2 协议内容分为：速度反馈（stm32 单片机==>机器人）、速度控制（stm32 单片机<==机器人）。速度反馈是 stm32 单片机向机器人发送数据，速度控制是机器人向 stm32 单片机发送数据。


速度反馈（stm32 单片机==>机器人）

帧头		左轮速度				右轮速度				校验和
top		enc1_sig	enc1_val			enc2_sig	enc2_val			checksum
ff	ff	xx	xx	xx	xx	xx	xx	xx	xx	xx

数据帧由 11 个字节组成，如上表所示，从左往右依次定义为：

top[0],top[1]:帧头，固定取值 ff ff

enc1_sig：左轮速度符号位，速度为负时取 0，否则取非 0 值

enc1_val：左轮速度，依次为高、中、低位，拼起来得到一个 24 位的正整数

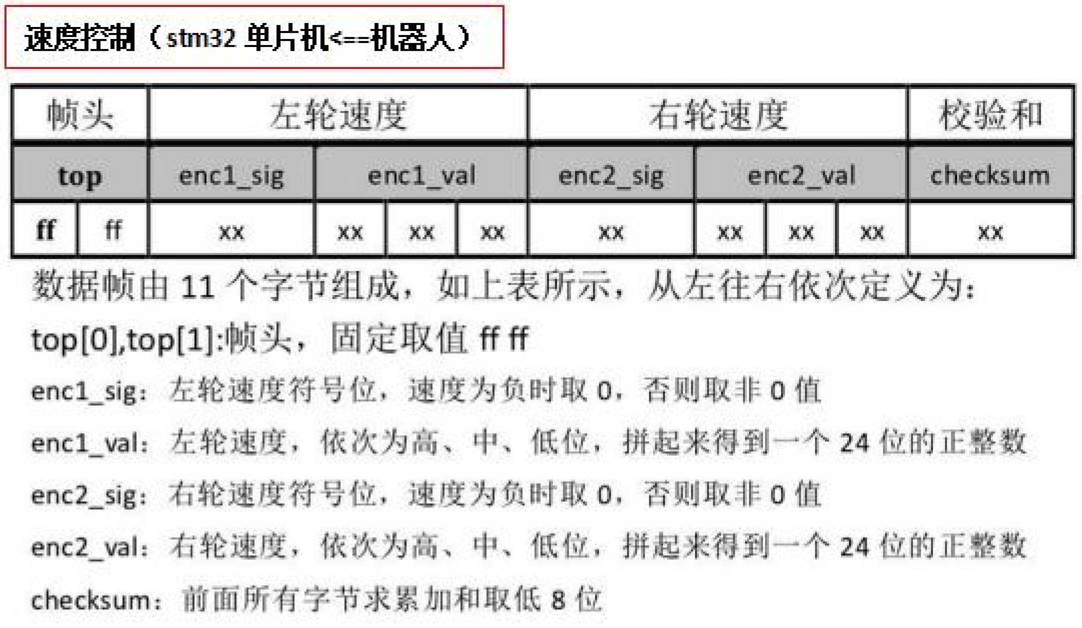
enc2_sig：右轮速度符号位，速度为负时取 0，否则取非 0 值

enc2_val：右轮速度，依次为高、中、低位，拼起来得到一个 24 位的正整数

checksum：前面所有字节求累加和取低 8 位

（图 34）速度反馈（stm32 单片机==>机器人）

在机器人端，对从串口获取的字符串数据流，按照这个数据帧格式进行解析，就可以从这 11 字节组成的数据帧中解析出左轮速度、右轮速度。


速度控制（stm32 单片机<==机器人）

帧头		左轮速度				右轮速度				校验和
top		enc1_sig	enc1_val			enc2_sig	enc2_val			checksum
ff	ff	xx	xx	xx	xx	xx	xx	xx	xx	xx

数据帧由 11 个字节组成，如上表所示，从左往右依次定义为：

top[0],top[1]:帧头，固定取值 ff ff

enc1_sig：左轮速度符号位，速度为负时取 0，否则取非 0 值

enc1_val：左轮速度，依次为高、中、低位，拼起来得到一个 24 位的正整数

enc2_sig：右轮速度符号位，速度为负时取 0，否则取非 0 值

enc2_val：右轮速度，依次为高、中、低位，拼起来得到一个 24 位的正整数

checksum：前面所有字节求累加和取低 8 位

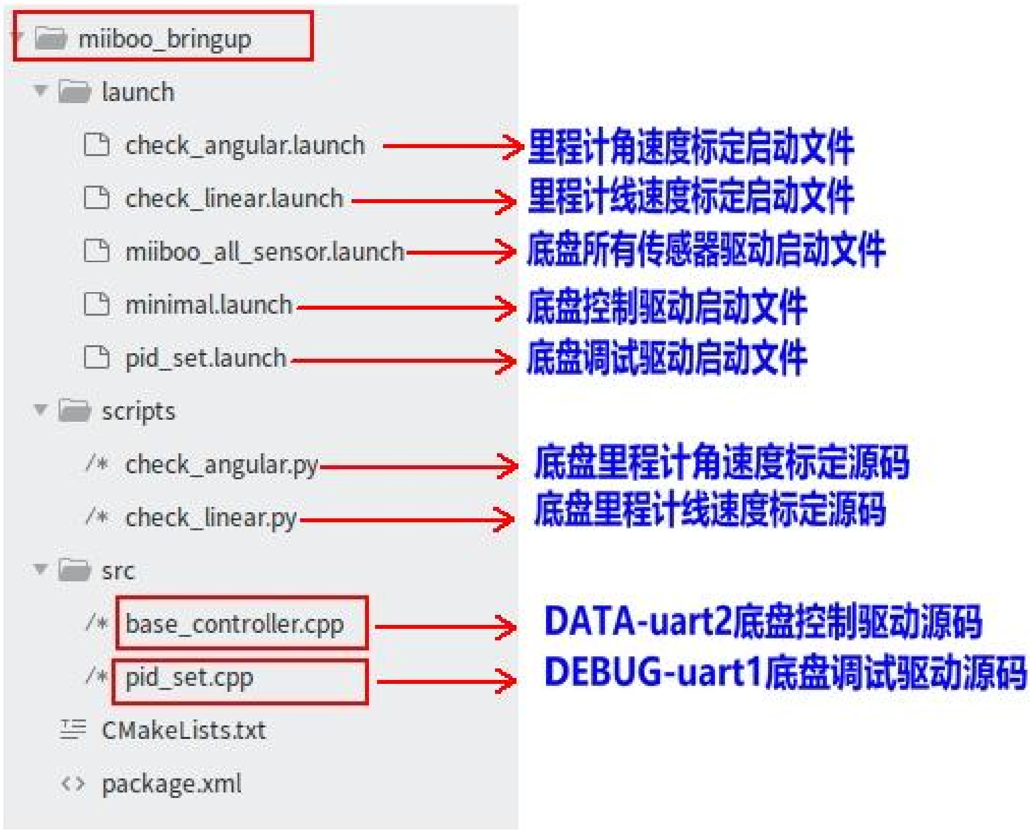
（图 35）速度控制（stm32 单片机<==机器人）

在机器人端，将要下发的目标速度控制值（左轮速度、右轮速度），按照这个数据帧格式进行封装，然后让串口下发由这 11 个字节组成的数据帧就行了。其实不难发现，速度反馈和速度控制遵循同样的数据帧格式，这也是很好理解的。

4. 底盘 ROS 驱动开发

对于做纯 SLAM 算法、机器人导航避障、或者别的需要用到移动底盘的应用，根据底盘的通信协议，直接使用底盘 ROS 驱动实现跟底盘的交互。miiboo 机器人底盘的 ROS 驱动代

码组织如图 36。



（图 36）miiboo 机器人底盘的 ROS 驱动代码组织

整个代码组织是一个完整的 ROS 功能包，功能包名为 miiboo_bringup，功能包中包含两个 ROS 节点源码（base_controller.cpp 和 pid_set.cpp），不难看出这两个节点正是对底盘通信协议中的 DATA-uart2 与 DEBUG-uart1 的具体实现。base_controller.cpp 负责对底盘控制驱动的具体实现，pid_set.cpp 负责对底盘调试驱动的具体实现。本节重点对这两个节点进行讲解，至于功能包名下的其他内容将放在后面的 miiboo 机器人 SLAM 导航实战中具体展开。

4.1.底盘控制节点

接口：

底盘控制节点对下与底盘 DATA-uart2 串口通信，对上开放 ROS 接口为应用层提功能数据发布与订阅，便于 SLAM 导航等功能的开发。

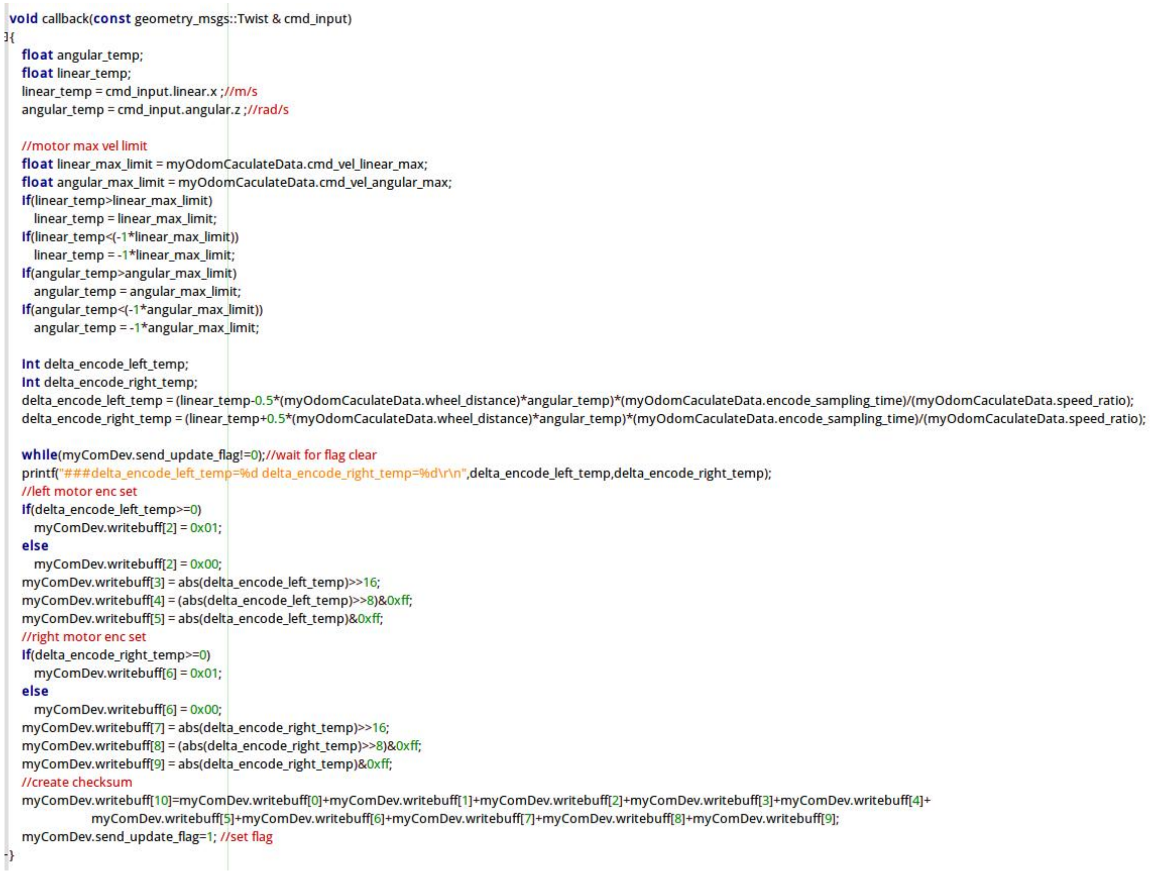
订阅 topic:	/cmd_vel	(geometry_msgs::Twist)
发布 topic:	/wheel_left_speed /wheel_right_speed /odom /tf	(msgs::Float32) (msgs::Float32) (nav_msgs::Odometry) (odom->base_footprint)

（图 37）底盘控制节点接口

节点实现源码解析：

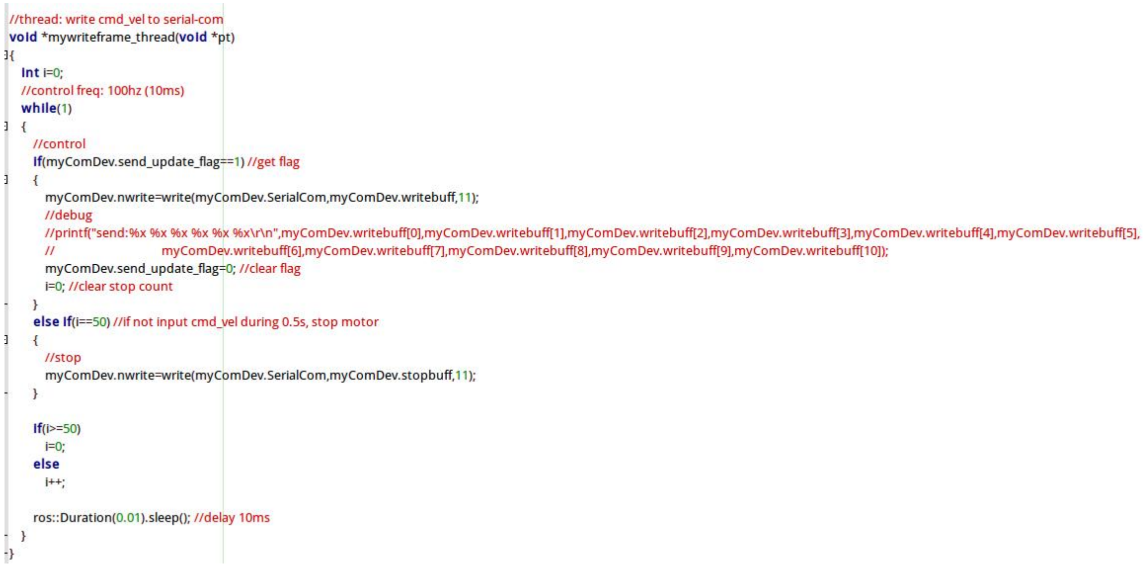
底盘控制节点由 base_controller.cpp 实现。程序主要分为两个过程：订阅 topic 数据并下发给底盘、从底盘接收数据并发布到 topic。

首先，程序订阅/cmd_vel 作为用户的控制输入，将控制输入的速度信息转换为通信协议中 DATA-uart2 规定的格式，然后通过串口下发给底盘,实现对底盘的运动控制。订阅/cmd_vel 的回调函数和串口下发函数分别如图 38 和图 39 所示。



```
void callback(const geometry_msgs::Twist & cmd_input)
{
  float angular_temp;
  float linear_temp;
  linear_temp = cmd_input.linear.x ;//m/s
  angular_temp = cmd_input.angular.z ;//rad/s

  //motor max vel limit
  float linear_max_limit = myOdomCaculateData.cmd_vel_linear_max;
  float angular_max_limit = myOdomCaculateData.cmd_vel_angular_max;
  if(linear_temp>linear_max_limit)
    linear_temp = linear_max_limit;
  if(linear_temp<(-1*linear_max_limit))
    linear_temp = -1*linear_max_limit;
  if(angular_temp>angular_max_limit)
    angular_temp = angular_max_limit;
  if(angular_temp<(-1*angular_max_limit))
    angular_temp = -1*angular_max_limit;

  int delta_encode_left_temp;
  int delta_encode_right_temp;
  delta_encode_left_temp = (linear_temp-0.5*(myOdomCaculateData.wheel_distance)*angular_temp)*(myOdomCaculateData.encode_sampling_time)/(myOdomCaculateData.speed_ratio);
  delta_encode_right_temp = (linear_temp+0.5*(myOdomCaculateData.wheel_distance)*angular_temp)*(myOdomCaculateData.encode_sampling_time)/(myOdomCaculateData.speed_ratio);

  while(myComDev.send_update_flag!=0);//wait for flag clear
  printf("###delta_encode_left_temp=%d delta_encode_right_temp=%d\r\n",delta_encode_left_temp,delta_encode_right_temp);
  //left motor enc set
  if(delta_encode_left_temp>=0)
    myComDev.writebuff[2] = 0x01;
  else
    myComDev.writebuff[2] = 0x00;
  myComDev.writebuff[3] = abs(delta_encode_left_temp)>>16;
  myComDev.writebuff[4] = (abs(delta_encode_left_temp)>>8)&0xff;
  myComDev.writebuff[5] = abs(delta_encode_left_temp)&0xff;
  //right motor enc set
  if(delta_encode_right_temp>=0)
    myComDev.writebuff[6] = 0x01;
  else
    myComDev.writebuff[6] = 0x00;
  myComDev.writebuff[7] = abs(delta_encode_right_temp)>>16;
  myComDev.writebuff[8] = (abs(delta_encode_right_temp)>>8)&0xff;
  myComDev.writebuff[9] = abs(delta_encode_right_temp)&0xff;
  //create checksum
  myComDev.writebuff[10]=myComDev.writebuff[0]+myComDev.writebuff[1]+myComDev.writebuff[2]+myComDev.writebuff[3]+myComDev.writebuff[4]+
              myComDev.writebuff[5]+myComDev.writebuff[6]+myComDev.writebuff[7]+myComDev.writebuff[8]+myComDev.writebuff[9];
  myComDev.send_update_flag=1; //set flag
}
```

（图 38）订阅/cmd_vel 的回调函数

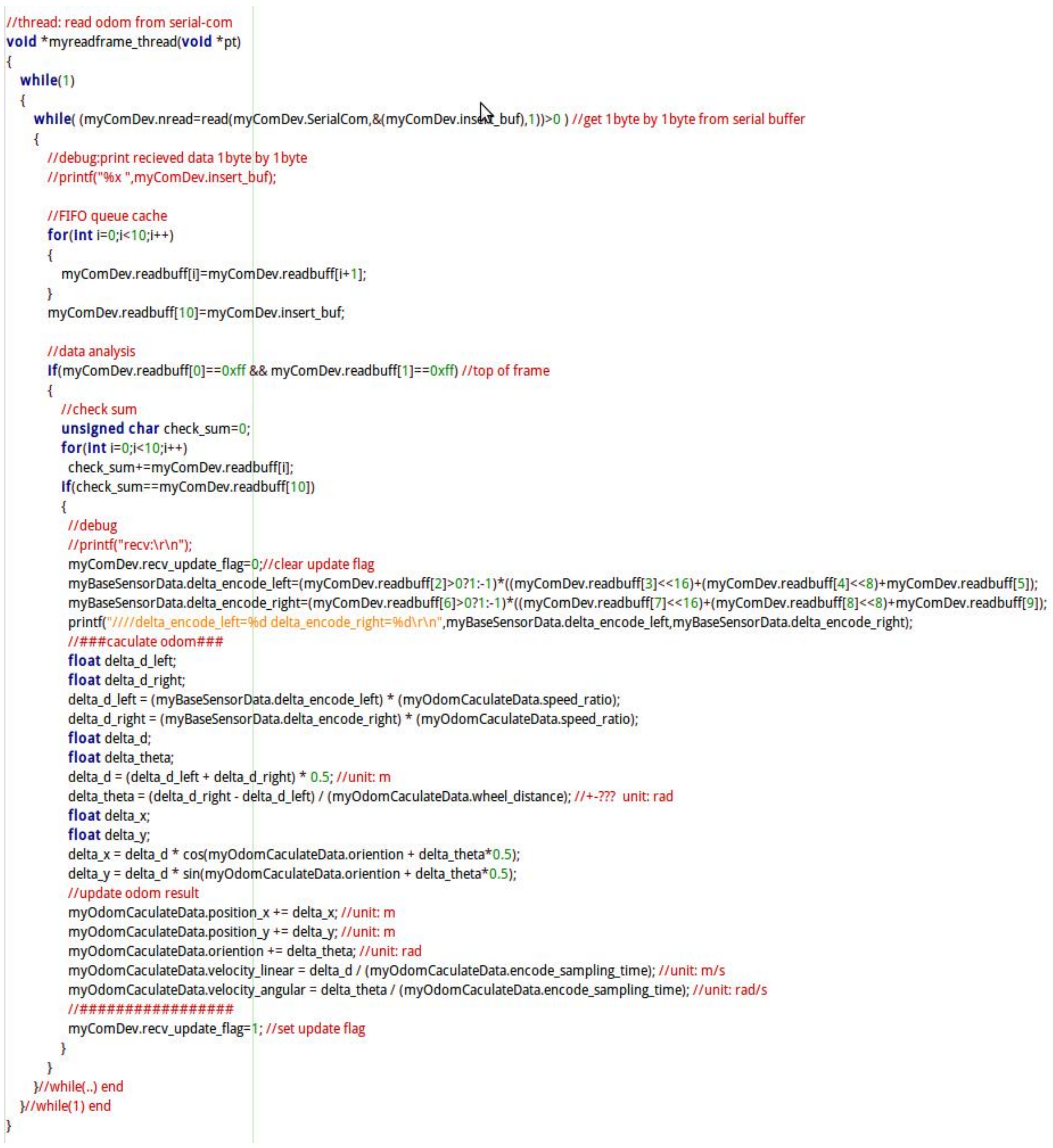


```
//thread: write cmd_vel to serial-com
void *mywriteframe_thread(void *pt)
{
  int i=0;
  //control freq: 100hz (10ms)
  while(1)
  {
    //control
    if(myComDev.send_update_flag==1) //get flag
    {
      myComDev.nwrite=write(myComDev.SerialCom,myComDev.writebuff,11);
      //debug
      //printf("send:%x %x %x %x %x %x\r\n",myComDev.writebuff[0],myComDev.writebuff[1],myComDev.writebuff[2],myComDev.writebuff[3],myComDev.writebuff[4],myComDev.writebuff[5],
      //                myComDev.writebuff[6],myComDev.writebuff[7],myComDev.writebuff[8],myComDev.writebuff[9],myComDev.writebuff[10]);
      myComDev.send_update_flag=0; //clear flag
      i=0; //clear stop count
    }
    else if(i==50) //if not input cmd_vel during 0.5s, stop motor
    {
      //stop
      myComDev.nwrite=write(myComDev.SerialCom,myComDev.stopbuff,11);
    }

    if(i>=50)
      i=0;
    else
      i++;

    ros::Duration(0.01).sleep(); //delay 10ms
  }
}
```

（图 39）串口下发函数

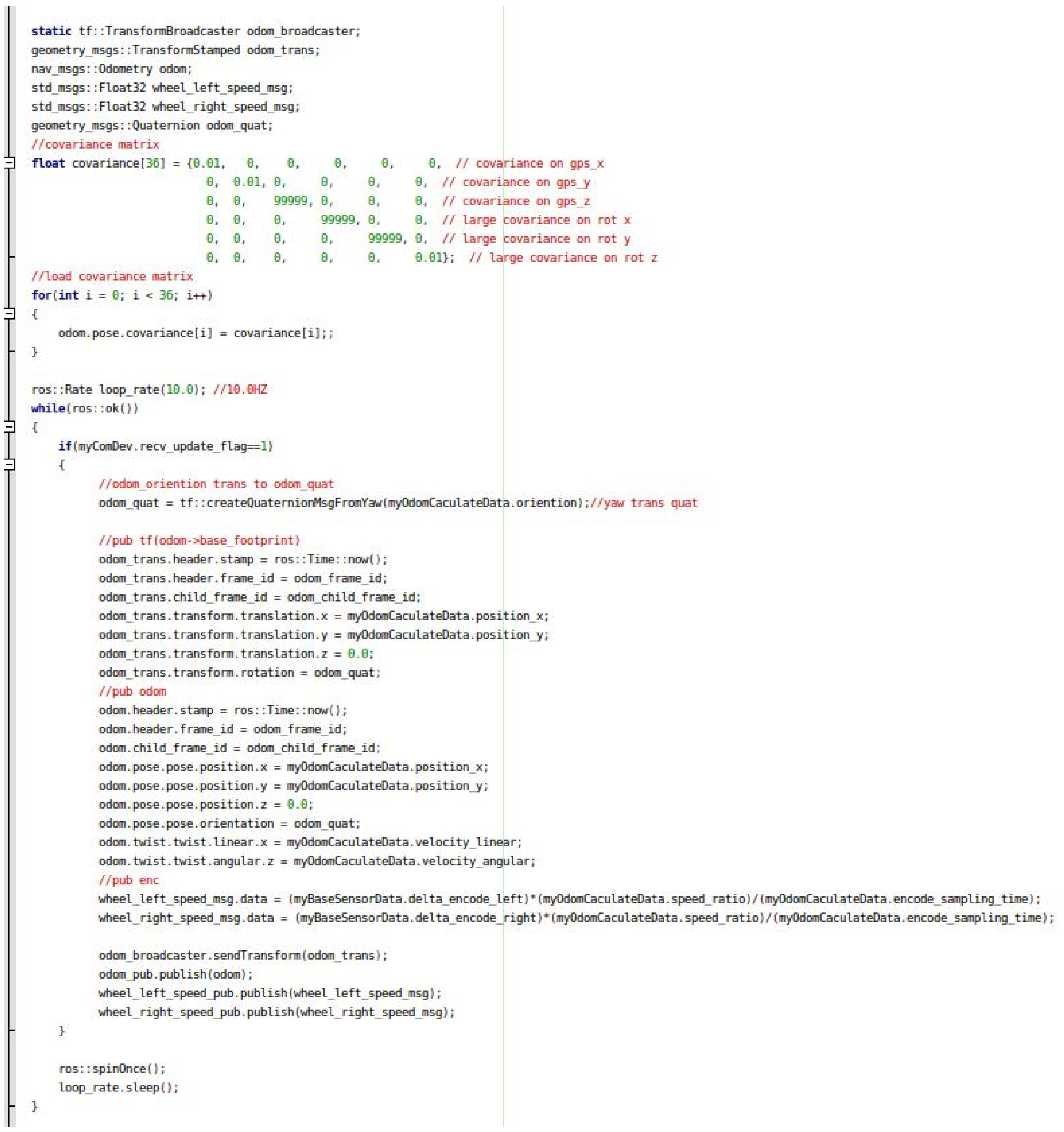
然后，程序从串口获取底盘的速度反馈，并将速度反馈数据放入航迹推演算法中进行解算，得到里程计，将反馈回来的左轮速度、右轮速度值分别发布到/wheel_left_speed 和

/wheel_right_speed 主题，将解算出来的里程计分别发布到/odom 和/tf 主题。由于不同的算法对里程计的格式要求不一样，所以将里程计同时发布到/odom 和/tf 主题，便于不同的算法使用。从串口获取速度反馈并求解里程计和发布反馈速度与里程计到 topic 分别如图 40 和图 41 所示。



```
//thread: read odom from serial-com
void *myreadframe_thread(void *pt)
{
  while(1)
  {
    while( (myComDev.nread=read(myComDev.SerialCom,&(myComDev.insert_buf),1))>0 ) //get 1byte by 1byte from serial buffer
    {
      //debug:print recieved data 1byte by 1byte
      //printf("%x ",myComDev.insert_buf);

      //FIFO queue cache
      for(int i=0;i<10;i++)
      {
        myComDev.readbuff[i]=myComDev.readbuff[i+1];
      }
      myComDev.readbuff[10]=myComDev.insert_buf;

      //data analysis
      if(myComDev.readbuff[0]==0xff && myComDev.readbuff[1]==0xff) //top of frame
      {
        //check sum
        unsigned char check_sum=0;
        for(int i=0;i<10;i++)
          check_sum+=myComDev.readbuff[i];
        if(check_sum==myComDev.readbuff[10])
        {
         //debug
         //printf("recv:\r\n");
         myComDev.recv_update_flag=0;//clear update flag
         myBaseSensorData.delta_encode_left=(myComDev.readbuff[2]>0?1:-1)*((myComDev.readbuff[3]<<16)+(myComDev.readbuff[4]<<8)+myComDev.readbuff[5]);
         myBaseSensorData.delta_encode_right=(myComDev.readbuff[6]>0?1:-1)*((myComDev.readbuff[7]<<16)+(myComDev.readbuff[8]<<8)+myComDev.readbuff[9]);
         printf("////delta_encode_left=%d delta_encode_right=%d\r\n",myBaseSensorData.delta_encode_left,myBaseSensorData.delta_encode_right);
         //###caculate odom###
         float delta_d_left;
         float delta_d_right;
         delta_d_left = (myBaseSensorData.delta_encode_left) * (myOdomCaculateData.speed_ratio);
         delta_d_right = (myBaseSensorData.delta_encode_right) * (myOdomCaculateData.speed_ratio);
         float delta_d;
         float delta_theta;
         delta_d = (delta_d_left + delta_d_right) * 0.5; //unit: m
         delta_theta = (delta_d_right - delta_d_left) / (myOdomCaculateData.wheel_distance); //+-??? unit: rad
         float delta_x;
         float delta_y;
         delta_x = delta_d * cos(myOdomCaculateData.oriention + delta_theta*0.5);
         delta_y = delta_d * sin(myOdomCaculateData.oriention + delta_theta*0.5);
         //update odom result
         myOdomCaculateData.position_x += delta_x; //unit: m
         myOdomCaculateData.position_y += delta_y; //unit: m
         myOdomCaculateData.oriention += delta_theta; //unit: rad
         myOdomCaculateData.velocity_linear = delta_d / (myOdomCaculateData.encode_sampling_time); //unit: m/s
         myOdomCaculateData.velocity_angular = delta_theta / (myOdomCaculateData.encode_sampling_time); //unit: rad/s
         //##################
         myComDev.recv_update_flag=1; //set update flag
        }
      }
    }//while(..) end
  }//while(1) end
}
```

（图 40）从串口获取速度反馈并求解里程计

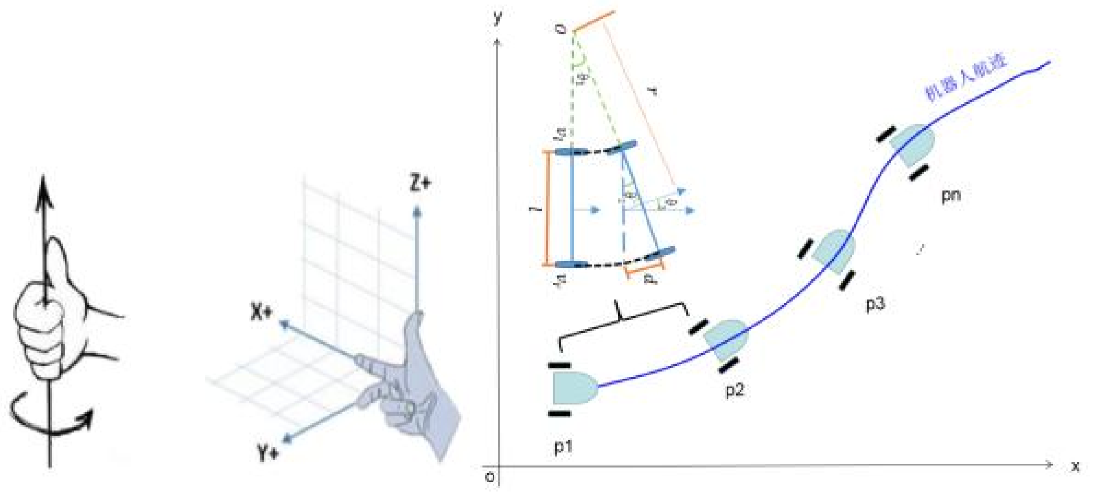


```
static tf::TransformBroadcaster odom_broadcaster;
geometry_msgs::TransformStamped odom_trans;
nav_msgs::Odometry odom;
std_msgs::Float32 wheel_left_speed_msg;
std_msgs::Float32 wheel_right_speed_msg;
geometry_msgs::Quaternion odom_quat;
//covariance matrix
float covariance[36] = {0.01,   0,    0,     0,     0,     0,  // covariance on gps_x
                          0,  0.01, 0,     0,     0,     0,  // covariance on gps_y
                          0,  0,    99999, 0,     0,     0,  // covariance on gps_z
                          0,  0,    0,     99999, 0,     0,  // large covariance on rot x
                          0,  0,    0,     0,     99999, 0,  // large covariance on rot y
                          0,  0,    0,     0,     0,     0.01};  // large covariance on rot z
//load covariance matrix
for(int i = 0; i < 36; i++)
{
    odom.pose.covariance[i] = covariance[i];;
}

ros::Rate loop_rate(10.0); //10.0HZ
while(ros::ok())
{
    if(myComDev.recv_update_flag==1)
    {
        //odom_oriention trans to odom_quat
        odom_quat = tf::createQuaternionMsgFromYaw(myOdomCaculateData.oriention);//yaw trans quat

        //pub tf(odom->base_footprint)
        odom_trans.header.stamp = ros::Time::now();
        odom_trans.header.frame_id = odom_frame_id;
        odom_trans.child_frame_id = odom_child_frame_id;
        odom_trans.transform.translation.x = myOdomCaculateData.position_x;
        odom_trans.transform.translation.y = myOdomCaculateData.position_y;
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation = odom_quat;
        //pub odom
        odom.header.stamp = ros::Time::now();
        odom.header.frame_id = odom_frame_id;
        odom.child_frame_id = odom_child_frame_id;
        odom.pose.pose.position.x = myOdomCaculateData.position_x;
        odom.pose.pose.position.y = myOdomCaculateData.position_y;
        odom.pose.pose.position.z = 0.0;
        odom.pose.pose.orientation = odom_quat;
        odom.twist.twist.linear.x = myOdomCaculateData.velocity_linear;
        odom.twist.twist.angular.z = myOdomCaculateData.velocity_angular;
        //pub enc
        wheel_left_speed_msg.data = (myBaseSensorData.delta_encode_left)*(myOdomCaculateData.speed_ratio)/(myOdomCaculateData.encode_sampling_time);
        wheel_right_speed_msg.data = (myBaseSensorData.delta_encode_right)*(myOdomCaculateData.speed_ratio)/(myOdomCaculateData.encode_sampling_time);

        odom_broadcaster.sendTransform(odom_trans);
        odom_pub.publish(odom);
        wheel_left_speed_pub.publish(wheel_left_speed_msg);
        wheel_right_speed_pub.publish(wheel_right_speed_msg);
    }

    ros::spinOnce();
    loop_rate.sleep();
}
```

（图 41）发布反馈速度与里程计到 topic

基于航迹推演算法的里程计解算：

首先，我们需要了解一下 ROS 下的机器人坐标系为右手坐标系，如图 42，机器人底盘的正前方为 x 轴正方向、机器人底盘的正上方为 z 轴正方向、机器人底盘的正左方向为 y 轴正方向、机器人航向角 theta 坐标轴以 x 轴为 0 度角并逆时针方向增大。一般以机器人底盘上电时刻，机器人底盘的位置建立里程计坐标系，也就是说机器人底盘的起始位姿为原点 O，机器人底盘在运动过程中，通过前一时刻的位置和左、右轮位移可以推算出机器人底盘的下一时刻位姿，这就是航迹推演算法。



（图 42）机器人右手坐标系与航迹推演

我们这里值讨论两轮差分底盘的情况，分析如图 41，通过前一时刻的位置和左、右轮位移可以推算出机器人底盘的下一时刻位姿。航迹推演的数学模型如图 43。

其中：

前一时刻位姿 $\mathcal{P}_i = (x, y, \theta)$

左轮位移 dleft

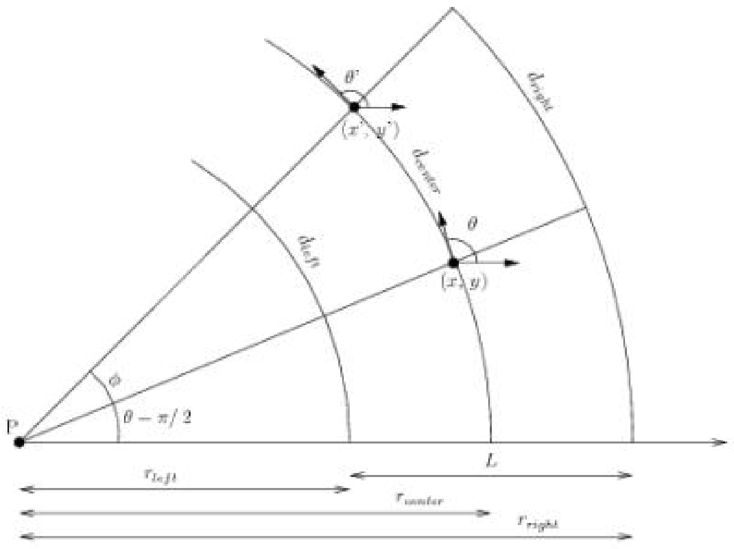
右轮位移 dright

中心位移 dcenter

航向角变换量 Φ

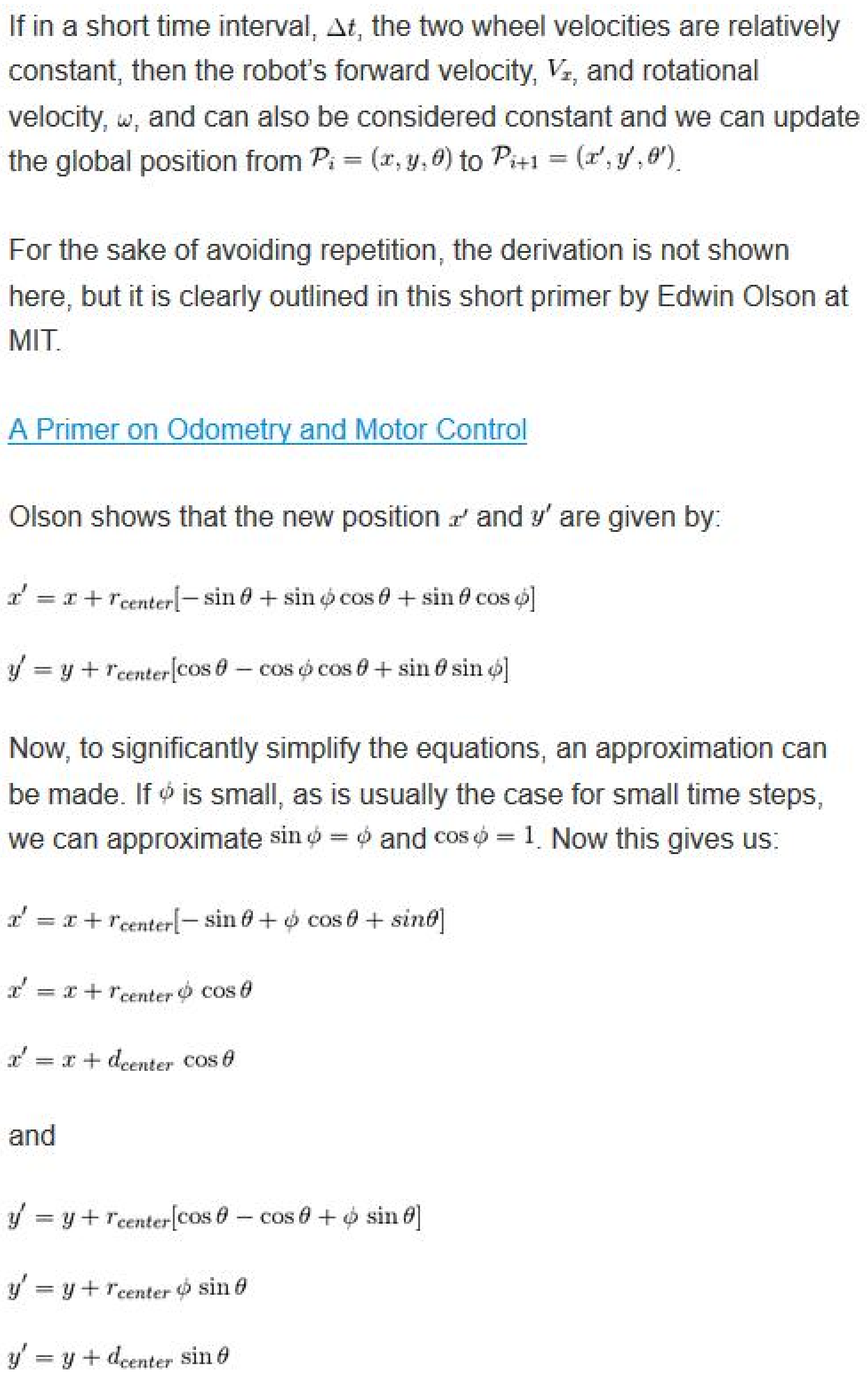
两轮间距 L

下一时刻位姿 $\mathcal{P}_{i+1} = (x', y', \theta')$



（图 43）航迹推演的数学模型

在很短的时间间隔里，前后两个机器人位姿满足一定的关系，具体看图 44 的推导。



If in a short time interval, Δt, the two wheel velocities are relatively constant, then the robot's forward velocity, V_x, and rotational velocity, ω, and can also be considered constant and we can update the global position from $\mathcal{P}_i = (x, y, \theta)$ to $\mathcal{P}_{i+1} = (x', y', \theta')$.

For the sake of avoiding repetition, the derivation is not shown here, but it is clearly outlined in this short primer by Edwin Olson at MIT.

A Primer on Odometry and Motor Control

Olson shows that the new position x' and y' are given by:

$$x' = x + r_{center}[-\sin\theta + \sin\phi\cos\theta + \sin\theta\cos\phi]$$

$$y' = y + r_{center}[\cos\theta - \cos\phi\cos\theta + \sin\theta\sin\phi]$$

Now, to significantly simplify the equations, an approximation can be made. If ϕ is small, as is usually the case for small time steps, we can approximate $\sin\phi = \phi$ and $\cos\phi = 1$. Now this gives us:

$$x' = x + r_{center}[-\sin\theta + \phi\,\cos\theta + sin\theta]$$

$$x' = x + r_{center}\,\phi\,\cos\theta$$

$$x' = x + d_{center}\,\cos\theta$$

and

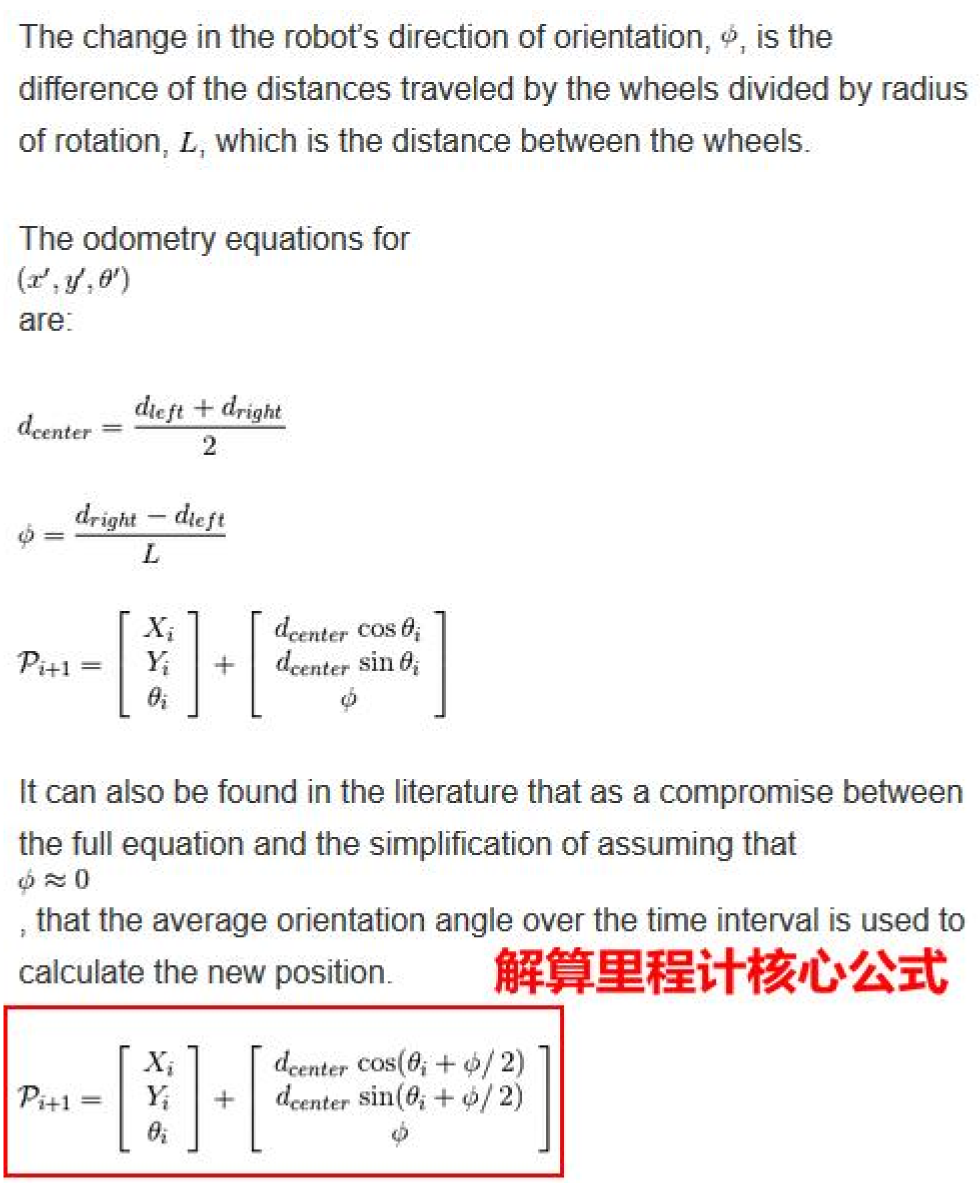
$$y' = y + r_{center}[\cos\theta - \cos\theta + \phi\,\sin\theta]$$

$$y' = y + r_{center}\,\phi\,\sin\theta$$

$$y' = y + d_{center}\,\sin\theta$$

（图 44）短时间内两个机器人位姿的约束关系

这样，经过进一步的化简，可以得到我们解算里程计的核心公式，如图 45。



The change in the robot's direction of orientation, ϕ, is the difference of the distances traveled by the wheels divided by radius of rotation, L, which is the distance between the wheels.

The odometry equations for
(x', y', θ')
are:

$$d_{center} = \frac{d_{left} + d_{right}}{2}$$

$$\phi = \frac{d_{right} - d_{left}}{L}$$

$$\mathcal{P}_{i+1} = \begin{bmatrix} X_i \\ Y_i \\ \theta_i \end{bmatrix} + \begin{bmatrix} d_{center} \cos\theta_i \\ d_{center} \sin\theta_i \\ \phi \end{bmatrix}$$

It can also be found in the literature that as a compromise between the full equation and the simplification of assuming that
$\phi \approx 0$
, that the average orientation angle over the time interval is used to calculate the new position.

解算里程计核心公式

$$\boxed{\mathcal{P}_{i+1} = \begin{bmatrix} X_i \\ Y_i \\ \theta_i \end{bmatrix} + \begin{bmatrix} d_{center} \cos(\theta_i + \phi/2) \\ d_{center} \sin(\theta_i + \phi/2) \\ \phi \end{bmatrix}}$$

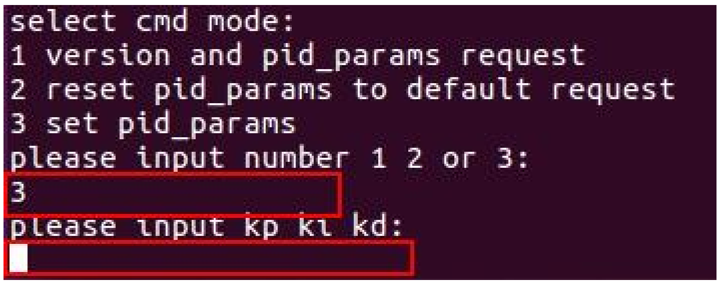
（图 45）解算里程计核心公式

不难发现，上面程序中解算里程计部分的代码就是这个核心公式的具体编程实现。关于航迹推演算法更详细的推导，请参考：
http://faculty.salina.k-state.edu/tim/robotics_sg/Control/kinematics/odometry.html

4.2.底盘调试节点

接口：

底盘调试节点对下与底盘 DEBUG-uart1 串口通信，对上通过命令行终端指令交互方式。



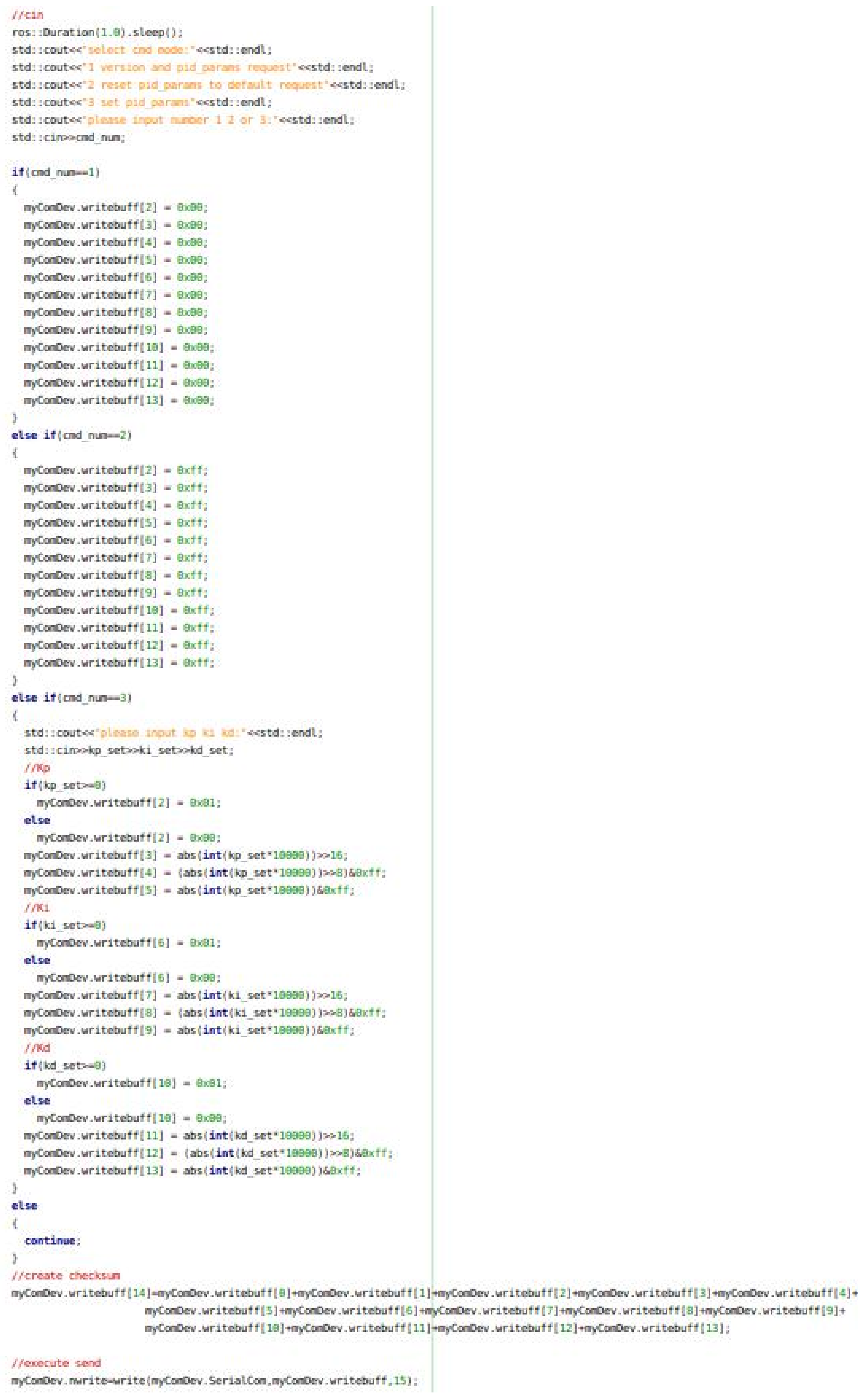
```
select cmd mode:
1 version and pid_params request
2 reset pid_params to default request
3 set pid_params
please input number 1 2 or 3:
3
please input kp ki kd:
```

（图 46）底盘调试节点接口

节点实现源码解析：

底盘调试节点由 pid_set.cpp 实现。程序主要分为两个过程：从终端获取调试命令并下发给底盘、从底盘接收应答数据并显示在终端。

首先，程序从终端获取调试命令，用户可输入数字 1,2,3，如果用户输入 3 会再要求输入 kp,ki,kd 这三个数，将调试命令转换为通信协议中 DEBUG-uart1 规定的格式，然后通过串口下发给底盘,实现对底盘的调试。调试命令获取与下发如图 47。

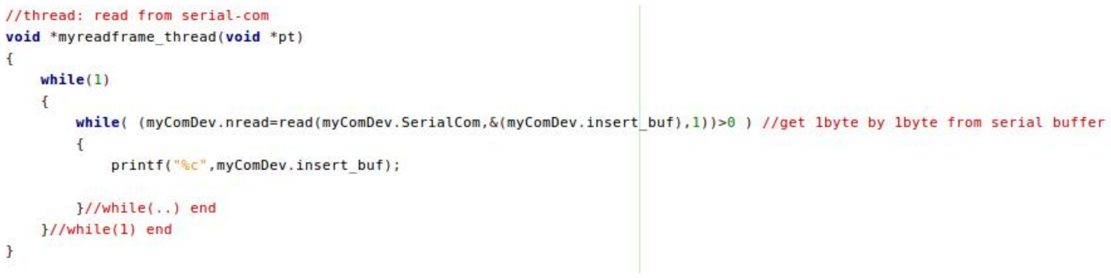


```
//cin
ros::Duration(1.0).sleep();
std::cout<<"select cmd mode:"<<std::endl;
std::cout<<"1 version and pid_params request"<<std::endl;
std::cout<<"2 reset pid_params to default request"<<std::endl;
std::cout<<"3 set pid_params"<<std::endl;
std::cout<<"please input number 1 2 or 3:"<<std::endl;
std::cin>>cmd_num;

if(cmd_num==1)
{
  myComDev.writebuff[2] = 0x00;
  myComDev.writebuff[3] = 0x00;
  myComDev.writebuff[4] = 0x00;
  myComDev.writebuff[5] = 0x00;
  myComDev.writebuff[6] = 0x00;
  myComDev.writebuff[7] = 0x00;
  myComDev.writebuff[8] = 0x00;
  myComDev.writebuff[9] = 0x00;
  myComDev.writebuff[10] = 0x00;
  myComDev.writebuff[11] = 0x00;
  myComDev.writebuff[12] = 0x00;
  myComDev.writebuff[13] = 0x00;
}
else if(cmd_num==2)
{
  myComDev.writebuff[2] = 0xff;
  myComDev.writebuff[3] = 0xff;
  myComDev.writebuff[4] = 0xff;
  myComDev.writebuff[5] = 0xff;
  myComDev.writebuff[6] = 0xff;
  myComDev.writebuff[7] = 0xff;
  myComDev.writebuff[8] = 0xff;
  myComDev.writebuff[9] = 0xff;
  myComDev.writebuff[10] = 0xff;
  myComDev.writebuff[11] = 0xff;
  myComDev.writebuff[12] = 0xff;
  myComDev.writebuff[13] = 0xff;
}
else if(cmd_num==3)
{
  std::cout<<"please input kp ki kd:"<<std::endl;
  std::cin>>kp_set>>ki_set>>kd_set;
  //Kp
  if(kp_set>=0)
    myComDev.writebuff[2] = 0x01;
  else
    myComDev.writebuff[2] = 0x00;
  myComDev.writebuff[3] = abs(int(kp_set*10000))>>16;
  myComDev.writebuff[4] = (abs(int(kp_set*10000))>>8)&0xff;
  myComDev.writebuff[5] = abs(int(kp_set*10000))&0xff;
  //Ki
  if(ki_set>=0)
    myComDev.writebuff[6] = 0x01;
  else
    myComDev.writebuff[6] = 0x00;
  myComDev.writebuff[7] = abs(int(ki_set*10000))>>16;
  myComDev.writebuff[8] = (abs(int(ki_set*10000))>>8)&0xff;
  myComDev.writebuff[9] = abs(int(ki_set*10000))&0xff;
  //Kd
  if(kd_set>=0)
    myComDev.writebuff[10] = 0x01;
  else
    myComDev.writebuff[10] = 0x00;
  myComDev.writebuff[11] = abs(int(kd_set*10000))>>16;
  myComDev.writebuff[12] = (abs(int(kd_set*10000))>>8)&0xff;
  myComDev.writebuff[13] = abs(int(kd_set*10000))&0xff;
}
else
{
  continue;
}
//create checksum
myComDev.writebuff[14]=myComDev.writebuff[0]+myComDev.writebuff[1]+myComDev.writebuff[2]+myComDev.writebuff[3]+myComDev.writebuff[4]+
                       myComDev.writebuff[5]+myComDev.writebuff[6]+myComDev.writebuff[7]+myComDev.writebuff[8]+myComDev.writebuff[9]+
                       myComDev.writebuff[10]+myComDev.writebuff[11]+myComDev.writebuff[12]+myComDev.writebuff[13];

//execute send
myComDev.nwrite=write(myComDev.SerialCom,myComDev.writebuff,15);
```

（图 47）调试命令获取与下发

然后，程序从串口获取底盘的应答信息，这里就比较简单了，直接将获取的应答数据原

样打印到终端就行了，如图 48。



```
//thread: read from serial-com
void *myreadframe_thread(void *pt)
{
    while(1)
    {
        while( (myComDev.nread=read(myComDev.SerialCom,&(myComDev.insert_buf),1))>0 ) //get 1byte by 1byte from serial buffer
        {
            printf("%c",myComDev.insert_buf);

        }//while(..) end
    }//while(1) end
}
```

（图 48）应答数据原样打印

5. 底盘 PID 控制参数整定

我们的 miiboo 机器人底盘的 stm32 控制板中已经内置了整定好的 PID 参数，如果选用我们提供的控制板和电机，一般情况下是不需要整定 PID 的。

对于想体验一下 PID 参数整定过程或将我们的 miiboo 机器人底盘的 stm32 控制板应用到其他地方的朋友，这里给出了整定 PID 的整个操作过程和思路，方便大家学习和更深层次的研究。首先，对 PID 三个参数定性的分析，先有个感性的认识，如图 49。

	优点	缺点
KP 增加	系统响应速度加快	超调量增大，会引起系统的不稳定（振荡）
KI 增加	稳态误差减小，趋于稳态值的速度加快	超调量增大，系统相对稳定性变差（振荡）
KD 增加	响应速度加快，调节时间减小，减小超调量，克服振荡，使系统的稳定性提高	过大的 KD 值会因为系统噪声或者受控对象的大时间延迟而出现问题。微分环节对于信号无变化或变化缓慢的系统不起作用。

（图 49）PID 参数定性分析

其次，由于我们的 miiboo 机器人底盘的 stm32 控制板中采用的是增量式 PID，所以这里对增量式 PID 参数的特殊性进行一些说明，如图 50。

位置型：

$$u(k) = K_P\left[e(k) + \frac{1}{T_I}\sum_{i=0}^{k} Te(i) + T_D\frac{e(k)-e(k-1)}{T}\right]$$

$$u(k) = K_P\left[e(k) + \frac{T}{T_I}\sum_{i=0}^{k} e(i) + T_D\frac{e(k)-e(k-1)}{T}\right]$$

$$u(k) = K_P e(k) + K_I\sum_{i=0}^{k} e(i) + K_D[e(k)-e(k-1)]$$

其中：

e(k) = input_target – feedback_current，输入目标值与当前反馈值之差；

K_P 为比例系数；

$K_I = K_P\frac{T}{T_I}$，为积分系数；

$K_D = K_P\frac{T_D}{T}$，为微分系数；

T 为采样周期；

T_I 为积分时间；

T_D 为微分时间。

增量型：

$$u(k) = K_P e(k) + K_I\sum_{i=0}^{k} e(i) + K_D[e(k)-e(k-1)]$$
$$u(k-1) = K_P e(k-1) + K_I\sum_{i=0}^{k-1} e(i) + K_D[e(k-1)-e(k-2)]$$
$$\Delta u(k) = u(k) - u(k-1)$$
$$\Delta u(k) = K_P[e(k)-e(k-1)] + K_I e(k) + K_D[e(k)-2e(k-1)+e(k-2)]$$

（图 50）离散域位置式 PID 与增量式 PID 数学表达式

位置型 PID 的参数整定过程一般是，先整定 KP，然后整定 KI，最后整定 KD；对比位置型 PID 与增量型 PID 的数学表达式，可以发现位置型 KP 和增量型 KI 一样，位置型 KI 和增量型 KD 一样，位置型 KD 和增量型 KP 一样，如图 51。这样，增量型 PID 应该先整定 KI，然后整定 KD，最后整定 KP。这一点需要特别注意，弄错顺序的话会发现整定规律完全不适用的。

作用	响应速度（比例）	稳态误差（积分）	超调量（微分）
广义 PID	KP	KI	KD
位置型 PID	KP	KI	KD
增量型 PID	KI	KD	KP

（图 51）增量式 PID 参数特殊性说明

在机器人上进行具体 PID 整定操作之前，先对整定原理做一些讲解。下面的表述是针对增量型 PID 的，即 KI 为比例参数、KD 为积分参数、KP 为微分参数。这里使用试凑法对 miiboo 机器人底盘的增量 PID 参数进行整定：

第 1 步：

首先只整定比例部分。比例系数 KI 由小变大，观察相应的系统响应，直到得到反应快，超调小的响应曲线。系统若无静差或静差已小到允许范围内，并且响应效果良好，那么只须用比例调节器即可。

第 2 步：

若稳态误差不能满足设计要求，则需加入积分控制。整定时先置 KD 为较小值，并将经第 1 步整定得到的 KI 减小些(如缩小为原值的 0.8 倍)，然后增大 KD，并使系统在保持良好动态响应的情况下，消除稳态误差。这种调整可根据响应曲线的状态，反复改变 KI 及 KD，以期得到满意的控制过程。

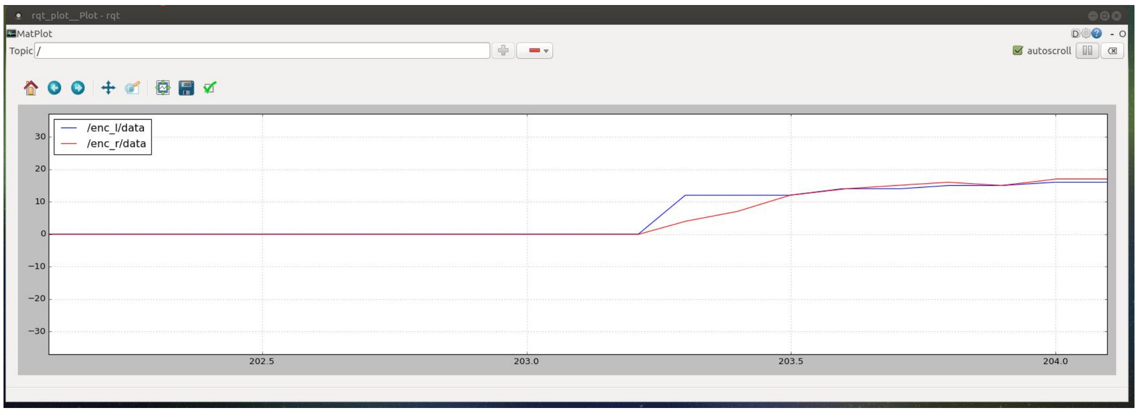
第 3 步：

若使用比例-积分调节器消除了稳态误差，但动态过程仍不能满意，则可加入微分环节。在第 2 步整定的基础上，逐步增大 KP，同时相应地改变 KI 和 KD，逐步试凑以获得满意的调节效果。

原理了解后，就要到实际的 miiboo 机器人上进行整定了，首先需要将底盘的 DATA-uart2 与 DEBUG-uart1 串口连接到机器人的主板树莓派 3 中，并确保被树莓派识别的串口设备号为底盘驱动设置的值，如果串口号不匹配需要先进行匹配，关于这部分内容将在 miiboo 机器人 SLAM 导航中做更详细的展开。然后，需要启动底盘控制节点、底盘调试节点、键盘控制节点。

```
#打开终端，启动底盘控制节点
roslaunch miiboo_bringup minimal.launch

#再打开一个终端，启动底盘调试节点，按提示输入命令
roslaunch miiboo_bringup pid_set.launch

#再打开一个终端，键盘控制节点
rosrun teleop_twist_keyboard teleop_twist_keyboard.py

#再打开一个终端，用 rqt_plot 对底盘速度曲线进行绘制，指定曲线数据来源的 topic
rosrun rqt_plot rqt_plot
```

键盘控制节点 teleop_twist_keyboard 需要通过 apt-get 命令来安装，rqt_plot 是 ROS 提供的绘图工具，关于这些的具体使用方法将在 miiboo 机器人 SLAM 导航中做更详细的展开。最后，就是通过观察速度曲线，按照试凑法的步骤，在底盘调试节点的终端中输入相应的 kp、ki、kd 参数，不断重复这个过程直到速度曲线达到一个比较满意的形状。rqt_plot 速度曲线的样子如图 52 所示。



（图 52）rqt_plot 速度曲线

6. 底盘里程计标定

机器人底盘运行的精度是衡量底盘的重要指标。底盘精度受里程计的走直线误差和转角误差影响。因此，需要对里程计的走直线和转角进行标定，尽量减小误差。miiboo 机器人底盘的 ROS 驱动中已经写好了相应的标定程序，跟里程计标定有关的文件主要有：
.../miiboo_bringup/launch/check_linear.launch 为里程计走直线标定启动文件
.../miiboo_bringup/launch/check_angular.launch 为里程计转角标定启动文件
.../miiboo_bringup/launch/minimal.launch 为设置标定参数及底盘控制启动文件

下面是标定步骤过程。

第一步：

打开终端，给标定脚本赋予可执行权限

```
cd miiboo_bringup/scripts/
sudo chmod +x ./*
```

第二步：

连接好底盘 DATA-uart2 串口，启动底盘

```
roslaunch miiboo_bringup minimal.launch
```

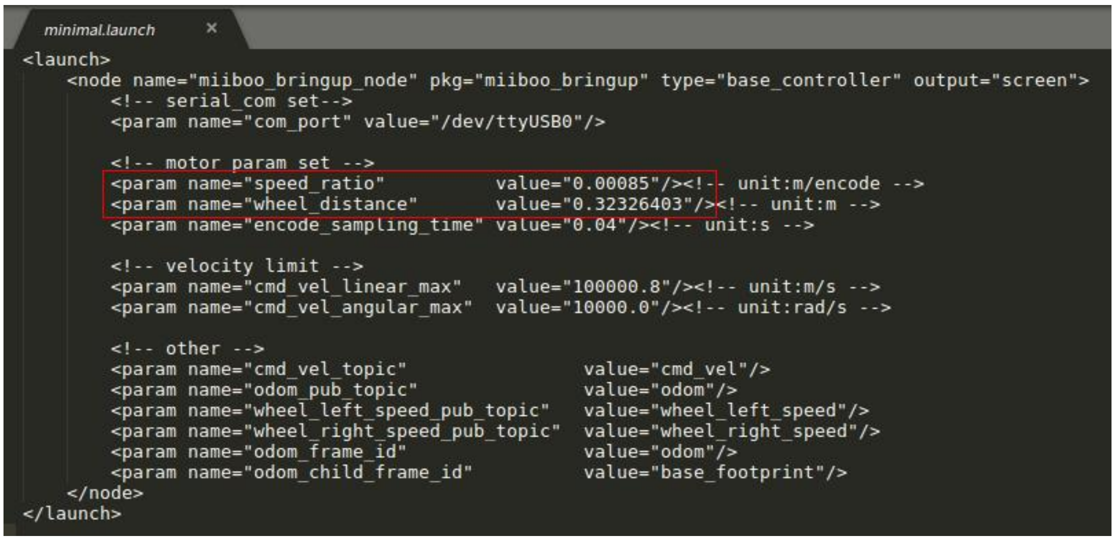
第三步：

设定前进 1 米的目标，对走直线进行标定

```
roslaunch miiboo_bringup check_linear.launch
```

测量底盘停止时实际走的直线距离 M，按下面的规则调整里程计直线参数 speed_ratio
如果 M > 1 米，增大 speed_ratio
如果 M < 1 米，减小 speed_ratio

里程计的参数存放在 miiboo_bringup/launch/minimal.launch 文件中，如图 53 所示。



```
<launch>
    <node name="miiboo_bringup_node" pkg="miiboo_bringup" type="base_controller" output="screen">
        <!-- serial_com set-->
        <param name="com_port" value="/dev/ttyUSB0"/>

        <!-- motor param set -->
        <param name="speed_ratio"          value="0.00085"/><!-- unit:m/encode -->
        <param name="wheel_distance"       value="0.32326403"/><!-- unit:m -->
        <param name="encode_sampling_time" value="0.04"/><!-- unit:s -->

        <!-- velocity limit -->
        <param name="cmd_vel_linear_max"   value="100000.8"/><!-- unit:m/s -->
        <param name="cmd_vel_angular_max"  value="10000.0"/><!-- unit:rad/s -->

        <!-- other -->
        <param name="cmd_vel_topic"                value="cmd_vel"/>
        <param name="odom_pub_topic"               value="odom"/>
        <param name="wheel_left_speed_pub_topic"   value="wheel_left_speed"/>
        <param name="wheel_right_speed_pub_topic"  value="wheel_right_speed"/>
        <param name="odom_frame_id"                value="odom"/>
        <param name="odom_child_frame_id"          value="base_footprint"/>
    </node>
</launch>
```

（图 53）里程计待标定参数

修改好参数后，需要保存，然后重新启动一下底盘节点，这样参数才能生效。

```
roslaunch miiboo_bringup minimal.launch
```

重复第三步的操作，直到走直线的误差达到我们能接受的范围（比如 1%的误差），则进入下一步。

第四步：

设定旋转 360 度的目标，对转角进行标定

```
roslaunch miiboo_bringup check_angular.launch
```

测量底盘停止旋转时实际转过的角度 A，按下面的规则调整里程计转角参数 wheel_distance
如果 A > 360 度，减小 wheel_distance
如果 A < 360 度，增大 wheel_distance

上面已经讲过，里程计的参数存放在 miiboo_bringup/launch/minimal.launch 文件中，如图 53 所示。

修改好参数后，需要保存，然后重新启动一下底盘节点，这样参数才能生效。

```
roslaunch miiboo_bringup minimal.launch
```

重复第四步的操作，直到走转角的误差达到我们能接受的范围（比如 1%的误差），则标定完成。

当然，有兴趣的朋友可以阅读 miiboo_bringup/scripts/中的标定脚本源码，结合航迹推演算法，理解里程计标定的整个原理。其实 wheel_distance 这个参数是编码脉冲值与电机轮胎位移值的一个比例系数，简单点说就是电机转过一个编码脉冲，这个时候电机轮胎走过多少距离；wheel_distance 这个参数是左右两个轮子的间距。有了这个认识后，我们可以在这两个参数的理论值附近对参数进行微调，标定起来会更快。

后记

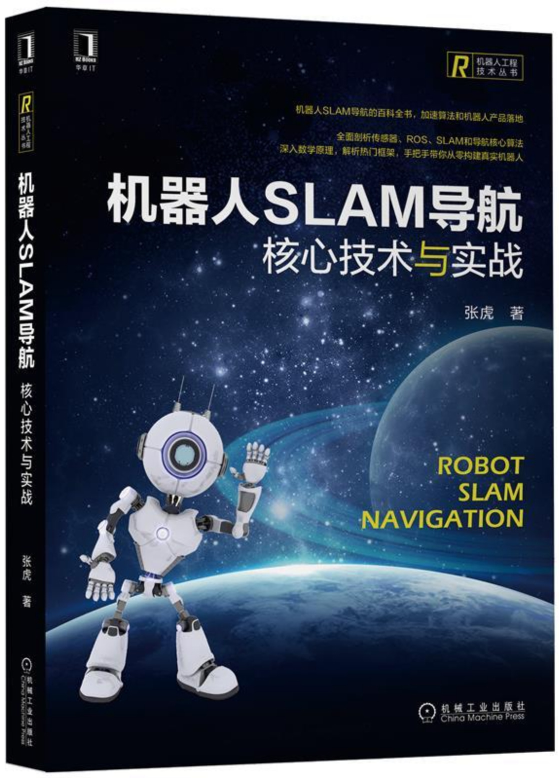
为了防止后续大家找不到本篇文章，我同步制作了一份文章的 pdf 和本专栏涉及的例程代码放在 github 和 gitee 方便大家下载，如果下面给出的 github 下载链接打不开，可以尝试 gitee 下载链接：

- github 下载链接：
  https://github.com/xiihoo/DIY_A_SLAM_Navigation_Robot

- gitee 下载链接：
  https://gitee.com/xiihoo-robot/DIY_A_SLAM_Navigation_Robot

参考文献

张虎,机器人 SLAM 导航核心技术与实战[M].机械工业出版社,2022.


机器人工程技术丛书
机器人SLAM导航的百科全书，加速算法和机器人产品落地
全面剖析传感器、ROS、SLAM和导航核心算法
深入数学原理，解析热门框架，手把手带你从零构建真实机器人
机器人SLAM导航
核心技术与实战
张虎 著
ROBOT
SLAM
NAVIGATION
机械工业出版社
China Machine Press