# 直流电机的的控制

## 硬件连接部分：

大多数的电机驱动接线基本一致，需要接入外接电源、PWM输入、还有单片共地。输出的话，也就两个输出线。（下图以L298N为电机驱动示例）

1. 输入12V

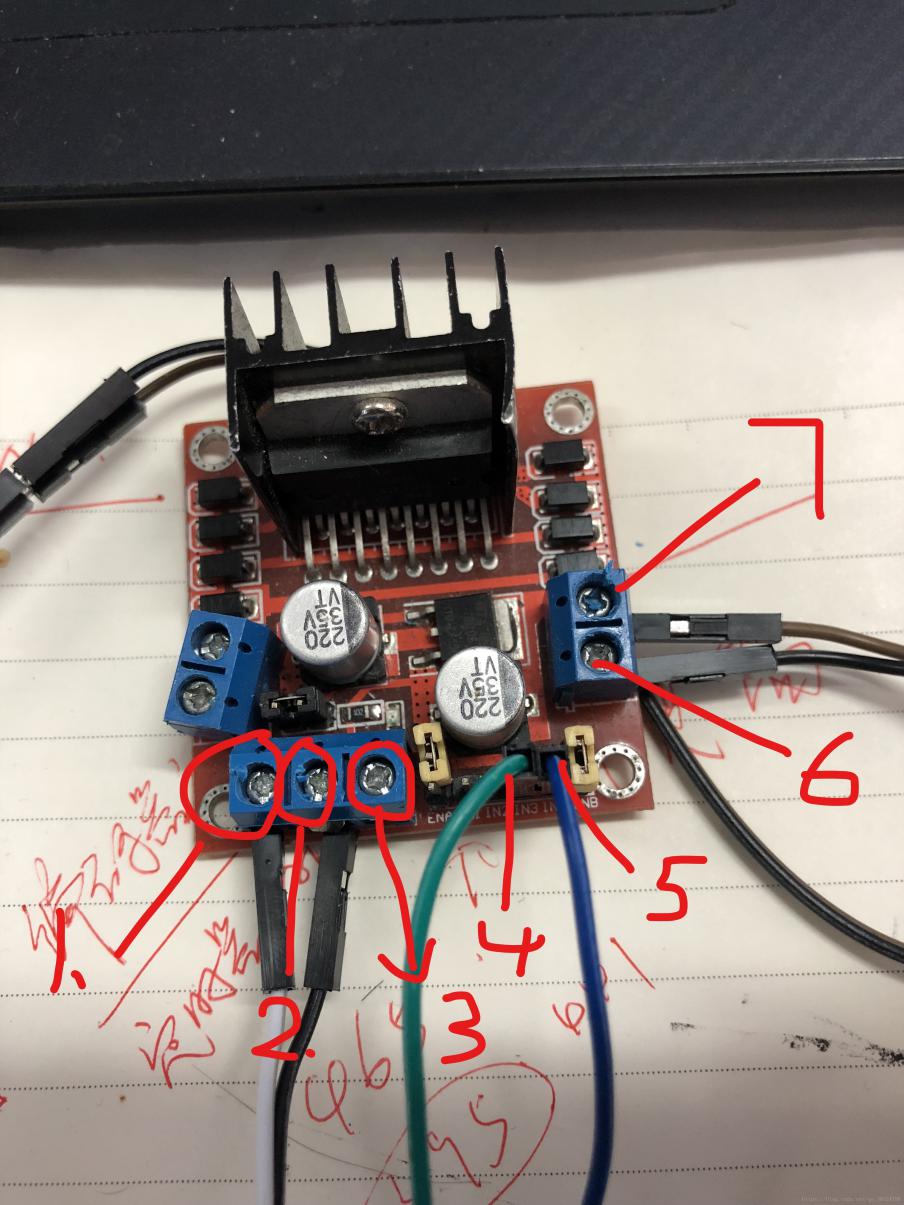
2.接地

3.输出5V

4.5 连接STM32的io

6.7 连接直流电机输入端

注：连接完成后还要有一个连接，就是将STM32的GND跟电源输入端GND连接在一起，否则电机不会转动，会出现一直抖动的情况。



原理：利用STM32 的IO输出不同占空比的脉冲信号来达到调速的过程，当占空比大的时候就表明在一个脉冲周期内高电平的时间越长，而脉冲信号输出频率极快，高电平的时间越长就表明在一段时间内IO口输出的控制电压越高，控制电压越高使得L298N输出的电压越高，这样就使得电机的转速越快。

## 软件部分：

配置pwm波输出

### PWM.H

#ifndef \_\_PWM\_\_H

#define \_\_PWM\_\_H

#include "sys.h"

#include "stm32f10x.h"

void TiM4\_PWM\_Init(u16 value ) ;

#endif

PWM.c

#include "PWM.h"

void TiM4\_PWM\_Init(u16 value )

{

TIM\_TimeBaseInitTypeDef TIM\_Timeinit;

TIM\_OCInitTypeDef TIM\_OCInitstruct;

GPIO\_InitTypeDef GPIO\_InitStrucyure;

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM4,ENABLE); //使能定时器4时钟

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB,ENABLE); // 使能GPIO

GPIO\_InitStrucyure.GPIO\_Mode=GPIO\_Mode\_AF\_PP; //改为复用推挽输出GPIO\_Mode\_AF\_PP

GPIO\_InitStrucyure.GPIO\_Pin=GPIO\_Pin\_6;

GPIO\_InitStrucyure.GPIO\_Speed=GPIO\_Speed\_50MHz;

GPIO\_Init(GPIOB,&GPIO\_InitStrucyure); //初始化io口

TIM\_Timeinit.TIM\_ClockDivision = TIM\_CKD\_DIV1 ; //

TIM\_Timeinit.TIM\_CounterMode = TIM\_CounterMode\_Up; //计数模式

TIM\_Timeinit.TIM\_Prescaler = 0; //预分频系数为0 则频率为不分频 72mhz/1 = 72Mhz

TIM\_Timeinit.TIM\_Period = 7199; // 72MHZ /(7199+1) = 10KHZ 7199 为计数总值

TIM\_TimeBaseInit(TIM4,&TIM\_Timeinit);

TIM\_OCInitstruct.TIM\_Pulse = value; //占空比 这里是0-2399

TIM\_OCInitstruct.TIM\_OCMode = TIM\_OCMode\_PWM2 ; //模式2

TIM\_OCInitstruct.TIM\_OCNPolarity = TIM\_OCPolarity\_High ; //极性输出高

TIM\_OCInitstruct.TIM\_OutputState = TIM\_OutputState\_Enable ; //

TIM\_OC1Init(TIM4,&TIM\_OCInitstruct); //TIM\_OCX 通道1 就OC1

TIM\_OC1PreloadConfig(TIM4,TIM\_OCPreload\_Enable);

TIM\_Cmd(TIM4,ENABLE); //使能定时器

}

这里利用的是  TIM\_OCInitstruct.TIM\_Pulse = value ;来实现不同的占空比，这里意思是计数器开始计数，直到到达value这个数值之前输出低电平，到达之后的输出高电平，value的值在（0-7199）之间。

利用PB6口对用的定时器4通道1输出PWM

假设主函数中调用

int main(void)

{

TiM4\_PWM\_Init(0);

}

输出全都是高电平

假设再主函数中调用

int main(void)

{

TiM4\_PWM\_Init(3200);

}

应该输出一个占空比为0.5的波形图。

注意几点：

1、电机启动了可是电机出现噪声，这是我们设置输出的PWM波的频率太低了，

TIM\_Timeinit.TIM\_Prescaler = 0; //预分频系数为0 则频率为不分频 72mhz/1 = 72Mhz

TIM\_Timeinit.TIM\_Period = 7199; // 72MHZ /(7199+1) = 10KHZ 7199 为计数总值

这样设置的时候输出的是10KHZ频率的PWM，可设置高一些 如 0 899，这样输出的是80KHZ的波形

2、L298N 发热严重有臭味，这是频率设置太高的原因，导致元件的发热严重，当我设置为 0 899 时候出现过这个问题。

施加过高的频率，即过高频率的输出电压由于感抗作用会使得电流很小，就会导致电机‘无力’，并且发热严重。

# 步进电机的控制

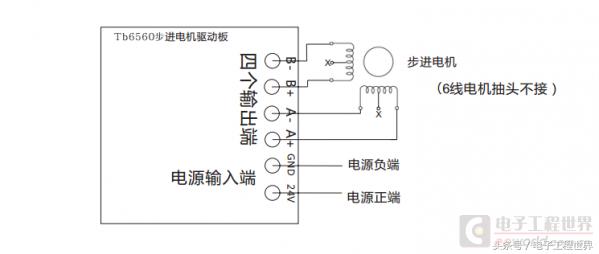
步进电机和直流电机可以说是我们平时最常用的俩种电机，相比直流电机，步进电机可以实现的精准控制也让其可以做到很多直流电机无法做到的事情。

需要注意的地方就是：步进是通过“数”脉冲的方法控制转速（也就说驱动器检测到一个脉冲，就会转一定的角度，脉冲给得越快速度也就越快。需要通过调节pwm波的频率来调节速度），而直流电机是通过改变占空比来调节速度，搞清楚了就很简单。

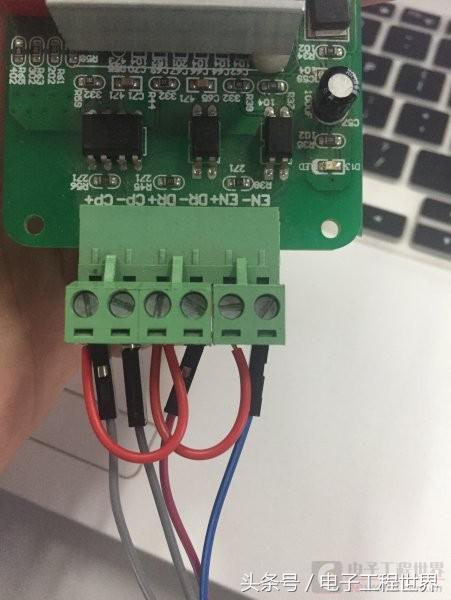
## 1.步进电机和TB6560的接线图

步进电机驱动TB6560都有俩个接口，一端为输出，一端为输入，输出部分接步进电机，输入部分接单片机IO口，辨别俩个端口的方法是，看俩个端口旁标注的接口定义，笔者的输出口六个接口分别为VCC GND A+ A- B+ B-,输入口分别为EN- EN+ DR- DR+ CP- CP+(如图)

输出端接线的时候只需要注意，将步进电机的同相的线接到一起（同为A或同为B，如图），确定哪跟线只需要用万用表测即可，短接的俩根线为同相。

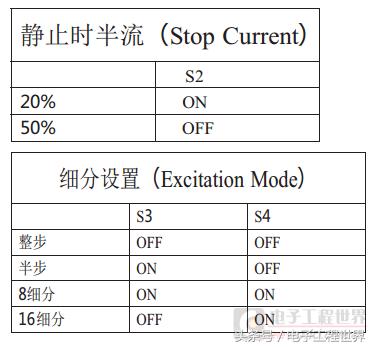


输入端接线的方法有俩种，一种是共阳法，另一种是共阴法，我使用的是共阳接法，所以只介绍共阳法，共阴类似共阳，请读者自己参照共阳研究。共阳法接线要把EN+ DR+ CP+短接，然后接到VCC（STM32的VCC为3.3V，如图）。



## 2.TB6560的参数说明（每种驱动器都有所差别、看说明进行调节选择）

首先要说明的是输入端口的几个字母的意思，EN为使能端，当EN端为高电平时，为工作状态；CP为脉冲输入端，该端口有脉冲时步进电机转动；DR为方向信号端，高电平和低电平时方向相反。然后驱动上有一排开关，驱动生产厂家不同，开关的功能也会有所区别，但共同的特点是都会有“细分设置”和“工作电流”，细分设置指的是步进电机每一步转多少度，以一整步为最大，分的越小，那么一步转过的角度就会越少，这里需要注意的是，最开始调的时候，最好调到整步，如果细分太多的话，即使步进电机转动了，可能你也不会有所察觉；工作电流要注意，设置电流不能超过电机的额定电流。具体设置这些参数根据驱动上的说明，打开相应开关即可。



## 程序设计思路

### a.让电机转起来（也可以跳过此步骤）

在测试的时候，首先要确保单片机和驱动接线是正确的，使能端接高电平或悬空，方向端接高电平或低电平或者悬空，脉冲管脚接一个STM32的IO口，这个管脚通过延时函数输出一个脉冲，但是这个脉冲的频率不能太快，否则电机不能正常启动，这个管脚最好接一个小灯，小灯每闪一次，步进电机会走一步，如果小灯在闪，步进电机没转的话，就可以排除程序问题，检查接线就好了。

### b.改变电机转速

等步进电机可以转起来的时候，就可以把延时函数改成定时器中断了，通过定时器控制脉冲频率。这时候改变计数器装入的值就可以改变频率快慢，就可以实现电机的转速控制，还要注意：步进电机速度越快，提速的加速度必须越小，否则步进电机就会卡死。

### c.电机的启动截至、改变转动方向

实现步进电机的转速控制，电机转动方向和启动截至就简单多了，只需要将另外俩个端口接到IO口：改变使能口的电平，即可控制启动截至；改变转动方向的电平就可以改变转动方向。

# RE40直流有刷电机

## RE40简介

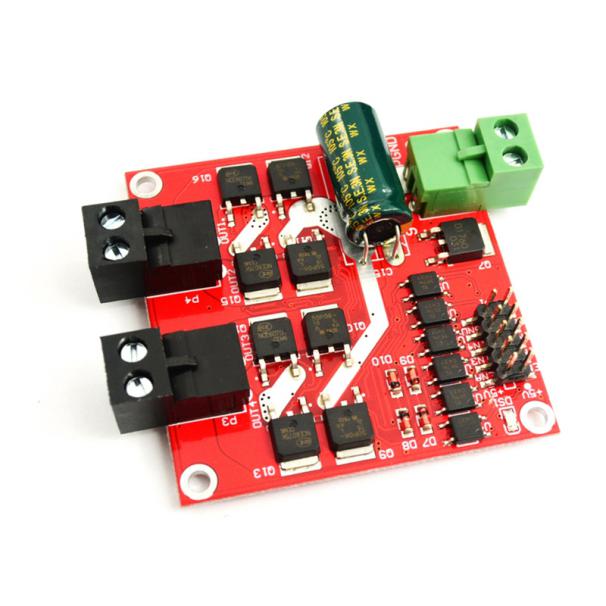


电机型号：24伏 RE40 maxon 148867  
  
减速箱型号：Faulhaber 38/2 12：1  
  
编码器型号：HEDS 5540 A11 500线 输出信号A、B、I  
  
(不带线驱，不是差分输出，加40元可以改装成差分输出，输出信号有A、A-、B、B-、I、I-)  
  
工作电压: 24V  
  
减速后速：620RPM（转每分钟）  
  
电机直径：40mm  
  
减速箱直径：38mm  
  
减速箱类型：全金属精密行星减速箱  
  
减速比： 12：1  
  
出轴直径：8mm  
  
出轴长度：18.5mm  
  
输出功率：150W  
  
连续扭矩：1.8Nm（18Kg·cm）  
  
堵转扭力：超大扭矩28Nm (IMG_256280Kg.cm)  
  
编码器：光电式ABI三通道输出 500线

## 电机特点

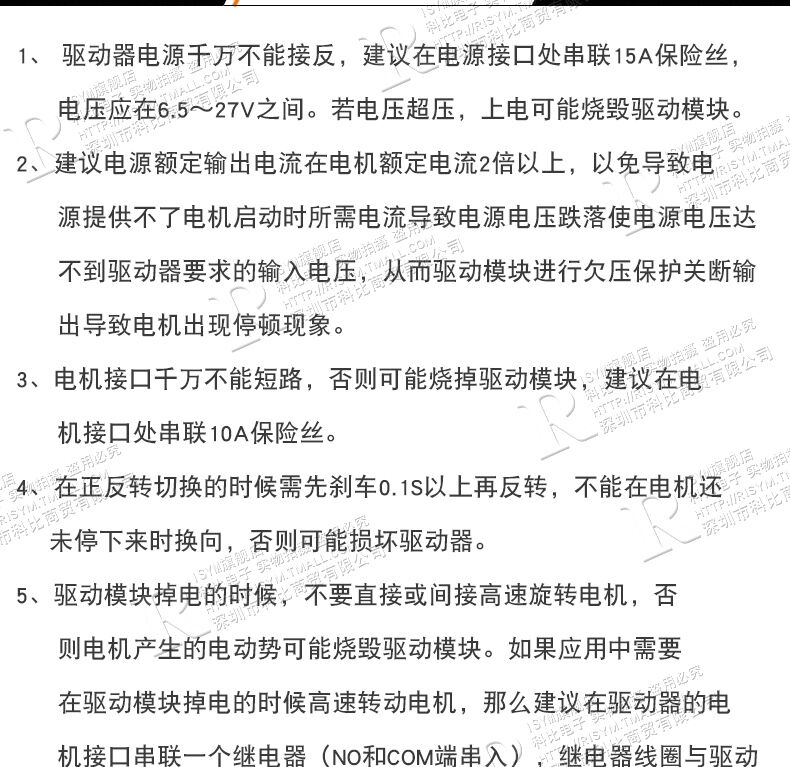
扭矩大、转速稳定、精度高、价格贵

## 大功率驱动



特点：功率大、性能稳定、偏贵

注意事项：



信号逻辑（与普通驱动一致）：



## 电机控制

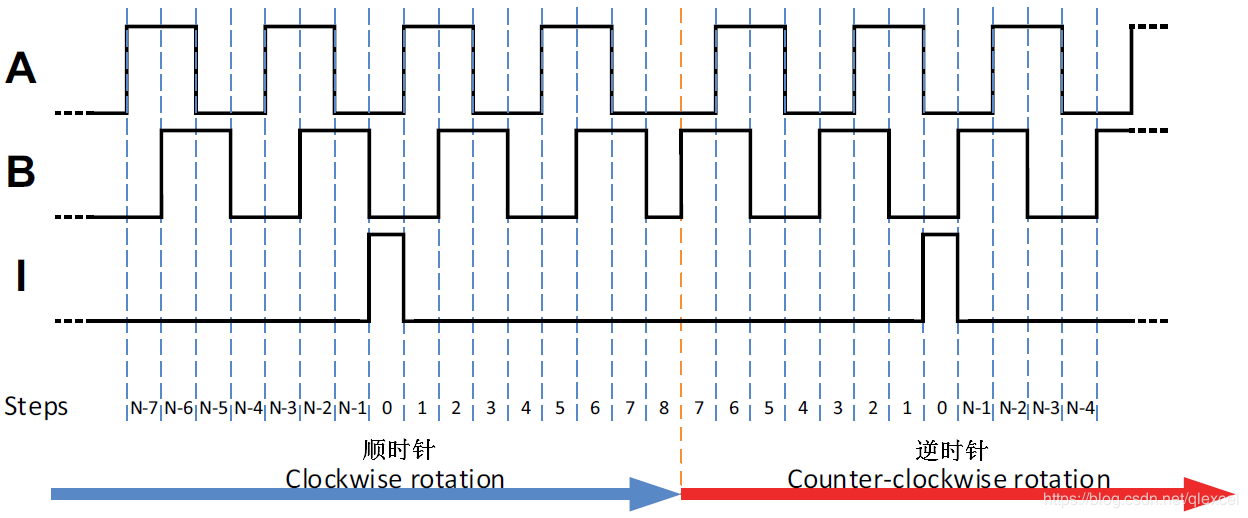
原理：利用STM32 的IO输出不同占空比的脉冲信号来达到调速的过程，当占空比大的时候就表明在一个脉冲周期内高电平的时间越长，而脉冲信号输出频率极快，高电平的时间越长就表明在一段时间内IO口输出的控制电压越高，控制电压越高使得驱动输出的电压越高，这样就使得电机的转速越快。

要求：配置pwm波进行控制，与直流减速电机调速一致。

## 编码器

Re40自带编码器 光电式ABI三通道输出 500线

ABI信号：



如上图，编码器输出三组方波脉冲A、B和I相；A、B两组脉冲相位差90度，根据谁先出现可以方便的判断旋转方向。而Z相为每转一圈输出一个脉冲，用于基准点定位。

编码器可采用 编码器介绍里的编码器模式方法是配置编码器。也可通过理解利用stm32脉冲计数进行配置使用、如需测速只要将一定时间内测出的脉冲数除以时间就可以得出速度，对于编码器的学习很灵活，可以自行突破。

示例：

#define ENCODENU 65536

void time4IOInit()

{  
GPIO\_InitTypeDef        GPIO\_InitStructure;  
RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE); //①使能 GPIOB 时钟  
  
GPIO\_InitStructure.GPIO\_Pin  = GPIO\_Pin\_6 | GPIO\_Pin\_7; //PB6 7 设置  
GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING;//浮空输入   
  
GPIO\_Init(GPIOB, &GPIO\_InitStructure);   
}  
  
  
void time4CountInit()  
{  
  
RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM4, ENABLE);   //使能 TIM4 时钟  
TIM4->ARR = ENCODENU-1;    //设定计数器重装值（因为没有使用更新中断所以将计数器设最大值，保证1s延时不会溢出）     
TIM4->PSC  =0;                            //预分频器  
TIM4->CR1 &=~(3<<8);                    // 选择时钟分频：不分频  
TIM4->CR1 &=~(3<<5);                   //边沿对齐（根据dir位向上或者向下计数）  
        //定时器4配置成编码器模式，双边沿触发  
TIM\_EncoderInterfaceConfig(TIM4, TIM\_EncoderMode\_TI12, TIM\_ICPolarity\_BothEdge ,TIM\_ICPolarity\_BothEdge);  
TIM\_SetCounter(TIM4, 0);                 //计数器清零  
}  
  
void encoderConfig()  
{  
        time4IOInit();  
time4CountInit();  
TIM\_Cmd(TIM4, ENABLE);  //计数器使能，开始工作  
}  
  
void main ()   
{  
int dir=0;  
u16 nu=0;  
encoderConfig();  
for (;;)   
       {  
dir=(TIM4->CR1 & 0x0010)>4;                                       //取方向标志位  
if(dir > 0){                                                           //向下计数  
 nu = (ENCODENU-TIM\_GetCounter(TIM4))/4;    //除以4是因为一对脉冲计数器计数4次  
FG\_Print("down ; TIM\_GetCounter=%d  \r\n",nu\*60/1000);//（\*60是因为延时1s转换成一分钟，/1000是因为编码器转一圈是1000个脉冲）  
}else{                                                            //向上计数  
  nu = TIM\_GetCounter(TIM4)/4;                                
          FG\_Print("up ; TIM\_GetCounter=%d  \r\n",nu\*60/1000);  
}  
  
  
TIM\_SetCounter(TIM4, 0);  
osDelay(1000);  
}  
}

## 突破

MAXON RE40直流有刷电机在Elmo SimplIQ cello系列驱动器的增益调节

CSDN网址：

https://blog.csdn.net/qq\_34935373/article/details/86561909

# 舵机

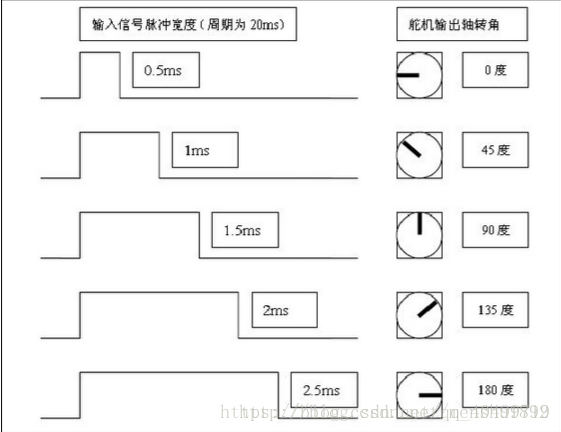
## 一、简介

舵机的主要组成部分为伺服电机，所谓伺服就是服从信号的要求而动作。在信号来之前，转子停止不动；信号来到之后，转子立即运动。因此我们就可以给舵机输入不同的信号,来控制其旋转到不同的角度。

舵机接收的是PWM信号，当信号进入内部电路产生一个偏置电压，触发电机通过减速齿轮带动电位器移动，使电压差为零时，电机停转，从而达到伺服的效果。简单来说就是给舵机一个特定的PWM信号，舵机就可以旋转到指定的位置。

舵机上有三根线，分别是GND、VCC和SIG，也就是地线、电源线和信号线，其中的PWM波就是从信号线输入给舵机的。

一般来说，舵机接收的PWM信号频率为50HZ，即周期为20ms。当高电平的脉宽在0.5ms-2.5ms之间时舵机就可以对应旋转到不同的角度。如下图。



# I2C

## 1、I2C总线简介

I2C是两线式串行总线，用于连接微控制器及其外围设备。I2C总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此I2C总线占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本。总线的长度可高达25英尺，并且能够以10Kbps的最大传输速率支持40个组件。I2C总线的另一个优点是，它支持多主控(multimastering)， 其中任何能够进行发送和接收的设备都可以成为主总线。一个主控能够控制信号的传输和时钟频率。当然，在任何时间点上只能有一个主控。

单片机的通讯模块常用的有UART、SPI、I2C、CAN等等。UART大多用于单片机与PC的通信，CAN总线常用于单片机与单片机通信，而SPI和I2C则用于单片机和外围设备、外围设备和其它外围设备的通信，根据自己的需要来设计通讯方式。

相比较SPI而言，I2C需要更少的接线，仅由数据线SDA和时钟SCL构成 。而SPI则需要四根引线，它们是SDI（数据输入），SDO（数据输出），SCLK（时钟），CS（片选）。但是常常因为I2C的通讯协议较为复杂，不容易在程序中实现而导致数据丢失、无应答、“死等”等问题。

下面我们就来通过I2C的时序来解析一下I2C的通讯。

## 2、I2C总线的时序

I2C总线是由数据线SDA和时钟SCL构成的串行总线，可发送和接收数据。在CPU与被控IC之间、IC与IC之间进行双向传送，最高传送速率100kbps。各种被控制电路均并联在这条总线上，但就像电话机一样只有拨通各自的号码才能工作，所以每个电路和模块都有唯一的地址，在信息的传输过程中，I2C总线上并接的每一模块电路既是主控器（或被控器），又是发送器（或接收器），这取决于它所要完成的功能。CPU发出的控制信号分为地址码和控制量两部分，地址码用来选址，即接通需要控制的电路，确定控制的种类；控制量决定该调整的类别及需要调整的量。这样，各控制电路虽然挂在同一条总线上，却彼此独立，互不相关。

I2C总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL为高电平时，SDA由高电平向低电平跳变，开始传送数据。

结束信号：SCL为高电平时，SDA由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的IC在接收到8bit数据后，向发送数据的IC发出特定的低电平脉冲，表示已收到数据。CPU向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

I2C总线通信过程中出现的几种信号状态和时序分析如下：

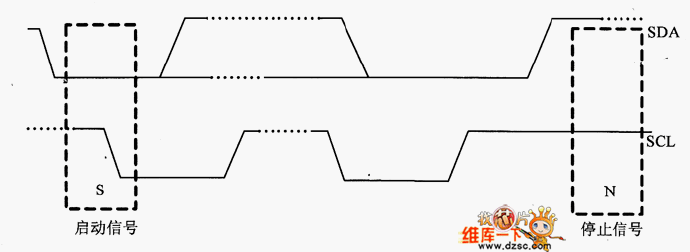
### （1）总线空闲状态

I2C总线总线的SDA和SCL两条信号线同时处于高电平时，规定为总线的空闲状态。此时各个器件的输出级场效应管均处在截止状态，即释放总线，由两条信号线各自的上拉电阻把电平拉高。

### （2）启动信号

在时钟线SCL保持高电平期间，数据线SDA上的电平被拉低（即负跳变），定义为I2C总线总线的启动信号，它标志着一次数据传输的开始。

启动信号是一种电平跳变时序信号，而不是一个电平信号。启动信号是由主控器主动建立的，在建立该信号之前I2C总线必须处于空闲状态，如图所示。



基于stm32的I2C总线通讯简介及使用操作（附参考代码）

### （3）停止信号

在时钟线SCL保持高电平期间，数据线SDA被释放，使得SDA返回高电平（即正跳变），称为I2C总线的停止信号，它标志着一次数据传输的终止。

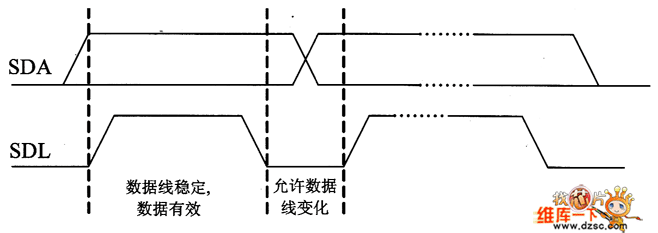
停止信号也是一种电平跳变时序信号，而不是一个电平信号，停止信号也是由主控器主动建立的，建立该信号之后，I2C总线将返回空闲状态。

### （4）数据位传送

在I2C总线上传送的每一位数据都有一个时钟脉冲相对应（或同步控制），即在SCL串行时钟的配合下，在SDA上逐位地串行传送每一位数据。

进行数据传送时，在SCL呈现高电平期间，SDA上的电平必须保持稳定，低电平为数据0，高电平为数据1。

只有在SCL为低电平期间，才允许SDA上的电平改变状态。逻辑0的电平为低电压，而逻辑1的电平取决于器件本身的正电源电压VDD（当使用独立电源时），如图所示。



基于stm32的I2C总线通讯简介及使用操作（附参考代码）

### （5） 应答信号

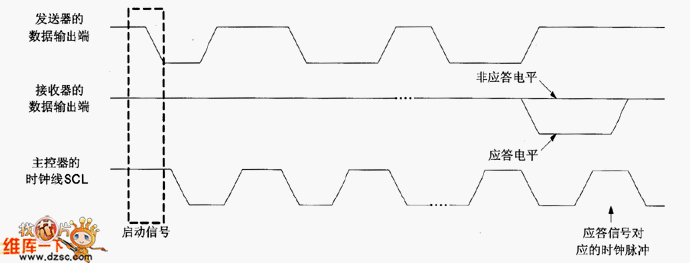
I2C总线上的所有数据都是以8位字节传送的，发送器每发送一个字节，就在时钟脉冲9期间释放数据线，由接收器反馈一个应答信号。

应答信号为低电平时，规定为有效应答位（ACK简称应答位），表示接收器已经成功地接收了该字节；应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功。

对于反馈有效应答位ACK的要求是，接收器在第9个时钟脉冲之前的低电平期间将SDA线拉低，并且确保在该时钟的高电平期间为稳定的低电平。

如果接收器是主控器，则在它收到最后一个字节后，发送一个NACK信号，以通知被控发送器结束数据发送，并释放SDA线，以便主控接收器发送一个停止信号P，如图所示。

基于stm32的I2C总线通讯简介及使用操作（附参考代码）



### （6）插入等待时间

如果被控器需要延迟下一个数据字节开始传送的时间，则可以通过把时钟线SCL电平拉低并且保持，使主控器进入等待状态。

一旦被控器释放时钟线，数据传输就得以继续下去，这样就使得被控器得到足够时间转移已经收到的数据字节，或者准备好即将发送的数据字节。

带有CPU的被控器在对收到的地址字节做出应答之后，需要一定的时间去执行中断服务子程序，来分析或比较地址码，其间就把SCL线钳位在低电平上，直到处理妥当后才释放SCL线，进而使主控器继续后续数据字节的发送，如图所示。

### （7）重启动信号

在主控器控制总线期间完成了一次数据通信（发送或接收）之后，如果想继续占用总线再进行一次数据通信（发送或接收），而又不释放总线，就需要利用重启动Sr信号时序。

重启动信号Sr既作为前一次数据传输的结束，又作为后一次数据传输的开始。利用重启动信号的优点是，在前后两次通信之间主控器不需要释放总线，这样就不会丢失总线的控制权，即不让其他主器件节点抢占总线。

### （8）时钟同步

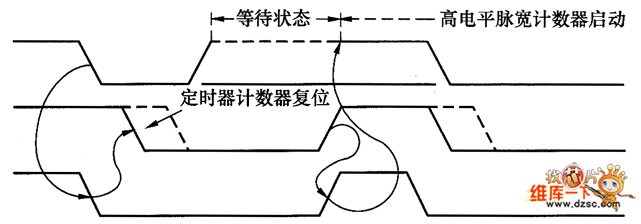
如果在某一I2C总线系统中存在两个主器件节点，分别记为主器件1和主器件2，其时钟输出端分别为CLK1和CLK0，它们都有控制总线的能力。

假设在某一期间两者相继向SCL线发出了波形不同的时钟脉冲序列CLK1和CLK2（时钟脉冲的高、低电平宽度都是依靠各自内部专用计数器定时产生的），在总线控制权还没有裁定之前这种现象是可能出现的。

鉴于I2C总线的“线与”特性，使得时钟线SCL上得到的时钟信号波形，既不像主器件1所期望的CLK1，也不像主器件2所期望的CLK2，而是两者进行逻辑与的结果。

CLKI和CLK2的合成波形作为共同的同步时钟信号，一旦总线控制权裁定给某一主器件，则总线时钟信号将会只由该主器件产生，如图所示。

基于stm32的I2C总线通讯简介及使用操作（附参考代码）



### （9）总线冲突和总线仲裁

假如在某I2C总线系统中存在两个主器件节点，分别记为主器件1和主器件2，其数据输出端分别为DATA1和DATA2，它们都有控制总线的能力，这就存在着发生总线冲突（即写冲突）的可能性。

假设在某一瞬间两者相继向总线发出了启动信号，鉴于：I2C总线的“线与”特性，使得在数据线SDA上得到的信号波形是DATA1和DATA2两者相与的结果，该结果略微超前送出低电平的主器件1，其DATA1的下降沿被当做SDA的下降沿。

在总线被启动后，主器件1企图发送数据“101……”，主器件2企图发送数据“100101……”。

两个主器件在每次发出一个数据位的同时都要对自己输出端的信号电平进行抽检，只要抽检的结果与它们自己预期的电平相符，就会继续占用总线，总线控制权也就得不到裁定结果。

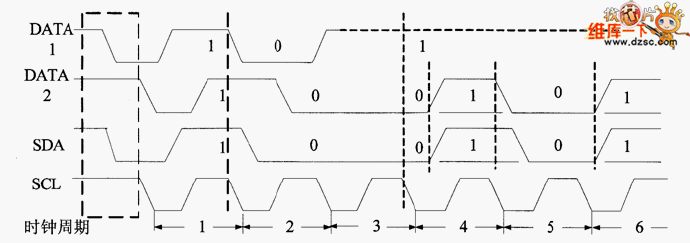
主器件1的第3位期望发送“1”，也就是在第3个时钟周期内送出高电平。

在该时钟周期的高电平期间，主器件1进行例行抽检时，结果检测到一个不相匹配的电平“0”，这时主器件1只好决定放弃总线控制杈；因此，主器件2就成了总线的惟一主宰者，总线控制权也就最终得出了裁定结果，从而实现了总线仲裁的功能。

从以上总线仲裁的完成过程可以得出：仲裁过程主器件1和主器件2都不会丢失数据；各个主器件没有优先级别之分，总线控制权是随机裁定的，即使是抢先发送启动信号的主器件1最终也并没有得到控制杈。

系统实际上遵循的是“低电平优先”的仲裁原则，将总线判给在数据线上先发送低电平的主器件，而其他发送高电平的主器件将失去总线控制权，如图所示。

基于stm32的I2C总线通讯简介及使用操作（附参考代码）



### （10）总线封锁状态。

在特殊情况下，如果需要禁止所有发生在I2C总线上的通信活动，封锁或关闭总线是一种可行途径，只要挂接于该总线上的任意一个器件将时钟线SCL锁定在低电平上即可。

## 3、参考代码

子函数文件 iic.c

#include "myi2c.h"

void I2C\_GPIO\_Config(void)

{

GPIO\_InitTypeDef GPIO\_InitStructure;

GPIO\_InitStructure.GPIO\_Pin = SCL\_PIN;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_OD;

GPIO\_Init(GPIOB, &GPIO\_InitStructure);

GPIO\_InitStructure.GPIO\_Pin = SDA\_PIN;

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_OD;

GPIO\_Init(GPIOB, &GPIO\_InitStructure);

}

void I2C\_delay(void)

{

u8 i=10; //这里可以优化速度 ，经测试最低到5还能写入

while(i)

{

i--;

}

}

void delay5ms(void)

{

int i=5000;

while(i)

{

i--;

}

}

uint16\_t I2C\_Start(void)

{

SDA\_H;

SCL\_H;

I2C\_delay();

if(!SDA\_read)return FALSE; //SDA线为低电平则总线忙,退出

SDA\_L;

I2C\_delay();

if(SDA\_read) return FALSE; //SDA线为高电平则总线出错,退出

SDA\_L;

I2C\_delay();

return TRUE;

}

void I2C\_Stop(void)

{

SCL\_L;

I2C\_delay();

SDA\_L;

I2C\_delay();

SCL\_H;

I2C\_delay();

SDA\_H;

I2C\_delay();

}

void I2C\_Ack(void)

{

SCL\_L;

I2C\_delay();

SDA\_L;

I2C\_delay();

SCL\_H;

I2C\_delay();

SCL\_L;

I2C\_delay();

}

void I2C\_NoAck(void)

{

SCL\_L;

I2C\_delay();

SDA\_H;

I2C\_delay();

SCL\_H;

I2C\_delay();

SCL\_L;

I2C\_delay();

}

uint16\_t I2C\_WaitAck(void) //返回为:=1有ACK,=0无ACK

{

SCL\_L;

I2C\_delay();

SDA\_H;

I2C\_delay();

SCL\_H;

I2C\_delay();

if(SDA\_read)

{

SCL\_L;

I2C\_delay();

return FALSE;

}

SCL\_L;

I2C\_delay();

return TRUE;

}

void I2C\_SendByte(unsigned char SendByte) //数据从高位到低位//

{

u8 i=8;

while(i--)

{

SCL\_L;

I2C\_delay();

if(SendByte&0x80)

SDA\_H;

else

SDA\_L;

SendByte<<=1;

I2C\_delay();

SCL\_H;

I2C\_delay();

}

SCL\_L;

}

unsigned char I2C\_RadeByte(void) //数据从高位到低位//

{

u8 i=8;

u8 ReceiveByte=0;

SDA\_H;

while(i--)

{

ReceiveByte<<=1;

SCL\_L;

I2C\_delay();

SCL\_H;

I2C\_delay();

if(SDA\_read)

{

ReceiveByte|=0x01;

}

}

SCL\_L;

return ReceiveByte;

}

//ZRX

//单字节写入\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

uint16\_t Single\_Write(unsigned char SlaveAddress,unsigned char REG\_Address,unsigned char REG\_data) //void

{

if(!I2C\_Start())return FALSE;

I2C\_SendByte(SlaveAddress); //发送设备地址+写信号//I2C\_SendByte(((REG\_Address & 0x0700) >>7) | SlaveAddress & 0xFFFE);//设置高起始地址+器件地址

if(!I2C\_WaitAck()){I2C\_Stop(); return FALSE;}

I2C\_SendByte(REG\_Address ); //设置低起始地址

I2C\_WaitAck();

I2C\_SendByte(REG\_data);

I2C\_WaitAck();

I2C\_Stop();

delay5ms();

return TRUE;

}

//单字节读取\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

unsigned char Single\_Read(unsigned char SlaveAddress,unsigned char REG\_Address)

{ unsigned char REG\_data;

if(!I2C\_Start())return FALSE;

I2C\_SendByte(SlaveAddress); //I2C\_SendByte(((REG\_Address & 0x0700) >>7) | REG\_Address & 0xFFFE);//设置高起始地址+器件地址

if(!I2C\_WaitAck()){I2C\_Stop();return FALSE;}

I2C\_SendByte((u8) REG\_Address); //设置低起始地址

I2C\_WaitAck();

I2C\_Start();

I2C\_SendByte(SlaveAddress+1);

I2C\_WaitAck();

REG\_data= I2C\_RadeByte();

I2C\_NoAck();

I2C\_Stop();

//return TRUE;

return REG\_data;

}

头文件 iic.h

#ifndef \_\_MYI2C\_H\_\_

#define \_\_MYI2C\_H\_\_

#include "stm32f10x.h"

#define FALSE 0

#define TRUE 1

#define SCL\_PIN GPIO\_Pin\_6

#define SDA\_PIN GPIO\_Pin\_7

#define SCL\_H GPIOB->BSRR = SCL\_PIN

#define SCL\_L GPIOB->BRR = SCL\_PIN

#define SDA\_H GPIOB->BSRR = SDA\_PIN

#define SDA\_L GPIOB->BRR = SDA\_PIN

#define SCL\_read GPIOB->IDR & SCL\_PIN

#define SDA\_read GPIOB->IDR & SDA\_PIN

void I2C\_GPIO\_Config(void);

void I2C\_delay(void);

void delay5ms(void);

uint16\_t I2C\_Start(void);

void I2C\_Stop(void);

void I2C\_Ack(void);

void I2C\_NoAck(void);

uint16\_t I2C\_WaitAck(void);

void I2C\_SendByte(unsigned char SendByte);

unsigned char I2C\_RadeByte(void);

uint16\_t Single\_Write(unsigned char SlaveAddress,unsigned char REG\_Address,unsigned char REG\_data);

unsigned char Single\_Read(unsigned char SlaveAddress,unsigned char REG\_Address);

#endif // \_\_MYI2C\_H\_\_

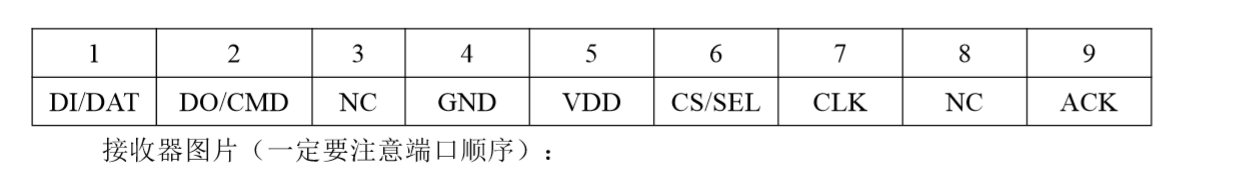
# PS2手柄的控制

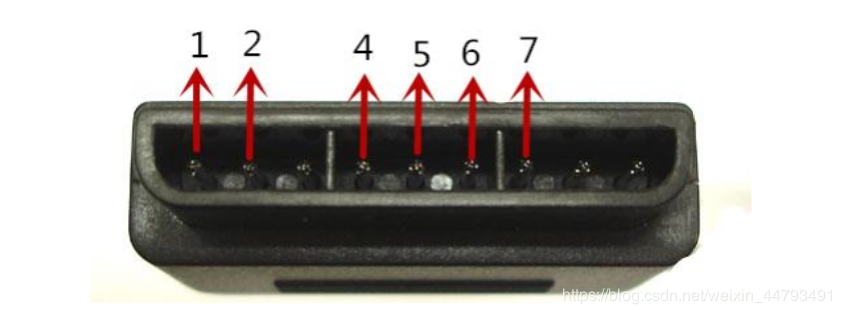
## 一.PS2介绍

PS2手柄是日本SONY公司的PlayStation2 游戏机的遥控手柄。索尼的 PSX系列游戏主机在全球都很畅销。不知什么时候便有人打起 PS2手柄的主意，破解了通讯协议，使得手柄可以接在其他器件上遥控使用，比如遥控我们熟悉的机器人。突出的特点是这款手柄性价比极高，按键丰富，方便扩展到其它应用中。

## 二.PS2通讯协议介绍

PS2采用的是SPI通信协议，SPI是串行外设接口（Serial Peripheral Interface）的缩写，是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线（DI、DO、CS、CLK），节约了芯片的管脚，同时为PCB的布局上节省空间。



PS2接收器上一共有九根引脚，按上图从左往右，依次为：

1.DI/DAT：信号流向，从手柄到主机，此信号是一个8bit 的串行数据，同步传送于时钟的下降沿。信号的读取在时钟由高到低的变化过程中完成。

2.DO/CMD：信号流向，从主机到手柄，此信号和 DI相对，信号是一个 8bit 的串行数据， 同步传送于时钟的下降沿。

3.NC：空端口。

4.GND：电源地。

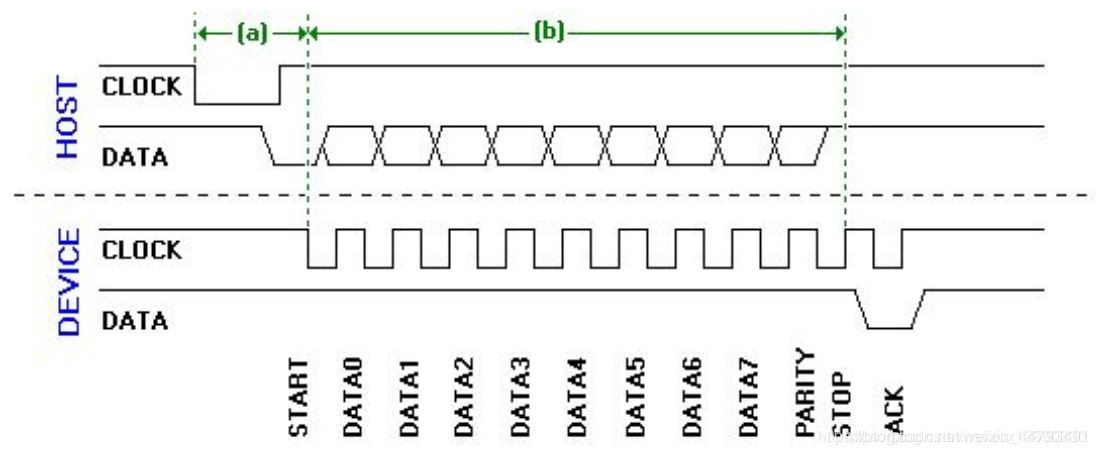
5.VCC：接收器工作电源，电源范围 3~5V。

6.CS/SEL：用于提供手柄触发信号。在通讯期间，处于低电平。

7.CLK：时钟信号，由主机发出，用于保持数据同步。

8.NC：空端口。

9.ACK：从手柄到主机的应答信号。此信号在每个8bits数据发送的最后一个周期变低并且CS一直保持低电平，如果CS信号不变低，约60微秒PS主机会试另一个外设。在编程时未使用ACK端口。（可以忽略）



注意的是：

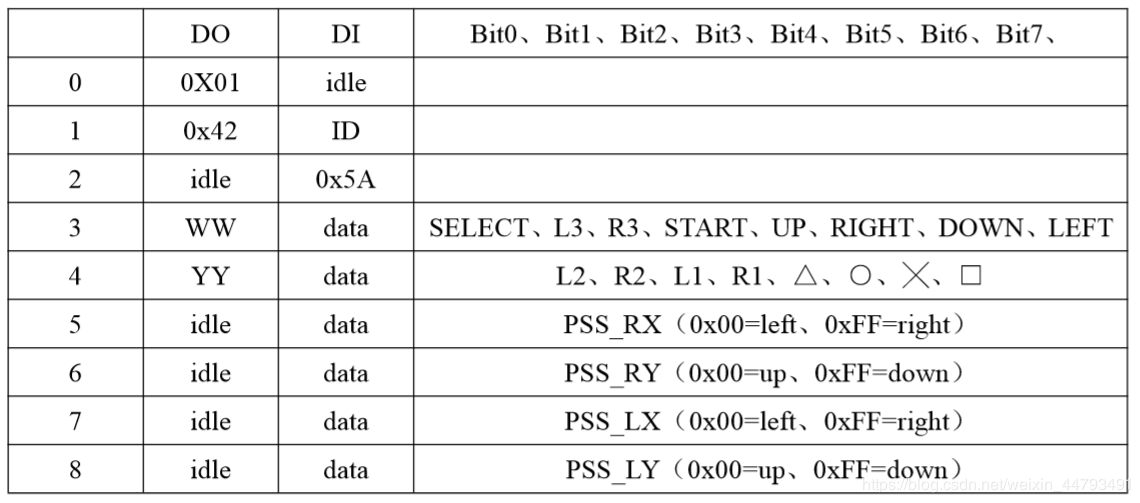
1. CS线在通讯期间拉低，通信过程中CS信号线在一串数据（9个字节,每个字节为8位）发送完毕后才会拉高，而不是每个字节发送完拉高。

2. DO、DI在在CLK时钟的下降沿完成数据的发送和读取。

下降沿：数字电平从高电平（数字“1”）变为低电平（数字“0”）的那一瞬间叫作下降沿。

4. CLK的每个周期为12us。若在某个时刻，CLK处于下降沿，若此时DO为高电平则取“1”，低电平则取“0”。连续读8次则得到一个字节byte的数据，连续读9个字节就能得到一次传输周期所需要的数据。DI也是一样的，发送和传输同时进行。

### 具体的通讯过程如下：



以STM32为例：

首先STM32拉低CS片选信号线，然后在每个CLK的下降沿读一个bit，每读八个bit（即一个byte）CLK拉高一小段时间，一共读九组bit。

第一个byte是STM32发给接收器命令“0X01” 。

PS2手柄会在第二个byte回复它的ID（0x41=绿灯模式，0x73=红灯模式），同时第二个byte时STM32发给PS2一个0x42请求数据。

红灯模式时：左右摇杆发送模拟值，0x00~0xFF 之间，且摇杆按下的 键值 L3 、 R3 有效；

绿灯模式时：左右摇杆模拟值为无效，推到极限时，对应发送 UP、RIGHT、DOWN、 LEFT、△、○、╳、□，按键 L3 、 R3 无效。

第三个byte PS2 会给主机发送 “0x5A” 告诉STM32数据来了。

从第四个byte开始全是接收器给主机发送数据，每个byte定义如上图，当有按键按下，对应位为“0 ”，例如当键“SELECT”被按下时， Data[3]=11111110。

对于整个通讯过程，你理解成下面的一段对话：

拉低CS，表示开始数据通信

byte 0 :

STM32（DO） : 0x01 ------------------------- [现在开始通信]

PS2手柄（DI） : 空 ---------------------------- [空]

byte 1 ：

STM32（DO） : 0x42 -------------------------- [请求发送数据]

PS2手柄（DI） : 红灯0x73

            绿灯0X41---------------------[现在的ID]

byte 2：

STM32（DO） : 空 ------------------------------ [空]

PS2手柄（DI） : 0X5A ------------------------- [数据来了]

byte 3：

STM32（DO） : 0X00~0XFF ------------------ [右侧小震动电机是否开启]

PS2手柄（DI） : 00000000~11111111 ------- [SELECT、 L3 、 R3、 START 、 UP、 RIGHT、 DOWN、 LEFT 是否被按下，若被按下对应位为0]

byte 4：

STM32（DO） : 0X00~0XFF ------------------ [左侧大震动电机振动幅度]

PS2手柄（DI） : 00000000~11111111 ------- [L2 、 R2、L1 、R1、△、○、╳、□ 是否被按下，若被按下对应位为0]

byte 5：

STM32（DO） : 空 -------------------------------- [空]

PS2手柄（DI） : 0X00~0XFF ------------------ [左侧X轴摇杆模拟量]

byte 6：

STM32（DO） : 空 -------------------------------- [空]

PS2手柄（DI） : 0X00~0XFF ------------------ [左侧Y轴摇杆模拟量]

byte 7：

STM32（DO） : 空 -------------------------------- [空]

PS2手柄（DI） : 0X00~0XFF ------------------ [右侧X轴摇杆模拟量]

byte 8：

STM32（DO） : 空 -------------------------------- [空]

PS2手柄（DI） : 0X00~0XFF ------------------ [右侧Y轴摇杆模拟量]

注意：模拟量只对红灯模式下有效，绿灯模式下摇杆推至极限分别对应 UP、RIGHT、DOWN、 LEFT、△、○、╳、□ 。L3、R3只对红灯模式下有效，在绿灯模式下无效。

在手柄通信前还需要一系列的初始化（是否启动振动电机、是否进行锁存等），详情可以参考下面代码。当然，不进行初始化也是可以的，手柄会默认之前的配置。

——————————————————————————————

## 基于STM32的PS2通信源码

下面的程序代码可供参考，但最好还是对应机器人上实际应用代码进行理解。

/\*DI->PB12；

DO->PB13；

CS->PB14；

CLK->PB15

\*/

void PS2\_Init(void) {

// 输入 DI->PB12

RCC->APB2ENR|=1<<3; // 使能 PORTB 时钟

GPIOB->CRH&=0XFFF0FFFF;//PB12 设置成输入 默认下拉

GPIOB->CRH|=0X00080000;

// DO->PB13 CS->PB14 CLK->PB15

RCC->APB2ENR|=1<<3; // 使能 PORTB 时钟

GPIOB->CRH&=0X000FFFFF;

GPIOB->CRH|=0X33300000; //PB13、 PB14 、 PB15 推挽输出

}

//端口初始化，PB12 为输入，PB13 、PB14 、PB15 为输出。

// 向手柄发送命令

void PS2\_Cmd(u8CMD)

{

volatile u16 ref=0x01;

Data[1]=0;

for(ref=0x01;ref<0x0100;ref<<=1)

{

if(ref&CMD)

{

DO\_H; // 输出一位控制位

}

else DO\_L;

CLK\_H; // 时钟拉高

delay\_us(10);

CLK\_L;

delay\_us(10);

CLK\_H;

if(DI)

{Data[1]=ref|Data[1];}

}

delay\_us(16);

}

// 判断是否为红灯模式，0x41=模拟绿灯，0x73=模拟红灯

// 返回值；0，红灯模式

// 其他，其他模式

u8PS2\_RedLight(void)

{

CS\_L;

PS2\_Cmd(Comd[0]); // 开始命令

PS2\_Cmd(Comd[1]); // 请求数据

CS\_H;

if( Data[1]== 0X73) return 0；

else return 1；

}

// 读取手柄数据

void PS2\_ReadData(void)

{

volatile u8 byte=0;

volatile u16 ref=0x01;

CS\_L;

PS2\_Cmd(Comd[0]); // 开始命令

PS2\_Cmd(Comd[1]); // 请求数据

for(byte=2;byte<9;byte++) // 开始接受数据

{

for(ref=0x01;ref<0x100;ref<<=1)

{

CLK\_H;

delay\_us(10);

CLK\_L;

delay\_us(10);

CLK\_H;

if(DI)

{Data[byte]= ref|Data[byte];}

}

delay\_us(16);

}

CS\_H;

}

/\*

上面两个函数分别为主机向手柄发送数据、手柄向主机发送数据。手柄向主机发送的数据缓存在数组 Data[]中，

数组中共有9个元素，每个元素的意义请见表1。

还有一个函数是用来判断手柄的发送模式，也就是判断 ID(红灯还是绿灯模式) 即 Data[1]的值。

\*/

// 对读出来的 PS2 的数据进行处理,只处理按键部分

//按下为0，未按下为1

u8PS2\_DataKey()

{

u8 index;

PS2\_ClearData();

PS2\_ReadData();

Handkey=(Data[4]<<8)|Data[3]; // 这是 16个按键 按下为 0 ， 未按下为 1

for(index=0;index<16;index++)

{

if((Handkey&(1<<(MASK[index]-1)))==0)

returnindex+1;

}

return 0; // 没有任何按键按下

}

// 得到一个摇杆的模拟量 范围 0~256

u8PS2\_AnologData(u8 button)

{

return Data[button];

}

// 清除数据缓冲区

void PS2\_ClearData()

{

u8 a;

for(a=0;a<9;a++)

{Data[a]=0x00;}

}

/\*

8 位数 Data[3]与 Data[4]，分别对应着 16个按键的状态，按下为 0，未按下为 1。

通过 对这两个数的处理，得到按键状态并返回键值。

另一个函数的功能就是返回模拟值，只有在“红灯模式”下值才是有效的，拨动摇杆， 值才会变化，这些值分别存储在 Data[5]、Data[6]、

Data[7]、 Data[8]。

\*/

//手柄配置初始化：

void PS2\_ShortPoll(void)

{

CS\_L;

delay\_us(16);

PS2\_Cmd(0x01);

PS2\_Cmd(0x42);

PS2\_Cmd(0X00);

PS2\_Cmd(0x00);

PS2\_Cmd(0x00);

CS\_H;

delay\_us(16);

}

//进入配置

void PS2\_EnterConfing(void)

{

delay\_us(16);

PS2\_Cmd(0x01);

PS2\_Cmd(0x43);

PS2\_Cmd(0X00);

PS2\_Cmd(0x01);

PS2\_Cmd(0x00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

CS\_H;

delay\_us(16);

}

// 发送模式设置

void PS2\_TurnOnAnalogMode(void)

{

CS\_L;

PS2\_Cmd(0x01);

PS2\_Cmd(0x44);

PS2\_Cmd(0X00);

PS2\_Cmd(0x01);//analog=0x01;digital=0x00 软件设置发送模式

PS2\_Cmd(0xEE);//Ox03 锁存设置，即不可通过按键“MODE ”设置模式。 //0xEE 不锁存软件设置，可通过按键“MODE ”设置模式。

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

CS\_H;

delay\_us(16);

}

// 振动设置

void PS2\_VibrationMode(void)

{

CS\_L;

delay\_us(16);

PS2\_Cmd(0x01);

PS2\_Cmd(0x4D);

PS2\_Cmd(0X00);

PS2\_Cmd(0x00);

PS2\_Cmd(0X01);

CS\_H;

delay\_us(16);

}

// 完成并保存配置

void PS2\_ExitConfing(void)

{

CS\_L;

delay\_us(16);

PS2\_Cmd(0x01);

PS2\_Cmd(0x43);

PS2\_Cmd(0X00);

PS2\_Cmd(0x00);

PS2\_Cmd(0x5A);

PS2\_Cmd(0x5A);

PS2\_Cmd(0x5A);

PS2\_Cmd(0x5A);

PS2\_Cmd(0x5A);

CS\_H;

delay\_us(16);

}

// 手柄配置初始化

void PS2\_SetInit(void)

{

PS2\_ShortPoll();

PS2\_ShortPoll();

PS2\_ShortPoll();

PS2\_EnterConfing(); // 进入配置模式

PS2\_TurnOnAnalogMode(); // “红绿灯”配置模式，并选择是否保存

PS2\_VibrationMode(); // 开启震动模式

PS2\_ExitConfing(); // 完成并保存配置

}

/\*

可以看出配置函数就是发送命令，发送这些命令后，手柄就会明白自己要做什么了，发送命令时，不需要考虑手柄发来的信息。

手柄配置初始化，PS2\_ShortPoll()被执行了3次，主要是为了建立和恢复连接。

具体的配置方式请看注释。

\*/

void PS2\_Vibration(u8motor1,u8motor2)

{

CS\_L;

delay\_us(16);

PS2\_Cmd(0x01); // 开始命令

PS2\_Cmd(0x42);// 请求数据

PS2\_Cmd(0X00);

PS2\_Cmd(motor1);

PS2\_Cmd(motor2);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

PS2\_Cmd(0X00);

CS\_H;

delay\_us(16);

}

//只 有 在 初 始 化 函 数 void PS2\_SetInit(void) 中 ， 对 震 动 电 机 进 行 了

初 始 化 （PS2\_VibrationMode();//开启震动模式），这个函数命令才会被执行。

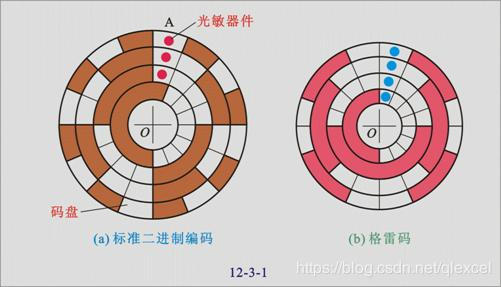
# 编码器

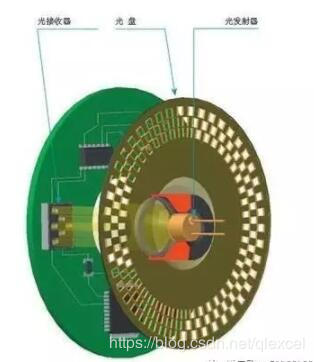
一、编码器的分类

根据检测原理，编码器可分为光学式、磁式、感应式和电容式，根据其刻度方法及信号输出形式，可分为增量式、绝对式以及混合式三种。

1、增量式编码器 增量式编码器是直接利用光电转换原理输出三组方波脉冲A、B和Z相；A、B两组脉冲相位差90。，从而可方便的判断出旋转方向，而Z相为每转一个脉冲，用于基准点定位。它的优点是原理构造简单，机械平均寿命可在几万小时以上，抗干扰能力强，可靠性高，适合于长距离传输。其缺点是无法输出轴转动的绝对位置信息。

2、绝对式编码器 绝对式编码器是直接输出数字的传感器，在它的圆形码盘上沿径向有若干同心码盘，每条道上有透光和不透光的扇形区相间组成，相邻码道的扇区数目是双倍关系，码盘上的码道数是它的二进制数码的位数，在码盘的一侧是光源，另一侧对应每一码道有一光敏元件，当码盘处于不同位置时，各光敏元件根据受光照与否转换出相应的电平信号，形成二进制数。这种编码器的特点是不要计数器，在转轴的任意位置都可读书一个固定的与位置相对应的数字码。显然，码道数越多精度越大。目前国内已有16位的绝对编码器产品。

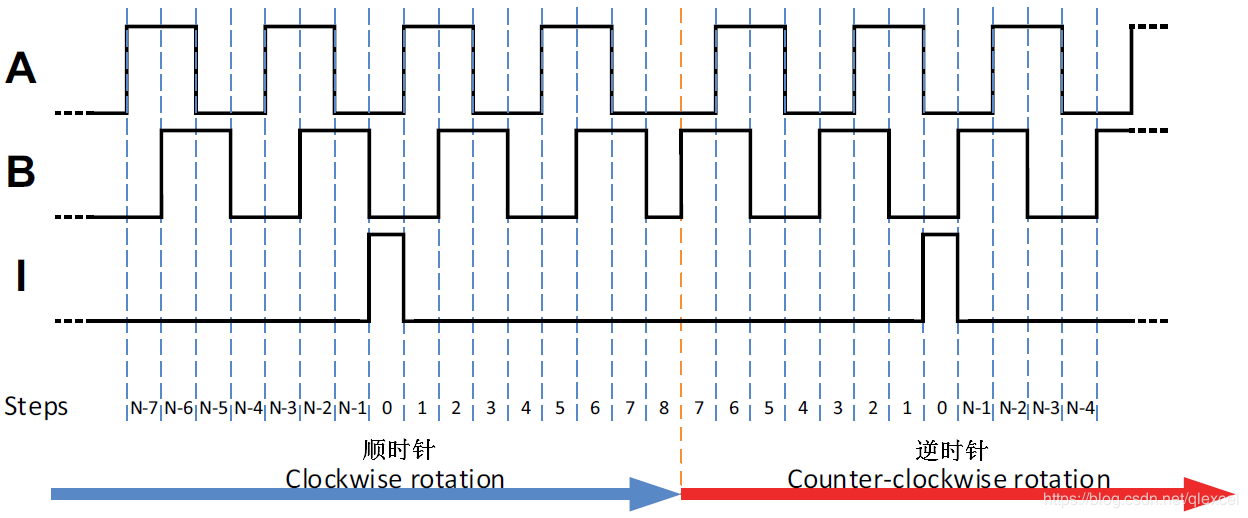




3、混合式绝对编码器 混合式绝对编码器，它输出两组信息，一组信息用于检测磁极位置，带有绝对信息功能；另一组则完全同增量式编码器的输出信息。

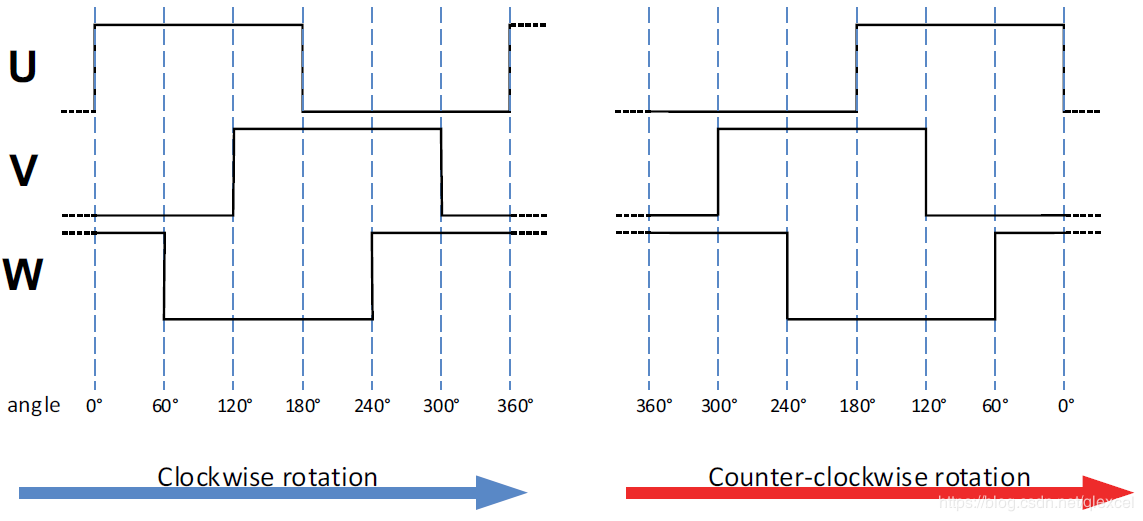
二、ABI（ABZ）信号和UVW信号

1、ABI信号



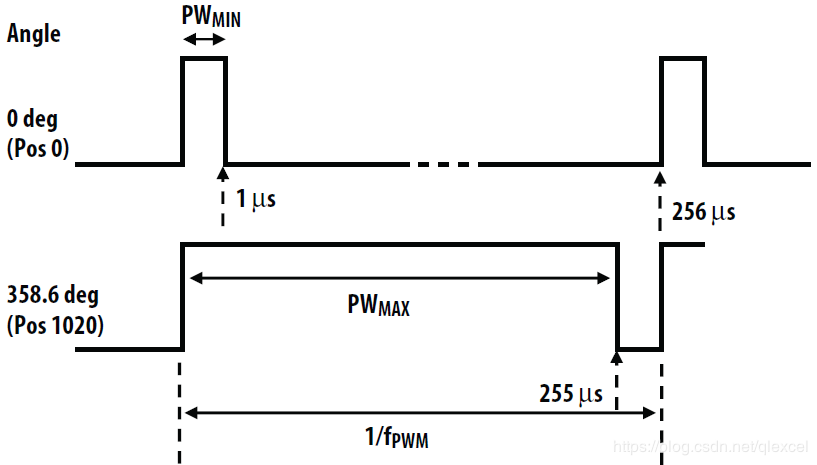
如上图，编码器输出三组方波脉冲A、B和I相；A、B两组脉冲相位差90度，根据谁先出现可以方便的判断旋转方向。而Z相为每转一圈输出一个脉冲，用于基准点定位。

2、UVW信号



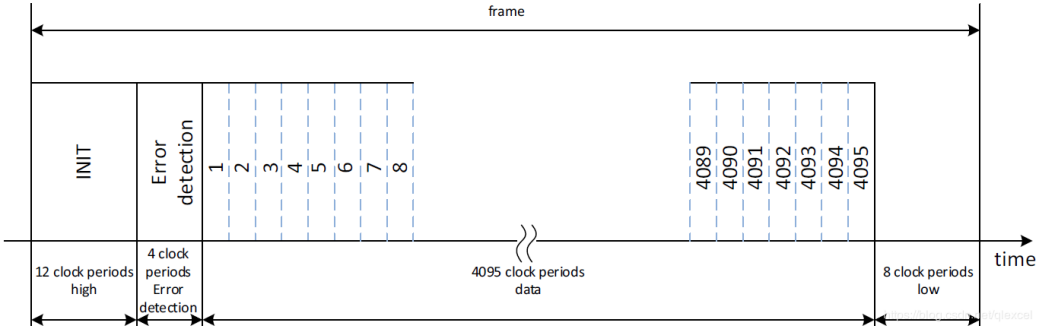
编码器还可以输出UVW信号来表示电机的位置变化，主要用于无刷直流电机的驱动。

3、PWM信号



编码器还可以输出PWM信号，改变PWM的占空比来表示编码器的绝对位置。

还可以在PWM信号上加上通信信息，传递给单片机：



如上图，把PWM的一个周期分为4119份，12份用来表示开始，4份用来传递错误信息，4095份用来传递位置信息，8份用来表示结束。

# 基于蓝牙的串口通信

自行参考正点原子教程

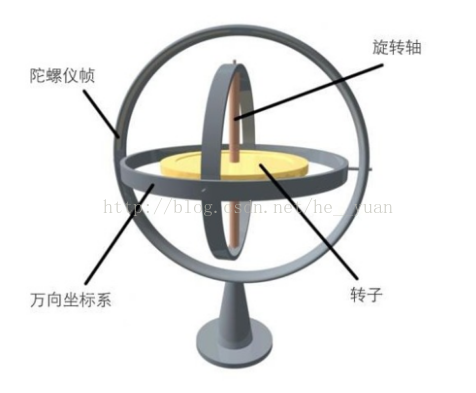
# 陀螺仪MPU6050

## 一·简介:

1.要想知道MPU6050工作原理，得先了解下面俩个传感器：

### ①陀螺仪传感器：

       陀螺仪的原理就是，一个旋转物体的旋转轴所指的方向在不受外力影响时，是不会改变的。人们根据这个道理，用它来保持方向。然后用多种方法读取轴所指示的方向，并自动将数据信号传给控制系统。我们骑自行车其实也是利用了这个原理。轮子转得越快越不容易倒，因为车轴有一股保持水平的力量。现代陀螺仪可以精确地确定运动物体的方位的仪器，它在现代航空，航海，航天和国防工业中广泛使用的一种惯性导航仪器。传统的惯性陀螺仪主要部分有机械式的陀螺仪，而机械式的陀螺仪对工艺结构的要求很高。70年代提出了现代光纤陀螺仪的基本设想，到八十年代以后，光纤陀螺仪就得到了非常迅速的发展，激光谐振陀螺仪也有了很大的发展。光纤陀螺仪具有结构紧凑，灵敏度高，工作可靠。光纤陀螺仪在很多的领域已经完全取代了机械式的传统的陀螺仪，成为现代导航仪器中的关键部件。光纤陀螺仪同时发展的除了环式激光陀螺仪外。



### ②加速度传感器：

      加速度传感器是一种能够测量加速度的传感器。通常由质量块、阻尼器、弹性元件、敏感元件和适调电路等部分组成。传感器在加速过程中，通过对质量块所受惯性力的测量，利用牛顿第二定律获得加速度值。根据传感器敏感元件的不同，常见的加速度传感器包括电容式、电感式、应变式、压阻式、压电式等。

其实说简单点，在mpu6050中我们用陀螺仪传感器测角度，用加速度传感器测加速度

MPU-60X0 :

      MPU-60X0是全球首例9轴运动处理传感器。它集成了3轴MEMS陀螺仪， 3轴MEMS 加速度计，以及一个可扩展的数字运动处理器 DMP（DigitalMotion Processor），可用 I2C 接口连接一个第三方的数字传感器，比如磁力计。扩展之后就可以通过其 I2C 或 SPI 接口 输出一个 9 轴的信号（SPI 接口仅在 MPU-6000 可用）。MPU-60X0 也可以通过其 I2C 接口 连接非惯性的数字传感器，比如压力传感器。 MPU-60X0 对陀螺仪和加速度计分别用了三个 16 位的 ADC，将其测量的模拟量转化 为可输出的数字量。为了精确跟踪快速和慢速的运动，传感器的测量范围都是用户可控的， 陀螺仪可测范围为±250，±500，±1000，±2000°/秒（dps），加速度计可测范围为±2，±4， ±8，±16g。 一个片上 1024 字节的 FIFO，有助于降低系统功耗。 和所有设备寄存器之间的通信采用 400kHz 的 I2C 接口或 1MHz 的 SPI 接口（SPI 仅 MPU-6000 可用）。对于需要高速传输的应用，对寄存器的读取和中断可用 20MHz 的 SPI。 另外，片上还内嵌了一个温度传感器和在工作环境下仅有±1%变动的振荡器。 芯片尺寸 4×4×0.9mm，采用 QFN 封装（无引线方形封装），可承受最大 10000g 的冲 击，并有可编程的低通滤波器。 关于电源，MPU-60X0 可支持 VDD 范围 2.5V±5%，3.0V±5%，或 3.3V±5%。另外 MPU-6050 还有一个 VLOGIC 引脚，用来为 I2C 输出提供逻辑电平。VLOGIC 电压可取 1.8±5%或者 VDD。

         数字运动处理器（DMP）:

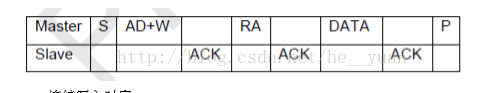
 DMP 从陀螺仪、加速度计以及外接的传感器接收并处理数据，处理结果可以从 DMP 寄存器读出，或通过 FIFO 缓冲。DMP 有权使用 MPU 的一个外部引脚产生中断。

## 二·数据传输:

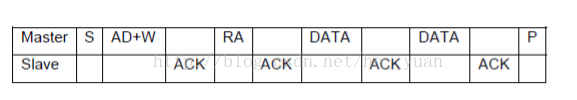
1.I2C原理，见I2C篇

如果要写 MPU-60X0 寄存器，主设备除了发出开始标志（S）和地址位，还要加一个 R/W 位，0 为写，1 为读。在第 9 个时钟周期（高电平时），MPU-60X0 产生应答信号。然 后主设备开始传送寄存器地址（RA），接到应答后，开始传送寄存器数据，然后仍然要有应 答信号，依次类推。

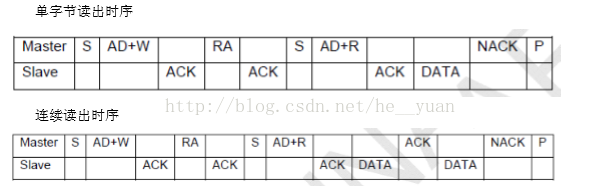
单字节写时序：



多字节写时序：



如果要读取 MPU-60X0 寄存器的值，首先由主设备产生开始信号（S），然后发送从设 备地址位和一个写数据位，然后发送寄存器地址，才能开始读寄存器。紧接着，收到应答信 号后，主设备再发一个开始信号，然后发送从设备地址位和一个读数据位。然后，作为从设 备的 MPU-60X0 产生应答信号并开始发送寄存器数据。通信以主设备产生的拒绝应答信号 （NACK）和结束标志（P）结束。拒绝应答信号（NACK）产生定义为 SDA 数据在第 9 个 时钟周期一直为高。



三·STM32控制MPU6050

1.硬件连接

实验采用正点原子公司的 AN1507 ATK-MPU6050 六轴传感器模块（直接用开发板的话，可以直接连接开发板上的排插位置）

MPU6050 STM32

VCC <---> VCC

GND <---> GND

SDA <---> PB9

SCL <---> PB8

INT <---> 不接

AD0 <---> 不接

以下为网上找到一些详细配置过程感兴趣的可以看一下。

其实如果我们用的话，直接参考例程，可以不看这些直接调用陀螺仪的测出的几个值：pitch、yaw、roll。

但是一定要清楚其中用到哪些串口、哪些定时器。

2. 重要寄存器

2.1 电源管理寄存器 1

DEVICE\_RESET 位用来控制复位，设置为 1，复位 MPU6050，复位结束后， MPU

硬件自动清零该位

SLEEEP 位用于控制 MPU6050 的工作模式，复位后，该位为 1，即进

入了睡眠模式（低功耗），所以我们要清零该位，以进入正常工作模式

TEMP\_DIS 用于设置是否使能温度传感器，设置为 0，则使能

CLKSEL[2:0]用于选择系统时钟源，选择关系如表

CLKSEL[2:0] 时钟源

000 内部 8M RC 晶振

001 PLL，使用 X 轴陀螺作为参考

010 PLL，使用 Y 轴陀螺作为参考

011 PLL，使用 Z 轴陀螺作为参考

100 PLL，使用外部 32.768Khz 作为参考

101 PLL，使用外部 19.2Mhz 作为参考

110 保留

111 关闭时钟，保持时序产生电路复位状态

\*\*默认是使用内部 8M RC 晶振的，精度不高，所以我们一般选择 X/Y/Z 轴陀螺作为参考

的 PLL 作为时钟源，一般设置 CLKSEL=001 即可\*\*

2.2 陀螺仪配置寄存器

FS\_SEL[1:0]这两个位，用于设置陀螺仪的满量程范围： 0，±250°

/S； 1，±500° /S； 2，±1000° /S； 3，±2000° /S；我们一般设置为 3，即±2000° /S，因

为陀螺仪的 ADC 为 16 位分辨率，所以得到灵敏度为： 65536/4000=16.4LSB/(° /S)

2.3 加速度传感器配置寄存器

AFS\_SEL[1:0]这两个位，用于设置加速度传感器的满量程范围： 0，

±2g； 1，±4g； 2，±8g； 3，±16g；我们一般设置为 0，即±2g，因为加速度传感器的

ADC 也是 16 位，所以得到灵敏度为： 65536/4=16384LSB/g

2.4 FIFO使能寄存器

该寄存器用于控制 FIFO 使能，在简单读取传感器数据的时候，可以不用 FIFO，设置

对应位为 0 即可禁止 FIFO，设置为 1，则使能 FIFO

加速度传感器的 3 个轴，全由 1

个位（ ACCEL\_FIFO\_EN）控制，只要该位置 1，则加速度传感器的三个通道都开启 FIFO

2.5 陀螺仪采样率分频寄存器

该寄存器用于设置 MPU6050 的陀螺仪采样频率，计算公式为：

采样频率 = 陀螺仪输出频率 / (1+SMPLRT\_DIV)

这里陀螺仪的输出频率，是 1Khz 或者 8Khz，与数字低通滤波器（ DLPF）的设置有关，

当 DLPF\_CFG=0/7 的时候，频率为 8Khz，其他情况是 1Khz。而且 DLPF 滤波频率一般设置

为采样率的一半。采样率，我们假定设置为 50Hz，那么 SMPLRT\_DIV=1000/50-1=19

2.6 配置寄存器

数字低通滤波器（ DLPF）的设置位，即： DLPF\_CFG[2:0]，加速

度计和陀螺仪，都是根据这三个位的配置进行过滤的。 DLPF\_CFG 不同配置对应的过滤情

况如表:

这里的加速度传感器，输出速率（ Fs）固定是 1Khz，而角速度传感器的输出速率（ Fs），

则根据 DLPF\_CFG 的配置有所不同。一般我们设置角速度传感器的带宽为其采样率的一半，

如前面所说的，如果设置采样率为 50Hz，那么带宽就应该设置为 25Hz，取近似值 20Hz，

就应该设置 DLPF\_CFG=100

2.7 电源管理寄存器 2

LP\_WAKE\_CTRL 用于控制低功耗时的唤醒频率

剩下的 6 位，分别控制加速度和陀螺仪的x/y/z轴是否进入待机模式，这里我们全部都不进入待机模式，所以全部设置为 0 即可

2.8 陀螺仪数据输出寄存器

通过读取这6个寄存器，就可以读到陀螺仪 x/y/z 轴的值，比如 x 轴的数据，可以通过读取

0X43（高 8 位）和 0X44（低 8 位）寄存器得到，其他轴以此类推

2.9 加速度传感器数据输出寄存器

通过读取这6个寄存器，就可以读到加速度传感器 x/y/z 轴的值，比如读 x 轴的数据，可以通过读取 0X3B（高 8 位）和0X3C（低8位）寄存器得到，其他轴以此类推

2.10 温度传感器数据输出寄存器

温度传感器的值，可以通过读取 0X41（高 8 位）和 0X42（低 8 位）寄存器得到，

温度换算公式为：

Temperature = 36.53 + regval/340

其中， Temperature 为计算得到的温度值，单位为℃， regval 为从 0X41 和 0X42 读到的

温度传感器值

2.11 中断使能寄存器

OT\_EN 该位置 1，该位使能运动检测（Motiondetection）产生中断。

FIFO\_OFLOW\_EN该位置1，该位使能FIFO缓冲区溢出产生中断。

I2C\_MST\_INT\_EN该位置1，该位使能I2C主机所有中断源产生中断。

DATA\_RDY\_EN 该位置 1，该位使能数据就绪中断（ Data Ready interrupt），所有的传感器寄存器写操作完成时都会产生

关闭所有中断则给此寄存器赋值0X00

3. 软件驱动

3.1 通过IIC对MPU6050寄存器进行读写

//IIC写一个字节

//reg: 寄存器地址

//data: 数据

//返回值: 0,正常

// 其他,错误代码

u8 IIC\_Write\_Byte(u8 reg,u8 data)

{

IIC\_Start();

IIC\_Send\_Byte((MPU\_ADDR<<1)|0);//发送器件地址+写命令

if(IIC\_Wait\_Ack()) //等待应答

{

IIC\_Stop();

return 1;

}

IIC\_Send\_Byte(reg); //写寄存器地址

IIC\_Wait\_Ack(); //等待应答

IIC\_Send\_Byte(data);//发送数据

if(IIC\_Wait\_Ack()) //等待ACK

{

IIC\_Stop();

return 1;

}

IIC\_Stop();

return 0;

}

//IIC读一个字节

//reg:寄存器地址

//返回值:读到的数据

u8 IIC\_Read\_Byte(u8 reg)

{

u8 res;

IIC\_Start();

IIC\_Send\_Byte((MPU\_ADDR<<1)|0);//发送器件地址+写命令

IIC\_Wait\_Ack();//等待应答

IIC\_Send\_Byte(reg);//写寄存器地址

IIC\_Wait\_Ack();//等待应答

IIC\_Start();

IIC\_Send\_Byte((MPU\_ADDR<<1)|1);//发送期间地址+读命令

IIC\_Wait\_Ack();//等待应答

res=IIC\_Read\_Byte(0);//读取数据，发送nACK

IIC\_Stop();//产生一个停止条件

return res;

}

//IIC连续写

//addr:器件地址

//reg: 寄存器地址

//len: 写入长度

//buf: 数据区

//返回值: 0,正常

// 其他，错误代码

u8 IIC\_Write\_Len(u8 addr,u8 reg,u8 len,u8 \*buf)

{

u8 i;

IIC\_Start();

IIC\_Send\_Byte((addr<<1)|0);//发送器件地址+写命令

if(IIC\_Wait\_Ack())//等待应答

{

IIC\_Stop();

return 1;

}

IIC\_Send\_Byte(reg);//写寄存器地址

IIC\_Wait\_Ack();//等待应答

for(i=0;i<len;i++)

{

IIC\_Send\_Byte(buf[i]);//发送数据

if(IIC\_Wait\_Ack())//等待ACK

{

IIC\_Stop();

return 1;

}

}

IIC\_Stop();

return 0;

}

//IIC连续读

//addr:器件地址

//reg:要读取的寄存器地址

//len:要读取得长度

//buf:读取到的数据存储区

//返回值: 0,正常

// 其他，错误代码

u8 IIC\_Read\_Len(u8 addr,u8 reg,u8 len,u8 \*buf)

{

IIC\_Start();

IIC\_Send\_Byte((addr<<1)|0);//发送器件地址+写命令

if(IIC\_Wait\_Ack())//等待应答

{

IIC\_Stop();

return 1;

}

IIC\_Send\_Byte(reg);//写寄存器地址

IIC\_Wait\_Ack();//等待应答

IIC\_Start();

IIC\_Send\_Byte((addr<<1)|1);//发送器件地址+读命令

IIC\_Wait\_Ack();//等待应答

while(len)

{

if(len==1) \*buf=IIC\_Read\_Byte(0);//读数据，发送nACK

else \*buf=IIC\_Read\_Byte(1);//读数据，发送ACK

len--;

buf++;

}

IIC\_Stop();//产生一个停止条件

return 0;

}

3.2 MPU6050初始化

//初始化MPU6050

//返回值: 0,成功

// 其他,错误代码

u8 MPU\_Init(void)

{

u8 res;

IIC\_Init();//初始化IIC总线

IIC\_Write\_Byte(MPU\_PWR\_MGMT1\_REG,0X80);//复位MPU6050

delay\_ms(100);

IIC\_Write\_Byte(MPU\_PWR\_MGMT1\_REG,0X00);//唤醒MPU6050

MPU\_Set\_Gyro\_Fsr(3); //陀螺仪传感器,±2000dps

MPU\_Set\_Accel\_Fsr(0); //加速度传感器 ±2g

MPU\_Set\_Rate(50); //设置采样率50HZ

IIC\_Write\_Byte(MPU\_INT\_EN\_REG,0X00); //关闭所有中断

IIC\_Write\_Byte(MPU\_USER\_CTRL\_REG,0X00);//I2C主模式关闭

IIC\_Write\_Byte(MPU\_FIFO\_EN\_REG,0X00);//关闭FIFO

IIC\_Write\_Byte(MPU\_INTBP\_CFG\_REG,0X80);//INT引脚低电平有效

res=IIC\_Read\_Byte(MPU\_DEVICE\_ID\_REG);

if(res==MPU\_ADDR)//器件ID正确

{

IIC\_Write\_Byte(MPU\_PWR\_MGMT1\_REG,0X01);//设置CLKSEL,PLL X 轴为参考

IIC\_Write\_Byte(MPU\_PWR\_MGMT2\_REG,0X00);//加速度陀螺仪都工作

MPU\_Set\_Rate(50); //设置采样率为50HZ

}else return 1;

return 0;

}

//设置MPU6050陀螺仪传感器满量程范围

//fsr:0,±250dps;1,±500dps;2,±1000dps;3,±2000dps

//返回值:0,设置成功

// 其他,设置失败

u8 MPU\_Set\_Gyro\_Fsr(u8 fsr)

{

return IIC\_Write\_Byte(MPU\_GYRO\_CFG\_REG,fsr<<3);//设置陀螺仪满量程范围

}

//设置MPU6050加速度传感器满量程范围

//fsr:0,±2g;1,±4g;2,±8g;3,±16g

//返回值:0,设置成功

// 其他,设置失败

u8 MPU\_Set\_Accel\_Fsr(u8 fsr)

{

return IIC\_Write\_Byte(MPU\_ACCEL\_CFG\_REG,fsr<<3);//设置加速度传感器满量程范围

}

//设置MPU6050的数字低通滤波器

//lpf:数字低通滤波频率(Hz)

//返回值:0,设置成功

// 其他,设置失败

u8 MPU\_Set\_LPF(u16 lpf)

{

u8 data=0;

if(lpf>=188) data=1;

else if(lpf>=98) data=2;

else if(lpf>=42) data=2;

else if(lpf>=42) data=3;

else if(lpf>=20) data=4;

else if(lpf>=10) data=5;

else data=6;

return IIC\_Write\_Byte(MPU\_CFG\_REG,data);//设置数字低通滤波器

}

//设置MPU6050的采样率(假定Fs=1KHz)

//rate:4~1000(Hz)

//返回值:0,设置成功

// 其他,设置失败

u8 MPU\_Set\_Rate(u16 rate)

{

u8 data;

if(rate>1000)rate=1000;

if(rate<4)rate=4;

data=1000/rate-1;

data=IIC\_Write\_Byte(MPU\_SAMPLE\_RATE\_REG,data); //设置数字低通滤波器

return MPU\_Set\_LPF(rate/2); //自动设置LPF为采样率的一半

}

3.3 读取MPU6050相关测得原始数据

//得到温度值

//返回值:温度值(扩大了100倍)

short MPU\_Get\_Temperature(void)

{

u8 buf[2];

short raw;

float temp;

IIC\_Read\_Len(MPU\_ADDR,MPU\_TEMP\_OUTH\_REG,2,buf);

raw=((u16)buf[0]<<8)|buf[1];

temp=36.53+((double)raw)/340;

return temp\*100;;

}

//得到陀螺仪值(原始值)

//gx,gy,gz:陀螺仪x,y,z轴的原始读数(带符号)

//返回值:0,成功

// 其他,错误代码

u8 MPU\_Get\_Gyroscope(short \*gx,short \*gy,short \*gz)

{

u8 buf[6],res;

res=IIC\_Read\_Len(MPU\_ADDR,MPU\_GYRO\_XOUTH\_REG,6,buf);

if(res==0)

{

\*gx=((u16)buf[0]<<8)|buf[1];

\*gy=((u16)buf[2]<<8)|buf[3];

\*gz=((u16)buf[4]<<8)|buf[5];

}

return res;

}

//得到加速度值(原始值)

//ax,ay,az:陀螺仪x,y,z轴的原始读数(带符号)

//返回值:0,成功

// 其他,错误代码

u8 MPU\_Get\_Accelerometer(short \*ax,short \*ay,short \*az)

{

u8 buf[6],res;

res=IIC\_Read\_Len(MPU\_ADDR,MPU\_ACCEL\_XOUTH\_REG,6,buf);

if(res==0)

{

\*ax=((u16)buf[0]<<8)|buf[1];

\*ay=((u16)buf[2]<<8)|buf[3];

\*az=((u16)buf[4]<<8)|buf[5];

}

return res;;

}

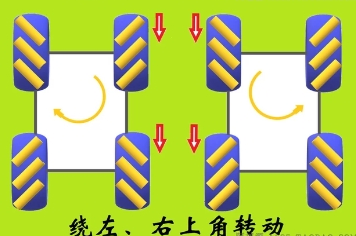
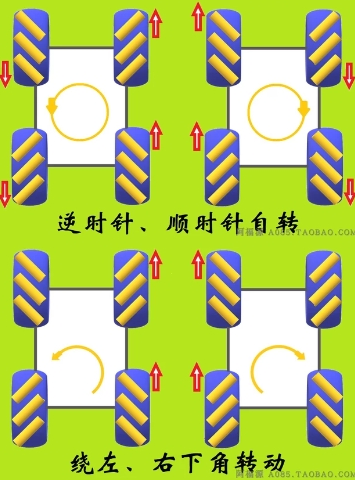
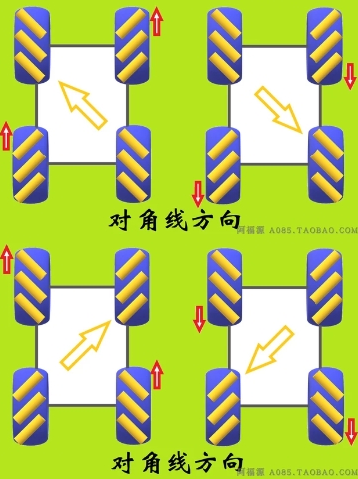
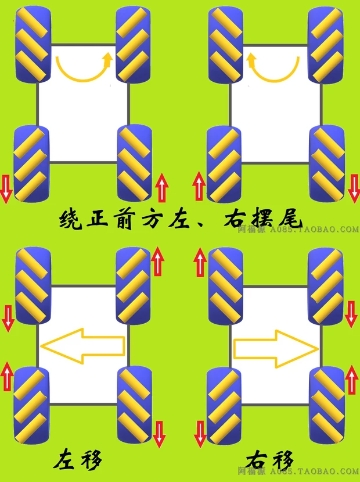
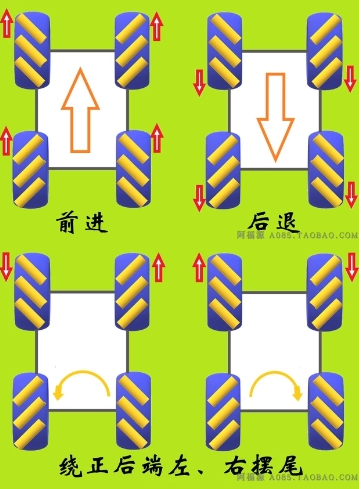
# 衡山π队内资料

执笔人：屈熙军

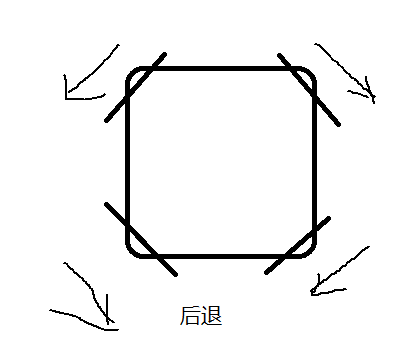
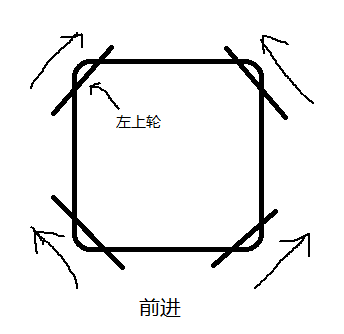
在每年的robocon比赛中，目前为止都是四轮为主，四足机器人也是去年比赛才加入到比赛中的。关于底盘的控制，我们衡山π参加robocon目前是第四届，算是一个年轻的队伍，相对于其他队伍的在底盘上的探索还是不够的。就拿去年来说，我们队用的是四个全向轮构成的底盘，这对我们来说也是第一次尝试，然而其他优秀的参赛队早已采用舵轮底盘，并且已经是很成熟的技术了。按照底盘的发展进程也是从恰好能起到支撑机器人机身的底盘（可以参考上上届做的底盘）→四轮悬挂式（第三届和第四届机器人底盘）→舵轮。基于各方面的技术欠缺，所以今年对底盘仍然采用全向轮，并在去年的基础进行了改进，在去年调试时遇到的一些问题，其中减震弹簧并未起到四个轮子贴地的效果，反而带来一些危害，例如在启动、停止的瞬间由于惯性，整个车身前翘；另一个危害就是安装在减震弹簧处的活页容易损伤。今年的底盘还没有测试。以上是对目前队内底盘设计方面的历史和经验的总结。下面介绍目前所掌握的底盘控制资料。

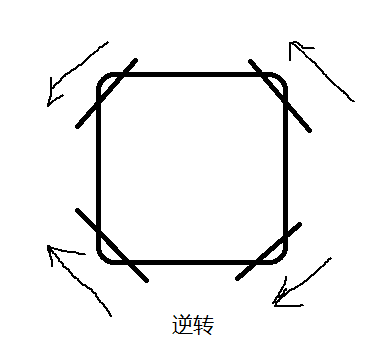
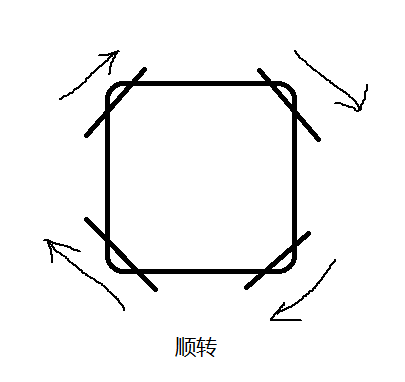
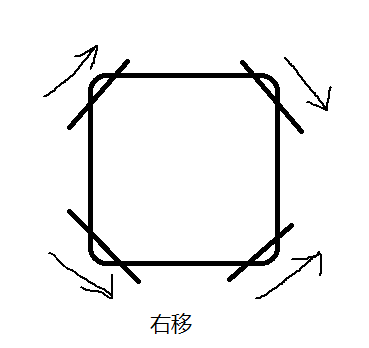
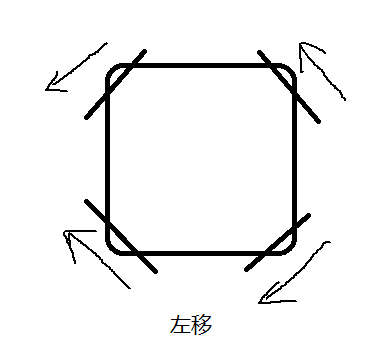
**四轮悬挂底盘**

麦克纳姆轮（robomaster惯用伎俩）：采用麦轮的话，承载能力高，但是过于笨重，不够灵活，价格也比较昂贵。控制驱动采用3508电机等。由于主轮与副轮成45°的特性，通过控制左上方轮子（右上方轮子）与右下方轮子（左下方轮子）相互挤压，即可实现车体的一系列运动。具体控制参考下图：



全向轮：相对麦轮比较灵活，质量较轻，承载力不如麦轮。由于全向轮的灵活性，全向轮的设计也可以分为四轮与三轮，如果可以的话是可以做到五轮六轮以上，随之带来的算法和控制也会变的复杂。四轮安装方式为相邻两个轮子互相垂直。通常的采用相对的两个轮子正转反转，另一对轮子采用副轮辅助运动不合理，一是方向得不到控制，速度也提不上来。所以最基本的运动姿态（各个轮子的转速理想上相同）如下图：





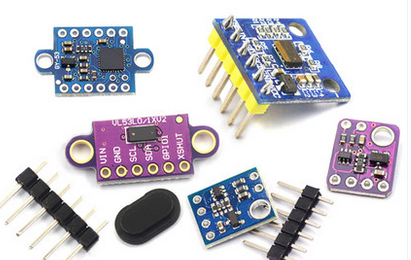
基于这些轮子的转速为什么都给相同的，一下做出解释。假如机器人要实现斜向上移动，先不管多少度，那么可以给左上轮和右下轮的速度提升，给右上轮和左下轮的速度降低，就可以实现。至于想要多少度，就需要得到每个轮子转速，将转速进行合成，使合成速度矢量为自己想要的。如果你随心所欲给每个轮子设定速度值，那么机器人就走的很飘逸。

舵轮：很遗憾这一届没能有机会制作舵轮，但是非常的希望能够在下届能够用上舵轮。舵轮的特点结构小巧，大承载；双支撑机构，受力均匀；多级行星减速，抗冲冲击；模块化设计,可自由安装多种电机。但是它的制造难度大，对零件加工精度要求高。根据其他学校的资料，他们基本都是采用大疆的2006配上u-10plue驱动作为航向驱动例如东大、武大，但是河北工程大学采用的是3508，他们也是第一次采用的舵轮，可惜的是还未测试。如果是我的话，也会选择用3508来测试。3508与2006的主要区别还是在尺寸上，2006显得更小巧些，但是3508在can通信的返回值上要比2006多。其他学校采用的主驱动电机有n5055，n5065的小体积大功率电机不等。

**辅助器件**

电子行业发展迅速，产品更新换代的速度也比较快。下面主要介绍几款常用的模块。各个模块的使用还是比较简单的，向购买的商家索取资料，结合博客上的一些例程，在了解各个模块通信的方式（can，rs485，iic，rs232等）的基础上，基本可以实现模块的基本功能。就算是新推出的产品也无非是基于这些通信方式，调用商家封装的API函数即可。

激光测距：VL53L0X，具体使用方法的资料以及例程，正点原子中也是给了的。这个模块采用IIC通信，具有单次测量、连续测量、定时测量多种模式，可以根据不同情况采用不同的工作模式。接线方式：VCC (5v电源)，GND(地)，SCL(IIC时钟线)，SDA(IIC数据线)，INT ，XSH



超声波测距：hc-sr04，单通道通信。该模块的特点是采集数据的时候误差比较大，测量范围也是非常的有限，容易受到外界干扰，以及自身存在的余震，需要对采集的数据进行滤波（均值滤波）。虽然存在许多的问题，但是对于一些小的项目这款性价比也是很好的。接线方式：VCC、TRIG（控制端）、 ECHO（接收端）、 GND



码盘定位：我们队主要使用的东北大学生产的ops-9定位系统。具体功能及实现见队内资料。码盘主要返回的值有航向角，横滚角，俯仰角，x轴坐标，y轴坐标，航向角速。我们主要使用的数据是航向角，x轴坐标，y轴坐标。

讲完了底盘最基本的驱动控制以及介绍完辅助器件，接下来是底盘的辅助控制。主要基于四轮全向轮（3508电机）。采用开环控制的话，设置每个3508电机的pwm即对应轮子的转速，封装成前进后退等动作的函数，采用ps2-ps4等手柄实现控制。确定好3508电机的死区，设置相应pwm值，转速基本可以保持一致，机器人相对跑的比较平稳，比赛现场的环境也是很好（至少对于底盘）。上面讲的是3508电机通信方式，如果采用can通信的话，通过返回的电流值和速度值实现双闭环控制，电流环为内环，速度环为外环。

算法部分待更新