# **PMAC教程**

## **PMAC卡-简介和预备知识**

PMAC卡是功能非常强大的运动控制卡，但是网上PMAC的教程很少，仅有的几个教程还是官网培训教程，罗列概念和记流水账，最不喜欢这样的教程。自己学习PMAC卡的时候走了许多弯路，刚好找完工作就写一下PMAC的相关教程，看完整个系列的教程，相信对于PMAC的使用是没有问题的，主要分为以下几个模块：

* 简介和预备知识
* PMAC硬件
* PMAC下位机编程
* PMAC上位机编程

### PMAC简介

PMAC基本上算是自动控制行业中功能最强大的运动控制卡了，虽然价格不菲，但是使用及其方便，功能也极其强大，怎么强大自己去搜啦，这是[官网链接](http://www.deltatau.com/)。

PMAC是一系列控制卡的简称，常用的有PMAC1、PMAC2、Turbo PMAC1、 Turbo PMAC2、UMAC、Clipper等，基本功能和使用方法是一样的，这里我使用的是Turbo PMAC1控制卡。

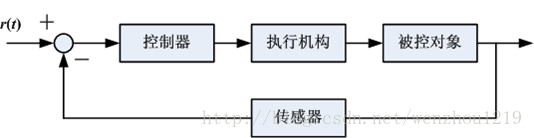
在国内PMAC有多家代理，笔者固定联系的是[苏州均信](http://www.servodynamics.com.cn/cn/index.asp)，维修的话有北京泰道公司，北京泰道公司的官方QQ群号为190220668 ，在群里面可以提问，有各路大神和官方人员解答疑问，群共享里面也有许多资料供学习。

### 运动控制基础

“工欲善其事，必先利其器”，在本文中还是先说一下运动控制相关理论，这里只是一些基本概念，深入了解还是要自行查看相关资料。

#### 定义：在复杂条件下将预定的控制方案、规划指令转变成期望的机械运动，实现机械运动精确的位置控制、速度控制、加速度(转矩)的控制。最简单的运动控制系统

我们考虑一个最简单的控制卡-驱动器-电机反馈运动控制系统



控制器：根据要求的参考输入信号（如位移、速度、力等），产生相应的控制信号，这里对应PMAC控制卡。

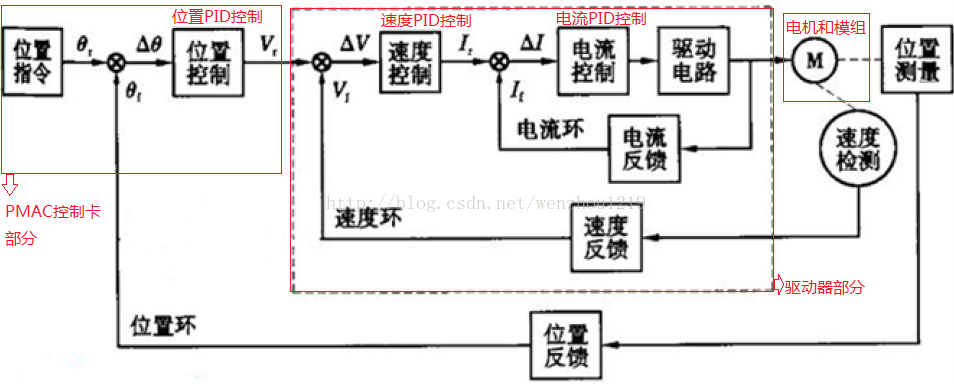
执行机构：根据运动控制器发出的控制信号，产生操作量，作用在被控对象上，带动被控对象产生相应的运动。运动控制中执行机构通常由电机及其驱动器组成的，其中驱动器提供电机功率，使得电机做旋转或直线运动。

被控对象：将被操纵的机器设备称为被控对象。这里对应常用旋转电机运动模组或直线电机模组。

传感器(反馈检测装置)：对被控对象的过程实际信号（如实际位移、实际速度、实际力等）进行检测、转换为电信号，经整形、放大提供给控制器，从而对被控对象构成闭环负反馈控制 。常用的传感器有光电编码器、光栅尺、测速发电机和张力/压力控制器等。如上图，可以看到运动控制系统有反馈检测装置，就拿这里的运动控制系统来说，不使用反馈控制的运动控制系统就叫开环系统，反馈控制的是通过编码器采集的信号来反馈的运动控制系统就叫半闭环控制系统(因为此时的被控对象的信号是间接的方式得到的)，反馈控制的是通过光栅尺采集的信号来反馈的运动控制系统就叫全闭环控制系统。

#### 三环控制

关于三环控制，有一篇比较通俗的文章([原文链接](http://wenku.baidu.com/view/1b993a54b90d6c85ec3ac6eb.html))，这里截取一段，稍作修改如下：



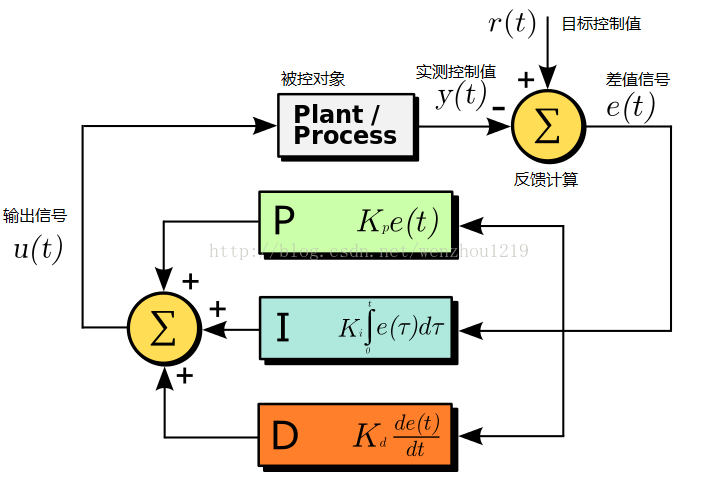
运动伺服一般都是三环控制系统，从内到外依次是电流环、速度环、位置环，这三环一起构成一个完整的运动控制系统。

1、电流环：电流环的输入是速度环PID 调节后的输出，我们称为“电流环给定”，然后电流环的这个给定和“电流环的反馈”值进行比较后的差值在电流环内做PID 调节输出给电机，“电流环的输出”就是电机的每相的相电流，“电流环的反馈”不是编码器的反馈而是在驱动器内部安装在每相的霍尔元件（磁场感应变为电流电压信号）反馈给电流环的。

2、速度环：速度环的输入就是位置环PID 调节后的输出以及位置设定的前馈值，我们称为“速度设定”，这个“速度设定”和“速度环反馈”值进行比较后的差值在速度环做PID 调节后输出就是上面讲到的“电流环的给定”。速度环的反馈来自于编码器(光栅尺)的反馈后的值经过“速度运算器”得到的。

3、位置环：位置环的输入就是外部的脉冲（如在控制卡中输入的待运动位置指令），外部的脉冲经过平滑滤波处理和电子齿轮计算后作为“位置环的设定”，设定和来自编码器反馈的脉冲信号经过偏差计数器的计算后的数值在经过位置环的PID 调节后输出和位置给定的前馈信号的合值就构成了上面讲的速度环的给定。位置环的反馈也来自于编码器(光栅尺)。

#### 在上图中，我标出了三环控制和运动控制系统基本组成的对应关系，可以看到三环控制是把控制器、执行结构、被控对象和反馈检测细化了，上图中是一种常用的伺服电机驱动模式——速度控制模式(即电机驱动器包括速度环和电流环)，其他的伺服电机驱动模式包括位置控制模式、力矩控制模式等等。还需要注意的是在每一环的控制中，我们引入了PID控制的概念。PID控制



PID是在工业控制中经典的控制算法，现在仍然被普遍的采用。

当我们知道被控对象目标控制值，和当前被控对象实际测量值，那么就可以计算出实际控制差值，这就是反馈，那么拿到这个反馈要如何处理得到输出信号给被控对象使得被控对象的实际测量值越来越接近目标控制值呢？这时候就要有一种计算输出信号的算法，这里就该PID工作了。

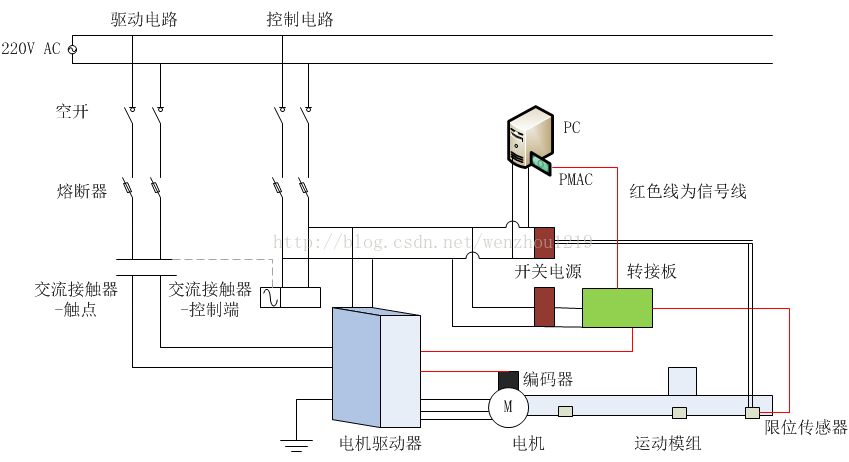
PID详细名称为：比例-积分-微分控制方式，是一种误差控制算法，综合作用可以使系统更加准确稳定的达到控制的期望。关于PID算法的原理和调节策略通俗讲解参见[这篇文章](http://blog.gkong.com/liaochangchu_117560.ashx)。

如果想深入研究，参见[这篇文章](http://wenku.baidu.com/link?url=ElpBCAT854OH6LNcFCl_DUDVLTJiq81MVhbhNPkLIWeB-KUbvPsnqbsc8_lyPO1eOo_ZbMhLkC0qP7LSD1EISV4QbH0FIu2u7-8cdoeYtVm)，英文好的好的话也可以参见[wiki](http://en.wikipedia.org/wiki/PID_controller)。在使用PMAC完成运动控制的时候，调整PID是必不可少的一个环节，一定要对PID原理有基本了解和知道PID的调节步骤。

## **2.PMAC硬件——一个运动控制系统的实例**

运动控制系统的构成有很多方式，你可以使用嵌入式系统、专用运动控制PLC、工控机+运动控制卡、专用数控装置等等。PMAC控制一般采用的都是工控机+运动控制卡（IPC）的方式，这样有搭建系统比较灵活，可扩展性强等优点。

下面是一个完整的PMAC运动控制系统硬件组成和接线及信号传输示意图。



整个运动控制系统按信号流通顺序，由PC->PMAC卡->PMAC卡转接板->电机驱动器->电机(编码器)->运动模组(限位传感器)构成。图中红色线为信号传输示意图。

对于一个控制系统来说，都有驱动电路和控制电路两路电。

图中，驱动电路主要用于给驱动器供电，处理后输出电机驱动电以供电机正常工作，这里不同的驱动器要求不一样(220V或380V),这里为了简单使用220V。

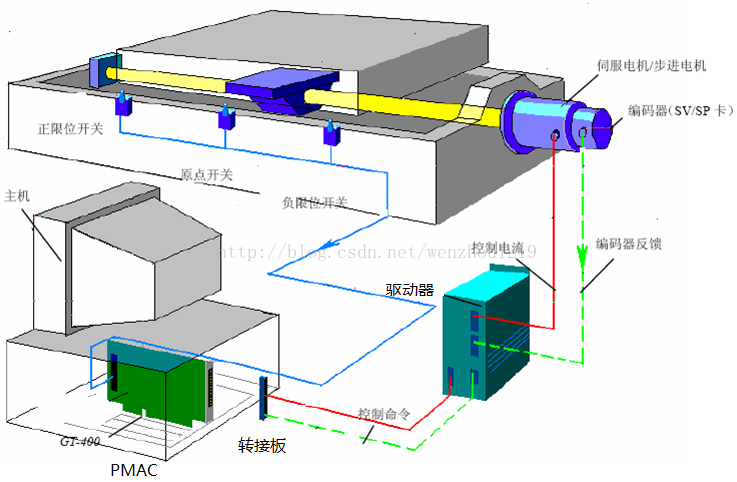
图中，控制电路主要控制整个系统：

1.交流接触器控制主驱动回路的通断（可采用直流接触器24V或交流接触器220V，这里用220V）

2.电机驱动器控制电路供电（不同的驱动器要求不一样，直流24V或交流220V，这里为了简单使用220V）

3.转接板供电（转接板是将PMAC的信号分流，可使用官方转接板或自行设计的转接板，220V变压后输出15V给转接板供电）

4.图中的限位传感器供电，220V变压后输出24V给转接板供电。(这里只是示意，实际中信号线和供电线都是从转接板走线)下图是更加形象的说明整个系统的构成：

图中的正负限位起保护作用，零限位用于编码器+滚阻丝杠模组回零时找到原点(因为编码器一般是增量编码器、不知道自己的当前位置)，如果是光栅尺反馈也可以使用标记原点来回零。

注意图中的运动模组，这里使用的是伺服电机+编码器+滚珠丝杠模组，构成半闭环系统，也可以使用直线电机+光栅尺(或伺服电机+编码器+滚珠丝杠模组+光栅尺)构成全闭环系统。

如果是滚珠丝杠模组，需要了解导程这个概念，即丝杠转动一周模组前进的距离，常见导程为10mm,20mm。

如果是使用光栅尺作反馈，需要了解光栅尺分辨率这个概念。光栅尺的分辨率，是指光栅尺可读取并输出的最小长度变化，对应的参数有：每毫米光栅刻线数、脉冲数、细分等。目前国内光栅尺分辨率一般有5μm、1μm、0.5μm、0.2μm、0.1μm。以每毫米50线光栅为例，经过4细分，就能得到很简单的5μm的分辨率，至于高分辨率的光栅尺，就必须采用电子细分技术。

再说整个模组的定位精度和重复定位精度，简单来说定位精度就是在一次定位测量中运动测量结果和实际目标指令距离的偏差范围，重复定位精度就是在多次定位测量中运动测量结果相互之间的偏差范围。

整个模组的定位精度和重复定位精度是我们在设计一个运动控制系统时必须考虑的因素，这个受诸多因素的影响，必须认真考虑。如光学部分、机械部分  、电气部分等等。这里有个常见的问题需要强调，整个模组的定位精度和重复定位精度是由实际测量元件和运动元件决定的，如果只是提高编码器、光栅尺等测量元件的精度，对于运动部分(电机、模组)的精度不够，那么整个模组的定位精度和重复定位精度依旧提不上去，举个最简单的例子，现在在100mm处，想运动到100.1mm处，光栅尺、编码器的精度可精确到0.1um，但是电机+模组精度有限，一个脉冲就跑了1mm，那么无论如何也是不能够跑到100.1mm处的啦。经常有人说我的测量精度是够的，为什么不能够到精确运动啊，这个很可能和你的运动部分有关系。至此，了解PMAC整个运动控制系统的构成，下面就可以开始使用它了。这篇教程主要是为了进行PMAC软件开发的人员写的，如果是设计整个控制系统的设计人员需要进一步去了解各方面的知识。

## **3.PMAC硬件-组成和接口及安装**

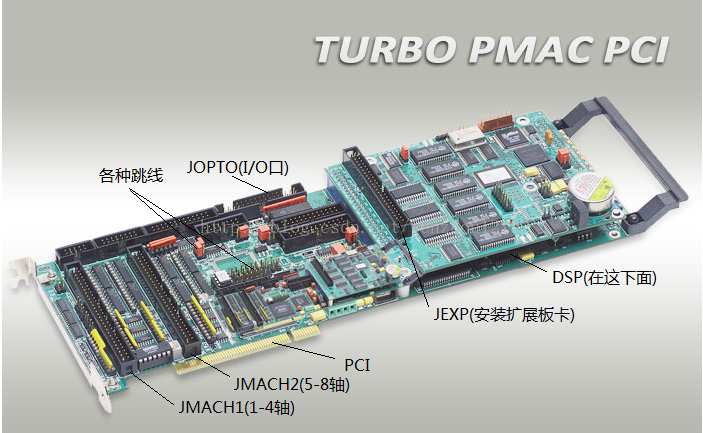
拿到一块PMAC卡，首先我们要看一下他的硬件组成(DSP)和提供的各种接口(I/O、串口等)，然后就开始做准备工作(设置跳线)，最后是安装(插槽PCI和转接板)。

### 硬件组成和接口

PMAC是基于Motorola DSP的伺服控制卡，不同型号的PMAC可控制的最大轴通道个数不一样，DSP型号也不一样。有专门的伺服芯片来完成编码器反馈计算等工作，PMAC中最容易坏的地方就是这组芯片，实际部署时要做好接地保护和高压保护，一定不要热插拔。

我所在实验室用的是Turbo PMAC PCI1，我也是用这个做示例，其他的诸如UMAC、Clipper等系列应该差不多，请自行研究。

如下图标出了常用接口：

**JEXP**是安装扩展板卡的扩展口，如常见的增加通道个数(默认为8个通道)的卡等等，这是官网的[扩展卡选配链接](http://www.deltatau.com/DT_Products/AccessoryCompatibleList.aspx)。

**JMACH1和JMACH2**分别是对应1-4轴及5-8轴的控制信号输出通道，一般和转接板相连。

**JOPTO**是8路输出和8路输入通道口，一般的IO信号输出输出使用它。

PMAC有各种和上位机(PC)相连的方式，这里采用**PCI**接口，不同的型号还有VME、USB等方式。其他接口概不详述，请查阅手册。

### 准备工作

在上图中，我标出了跳线的位置，图中包括两种：使用跳线帽(红色)跳线和未跳线。

跳线就相当于我们使用的软件设置选项一样，通过设置不同的跳线来配置不同的硬件功能，这是根据使用需求决定的，一般默认跳线是满足通用需求的。

常用跳线如下：

#### 1.供电跳线

一般采用默认如下：

E85、E87、E88选择OFF，E89 跳线选择ON

E90跳线选择1-2

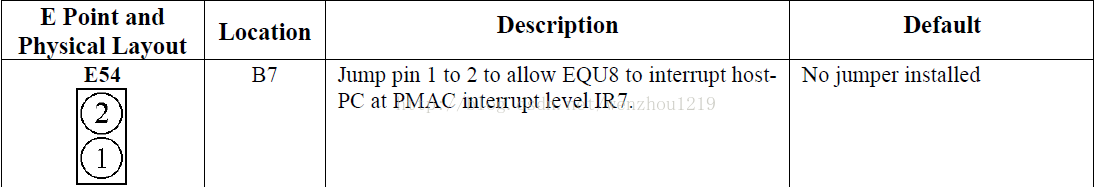
E100跳线选择1-2

#### 2.电机控制跳线

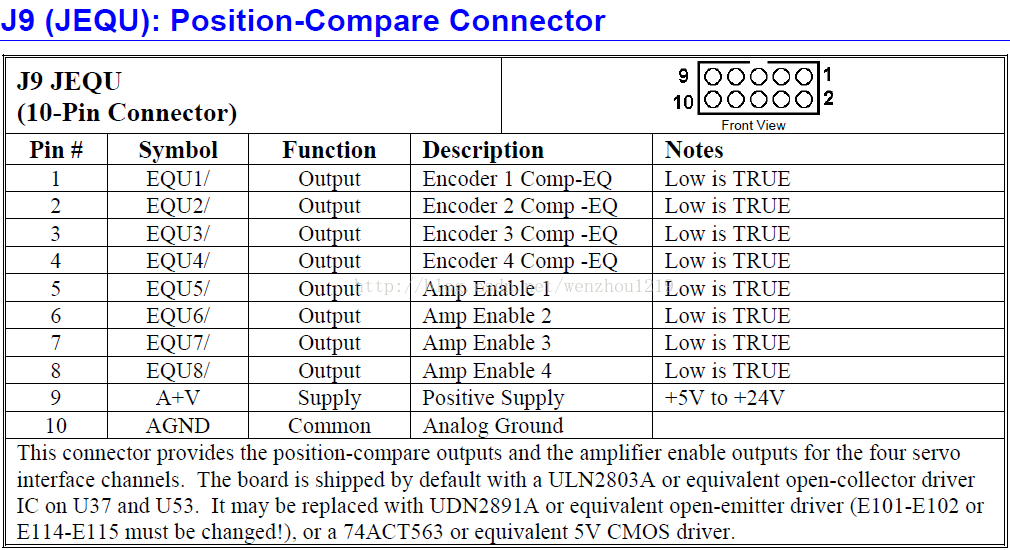
E17A、E17B、E17C、E17D跳线选择OFF（控制伺服电机选择OFF，控制音圈电机/直线电机选择ON）

#### 3.中断跳线

中断是下位机(PMAC)向上位机通信的机制，这里我们需要通过跳线配置中断触发的通道和变量。查阅硬件手册E54-E65用于设置中断，这里手册有点不清楚，我来解释一下，比如对于E54：



可知默认没有跳线，一旦找到跳线，使用跳线帽连接，允许通过**EQU8**通道来向上位机发送中断请求以完成向上通信，这个中断级别为**IR7**，级别越高，越先被响应，PMAC用户可使用的有IR5、IR6、IR7三个中断级别，IR1-IR4应该是被PMAC自身保留采用。那么这里EQU8是什么通道呢，再查手册，可以看到EQU8是JEQU接口的一个通道，如下图



JEQU这里可以理解为允许外部触发产生中断，外部触发向上位机通信。另外，这里涉及到后面的知识，查阅PMAC 软件手册，看到

http://img.blog.csdn.net/20141121112200508?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

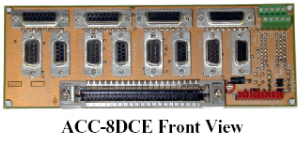
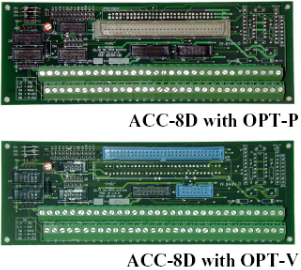
即可以在PMAC下位机程序中，通过设置M812或M813的值来模拟外部触发，从而完成下位机编程时向上位机中断请求。这里暂时看不懂不要紧，后面还会讲到，这里是为了连贯，把这些知识放在一起。

#### 4.其它设置跳线

这里主要是说一下常见需要注意的参数：如E51跳线连接后开机会重置PMAC卡所有参数到初始状态，这一般是我们不需要的，注意检查，默认为未连接，但是代理公司维修时经常将此跳线连接。

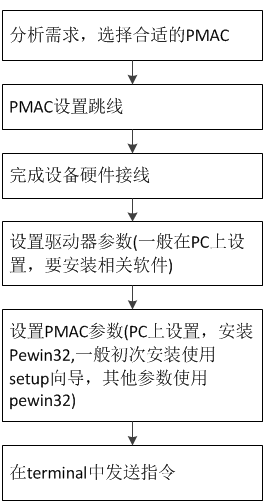
### 安装

PMAC安装比较简单，不同的接口和PC直接连起来就行，需要注意的是JMACH接口的接线，JMACH包含了供电、驱动器、编码器、限位各种信号，需要将其分流，最好使用转接板，就是把60针的排线分开，最简单的就是使用官方的ACC-8D和ACC-8DCE转接板，分别如下图：



### 前者直接将排线分各个接线柱，后者分成若干个接线DB头。其他功能更强大的转接板，请在上面给出的扩展板链接中查找。当然，也可以查手册自行了解各个针对应的信号意义，自行设计转接板，如Turbo PMAC PCI的手册上各个针的意义如下：http://img.blog.csdn.net/20141121121116307?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center安装后的设置

假设我们所有硬件都已经安装完成，下一步就是对不同的硬件做参数调节了，具体的PMAC安装流程如下示意图。



这里驱动器设参数设置不同的驱动器不一样，请自行查阅手册。

PMAC参数设置会用到pewin32软件，这将会在下一节讲到。

那么基本上关于PMAC硬件的介绍就是这么多了。再次说明，这部分主要是为了进行PMAC软件开发的人员写的，如果是设计整个控制系统的设计人员需要进一步去了解各方面的知识和查阅硬件手册。

## **4.PMAC下位机-PMAC安装参数设置**

PMAC硬件安装完成后，必须使用工具对PMAC相关参数进行调节才能正常工作。本文先介绍PMAC PC操作工具PEWIN32，然后一步步说明PMAC安装参数设置过程。注意在此之前要完成驱动器的参数设置(即通常说的驱动器整定)。

### 下一节介绍下位机相关知识和下位机编程，这些都是在PEWIN32中。PEWIN32下载安装

PMAC功能非常强大，主要用来配置PMAC卡、查看PMAC控制卡的各种状态变量及编写、调试PMAC程序等。(这里推荐使用PEWIN32PRO，不推荐使用PEWIN32 PRO 2，这个新版本有很多bug且不稳定)

PEWIN32的相关安装包和license在购买pmac时向代理商索要即可。

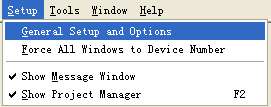
在安装PEWIN32之前要安装PComm32 PRO，这是PMAC PC驱动和编程接口库，没有安装这个，PC是无法和PMAC通信的，PEWIN32和自己编写的程序都工作不了。安装界面如下：



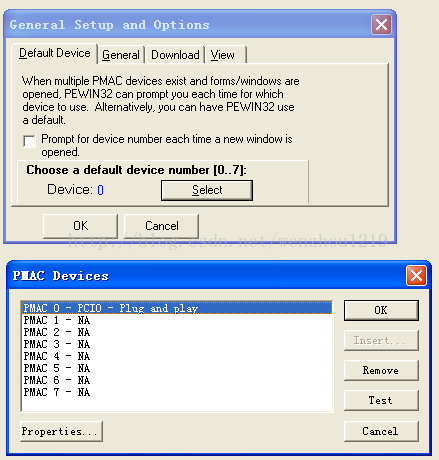
接下来安装PEWIN32，注意PEWIN32只是一个主程序，需要将附加的一系列调试工具plot、tuning、setup安装。安装界面如下：

#### http://img.blog.csdn.net/20141218215530665?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center安装参数设置过程1.建立连接

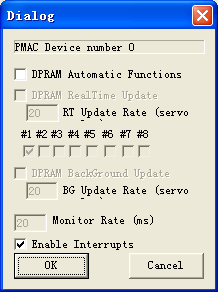
如下图，在setup中选择打开安装选项



在Default Device中选择当前Pewin32连接的卡号



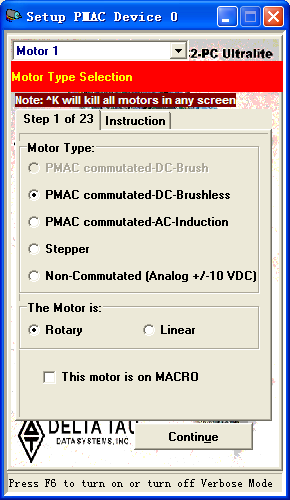
注意这里打开Properties勾选上Enabale Interrupts,否则后续的中断程序无法成功发送中断



同样确认这里的Download标签页中的第四项Do not download...没有勾选，否则后续的下位机程序无法成功download

#### http://img.blog.csdn.net/20150411181434964?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center2.配置电机和反馈

建立连接后下面就是配置电机和反馈，这包括根据当前的电机类型、反馈形式(编码器、光栅尺)、轴通道选择等设置PMAC卡参数。PEWIN32中在Tools选择对应的Setup工具，如本人使用的Turbo Pmac1的Setup界面如下：



#### 按照引导一步步设置参数即可。3.开环测试

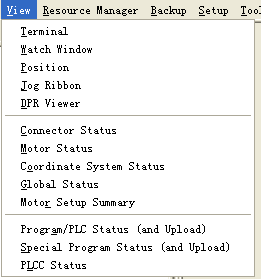
如图，选择打开Tuning Pro，对指定轴做开环测试

#### http://img.blog.csdn.net/20150411182821734?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Centerhttp://img.blog.csdn.net/20150411182914599?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center4.PID调节

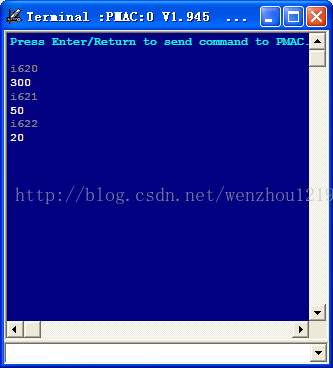
开环测试通过后，同样在Tuning Pro中调节PID参数使电机稳定运行。

#### http://img.blog.csdn.net/20150411182834027?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center5.当前硬件状态测试

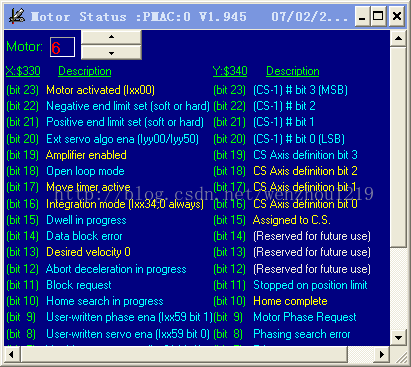
当所有的参数设置完成后，可以如下图打开Terminal界面、Motor Status界面和Position监控界面



Terminal界面主要用来动态发送命令、设置参数和查看参数、运行程序等，连接成功后打开会显示如下图最上面一行文字。



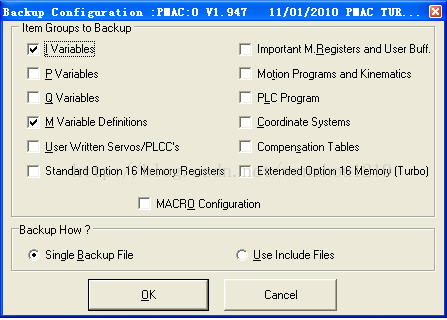
Motor Status主要用来查看当前电机激活状态、各个限位开关状态，可以用手遮挡各个限位开关，查看各个状态是否做出正确变化



Position默认用来监测当前各个轴的位置，右键可以添加对速度、跟随误差的监控，这里可以用手推动各个轴看反馈是否正常。

#### http://img.blog.csdn.net/20150411183313685?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center6.保存参数

在命令行中输入save命令将当前配置的参数写到EEPROM中，这样下次启动时就会使用新配置的参数。同时可以使用Backup->Upload params将参数备份到磁盘文件中以防止意外修改了当前参数可以使用Backup->Restore 来恢复参数。

这里只是简单的列数了整个安装设置过程，对于软件开发人员来说了解一下即可。对于Setup的过程、开环测试和PID调节方法等涉及到很多繁琐的细节，前人已经写了比较好的教程——《[PMAC入门教程-让电机动起来](http://wenku.baidu.com/link?url=LGDiwcfyN6oqveDBJys3Otfhwz0CkLoe-eUoBBKzNUlmiS6WKFGCg_KU3iWuFXo8Fof1nrac4teXVlTyIBvTLl6bvzdb25if0Da34DJ1RPK)》，这篇教程中的主要针对网卡通信方式连接的PMAC，但是本质上差不多，如果希望作深入了解可以看一下。

## **5.PMAC下位机-下位机编程基础**

在开始下位机编程之前，先要了解下位机编程的基础-各种变量、运动参数和命令行操作。

### 变量

PMAC的下位机实际上是一种类Basic的编程语言，但是又类似汇编语言，所有变量的存储空间都是预先分配好的，操作的时候存储数据只能在指定的几种变量类型和变量名中存储，整个系统的参数也是存储在变量中。

变量类型和作用如下：

#### 1.I变量

电机和编码器等的参数变量，用于设置电机的速度，精度，回零等数值，以及坐标系的状态，编码器的反馈形式，PID参数等，一句话，设置系统参数就靠它。

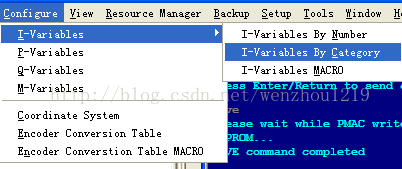
举例子如下：

Ixx19->最大手动加速度限制

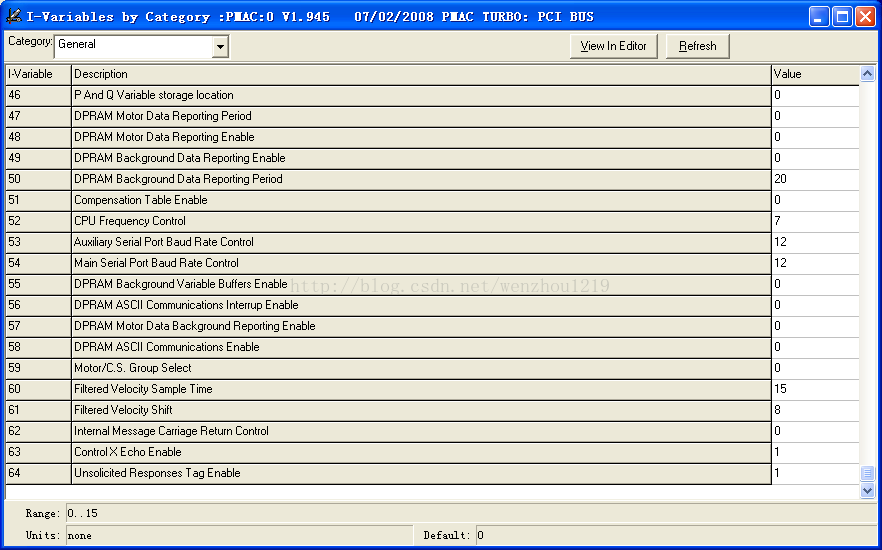
Ixx20->手动加速时间

Ixx22->手动速度

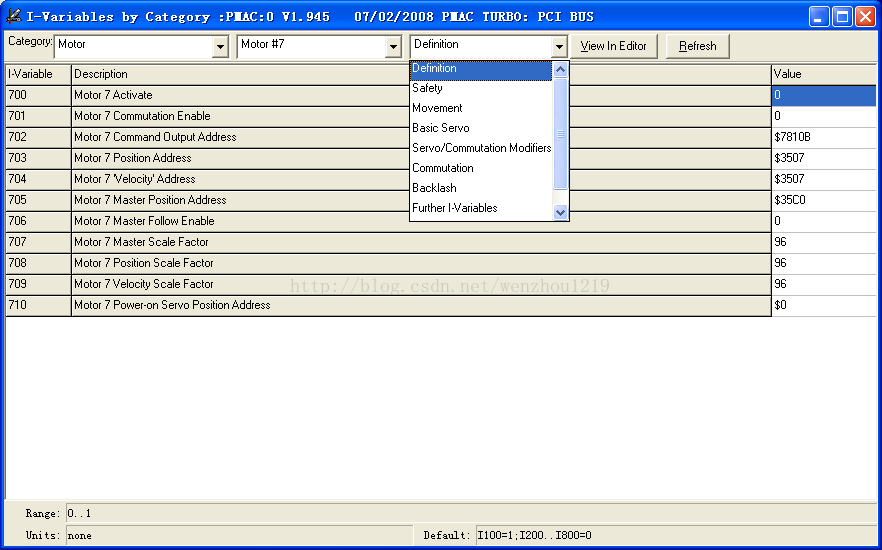
在PEWIN32中，如图打开I参数设置，一般我们使用第二个



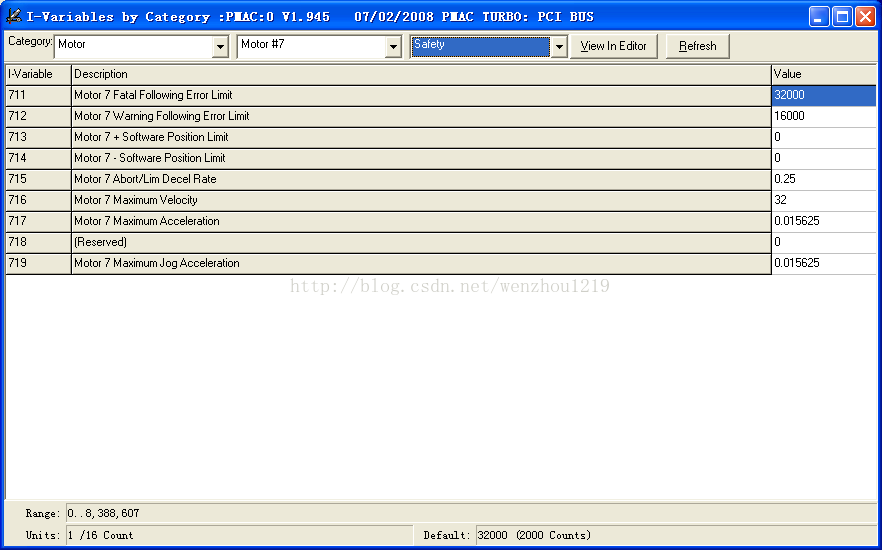
如下图，为全局I参数设置，可以根据Description看到不同I参数的含义



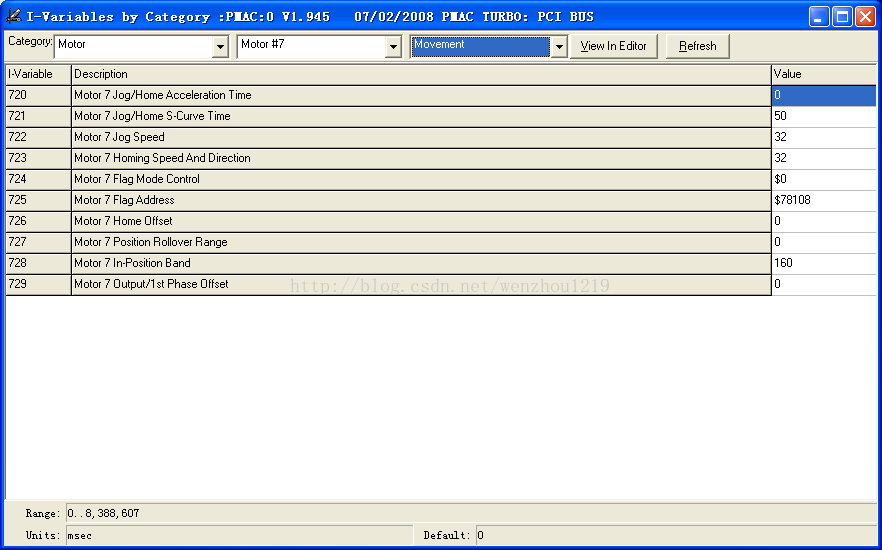
如下图，为当前电机的定义，如编码器反馈地址等等。实际上，上节讲到Setup引导设置也是设置电机的I参数，如果我们知道要设置哪些参数，完全可以手动在此修改，不用借助setup程序，setup程序只是做一个人性化界面引导而已。



如下图，安全设置包括电机的跟随误差(实际运动位置和命令运动位置的偏差)的Fatal(电机被停止)和Warning(只在Terminal报警，不停止电机)，还包括后面讲到Jog的最大加速度设置、运动程序的最大速度和加速度设置，当我们编程序时发现始终运行不到指定的速度很有可能就是此处的设置问题。



如下图，为电机的运动设置，包括后续的Jog的速度、加速度设置等。



如下图，这里保存的就是Tuning Pro调试的结果，同样，如果我们知道要使用什么PID参数，直接在此设置即可。

#### http://img.blog.csdn.net/20150411190100907?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center2.M变量

在PMAC工作时，会将当前运行状态(电机参数，编码器位置，IO参数)放到指定的内存中，我们要查看这部分参数，只需要用M变量绑定指定内存即可访问和设置，有点类似C语言的指针，如下图，这里可以电机Download Suggested M-variables使用PEWIN32推荐的M参数设置，一般我们使用默认设置。

举例子如下：

读取电机状态->给定位置（mx61)、实际位置(mx62)、位置偏差（mx64) 其中x=1-16或1－24

读取I/O口状态->m11

在上位机的学习软件、监控程序以及下位机程序的编写过程中，电机当前位置、I/O口的状态等数据，采集时均由M变量来完成。

#### http://img.blog.csdn.net/20150411191215109?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center3.P变量和Q变量

P变量和Q变量都可用于PMAC编程中的计算，48-位浮点形式，P变量为 P0 ~ P8191，Q变量为Q0~Q8191，不同的卡P变量和Q变量数量稍有不同，他们类似于我们在C语言中编程变量，只不过他们的名字是预先指定的。

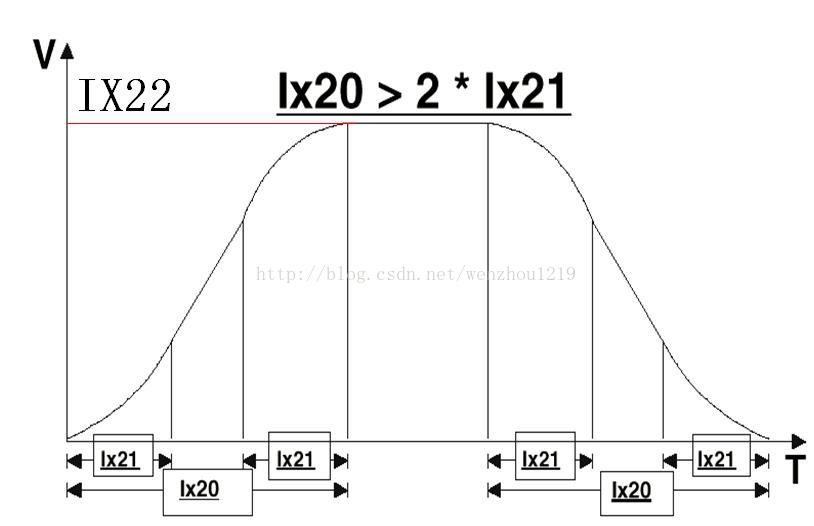
P变量和Q变量不同之处是前者是全局变量，可在任意时刻任意程序或任意位置进行改变，从而对其它程序或状态直接产生影响；而Q变量是单一坐标系的局部变量，坐标系的概念稍后介绍。

主要用途如下：

1. 可用于计算功能。                     P100=P101\*45

### 2. 可作为标志位软件触发             IF( M1!= 1 AND P10 = 0)PMAC中运动过程参数

对于PMAC中一个指定的运动加速到指定的速度再停止，整个过程如下：



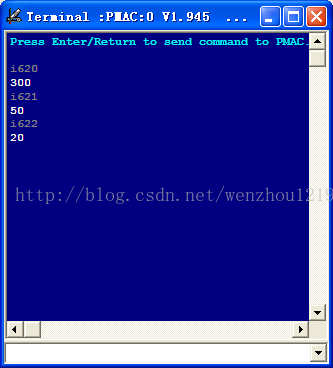
### 其中IX20指定的为整个加速时间，IX21指定的为开始加速和停止加速时的S曲线加速时间，IX22为最终的速度，这里IX20、IX21、IX22设置的是手动Jog的总加速时间、S曲线加速时间和运动速度。在对应的运动程序中，使用TA、TS、F指令来完成相应的同样的设置。手动操作

PMAC的手动操作主要是为了调试方便，主要指令如下，可以自行在view->terminal窗口中测试：

1.查看和设置各种类型变量I/P/Q/M变量的值

2.控制电机Jog运动

3.控制程序的运行状态如下图，为设置6号轴电机的运动参数(总加速时间Ixx20、S曲线加速时间Ixx21、运动速度Ixx22)



可以terminal窗口控制电机的运行，常用的几个电机手动操作指令是需要记下的，如下：

J/    - Jog 运动停止 (或者闭环)

J+   -  Jog正向连续运动

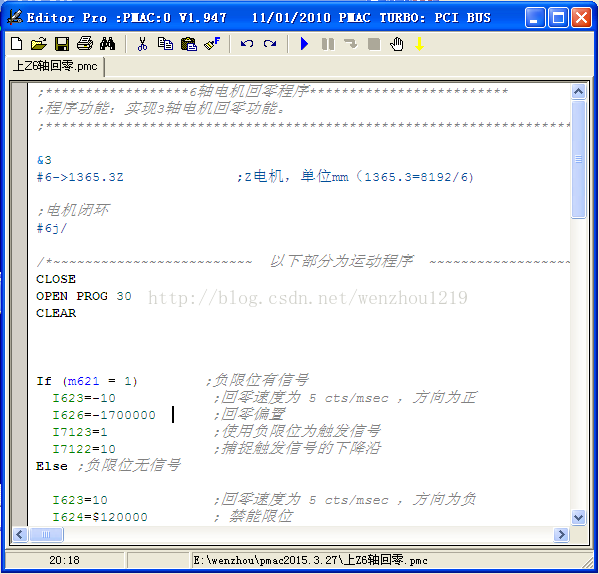
J-    - Jog 反向连续运动

J={常数}  -  Jog到指定的位置(单位是脉冲counts)

J^{常数}  -  Jog从当前实际位置运动指定的距离

## Ctrl +K K掉所有电机如控制6轴正向运动，#6j+即可**6.PMAC下位机-下位机编程**

PMAC的下位机编程包括三种程序：命令序列程序、运动(Program)程序和PLC程序。如下图,PMAC中编写程序，直接在File->new新建文件会打开编辑器，所有文件为PMC后缀。

查看当前程序可以在File->Uplaod Programs查看当前已有的Program程序和PLC程序及他们的编号、状态。

### http://img.blog.csdn.net/20150411194543101?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center1.命令序列程序

上节讲到，PMAC卡中可以使用手动操作，在命令行设置参数状态、控制电机等，但是它一次只能发送一条命令，那么为了一次性运行多个命令序列，即可在PMC程序中如下编写：

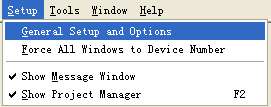
**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. i620=200
2. i621=20
3. i622=10
5. #6j/

这里先设置6轴电机的运动参数，然后激活电机，类似的写法经常用于PMAC参数的初始化中。

程序编写完成后，点击Download按钮，命令系列程序在下载的过程中就会执行。

如下图，勾选Show Message Window，打开Mesage窗口



如下图，在Message窗口可以可以看到当前程序是否有错误等，类似C语言编译过程。

### http://img.blog.csdn.net/20150411195511765?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center命令序列经常和PLC程序及运动Program程序混用，下面会说明。2.运动(Program)程序

为了便于编写运动程序，PMAC中使用一个坐标系的概念，比如有6号电机和8号电机，我们在一个的坐标系下将其分别映射到X和Y坐标，如下所示

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. &1
2. #6->819.2Y ;机械手Y向，单位mm,(819.2=8192/10,丝杆导程为10mm)
3. #8->819.2X ;机械手X向，单位mm,(819.2=8192/10,丝杆导程为10mm)

这样我们编程的时候，可以直接把当前运动想成现实坐标系的运动，实际行我们在程序中指定X运动1个单位就是电机运动819.2个脉冲，换算成轴的运动距离就是1mm,这样编写程序简洁易懂。标准运动程序的模板如下:

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. &1                      ;坐标系为1
2. CLOSE                   ;确认所有缓冲区被关闭
4. OPEN PROG 40            ;程序号为40
5. CLEAR
6. ;;;;;;;;;;;;;;;;;;;;;运动程序主体-开始
8. ;;;;;;;;;;;;;;;;;;;;;运动程序主体-结束
9. CLOSE

这些指令都是命令指令，所以他们都是在程序下载的时候运行，之间的运动程序主体会被写到特定的缓冲区，但是在这时候程序是不运行的。下面写编写一个简单的程序，如下：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. &1
2. CLOSE           ;确认所有缓冲区被关闭
3. #6->819.2X       ;机械手X向，单位mm,(819.2=8192/10,丝杆导程为10mm)
4. #8->819.2Y       ;机械手Y向，单位mm,(819.2=8192/10,丝杆导程为10mm)
6. #6j/
7. #8j/
9. OPEN PROG 40
10. CLEAR
12. INC             ;增量运动模式
13. X(10) Y(10)     ;X正方向前进10mm,Y正方向前进10mm
15. CLOSE

注意这里，使用命令序列完成了轴的定义和电机的激活，但是中间的两行运动程序代码只是被写到对应的缓冲区，还没有执行。在terminal窗口输入命令&1b40r程序才运行。注意一个坐标系下关联的所有电机必须都激活，运动程序才能正常运行。在实际中，轴的定义和电机的激活最好放到单独的命令序列文件中。下面是一个比较复杂的程序：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. &1
2. CLOSE               ;确认所有缓冲区被关闭
3. #6->819.2X           ;机械手X向，单位mm,(819.2=8192/10,丝杆导程为10mm)
4. #8->819.2Y           ;机械手Y向，单位mm,(819.2=8192/10,丝杆导程为10mm)
6. #6j/
7. #8j/
9. OPEN PROG 40
10. CLEAR
12. ;运动参数设置
13. TA(200)             ;ms
14. TS(20)              ;ms
15. F(50)               ;mm/s
17. ABS                 ;绝对模式
18. Linear              ;直线运动
20. p1 = 0
21. While (p1 < 10)
22. F(50)
23. X(30) Y(10)
24. F(10)
25. X(10) Y(30)
27. If (p1 > 2)
28. Return
29. EndIf
31. p1=p1+1
32. EndWhile
34. CLOSE

上面程序的含义就是从当前位置以速度50mm/s直线运动到绝对坐标(30,10)的位置，再从当前位置以速度10mm/s直线运动到绝对坐标(10,30)的位置，如此循环，如果没有If语句的话，循环10次，但是这里加了If语句就只循环2次了。基本的运动程序就是这样，再复杂的运动程序都是在此基础上延伸出来的。各种运动指令请查PMAC Software Manual手册，相应的程序编写请查PMAC User Manual手册。

### 3.PLC运动程序

在电气系统中，PLC是经常用的控制元件，就是有快速的扫描各个状态，对扫描到的状态变化做相应的处理。他的实时性非常高，简洁可靠。在PMAC中有模拟PLC的功能。

同样对于运动程序中的最后一个实例功能，改写如下：运动程序：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. &1
2. CLOSE               ;确认所有缓冲区被关闭
3. #6->819.2X           ;机械手X向，单位mm,(819.2=8192/10,丝杆导程为10mm)
4. #8->819.2Y           ;机械手Y向，单位mm,(819.2=8192/10,丝杆导程为10mm)
6. #6j/
7. #8j/
9. OPEN PROG 40
10. CLEAR
12. TA(200)             ;ms
13. TS(20)              ;ms
14. F(50)               ;mm/s
16. ABS                 ;绝对模式
17. Linear              ;直线运动
19. p1 = 0
20. While (p1 < 10)
21. F(50)
22. X(30) Y(10)
23. F(10)
24. X(10) Y(30)
26. p1=p1+1
27. EndWhile

CLOSE

PLC程序：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/44998165) [copy](http://blog.csdn.net/wenzhou1219/article/details/44998165)

1. OPEN PLC 20
2. CLEAR
4. If (p1 = 2)
5. COMMAND "&1b40a"
6. EndIf
8. CLOSE

enable plc 20  一般我们让PLC程序始终在后台运行，扫描指定的状态，这里我们在download PLC的时候就激活了PLC程序。这样当在terminal中输入&1b40r运行运动程序时，一旦p1=2,PLC可默认为立即扫描到该状态，这时候就会终止该程序。这里本来运动程序会运行两个循环，但是由于PMAC的Lookhead的功能，这里只运行了一个循环就中止程序了，先留个悬念，马上讲到。

### 4.PMAC运动程序Lookhead功能

PMAC在执行的时候会有Lookhead的功能，顾名思义就是前瞻的功能，就是在执行的时候会预先读取执行下面一段代码，对于运动轨迹相关的代码会进行预先计算，变量赋值会提前进行。分析如上运动程序，在第二个循环的时候，预先读取了整个循环内容，即运动指令还没运行时，p1=p1+1就已经被执行了，这时候PLC检测到p1的改变就会中止程序了。

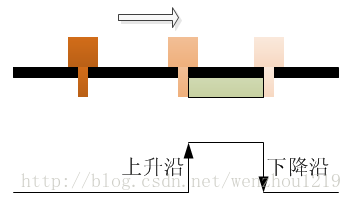
那么这里要怎么做才能避免预读取带来的副作用呢，其实很简单，在X(10) Y(30)后加一句dwell 0即可，这告诉运动程序，预读取就到此位置或者说必须dwell 0前的程序执行完毕才会进行后续操作。Lookhead还会带来其他的影响，如对一个直角拐弯，开启了lookhead后实际运行轨迹为圆角拐弯，这个要注意，具体的请参考手册。

## **7.PMAC下位机-回零程序的编写**

在运动控制中，运动轴的回零是一个共性的问题：对于半闭环的编码器+电机+模组或全闭环的直线电机+光栅尺来说，它们之所以能够知道自己当前处于一个什么样的位置就是靠编码器和光栅尺来记录当前的位置，问题在于一般我们使用的都是相对式的编码器或光栅尺，换言之，我们必须告诉编码器和光栅尺以什么位置作为零点开始计算当前的坐标，指定当前的零点的过程就是**回零**的过程。相对式的编码器或光栅尺一旦掉电，就必须重新回零，这也是大多数数控机床上电要回零的原因。如下图，为了指定当前运动系统的零点，我们使用一个零位传感器来标记，对于编码器+电机+模组，零位传感器就是一个处在零位的限位传感器，对于直线电机+光栅尺可以不用设置特殊的硬件，直接捕获光栅尺的零位信号就行，当然你也可以不用零位信号，而统一使用限位传感器来指定零位。同时每个运动系统一般都是有正负限位的，一方面是作为安全用途，一旦模组运动超出正负限位范围自动停掉电机防止造成危害，另一方面，也可作为触发使用，下面会讲到。

### http://img.blog.csdn.net/20150424175000887?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvd2VuemhvdTEyMTk=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast1.信号的触发

为了形象的说明问题，考虑模组和传感器的如下相对运动，这里传感器既可以是限位传感器也可以是零位点触发，基本原理是一样的。这里为了说明问题，将传感器拉长了，实际上传感器的接触长度很短。

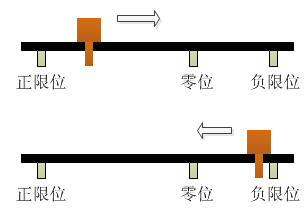


可以看到，当前模组和传感器相对运动的时候，还没接触的时候信号为低电平，一旦模组开始进入传感器范围，这时候电平跳变为高电平，存在一个上升沿，捕获这个上升沿就是模组刚刚进入传感器范围的时刻。模组继续运动，一旦模组完全离开传感器范围，这时候电平跳变为低电平，存在一个下降沿，捕获这个下降沿就是模组刚刚完全离开传感器范围的时刻。

### 因为最开始的时候没有回零，这时候模组是不知道自己当前到底处于什么位置的，但是在高低电平跳变的时候会发送中断，我们可以捕获这两个中断从而判断当前模组是否运动到指定的位置，从而将当前指定位置标记为零点。2.回零策略

上面其实已经叙述了最基本的思想：利用信号触发的上下沿来通知PMAC是否已经运动到指定位置，从而将当前模组在的位置标记为相对零点。

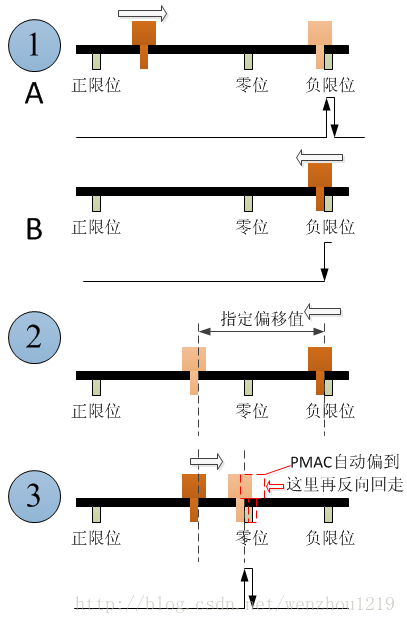
但是，我们需要面临的一个问题是，模组的最开始位置是不定的。同样用之前的图，最开始时候模组可能处于零位的正向也可能处于负向，甚至可能直接就在正负限位或零点上。这时候该怎么编写一个回零程序，在最开始不知道模组的位置的时候让模组正确的运动到零点呢？



毫无疑问，模组处于正向的时候必须向负向移动才能到零点触发，模组在负向的时候必须向正向移动才能到零点触发，但是它们的方向是完全相反的，这让我们编写运动程序很为难。为了解决这个问题，我们采用如下策略：

* 1.不管当前模组处于什么位置，都先向负向运动直到触发负限位
* 2.从负限位触发点开始向正向移动，PMAC是通过指定一个偏移值来完成的，所谓偏移值就是到达触发点后再偏移多少距离，这个偏移要求必须让模组移动到零位的正向

2.然后，模组负向运动直到准确停在零点触发点，当前标记为零点好了，现在基本策略定下来了。现在说说偏移值的问题。当我们运动到指定限位导致电平出现上下沿信号的时候，这时候PMAC接到中断，但是此时还在运动，不会准确的停在指定点上。对于这种问题，PMAC提供的解决方案是，既然不能准确的停止那就不让它准确的停在指定的触发点上，而是指定一个偏移值，当第一次上下沿触发的时候会记录当前的位置，这个当前的位置是可以准确记录的，这样运动到偏移一个指定值时是可以完成的，当然这个值是不能太小的，否则也停不准。那么要精确的停止在某一点怎么做呢，很简单，借鉴这种方法，既然你要跑过，那就让你先跑过，再反向回走偏移的值就行，这就是很多机床看回零时先过零点再反向走回来的原因，看运动平台在那不知所谓的来回捣鼓，其实干的就是这事儿。PMAC中指定偏移值为0，仔细观察PMAC会自己帮我们完成走过再回来的流程，这个需要了解一下以便于我们清楚整个回零轨迹为什么这样跑动和为什么使用偏移值。下面结合示意图看回零过程



可以看到:

* 1.负向运动到负限位直到触发，这里分为两种情况：

A.不在负限位的时候，直接负向移动直到上升沿触发

B.在负限位的时候，正向移动直到下降沿触发

注意一个使用上升沿，一个使用下降沿，是为了保证两种情况触发时模组位置是一样的，看示意图就懂了

* 2.按照步骤1指定的移动偏移值移动到零位的正向去，只要指定了偏移值，PMAC会自动完成这个偏移过程

### 3.从偏移后的位置负向移动，按照之前讲的先走过一点再反向走回从而精确停在零位，这个偏移可能很小，人眼不易观察出来。3.PMAC回零代码

一个单独的轴的典型回零程序如下：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45249305) [copy](http://blog.csdn.net/wenzhou1219/article/details/45249305)

1. ;程序功能：实现6轴回零运动。
2. ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
4. CLOSE ;确认所有缓冲区被关闭
5. &1
7. ;~~~~~~~~~~~~~~~~~~~~~~~~~  以下部分为运动程序  ~~~~~~~~~~~~~~~~~~~~~~~
8. OPEN PROG 20
9. CLEAR
11. ;6轴回零
13. If (M622 = 1)           ;负限位有信号
14. I623=50                 ;回零速度(Counts / msec),方向为正
15. I626=81920\*16           ;回零偏置,单位1/16 Count
16. I7123=2                 ;使用负限位为触发信号
17. I7122=10                ;捕捉触发信号的下降沿
18. Home 6                  ;回零
19. Dwell 20
20. Else                    ;负限位无信号
21. I623=-50                ;回零速度(Counts / msec),方向为负
22. I624=$120000            ;禁能限位
23. I626=81920\*16           ;回零偏置,单位1/16 Count
24. I7123=2                 ;使用负限位为触发信号
25. I7122=6                 ;捕捉触发信号的上升沿
26. Home 6                  ;回零
27. Dwell 20
28. EndIf
30. I623=-50                ;回零速度(Counts / msec),方向为负
31. I624=$100000            ;使能限位
32. I626=0                  ;回零偏置,单位1/16 Count
33. I7123=0                 ;使用原位信号为触发信号
34. I7122=7                 ;捕捉触发信号的上升沿及Z向的下降沿信号
35. Home 6                  ;回零
36. Dwell 50
38. CLOSE

代码中需要说明如下：

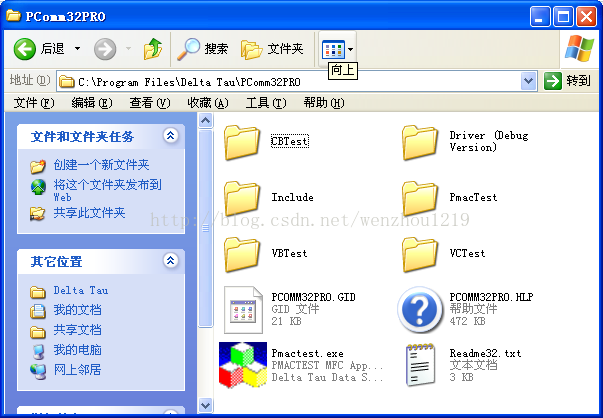
* 1.使用负限位做标记的时候，必须禁能限位，否则一旦模组运动到限位传感器处会自动停止当前电机
* 2.这里在回原位(零位)的时候，捕捉模式设为7，既支持编码器的上升沿也支持光栅尺的零位信号

3.在前两步时，速度可以较快，最后一步准确回零的时候速度要慢这里的回零代码笔者在各个项目中使用过，稳定可靠，具体的参数含义请自行查阅手册。

## **8.PMAC上位机-VC编程环境配置**

### 前面讲的知识都是在PE-WIN32中操作的，都是针对PMAC下位机编程的，现在开始讲解PMAC上位机——PC端如何与PMAC通信：给PMAC发送控制信号及处理PMAC的中断响应。本文先说PMAC VC编程的环境配置。1.文件夹结构

PMAC PC编程必须安装PComm32，安装完文件夹结构如下：

Pmactest.exe用于在PAMC按完成后测试当前安装是否成功,PmacTest是该程序的源码

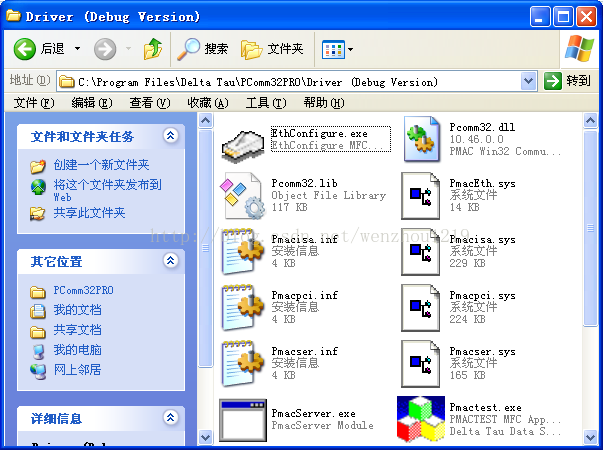
PCOMM32PRO.HLP是PComm32的编程帮助手册

VCTest和VBTest文件夹是上位机编程的小Demo

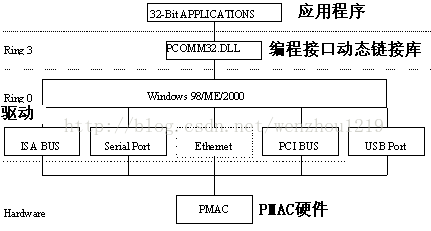
Driver (Debug Version)文件夹里是PMAC的各种驱动和动态链接库及其导入库等

### Include文件尾PMAC的接口头文件2.驱动结构模型

Driver (Debug Version)文件夹内容如下



这里提供了Pcomm32.dll动态链接库和导入库Pcomm32.lib，.sys都是PMAC的各种接口的驱动。PMAC的整个驱动结构模型如下图：



### 可以看到，我们针对动态链接库提供接口编程就行了3.代码演示

* 1.打开VCTest目录，会发现有myRuntime.h和myRuntime.cpp文件，连同上面的Pcomm32.dll一同拷贝进当前项目目录。
* 2.然后将myRuntime.h和myRuntime.cpp添加到当前项目中

3.就可以开始使用了最简单的测试程序如下:

新建MFC程序PmacConnect，PmacConnectDlg.cpp引入头文件

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45267543) [copy](http://blog.csdn.net/wenzhou1219/article/details/45267543)

1. //添加PComm操作库头文件
2. #include "myRuntime.h"

添加一个按钮响应，如下：

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45267543) [copy](http://blog.csdn.net/wenzhou1219/article/details/45267543)

1. void CPmacConnectDlg::OnConnect()
2. {
3. //动态加载Pcomm32.dll函数
4. if (NULL != OpenRuntimeLink())
5. {
6. AfxMessageBox(TEXT("加载Pcomm32.dll成功!"));
7. }
8. else
9. {
10. AfxMessageBox(TEXT("加载Pcomm32.dll失败!"));
11. return;
12. }
14. //打开指定卡号PMAC,这里打开0号卡
15. if (TRUE == OpenPmacDevice(0))
16. {
17. AfxMessageBox(TEXT("打开PMAC成功!"));
18. }
19. else
20. {
21. AfxMessageBox(TEXT("打开PMAC失败!"));
22. }
24. //关闭到PMAC连接
25. ClosePmacDevice(0);
27. //动态关闭Pcomm32.dll的使用
28. CloseRuntimeLink();

}  这里myRuntime.h和myRuntime.cpp封装了动态加载PComm.dll的函数，其实就是LoadLibrary的调用，感兴趣的可以自己看一下。这里只封装了常用部分的，需要的自己可以仿照它的形式从Include/Runtime.h中找到对应函数的原型来添加封装。

这里只是为了测试连接功能，实际编写程序的时候，我们需要在InitInstance或InitDialog时初始化连接，在程序退出的时候关闭连接。简单来说就是最开始打开一次和最后关闭一次连接，不需要每次操作时都要频繁的打开和关闭。代码演示[下载链接](http://download.csdn.net/detail/wenzhou1219/8630849)

## **9.PMAC上位机-上位机发送指令**

通常我们说PC控制其他硬件工作，指的是PC给相应的硬件发送指令来控制对应硬件。同时在发送完指令后可能会接受到相对应的反馈消息，告诉PC当前硬件的状态和参数等数据，这就是常见的PC和硬件通信。PMAC已经帮我们封装好了这通信过程，直接调用相关函数即可，这里讲通过上位机给PMAC发送指令控制电机运动或设置参数或返回当前指定参数值等。

PMAC提供PmacGetResponse和PmacGetResponseEX给PMAC发送指令。这一过程非常类似在Terminal中我们手动给PMAC发送指令，这里两个函数支持发送的命令也是在Terminal中输入的在线指令(Online Command)。

先看几个最简单示例额，在对话框上分别添加几个按钮和其响应函数，在响应函数中添加代码实现如下功能：

### 1.设置参数

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45787827) [copy](http://blog.csdn.net/wenzhou1219/article/details/45787827)

1. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
2. OpenRuntimeLink();
4. //打开PMAC卡连接
5. OpenPmacDevice(0);
7. TCHAR szRes[MAX\_PATH];
8. TCHAR szCmd[MAX\_PATH];
10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
11. lstrcpy(szCmd, "p1=1");
13. //发送指令和接受往回发的指令填充到szRes
14. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
15. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
17. //关闭PMAC卡连接
18. ClosePmacDevice(0);
20. //FreeLibrary
21. CloseRuntimeLink();

### 2.获取参数值

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45787827) [copy](http://blog.csdn.net/wenzhou1219/article/details/45787827)

1. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
2. OpenRuntimeLink();
4. //打开PMAC卡连接
5. OpenPmacDevice(0);
7. TCHAR szRes[MAX\_PATH];
8. TCHAR szCmd[MAX\_PATH];
10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
11. lstrcpy(szCmd, "p1");
13. //发送指令和接受往回发的指令填充到szRes
14. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
15. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
17. //显示当前参数值
18. AfxMessageBox(szRes);
20. //关闭PMAC卡连接
21. ClosePmacDevice(0);
23. //FreeLibrary
24. CloseRuntimeLink();

### 3.控制电机点动

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45787827) [copy](http://blog.csdn.net/wenzhou1219/article/details/45787827)

1. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
2. OpenRuntimeLink();
4. //打开PMAC卡连接
5. OpenPmacDevice(0);
7. TCHAR szRes[MAX\_PATH];
8. TCHAR szCmd[MAX\_PATH];
10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
11. lstrcpy(szCmd, "#6j+");
13. //发送指令和接受往回发的指令填充到szRes
14. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
15. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
17. //关闭PMAC卡连接
18. ClosePmacDevice(0);
20. //FreeLibrary
21. CloseRuntimeLink();

### 4.控制程序运行

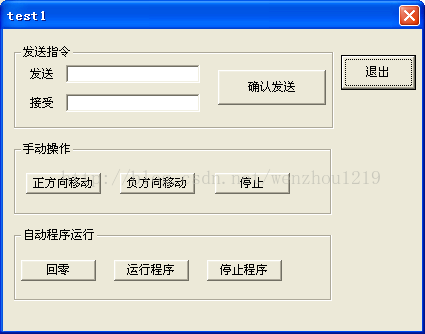
**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45787827) [copy](http://blog.csdn.net/wenzhou1219/article/details/45787827)

1. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
2. OpenRuntimeLink();
4. //打开PMAC卡连接
5. OpenPmacDevice(0);
7. TCHAR szRes[MAX\_PATH];
8. TCHAR szCmd[MAX\_PATH];
10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
11. lstrcpy(szCmd, "&1b20r");
13. //发送指令和接受往回发的指令填充到szRes
14. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
15. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
17. //关闭PMAC卡连接
18. ClosePmacDevice(0);
20. //FreeLibrary
21. CloseRuntimeLink();

注意这里的轴号和程序号和自己编写并下载的程序一致。

### 看到了吗，在/\*\*/注释间的是不是很熟悉，其实就是我们在PEWin32中Terminal窗口发送的指令，在/\*\*/之前和之后其实就是建立和释放操作环境的过程。很多时候，对于上位机程序来说，我们甚至只需要调用PmacGetResponse和PmacGetResponseEX就足以满足我们的需求了。5.演示程序

总体演示，下面我们把上面的常用功能集成到如下界面中：

对应的操作代码和上述演示代码一样，这里为了演示方便每次发送指令前都要建立环境和连接PMAC、发送指令、断开连接和释放环境，在实际项目中，只需要再程序实例化或窗口实例化时建立环境和连接PMAC，在窗口关闭或程序退出时断开连接和释放环境即可。代码如下：

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45787827) [copy](http://blog.csdn.net/wenzhou1219/article/details/45787827)

1. //发送指令
2. void CTest1Dlg::OnSend()
3. {
4. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
5. OpenRuntimeLink();
7. //打开PMAC卡连接
8. OpenPmacDevice(0);

11. TCHAR szRes[MAX\_PATH];
12. TCHAR szCmd[MAX\_PATH];
14. //获得输入的待发送指令字符串
15. GetDlgItemText(IDE\_COMMAND, szCmd, MAX\_PATH);
17. //发送指令和接受往回发的指令填充到szRes
18. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
20. //显示接受指令
21. SetDlgItemText(IDE\_RECEIVE, szRes);
23. //关闭PMAC卡连接
24. ClosePmacDevice(0);
26. //FreeLibrary
27. CloseRuntimeLink();
28. }
30. //正方向移动
31. void CTest1Dlg::OnMovePos()
32. {
33. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
34. OpenRuntimeLink();
36. //打开PMAC卡连接
37. OpenPmacDevice(0);

40. TCHAR szRes[MAX\_PATH];
41. TCHAR szCmd[MAX\_PATH];
43. lstrcpy(szCmd, "#6j+");
45. //显示输入的待发送指令字符串
46. SetDlgItemText(IDE\_COMMAND, szCmd);
48. //发送指令和接受往回发的指令填充到szRes
49. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
51. //显示接受指令
52. SetDlgItemText(IDE\_RECEIVE, szRes);
54. //关闭PMAC卡连接
55. ClosePmacDevice(0);
57. //FreeLibrary
58. CloseRuntimeLink();
59. }
61. //负方向移动
62. void CTest1Dlg::OnMoveNeg()
63. {
64. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
65. OpenRuntimeLink();
67. //打开PMAC卡连接
68. OpenPmacDevice(0);

71. TCHAR szRes[MAX\_PATH];
72. TCHAR szCmd[MAX\_PATH];
74. lstrcpy(szCmd, "#6j-");
76. //显示输入的待发送指令字符串
77. SetDlgItemText(IDE\_COMMAND, szCmd);
79. //发送指令和接受往回发的指令填充到szRes
80. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
82. //显示接受指令
83. SetDlgItemText(IDE\_RECEIVE, szRes);
85. //关闭PMAC卡连接
86. ClosePmacDevice(0);
88. //FreeLibrary
89. CloseRuntimeLink();
90. }
92. //停止
93. void CTest1Dlg::OnStopMove()
94. {
95. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
96. OpenRuntimeLink();
98. //打开PMAC卡连接
99. OpenPmacDevice(0);

102. TCHAR szRes[MAX\_PATH];
103. TCHAR szCmd[MAX\_PATH];
105. lstrcpy(szCmd, "#6k");
107. //显示输入的待发送指令字符串
108. SetDlgItemText(IDE\_COMMAND, szCmd);
110. //发送指令和接受往回发的指令填充到szRes
111. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
113. //显示接受指令
114. SetDlgItemText(IDE\_RECEIVE, szRes);
116. //关闭PMAC卡连接
117. ClosePmacDevice(0);
119. //FreeLibrary
120. CloseRuntimeLink();
121. }
123. //回零
124. void CTest1Dlg::OnHome()
125. {
126. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
127. OpenRuntimeLink();
129. //打开PMAC卡连接
130. OpenPmacDevice(0);

133. TCHAR szRes[MAX\_PATH];
134. TCHAR szCmd[MAX\_PATH];
136. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
137. /\*激活电机                                                              \*/
138. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
139. lstrcpy(szCmd, "#6j/#8j/");
141. //显示输入的待发送指令字符串
142. SetDlgItemText(IDE\_COMMAND, szCmd);
144. //发送指令和接受往回发的指令填充到szRes
145. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
147. //显示接受指令
148. SetDlgItemText(IDE\_RECEIVE, szRes);
150. Sleep(1000);

153. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
154. /\* 调用回零程序,注意这里自己的回零程序号                                \*/
155. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
156. lstrcpy(szCmd, "&1b20r");
158. //显示输入的待发送指令字符串
159. SetDlgItemText(IDE\_COMMAND, szCmd);
161. //发送指令和接受往回发的指令填充到szRes
162. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
164. //显示接受指令
165. SetDlgItemText(IDE\_RECEIVE, szRes);
167. //关闭PMAC卡连接
168. ClosePmacDevice(0);
170. //FreeLibrary
171. CloseRuntimeLink();
172. }
174. //运行程序
175. void CTest1Dlg::OnRunProg()
176. {
177. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
178. OpenRuntimeLink();
180. //打开PMAC卡连接
181. OpenPmacDevice(0);

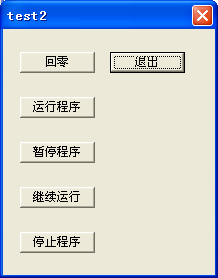
184. TCHAR szRes[MAX\_PATH];
185. TCHAR szCmd[MAX\_PATH];
187. lstrcpy(szCmd, "&1b40r");
189. //显示输入的待发送指令字符串
190. SetDlgItemText(IDE\_COMMAND, szCmd);
192. //发送指令和接受往回发的指令填充到szRes
193. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
195. //显示接受指令
196. SetDlgItemText(IDE\_RECEIVE, szRes);
198. //关闭PMAC卡连接
199. ClosePmacDevice(0);
201. //FreeLibrary
202. CloseRuntimeLink();
203. }
205. //停止程序
206. void CTest1Dlg::OnStopProg()
207. {
208. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
209. OpenRuntimeLink();
211. //打开PMAC卡连接
212. OpenPmacDevice(0);

215. TCHAR szRes[MAX\_PATH];
216. TCHAR szCmd[MAX\_PATH];
218. lstrcpy(szCmd, "&1b40a");
220. //显示输入的待发送指令字符串
221. SetDlgItemText(IDE\_COMMAND, szCmd);
223. //发送指令和接受往回发的指令填充到szRes
224. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
226. //显示接受指令
227. SetDlgItemText(IDE\_RECEIVE, szRes);
229. //关闭PMAC卡连接
230. ClosePmacDevice(0);
232. //FreeLibrary
233. CloseRuntimeLink();
234. }

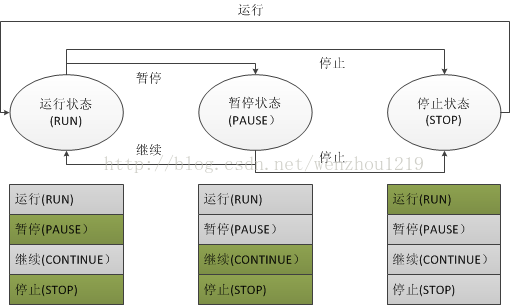
## **10.PMAC上位机-构造一个足够安全的工业设备程序**

### 对于编写工业程序来说，大多数时候我们首先保证的是安全和稳定，试想一个几吨的大型机床程序出了问题，可能一个数百万的零件就此报废，甚至危害加工人员的人身安全。本文就在PMAC中如何构造足够安全的工业设备程序给出一点实用方法。1.状态\_迁移图

先想想对于程序来说，怎样才算足够安全？对于每一种可能出现的操作都事先考虑到并给出对应的处理方案，这样的程序就是足够安全的。那么我们首先要做的就是列举每种可能出现的问题，其实前人们早就考虑到这个问题了，人们提出一个“状态\_迁移图”的概念，用来描述程序不同的状态和状态之间的跳变过程，如下图对于如下操作界面的程序：



这里为了简便，仅考虑程序运行的状态，可以列表如下：



程序包括三种状态：运行状态、暂停状态和停止状态。不同的状态间跳变，通过点击下方的运行、暂停、继续和停止按钮来完成。

上方的箭头标明不同的转态之间跳变关系，中间为完成某一个跳变必须完成的事件(点击的按钮)。

### 下方的按钮灰色表示当前按钮不能点击、绿色表示当前按钮能够点击。按钮能不能点击其实就是标明当前状态能否跳变到对应的点击后状态，即上方箭头标明的跳变图和下方灰色和绿色标明的按钮状态列表是等价的。2.编写安全程序方法

那么，我们编写安全程序就可以分别按照如下两种方法：

1.跳变图：当点击某一个按钮时，检查当期的状态能不能跳变到点击后的状态，如果不能则当前点击无效，避免误操作，对应上图的箭头跳变图。这种方法需要一个变量来记录当前的状态，根据不同的点击实时更新当前状态。

### 2.状态列表：在某一状态下，不能操作的按钮全部灰度化和禁用，只开放可发生状态跳变的按钮供用户点击，对应上图的按钮状态列表。这种方法需要在状态跳变后实时更改当前按钮状态。3.代码演示

#### 方法1.跳变图

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/17135755) [copy](http://blog.csdn.net/wenzhou1219/article/details/17135755)

1. void CTest2Dlg::OnRun()
2. {
3. TCHAR szRes[MAX\_PATH];
4. TCHAR szCmd[MAX\_PATH];
6. lstrcpy(szCmd, "&1b40r");
8. //判断跳变是否有效，进行相关操作和更新状态
9. if (MOVE\_STATE\_STOP == m\_moveState)
10. {
11. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
12. m\_moveState = MOVE\_STATE\_RUN;
13. }
14. else
15. {
16. AfxMessageBox(TEXT("当前不能进行该操作!"));
17. }
18. }
20. void CTest2Dlg::OnPause()
21. {
22. TCHAR szRes[MAX\_PATH];
23. TCHAR szCmd[MAX\_PATH];
25. lstrcpy(szCmd, "&1b40h");
27. //判断跳变是否有效，进行相关操作和更新状态
28. if (MOVE\_STATE\_RUN == m\_moveState)
29. {
30. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
31. m\_moveState = MOVE\_STATE\_PAUSE;
32. }
33. else
34. {
35. AfxMessageBox(TEXT("当前不能进行该操作!"));
36. }
37. }
39. void CTest2Dlg::OnContinue()
40. {
41. TCHAR szRes[MAX\_PATH];
42. TCHAR szCmd[MAX\_PATH];
44. lstrcpy(szCmd, "&1b40r");
46. //判断跳变是否有效，进行相关操作和更新状态
47. if (MOVE\_STATE\_PAUSE == m\_moveState)
48. {
49. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
50. m\_moveState = MOVE\_STATE\_RUN;
51. }
52. else
53. {
54. AfxMessageBox(TEXT("当前不能进行该操作!"));
55. }
56. }
58. void CTest2Dlg::OnStop()
59. {
60. TCHAR szRes[MAX\_PATH];
61. TCHAR szCmd[MAX\_PATH];
63. lstrcpy(szCmd, "&1b40a");
65. //判断跳变是否有效，进行相关操作和更新状态
66. if (MOVE\_STATE\_RUN==m\_moveState || MOVE\_STATE\_PAUSE==m\_moveState)
67. {
68. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
69. m\_moveState = MOVE\_STATE\_STOP;
70. }
71. else
72. {
73. AfxMessageBox(TEXT("当前不能进行该操作!"));
74. }

#### }  方法2.状态列表

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/17135755) [copy](http://blog.csdn.net/wenzhou1219/article/details/17135755)

1. void CTest2Dlg::OnRun()
2. {
3. TCHAR szRes[MAX\_PATH];
4. TCHAR szCmd[MAX\_PATH];
6. lstrcpy(szCmd, "&1b40r");
7. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
9. //更新状态,仅开放当前能操作的按钮
10. GetDlgItem(IDB\_RUN)->EnableWindow(FALSE);
11. GetDlgItem(IDB\_PAUSE)->EnableWindow(TRUE);
12. GetDlgItem(IDB\_CONTINUE)->EnableWindow(FALSE);
13. GetDlgItem(IDB\_STOP)->EnableWindow(TRUE);
14. }
16. void CTest2Dlg::OnPause()
17. {
18. TCHAR szRes[MAX\_PATH];
19. TCHAR szCmd[MAX\_PATH];
21. lstrcpy(szCmd, "&1b40h");
22. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
24. //更新状态,仅开放当前能操作的按钮
25. GetDlgItem(IDB\_RUN)->EnableWindow(FALSE);
26. GetDlgItem(IDB\_PAUSE)->EnableWindow(FALSE);
27. GetDlgItem(IDB\_CONTINUE)->EnableWindow(TRUE);
28. GetDlgItem(IDB\_STOP)->EnableWindow(TRUE);
29. }
31. void CTest2Dlg::OnContinue()
32. {
33. TCHAR szRes[MAX\_PATH];
34. TCHAR szCmd[MAX\_PATH];
36. lstrcpy(szCmd, "&1b40r");
37. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
39. //更新状态,仅开放当前能操作的按钮
40. GetDlgItem(IDB\_RUN)->EnableWindow(FALSE);
41. GetDlgItem(IDB\_PAUSE)->EnableWindow(TRUE);
42. GetDlgItem(IDB\_CONTINUE)->EnableWindow(FALSE);
43. GetDlgItem(IDB\_STOP)->EnableWindow(TRUE);
44. }
46. void CTest2Dlg::OnStop()
47. {
48. TCHAR szRes[MAX\_PATH];
49. TCHAR szCmd[MAX\_PATH];
51. lstrcpy(szCmd, "&1b40a");
52. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
54. //更新状态,仅开放当前能操作的按钮
55. GetDlgItem(IDB\_RUN)->EnableWindow(TRUE);
56. GetDlgItem(IDB\_PAUSE)->EnableWindow(FALSE);
57. GetDlgItem(IDB\_CONTINUE)->EnableWindow(FALSE);
58. GetDlgItem(IDB\_STOP)->EnableWindow(FALSE);
59. }

可以看到，方法1相对方法2，可以多了一个状态记录变量m\_moveState,可以随时查看当前状态，但是用户操作不能第一时间知道当前操作是否有效，必须点击后才会知道当前操作能不能进行。方法2，每一时刻用户可以进行的操作都很清楚，当时不能实时查看当前状态。

通常，我们使用第二种方法，当需要查询状态时，结合方法1，使用一个变量记录当前状态即可。

这里讲的都是在上位机改变状态，只有用户点击后程序状态才会改变，但是在实际中，当程序运行完毕(如这里的回零按钮实际上也应该加入到状态控制中，开始点击后程序为回零状态，所有按钮都必须灰度化，等待回零完毕后自动改变程序状态为运行状态)或特定动作发生(某一行为触发指定状态改变)，程序的状态也会发生改变，如何捕获这一状态改变呢？这是后续中断响应需要讲的内容。演示源代码[下载链接](http://download.csdn.net/detail/wenzhou1219/8711479)

## **11.PMAC上位机-设置参数**

在实际工业程序中，经常需要实时更改当前的参数设置。如对于PMAC运动程序来说，我们可能想在运动的时候在窗口界面改变当前的运动速度，这样就可以完成在PC端控制设备的一切行为。

### 1.思路

对于如下程序

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45819029) [copy](http://blog.csdn.net/wenzhou1219/article/details/45819029)

1. &1
2. CLOSE ;确认所有缓冲区被关闭
4. OPEN PROG 40
5. CLEAR
7. TA(200)
8. TS(20)
9. F(30)
10. INC
11. Linear
13. p1=1
14. While (p1 > 0)
15. X(20) Y(0)
16. Dwell 0
17. X(-20) Y(0)
18. Dwell 0
19. EndWhile
21. CLOSE

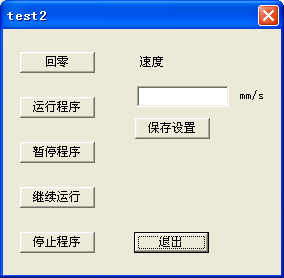
如果希望动态改变运动速度，那么将其改造如下：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45819029) [copy](http://blog.csdn.net/wenzhou1219/article/details/45819029)

1. &1
2. CLOSE ;确认所有缓冲区被关闭
4. OPEN PROG 40
5. CLEAR
7. TA(200)
8. TS(20)
9. F(30)
10. INC
11. Linear
13. p1=1
14. While (p1 > 0)
15. F(p2)
16. X(20) Y(0)
17. Dwell 0
18. F(p2)
19. X(-20) Y(0)
20. Dwell 0
21. EndWhile
23. CLOSE

### 这样我们在上位机通过发送命令动态改变p2的值就完成控制运动程序的速度功能了，这就最基本的参数设置思路。再复杂的程序也是通过这种变量控制的方式来完成的，这样的好处在于可以实现规定上下位机交互的变量，直接修改对应的变量值即可，对于自己通过其他方式生成的程序也可以完成参数控制。 2.代码演示

在之前界面上增加参数设置功能，控制运动速度，界面如下:



参数设置代码如下：

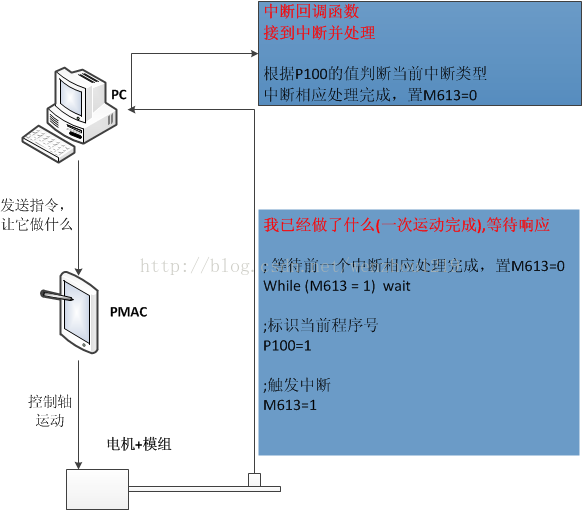
**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45819029) [copy](http://blog.csdn.net/wenzhou1219/article/details/45819029)

1. void CTest2Dlg::OnSaveSet()
2. {
3. TCHAR szSpeed[MAX\_PATH];
5. GetDlgItemText(IDE\_SPEED, szSpeed, MAX\_PATH);
6. SetSpeed(szSpeed);
8. }
10. void CTest2Dlg::SetSpeed(TCHAR\* szSpeed)
11. {
12. TCHAR szRes[MAX\_PATH];
13. TCHAR szCmd[MAX\_PATH];
15. sprintf(szCmd, "p2=%s", szSpeed);
16. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
17. }

注意这里的状态列表新增参数设置按钮控制。 演示源代码[下载链接](http://download.csdn.net/detail/wenzhou1219/8711801)

## **12.PMAC上位机-中断通信(下位机通知上位机)**

之前说过，上位机控制硬件都是通过发送指令进行的，这是单向的。那么反过来，下位机硬件给上位机发送指令怎么才能实现呢？对于PMAC来说，最常见的情况就是，当PMAC一次运动完成时通知当前PC程序运动已完成。1.下位机通知上位机原理如下图所示，



PC给PMAC发送指令控制它做什么，当运动程序完成后中断通知上位机，在中断的同时发送一个标识P100表明完成的程序号，这个标明到底是1号轴运动完，还是是2号轴运动完。在PC端有一个中断回调函数，下位机发送中断后，会自动跳转到该函数中，中断函数接受中断，根据标识P100来决定不同的情况不同的处理。

通俗来说，就是如下流程：

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45866239) [copy](http://blog.csdn.net/wenzhou1219/article/details/45866239)

1. PC->PMAC：发送指令，告诉它做什么         (&1b40r)
2. PMAC:做完了，等待PC处理完前一个通知           (While (M613 = 1)  wait)
3. PMAC->PC：某某事做完了，PC你就看着处理吧        (P100=1 M613=1)
5. PC：针对PMAC通知的事判断属于什么类型并处理        (根据P100判断)
6. PC->PMAC：处理完了，下位机可以发下一个通知消息了 (P100=0 M613=0)

### 如上图所示，M613这个是用硬件跳线配置的，指定当前中断响应通道，具体的设置[在这里](http://blog.csdn.net/wenzhou1219/article/details/41343637)。P100是我们指定的上下位机通信变量，这个是自己定义的，也可以是P1/P2/P3等。2.下位机代码

**[plain]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45866239) [copy](http://blog.csdn.net/wenzhou1219/article/details/45866239)

1. &1
2. CLOSE ;确认所有缓冲区被关闭
4. OPEN PROG 40
5. CLEAR
7. TA(200)
8. TS(20)
9. F(30)
10. INC
11. Linear
13. X(10) Y(40)
14. dwell 0
16. ;中断发送程序段
17. While (M613 = 1)  wait  ;等待上一次中断响应处理完成
18. P100=1                  ;标明当前完成的程序，可为0、1、2等自定义的值
19. M613=1                  ;向上位机发送中断

### CLOSE  3.上位机处理代码

注册中断处理函数：

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45866239) [copy](http://blog.csdn.net/wenzhou1219/article/details/45866239)

1. BOOL CInterruptDlg::ConnectPmac()
2. {
3. //链接Pcomm32.dll函数库,注意引入myRuntimeLink头文件
4. if (NULL == PmacRuntimeLink(PMAC\_NUM))
5. {
6. AfxMessageBox(TEXT("链接Pcomm32.dll函数库和打开PMAC卡连接失败!"));
7. return m\_bIsConnect = FALSE;
8. }
10. //中断函数绑定
11. if(  FALSE == PmacINTRFuncCallInit(PMAC\_NUM, InterruptFunc2, 0, 0xFF1F))
12. {
13. AfxMessageBox(TEXT("PMAC函数中断初始化失败"));
14. return m\_bIsConnect = FALSE;
15. }
17. return m\_bIsConnect = TRUE;
18. }

对应的中断函数处理如下：

**[cpp]** [view plain](http://blog.csdn.net/wenzhou1219/article/details/45866239) [copy](http://blog.csdn.net/wenzhou1219/article/details/45866239)

1. //中断处理
2. void WINAPI InterruptFunc2(DWORD msg, PINTRBUFFER pBuffer)
3. {
4. extern CInterruptApp theApp;
5. TCHAR szRes[MAX\_PATH];
6. TCHAR szCmd[MAX\_PATH];
8. if (pBuffer->dwInterruptType == ISR\_IR6)
9. {
10. if (1 == PmacGetVariable(PMAC\_NUM, 'P', 100, 0))
11. {
12. AfxMessageBox(TEXT("处理完成!"));
14. //重置中断
15. lstrcpy(szCmd, "M613=0 P100=0");
16. PmacGetResponse(0,szRes,MAX\_PATH,szCmd);
17. }
18. }

}  演示程序[源代码下载地址](http://download.csdn.net/detail/wenzhou1219/8720107)，在该程序中点击电机开始回零后，回零按钮灰度化。直到下位机完成回零发送中断给上位机程序，上位机程序判断后处理使能按钮。这个程序是最简单的中断处理程序，主要是为了阐明原理，再复杂的程序都和这个原理一样的，基于这个原理我们可以开发更复杂的上下位机交互程序。

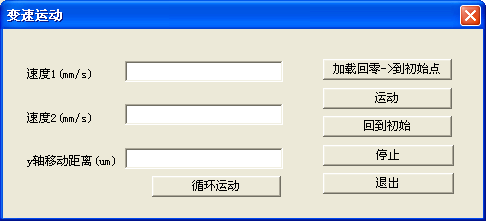
注意这里不能用默认安装完成后的runtime.h，原本的我试过有问题，必须用我放在程序源文件夹里的重写myRuntime.h。

更多PMAC代码参见[我的网站](http://jimwen.net/tag.php?tag=PMAC)，基本上都是这个系列涉及到的知识点。

到此为止，PMAC的使用系列基本上写完了，看完这一个系列，基本的PMAC编程和调试应该没有问题了，剩下的就是Read The Fuck Manual了。

## [**Pmac往复运动程序**](http://jimwen.net/blog.php?id=36)

程序界面如下

主要功能:

基本上是包含完整控制功能的最简单的PMAC上位机程序

1.初始化、回零、单次运动、循环运动、停止

2.上位机和PMAC卡交互

3.供PMAC新手学习

完整程序源代码[下载链接](http://pan.baidu.com/share/link?shareid=3764422426&uk=2936971062)