hi用到的串口通信编程方法有：使用通信控件、在高级语言中嵌入汇编以及使用ＡＰＩ函数。在这几种方法中，使用ＡＰＩ函数编写的串口通信程序最为高效、灵活。串口通信编程将用到三种ＡＰＩ函数——串口通信相关ＡＰＩ函数、多线程ＡＰＩ函数和实现消息机制的ＡＰＩ函数，下面将分别介绍这几种ＡＰＩ函数。

１ 与串口通信有关的ＡＰＩ函数  
　　Ｗｉｎｄｏｗｓ系统通信一般都以ＷＯＳＡ（Ｗｉｎｄｏｗｓ Ｏｐｅｎ Ｓｅｒｖｉｃｅｓ Ａｒｃｈｉｔｅｃｔｕｒｅ，即Ｗｉｎｄｏｗｓ开放式服务体系）模型为基础，在此模型中位于上层的应用程序通过调用各种通信ＡＰＩ与位于下层的设备驱动程序进行数据交换。下面将按一般串口通信程序的流程顺序介绍这些ＡＰＩ函数。  
１．１ 打开串口  
　　Ｗｉｎ３２操作系统把串口看作一个文件，因此打开串口我们将要用到ＣｒｅａｔｅＦｉｌｅ函数。该函数第一个参数指明要打开的文件名称，对串口操作来说，就是ＣＯＭ１、ＣＯＭ２等。第二个参数为读写模式设置，因为要对串口进行读写，所以该参数应设为ＧＥＮＥＲＩＣ＿ＲＥＡＤ｜ＧＥＮＥＲＩＣ＿ＷＲＩＴＥ。第三个参数值必须为０，表示不将串口与其他应用程序共享。第四个参数指向一个Ｓｅｃｕｒｉｔｙ Ａｔｔｒｉｂｕｔｅ结构，通常设为ＮＵＬＬ。第五个参数指定如何打开文件，在打开设备（串口是一种设备）时，此参数必须指定为ＯＰＥＮ＿ＥＸＩＳＴＩＮＧ。第六个参数指定文件属性及相关标　　志，但是对于串行口，唯一有意义的设置是ＦＩＬＥ＿ＦＬＡＧ＿ＯＶＥＲＬＡＰＰＥＤ或０；最后一个参数必须为ＮＵＬＬ。若该函数打开串口成功，则返回创建的句柄，该句柄供随后对串行口的设置、读写等操作用；否则返回ＩＮＶＡＬＩＤ＿ＨＡＮＤＬＥ＿ＶＡＬＵＥ。  
１．２ 设置串口  
　　串口打开后，即可进行一系列初始化设置。最基本的初始化设置将通过ＧｅｔＣｏｍｍＳｔａｔｅ和ＳｅｔＣｏｍｍＳｔａｔｅ函数来实现。先调用ＧｅｔＣｏｍｍＳｔａｔｅ函数获取当前串口配置填充设备控制块（ＤＣＢ），然后将ＤＣＢ结构中几个重要参数如波特率、数据位、停止位、校验位改成符合实际设计要求的值，最后用ＳｅｔＣｏｍｍＳｔａｔｅ将刚刚所做的改动重新设置串口。串口Ｉ／Ｏ缓冲区的大小用ＳｅｔｕｐＣｏｍｍ函数设置。通信速率越高，缓冲区应设置得越大，但不能超出设备驱动程序所能 处理的范围。另一个很重要的设置是串口超时设置。通信中因未知原因将出现不可预测的事件，譬如：接收数据过程中突然被中断，或者发送数据突然停止等。如果不认真对待，这些情况可能会引起Ｉ／Ｏ线程挂起或者线程被无限阻塞。Ｗｉｎｄｏｗｓ对于这类问题提供了安全措施，它可通过超时设置来决定通信是否异常并作相应处理，因此超时设置在串行通信中显得尤为重要。超时设置过程分为两步，首先设置Ｃｏｍｍｔｉｍｅｏｕｔｓ结构中的五个成员，然后调用ＳｅｔＣｏｍｍＴｉｍｅｏｕｔｓ函数设置超时值。Ｃｏｍｍｔｉｍｅｏｕｔｓ结构的五个成员分别是：读串口间隔超时、读串口总超时乘数、读串口总超时常数（ｍｓ）、写串口总超时乘数、写串口总超时常数（ｍｓ）。  
１．３ 读写串口  
　　设置工作完成后，即可用ＲｅａｄＦｉｌｅ和ＷｒｉｔｅＦｉｌｅ对串口进行读写操作。在调用读写操作函数之前，应先用ＣｌｅａｒＣｏｍｍＥｒｒｏｒ函数清除错误标志和获取当前串口状态。读写操作分为同步和重叠Ｉ／Ｏ（异步）。同步执行时，函数直到操作完成后才返回；重叠Ｉ／Ｏ操作时，即使操作尚未完成，调用的函数也会立即返回，费时的Ｉ／Ｏ操作在后台进行。可见，同步操作线程被阻塞，效率低，只能用在对通信要求比较低的场合，我们一般用到的都是效率较高的重叠Ｉ／Ｏ操作。前面提到的ＣｒｅａｔｅＦｉｌｅ函数第六个参数设置为ＦＩＬＥ＿ＦＬＡＧ＿ＯＶＥＲＬＡＰＰＥＤ即可指定Ｒｅａｄ Ｆｉｌｅ和ＷｒｉｔｅＦｉｌｅ函数为重叠Ｉ／Ｏ执行。使用重叠Ｉ／Ｏ还需为读写函数指定一个Ｏｖｅｒｌａｐｐｅｄ结构，该结构有五个数据成员，对串口通信来说，其中的Ｏｆｆｓｅｔ和ｈＥｖｅｎｔ成员是很重要的。Ｏｆｆｓｅｔ指示文件指针偏移量，在重叠Ｉ／Ｏ操作时系统不能自动维护文件指针，所以要靠Ｏｆｆｓｅｔ在程序中手动调整文件指针。而ｈＥｖｅｎｔ标志读写操作是否完成。若操作完成，

　　则将ｈＥｖｅｎｔ置为信号态；否则即置为非信号态。最后要说明的是，在重叠Ｉ／Ｏ操作时，读写函数返回值是ＦＡＬＳＥ并不能说明操作失败，应该调用ＧｅｔＬａｓｔＥｒｒｏｒ函数分析返回结果。如果 此时ＧｅｔＬａｓｔＥｒｒｏｒ函数返回值是ＥＲＲＯＲ＿ＩＯ＿ＰＥＮＤＩＮＧ，则说明操作未完成（并不是操作失败）。我们将用等待函数来等待操作的完成。典型的两个等待函数有ＷａｉｔＦｏｒＳｉｎｇｌｅＯｂｊｅｃｔ和ＧｅｔＯｖｅｒｌａｐｐｅｄＲｅｓｕｌｔ。函数的相同之处为都是等待读写操作指定的Ｏｖｅｒｌａｐｐｅｄ结构ｈＥｖｅｎｔ成员置为信号态（即代表操作完成）；不同之处是ＷａｉｔＦｏｒＳｉｎｇｌｅＯｂｊｅｃｔ是可设置超时，但无法得到重叠Ｉ／Ｏ操作的结果，ＧｅｔＯｖｅｒｌａｐｐｅｄＲｅｓｕｌｔ用来得到重叠Ｉ／Ｏ操作的结果，但无法设置超时。因此，我们经常两者结合起来使用，在用ＷａｉｔＦｏｒＳｉｎｇｌｅＯｂｊｅｃｔ等待操作结束后，用ＧｅｔＯｖｅｒｌａｐ－ｐｅｄＲｅｓｕｌｔ得到操作结果。  
１．４ 关闭串口  
　　串行口是非共享资源，某应用程序打开串行口后，即独占该资源，使其它应用程序无法再访问，直到该应用程序释放串口。所以对串口操作完成后，一定要关闭串口。关闭串口使用ＣｌｏｓｅＨａｎｄｌｅ函数，该函数唯一参数即为用ＣｒｅａｔｅＦｉｌｅ打开串口时所创建的句柄。  
２ 多线程ＡＰＩ函数  
　　Ｗｉｎｄｏｗｓ是多线程（ｍｕｌｔｉ－ｔｈｒｅａｄｅｄ）、抢先多任务的（ｐｒｅｅｍｐｔｉｂｌｅ ）。Ｗｉｎｄｏｗｓ中，一个可执行程序的运行时刻实例称为进程（ｐｒｏｃｅｓｓ）。一个进程可以有多个线程（ｔｈｒｅａｄ），Ｗｉｎｄｏｗｓ是按照线程分配ＣＰＵ时间片的，而分配的机制就是抢先多任务方式。  
　　对于读写串口这种耗时的工作，使用多线程技术，创建辅助线程来管理串口是一个常用的方案。这样在进行串口读写的同时，能对读入的数据进行处理。如果使用单线程，就需要等待串口读写操作完成，整个进程都被阻塞。而使用多线程就可以避免这种情况。  
　　多线程也会带来一些新的问题，其中的一个问题就是线程的同步，如果同步问题解决不好，程序的稳定性会受到很大的影响。通常用到的几种线程同步的方法有互斥体对象（Ｍｕｔｅｘ）、利用信号（Ｓｅｍａｐｈｏｒｅ）、利用事件对象（Ｅｖｅｎｔ）和设置临界区（Ｃｒｉｔｉｃａｌ Ｓｅｃｔｉｏｎ）。笔者在实际应用中使用的是事件对象结合Ｗｉｎｄｏｗｓ消息机制使线程同步，收到了很好的效果。  
　　创建线程函数为ＣｒｅａｔｅＴｈｒｅａｄ，用ＳｕｓｐｅｎｄＴｈｒｅａｄ和ＲｅｓｕｍｅＴｈｒｅａｄ函数来挂起和唤醒线程。创建事件函数为ＣｒｅａｔｅＥｖｅｎｔ，用ＳｅｔＥｖｅｎｔ和ＲｅｓｅｔＥｖｅｎｔ函数来将事件置为信号态和非信号态，以此来同步线程。串口通信的辅助线程管理经常还要用到ＳｅｔＣｏｍｍＭａｓｋ和ＷａｉｔＣｏｍｍＥｖｅｎｔ函数。ＳｅｔＣｏｍｍＭａｓｋ用来指定一系

　　列事件监视串口，比如监视串口是否有数据收到；ＷａｉｔＣｏｍｍＥｖｅｎｔ则用来等待指定的事件发生。笔者在实际应用中，就是在辅助线程中用ＳｅｔＣｏｍｍＭａｓｋ指定串口监视接收数据事件，然后用ＷａｉｔＣｏｍｍＥｖｅｎｔ等到串口真的接收到数据时，用ＰｏｓｔＭｅｓｓａｇｅ发出消息通知主线程，由主线程处理接收到的数据。  
　　３　实现消息机制的ＡＰＩ函数  
　　Ｗｉｎｄｏｗｓ是一个消息驱动操作系统，简单的说 消息就是指通过输入设备向程序发出的指令以要求

　　其执行某个操作。具体的操作由消息处理函数实现。用户可以自定义消息在线程之间传递。把ＷＭ＿ＵＳＥＲ（它的值等于０×０４００）当作基数，然后顺序地去加序号，譬如：  
　　ＷＭ＿ＣＯＭＭＮＯＴＩＦＹ ｅｑｕ ＷＭ＿ＵＳＥＲ＋１００ｈ（小于ＷＭ＿ＵＳＥＲ的值是Ｗｉｎｄｏｗｓ系统的保留值，大于该值留给用户来使用）。  
　　前一节已经提到在串口通信编程中对消息机制的利用，这里将继续说明怎样实现消息机制。由于笔者使用的编程工具是Ｂｏｒｌａｎｄ公司的Ｃ＋＋ Ｂｕｉｌｄｅｒ（ＢＣＢ），因此对于消息机制的实现有其特殊之处。在ＢＣＢ中实现消息的方法有三种：使用消息映射 Ｍｅｓｓａｇｅ Ｍａｐ 重载ＴＯｂｊｅｃｔ的Ｄｉｓｐａｔｃｈ虚成员函数；重载ＴＣｏｎｔｒｏｌ的ＷｎｄＰｒｏｃ方法；重载Ａｐｐｌｉ

　　ｃａｔｉｏｎ的ＯｎＭｅｓｓａｇｅ方法。其中以第三种方法最快，因为一般情况下，ＢＣＢ会为每个程序自动生成一个ＴＡｐｐｌｉｃａｔｉｏｎ类的实例，消息到达ＢＣＢ程序时，最先得到它们的就是ＴＡｐｐｌｉｃａｔｉｏｎ对象。经由ＴＡｐｐｌｉｃａｔｉｏｎ之后，才传递给Ｆｏｒｍ的。前两种方法都是重载ＴＦｏｒｍ的方法，显然比直接重载Ａｐｐｌｉｃａｔｉｏｎ的ＯｎＭｅｓｓａｇｅ方法要晚一些收到消息。我们要做的只是定义好自己的消息处理函数：  
　　ｖｏｉｄ ＿＿ｆａｓｔｃａｌｌ ＴＦｏｒｍ１ ＭｙＯｎＭｅｓｓａｇｅ ｔａｇＭＳＧ ＆Ｍｓｇ  
ｂｏｏｌ ＆Ｈａｎｄｌｅｄ   
　　   
　　ＴＭｅｓｓａｇｅ Ｍｅｓｓａｇｅ   
　　ｓｗｉｔｃｈ（Ｍｓｇ．ｍｅｓｓａｇｅ）  
　　   
　　ｃａｓｅ ＷＭ＿ＣＯＭＭＮＯＴＩＦＹ   
　　Ｍｅｓｓａｇｅ．Ｍｓｇ＝Ｍｓｇ．ｍｅｓｓａｇｅ；  
　　Ｍｅｓｓａｇｅ．ＷＰａｒａｍ＝Ｍｓｇ．ｗＰａｒａｍ；  
　　Ｍｅｓｓａｇｅ．ＬＰａｒａｍ＝Ｍｓｇ．ｌＰａｒａｍ；  
　　／／此处添加处理该消息的代码  
　　Ｈａｎｄｌｅｄ＝ｔｒｕｅ；  
　　ｂｒｅａｋ；  
　　　　   
　　   
　　然后在窗口创建时用自定义的消息处理函数重载Ａｐｐｌｉｃａｔｉｏｎ的ＯｎＭｅｓｓａｇｅ方法：  
　　ｖｏｉｄ ＿＿ｆａｓｔｃａｌｌ ＴＦｏｒｍ１ ＦｏｒｍＣｒｅａｔｅ（ＴＯｂｊｅｃｔ Ｓｅｎｄｅｒ）  
　　   
　　Ａｐｐｌｉｃａｔｉｏｎ－＞ＯｎＭｅｓｓａｇｅ ＝ ＭｙＯｎＭｅｓｓａｇｅ；  
　　   
　　这样就可以在程序中收到自定义的消息并作出相应处理。  
　　值得注意的是，使用Ａｐｐｌｉｃａｔｉｏｎ－＞ＯｎＭｅｓｓａｇｅ并不能捕获非队列消息，它无法捕获使用ＳｅｎｄＭｅｓｓａｇｅ直接发送给窗口的消息，这是因为其不通过消息队列。但可以使用另一个发送消息的ＡＰＩ函数——ＰｏｓｔＭｅｓｓａｇｅ，该函数发出的消息是队列消息。  
　　４　结束语  
　　利用多线程、消息机制和重叠Ｉ／Ｏ的ＡＰＩ函数进行串口编程的方法，可实现串口通信的实时高效，为开发Ｗｉｎｄｏｗｓ系统下串口驱动程序提供了有益参考。

C语言的串口通信源程序

//串口中断服务程序，仅需做简单调用即可完成串口输入输出的处理  
//出入均设有缓冲区，大小可任意设置。  
//可供使用的函数名:  
//char getbyte(void);从接收缓冲区取一个byte,如不想等待则在调用前检测inbufsign是否为1。  
//getline(char idata \*line, unsigned char n); 获取一行数据回车结束，必须定义最大输入字符数  
//putbyte(char c);放入一个字节到发送缓冲区  
//putbytes(unsigned char \*outplace,j);放一串数据到发送缓冲区，自定义长度  
//putstring(unsigned char code \*puts);发送一个定义在程序存储区的字符串到串口  
//puthex(unsigned char c);发送一个字节的hex码，分成两个字节发。  
//putchar(uchar c,uchar j);输出一个无符号字符数的十进制表示，必须标示小数点的位置,自动删除前面无用的零  
//putint(uint ui,uchar j);输出一个无符号整型数的十进制表示，必须标示小数点的位置,自动删除前面无用的零  
//delay(unsigned char d); 延时n x 100ns  
//putinbuf(uchar c);人工输入一个字符到输入缓冲区  
//CR;发送一个回车换行  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
#include <w77e58.h>   
#define uchar unsigned char  
#define uint unsigned int  
#define OLEN 32 /\* size of serial transmission buffer \*/  
idata unsigned char outbuf[OLEN]; /\* storage for transmission buffer \*/  
unsigned char idata \*outlast=outbuf; //最后由中断传输出去的字节位置   
unsigned char idata \*putlast=outbuf; //最后放入发送缓冲区的字节位置   
#define ILEN 12 /\* size of serial receiving buffer \*/  
idata unsigned char inbuf[ILEN];   
unsigned char idata \*inlast=inbuf; //最后由中断进入接收缓冲区的字节位置  
unsigned char idata \*getlast=inbuf; //最后取走的字节位置   
bit outbufsign0; //最后一个数据覵BUF发完标志 发完＝0  
bit outbufsign; //输出缓冲区非空标志 有=1   
bit inbufsign; //接收缓冲区非空标志 有=1  
bit inbufful; //输入缓冲区满标志 满=1  
#define CR putstring("\r\n") //CR=回车换行</P><P>//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//延时n x 100ns   
void delay(unsigned char d) //在源程序开头定义是否用w77e58或22。1184M晶振  
{unsigned char j;  
do{ d--; </P><P>//110592 & 89c52  
#ifndef cpuw77e58   
#ifndef xtal221184   
j=21; //k=38 cpu80320 100us k=21 cpu 8052  
#else   
j=42;   
#endif  
#else  
#ifndef xtal221184  
j=38;  
#else  
j=76;   
#endif  
#endif

do {j--;} while(j!=0);  
}while(d!=0);  
}   
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
//放入一个字节到发送缓冲区</P><P>  
putbyte(char c)   
{   
uchar i,j;  
ES=0; /\*暂停串行中断，以免数据比较时出错? \*/  
//if (outlast=putlast)  
while ( (((outlast-putlast)==2) && (outlast > putlast )) || ((outlast < putlast) && (OLEN-(putlast-outlast)==2)))  
{   
ES=1; c++;c--;ES=0;  
// i=(0-TH1);  
// do{i--;j=39; do {j--;}while(j!=0); }while(i!=0); //i=39  
}   
\*putlast=c; //放字节进入缓冲区  
Putlast++; //发送缓冲区指针加一  
if (putlast==outbuf+OLEN) putlast=outbuf; //指针到了顶部换到底部  
outbufsign=1;  
if (!outbufsign0)   
{  
outbufsign0=1;TI=1;   
} //缓冲区开始为空置为有，启动发送  
ES=1;  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//放一串数据到发送缓冲区  
putbytes(unsigned char \*outplace,unsigned char j)   
{ int i;  
for(i=0;i<j;i++)  
{putbyte(\*outplace);  
outplace++;  
}  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//输出一个无符号字符数的十进制表示，必须标示小数点的位置,自动删除前面无用的零  
//例如putchar(0x32,2),输出"4.8".  
//putchar(0x32,3),输出"0.48".  
//putchar(0x32,1),输出"48".  
putchar(uchar c,uchar j)  
{uchar idata free[4];uchar data i;   
i=0;   
free[i++]=(c/100+0x30);  
if (j==3) free[i++]='.';   
free[i++]=(c%100)/10+0x30;  
if (j==2) free[i++]='.';  
if (j==2 && free[i-3]==0x30) free[i-3]=0x20;  
free[i++]=(c%10)+0x30;  
if (j==1 && free[i-3]==0x30) free[i-3]=0x20;  
if (j==1 && free[i-3]==0x20 && free[i-2]==0x30) free[i-2]=0x20;   
putbytes(free,i);  
}</P><P>//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//输出一个无符号整型数的十进制表示，必须标示小数点的位置,自动删除前面无用的零  
putint(uint ui,uchar j)  
{uchar idata free[6];  
uchar data i;  
i=0;  
free[i++]=(ui/10000+0x30);  
if (j==5) free[i++]='.';  
free[i++]=((ui%10000)/1000+0x30);  
if (j==4) free[i++]='.';  
if (j==4 && free[i-3]==0x30) free[i-3]=0x20;  
free[i++]=((ui%1000)/100+0x30);  
if (j==3) free[i++]='.';  
if (j==3 && free[i-4]==0x30) free[i-4]=0x20;  
if (j==3 && free[i-4]==0x20 && free[i-3]==0x30) free[i-3]=0x20;  
free[i++]=((ui%100)/10+0x30);  
if (j==2) free[i++]='.';  
if (j==2 && free[i-5]==0x30) free[i-5]=0x20;  
if (j==2 && free[i-5]==0x20 && free[i-4]==0x30) free[i-4]=0x20;  
if (j==2 && free[i-5]==0x20 && free[i-4]==0x20 && free[i-3]==0x30) free[i-3]=0x20;  
free[i++]=(ui%10+0x30);  
if (j==1 && free[i-5]==0x30) free[i-5]=0x20;  
if (j==1 && free[i-5]==0x20 && free[i-4]==0x30) free[i-4]=0x20;  
if (j==1 && free[i-5]==0x20 && free[i-4]==0x20 && free[i-3]==0x30) free[i-3]=0x20;  
if (j==1 && free[i-5]==0x20 && free[i-4]==0x20 && free[i-3]==0x20 && free[i-2]==0x30) free[i-2]=0x20;  
putbytes(free,i);  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//发送一个定义在程序存储区的字符串到串口  
putstring(unsigned char \*puts)  
{for (;\*puts!=0;puts++) //遇到停止符0结束  
putbyte(\*puts);  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//发送一个字节的hex码，分成两个字节发。  
unsigned char code hex\_[]={"0123456789ABCDEF"};   
puthex(unsigned char c)  
{int ch;  
ch=(c>>4)&0x0f;  
putbyte(hex\_[ch]);  
ch=c&0x0f;  
putbyte(hex\_[ch]);  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//从接收缓冲区取一个byte,如不想等待则在调用前检测inbufsign是否为1。  
uchar getbyte (void)   
{ char idata c ;  
while (!inbufsign); //缓冲区空等待  
ES=0;   
c= \*getlast; //取数据   
getlast++; //最后取走的数据位置加一  
inbufful=0; //输入缓冲区的满标志清零   
if (getlast==inbuf+ILEN) getlast=inbuf; //地址到顶部回到底部  
if (getlast==inlast) inbufsign=0; //地址相等置接收缓冲区空空标志，再取数前要检该标志  
ES=1;  
return (c); //取回数据  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//接收一行数据，必须定义放数据串的指针位置和大小 del=0x7f,backspace=0x08,cr=0x0d,lf=0x0a  
void getline (uchar idata \*line, unsigned char n)   
{ unsigned char cnt = 0; //定义已接收的长度  
Char c;  
do   
{  
if ((c = getbyte ()) == 0x0d) c = 0x00; //读一个字节，如果是回车换成结束符  
if (c == 0x08 || c == 0x7f) //BACKSPACE 和 DEL 的处理  
{ if (cnt != 0) //已经输入退掉一个字符  
{ cnt--; //总数目减一  
line--; //指针减一  
Putbyte (0x08); //屏幕回显的处理  
Putbyte (' ');  
Putbyte (0x08);  
}  
}  
else   
{  
putbyte (\*line = c); //其他字符取入，回显  
line++; //指针加一  
Cnt++; //总数目加一  
}  
}   
while (cnt < n - 1 && c != 0x00 && c!=0x1b); //数目到了，回车或ESC停止   
\*line = 0; //再加上停止符0  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//人工输入一个字符到输入缓冲区  
putinbuf(uchar c)  
{ES=0; if(!inbufful)  
{\*inlast= c; //放入数据   
inlast++; //最后放入的位置加一   
if (inlast==inbuf+ILEN) inlast=inbuf;//地址到顶部回到底部  
if (inlast==getlast) inbufful=1; //接收缓冲区满置满标志  
inbufsign=1;   
}  
ES=1;  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//串口中断处理  
serial () interrupt 4   
{ if (TI )   
{ TI = 0;   
if (outbufsign)   
//if (putlast==outlast) outbufsign=0;  
//else   
{SBUF=\*outlast; //未发送完继续发送  
outlast++; //最后传出去的字节位置加一  
if (outlast==outbuf+OLEN) outlast=outbuf;//地址到顶部回到底部  
if (putlast==outlast) outbufsign=0; //数据发送完置发送缓冲区空标志   
}   
else outbufsign0=0;  
}  
if (RI)   
{ RI = 0;   
if(!inbufful)  
{  
\*inlast= SBUF; //放入数据   
inlast++; //最后放入的位置加一   
inbufsign=1;   
if (inlast==inbuf+ILEN) inlast=inbuf;//地址到顶部回到底部  
if (inlast==getlast) inbufful=1; //接收缓冲区满置满标志

}  
}  
}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//串口初始化 0xfd=19200,0xfa=9600,0xf4=4800,0xe8=2400,0xd0=1200  
serial\_init () {  
SCON = 0x50; /\* mode 1: 8-bit UART, enable receiver \*/  
TMOD |= 0x20; /\* timer 1 mode 2: 8-Bit reload \*/  
PCON |= 0x80; TH1 = 0xfA;//fa, //baud\*2 /\* reload value 19200 baud \*/  
TR1 = 1; /\* timer 1 run \*/  
ES = 1; REN=1; EA=1; SM2=1; //SM2=1时收到的第9位为1才置位RI标志  
TMOD |=0x01; //th1 auto load 2X8,th0 1X16  
TH0=31; TL0=0; //X 32 =1S  
TR0=1; //ET0=1;</P><P>}  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//测试用主函数

void main(void)  
{char c;  
idata unsigned char free[16];   
unsigned char idata \*freep=free;   
serial\_init();</P><P>  
putstring("jdioptuejls;j;klj");  
delay(10);</P><P> while(1)   
{ putstring("com is ready! ");}  
c=getbyte();  
putbyte(0x20);  
puthex(c);  
switch(c)  
{case 'r':  
putbytes(inbuf,ILEN);   
break;  
case 'g':  
getline(freep,10);  
putbyte(0x20);  
putstring(freep);   
break;  
default:  
putbyte(c);  
// }  
}  
}

　　在当今,流行的编程软件种类繁多，它们编程方便、易于维护，但是在与硬  
件直接打交道和编制系统软件时却束手无策，于是C语言就有了用武之地。C语言  
作为汇编语言与高级语言之间的一种过渡语言，兼有汇编语言的高效和高级语言  
的方便。   
　　在通讯中，为了保证行运安全可靠，标准的串行口必须具有许多握手信号和  
状态信息。这是因为通讯的各个计算机CPU速度不一样（这会导致“错帧”）以  
及发送机发送数据速度比接收机接收速度快（这会导致“过冲”）。为解决这个  
问题，我们采用一个简单的握手信号，即发送机每次仅发送半个字节（低4位）  
的数据，而另外半个字节（高4位）则用来传送信息。我们可以对信息位（高4位）  
进行如下简单的编码：

0H：发送的是新的半个字节数据  
1H：重新发送上次传送错误的数据  
2H：文件名结束  
3H：文件结束

这样，每当发送机发送一个字节以后，就等待接受机发回送信号，这回送信号就  
是发送机发送过来的那个字节。发送机接收到回送信号后，把它与刚发送的字节  
相比较，假如相同，就发送新的半个字节，否则就重新发送。新数据与旧数据通  
过信息位来区分。下面就是用C语言编写控制串行口的程序。

#include "dos.h"  
#include "stdlib.h"  
#include "stdio.h"  
#define PORT 0  
void SendFile(char \*fname); /\* 发送文件\*/  
void Send(int s); /\*发送一个字节\*/  
void SendFileName(char \*fname); /\*发送文件名\*/  
void ReceiveFile(); /\*接收文件\*/  
void GetFileName(char \*f); /\*接收文件名\*/  
void InitPort(int port,unsigned char para); /\*初始化端口\*/  
void SendPort(int port,char c); /\*端口发送\*/  
int ReadPort(int port); /\*读端口字节\*/  
int CheckState(int port); /\*检查端口状态\*/  
int Receive(int port,int \*G); /\*接收一个字节\*/  
main(int argc,char \*argv[])  
{  
if(argc<2){  
printf("Please input R(receive) or S(sent) parametre:");  
exit(1);  
}  
InitPort(PORT,231);  
if(\*argv[1]==''''S'''') /\*检查选择的有效性\*/  
SendFile(argv[2]);  
else if(\*argv[1]==''''R'''')  
ReceiveFile();  
else{  
printf("Error parament.Please input again.");  
exit(1);  
}  
}  
void SendFile(char \*fname)  
{  
FILE \*fp;  
int ch,s;  
if((fp=fopen(fname,"rb"))==NULL)  
{  
printf("Can''''t open the file.\n");  
exit(1);  
}  
SendFileName(fname);  
do{  
ch=(int)getc(fp);  
if(ferror(fp)){  
printf("Error reading file.\n");  
break;  
}  
s=ch%16; /\*取文件中一个字节的低4位\*/  
Send(s);  
s=ch/16; /\*取文件中一个字节的高4位\*/  
Send(s);  
}while(!feof(fp));  
s=46; /\*发送文件结束信息\*/  
Send(s);  
Send(s);  
fclose(fp);  
}  
void Send(s)  
int s;  
{  
int G;  
SendPort(PORT,s);  
G=ReadPort(PORT); /\*等待握手信号\*/  
if(s!=G)  
s=s+16;  
do{  
SendPort(PORT,s);  
G=ReadPort(PORT);/\*等待握手信号\*/  
}while(s!=G);  
}  
void SendFileName(fname)  
char \*fname;  
{  
int s,ch;  
printf("Now transmit the file.Please wait...");  
while(\*fname){  
ch=(int)fname++;  
s=ch%16; /\*取文件名中一个字节的低4位\*/  
Send(s);  
s=ch/16;  
Send(s); /\*取文件名中一个字节的低4位\*/  
}  
s=32; /\*发送文件名结束标志\*/  
Send(s);  
Send(s);  
}  
void ReceiveFile(){  
FILE \*fp;  
char ch;  
int G1,G2,G3;  
char fname[15];  
GetFileName(fname);  
printf("Receiving file %s.\n",fname);  
remove(fname);  
if((fp=fopen(fname,"wb"))==NULL)  
{  
printf("Can''''t open output file.\n");  
exit(1);  
}  
/\*循环为检测每次接受的数据是否为新数据，假如不是，\*/  
/\*则用此次接收的数据覆盖上次接收的数据\*/  
G1=ReadPort(PORT);  
G2=Receive(PORT,&G1);  
do{   
G3=Receive(PORT,&G2);  
ch=(char)(G1%16+G2\*16);/\*恢复分开的数据，组合高4位和低4位\*/  
putc(ch,fp);  
if(ferror(fp)){  
printf("\nError writing file.");  
exit(1);  
}  
G2=Receive(PORT,&G3);  
G1=G3;  
}while(G1/16!=48);  
printf("\nTransmit finished.");  
fclose(fp);  
}  
int Receive(port,G)  
int port,\*G;  
{  
int GM;  
SendPort(port,\*G);  
GM=ReadPort(port);  
if(GM/16==0)  
return GM;  
else if(GM/16==1){  
do{  
\*G=GM;  
SendPort(port,GM);  
GM=ReadPort(port);  
}while(GM/16==1);  
}  
return GM;  
}  
void GetFileName(char \*f)  
{  
int G1,G2,G3;  
char ch;  
G1=ReadPort(PORT);  
G2=ReadPort(PORT);  
do{  
G3=Receive(PORT,&G3);  
ch=(char)(G1%16+G2/16);  
\*f=ch;  
\*f++;  
G2=Receive(PORT,&G3);  
G1=G3;  
}while(G1/16!=32);  
printf("File name transmit finished.\n");  
}  
void InitPort(port,para)  
int port;  
unsigned char para;  
{  
union REGS reg;  
reg.x.dx=port;  
reg.h.ah=0;  
reg.h.al=para;  
int86(0x14,®,®);  
}  
void SendPort(port,c)  
int port;  
char c;  
{  
union REGS reg;  
reg.x.dx=port;  
reg.h.al=c;  
reg.h.ah=1;  
int86(0x14,®,®);  
if(reg.h.ah&128){  
printf("\nSend mistakes!");  
exit(1);  
}  
}  
int ReadPort(port)  
int port;  
{  
union REGS reg;  
while(!(CheckState(port)&256)){  
if(kbhit()){/\*如端口长期无数据可人为终止等待\*/  
printf("Press any key to exit.");  
getch();  
exit(1);  
}  
}  
reg.x.dx=port;  
reg.h.ah=2;  
int86(0x14,®,®);  
if(reg.h.ah&128){  
printf("\nRead mistake!");  
exit(1);  
}  
return reg.h.al;  
}  
int CheckState(port)  
int port;  
{  
union REGS reg;  
reg.x.dx=port;  
reg.h.ah=3;  
int86(0x14,®,®);  
return reg.x.ax;  
}

以上程序可传送各种格式的文件，也有一定的自动纠错能力，但对于异常情况的  
处理能力比较弱，读者可以自己改进。由于篇幅限制，对于中断14H的功能、入  
口参数及返回参数的意义请读者自己查有关资料。  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
附录：   
现在大多数串行口都遵循RS-232标准，以下是最常用的RS-232信号：   
名称 针号 含义  
RTS　4　　Request to send（请求发送）  
CTS　5　　Clear to send（清除发送）  
DSR　6　　Data set ready（数据设备预备好）  
DTR　20 　Data terminal ready（数据终端预备好）  
TXD　2　　Transmit data（发送数据）  
RXD　3　　Receive data（接收数据）  
GRD　7　　Ground（接地

ioctl是[设备驱动程序](http://baike.baidu.com/view/15565.htm)中对设备的[I/O通道](http://baike.baidu.com/view/1431426.htm)进行管理的函数。所谓对I/O通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等等。它的参数个数如下：int ioctl(int fd, int cmd, …)；其中fd就是用户程序打开设备时使用open函数返回的文件标示符，cmd就是用户程序对设备的控制命令，至于后面的省略号，那是一些补充参数，一般最多一个，有或没有是和cmd的意义相关的。ioctl函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对ioctl的支持，用户就能在用户程序中使用ioctl函数控制设备的I/O通道。

目录

1[功 能](http://baike.baidu.com/view/1081282.htm?fr=aladdin" \l "1)

2[必要性](http://baike.baidu.com/view/1081282.htm?fr=aladdin#2)

3[实现操作](http://baike.baidu.com/view/1081282.htm?fr=aladdin#3)

4[其他信息](http://baike.baidu.com/view/1081282.htm?fr=aladdin#4)

5[总结](http://baike.baidu.com/view/1081282.htm?fr=aladdin#5)

1功 能[编辑](http://baike.baidu.com/view/1081282.htm?fr=aladdin)

控制I/O设备 ，提供了一种获得设备信息和向设备发送控制参数的手段。用于向设备发控制和配置命令 ，有些命令需要控制参数，这些数据是不能用read / write 读写的,称为Out-of-band数据。也就是说,read / write 读写的数据是in-band数据,是I/O操作的主体，而ioctl 命令传送的是控制信息,其中的数据是辅助的数据。

用 法: int ioctl(int handle, int cmd,[int \*argdx, int argcx]);

返回值：成功为0，出错为-1

usr/include/asm-generic/ioctl.h中定义的宏的注释：

#define \_IOC\_NRBITS 8 //序数（number）字段的[字位](http://baike.baidu.com/view/2307934.htm)宽度，8bits

#define \_IOC\_TYPEBITS 8 //幻数（type）字段的[字位](http://baike.baidu.com/view/2307934.htm)宽度，8bits

#define \_IOC\_SIZEBITS 14 //大小（size）字段的[字位](http://baike.baidu.com/view/2307934.htm)宽度，14bits

#define \_IOC\_DIRBITS 2 //方向（direction）字段的[字位](http://baike.baidu.com/view/2307934.htm)宽度，2bits

#define \_IOC\_NRMASK ((1 << \_IOC\_NRBITS)-1) //序数字段的掩码，0x000000FF

#define \_IOC\_TYPEMASK ((1 << \_IOC\_TYPEBITS)-1) //幻数字段的掩码，0x000000FF

#define \_IOC\_SIZEMASK ((1 << \_IOC\_SIZEBITS)-1) //大小字段的掩码，0x00003FFF

#define \_IOC\_DIRMASK ((1 << \_IOC\_DIRBITS)-1) //方向字段的掩码，0x00000003

#define \_IOC\_NRSHIFT 0 //序数字段在整个字段中的位移，0

#define \_IOC\_TYPESHIFT (\_IOC\_NRSHIFT+\_IOC\_NRBITS) //幻数字段的位移，8

#define \_IOC\_SIZESHIFT (\_IOC\_TYPESHIFT+\_IOC\_TYPEBITS) //大小字段的位移，16

#define \_IOC\_DIRSHIFT (\_IOC\_SIZESHIFT+\_IOC\_SIZEBITS) //方向字段的位移，30

/\*

\* Direction bits.

\*/

#define \_IOC\_NONE 0U //没有数据传输

#define \_IOC\_WRITE 1U //向设备写入数据，驱动程序必须从[用户空间](http://baike.baidu.com/view/4274331.htm)读入数据

#define \_IOC\_READ 2U //从设备中读取数据，[驱动程序](http://baike.baidu.com/view/1048.htm)必须向[用户空间](http://baike.baidu.com/view/4274331.htm)写入数据

#define \_IOC(dir,type,nr,size) \

(((dir) << \_IOC\_DIRSHIFT) | \

((type) << \_IOC\_TYPESHIFT) | \

((nr) << \_IOC\_NRSHIFT) | \

((size) << \_IOC\_SIZESHIFT))

/\*

\* used to create numbers

\*/

//构造无参数的命令编号

#define \_IO(type,nr) \_IOC(\_IOC\_NONE,(type),(nr),0)

//构造从[驱动程序](http://baike.baidu.com/view/1048.htm)中读取数据的命令编号

#define \_IOR(type,nr,size) \_IOC(\_IOC\_READ,(type),(nr),sizeof(size))

//用于向[驱动程序](http://baike.baidu.com/view/1048.htm)写入数据命令

#define \_IOW(type,nr,size) \_IOC(\_IOC\_WRITE,(type),(nr),sizeof(size))

//用于双向传输

#define \_IOWR(type,nr,size) \_IOC(\_IOC\_READ|\_IOC\_WRITE,(type),(nr),sizeof(size))

/\*

\*used to decode ioctl numbers..

\*/

//从命令参数中解析出数据方向，即写进还是读出

#define \_IOC\_DIR(nr) (((nr) >> \_IOC\_DIRSHIFT) & \_IOC\_DIRMASK)

//从命令参数中解析出幻数type

#define \_IOC\_TYPE(nr) (((nr) >> \_IOC\_TYPESHIFT) & \_IOC\_TYPEMASK)

//从命令参数中解析出序数number

#define \_IOC\_NR(nr) (((nr) >> \_IOC\_NRSHIFT) & \_IOC\_NRMASK)

//从命令参数中解析出用户数据大小

#define \_IOC\_SIZE(nr) (((nr) >> \_IOC\_SIZESHIFT) & \_IOC\_SIZEMASK)

/\* ...and for the drivers/sound files... \*/

#define IOC\_IN (\_IOC\_WRITE << \_IOC\_DIRSHIFT)

#define IOC\_OUT (\_IOC\_READ << \_IOC\_DIRSHIFT)

#define IOC\_INOUT ((\_IOC\_WRITE|\_IOC\_READ) << \_IOC\_DIRSHIFT)

#define IOCSIZE\_MASK (\_IOC\_SIZEMASK << \_IOC\_SIZESHIFT)

#define IOCSIZE\_SHIFT (\_IOC\_SIZESHIFT)

程序例:

#include <stdlib.h>

#include <stdio.h>

#include <sys/ioctl.h>

int main(void) {

..int stat;

/\* use func 8 to determine if the default drive is removable \*/

..stat = ioctl(0, 8, 0, 0);

..if (!stat)

....printf("Drive %c is removable.\n", getdisk() + 'A');

..else

....printf("Drive %c is not removable.\n", getdisk() + 'A');

..return 0;

}

int ioctl( int fd, int request, .../\* void \*arg \*/ ) 详解

第三个参数总是一个[指针](http://baike.baidu.com/view/159417.htm)，但指针的类型依赖于request 参数。我们可以把和网络相关的请求划分为6 类：

[套接口](http://baike.baidu.com/view/567586.htm)操作

文件操作

接口操作

ARP 高速缓存操作

路由表操作

流系统

下表列出了网络相关ioctl请求的request 参数以及arg 地址必须指向的[数据类型](http://baike.baidu.com/view/675645.htm)：

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | Request | 说明 | 数据类型 |
| **套**  **接**  **口** | SIOCATMARK  SIOCSPGRP  SIOCGPGRP | 是否位于带外标记  设置套接口的进程ID 或进程组ID  获取套接口的进程ID 或进程组ID | int  int  int |
| **文**  **件** | FIONBIO  FIOASYNC  FIONREAD  FIOSETOWN  FIOGETOWN | 设置/ 清除非阻塞I/O 标志  设置/ 清除信号驱动异步I/O 标志  获取接收缓存区中的字节数  设置文件的进程ID 或进程组ID  获取文件的进程ID 或进程组ID | int  int  int  int  int |
| **接**  **口** | SIOCGIFCONF  SIOCSIFADDR  SIOCGIFADDR  SIOCSIFFLAGS  SIOCGIFFLAGS  SIOCSIFDSTADDR  SIOCGIFDSTADDR  SIOCGIFBRDADDR  SIOCSIFBRDADDR  SIOCGIFNETMASK  SIOCSIFNETMASK  SIOCGIFMETRIC  SIOCSIFMETRIC  SIOCGIFMTU  SIOCxxx | 获取所有接口的清单  设置接口地址  获取接口地址  设置接口标志  获取接口标志  设置点到点地址  获取点到点地址  获取广播地址  设置广播地址  获取子网掩码  设置子网掩码  获取接口的测度  设置接口的测度  获取接口MTU  （还有很多取决于系统的实现） | struct ifconf  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq  struct ifreq |
| **ARP** | SIOCSARP  SIOCGARP  SIOCDARP | 创建/ 修改ARP 表项  获取ARP 表项  删除ARP 表项 | struct arpreq  struct arpreq  struct arpreq |
| **路**  **由** | SIOCADDRT  SIOCDELRT | 增加路径  删除路径 | struct rtentry  struct rtentry |
| **流** | I\_xxx |  |  |

**套接口操作：**

明确用于套接口操作的ioctl请求有三个, 它们都要求ioctl的第三个参数是指向某个整数的一个指针。

SIOCATMARK: 如果本套接口的的度指针当前位于带外标记，那就通过由第三个参数指向的整数返回一个非0 值；否则返回一个0 值。POSIX 以函数sockatmark 替换本请求。

SIOCGPGRP ： 通过第三个参数指向的整数返回本套接口的进程ID 或进程组ID ，该ID 指定针对本套接口的SIGIO 或SIGURG 信号的接收进程。本请求和fcntl 的F\_GETOWN 命令等效，POSIX 标准化的是fcntl 函数。

SIOCSPGRP ： 把本套接口的进程ID 或者进程组ID 设置成第三个参数指向的整数，该ID 指定针对本套接口的SIGIO 或SIGURG 信号的接收进程，本请求和fcntl 的F\_SETOWN 命令等效，POSIX 标准化的是fcntl 操作。

**文件操作：**

以下5 个请求都要求ioctl的第三个参数指向一个整数。

FIONBIO ： 根据ioctl的第三个参数指向一个0 或非0 值分别清除或设置本套接口的非阻塞标志。本请求和O\_NONBLOCK 文件状态标志等效，而该标志通过fcntl 的F\_SETFL 命令清除或设置。

FIOASYNC ： 根据ioctl 的第三个参数指向一个0 值或非0 值分别清除或设置针对本套接口的信号驱动异步I/O 标志，它决定是否收取针对本套接口的异步I/O 信号（SIGIO ）。本请求和O\_ASYNC 文件状态标志等效，而该标志可以通过fcntl 的F\_SETFL 命令清除或设置。

FIONREAD ： 通过由ioctl的第三个参数指向的整数返回当前在本套接口接收缓冲区中的字节数。本特性同样适用于文件，管道和终端。

FIOSETOWN ： 对于套接口和SIOCSPGRP 等效。

FIOGETOWN ： 对于套接口和SIOCGPGRP 等效。

2必要性[编辑](http://baike.baidu.com/view/1081282.htm?fr=aladdin)

如果不用IOCTL的话，也能实现对设备I/O通道的控制，但那就是蛮拧了。例如，我们可以在驱动程式中实现WRITE的时候检查一下是否有特别约定的数据流通过，如果有的话，那么后面就跟着控制命令（一般在SOCKET编程中常常这样做）。不过如果这样做的话，会导致代码分工不明，程式结构混乱，程式员自己也会头昏眼花的。所以，我们就使用IOCTL来实现控制的功能。要记住，用户程式所作的只是通过命令码告诉驱动程式他想做什么，至于怎么解释这些命令和怎么实现这些命令，这都是驱动程式要做的事情。

3实现操作[编辑](http://baike.baidu.com/view/1081282.htm?fr=aladdin)

读者只要把write换成ioctl，就知道用户程式的ioctl是怎么和驱动程式中的ioctl实现联系在一起的了。我这里说一个大概思路，因为我觉得《Linux设备驱动程式》这本书已说的非常清晰了，不过得花一些时间来看。在驱动程式中实现的ioctl函数体内，实际上是有一个switch{case}结构，每一个case对应一个命令码，做出一些相应的操作。怎么实现这些操作，这是每一个程式员自己的事情，因为设备都是特定的，这里也没法说。关键在于怎么样组织命令码，因为在ioctl中命令码是唯一联系用户程式命令和驱动程式支持的途径。命令码的组织是有一些讲究的，因为我们一定要做到命令和设备是一一对应的，这样才不会将正确的命令发给错误的设备，或是把错误的命令发给正确的设备，或是把错误的命令发给错误的设备。这些错误都会导致不可预料的事情发生，而当程式员发现了这些奇怪的事情的时候，再来调试程式查找错误，那将是非常困难的事情。所以在Linux核心中是这样定义一个命令码的：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| 设备类型 | 序列号 | 方向 |数据尺寸|

|----------|--------|------|--------|

| 8 bit | 8 bit |2 bit |8~14 bit|

|----------|--------|------|-------|

这样一来，一个命令就变成了一个整数形式的命令码。不过命令码非常的不直观，所以Linux Kernel中提供了一些宏，这些宏可根据便于理解的字符串生成命令码，或是从命令码得到一些用户能理解的字符串以标明这个命令对应的设备类型、设备序列号、数据传送方向和数据传输尺寸。这些宏我就不在这里解释了，具体的形式请读者察看Linux核心原始码中的和，文件里给除了这些宏完整的定义。这里我只多说一个地方，那就是"幻数"。幻数是个字母，数据长度也是8，所以就用一个特定的字母来标明设备类型，这和用一个数字是相同的，只是更加利于记忆和理解。就是这样，再没有更复杂的了。更多的说了也没有，读者还是看一看原始码吧，推荐各位阅读《Linux 设备驱动程式》所带原始码中的short一例，因为他比较短小，功能比较简单，能看明白ioctl的功能和细节。

4其他信息[编辑](http://baike.baidu.com/view/1081282.htm?fr=aladdin)

cmd参数怎么得出？

这里确实要说一说，cmd参数在用户程式端由一些宏根据设备类型、序列号、传送方向、数据尺寸等生成，这个整数通过[系统调用](http://baike.baidu.com/view/47173.htm)传递到内核中的驱动程式，再由驱动程式使用解码宏从这个整数中得到设备的类型、序列号、传送方向、数据尺寸等信息，然后通过switch{case}结构进行相应的操作。要透彻理解，只能是通过阅读原始码，我这篇文章实际上只是个引子。Cmd参数的组织还是比较复杂的，我认为要搞熟他还是得花不少时间的，不过这是值得的，驱动程式中最难的是对中断的理解。