**使用python，通过串口ROS直接控制电机驱动器(1)**

2019-01-03 10:56:54 [庆钊你好呀](https://me.csdn.net/zengqz123) 阅读数 1461更多

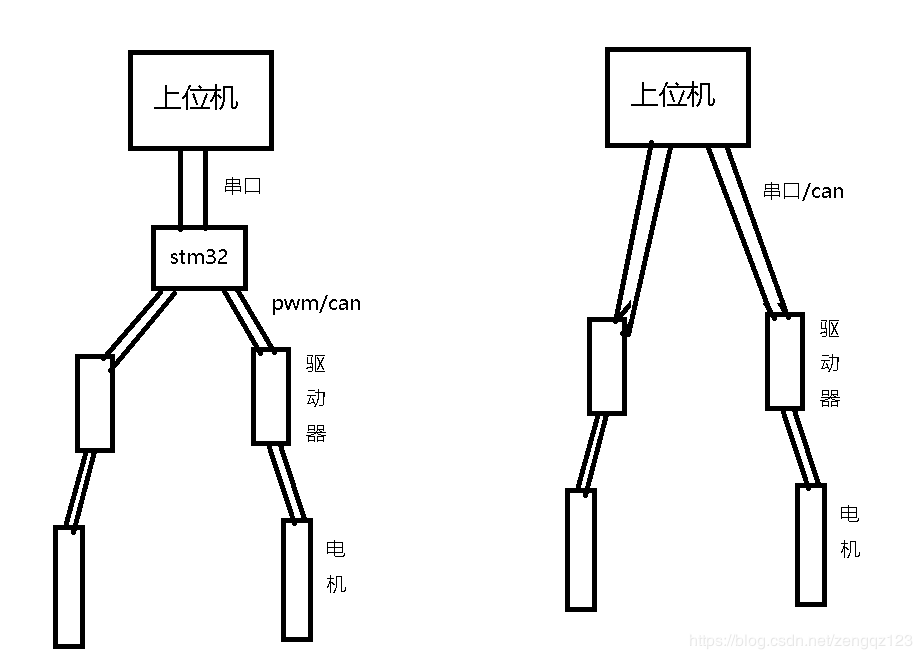
分类专栏： [ROS](https://blog.csdn.net/zengqz123/article/category/8044415) [ROS小车搭建](https://blog.csdn.net/zengqz123/article/category/8044418)

版权声明：本文为博主原创文章，遵循 [CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/) 版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/zengqz123/article/details/85682039>

最近在一个公司实习，公司需要我用ROS搭建一个机器车地盘，因此我开始学习关于ROS方面的东西。通常传统的方法大家搭建机器车地盘的时候会想到用一个stm32开发板作为中间层次，对上层上位机通过ros-stm32串口的方式通讯，对下层电机驱动器通过pwm的模拟电压来控制电机。但是公司给我的驱动器的通讯方式不仅仅是模拟量的输出，还可以通过can或者串口的形式来控制驱动器。

我之前的学习基本是在上位机上进行开发，因此不是很熟悉stm32，在捣鼓一两天后我便有点放弃stm32通过pwm控制电机。电机的厂家虽然提供了can通讯的stm32工程文件。但是我测试很久后还是无法驱动电机。因此我产生了一个新的想法。为什么我们有一个这么智能的驱动器，却还需要和传统有刷电机那样用pwm的形式来控制电机呢？白白浪费驱动器的性能。因此我想到了一个新方法：



我能否能通过Python，通过调用串口或者can通讯的形式来驱动电机。于是我查了一下可行性。串口可以很方便的通过pyserial模块来通讯，而can通讯的资料就十分少了。因此我选择用串口来完成这个项目。

相对于传统的电机控制结构，上位机直接控制有以下几个优点：

1. 通过直接驱动的方式，大大降低了stm32的开发工作量。减少学习成本和开发时间成本
2. 如果后期要新加入电机时，只需要增加串口就可以直接控制电机，而不用修改底层代码

通过与公司其他做嵌入式开发的交流，我意识到我的这种方案可能会有以下几点问题：

1. 串口通讯的稳定性不是很好，因此要是有电磁辐射会造成丢包问题。但是我认为传统的上位机与stm32也是以串口的形式通讯，也是存在同样问题
2. 如果上位机死机后，传统的形式stm32可以关闭电机，而新方案无法实现这个功能。针对这个问题，驱动器需要上位机不停的查询驱动器当前状态，否则会锁死电机
3. 传统的方案可以做二次pid，新方案只能使用驱动器的pid，因此控制精度没有太高。

在阅读本教程之前，我先交代一下我开发环境以及需要提前掌握的内容：

* 1. 本项目使用的电机为中菱机器人轮毂电机，控制器为中菱ZLAC706。如果你有自己的带串口的电机驱动器，可以参考本教程。
  2. 本项目首先是在windows系统下用anaconda进行测试，然后使用pycharm进行开发。最后移植到ubuntu 16.04下的ROS Kinetic。
  3. 懂Python语言以及编程能力（建议学习廖雪峰的Python，会运用到多线程等比较难的功能）
  4. 知道ROS（Kinetic）系统的基本工作原理

第一节：运用测试程序测试电机是否正常工作

在第一节中，我们需要测试一下用Python串口控制电机的可行性。请确保自己安装好anaconda。

第一步：安装pyserial模块（遇到问题百度解决）

打开命令行输入:

pip install pyserial

第二步，参考驱动器使用手册，了解驱动器串口(RS232)通讯协议：

第三步：根据通讯协议写测试代码：

import serial import time motor\_enable\_date=b'\x00\x00\x01\x01' motor\_speed\_mode=b'\x02\x00\xC4\xC6' motor\_accdec\_set=b'\x0A\x14\x14\x32' motor\_speed\_set=b'\x06\x00\x88\x8E' motor\_status=b'\x80\x00\x80' #设置速度模式02 00 C4 C6 #设置加减速时间0A 14 14 32 #设置转速06 00 88 8E #使能电机00 00 01 01 #停机电机 00 00 00 00 s=serial.Serial('com4',57600) s.bytesize = 8 s.stopbits = 1 s.parity = "N" motor\_enable(s) time.sleep(1) n=s.readline() if n: data= [hex(x) for x in bytes(n)] print(data) s.flush() #print('ok') # 关闭 串口 if s.isOpen(): s.close() def motor\_enable(serial): try: serial.write(motor\_speed\_mode) serial.write(motor\_accdec\_set) time.sleep(0.1) serial.write(motor\_speed\_set) time.sleep(0.1) serial.write(motor\_enable\_date) time.sleep(0.1) serial.write(motor\_status) time.sleep(1) serial.write(motor\_status) time.sleep(1) serial.write(motor\_status) time.sleep(1) serial.write(motor\_status) time.sleep(1) print('send ok') except: print('cant write motor\_enable')

在使用卖家提供的软件测试电路没有问题后，便可以开始测试编写程序了；一下是我给我使用的电机编写的测试程序。

代码介绍：

首先在查看电脑设备管理器中查看电机所对应的串口名字，我的串口名字为com4，因此我通过使用s=serial.Serial('com4',57600)，创建串口对象为com4，波特率为57600

接着的是串口的一些参数配置，接下来我调用了一个我创建的motor\_enable函数，根据使用手册发送速度控制步骤。

注意我在每个步骤中间加入了一个time.sleep()函数，开始我没加入这个函数时，我这么设置也无法让电机工作。后来觉得是不是每个信息发送太快了，于是在每个步骤中间加入延迟函数，最后能正常工作。在起动电机后，根据使用手册可得需要不停的发送状态查询指令，电机才不会锁轴。最终关闭串口。

本节总结：

万事开头难，虽然只是短短的几步控制。作为初学者的我也遇到了很多的坑。例如用卖家软件直接控制驱动器的时候怎么也连接不上，后来问别人才知道我用的线是另一款电机的线，引脚不同无法测试成功。接下来是与驱动器通讯的过程中要用十六进制进行通讯。其他教程中都要用到一个bytes.fromhex()的函数，把字符转为十六进制的数。但是这个函数只能在python3才能提供，而ROS上搭载的python2是无法运行这个程序的，因此又得重新改程序。初期调试成功对后期的开发有很大的帮助，同时对python的理解能够更深一步！！！！！！运用python最大的好处就是我们能够通过面向对象的方式来编写程序，因此在这一节里我们首先创建一个电机的对象，在脑海里构思，电机有哪些属性，需要哪些控制方法！

在考虑到这么多的属性与方法之后，我参考卖家基于stm32的can通讯类方法，写了以下设置初始化方式：

**class** SpeedMotor:  
 **def** \_\_init\_\_(self, device, acc, dcc):  
 *#z axia real speed真实速度* self.rel\_speed = 0  
 *#set speed 设置的速度* self.set\_speed = 0  
 *#running state运行状态 s* elf.run = **False** *#fault condition故障状态* self.fault = **None** *#moter voltage value电机电压* self.voltage = 0  
 *#motor current value 电机电流* self.current = 0  
 self.serial = serial.Serial(device, 57600)  
 self.serial.timeout = 0  
  
 **def** motor\_start(self):  
  
 **def** motor\_stop(self):  
  
 **def** motor\_speed\_set(self, speed):

Initialisation method: 初始化方式是：

m = SpeedMotor('com4',20,20)

During initialisation, SpeedMotor is set to serialport com4 with starting aceleration set to 20 and deceleration to -20

初始化SpeedMotor，设置device为com4，加速度为20，减速度为20。

Need to write 3 more function motor\_start(),motor\_stop(),motor\_speed\_set()

并设置了三个方法为：motor\_start(),motor\_stop(),motor\_speed\_set()

继续上节的工程，我给电机做了一个简单的建模后，就需要具体实现每一个功能了。

首先我们要在脑海里建立好一个模型，我们一方面向串口输入数据，一方面还要从串口里接收数据。在刚开始的时候，我定义了很多个函数，在发送完数据后立马就接收驱动器回复的数据，在后期想用多线程的时候很容易产生错误，导致驱动器不知道我发送的是什么指令需要使用线程锁来编写，又导致程序非常的复杂。在参考网上的的串口多线程编程的过程中，我转变了思想，为什么不定义一个线程向驱动器写数据，另一个线程接收驱动器数据并解析数据呢？这样很大的提高了程序的性能。

*#!/usr/bin/env python***import** serial  
**import** time  
**import** threading  
motor\_speed\_mode = **b'\x02\x00\xC4\xC6'**motor\_status = **b'\x80\x00\x80'**motor\_start = **b'\x00\x00\x01\x01'  
class** SpeedMotor:  
 **def** \_\_init\_\_(self, device, acc, dcc):  
 *# 真实速度* self.rel\_speed = 0 *# 设置的速度* self.set\_speed = 0 *# 运行状态* self.run = **False** *# 故障状态* self.fault = **None** *# 电机电压* self.voltage = 0 *# 电机电流* self.current = 0 *# 设置串口通讯* self.serial = serial.Serial(device, 57600)  
 self.serial.timeout = 0 *# 开启读数据线程* threads = []  
 t1 = threading.Thread(target=self.read\_motor)  
 threads.append(t1)  
 **for** t **in** threads:  
 t.start()

通过初始函数的过程，定义一个读线程。

下面是读取线程的代码：

**def** read\_motor(self):  
 **while True**:  
 n = self.serial.inWaiting() *# 等待数据的到来，并得到数据的长度* **if** n: *# 如果有数据* n = self.serial.read(n) *# 读取n位数据* s = [hex(x) **for** x **in** bytes(n)]  
 **for** i **in** range(len(s)):  
 s[i] = int(s[i], 16)  
 print(s)

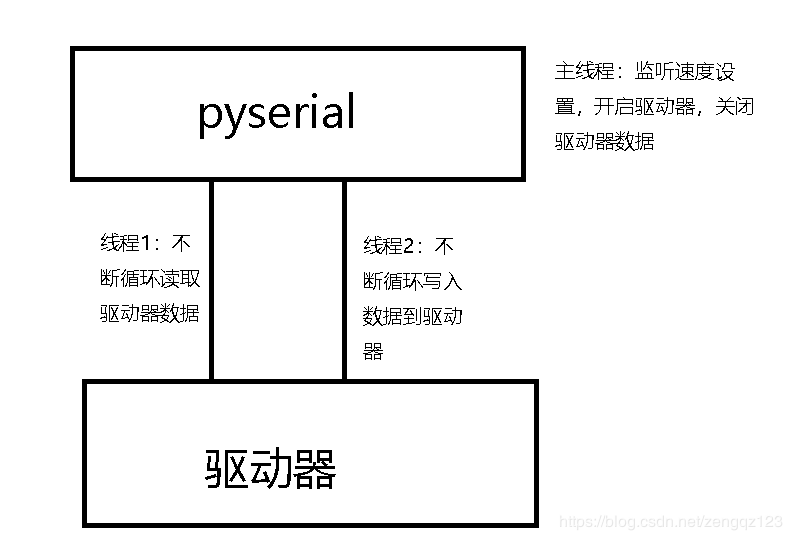
通过不断的循环，从而不断的接收驱动器的数据。

接下来我便是驱动电机的代码实现了，由于需要发送十六进制的字节码，因此我们要引入Python的struct模块，把数字转换为字节串。

**def** motor\_speed\_set(self):  
 a1 = 6  
 a4 = check\_code(a1, self.set\_speed)  
 self.serial.write(struct.pack(**">BhB"**, a1, self.set\_speed, a4))  
  
**def** check\_code(a1, a2):  
 buffer = struct.pack(**">bh"**, a1, a2)  
 buffer = buffer[0] + buffer[1] + buffer[2]  
 check\_num = (struct.pack(**">l"**, buffer)[-1])  
 **return** check\_num

如图，代码为图片的实现形式，check\_code的功能是根据地址位与数据位算出校验位，并取校验位的最低位。

python驱动电机部分完结，现在把全部代码公布，基本实现驱动器速度控制部分功能。接下来我将把电机驱动器移植到ROS上，实现驱动器与ROS之间的通讯。下图为我代码的基本框架：

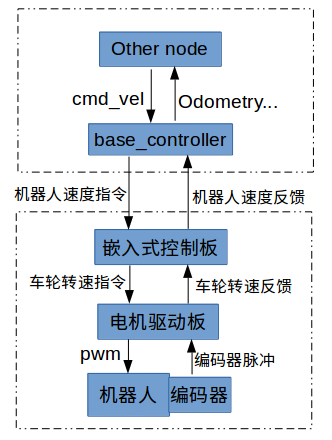


*#!/usr/bin/env python***import** serial  
**import** time  
**import** threading  
**import** struct  
motor\_speed\_mode = **b'\x02\x00\xC4\xC6'**motor\_status = **b'\x80\x00\x80'**motor\_start = **b'\x00\x00\x01\x01'**motor\_stop = **b'\x00\x00\x00\x00'  
class** SpeedMotor:  
 **def** \_\_init\_\_(self, device,speed):

self.rel\_speed = 0 *# 真实速度* self.set\_speed = speed *# 设置的速度* self.run = **False** *# 运行状态*

self.fault = **None** *# 故障状态* self.voltage = 0 *# 电机电压* self.current = 0 *# 电机电流* self.serial = serial.Serial(device, 57600) *# 设置串口通讯*  
 self.serial.timeout = 0 *# 开启读数据线程* threads = []  
 t1 = threading.Thread(target=self.read\_motor)  
 threads.append(t1)  
 t2 = threading.Thread(target=self.send\_motor)  
 threads.append(t2)  
 **for** t **in** threads:  
 t.start() *# 设置为速度模式* self.serial.write(motor\_speed\_mode)  
 time.sleep(0.1) *# 设置加减速度* self.serial.write(**b'\x0A\x14\x14\x32'**)  
 time.sleep(0.1)  
  
 **def** motor\_speed\_set(self):  
 a1 = 6  
 a4 = check\_code(a1, self.set\_speed)  
 self.serial.write(struct.pack(**">BhB"**, a1, self.set\_speed, a4))  
 **def** set\_status(self):  
 self.serial.write(motor\_status)  
 time.sleep(0.2)  
 **def** motor\_start(self):  
 self.serial.write(motor\_start)  
 self.run = **True  
 def** motor\_stop(self):  
 self.run = **False** self.serial.write(motor\_stop)  
 **def** read\_motor(self):  
 **while True**:  
 n = self.serial.inWaiting() *# 等待数据的到来，并得到数据的长度* **if** n: *# 如果有数据* n = self.serial.read(n) *# 读取n位数据* s = [hex(x) **for** x **in** bytes(n)]  
 **for** i **in** range(len(s)):  
 s[i] = int(s[i], 16)  
 **if** len(s) == 32:  
 **for** i **in** range(int(len(s) / 4)):  
 addr = s[4 \* i]  
 **if** addr == 128:  
 **if** s[2] == 0:  
 print(**"Stop停止状态"**)  
 **elif** s[2] == 1:  
 print(**"start up state启动状态"**)  
 **elif** s[2] == 2:  
 print(**"over current 过流"**)  
 **elif** s[2] == 4:  
 print(**"over voltage过压"**)  
 **elif** s[2] == 8:  
 print(**"encoder not working编码器故障"**)  
 **elif** s[2] == 16:  
 print(**"over heated过热"**)  
 **elif** s[2] == 32:  
 print(**"under voltage欠压"**)  
 **elif** s[2] == 64:  
 print(**"overload过载"**)  
 **elif** addr == 225:  
 high\_data = s[4 \* i + 1]  
 low\_data = s[4 \* i + 2]  
 self.voltage = high\_data \* 256 + low\_data  
 print(**"电压:"** + str(self.voltage))  
 **elif** addr == 226:  
 high\_data = s[4 \* i + 1]  
 low\_data = s[4 \* i + 2]  
 self.current = (int(high\_data \* 256 + low\_data))/100  
 print(**"电流:"** + str(self.current))  
 **elif** addr == 228:  
 high\_data = s[4 \* i + 1]  
 low\_data = s[4 \* i + 2]  
 self.rel\_speed = (int(high\_data \* 256 + low\_data))\*6000 / 16384  
 print(**"转速:"** + str(self.rel\_speed))  
 **elif** addr == 230:  
 **None  
 elif** addr == 231:  
 **None  
 elif** addr == 232:  
 **None  
 elif** addr == 233:  
 **None  
 elif** len(s) == 2:  
 **if** s[0] == 6:  
 print(**"速度设置成功"**)  
 time.sleep(0.1)  
 **def** send\_motor(self):  
 **while True**:  
 **if** self.run:  
 self.set\_status()  
 time.sleep(0.2)  
 self.motor\_speed\_set()  
 time.sleep(0.3) **'''通过a1,a2,a3三个码计算出校验和码'''  
 def** check\_code(a1, a2):  
 buffer = struct.pack(**">bh"**, a1, a2)  
 buffer = buffer[0] + buffer[1] + buffer[2]  
 check\_num = (struct.pack(**">l"**, buffer)[-1])  
 **return** check\_num  
  
m = SpeedMotor(**'com4'**,100)  
m.motor\_start()  
**for** i **in** range(100):  
 m.set\_speed = i  
 time.sleep(0.5)  
 m.motor\_stop()

在做好电机的驱动后，接下来我们便可以把它移植到ROS平台上，移植的过程其实很简单，就是按照官方的例程订阅和发布消息。首先我们要确认我们的程序要订阅些什么内容，比如说很容易想到我们要订阅速度信息，并发布实际速度信息，等等。



如图所示是传统电机的驱动模型，因此我们移植到ROS平台上需要订阅cmd\_vel(速度信息),和发布Odometry(里程计信息)。

在订阅了速度信息后，我的地盘是两车轮模型，我要把系统的速度分解到每个车轮上，因此参考网上的两车轮差速模型计算出每个车轮的速度。

在参考在GitHub上Forrest的电机实现后GitHub地址，我有了更深的理解：

Taking reference to GitHub Forrest Motor driver, I have a better understanding:

基于串口通信的ROS小车基础控制器，功能如下：

Base on the ROS serial communication of a small vehicle, the capability is listed below:

1. Using ROS control with fixed format for serial communication, the small vehicle is able to move.
2. Subscribe /cmd\_vel topic
3. published the distance to /odom

1.实现ros控制数据通过固定的格式和串口通信，从而达到控制小车的移动

2.订阅了/cmd\_vel主题，只要向该主题发布消息，就能实现对控制小车的移动

3.发布里程计主题/odom

首先我们可以参考里面的两车轮模型，由于我们订阅的是速度消息类型为Twist，查询文档（文档链接）后可知twist包含了六个数据，分别为线速度的xyz和角速度的xyz。因此我们参考解决方法：

Taking reference to tist which has six values, x,y,z velocities and x,y,z, accelerations. We then take reference with this to find our solution:

void callback(const geometry\_msgs::Twist & cmd\_input) //subscribe /cmd\_vel topic订阅/cmd\_vel主题回调函数

{

string port("/dev/ttyUSB0"); //小车串口号

unsigned long baud = 115200; //小车串口波特率

serial::Serial my\_serial(port, baud, serial::Timeout::simpleTimeout(1000)); //配置串口

angular\_temp = cmd\_input.angular.z ;//获取/cmd\_vel的角速度,rad/s

linear\_temp = cmd\_input.linear.x ;//获取/cmd\_vel的线速度.m/s //将转换好的小车速度分量为左右轮速度

left\_speed\_data.d = linear\_temp - 0.5f\*angular\_temp\*D ;

right\_speed\_data.d = linear\_temp + 0.5f\*angular\_temp\*D ; //存入数据到要发布的左右轮速度消息

left\_speed\_data.d\*=ratio; //放大１０００倍，mm/s

right\_speed\_data.d\*=ratio;//放大１０００倍，mm/s

for(int i=0;i<4;i++) //将左右轮速度存入数组中发送给串口

{

speed\_data[i]=right\_speed\_data.data[i];

speed\_data[i+4]=left\_speed\_data.data[i];

} //在写入串口的左右轮速度数据后加入”/r/n“

speed\_data[8]=data\_terminal0;

speed\_data[9]=data\_terminal1; //写入数据到串口

在有了别人的代码后，我就很容易的参考他的方法来实现电机的这个功能，因此我简单的写了一个Python程序，通过调用之前写的Python驱动程序，实现了对单轮电机的速度控制。

该程序的实现原理：

第一：将导入单轮电机的类实例化，创建对象。

第二，创建一个叫base\_controller的节点，然后订阅/cmd\_vel（速度）主题，定义要发布的/odom（里程计）主题（里程计功能我还暂未实现，但是可以通过读取驱动器返回的代码实现里程计功能）。

对于车轮的速度控制，我们可以安装一个简易的程序，这个程序能注册一个cmd\_vel节点，通过键盘来控制

下载键盘控制的ROS包 ( Donwload ROS package )

>> cd catkin\_ws

>> mkdir srcgit

>> cd ~/catkin\_ws/srcgit

>> clone <https://github.com/ncnynl/teleop_twist_keyboard.git>

Go into the download folder select the keyboard\_teleop\_zbot.py, right clock the mouse to make it executable.

进入下载好的ROS包的文件夹，选中 keyboard\_teleop\_zbot.py ，右键->设为可执行文件

最后一步:

Last step:

$ cd ~/catkin\_ws

$ catkin\_make

这样子我们的键盘控制包就能使用了

With this we are able to use the keyboard to control the motor.