ioctl

ioctl是设备驱动程序中对设备的I/O通道进 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。

中文名

ioctl

属　性

控制I/O设备

功　能

进行管理的函数

特　点

特性进行控制

目录

功 能

编辑

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）字段的字位宽度，8bits

#define _IOC_TYPEBITS 8 //幻数（type）字段的字位宽度，8bits

#define _IOC_SIZEBITS 14 //大小（size）字段的字位宽度，14bits

#define _IOC_DIRBITS 2 //方向（direction）字段的字位宽度，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读入数据

#define _IOC_READ 2U //从设备中读取数据，驱动程序必须向用户空间写入数据

```
#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，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：

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

必要性

编辑

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。

实现操作

编辑

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设备驱动程序》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 设备驱动程序》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。

其他信息

编辑

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。