# 第3章 设备文件和低级文件的输入输出

- 设备文件和文件输入输出函数
- 低级文件的输入输出函数
- 设备文件相关函数
- 低级文件输入输出函数的应用实例
- mknod命令和低级文件输入输出函数

最近，频繁出现把Linux运行在嵌入式系统的例子。为了直接控制硬件，开发人员也开始学习编写Linux设备驱动程序。但是，他们往往没有正确理解Linux的编程方法，就只学习设备驱动程序了。既便是为了控制硬件，单靠理解设备驱动程序也是不能直接投入实际工作的。Linux中把所有硬件表示为设备文件。要想运行设备驱动程序，必须掌握设备文件。另外，低端文件的输入输出又控制着设备文件，因此必须正确理解低端文件的输入输出函数。

本章结合简单的例子，介绍了学习设备驱动程序的基础–基本低端文件函数的使用方法，即利用open( )、write( )、close( )函数控制处理I/O的“/dev/port”设备文件，从而熄灭打印端口pin上连接的LED，并利用ioctl()函数控制处理打印的“/dev/lp0”设备文件，检查打印端口pin上连接的金属Clip的接触情况。

- 本章的内容适合内核的各个版本。

# 3.1　设备文件和文件输入输出函数

要想利用Linux中运行的应用程序控制硬件，必须掌握设备文件及应用程序中使用的文件输入输出函数。本节简单介绍了应用程序中控制硬件时必要的文件输入输出函数。

Linux中控制硬件的情况不多。多数程序只具有理论计算的功能，不去考虑硬件，而是直接控制文件处理过程。客户端的输入输出内容都是由标准输入输出函数处理的。但是播放音乐或利用外设进行控制和测试时，直接控制程序的硬件。因此，必须掌握Linux应用程序的硬件控制方法。

## 3.1.1　应用程序中控制硬件的方法

Linux主要利用C函数库里的函数编写程序。Linux是模仿UNIX形成的操作系统。虽然Linux的名称带有Linux Not UNIX的意义，但是设计思想上继承了Linux的很多特点。UNIX与其他操作系统的最大区别之一是硬件的控制方法。UNIX没有配置专门用于控制硬件的系统调用函数，只是在应用程序上处理为文件。Linux继承了它的处理方式，其中存在普通文件和特殊文件——设备文件。需要控制硬件的应用程序利用文件输入输出函数读取或写入相应硬件的设备文件，运行了连接该设备文件的设备驱动程序，从而启动硬件。此类方法不存在控制硬件的函数，就没有必要特别记住硬件相关函数，只需几个简单的文件处理函数就能控制多个硬件，因此非常方便。

另外，只需体现标准化的文件输入输出函数，统一的方式管理起来更是方便。

Linux系统可以使用文件输入输出函数控制硬件。Linux提供的C函数库里没有硬件相关的函数，它利用设备文件控制硬件。

## 3.1.2　设备文件

Linux中运行的应用程序利用设备文件控制硬件，“/dev”目录中集中管理设备文件。

```
[root@]# ls /dev
console     hdb     mtd4    mtdr10    random    ttyS0
cusa0       hdb1    mtd5    mtdr11    root      ttyS1
cusa1       hdb2    mtd6    mtdr12    rs232sa1  ttyS2
fb          hdb3    mtd7    mtdr13    rs232sa2
fb0         hdb4    mtd8    mtdr14    sound_dsp
⋮
```

设备文件不使用create()函数，由mknod实用程序生成。Mknod也能利用系统呼叫，因此可以建立具有直接生成设备文件功能的应用程序。通常把设备文件集中到“/dev/”目录中，当然这也不是绝对的。然而，网络文件系统或其他支持Linux处理标准inode的文件系统不能建立设备文件。用于建立设备文件的mknod命令格式如下。

```
mknod[设备文件名][设备文件类型] [主设备号] [次设备号]
```

对于字符设备驱动程序，主设备号为240，次设备号为1的设备文件可编写成下列命令句

柄。运行该命令之后，在/dev/目录底下生成命名为devfile的设备文件。

```
[root@] # mknod /dev/devfile c 240 1
```

设备文件的名称要便于查找相应的设备。例如，输入设备“鼠标”的文件名称为/dev/mouse，主输入输出设备的文件名为/dev/console。我们将在Document/device.txt文件上确认Linux设备文件的目录和作用，至于更详细的内容请参考第9章主设备号和次设备号处理的内容。

只有mknod命令能够建立设备文件，但是却可以使用文件删除命令rm删除设备文件。另外，可以通过符号连接（symbolic link）指向其他文件名，这样也可以在应用程序中打开或使用，代表性的例子包括鼠标。

```
[root@] # ls -al /dev/mouse
lrwxrwxrwx    1 root    root    5    Dec  4  2001  /dev/mouse  -> psaux
```

应用程序为了使用鼠标而打开/dev/mouse，但是由于设置了符号连接，实际打开的设备文件是/dev/psaux。

利用设备文件控制硬件，但是设备文件本身并不属于硬件，而是一种信息文件。普通文件的目的在于存储数据，那么设备文件的目的在于向内核提供控制硬件的设备驱动程序的信息。设备文件保存了多种信息，其中重要内容包括设备类型信息、主设备号（major）、次设备号（minor），而其他内容意义不大。通常，利用ls命令就可以查看设备文件的信息，如/dev/console中包含以下内容。

```
[root@] # ls -al /dev/console
crw-------    1 root    root    5, 1  Jan  1 00:00/dev/console
```

头字母c表示字符设备（“b”和“c”分别表示块设备和字符设备），其中/dev/console为字符格式，5和1分别代表主设备号和次设备号。

下面简单整理了设备文件所包含的内容。

• 设备类型的信息　区别设备的类型，包括字符设备和块设备。此外，不存在网络相关的设备文件。

• 主设备号与次设备号　起到连接应用程序和设备驱动程序的作用。

应用程序利用open()函数打开设备文件后，内核从相应的设备文件中得到主设备号，从而查找相应的设备驱动程序。连接了应用程序和设备驱动程序，由次设备号查找实际设备。对于不同的设备驱动程序，次设备号的意义并不是完全一样的，因此不能说都遵守上述的运行规律，但是都表示实质的设备。

• 设备文件和普通文件最大的区别　普通文件存储程序写入的数据，并且相应地增加文件大小。设备文件体现实际硬件，不能保存写入的数据，它起着向硬件传达数据的作用，并且可以读取硬件所发生的数据。

要想呼叫设备驱动程序，就要提供主设备号和次设备号的信息。要想生成提供主次设备号的设备文件，必须使用mknod实用程序。内核可以通过注册的设备驱动程序自动生成设备文件，但是不能支持设备驱动程序，因此在功能上受限制。

提示

**Devfs**

/dev中存在某个设备文件，但是不能说在控制相应设备文件的所有硬件。因为，有可能根本不存在设备文件所对应的硬件。另外，即使具备了硬件，内核如果没有包含相应的设备驱动程序，同样不能控制该硬件。/dev/目录底下包括很多设备文件。设备文件属于小容量信息文件，因此习惯于先建立设备文件。但是设备文件的数量过多，内核自动提供能够实际控制的设备文件，该功能便是devfs文件系统。只要找到现存硬件相关的设备驱动程序，就自动生成该文件系统。当然，内核不是自动支持该功能的，需要在设备驱动程序上配置支持devfs文件系统的功能，此时以下列方式mount后，使用该文件系统。

```
[root@] # mount -n -t devfs none /dev
```

但是，如嵌入式系统，已经指定了设备驱动程序和设备时，无需使用devfs文件系统。

### 3.1.3　文件输入输出函数

如表3-1所示，C函数库中包含了功能相同名称不同的函数。

表3-1　　功能相同名称不同的函数

| 文件输入输出函数 | 功能 |
|---|---|
| fopen(),open() | 打开文件 |
| fread(),read() | 读取文件 |
| fwrite(),write() | 文件上写入数据 |
| fclose(),close() | 关闭文件 |

那么，怎么会存在功能相同名称却不同的函数呢？依据内部表现方式，该函数大体上分为流（stream）文件输入输出函数和低级文件输入输出函数。系统提供的文件函数称为低级文件输入输出函数。低级文件输入输出函数为基础，引入了中间缓存概念，从而实现了流处理。那么，把程序中便于使用的函数称做流文件输入输出函数。

低级文件输入输出函数是由glibc函数库以C函数形态提供的函数。它是处理基本文件的函数，用于实际建立设备文件。

流文件输入输出函数的名称头文字多数情况下以“f”开头，因此容易辨别该函数与低级文件输入输出函数。流文件输入输出函数具有两个特点。

第一，为了有效处理内部的输入输出，配置了用于中间处理的缓存。由FILE结构体管理缓存。即文件上写入数据或读取文件数据时，不是直接传送到文件，而是通过缓存进行预处理，从而提高输入输出效率。另外，由于中间位置配置了缓存，还可以使用重新写入数据的unreading手法。

第二，不是简单地写入或读取文件上的数据，可以利用fprintf( )，fscanf( )等函数，以规则化方式读取或写入数据。

流文件输入输出函数的输入输出处理效率高，形成文件的函数多，而且使用起来比较方便。因此，初学者开始学起设备驱动程序时，愿意在设备文件上使用流文件输入输出函数。但是，读取或写入设备文件的数据时，不能使用流文件输入输出函数。流文件输入输出函数不能把写入的内容直接保存到相应文件上，而是通过缓存先记录相应内容，再等到条件具备

时才把数据记录到文件上。另外，从文件读取数据时，先读取更多的数据。即使在连接硬件的设备文件上写入数据，也有可能不出现数据相关的反应。因此，希望读者不要在设备文件上使用流文件输入输出函数。

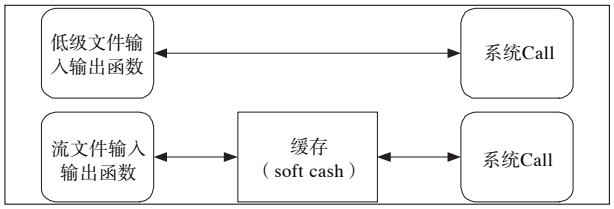


图3–1 低级文件和流文件的输入输出流程图

## 3.2 低级文件的输入输出函数

由低级文件函数可以生成设备文件。初次接触设备驱动程序时，多数人都不理解Linux中运行的低级文件输入输出函数。在这里详细介绍了用于形成设备文件的大部分低级文件输入输出函数。

低级文件输入输出函数是把内核提供的文件相关的系统调用整理成了函数库。它和流文件输入输出函数的区别在于不存在中间缓存。因此，利用写入函数在文件上写入数据后，直接把数据记录到相应的文件上。利用该函数的直接调用功能，我们可以使用低级文件输入输出函数调用设备文件，从而直接启动该设备文件的设备驱动程序。

低级文件输入输出函数是为普通文件而形成的，因此函数的种类非常丰富。但是，只有下列几种函数才能用在启动设备驱动程序的设备文件上。

表3–2 用在设备文件上的函数

| 文件输入输出函数 | 功能 |
|---|---|
| open() | 打开文件或设备 |
| close() | 关闭文件 |
| read() | 从文件读取数据 |
| write() | 文件上写入数据 |
| lseek() | 改变文件的写入或读取位置 |
| ioctl() | 实现read( )、write( )外的特殊控制 |
| fsync() | 实现写入文件上的数据和实际硬件的同步 |

设备驱动程序的编译者一定要正确理解函数中传达的每个变量和函数运行结果的互换值及其意义。当换算值与error相关时更是具有重要意义。实际上设备驱动程序处理文件相关函数的传送值，并且提供换算值。

### 3.2.1 打开和关闭文件函数open( )和close( )

使用低级文件函数时，应预备正数变量用于保存由open()函数返回的文件描述符（descriptor）。可以把整数型变量存储的值看作内核返回的内部文件处理index，并利用该变量值分辨进行处理的文件，它的格式如下。

```
int   fd = -1 ;
```

该变量是文件处理函数的重要要素，关闭文件之前必须保持该变量值。因此，变量用于整体程序时定义为全局变量。当然，若能通过一个函数就能处理所有文件，那么也可以定义为局部变量。利用open( )函数打开文件时变量自动超过0。把初始值代入-1，可以防止文件关闭状态下使用低级文件输入输出函数时发生的错误。若能在同一函数上处理所有文件，也没有必要代入初始值。

要想打开文件，必须了解文件名和打开文件时必要的属性。文件名必须以NULL收尾，通常使用“/dev/”目录中的设备文件。

```
fd=open ("/dev/mem", O_RDWR |O_NONBLOCK);
if(fd<0)
{
//处理error
}
⋮
close(fd);
```

设备文件不能使用低级文件函数的文件生成选项和函数，因此不能使用O_CREATE相关的选项。利用open( )函数顺利打开设备文件时，返回大于0的变量值，若发生错误就返回小于0的变量值。下列几种情况有可能发生错误，我们可以参考返回值或error变量判断发生错误的原因。

- 设备文件不存在；
- 设备文件上没有连接的设备驱动程序；
- 设备驱动程序上发生了错误条件。

发生错误后，不能使用低级文件输入输出函数，因此最好终止程序。

但是由于在没有终止功能的系统上难以实现该操作，此类程序通常采用发生错误时不运行相应设备文件的结构。正常的操作过程中基本不会引起此类error，而是多数发生在设备驱动程序的测试阶段。

与普通的文件不同，设备文件很少执行读/写操作，通常给定O_RDWR选项。

因此，要把重点放在设备执行读写的整个过程中低级输入输出函数要等待的阻塞（blocking）上。阻断时间与硬件的结构有关，有可能发生无限等待的情况。因此，对于需要瞬间处理的情况，即使读写未完成，使用O_NONBLOCK或O_NDELAY选项，也会立即终止低级文件的输入输出函数。使用这两个命令采取非阻塞方式，即没有具备执行条件，函数仍能正常运行，因此需要重视该处理过程。

使用close( )函数关闭由open( )函数打开的设备文件。但是使用close( )函数关闭设备文件时，多数反馈值都得不到确认。因为即使反馈了error也无法解决。

### 3.2.2　文件的读取和写入函数read( )和write( )

利用open( )函数打开设备文件控制硬件时，经常使用read( )和write( )函数。该函数的使用方法与普通函数相同。利用文件的描述符（descriptor）指定使用的文件，给定需要处理的

数据地址和处理数据的具体值。

```
ret_num=read(fd, buff, 10);
ret_num=write(fd, buff, 10);
```

设备文件和普通文件在处理方式上的区别：

- 是否处理为阻塞模式；
- 请求数据和被处理数据之间的差异。

虽然在上述两个内容上有差异，但其他处理方式基本相似。对于阻断模式，没有引起错误时，请求数据的数量和被处理数据的数量基本一致。但是，实际编写程序时，需要考虑不一致的状态。

```
ret_num=read(fd, buff, 10);
if(ret_num<0)
{
`    //处理error
}
if(ret_num !=10)
{
    //与请求不一致时执行
}
```

与请求内容不同时，根据设备驱动程序的结构和程序的进程结构采取不同的处理方式。必要时可以创作成满足读写条件时再次呼叫的结构，但是多数采取默认正常读写状态的处理方式。

利用read( )或write( )函数读取或编写设备文件的数据时，与普通文件的区别在于不保存相应数据。既便利用write( )函数编写了数据，数据只是存储在硬件上。如果设备驱动程序可以利用read( )函数重新记忆和读取写在硬件上的数据，那么数据当然能够保存。但是除了控制记忆元件的ram等文件外，很难找到其他可保存的设备驱动程序文件。因此，没有必要理解文件的尾数据。

### 3.2.3 文件指针处理函数lseek( )

设备文件中文件指针的意义与普通的文件有区别。普通文件指的是文件上写入或读取数据的位置，并且使用read( )或write( )函数进行更新。要想强行改变读取和写入位置，必须使用lseek( )函数。不同的设备驱动程序对设备文件中文件指针的位置下了不同的定义，因此多数设备文件不使用文件指针，只有在存储器或硬盘等少数的记忆相关设备上使用文件指针。若某个设备驱动程序上需要使用文件指针，可以描述为以下格式。

```
off_t ret_pos;
        ⋮
ret_pos=lseek(fd,1234,SEEK_CUR);
```

位置变化值的单位为位（bit），多数以当前位置为基准，使用SEEK_CUR指定文件的位置。

普通文件利用文件指针位置的变化对处理时间的影响关系来指定相对位置。对于多数的设备文件不存在文件结尾的概念，也可能没有相对位置的概念，因此需要使用SEEK_CUR。

### 3.2.4　设备控制函数ioctl( )

连接的硬件决定设备文件的控制方法。read( )和write( )函数难以处理所有控制过程，因此使用普通文件上从未使用的ioctl( )函数。普通文件不能使用该函数，它只是针对设备文件设计的，第10章设备控制中将详细说明该内容。

### 3.2.5　同步处理函数fsync( )

普通文件在写入数据的同时就被存储在文件里。受被连接设备驱动程序体现方式的影响，保存在设备文件上的数据有可能保存到实际硬件上，也有可能保存在设备驱动程序内部缓存上。通过设备文件可控制硬件。需要运行设备驱动程序内部缓存上的所有数据时，使用fsync( )函数。它的使用方法非常简单。

```
int ret;
⋮
ret=fsync  (fd);
```

要知道使用该函数时设备文件不同，运行时间的长短也不一致，甚至有可能无法结束函数，因此通过报警（Alarm）处理指定时间范围。持续时间过长时，需要修改设备驱动程序，但是多数的文件不会持续过长的时间。

此外，与设备文件相关的函数还包括select( )、poll( )、mmap( )、ummap( )函数。在本书的第14章输入输出多路技术和第18章内存映射中详细说明了上述函数。

## 3.3　设备文件相关函数

在shell中利用mknod程序可以编写设备文件，有时也可能在应用程序里编写。本节介绍在应用程序中编译设备文件时必需的mknod()函数。

### 3.3.1　设备文件的生成函数mknod( )

由mknod系统应用程序或内核选项生成设备文件虽然非常少见，但是也会出现应用程序编写设备文件的情况。此时使用mknod( )函数，它的原型如下：

```
int mknod(const char *pathname,mode_t mode,dev_t dev);
```

为了使用该函数，应把下列头文件包含到源代码上。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

函数中使用的pathname是设备文件名字符串。此外，mode是执行权限属性和设备类型，主要参数值如下。

• 设备文件类型
S_IFCHR 文字设备
S_IFBLK 块设备

• 使用权限
S_IRWXU 用户具有读写权限。
S_IRWXG 组具有读写权限。

Dev的每个变量分别指定实际设备的主设备号和次设备号。
利用mknod可以创建主设备号为240、次设备号为1的字符设备文件。

```
[root@] # mknod / dev/ test  c 240 1
```

具有相同功能的程序如下。

```
mknod("dev/test", s_IRWXU | S_IRWXG | S_IFCHR, (240<< 8) | 1 );
```


2.6内核中构成设备文件主设备号和次设备号的位数发生了变化，用于设计应用程序的glibc库却不执行该变化。但是2.6内核内部本身就具有维持低版本互换性的功能，因此不会影响它的使用功能。

## 3.3.2 Error处理函数perror( )

除了已介绍的设备文件函数外，还有用于调试程序的perror( )函数。低级文件输入输出函数运行失败后，相应的error值为整数变量，单看该值无法得知实际意义。此时以容易理解的文字叙述该错误的函数便是perror( )。

```
fd= open (" /dev/ ram", O _RDONLY);
if (fd<0)
{
      perror ("open") ;
}
```

该实例是未能打开设备文件时进行的处理方式，即失败时同时显示出perror( )中指定的文字列“open”和解析失败原因错误序列号的文字列。与设备驱动程序相关的错误序列号的意义可以参考内核源代码目录include/asm/errno.h。该头文件以描述语注释各个error序号的意义。虽然各个注释具有标准化的意义，但是在不同的设备驱动程序上也有可能具有不同的意义。今后处理错误返回值时，最好使用可执行的注释相关错误值。

## 3.4　低级文件输入输出函数的应用实例

在shell中利用mknod程序可以编写设备文件，有时也可能在应用程序里编写。本节介绍了在应用程序中编译设备文件时所必需的mknod( )函数。

在这里通过介绍两个应用实例，加强了对前述内容的理解。第一个应用实例使用/dev/port设备文件，熄灭连接在打印机端口上的LED；而第二个实例使用/dev/lp( )设备文件，确认连接在打印机端口上的金属clip是否接触到了。

在这里介绍两个例子的目的在于为读者提供利用电脑上的设备文件直接控制低级文件输入输出函数的机会。本节中通过由/dev/port设备文件构成的实例熟悉最基础的低级输入输出函数open( )、write( )、lseek( )、close( )的使用方法；而通过由/dev/lp( )设备文件构成的实例，使读者掌握专门控制设备固有特性的ioctl( )函数的使用方法。

### 3.4.1　准备零部件

首先准备好一个LED和两个金属clip。把LED的“＋”侧（长脚）连接到打印机端口的2号pin上，“–”侧（短脚）连接在18号pin上，金属clip分别夹在13号和25号pin上。但是打印机端口间距较窄，夹具又大，不小心会维持接触的状态。平时应断开连接，必要时再手动接触。根据作者的经验，如图3–2所示，clip的一侧伸长，一侧保持原样，就更容易接触了。

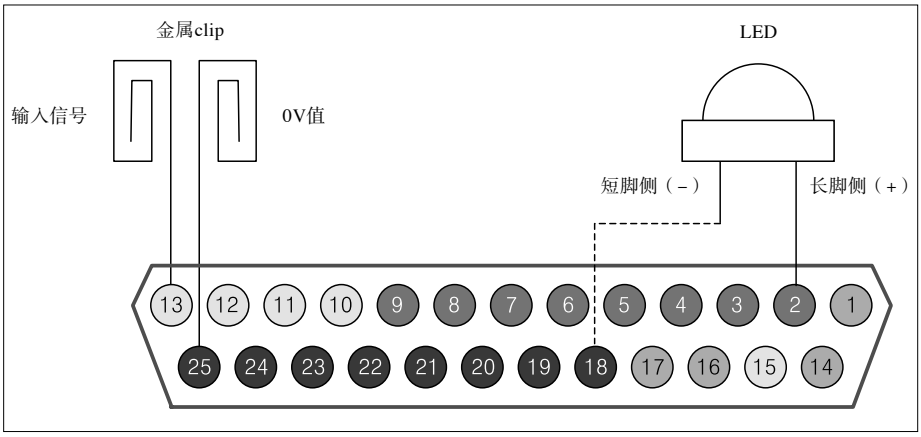


图3–2　连接图

### 3.4.2　基本低级文件输入输出函数的使用例子

本节介绍如何利用/dev/port设备文件熄灭LED，从而掌握低级文件输入输出函数的使用方法。包括以下内容：

- 打开设备文件的方法；
- 移动设备文件的文件指针的方法；
- 设备文件上写入数据的方法；
- /dev/port的控制方法；
- 错误输出的处理方法。

### 1. /dev/port设备文件

Linux内核具有众多的设备文件。但是多数开发人员都不知道还存在目录和输入输出端口相关的设备文件，因此，实际操作中根本不会使用该功能。该设备文件中还包括/dev/port设备文件，它可使应用程序直接接触系统的I/O端口。该文件利用前述的read、write、lseek命令直接接触硬件的I/O区域，执行下述的功能。

- lseek　指定接近的I/O地址；
- read　从指定的I/O中读取数据；
- write　在指定的I/O上写入数据。

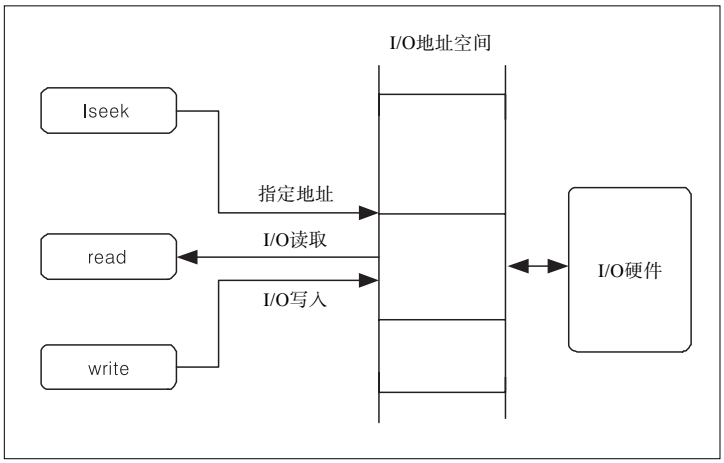


图3-3　/dev/port和低级I/O函数

值得一提的是，所有设备文件的root上均赋予了全部权限。因此，我们要利用root权限运行下面的例子。第1章中已经提到root权限的获取方法有两种。其一是从开始就挂载为root用户，还有一种是作为普通用户使用“su”命令完成转换的方法。

在/dev目录中利用ls命令可以确认系统中是否存在/dev/port设备文件。

```
[root@] # ls -al /dev/port
crw-r-----    1 root    kmem    1, 4 Mar 24 2001 /dev/port
```

此时，它是主设备号为1，次设备号为4的设备文件。如果没有该文件，可以利用mknod命令生成该文件。

```
[root@]  #  mknod   /dev/port  c 1  4
```

### 2. 实例源代码

程序不算太复杂。要想控制连接在打印机端口的LED，必须知道打印机端口的输入输出地址，这就是向打印机端口输出数据的地址。本实例为了向地址写入特定的数据，先打开/dev/port设备文件，再利用lseek( )函数指定0x378。然后利用write命令把需要的数据写入LED进行控制，最后利用close( )函数关闭设备文件。

### 实例

[实例3-1]　sample/2.4/test.c

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05
06 int main (int argc, char **argv)
07 {
08     int fd;
09     int lp;
10
11     unsigned char buff [128];
12
13     fd=open ("/dev/port", O_RDWR);
14     if (fd < 0)
15     {
16          perror ("/dev/ port open error");
17          exit (1);
18     }
19
20     for (lp = 0; lp < 10; lp++)
21     {
22          lseek (fd, 0x378, SEEK_SET);
23          buff[0] = 0xFF;
24          write (fd, buff, 1);
25          sleep (1);
26          lseek (fd, 0x378, SEEK_SET);
27          buff[0] = 0x00;
28          write (fd, buff, 1);
29          sleep (1);
30     }
31     close (fd);
32
33     return 0;
34 }
```

### 源代码解析

（1）[main.c] 13~18行

为了熄灭LED，打开控制输入输出端口的/dev/port设备文件。open( )函数的第二个变量使用了O_RDWR，从而使设备文件的读写均具有可能性。/dev/port设备文件可读写输入输出，不指定O_NONBLOCK或O_NDELAY，也会自动处理为不可阻断模式。这也是/dev/port设备文件的特征。运行open( )函数后不能正常打开文件时，进行错误处理。open( )函数规定失败时返回-1，但是实际程序中还要比较是否小于0。因为有时会不遵守该规定。不用说，还是这种类型的代码比较安全。open()函数失败后，以文字方式确认error类型的方法更有利于函数的调试。

因此，利用perror( )函数输出错误值的意义。通常，perror( )函数的要素传达的文字中包含了运行的位置或内容，可以直接追击发生错误的地点。

（2）[main.c] 22行

若正常打开了设备文件，通过lseek( )函数把地址指定为文件端口值。这也是/dev/port设备文件的设备驱动程序的特征。当然，并不是说所有设备文件的设备驱动程序都具有这样的特点。实例中的lseek语句运行后，文件端口的位置被指定为0x378。第二个要素是SEEK_SET，0x378直接转向文件端口值。

（3）[main.c] 24 ~ 29行

指定要接近的地址后，利用write( )函数向打印机端口输出数据。write( )函数总是传达存储数据的地址，应以相同的字符形式传送数据。

（4）为了驱动LED，连接到LED的pin针上，即在相应的数据位上写入“1”。实例源代码中指定了所有的0xFF值。在原则上属于位0的pin针位置，因此正确值是0x01。为了熄灭LED，把0x00放到缓存位置后，采用相同的方式运行文件。

（5）[main.c]31行

重复10号熄灭LED后，为了终止程序，使用close( )函数。通常不确认close( )函数的值，但是最好确认返回的参数值。

实例源代码中为了便于说明并且简化源代码，除了open( )函数，都没有确认错误。建立实际应用程序时，确认lseek( )和write( )函数的返回值。

#### 3. 运行方法

本实例是单位源代码，没有必要建立Makefile，只编译下列内容。

```
[root@] # gcc -o test main.c
```

利用编译的结果设计test程序后，运行结果如下。

```
[root@] #./test
```

此时能够看到连接在打印机端口的LED隔2s被熄灭。

### 3.4.3 Ioctl( )函数的使用例子

通过使用/dev/lp0 设备文件表示金属clip接触状态的例子，熟练掌握ioctl( )函数的使用方法。

#### 1. /dev/lp0 设备文件和ioctl( )函数

/dev/lp0 设备文件可以控制打印机端口。该设备文件没有直接连接到打印机的端口上。连接/dev/parport设备文件的设备驱动程序和内核后，可控制使用打印端口的打印机。

利用/dev/lp0 设备文件，可以确认打印机的状态。连接金属clip的13号pin针是表示打印机连接状态的select信号。利用select信号可以判断clip是否接触了。

/dev/lp0 设备文件中使用ioctl( )函数可以判断出打印机的状态。

ioctl( )是设备文件的专用函数，多数用在利用设备文件的控制上。设备文件不同，ioctl( )

函数的使用方法也不同。因此，要想利用ioctl( )函数必须了解相应设备文件中的使用方法。利用下列命令可以得知类似的方法。

```
[root@] # man ioctl _ list
```

为了掌握/dev/lp0 设备文件的使用方法，必须查看内核源代码中的/include/linux/lp.h文件。

为了得到打印机端口状态，在ioctl( )函数的各个变量上给定LPGETSTATUS，为第三个变量argp给定可得到状态的int型变数地址。那么，返回值便是打印状态值。

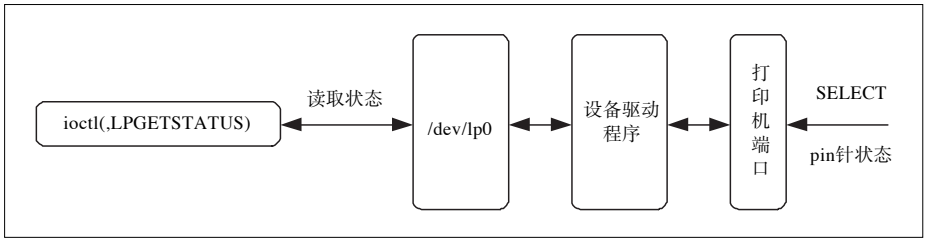


图3-4　利用lp0设备文件读取SELECT针的状态

### 2. 实例源代码

实例3-2中确认连接在13号针上金属clip和连接在地脚上的25号针直接的接触状态。首先打开/dev/lp0设备文件，利用ioctl( )函数探出打印机端口的状态。此时，若有select信号则输出“ON”，没有select信号则输出“OFF”字符。金属clip没被接触时输出“ON”信号，被接触时在13号针上出现GND信号，并输出“OFF”字符。

#### 实例

[实例3-2]　/dev/lp 实例 (common/lp/main.c)

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05
06 #include <linux/lp.h>
07
08 int main (int argc, char **argv)
09 {
10
11     int fd;
12     int prnstate;
13     int lp;
14
15     unsigned char buff [128];
16
17     fd=open ("/dev/lp0", O_RDWR | O_NDELAY);
18     if (fd < 0)
```

```
19      {
20           perror ("open error");
21           exit (1);
22      }
23
24      while (1)
25      {
26          ioctl (fd, LPGETSTATUS, &prnstate);
27
28          // 13 Pin <--> GND Pin
29          if (prnstate & LP_PSELECD) printf ("ON\n");
30          else                       printf ("OFF\n");
31          usleep (50000);
32      }
33
34      close (fd);
35
36      return 0;
37
38  }
```

**源代码解析**

（1）[main.c]17~22行

运行open( )函数，打开/dev/lp0设备文件。与前一实例的区别在于O_NDELAY选项，即以不可阻断的方式打开设备文件。/dev/lp0是连接打印机后实现控制的设备文件，它的读写过程必须具备相应的条件。但是要想进行测试时，连接LED和金属clip，而不连接打印机，也能形成阻断条件。因此，以不可阻断方式打开设备文件。

（2）[main.c]26~30行

为了读取打印机端口的状态，使用ioctl( )函数。利用该函数在prnstate变量上可以确认打印机端口当前的输入状态。为该值运行LP_PSELECD和位AND计算，真值输出“ON”，否则输出“OFF”值。

（3）[main.c]31行

利用usleep( )函数延时50ms。这里没有特别的理由，主要原因在于缓解过多的信息同时出现在画面上的压力。

按下CTRL+C键，可以终止该程序的运行。因此，不能执行close( )函数。一旦终止了进程，依据Linux 进程的管理规则，所有的文件都会被关闭，因此/dev/lp0设备文件也会自动关闭。

**3. 运行方法**

本实例也不用制作Makefile，而是直接编译下列描述内容就行。

```
[root@] #gcc -o  test  main.c
```

利用编译结果创建了test程序后，运行下列描述语。

```
[root@] #. /test
```

此时运行了程序，持续输出“ON”字符。接触连接打印机端口的两个金属clip后，也输出“OFF”字符。然后按下CTRL+C键，终止程序。

## 3.5　mknod命令和低级文件输入输出函数

除了mknod命令，低级文件输入输出函数的描述与man或info命令有所差异。通常，在目录上描述函数时，都是以普通文件为基础进行描述的。但是，在这里我们只限说明处理设备文件的情况。因此，省略了对设备文件不起作用的普通文件的诸多状态值和返回值。

**mknod命令**

功能：　用于建立特殊文件。

原型：　#mknod[options]设备文件名{bcu}主设备号　次设备号
选项　[–m mode] [––mode=mode] [––help] [––version]

选项：
- –m,––mode mode　指定生成文件模式的选项。作为mode的包括chmod中使用的记号或数字形式；
- ––help　显示帮助内容；
- ––version　显示版本信息。

说明：　mknod命令用于建立FIFO、字符设备文件及块设备文件等。建立文件模式初值为0666。写入设备文件名后，指定该文件的特殊状态值。在这里能够使用下列设置。

- p　FIFO
- b　块设备文件
- c或u　文件设备文件

建立块或字符特殊文件时，必须指定该设备文件的主设备号和次设备号。

**Open( )函数**

功能：　打开设备文件

原型：　#include<sys/types.h>
- #include<sys/stat.h>
- #include<fcntl.h>

int open (const char* pathname, int flags);

说明：　利用flags指定的属性打开表示pathname上指定字符的设备文件。通常，pathname上指定的位置为“/dev/”目录中的设备文件。

变量：
- pathname 指定设备文件字符的地址；
- flags 指定接近设备文件的属性；
  - –O_RDONLY 以只读方式打开文件；
  - –O_WRONLY 以只写方式打开文件；
  - –O_RDWR 以可读写方式打开文件；
  - –O_NOCTY 如果欲打开的文件为终端机设备时，则不会将该终端机当成进程控制终端机；
  - –O_NONBLOCK 以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即返回进程之中；
  - –O_NDELAY 以不可阻断的方式打开文件；
  - –O_SYNC 以同步的方式打开文件。设备上写入的内容记录到硬件之前，呼叫进程处于阻断状态。

返回值：若成功打开文件，则返回文件描述符，失败则返回–1值。若所有欲核查的权限都通过了检查则返回0 值，表示成功，只要有一个权限被禁止则返回–1。得到–1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码
  - –ENXIO 文件为设备文件，但是没有相应的设备；
  - –ENODEV 不存在设备文件相关的设备驱动程序或硬件；
  - –ENOMEM 核心内存不足。

### Close( )函数

功能： 关闭设备文件。

原型：
```
# include <unistd.h>
int close (int fd);
```

说明： 为了打开设备文件，关闭open( )函数返回的文件描述符fd相应的设备文件。

变量：
- fd open( )函数运行结果返回的文件描述符。

返回值：成功关闭则返回0值，失败则返回–1值。

### Read( ) 函数

功能： 由设备文件读取数据。

原型：
```
# include <unistd.h>
ssize_t read(int fd,void * buf ,size_t count);
```

说明：　read( )会把参数fd 所指的设备文件传送count个字节到buf指针所指的内存中。此时count值应小于SSIZE_MAX。open ( )函数没有指定为O_NONBLOCK 或O_NDELAY时，阻断到可读取相应计数值的大小。设备文件的设备驱动程序没有体现O_NONBLOCK或O_NDELAY时，没有指定相应的选项也有可能被阻断。原则上这是错误的设备驱动程序。创建程序时，也要对比上述情况，因此必须确认返回的结果。此外文件读写位置会随读取到的字节移动。

变量：
- fd　由open ( ) 函数运行结果返回的描述符；
- buf　存储读取数据的空间位置。该存储空间应大于计数字节；
- count　设备文件中读取的数据大小。该值应小于SSIZE_MAX。返回值为0,立即终止运行。

返回值：设备文件正常读取了数据后，返回读取的字节数。即使该值小于相应的字节数，也不是错误。几乎没有实际可用的字节数或被中断信号时发生上述现象。如果失败则返回–1。得到–1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码
  - –EINTR　此调用被信号所中断；
  - –EAGAIN　当使用不可阻断（O_NONBLOCK）打开文件后，read呼叫无可读取的数据；
  - –EIO　设备文件读取数据时发生输入输出错误；
  - –EBADF　参数fd 非有效的文件描述词，或该文件已关闭；
  - –EINVAL　fd连接到不适合读取的对象上；
  - –EFAULT　参数buf为无效指针，指向无法存在的内存空间。

### Write ( )函数

功能：　将数据写入设备文件内。

原型：　#include<unistd.h>表头文件
ssize_t write (int fd,const void * buf,size_t count); 定义函数

说明：　write()会把参数buf所指的内存中的count个字节写入到参数fd所指的文件内。此时count值应小于SSIZE_MAX。open( )函数没有指定为O_NONBLOCK 或O_NDELAY时，阻断到可读取相应count值的大小。设备文件的设备驱动程序没有体现O_NONBLOCK 或O_NDELAY时，没有指定相应的选项也有可能被阻断。原则上这是错误的设备驱动程序。创建程序时，也要对比上述情况，因此必须确认返回的结果。当然，文件指针的位置也会随之移动相应的字节数。

变量：
- fd　由open( )函数运行结果返回的描述符；
- buf　存储写入数据的空间位置，该地址所指的存储空间应大于count字节；
- count　设备文件中要写入数据的大小。该值应小于SSIZE_MAX。返回值为0则立即中断。

返回值：设备文件内正常写入数据后，返回写入的字节数。即使该值小于相应的必要字节数，也不是错误。可能没有写入实际需要的字节数，或者被某种信号中断了。如果失败则返回-1。得到-1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码

-EBADF 参数fd 非有效的文件描述词，或该文件没有处于可写状态；
-EINVAL fd连接到不适合写入的对象上；
-EFAULT 参数buf为无效指针，指向无法存在的内存空间；
-EAGAIN 虽然使用不可阻断（O_NONBLOCK）打开了文件，但是没有处于read呼叫后可直接处理的状态；
-EINTR 写完数据前，此调用被信号所中断；
-ENOSPC 包含fd文件的设备上不存在相应的数据空间；
-EIO 设备文件写入数据的过程中发生了输入输出错误。

**Lseek（）函数**

功能： 移动文件的读写位置。

原型：
```
#include<sys/types.h>  表头文件
#include<unistd.h>

off_t lseek(int fld,off_t offset ,int whence); 定义函数
```

说明： lseek( )函数用来控制该文件的读写位置。把文件描述符fd所指的设备文件向上移到文件指针的位置。把文件指针的位置移到whence所指选项offset值的位置上。文件指针的位置随设备文件所连接设备驱动程序的处理方式而变化。例如，管理内存的设备文件可以利用内存的位置。但是，多数字符设备驱动程序不使用该功能。

变量：
- fd 运行open（）函数后返回的描述符；
- offset 以字节为单位，指定被移动文件指针的位置。该值为负数。该值随whence解释为实际移动位置；
- whence 指定用来解释offsett的条件；

-SEEK_SET 参数offset即为新的读写位置；
-SEEK_CUR 以目前的读写位置往后增加offset个位移量；
-SEEK_END 将读写位置指向文件尾后再增加offset个位移量。多数设备文件的文件尾定义较为模糊，通常不使用该值。

返回值：当调用成功时则返回目前的读写位置，也就是距离文件开头多少个字节。若有错误则返回(off_t) -1，errno 会存放错误代码。得到了文件指针的正常移动位置后，返回移到的实际位置。如果失败则返回-1。得到-1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码

-EINVAL whence指定的值不适合。

### Ioctl ( ) 函数

功能：　控制设备文件。

原型：#include<sys/ioctl.h>

int ioctl (int fd, int request, ...)

说明：　ioctl ( ) 函数在文件描述符fd相应的设备文件上实现read( )和write ( )函数难以完成的输入输出处理。 该函数的各个变量中除了fd外，其他变量没有标准值。只是指定了几个macro值定义的标准。不同的设备文件具有表示不同意义的值。ioctl ( ) 函数的各个变量是可变的，虽然在语法上能够表现出来，但是最多可容纳3个。第三个因数表示char*argp。

变量：
- fd　运行open ( ) 函数后返回的文件描述符；
- request　定义连接设备文件的设备驱动程序应调用的命令。根据宏判断是输出命令还是输入命令，argp指定的值作为存储地址时，以字节为单位显示出传达因子；
- ...　第三个变量被称为argp，是可省略的变量，与request有关。是处理request命令的辅助信息。

返回值：当调用成功时返回0，失败时返回-1。得到-1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码
  - -EFAULT　参数argp无效，指向无法存在的内存空间；
  - -ENOTTY　fd与字符设备文件无关；
  - -EINVAL　连接设备文件的设备驱动程序不能处理request或者argp。

### Fsync ( ) 函数

功能：　将缓冲区数据写回磁盘。

原型：#include<unistd.h>
int fsync(int fd);

说明：　fsync( )负责将参数fd所指的文件数据由系统缓冲区写回磁盘，以确保数据同步。

变量：
- fd　运行open ( ) 函数后返回的文件描述符。

返回值：运行成功则返回0，失败则返回-1。得到-1值时参考全局变量错误值，可以确认实际设备驱动程序中返回的值。

- 错误代码
  - -EROFS，EINVAL　设备文件不支持该函数的处理过程；
  - -EIO　同步过程中发生了错误。