第5章 字符设备驱动

现在，你已经准备就绪了，可以尝试去写一个简单、但实用的设备驱动了。在这一章，我们将深入字符设备驱动的内幕：顺序存取设备数据的内核代码。字符设备驱动能从如下几类设备获取原始的数据：如打印机、鼠标、看门狗、键盘、内存、实时时钟等，但它不适合用于以块方式存储的、随机访问的设备，如硬盘、软盘和光盘。

字符设备驱动基础

让我们以自顶向下的方式开始字符设备驱动学习之旅。为了访问一个字符设备，系统用户需要调用相应的应用程序。此应用程序负责和设备交互，为了实现此目的，需要得到相应驱动的标识符。驱动通过/dev目录给用户提供接口：

bash> ls -l /dev

total 0

crw------- 1 root root 5, 1 Jul 16 10:02 console

...

lrwxrwxrwx 1 root root 3 Oct 6 10:02 cdrom -> hdc

...

brw-rw---- 1 root disk 3, 0 Oct 6 2007 hda

brw-rw---- 1 root disk 3, 1 Oct 6 2007 hda1

...

crw------- 1 root tty 4, 1 Oct 6 10:20 tty1

crw------- 1 root tty 4, 2 Oct 6 10:02 tty2

ls命令输出结果的每一行的第一个字符表示驱动的类型：c表示字符设备驱动，b代表块设备驱动，l表示符号链接。第五列的数字是主设备号，第六列是次设备号。主设备号通常标识设备对应的驱动程序，次设备号用于确定驱动程序所服务的设备。例如，IDE块存储设备驱动/dev/had的主设备号为3，负责处理系统的硬盘；当进一步指明其次设备号为1时（/dev/hda1），它指向第一个硬盘分区。字符设备驱动与块设备驱动占用不同空间，因此可以将同一个主设备号分配给字符设备和块设备驱动。

让我们进一步深入字符设备驱动。从程序结构的角度看，字符设备驱动包括如下内容：

· 初始化例程init()，负责初始化设备并且将驱动和内核的其它部分通过注册函数实现无缝连接。

· 入口函数（或方法）集，如open()，read()，ioctl()，llseek()，和write()，这些函数直接对应相应的的I/O系统调用，由用户应用程序通过对应的/dev节点调用。

· 中断例程，底半部例程，定时器处理例程，内核辅助线程，以及其他的组成部分。它们大部分对用户应用程序是透明的。

从数据流的角度看，字符设备驱动包括如下关键的数据结构：

1. 特定设备相关(per-device)的数据结构。此结构保存着驱动频繁使用的信息。

2. cdev结构，针对字符设备驱动的内核抽象。这个结构通常作为前面讨论过的特定设备相关(per-device)结构的成员。

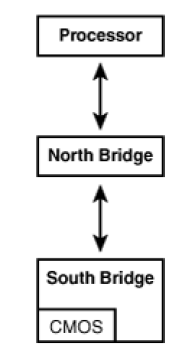
3. file\_operations结构，包括所有设备驱动入口函数的地址。

4. file结构,包括关联的/dev节点的信息。

设备例子：系统CMOS

让我们实现一个字符设备驱动以访问系统CMOS。在PC兼容的硬件上（见图5.1）BIOS使用CMOS存储系统信息，如启动选项，引导顺序，系统数据等，我们可以通过BIOS设置菜单对其进行配置。我们的CMOS设备驱动使你像访问普通文件一样访问两个PC CMOS块bank。应用程序可以在/dev/cmos/0和/dev/cmos/1上使用I/O系统调用从两个块bank存取数据。因为BIOS分配给CMOS域的存取粒度是比特级的，所以驱动程序能够进行比特级的访问。因此，read()可以获取指定数目的比特，并根据读取的比特数移动内部文件指针。

图 5.1. PC兼容系统的CMOS



通过两个I/O地址访问CMOS，一个索引寄存器和一个数据寄存器，如表5.1所示。你必须在索引寄存器中指定准备访问的CMOS存储器的偏移，然后通过数据寄存器来交换数据。

表 5.1. CMOS中寄存器分布

寄存器名称

描述

CMOS\_BANK0\_INDEX\_PORT

在此寄存器中指定了CMOS块bank 0的偏移。

Specify the desired CMOS bank 0 offset in this register.

CMOS\_BANK0\_DATA\_PORT

对CMOS\_BANK0\_INDEX\_PORT指定的地址读取/写入数据。

CMOS\_BANK1\_INDEX\_PORT

在此寄存器中指定了CMOS块bank 1的偏移。

CMOS\_BANK1\_DATA\_PORT

对CMOS\_BANK0\_INDEX\_PORT指定的地址读取/写入数据。

因为每个驱动方法都有一个对应的由应用程序使用的系统调用，我们将看看系统调用和相应的驱动方法。

驱动初始化

init()函数是注册机制的基础。它负责完成如下工作：

申请分配主设备号。

为特定设备相关(per-device)的数据结构分配内存。

将入口函数（open()，read()等）和字符驱动的cdev抽象相关联。

将主设备号和驱动的cdev相关联。

在/dev 和 /sys下创建节点。如在第4章“打下基础”中所讨论的，/dev的管理经历了2.2版本内核中的静态设备节点，到2.4中的动态指定name， 再到2.6中的用户空间的守护进程（udevd）的历程。

初始化硬件。在本例的简单CMOS驱动中不涉及此部分。

清单5.1实现了CMOS驱动的init()函数。

清单5.1. CMOS 驱动初始化

Code View:

#include <linux/fs.h>

/\* Per-device (per-bank) structure \*/

struct cmos\_dev {

unsigned short current\_pointer; /\* Current pointer within the

bank \*/

unsigned int size; /\* Size of the bank \*/

int bank\_number; /\* CMOS bank number \*/

struct cdev cdev; /\* The cdev structure \*/

char name[10]; /\* Name of I/O region \*/

/\* ... \*/ /\* Mutexes, spinlocks, wait

queues, .. \*/

} \*cmos\_devp;

/\* File operations structure. Defined in linux/fs.h \*/

static struct file\_operations cmos\_fops = {

.owner = THIS\_MODULE, /\* Owner \*/

.open = cmos\_open, /\* Open method \*/

.release = cmos\_release, /\* Release method \*/

.read = cmos\_read, /\* Read method \*/

.write = cmos\_write, /\* Write method \*/

.llseek = cmos\_llseek, /\* Seek method \*/

.ioctl = cmos\_ioctl, /\* Ioctl method \*/

};

static dev\_t cmos\_dev\_number; /\* Allotted device number \*/

struct class \*cmos\_class; /\* Tie with the device model \*/

#define NUM\_CMOS\_BANKS 2

#define CMOS\_BANK\_SIZE (0xFF\*8)

#define DEVICE\_NAME "cmos"

#define CMOS\_BANK0\_INDEX\_PORT 0x70

#define CMOS\_BANK0\_DATA\_PORT 0x71

#define CMOS\_BANK1\_INDEX\_PORT 0x72

#define CMOS\_BANK1\_DATA\_PORT 0x73

unsigned char addrports[NUM\_CMOS\_BANKS] = {CMOS\_BANK0\_INDEX\_PORT,

CMOS\_BANK1\_INDEX\_PORT,};

unsigned char dataports[NUM\_CMOS\_BANKS] = {CMOS\_BANK0\_DATA\_PORT,

CMOS\_BANK1\_DATA\_PORT,};

/\*

\* Driver Initialization

\*/

int \_\_init

cmos\_init(void)

{

int i;

/\* Request dynamic allocation of a device major number \*/

if (alloc\_chrdev\_region(&cmos\_dev\_number, 0,

NUM\_CMOS\_BANKS, DEVICE\_NAME) < 0) {

printk(KERN\_DEBUG "Can't register device\n"); return -1;

}

/\* Populate sysfs entries \*/

cmos\_class = class\_create(THIS\_MODULE, DEVICE\_NAME);

for (i=0; i<NUM\_CMOS\_BANKS; i++) {

/\* Allocate memory for the per-device structure \*/

cmos\_devp = kmalloc(sizeof(struct cmos\_dev), GFP\_KERNEL);

if (!cmos\_devp) {

printk("Bad Kmalloc\n"); return 1;

}

/\* Request I/O region \*/

sprintf(cmos\_devp->name, "cmos%d", i);

if (!(request\_region(addrports[i], 2, cmos\_devp->name)) {

printk("cmos: I/O port 0x%x is not free.\n", addrports[i]);

return –EIO;

}

/\* Fill in the bank number to correlate this device

with the corresponding CMOS bank \*/

cmos\_devp->bank\_number = i;

/\* Connect the file operations with the cdev \*/

cdev\_init(&cmos\_devp->cdev, &cmos\_fops);

cmos\_devp->cdev.owner = THIS\_MODULE;

/\* Connect the major/minor number to the cdev \*/

if (cdev\_add(&cmos\_devp->cdev, (dev\_number + i), 1)) {

printk("Bad cdev\n");

return 1;

}

/\* Send uevents to udev, so it'll create /dev nodes \*/

class\_device\_create(cmos\_class, NULL, (dev\_number + i),

NULL, "cmos%d", i);

}

printk("CMOS Driver Initialized.\n");

return 0;

}

/\* Driver Exit \*/

void \_\_exit

cmos\_cleanup(void)

{

int i;

/\* Remove the cdev \*/

cdev\_del(&cmos\_devp->cdev);

/\* Release the major number \*/

unregister\_chrdev\_region(MAJOR(dev\_number), NUM\_CMOS\_BANKS);

/\* Release I/O region \*/

for (i=0; i<NUM\_CMOS\_BANKS; i++) {

class\_device\_destroy(cmos\_class, MKDEV(MAJOR(dev\_number), i));

release\_region(addrports[i], 2);

}

/\* Destroy cmos\_class \*/

class\_destroy(cmos\_class);

return();

}

module\_init(cmos\_init);

module\_exit(cmos\_cleanup);

cmos\_init()完成的大部分工作是通用的，因此如果替换掉和CMOS数据结构相关的部分，你可以使用清单5.1作为模板开发其他的字符设备驱动。

首先，cmos\_init()调用alloc\_chrdev\_region()，动态申请一个未用的主设备号。如果调用成功，dev\_number包含系统分配的主设备号。alloc\_chrdev\_region()的第二个参数是请求的起始次设备号，第三个参数是支持的次设备号数目。最后一个参数是和CMOS关联的设备名称，它将出现在/proc/devices中：

bash> cat /proc/devices | grep cmos

253 cmos

253是动态分配给CMOS设备的主设备号。在2.6内核以前，不支持动态设备节点分配，因此字符驱动使用register\_chrdev()调用去静态申请指定的主设备号。

继续分析代码之前，让我们先快速浏览一下清单5.1中所使用的数据结构。cmos\_dev就是和特定设备相关（per-device）的数据结构。cmos\_fops是file\_operations类型的数据结构，它包含驱动入口地址。cmos\_fops还包括一个owner域，其值被置为THIS\_MODULE，也就是正讨论的驱动模块的地址。在处理设备的打开和释放时，知晓结构拥有者的身份可以让内核帮助驱动减轻一些事务管理的负担，如跟踪使用计数。

正如你所看到的，内核用cdev结构来表示字符设备。字符驱动通常将cdev结构包含于字符设备特定的数据结构per-device中。在我们的例子中，cdev定义于cmos\_dev中。cmos\_init()为本驱动程序服务的每一个设备（在本例子中是CMOS bank）分配相关数据结构的内存：包括设备特定（pei-device）的数据结构、及其cdev结构。cev\_init()将文件操作（cmos\_fops）和cdev关联，cdev\_add()将通过alloc\_chrdev\_region()分配的主/次设备号和cdev连接在一起。

class\_create() 为此设备构建了sysfs入口点，class\_device\_create()导致了两个uevent的产生：cmos0和cmos1。正如你在第四章中所学习的，udevd监听uevent，在查阅规则数据库后创建设备节点。为了在接收到相应的uevent（cmos0和cmos1）后，创建和两个CMOS 块bank（/dev/cmos/0和/dev/cmos/1）对应的设备节点，需要将如下内容添加进udev规则目录中（/etc/udev/rules.d/）：

KERNEL="cmos[0-1]\*", NAME="cmos/%n"

通过调用request\_region()申请I/O地址后，设备驱动在申请的I/O地址中操作运行。此机制确保其他程序不能申请此区域，直至通过调用release\_region()释放占用的区域。request\_region()通常被I/O总线驱动程序调用，例如PCI和ISA，以指明对于处理器地址空间中on-card内存的拥有权（更多细节请见第十章，“PCI总线”）。cmos\_init()调用request\_region()去申请每个CMOS bank的I/O。request\_region()的最后一个参数是一个被/proc/ioports使用的标识符，如果你查看文件/proc/ioports将会看到：

bash> grep cmos /proc/ioports

0070-0071 : cmos0

0072-0073 : cmos1

完成注册过程后，cmos\_init()打印一条信息，宣告其一切顺利！

Open 与 Release

当应用程序打开设备节点时，内核调用相应驱动的open()函数。你可以在shell中执行：

bash> cat /dev/cmos/0

来触发cmos\_open()的执行。

当应用程序关闭一个已经打开的设备时，内核调用release()函数。因此当读取CMOS bank 0的内容后，cat关闭和/dev/cmos/0关联的文件描述符，内核调用cmos\_release()。

清单5.2显示了cmos\_open()和cmos\_release()的具体实现。让我们近距离接触cmos\_open()。有两点值得我们关注。首先是cmos\_dev的获取。cmos\_open()的输入参数inode包含cdev结构在初始化过程中分配的地址。正如清单5.1中所显示的，cdev定义于cmos\_dev中。为了获取容器结构coms\_dev的地址，cmos\_open()使用了内核的辅助函数container\_of()。

cmos\_open()中另外值得关注的是第二个输入参数、file结构中private\_data数据域的使用。你可以使用此数据域（file->private\_data）作为占位符，方便的关联到其它驱动方法。

CMOS驱动程序中使用此数据域来存储cmos\_dev的地址。阅读cmos\_release()（以及函数的其他部分）可以发现private\_data被如何用来直接获取属于相应CMOS块bank的cmos\_dev结构体的处理程序。

清单5.2. Open与Release

Code View:

/\*

\* Open CMOS bank

\*/

int

cmos\_open(struct inode \*inode, struct file \*file)

{

struct cmos\_dev \*cmos\_devp;

/\* Get the per-device structure that contains this cdev \*/

cmos\_devp = container\_of(inode->i\_cdev, struct cmos\_dev, cdev);

/\* Easy access to cmos\_devp from rest of the entry points \*/

file->private\_data = cmos\_devp;

/\* Initialize some fields \*/

cmos\_devp->size = CMOS\_BANK\_SIZE;

cmos\_devp->current\_pointer = 0;

return 0;

}

/\*

\* Release CMOS bank

\*/

int

cmos\_release(struct inode \*inode, struct file \*file)

{

struct cmos\_dev \*cmos\_devp = file->private\_data;

/\* Reset file pointer \*/

cmos\_devp->current\_pointer = 0;

return 0;

}

数据交换

read()和write()是负责在用户空间和设备之间交换数据的主要的字符驱动函数。扩展的read()/write()函数族包括几个其他的函数：fsync()，aio\_read()，aio\_write()，和mmap()。

CMOS驱动在一个简单的存储设备上操作，而不必处理其他一般的字符驱动必须面对的复杂问题。

· CMOS数据访问例程不必“休眠——等待”设备I/O完成，其他字符驱动的read()和write()则必须支持阻塞和非阻塞操作模式。除非设备文件在非阻塞方式（O\_NONBLOCK）下打开，否则read()和write()将会使调用进程休眠至I/O操作完成。

· CMOS驱动操作基于同步方式，不必依赖中断。然而，很多驱动的数据访问函数依靠中断来获取数据，并且需要通过等待队列等数据结构和中断上下文代码通信。

清单5.3包括CMOS驱动的read()和write()函数。你不能从内核中直接访问用户空间的缓冲区，反之亦然。因此为了拷贝CMOS存储器数据至用户空间，cmos\_read()需要调用copy\_to\_user()。cmos\_write()通过调用copy\_from\_user()完成相反的工作。因为copy\_to\_user()和copy\_from\_user()可能会睡眠，所以在调用这两个函数的时候不能持有自选锁。

如同你在前面所看到的，通过操作一对I/O地址，完成CMOS内存访问。为了从I/O地址中读取不同大小的数据，内核提供了一组和结构无关的函数：in[b|w|l|sb|sl]()。类似地，可以通过out[b|w|l|sb|sl]()去写I/O区域。清单5.3中port\_data\_in()和port\_data\_out()使用inb()和outb()进行数据传输。

清单 5.3. Read and Write

Code View:

/\*

\* Read from a CMOS Bank at bit-level granularity

\*/

ssize\_t

cmos\_read(struct file \*file, char \*buf,

size\_t count, loff\_t \*ppos)

{

struct cmos\_dev \*cmos\_devp = file->private\_data;

char data[CMOS\_BANK\_SIZE];

unsigned char mask;

int xferred = 0, i = 0, l, zero\_out;

int start\_byte = cmos\_devp->current\_pointer/8;

int start\_bit = cmos\_devp->current\_pointer%8;

if (cmos\_devp->current\_pointer >= cmos\_devp->size) {

return 0; /\*EOF\*/

}

/\* Adjust count if it edges past the end of the CMOS bank \*/

if (cmos\_devp->current\_pointer + count > cmos\_devp->size) {

count = cmos\_devp->size - cmos\_devp->current\_pointer;

}

/\* Get the specified number of bits from the CMOS \*/

while (xferred < count) {

data[i] = port\_data\_in(start\_byte, cmos\_devp->bank\_number)

>> start\_bit;

xferred += (8 - start\_bit);

if ((start\_bit) && (count + start\_bit > 8)) {

data[i] |= (port\_data\_in (start\_byte + 1,

cmos\_devp->bank\_number) << (8 - start\_bit));

xferred += start\_bit;

}

start\_byte++;

i++;

}

if (xferred > count) {

/\* Zero out (xferred-count) bits from the MSB

of the last data byte \*/

zero\_out = xferred - count;

mask = 1 << (8 - zero\_out);

for (l=0; l < zero\_out; l++) {

data[i-1] &= ~mask; mask <<= 1;

}

xferred = count;

}

if (!xferred) return -EIO;

/\* Copy the read bits to the user buffer \*/

if (copy\_to\_user(buf, (void \*)data, ((xferred/8)+1)) != 0) {

return -EIO;

}

/\* Increment the file pointer by the number of xferred bits \*/

cmos\_devp->current\_pointer += xferred;

return xferred; /\* Number of bits read \*/

}

/\*

\* Write to a CMOS bank at bit-level granularity. 'count' holds the

\* number of bits to be written.

\*/

ssize\_t

cmos\_write(struct file \*file, const char \*buf,

size\_t count, loff\_t \*ppos)

{

struct cmos\_dev \*cmos\_devp = file->private\_data;

int xferred = 0, i = 0, l, end\_l, start\_l;

char \*kbuf, tmp\_kbuf;

unsigned char tmp\_data = 0, mask;

int start\_byte = cmos\_devp->current\_pointer/8;

int start\_bit = cmos\_devp->current\_pointer%8;

if (cmos\_devp->current\_pointer >= cmos\_devp->size) {

return 0; /\* EOF \*/

}

/\* Adjust count if it edges past the end of the CMOS bank \*/

if (cmos\_devp->current\_pointer + count > cmos\_devp->size) {

count = cmos\_devp->size - cmos\_devp->current\_pointer;

}

kbuf = kmalloc((count/8)+1,GFP\_KERNEL);

if (kbuf==NULL)

return -ENOMEM;

/\* Get the bits from the user buffer \*/

if (copy\_from\_user(kbuf,buf,(count/8)+1)) {

kfree(kbuf);

return -EFAULT;

}

/\* Write the specified number of bits to the CMOS bank \*/

while (xferred < count) {

tmp\_data = port\_data\_in(start\_byte, cmos\_devp->bank\_number);

mask = 1 << start\_bit;

end\_l = 8;

if ((count-xferred) < (8 - start\_bit)) {

end\_l = (count - xferred) + start\_bit;

}

for (l = start\_bit; l < end\_l; l++) {

tmp\_data &= ~mask; mask <<= 1;

}

tmp\_kbuf = kbuf[i];

mask = 1 << end\_l;

for (l = end\_l; l < 8; l++) {

tmp\_kbuf &= ~mask;

mask <<= 1;

}

port\_data\_out(start\_byte,

tmp\_data |(tmp\_kbuf << start\_bit),

cmos\_devp->bank\_number);

xferred += (end\_l - start\_bit);

if ((xferred < count) && (start\_bit) &&

(count + start\_bit > 8)) {

tmp\_data = port\_data\_in(start\_byte+1,

cmos\_devp->bank\_number);

start\_l = ((start\_bit + count) % 8);

mask = 1 << start\_l;

for (l=0; l < start\_l; l++) {

mask >>= 1;

tmp\_data &= ~mask;

}

port\_data\_out((start\_byte+1),

tmp\_data |(kbuf[i] >> (8 - start\_bit)),

cmos\_devp->bank\_number);

xferred += start\_l;

}

start\_byte++;

i++;

}

if (!xferred) return -EIO;

/\* Push the offset pointer forward \*/

cmos\_devp->current\_pointer += xferred;

return xferred; /\* Return the number of written bits \*/

}

/\*

\* Read data from specified CMOS bank

\*/

unsigned char

port\_data\_in(unsigned char offset, int bank)

{

unsigned char data;

if (unlikely(bank >= NUM\_CMOS\_BANKS)) {

printk("Unknown CMOS Bank\n");

return 0;

} else {

outb(offset, addrports[bank]); /\* Read a byte \*/

data = inb(dataports[bank]);

}

return data;

}

/\*

\* Write data to specified CMOS bank

\*/

void

port\_data\_out(unsigned char offset, unsigned char data,

int bank)

{

if (unlikely(bank >= NUM\_CMOS\_BANKS)) {

printk("Unknown CMOS Bank\n");

return;

} else {

outb(offset, addrports[bank]); /\* Output a byte \*/

outb(data, dataports[bank]);

}

return;

}

如果一个字符驱动的write()成功返回，则表示驱动已经完成了将数据传送下去的任务。然而，这并不能保证数据已经被成功的写到设备了。如果某个应用程序需要确保成功，可以调用fsync()。相应的fsync()驱动函数确保数据从驱动缓冲区中排出，并且写到设备。CMOS驱动不需要fsync()，因为在CMOS驱动中，驱动写出等同于设备写出。

如果用户程序有数据存储在多个缓冲区中，需要发送至设备，可以通过多个驱动写操作来完成。但是由于以下原因，这种做法不可行：

1. 多个系统调用的开销和相应的上下文切换。

2. 驱动和设备紧密相连，它应该能够以更合理的方式完成以上工作：从不同的缓冲区收集数据，并将其分发至设备。

由于以上原因，Linux和其它UNIX版本支持read()和write()的向量版本。Linux字符设备驱动通常提供两个专门的函数以完成向量操作：readv()和writev()。从2.6.19内核发布版开始，这两个函数已经集成进Linux异步I/O（AIO）层。Linux AIO是一个广泛的话题，超出了本章的讨论范围，我们把精力集中在AIO提供的同步向量能力上。

向量驱动函数的原型如下：

ssize\_t aio\_read(struct kiocb \*iocb, const struct iovec \*vector,

unsigned long count, loff\_t offset);

ssize\_t aio\_write(struct kiocb \*iocb, const struct iovec \*vector,

unsigned long count, loff\_t offset);

aio\_read()/aio\_write()的第一个参数描述了AIO操作，第二个参数是一个iovec数组，这个参数是向量函数所使用的主要的数据结构，包含了数据缓冲区的地址和长度。实际上，此机制等同于第十章所讨论的分散/聚集式DMA在用户空间的实现。可以在include/linux/uio.h文件里看到iovec的定义，在drivers/net/tun.c[1]文件里有向量字符驱动函数的例子实现。

另一个数据访问函数是mmap()，它将设备内存和用户的虚拟内存关联在一起。应用程序可以调用相应的系统调用，也可以调用mmap()，直接在返回的内存区操作，以访问设备驻留的内存。很多驱动都没有实现mmap()，因此我们在这里将不会深入讨论。在drivers/char/mem.c里可以看到mmap()的一个实现示例。第19章“用户空间的驱动程序”的“访问内存区域”演示了如何使用mmap()。我们的CMOS驱动没有实现mmap()。

你可能已经注意到port\_data\_in() 和 port\_data\_out()通过调用宏unlikely()进行了块bank序号的健壮性检查。宏likely() 和 unlikely()通知GCC，条件比较成功的可能性。GCC使用这两个宏来预测分支。因为我们将bank完整性检查失败标记为“不可能”，GCC会将else{}中的代码插入代码流的当前顺序执行的位置。如果你使用的是likely() 而不是unlikely()，将会有相反的结果。

Seek

内核使用一个内部指针跟踪当前文件访问的位置。应用程序使用lseek()系统调用去申请内部文件指针的重定位。使用lseek()服务，你可以将文件指针重设至文件内的任意偏移位置。字符驱动相对应的lseek()是llseek()函数。cmos\_llseek()实现了CMOS驱动中的lseek（）函数。

像我们前面看到的，CMOS的内部文件指针按比特移动而不是按字节移动。如果从CMOS驱动中读1字节数据，文件指针移动8次，应用程序根据此规律做查找。根据CMOS 块bank的大小，cmos\_llseek()也实现了针对文件结束标志的相关操作。

为了理解llseek()的功能，让我们查看lseek()系统调用所支持的命令：

1. SEEK\_SET, 设置文件指针至指定的偏移。

2. SEEK\_CUR, 计算相对于当前位置的偏移。

3. SEEK\_END, 计算相对于文件结尾的偏移。此命令可巧妙地将文件指针移出文件结尾，却并不改变文件的大小。读取超出文件结尾的数据时，如果之前没有数据被显式写入，读到的数据将毫无意义。此技术经常被用于创建大文件。CMOS驱动中不支持SEEK\_END。

我们可以联系前面的定义，阅读清单5.4中的cmos\_llseek()的代码。

清单 5.4. Seek

Code View:

/\*

\* Seek to a bit offset within a CMOS bank

\*/

static loff\_t

cmos\_llseek(struct file \*file, loff\_t offset,

int orig)

{

struct cmos\_dev \*cmos\_devp = file->private\_data;

switch (orig) {

case 0: /\* SEEK\_SET \*/

if (offset >= cmos\_devp->size) {

return -EINVAL;

}

cmos\_devp->current\_pointer = offset; /\* Bit Offset \*/

break;

case 1: /\* SEEK\_CURR \*/

if ((cmos\_devp->current\_pointer + offset) >=

cmos\_devp->size) {

return -EINVAL;

}

cmos\_devp->current\_pointer = offset; /\* Bit Offset \*/

break;

case 2: /\* SEEK\_END - Not supported \*/

return -EINVAL;

default:

return -EINVAL;

}

return(cmos\_devp->current\_pointer);

}

Control

另一个常见的字符驱动函数被称作I/O控制（或者ioctl）。当应用程序需要请求设备某些特定的动作时，这个例程接收并实现应用程序的命令。由于CMOS内存被BIOS用来存储关键信息，例如启动设备顺序，它通常用循环冗余校验（Cyclic Redundancy Check）算法进行保护。为了检测数据是否被破坏，CMOS驱动支持两个ioctl命令：

1. 调整校验和，用于CMOS内容被修改后重新计算CRC。计算出的校验和被存储在CMOS bank 1预先指定的偏移上。

2. 验证校验和，用于检查CMOS内容是否完好。通过比较针对当前内容的计算出的CRC和以前存储的CRC来完成。

当应用程序需要进行校验和相关操作时，通过ioctl()系统调用发送命令给驱动程序。清单5.5 中cmos\_ioctl()函数是CMOS驱动ioctl函数的具体实现。adjust\_cmos\_crc(int bank, unsigned short seed)实现了标准的CRC算法，未在清单中列出。

清单 5.5. I/O Control

Code View:

#define CMOS\_ADJUST\_CHECKSUM 1

#define CMOS\_VERIFY\_CHECKSUM 2

#define CMOS\_BANK1\_CRC\_OFFSET 0x1E

/\*

\* Ioctls to adjust and verify CRC16s.

\*/

static int

cmos\_ioctl(struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg)

{

unsigned short crc = 0;

unsigned char buf;

switch (cmd) {

case CMOS\_ADJUST\_CHECKSUM:

/\* Calculate the CRC of bank0 using a seed of 0 \*/

crc = adjust\_cmos\_crc(0, 0);

/\* Seed bank1 with CRC of bank0 \*/

crc = adjust\_cmos\_crc(1, crc);

/\* Store calculated CRC \*/

port\_data\_out(CMOS\_BANK1\_CRC\_OFFSET,

(unsigned char)(crc & 0xFF), 1);

port\_data\_out((CMOS\_BANK1\_CRC\_OFFSET + 1),

(unsigned char) (crc >> 8), 1);

break;

case CMOS\_VERIFY\_CHECKSUM:

/\* Calculate the CRC of bank0 using a seed of 0 \*/

crc = adjust\_cmos\_crc(0, 0);

/\* Seed bank1 with CRC of bank0 \*/

crc = adjust\_cmos\_crc(1, crc);

/\* Compare the calculated CRC with the stored CRC \*/

buf = port\_data\_in(CMOS\_BANK1\_CRC\_OFFSET, 1);

if (buf != (unsigned char) (crc & 0xFF)) return -EINVAL;

buf = port\_data\_in((CMOS\_BANK1\_CRC\_OFFSET+1), 1);

if (buf != (unsigned char)(crc >> 8)) return -EINVAL;

break;

default:

return -EIO;

}

return 0;

}

检测数据可获得

很多用户应用程序功能复杂，open()/read()/write()/close()这些系统调用不能满足其需求。它们希望当设备上有数据到来时，或者驱动准备好接收新数据时，系统能够采用同步或异步的方式通知它们。在这一小节，我们将看到两个能够感知数据是否可获得的字符驱动方法：poll()和fasync()。前者是同步的，后者是异步的。因为这些机制属于相对高级的话题，在探究底层驱动如何实现之前，让我们首先理解如何应用这些特性。前面讨论的CMOS内存设备由于较为简单，所以不需要感知数据是否可用；因此让我们从流行的用户空间应用程序：X Window服务器，来开始一个新的应用情景。

轮询

考虑如下的从X Windows源代码树摘录的代码片断（从www.xfree86.org下载而来），看它如何处理鼠标事件：

xc/programs/Xserver/hw/xfree86/input/mouse/mouse.c:

case PROT\_THINKING: /\* ThinkingMouse \*/

/\* This mouse may send a PnP ID string, ignore it. \*/

usleep(200000); xf86FlushInput(pInfo->fd);

/\* Send the command to initialize the beast. \*/

for (s = "E5E5"; \*s; ++s) {

xf86WriteSerial(pInfo->fd, s, 1);

if ((xf86WaitForInput(pInfo->fd, 1000000) <= 0))

break;

xf86ReadSerial(pInfo->fd, &c, 1);

if (c != \*s) break;

}

break;

实际上，这段代码首先发送初始化命令给鼠标，然后进入轮询，直至检测到输入数据，随后从设备读取响应。如果你查看Xf86WaitForInput()函数的具体实现，将会看到对select()系统调用的调用：

Code View:

xc/programs/Xserver/hw/xfree86/os-support/shared/posix\_tty.c:

int

xf86WaitForInput(int fd, int timeout)

{

fd\_set readfds;

struct timeval to;

int r;

FD\_ZERO(&readfds);

if (fd >= 0) {

FD\_SET(fd, &readfds);

}

to.tv\_sec = timeout / 1000000;

to.tv\_usec = timeout % 1000000;

if (fd >= 0) {

SYSCALL (r = select(FD\_SETSIZE, &readfds, NULL, NULL, &to));

} else {

SYSCALL (r = select(FD\_SETSIZE, NULL, NULL, NULL, &to));

}

if (xf86Verbose >= 9)

ErrorF ("select returned %d\n", r);

return (r);

}

你可以提供多个文件描述符给select()，让它一直守望至相应的数据状态发生改变。你也可以设定一个超时值，在时间到达后不管数据是否可获得都返回。如果你设定的超时值为NULL，select()将永远阻塞等待而不会超时。详细文档可以参考select()的man或info提示页面。前面的X server轮询等待鼠标数据的程序片断中，对select()调用即设置了超时。

Linux支持另一个系统调用poll()，其功能和select类似。2.6内核支持一个新的非POSIX系统调用epoll()，它是poll()的超集。所有这些系统调用都依赖相同的底层字符驱动方法：poll()。

大多数I/O系统都是和POSIX兼容的，而不是Linux所特有的（毕竟像X Windows这些程序也运行于各种UNIX系统上，而不仅仅是Linux），但是驱动的具体实现是和操作系统相关的。在Linux上，poll()驱动方法是select()系统调用的支柱。在前面的X server情景中，鼠标驱动的poll()方法看起来像这样：

static DECLARE\_WAIT\_QUEUE\_HEAD(mouse\_wait); /\* Wait Queue \*/

static unsigned int

mouse\_poll(struct file \*file, poll\_table \*wait)

{

poll\_wait(file, &mouse\_wait, wait);

spin\_lock\_irq(&mouse\_lock);

/\* See if data has arrived from the device or

if the device is ready to accept more data \*/

/\* ... \*/

spin\_unlock\_irq(&mouse\_lock);

/\* Availability of data is detected from interrupt context \*/

if (data\_is\_available()) return(POLLIN | POLLRDNORM);

/\* Data can be written. Not relevant for mice \*/

if (data\_can\_be\_written()) return(POLLOUT | POLLWRNORM);

return 0;

}

当Xf86WaitForInput()调用select()，通用内核的轮询实现（定义于fs/select.c）调用mouse\_poll()。mouse\_poll()有两个参数，常见的文件指针（struct file \*）和一个指向内核数据结构poll\_table的指针。poll\_table是一个等待队列表，由被轮询等待数据的设备驱动所拥有。

mouse\_poll()使用库函数poll\_wait()，添加一个等待队列（mouse\_wait）给内核poll\_table后休眠。正如你在第三章“内核组件”所看到的，设备驱动通常持有几个等待队列，这些队列阻塞直至检测到数据状态的变化。数据状态的改变可能是从设备上有新数据到来，驱动自动将新数据传送给应用程序；也可能是设备（或者驱动）准备就绪接收新数据。这些状态通常（但不是永远）被驱动的中断处理例程所检测。当鼠标驱动的中断处理例程检测到鼠标移动时，它调用wake\_up\_interruptible(&mouse\_wait)以唤醒处于休眠中的mouse\_poll()。

如果数据状态没有变化，poll()方法返回0。如果驱动准备好发送至少一个字节数据给应用程序，它返回POLLIN|POLLRDNORM。如果驱动已经准备好从应用程序接收至少一个字节数据，它将返回POLLOUT|POLLWRNORM[2]。因此鼠标如果没有移动，mouse\_poll()返回0，同时调用线程被设置成休眠状态。当鼠标中断处理例程检测到设备数据变化，唤醒mouse\_wait队列后，内核重新调用mouse\_poll()。运行一定时间，mouse\_poll() 返回 POLLIN|POLLRDNORM，因此select()调用，Xf86WaitForInput()返回正值。X server的鼠标处理例程（xc/programs/Xserver/hw/xfree86/input/mouse/mouse.c）继续从鼠标读取数据。

对设备驱动进行轮询的用户应用程序通常更关注于驱动的特性而非设备的特性。譬如，由于使用了缓冲区，在设备可以接收数据之前，驱动可能已经可以从应用程序接收新数据了。

Fasync

一些应用程序，因为性能的原因，希望设备驱动异步通知它们。假设起搏器设备上基于Linux的应用程序忙于处理复杂的计算，但希望当植入的起搏器有数据到达时能够及时得到通知。在这种情况下select()/poll()机制由于阻塞于计算而不可用。此时应用程序需要的是异步事件报告。如果telemetry驱动检测到从pacemaker来的数据后，能够马上异步分发一个信号（通常是SIGIO），应用程序就可以使用信号处理程序捕获它，同时驱动相应的代码流。

作为实际的异步通知的例子，让我们看看当X server检测到输入设备有数据到来时的处理过程。以下片断来自于X server源代码：

Code View:

xc/programs/Xserver/hw/xfree86/os-support/shared/sigio.c:

int xf86InstallSIGIOHandler(int fd, void (\*f)(int, void \*),

void \*closure)

{

struct sigaction sa;

struct sigaction osa;

if (fcntl(fd, F\_SETOWN, getpid()) == -1) {

blocked = xf86BlockSIGIO();

/\* O\_ASYNC is defined as SIGIO elsewhere by the X server \*/

if (fcntl(fd, F\_SETFL, fcntl(fd, F\_GETFL) | O\_ASYNC) == -1) {

xf86UnblockSIGIO(blocked); return 0;

}

sigemptyset(&sa.sa\_mask);

sigaddset(&sa.sa\_mask, SIGIO);

sa.sa\_flags = 0;

sa.sa\_handler = xf86SIGIO;

sigaction(SIGIO, &sa, &osa);

/\* ... \*/

return 0;

}

static void

xf86SIGIO(int sig)

{

/\* Identify the device that triggered generation of this

SIGIO and handle the data arriving from it \*/

/\* ... \*/

}

正如你从以上片断中所看到的，X server做了如下事情：

· 调用fcntl(F\_SETOWN)。fcntl()系统调用用于处理文件描述符的行为。F\_SETOWN设置调用进程对描述符的拥有关系。由于内核需要知道在哪里发送异步信号，因此这一步是必须的。这一步对于设备驱动是透明的。

· 调用fcntl(F\_SETFL)。任何时候只要有数据被读取，或者驱动准备好接收更多的用户程序数据，F\_SETFL请求驱动发送SIGIO信号给应用程序。fcntl(F\_SETFL)调用导致fasync()驱动方法的调用。fasync()的责任是负责从将被发送SIGIO信号的进程列表里添加或删除条目。最后，fasync()利用内核库函数提供的服务fasync\_helper()。

· 按照其代码结构实现SIGIO信号处理例程xf86SIGIO()，并使用sigaction()系统调用完成安装。当底层的输入设备驱动检测到数据状态的改变，它发送SIGIO给注册的请求，激发xf86SIGIO()的执行[3]。字符驱动调用kill\_fasync()发送SIGIO给注册的进程。为了通知一个读事件，POLLIN作为kill\_fasync()的参数。相应的写事件传递的参数是POLLOUT。

为了理解驱动端的异步通知链的实现机制，让我们看看一个虚构的输入设备驱动的fasync()方法：

Code View:

/\* This is invoked by the kernel when the X server opens this

\* input device and issues fcntl(F\_SETFL) on the associated file

\* descriptor. fasync\_helper() ensures that if the driver issues a

\* kill\_fasync(), a SIGIO is dispatched to the owning application.

\*/

static int

inputdevice\_fasync(int fd, struct file \*filp, int on)

{

return fasync\_helper(fd, filp, on, &inputdevice\_async\_queue);

}

/\* Interrupt Handler \*/

irqreturn\_t

inputdevice\_interrupt(int irq, void \*dev\_id)

{

/\* ... \*/

/\* Dispatch a SIGIO using kill\_fasync() when input data is

detected. Output data is not relevant since this is a read-only

device \*/

wake\_up\_interruptible(&inputdevice\_wait);

kill\_fasync(&inputdevice\_async\_queue, SIGIO, POLL\_IN);

/\* ... \*/

return IRQ\_HANDLED;

}

为了理解SIGIO的传送机理，考虑tty驱动的例子（在第六章“串行驱动”中讨论）。设定的应用程序在如下情形下得到通知：

· 如果底层的驱动未准备就绪接收应用程序数据，它将调用进程置为休眠。当驱动中断处理例程随后发现可以接收更多的数据时，它唤醒应用程序并调用kill\_fasync(POLLOUT)。

· 如果收到一个“新起一行”(newline)字符，tty层调用kill\_fasync(POLLIN)。

· 当驱动在检测到足够的超过门限的数据从设备到来后，唤醒一个休眠的读线程，通过调用kill\_fasync（POLLIN）发送信息给对应进程。

和并口交互

并口是在PC兼容系统上普遍可见的25针接口。并口的能力（是单向还是双向，是否支持DMA等）依赖于芯片组。从第4章的图4.1可以看出PC结构如何支持并口。

drivers/parport/目录包括IEEE 1284并口通信的具体实现代码（称为parport）。几个联接到并口的设备像打印机、扫描仪使用parport的服务。parport有一个结构无关的模块parport.ko和一个结构相关的模块（PC结构称为parport\_pc.ko）提供可编程接口给并口设备驱动。

让我们看看并行打印机驱动的例子：drivers/char/lp.c。打印一个文件需要完成如下“高层”步骤：

1.

打印机驱动创建字符设备节点/dev/lp0到/dev/lpN，每一个连接到一个打印机。

2.

通用UNIX打印系统（Common UNIX Printing System，CUPS）是Linux上提供打印功能的框架。CUPS配置文件（在某些发布版上为/etc/printers.conf）完成字符设备节点到打印机的映射（/dev/lpX）

3.

CUPS根据配置文件，将数据流流向相应的设备节点。因此，如果在你的系统上有打印机连接到第一个并口，并且键入命令：lpr myfile，数据流将会通过/dev/lp0流向打印机的write()方法，即定义在drivers/char/lp.c中的lp\_write()。

4.

lp\_write()用parport的提供的服务将数据发送至打印机。

Apple公司已经获得了CUPS软件的拥有权。这些代码在GPLv2下继续被授权使用。

字符驱动ppdev（drivers/char/ppdev.c）输出/dev/parportX设备节点，使用户应用程序可以直接和并口通信（在第19章中我们将详细讨论ppdev）。

设备例子：并口LED板

为了学习parport提供的服务，让我们编写一个简单的驱动。考虑一个有8个发光二极管（LED）、提供和标准25针并口接口的电路板。因为PC上的8bit并口数据寄存器直接映射到并口的2～9针，这些针脚和电路板上的LED连通。向并口数据寄存器写数据可以控制这些针脚的电平，进而控制LED的开关。清单5.6实现了一个字符设备驱动，它通过系统并口和此电路板通信。代码内的注释解释了其中所使用的parport服务例程。

清单 5.6. 并口LED电路板驱动（led.c）

Code View:

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/parport.h>

#include <asm/uaccess.h>

#include <linux/platform\_device.h>

#define DEVICE\_NAME "led"

static dev\_t dev\_number; /\* Allotted device number \*/

static struct class \*led\_class; /\* Class to which this device

belongs \*/

struct cdev led\_cdev; /\* Associated cdev \*/

struct pardevice \*pdev; /\* Parallel port device \*/

/\* LED open \*/

int

led\_open(struct inode \*inode, struct file \*file)

{

return 0;

}

/\* Write to the LED \*/

ssize\_t

led\_write(struct file \*file, const char \*buf,

size\_t count, loff\_t \*ppos)

{

char kbuf;

if (copy\_from\_user(&kbuf, buf, 1)) return -EFAULT;

/\* Claim the port \*/

parport\_claim\_or\_block(pdev);

/\* Write to the device \*/

parport\_write\_data(pdev->port, kbuf);

/\* Release the port \*/

parport\_release(pdev);

return count;

}

/\* Release the device \*/

int

led\_release(struct inode \*inode, struct file \*file)

{

return 0;

}

/\* File Operations \*/

static struct file\_operations led\_fops = {

.owner = THIS\_MODULE,

.open = led\_open,

.write = led\_write,

.release = led\_release,

};

static int

led\_preempt(void \*handle)

{

return 1;

}

/\* Parport attach method \*/

static void

led\_attach(struct parport \*port)

{

/\* Register the parallel LED device with parport \*/

pdev = parport\_register\_device(port, DEVICE\_NAME,

led\_preempt, NULL,

NULL, 0, NULL);

if (pdev == NULL) printk("Bad register\n");

}

/\* Parport detach method \*/

static void

led\_detach(struct parport \*port)

{

/\* Do nothing \*/

}

/\* Parport driver operations \*/

static struct parport\_driver led\_driver = {

.name = "led",

.attach = led\_attach,

.detach = led\_detach,

};

/\* Driver Initialization \*/

int \_\_init

led\_init(void)

{

/\* Request dynamic allocation of a device major number \*/

if (alloc\_chrdev\_region(&dev\_number, 0, 1, DEVICE\_NAME)

< 0) {

printk(KERN\_DEBUG "Can't register device\n");

return -1;

}

/\* Create the led class \*/

led\_class = class\_create(THIS\_MODULE, DEVICE\_NAME);

if (IS\_ERR(led\_class)) printk("Bad class create\n");

/\* Connect the file operations with the cdev \*/

cdev\_init(&led\_cdev, &led\_fops);

led\_cdev.owner = THIS\_MODULE;

/\* Connect the major/minor number to the cdev \*/

if (cdev\_add(&led\_cdev, dev\_number, 1)) {

printk("Bad cdev add\n");

return 1;

}

class\_device\_create(led\_class, NULL, dev\_number,

NULL, DEVICE\_NAME);

/\* Register this driver with parport \*/

if (parport\_register\_driver(&led\_driver)) {

printk(KERN\_ERR "Bad Parport Register\n");

return -EIO;

}

printk("LED Driver Initialized.\n");

return 0;

}

/\* Driver Exit \*/

void \_\_exit

led\_cleanup(void)

{

unregister\_chrdev\_region(MAJOR(dev\_number), 1);

class\_device\_destroy(led\_class, MKDEV(MAJOR(dev\_number), 0));

class\_destroy(led\_class);

return;

}

module\_init(led\_init);

module\_exit(led\_cleanup);

MODULE\_LICENSE("GPL");

除了如下两点，led\_init() 类似于清单5.1中的cmos\_init()：

1. 正如第4章中所看到的，新的设备模型将驱动和设备区分开来。led\_init()通过parport\_register\_driver()调用向parport注册LED驱动。当内核在led\_attach()中查找到LED板时，它调用parport\_register\_device()注册设备。

2. led\_init()创建设备节点/dev/led，你可以此设备节点控制每个LED的状态。

编译并将驱动模块加入到内核中：

bash> make –C /path/to/kerneltree/ M=$PWD modules

bash> insmod ./led.ko

LED Driver Initialized

为了有选择地驱动一些并口针脚，点亮相应的LED，赋相应的值给/dev/led：

bash> echo 1 > /dev/led

因为1的ASCII值时31（00110001），第1、5和6个LED将会发亮。

前述的命令触发led\_write()调用。此驱动方法首先通过copy\_from\_user()拷贝用户内存数据（在本例中为31）到内核缓冲区。然后占用并口，写入数据，释放端口，所有这些都使用parport接口。

相比于/dev，sysfs是更好的控制设备状态的地方。因此将LED控制委托给sysfs效果更佳。清单5.7包含了此种驱动的实现。清单中的sysfs操作代码也可作为模板，用于其它的设备控制中去。

清单 5.7. 使用Sysfs 控制并口LED电路板

Code View:

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/parport.h>

#include <asm/uaccess.h>

#include <linux/pci.h>

static dev\_t dev\_number; /\* Allotted Device Number \*/

static struct class \*led\_class; /\* Class Device Model \*/

struct cdev led\_cdev; /\* Character dev struct \*/

struct pardevice \*pdev; /\* Parallel Port device \*/

struct kobject kobj; /\* Sysfs directory object \*/

/\* Sysfs attribute of the leds \*/

struct led\_attr {

struct attribute attr;

ssize\_t (\*show)(char \*);

ssize\_t (\*store)(const char \*, size\_t count);

};

#define glow\_show\_led(number) \

static ssize\_t \

glow\_led\_##number(const char \*buffer, size\_t count) \

{ \

unsigned char buf; \

int value; \

\

sscanf(buffer, "%d", &value); \

\

parport\_claim\_or\_block(pdev); \

buf = parport\_read\_data(pdev->port); \

if (value) { \

parport\_write\_data(pdev->port, buf | (1<<number)); \

} else { \

parport\_write\_data(pdev->port, buf & ~(1<<number)); \

} \

parport\_release(pdev); \

return count; \

} \

\

static ssize\_t \

show\_led\_##number(char \*buffer) \

{ \

unsigned char buf; \

\

parport\_claim\_or\_block(pdev); \

\

buf = parport\_read\_data(pdev->port); \

parport\_release(pdev); \

\

if (buf & (1 << number)) { \

return sprintf(buffer, "ON\n"); \

} else { \

return sprintf(buffer, "OFF\n"); \

} \

} \

\

static struct led\_attr led##number = \

\_\_ATTR(led##number, 0644, show\_led\_##number, glow\_led\_##number);

glow\_show\_led(0); glow\_show\_led(1); glow\_show\_led(2);

glow\_show\_led(3); glow\_show\_led(4); glow\_show\_led(5);

glow\_show\_led(6); glow\_show\_led(7);

#define DEVICE\_NAME "led"

static int

led\_preempt(void \*handle)

{

return 1;

}

/\* Parport attach method \*/

static void

led\_attach(struct parport \*port)

{

pdev = parport\_register\_device(port, DEVICE\_NAME,

led\_preempt, NULL, NULL, 0,

NULL);

if (pdev == NULL) printk("Bad register\n");

}

/\* Parent sysfs show() method. Calls the show() method

corresponding to the individual sysfs file \*/

static ssize\_t

l\_show(struct kobject \*kobj, struct attribute \*a, char \*buf)

{

int ret;

struct led\_attr \*lattr = container\_of(a, struct led\_attr,attr);

ret = lattr->show ? lattr->show(buf) : -EIO;

return ret;

}

/\* Sysfs store() method. Calls the store() method

corresponding to the individual sysfs file \*/

static ssize\_t

l\_store(struct kobject \*kobj, struct attribute \*a,

const char \*buf, size\_t count)

{

int ret;

struct led\_attr \*lattr = container\_of(a, struct led\_attr, attr);

ret = lattr->store ? lattr->store(buf, count) : -EIO;

return ret;

}

/\* Sysfs operations structure \*/

static struct sysfs\_ops sysfs\_ops = {

.show = l\_show,

.store = l\_store,

};

/\* Attributes of the /sys/class/pardevice/led/control/ kobject.

Each file in this directory corresponds to one LED. Control

each LED by writing or reading the associated sysfs file \*/

static struct attribute \*led\_attrs[] = {

&led0.attr,

&led1.attr,

&led2.attr,

&led3.attr,

&led4.attr,

&led5.attr,

&led6.attr,

&led7.attr,

NULL

};

/\* This describes the kobject. The kobject has 8 files, one

corresponding to each LED. This representation is called the

ktype of the kobject \*/

static struct kobj\_type ktype\_led = {

.sysfs\_ops = &sysfs\_ops,

.default\_attrs = led\_attrs,

};

/\* Parport methods. We don't have a detach method \*/

static struct parport\_driver led\_driver = {

.name = "led",

.attach = led\_attach,

};

/\* Driver Initialization \*/

int \_\_init

led\_init(void)

{

struct class\_device \*c\_d;

/\* Create the pardevice class - /sys/class/pardevice \*/

led\_class = class\_create(THIS\_MODULE, "pardevice");

if (IS\_ERR(led\_class)) printk("Bad class create\n");

/\* Create the led class device - /sys/class/pardevice/led/ \*/

c\_d = class\_device\_create(led\_class, NULL, dev\_number,

NULL, DEVICE\_NAME);

/\* Register this driver with parport \*/

if (parport\_register\_driver(&led\_driver)) {

printk(KERN\_ERR "Bad Parport Register\n");

return -EIO;

}

/\* Instantiate a kobject to control each LED

on the board \*/

/\* Parent is /sys/class/pardevice/led/ \*/

kobj.parent = &c\_d->kobj;

/\* The sysfs file corresponding to kobj is

/sys/class/pardevice/led/control/ \*/

strlcpy(kobj.name, "control", KOBJ\_NAME\_LEN);

/\* Description of the kobject. Specifies the list of attribute

files in /sys/class/pardevice/led/control/ \*/

kobj.ktype = &ktype\_led;

/\* Register the kobject \*/

kobject\_register(&kobj);

printk("LED Driver Initialized.\n");

return 0;

}

/\* Driver Exit \*/

void

led\_cleanup(void)

{

/\* Unregister kobject corresponding to

/sys/class/pardevice/led/control \*/

kobject\_unregister(&kobj);

/\* Destroy class device corresponding to

/sys/class/pardevice/led/ \*/

class\_device\_destroy(led\_class, MKDEV(MAJOR(dev\_number), 0));

/\* Destroy /sys/class/pardevice \*/

class\_destroy(led\_class);

return;

}

module\_init(led\_init);

module\_exit(led\_cleanup);

MODULE\_LICENSE("GPL");

清单5.7中定义的宏glow\_show\_led()使用了内核源代码中经常使用的技术，以便简洁地定义几个类似的函数。定义的read()和write()方法（在sysfs中用术语show()和store()表示）关联于八个/sys文件，电路板上的每个LED一个。因此glow\_show\_led(0)将glow\_led\_0()和show\_led\_0()关联于第一个LED对应的/sys文件。这些函数分别负责点亮/熄灭第一个LED，并读取其状态。##在宏定义中用于把字符串连接在一起，因此当编译器处理语句glow\_show\_led(0)时，glow\_led\_##number预处理成glow\_led\_0()。

基于sysfs版本的驱动使用了kobject，用于代表“控制”抽象，它模拟了一个软件按钮控制LED。sysfs下的每个目录名代表一个kobject，因此清单5.7中的kobject\_register()导致了/sys/class/pardevice/led/control/目录的创建。

ktype描述了kobject。ktype\_led结构描述了“控制”kobject，它包含指向属性数组的指针led\_attrs[]。led\_attrs[]数组包含每个LED的设备属性的地址。每个LED的属性通过下面语句连接在一起：

static struct led\_attr led##number =

\_\_ATTR(led##number, 0644, show\_led\_##number, glow\_led\_##number);

其结果是为每个LED产生一个控制文件的实例/sys/class/pardevice/led/control/ledX，其中X是LED序号。为了改变ledX的状态，echo 1（或者0）给相应的控制文件。如为了点亮第一个LED，可做如下操作：

bash> echo 1 > /sys/class/pardevice/led/control/led0

在模块退出期间，驱动使用kobject\_unregister()，class\_device\_destroy()，和class\_destroy()移除kobject和class。

第7章《输入设备驱动》的清单7.2，使用另一套方法在sysfs中创建文件。

编写字符驱动不再像2.4内核中那样简单了。上面为了开发简单的LED驱动，我们使用了半打的数据抽象：cdev，sysfs，kobjects，classes，class device，和parport。当然，这些数据抽象也为我们带来了几个好处，如：无bug的编译模块，代码重用以及优美的设计。

RTC 子系统

内核中对RTC的支持分成两层：硬件无关的顶层字符驱动，用于实现内核的RTC API；以及硬件相关的底层驱动，用于和底层的总线通信。RTC API在Documentation/rtc.txt中定义，是一些标准ioctl的集合，hwclock等应用程序遵守这些接口对/dev/rtc进行操作。

这些API也定义了sysfs（/sys/class/rtc/）和procfs（/proc/driver/rtc）中的属性。 RTC API保证用户空间的工具独立于底层平台和RTC芯片。底层的RTC驱动由总线决定。嵌入式设备通过RTC芯片和I2C总线相连，由I2C客户驱动程序来驱动，其细节将在第8章“I2C协议”的“设备例子：实时时钟”章节中讨论。

内核有一个专门的RTC子系统，提供了顶层的字符驱动，并给出了用于顶层和底层RTC驱动进行捆绑的核心基础结构，核心基础结构的主要组成部分是rtc\_class\_ops结构和注册函数rtc\_device\_[register|unregister]()。分散在不同的总线有关的目录下的底层的RTC驱动通过此子系统统一在drivers/rtc/下。

RTC子系统使系统可以拥有一个以上的RTC。它通过提供多个/dev/rtcN和/sys/class/rtc/rtcN接口实现此功能，其中N是RTC在系统上的序号。例如一些嵌入式系统有两个RTC：一个集成在微控制器上用于支持一些复杂的操作，如产生周期性中断；另一个使用低功耗电池的外部RTC用于时钟保持。由于RTC相关的应用通过/dev/rtc操作，因此需要创建符号链接，以便某个被创建的/dev/rtcX节点可以通过/dev/rtc来访问。

为了使能RTC子系统，在内核配置过程中需要选中CONFIG\_RTC\_CLASS配置选项。

Legacy PC RTC 驱动

PC系统上，通过使用legacy RTC驱动drivers/char/rtc.c，可以将RTC子系统旁路。在PC兼容的系统上，此驱动为RTC提供顶层和底层驱动，并提供/dev/rtc和/proc/driver/rtc给用户程序。为了使能此驱动，在内核配置过程中需要选中CONFIG\_RTC选项。

伪字符驱动

有几个常用的内核工具没有和任何物理硬件相连接，它们被灵巧的实现为字符设备。null池，0产生源和内核随机数产生器被当作虚拟设备，并使用伪字符设备驱动来访问。

/dev/null字符设备接收你不想显示在屏幕上的数据。因此如果你需要从CVS库中提取源文件，而又不想将文件名显示在整个屏幕上时，可以做如下操作：

bash> cvs co kernel > /dev/null

此操作将命令输出重定向到属于/dev/null驱动的写入口点。此驱动的read()和write()方法只是简单的返回success，而忽略相应缓冲区中的输入和输出内容。

如果你想用0来填充一个图像文件时，可以调用/dev/zero来达到目的：

bash> dd if=/dev/zero of=file.img bs=1024 count=1024

它从/dev/zero驱动的read()方法中获取一串0。此驱动没有write()方法。

内核有一个内部的随机数发生器。当内核用户希望使用随机序列时，随机数发生器提供API，如get\_random\_bytes()。对于用户模式的程序，它提供两个字符接口：/dev/random 和 /dev/urandom。从/dev/random读取的随机数的随机性要高于/dev/urandom。当用户程序从/dev/random读取时，可获得近乎随机（或真正）的随机数，而从/dev/urandom读取的为伪随机数。/dev/random驱动并未使用公式去产生近乎随机的随机数，而是收集“环境噪声”（中断间隔、键盘敲击等等）维持一个混乱池（称为熵池）以产生随机流。为了了解内核输入子系统（在第7章讨论）在检测到按键和鼠标移动时对熵池的作用，让我们看看在drivers/input/input.c中定义的input\_event()函数：

void

input\_event(struct input\_dev \*dev, unsigned int type,

unsigned int code, int value)

{

/\* ... \*/

add\_input\_randomness(type, code, value); /\* Contribute to entropy

pool \*/

/\* ... \*/

}

为了了解核心的中断处理层通过内部中断周期对熵池所起的作用，让我们阅读定义于kernel/irq/handle.c中的handle\_IRQ\_event()函数：

irqreturn\_t handle\_IRQ\_event(unsigned int irq,

struct irqaction \*action)

{

/\* ... \*/

if (status & IRQF\_SAMPLE\_RANDOM)

add\_interrupt\_randomness(irq); /\* Contribute to entropy pool \*/

/\* ... \*/

}

近乎随机的随机数的产生依赖于熵池的大小：

bash> od –x /dev/random

0000000 7331 9028 7c89 4791 7f64 3deb 86b3 7564

0000020 ebb9 e806 221a b8f9 af12 cb30 9a0e cc28

0000040 68d8 0bbf 68a4 0898 528e 1557 d8b3 57ec

0000060 b01d 8714 b1e1 19b9 0a86 9f60 646c c269

在输出一定行数后停止输出，表明熵池已经耗尽。为了补充熵池并重新输出随机的数据流，可以在切换至一个未用的终端后快速敲击键盘，或者沿屏幕按动鼠标。

而/dev/ urandom则会输出连续的、不会停止的伪随机数流。

/dev/mem 和 /dev/ kmem是典型的伪字符设备，它们提供了查看系统内存的工具。这些字符节点提供原始的接口，分别与物理内存和内核虚拟内存相联。为了使用系统内存，你可以mmap()这些节点，并在返回的区域执行操作。作为一个练习，尝试着通过访问/dev/mem以修改你的系统的主机名。

在本节中讨论的所有字符设备（null，zero，random，urandom，mem和kmem）拥有不同的次设备号，但拥有静态分配的相同的主设备号：1。阅读drivers/char/mem.c和drivers/char/random.c可了解其具体实现。两个其它的伪驱动属于同一个主设备号族：其中/dev/full模拟一个总是处于满的设备，/dev/port查看系统的I/O端口。在第19章中我们将会用到。

混杂驱动

混杂驱动是那些简单的字符驱动，它们共享一些相同的特性。内核将这些共同性抽象至一个API中（具体实现在drivers/char/misc.c），这简化了这些驱动初始化的方式。所有的混杂设备都被分配一个主设备号10，但每个设备可选择一个单独的次设备号。因此，假如像我们前面讨论的CMOS例子中，一个字符驱动需要驱动多个设备，混杂驱动将不会适用。

考虑一个字符驱动完成初始化的顺序：

· 通过alloc\_chrdev\_region()及相关函数分配主/次设备号

· 使用class\_device\_create()创建 /dev 和 /sys 节点

· 使用cdev\_init() 和 cdev\_add()注册自身为字符驱动

混杂驱动通过一个简单的misc\_register()调用完成以上所有步骤：

static struct miscdevice mydrv\_dev = {

MYDRV\_MINOR,

"mydrv",

&mydrv\_fops

};

misc\_register(&mydrv\_dev);

在前述的例子中，MYDRV\_MINOR是你准备静态分配给混杂驱动的次设备号。你也可以在mydrv\_dev结构中通过指定MISC\_DYNAMIC\_MINOR而不是MYDRV\_MINOR以要求动态分配次设备号。

每一个混杂驱动自动出现在/sys/class/misc/，而不必驱动编写者做额外的工作。由于混杂驱动是字符驱动，前面讨论的有关字符驱动入口点同样适用于混杂驱动。让我们看一个混杂驱动的实例。

设备例子：看门狗定时器

看门狗的功能是将失去响应的系统重新设置成可操作的状态。它周期性的检查系统脉搏，若检测不到则对系统重启[4]。应用软件负责注册此脉搏（或者心跳）并利用看门狗设备驱动提供的服务周期性“喂狗”。大多数嵌入式控制器内嵌了看门狗模块。也可使用外部的看门狗芯片，例如Netwinder的W83977AF芯片。

Linux看门狗驱动使用misc驱动来实现，位于drivers/char/watchdog/目录。看门狗驱动类似于RTC驱动，提供给用户标准的设备接口，因此，应用程序可以独立于看门狗具体硬件实现。此API位于内核源码树Documentation/watchdog/watchdog-api.txt。系统看门狗服务的程序可在/dev/watchdog上操作，设备节点的misc次设备号为130。

清单5.9实现了一个集成于嵌入式控制器的、构想的看门狗模块设备驱动。在此例中，看门狗包括两个主要的寄存器（见表5.2）：一个服务寄存器（WD\_SERVICE\_REGISTER）和一个控制寄存器（WD\_CONTROL\_REGISTER）。为了喂狗，驱动程序向服务寄存器写入了一个特定的序列（0xABCD）。为了对看门狗超时编程，驱动程序在控制寄存器中编写了特定的比特位。

表 5.2. 看门狗模块寄存器分布

寄存器名

描述

WD\_SERVICE\_REGISTER

写入特定的序列至此寄存器以喂狗。

WD\_CONTROL\_REGISTER

写入看门狗超时时间至此寄存器。

使用看门狗的目的是检测并响应应用程序和内核的挂起，因此一般在用户空间喂狗。清单5.10中，某个关键的应用程序[5]，如图形引擎打开了清单5.9中提供的看门狗驱动，周期性的向其写入数据。如果由于某个应用程序挂起或由于内核崩溃，在看门狗的超时期限内没有写入数据，看门狗将触发系统重启。在清单5.10中，当出现如下情况时，看门狗将重启系统：

· 应用程序在process\_graphics()内挂起

· 内核崩溃，从而引起应用程序死亡

当应用程序打开/dev/watchdog时，看门狗开始计数。关闭设备节点将停止看门狗，除非在内核配置期间你设置了CONFIG\_WATCHDOG\_NOWAYOUT选项。当系统仍在运行时，看门狗监控进程（例如清单5.10）有可能被某个信号杀死，设置此选项可帮助你消除此可能。

清单 5.9. 看门狗定时器例子（An Example Watchdog Driver）

Code View:

#include <linux/miscdevice.h>

#include <linux/watchdog.h>

#define DEFAULT\_WATCHDOG\_TIMEOUT 10 /\* 10-second timeout \*/

#define TIMEOUT\_SHIFT 5 /\* To get to the timeout field

in WD\_CONTROL\_REGISTER \*/

#define WENABLE\_SHIFT 3 /\* To get to the

watchdog-enable field in

WD\_CONTROL\_REGISTER \*/

/\* Misc structure \*/

static struct miscdevice my\_wdt\_dev = {

.minor = WATCHDOG\_MINOR, /\* defined as 130 in

include/linux/miscdevice.h\*/

.name = "watchdog", /\* /dev/watchdog \*/

.fops = &my\_wdt\_dog /\* Watchdog driver entry points \*/

};

/\* Driver methods \*/

struct file\_operations my\_wdt\_dog = {

.owner = THIS\_MODULE,

.open = my\_wdt\_open,

.release = my\_wdt\_close,

.write = my\_wdt\_write,

.ioctl = my\_wdt\_ioctl

}

/\* Module Initialization \*/

static int \_\_init

my\_wdt\_init(void)

{

/\* ... \*/

misc\_register(&my\_wdt\_dev);

/\* ... \*/

}

/\* Open watchdog \*/

static void

my\_wdt\_open(struct inode \*inode, struct file \*file)

{

/\* Set the timeout and enable the watchdog \*/

WD\_CONTROL\_REGISTER |= DEFAULT\_WATCHDOG\_TIMEOUT << TIMEOUT\_SHIFT;

WD\_CONTROL\_REGISTER |= 1 << WENABLE\_SHIFT;

}

/\* Close watchdog \*/

static int

my\_wdt\_close(struct inode \*inode, struct file \*file)

{

/\* If CONFIG\_WATCHDOG\_NOWAYOUT is chosen during kernel

configuration, do not disable the watchdog even if the

application desires to close it \*/

#ifndef CONFIG\_WATCHDOG\_NOWAYOUT

/\* Disable watchdog \*/

WD\_CONTROL\_REGISTER &= ~(1 << WENABLE\_SHIFT);

#endif

return 0;

}

/\* Pet the dog \*/

static ssize\_t

my\_wdt\_write(struct file \*file, const char \*data,

size\_t len, loff\_t \*ppose)

{

/\* Pet the dog by writing a specified sequence of bytes to the

watchdog service register \*/

WD\_SERVICE\_REGISTER = 0xABCD;

}

/\* Ioctl method. Look at Documentation/watchdog/watchdog-api.txt

for the full list of ioctl commands. This is standard across

watchdog drivers, so conforming applications are rendered

hardware-independent \*/

static int

my\_wdt\_ioctl(struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg)

{

/\* ... \*/

switch (cmd) {

case WDIOC\_KEEPALIVE:

/\* Write to the watchdog. Applications can invoke

this ioctl instead of writing to the device \*/

WD\_SERVICE\_REGISTER = 0xABCD;

break;

case WDIOC\_SETTIMEOUT:

copy\_from\_user(&timeout, (int \*)arg, sizeof(int));

/\* Set the timeout that defines unresponsiveness by

writing to the watchdog control register \*/

WD\_CONTROL\_REGISTER = timeout << TIMEOUT\_BITS;

break;

case WDIOC\_GETTIMEOUT:

/\* Get the currently set timeout from the watchdog \*/

/\* ... \*/

break;

default:

return –ENOTTY;

}

}

/\* Module Exit \*/

static void \_\_exit

my\_wdt\_exit(void)

{

/\* ... \*/

misc\_deregister(&my\_wdt\_dev);

/\* ... \*/

}

module\_init(my\_wdt\_init);

module\_exit(my\_wdt\_exit);

清单 5.10. 看门狗用户（A Watchdog User）

#include <fcntl.h>

#include <asm/types.h>

#include <linux/watchdog.h>

int

main()

{

int new\_timeout;

int wfd = open("/dev/watchdog", O\_WRONLY);

/\* Set the watchdog timeout to 20 seconds \*/

new\_timeout = 20;

ioctl(fd, WDIOC\_SETTIMEOUT, &new\_timeout);

while (1) {

/\* Graphics processing \*/

process\_graphics();

/\* Pet the watchdog \*/

ioctl(fd, WDIOC\_KEEPALIVE, 0);

/\* Or instead do: write(wfd, "\0", 1); \*/

fsync(wfd);

}

}

外部看门狗

为了确保系统在处理器故障时仍然能够进行恢复，即使主处理器上集成有强大的看门狗模块（例如我们的例子中），一些规则制定者规定要使用外部看门狗芯片。由于此要求，一些嵌入式设备有时使用较为便宜的、简单的看门狗芯片（例如Maxim的MAX6730），外部的看门狗芯片通过硬件连线发挥作用，而不像片上集成看门狗，通过寄存器接口产生效果。看门狗的输入引脚在固定的复位超时时间内如果没有检测到电压脉冲，就会设置复位引脚。复位引脚和处理器的复位逻辑相连，而输入引脚和处理器的通用目的I/O端口（GPIO）相连。软件必须周期性的在芯片的复位超时时间内向输入引脚输送脉冲，以免看门狗复位。如果你为此类设备编写驱动，ioctl()方法可忽略。当应用软件需要向相应的设备节点写入数据时，就会利用驱动程序提供的write()方法向输入引脚输送脉冲。为了帮助生产和现场排障，看门狗通过导线和处理器相连，这样可以通过断开GPIO引脚和看门狗的连接来停用看门狗。

考虑到引导时间，外部看门狗芯片通常允许较长的初始超时时间，随后的超时时间会变短。

对于那些不提供硬件看门狗模块的平台，内核实现了一个软件看门狗，也称为softdog。softdog驱动是drivers/char/watchdog/softdog.c，由于不对具体硬件进行操作，所以是一个伪混杂驱动。softdog驱动必须完成两件硬件看门狗驱动不必做的任务，其中第二个任务通过硬件完成：

· 实现超时机制

· 如果系统无响应，启动软复位

只要应用程序向软件看门狗写入数据，就会延迟定时器处理程序的执行。如果在超时时间内没有数据写入软件看门狗，定时器处理程序就会触发并重启系统。

2.6内核中相关的支持是软件锁定的检测，当10秒或更长时间未发生调度时，就会发生。内核线程watchdog/N每秒访问CPU的时戳一次，其中N是CPU号。如果线程超过10秒没有访问时戳，系统必定已被锁定。软件锁定检测（在kernel/softlockup.c中实现）可帮助我们调试内核崩溃，第21章《设备驱动的调试》的“Kdump”一节中将进行讨论。

内核中还有几个混杂驱动。Qtronix红外键盘驱动（drivers/char/qtronix.c）是另一个具备混杂特性的字符驱动。在drivers/char/目录下运行“grep misc\_register()”命令可找到内核中其他的混杂设备。

Character Caveats

驱动的方法，也就是被用户程序调用的系统调用，可能失败或部分成功。你的应用程序必须能意识到这些情况而非莫名惊诧。让我们看看一些常见的问题：

· open()调用可能由于几个原因而失败。某些字符驱动在同一时间仅支持单用户，因此当应用程序试图打开一个已经在使用的设备时就会返回-EBUSY失败。当打印机缺纸时，如果你进行open()操作就会返回-ENOSPC失败。

· read()和write()可能成功返回1字节至请求的字节，因此你的应用程序需要充分考虑以处理这些情况。

· 即使仅仅1个字节的数据读或写就绪，select()也会返回成功。

· 一些设备，例如鼠标和触摸屏只能输入，因此其驱动不支持写方法族（write()/aio\_write()/fsync()），另一些设备是仅供输出的，因此其驱动不支持读方法族（read()/aio\_read()）。同样，很多字符驱动方法是可选的，因此在所有的驱动中并不是所有的方法都提供。调用未提供的方法就会返回失败。

查看源代码

字符驱动并不仅仅在drivers/char/目录下。这里有一些“超级”字符驱动的例子，它们受到特别的对待，其目录为：

· 串行驱动是管理计算机串口的字符驱动。然而，它们不仅仅是简单的字符驱动，被分开放在drivers/serial/目录下。下一章将会讨论串行驱动。

· 输入驱动负责像键盘、鼠标和操纵杆这样的设备。它们位于单独的源文件目录：drivers/input/。第7章将会专门讨论。

· 帧缓冲区（/dev/fb/\*）提供对显存的访问，/dev/mem提供对系统内存的访问途径。第12章“显卡驱动”中可看到帧缓冲区驱动。

· 一些设备类支持少量采用字符接口的硬件，。例如SCSI设备一般是块设备，但SCSI磁带是字符设备。

· 一些子系统提供额外的字符接口，以向用户空间提供原始的设备模型。MTD子系统通常用于在多种flash存储器的上面模拟磁盘，但对一些应用程序，如果提供基于flash存储器的原始视图会更好。MTD字符驱动（drivers/mtd/mtdchar.c）完成此任务。我们将在第17章“内存技术设备”中讨论。

· 一些内核层提供钩子，通过导出相应的字符接口，可用于实现用户空间的设备驱动。应用程序通过这些接口可以直接访问设备内部。例如通用的SCSI驱动（drivers/scsi/sg.c）被用来实现用户空间的设备驱动，用于SCSI扫描者和CD驱动器。另一个例子是I2C设备接口，i2c-dev。这些字符接口将会在第19章讨论。

与此同时，在drivers/目录下运行grep –r register\_chrdev可得知内核中字符驱动的大致情况。

表5.3包括本章用到的主要数据结构的概述，以及在源码树中被定义的位置。表5.4列出了本章你用到的主要内核编程接口，以及其定义的地方。

表 5.3. 数据结构概述

数据结构

位置

描述

cdev

include/linux/cdev.h

字符设备在内核中的抽象

file\_operations

include/linux/fs.h

字符设备驱动操作集

dev\_t

include/linux/types.h

主/次设备号Device major/minor numbers

poll\_table

include/linux/poll.h

正处理polling状态以等待数据的驱动拥有的一个等待队列表

pardevice

include/linux/parport.h

并口设备的内核抽象

rtc\_class\_ops

include/linux/rtc.h

顶层和底层RTC驱动的交互接口

miscdevice

include/linux/miscdevice.h

代表一个misc 设备

表 5.4. 内核编程接口概述

内核接口

位置

描述

alloc\_chrdev\_region()

fs/char\_dev.c

申请动态分配主设备号

unregister\_chrdev\_region()

fs/char\_dev.c

alloc\_chrdev\_region()的逆作用

cdev\_init()

fs/char\_dev.c

将字符设备驱动操作集与相关的cdev绑定

cdev\_add()

fs/char\_dev.c

将设备号与cdev绑定

cdev\_del()

fs/char\_dev.c

移除一个cdev

container\_of()

include/linux/kernel.h

根据结构体的成员得到包含它的结构体的地址

copy\_from\_user()

arch/x86/lib/usercopy\_32.c (For i386)

从用户空间向内核空间拷贝数据

copy\_to\_user()

arch/x86/lib/usercopy\_32.c (For i386)

从内核空间向用户空间拷贝数据

likely()unlikely()

include/linux/compiler.h

知会GCC相关条件为真/假的可能性

request\_region()

include/linux/ioport.h kernel/resource.c

获得一片I/O区域

release\_region()

include/linux/ioport.h kernel/resource.c

释放一片I/O区域

in[b|w|l|sn|sl]()out[b|w|l|sn|sl]()

include/asm-your-arch/io.h

与I/O区域进行数据交互的函数家族

poll\_wait()

include/linux/poll.h

在内核的poll\_table中增加一个等待队列

fasync\_helper()

fs/fcntl.c

确保在驱动调用了kill\_fasync()的情况下，一个SIGIO信号会被发送给拥有者应用程序

kill\_fasync()

fs/fcntl.c

Dispatches a SIGIO to the owning application

派送一个SIGIO信号给拥有者应用程序

parport\_register\_device()

drivers/parport/share.c

注册一个parport并口设备

parport\_unregister\_device()

drivers/parport/share.c

注销一个并口设备

parport\_register\_driver()

drivers/parport/share.c

注销一个parport并口设备

parport\_unregister\_driver()

drivers/parport/share.c

注销一个并口驱动

parport\_claim\_or\_block()

drivers/parport/share.c

占有并口

parport\_write\_data()

include/linux/parport.h

向并口写数据

parport\_read\_data()

include/linux/parport.h

从并口读数据

parport\_release()

drivers/parport/share.c

释放一个并口

kobject\_register()

lib/kobject.c

注册一个kobject并在sysfs中创建关联的文件

kobject\_unregister()

lib/kobject.c

kobject\_register()的逆作用

rtc\_device\_register()/rtc\_device\_unregister()

drivers/rtc/class.c

注册/注销RTC子系统的一个底层驱动

misc\_register()

drivers/char/misc.c

注销一个misc驱动

misc\_deregister()

drivers/char/misc.c

注销一个misc驱动

--------------------------------------------------------------------------------

[1] 在第15章“网络接口卡”中的"TUN/TAP驱动"中讨论。

[2] 返回代码完整清单在include/asm-generic/poll.h中定义。其中部分定义仅在网络协议栈中被使用。

[3]如果信号处理程序需要处理从多个设备来的异步事件，需要额外的机制，例如在处理程序里调用select()，以解决产生事件的设备的身份。

[4]看门狗可能发出蜂鸣声而不是重启系统。例如由于电源供应问题而产生的超时，假设此时看门狗电路由电池或大电容供电。

[5] 如果你需要监控几个应用程序的状态，在看门购设备驱动中你需要实现复用器。如果这些进程中的任一个打开驱动无响应，看门狗将会试着自恢复系统。

本文来自CSDN博客，转载请标明出处：http://blog.csdn.net/breeze\_vickie/archive/2010/04/08/5462448.aspx