Linux设备驱动开发入门

本文以快捷而简单的方式讲解如何像一个内核开发者那样开发linux设备驱动

源作者： Xavier Calbet

翻译： 顾宏军（http://www.ossp.cn）

知识准备

要开发Linux设备驱动，需要掌握以下知识：

- C编程 需要掌握深入一些的C语言知识，比如，指针的使用，位处理函数，等。
- 微处理器编程 需要理解微机的内部工作原理：存贮器地址，中断，等。这些内容对一个汇编程序员应该比较熟悉。

Linux下有好几种不同的设备。为简单起见，本文只涉及以模块形式加载的字符设备。使用2.6.x的内核。（特别是Debian Sarge使用的2.6.8内核。）

用户空间和内核空间

当你开发设备驱动时，需要理解“用户空间”和内核空间之间的区别。

- 内核空间：Linux操作系统，特别是它的内核，用一种简单而有效的方法管理机器的硬件，给用户提供一个简捷而统一的编程接口。同样的，内核，特别是它的设备驱动程序，是连接最终用户/程序员和硬件的一坐桥或者说是接口。任何子程序或者函数只要是内核的一部分（例如：模块，和设备驱动），那它也就是内核空间的一部分。

- 用户空间. 最终用户的应用程序，像UNIX的shell或者其它的GUI的程序(例如，gedit),都是用户空间的一部分。很显然，这些应用程序需要和系统的硬件进行交互。但是，他们不是直接进行，而是通过内核支持的函数进行。

它们的关系可以通过下图表示:

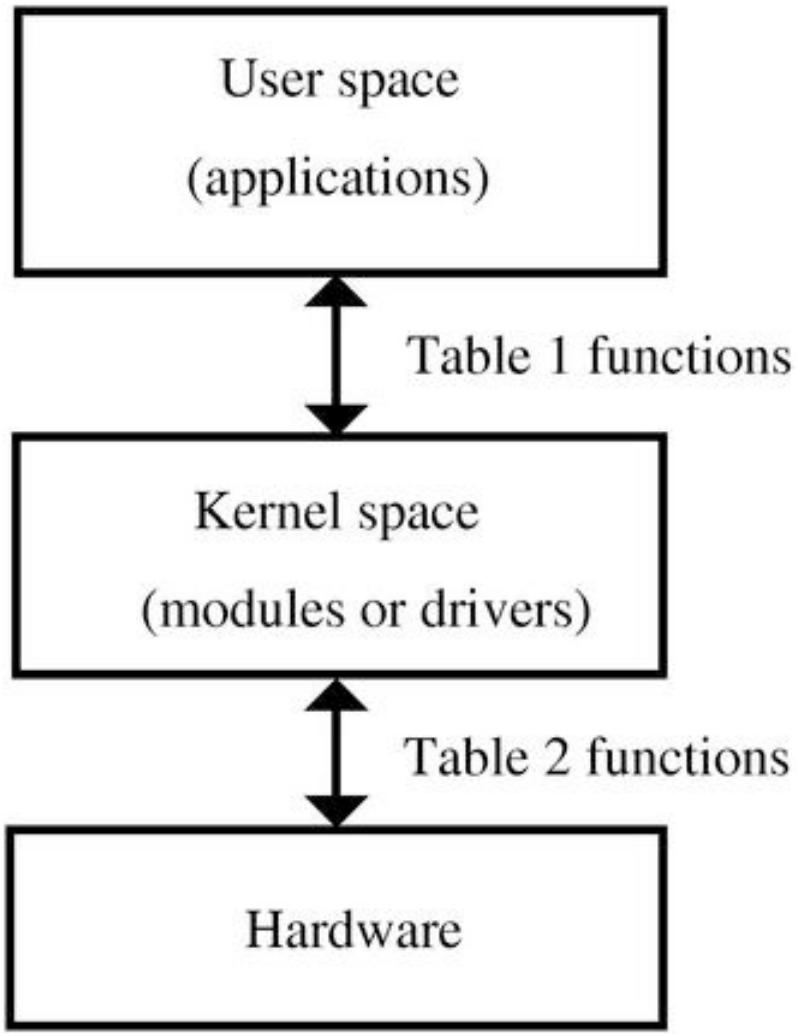

图1: 应用程序驻留在用户空间, 模块和设备驱动驻留在内核空间

用户空间和内核空间之间的接口函数

内核在用户空间提供了很多子程序或者函数，它们允许用户应用程序员和硬件进行交互。通常，在UNIX或者Linux系统中，这种交互是通过函数或者子程序进行的以便文件的读和写操作。这是因为从用户的视角看，UNIX的设备就是一个个文件。

从另一方面看，在Linux内核空间同样提供了很多函数或者子程序以在底层直接地对硬件进行操作，并且允许从内核向用户空间传递信息。

通常，用户空间的每个函数（用于使用设备或者文件的），在内核空间中都有一个对应的功能相似并且可将内核的信息向用户传递的函数。这种关系可从下表看出来。目前这个表是空的，在我们后面每个表项都会填入对应的函数。

表 1. 设备驱动事件和它们在内核和用户空间的对应的接口函数

事件	用户函数	内核函数
加载模块		
打开设备		
读设备		
写设备		
关闭设备		
卸载模块		

内核空间和硬件设备之间的接口函数

在内核空间同样有可以控制设备或者在内核和硬件之间交换信息的函数。表2解释了这些概念。同样的，这个表将在介绍到相应内容时填写上。

表 2. 设备驱动事件和它们在内核空间与硬件设备之间对应的接口函数

事件	内核函数

读数据	
写数据	

第一个驱动：在用户空间加载和卸载驱动

这一节将向你展示如何开发你的第一个Linux设备驱动，该驱动作为一个内核模块存在。

首先，写一个文件名为nothing.c的文件，代码如下：

<nothing.c> =

```
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");
```

内核从2.6.x开始，编译模块变得稍微复杂些。首先，你需要有一份完整的，编译了的内核源代码树。如果你使用的是Debian Sarge系统，你可以按照附录B（在本文末尾）的步骤进行操作。在以下的内容里，假设你使用的是2.6.8内核。

接下来，你需要撰写一个makefile。本例子所用的makefile文件名称为Makefile，内容如下：

<Makefile1> =

```
obj-m := nothing.o
```

和之前版本的内核不同，你需要使用和你当前系统所用内核版本相同的代码来编译将要加载和使用的模块。编译该模块，可以使用以下命令：

$ make -C /usr/src/kernel-source-2.6.8 M=`pwd` modules

这个非常简单的模块在加载之后，将属于内核空间，是内核空间

的一部分。

在用户空间，你可以以root账号加载该模块，命令如下：

```
# insmod nothing.ko
```

insmod命令用于将模块安装到内核里。但是这个特殊的模块不常用。

要查看模块是否已经安装完成，可以通过查看所有已安装模块来进行：

```
# lsmod
```

最后，模块可以通过以下命令从内核中移除：

```
# rmmod nothing
```

同样的，使用lsmod命令，可以用于验证该模块已不在内核中。

主要内容整理在如下表格里。

表3. 设备驱动事件和它们在用户空间，内核空间对应的接口函数。

Events	User functions	Kernel functions
Load module	insmod	
Open device		
Read device		
Write device		
Close device		
Remove module	rmmod	

“hello world”驱动：在内核空间加载和移除驱动

当一个模块设备驱动被加载到内核时，一些通常要做的事情包括：设备复位，初始化RAM，初始化中断，初始化输入/输出端

口，等。

这些动作在内核空间进行，通过下面将介绍的两个函数进行：module_init 和module_exit；它们和用户空间的用于安装和卸载模块的命令insmod 和 rmmod对应。也可以说，用户空间的命令 insmod 和rmmod使用内核空间的函数 module_init和module_exit进行。

我们通过一个最基本的hello world 程序，看实际的例子：

<hello.c> =

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void) {
  printk("<1> Hello world!\n");
  return 0;
}
static void hello_exit(void) {
  printk("<1> Bye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

实际的函数hello_init和hello_exit可以用任何其他名称。但是为了

使系统能够正确的识别它们是加载和卸载函数，需要把它们作为module_init和module_exit的参数。

以上代码里还包括了printk函数。它和我们非常熟悉的printf函数很相似，只是它只在内核内有效。符号<1> 表示该消息的优先级（数字）。这样就可以通过内核的日志文件里看到该消息，该消息也会在系统控制台中显示。

这个模块可以使用和之前那个相同的命令进行编译，当然前提是把它的名字加在Makefile文件里。

<Makefile2> =

```
obj-m := nothing.o hello.o
```

本文中，把写makefile的事情留给读者自行练习。在附录A里，有一个完整的可以编译所有模块的Makefile。

当模块被加载或是卸除时，在printk声明里的消息将打印在系统控制台上。如果这个消息没有在控制台上显示，可以通过dmesg命令，或者查看系统的日志文件cat /var/log/syslog命令看到。

表4 填入了两个新函数。

设备驱动事件和在内核空间和用户空间之间实现该功能的函数

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device		
Read device		
Write device		
Close device		
Remove module	rmmod	module_exit()

一个完整的驱动“memory”:驱动的初始化

现在我开始构建一个完整的设备驱动：memory.c。可以从这个设备读取和写入一个字符。虽然这个设备没时么用途，但提供了个很好的样例，它是一个完整的驱动；很容易实现，因为它不操作实际的硬件设备（它是电脑内部模拟的硬件）。

要开发驱动，一些在设备驱动中很常见的#include声明，需要首先要加进来：

<memory initial> =

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of memory.c functions */
int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file
*filp);
```

```
ssize_t memory_read(struct file *filp, char *buf, size_t
count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, char *buf,
size_t count, loff_t *f_pos);
void memory_exit(void);
int memory_init(void);

/* Structure that declares the usual file */
/* access functions */
struct file_operations memory_fops = {
  read: memory_read,
  write: memory_write,
  open: memory_open,
  release: memory_release
};

/* Declaration of the init and exit functions */
module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */
/* Major number */
int memory_major = 60;
/* Buffer to store data */
char *memory_buffer;
```

在#include之后，就是即将定义的函数的声明。通用的用于处理文件的函数在file_operations里声明。这些在过后会讲解。接下来是初始化和卸载函数——在模块加载和卸载时执行——对内核声明。最后，是该驱动的全局变量声明：一个是“主设备号”，另外一个是内存指针，memory_buffer,将用于存储该驱动的数据。

“memory”驱动：连接到设备

在UNIX和Linux中，设备可以用和文件一样的方式从用户空间访

问。这些设备文件通常在/dev目录下。

要把一般文件和内核模块链接在一起需要两个数据：主设备号和从设备号。主设备号用于内核把文件和它的驱动链接在一起。从设备号用于设备内部使用，为简单起见，本文并不对它进行解释。

需要创建一个文件（该设备文件用于和设备驱动操作），

```
# mknod /dev/memory c 60 0
```

其中，c说明创建的是字符设备，60是主设备号，0是从设备号。

在这个驱动里，register_chrdev函数用于在内核空间，把驱动和/dev下设备文件链接在一起。它又三个参数：主设备号，模块名称和一个file_operations结构的指针。在安装模块时将调用该函数：

<memory init module> =

```
int memory_init(void) {
  int result;

  /* Registering device */
  result = register_chrdev(memory_major, "memory",
&memory_fops);
  if (result < 0) {
    printk(
      "<1>memory: cannot obtain major number %d\n",
memory_major);
    return result;
  }

  /* Allocating memory for the buffer */
  memory_buffer = kmalloc(1, GFP_KERNEL);
  if (!memory_buffer) {
    result = -ENOMEM;
    goto fail;
  }
  memset(memory_buffer, 0, 1);

  printk("<1>Inserting memory module\n");
  return 0;

  fail:
    memory_exit();
```

```
    return result;
}
```

以上代码使用了kmalloc函数。这个函数工作在内核空间，用于为该驱动程序的缓冲区分配内存。它和我们熟悉的malloc函数很相似。最后，如果注册主设备号或者分配内存失败，模块将退出。

“memory”驱动：卸载驱动

为通过memory_exit函数卸载模块，需要定义unregsiter_chrdev函数。这将释放驱动之前向内核申请的主设备号。

<memory exit module> =

```
void memory_exit(void) {
  /* Freeing the major number */
  unregister_chrdev(memory_major, "memory");

  /* Freeing buffer memory */
  if (memory_buffer) {
    kfree(memory_buffer);
  }

  printk("<1>Removing memory module\n");

}
```

为了完全的卸载该驱动，缓冲区也需要通过该函数进行释放。

“momory”驱动：像文件一样打

开设备

内核空间打开文件的函数是open，和用户空间打开文件的函数fopen对应：在file_operations结构里，用于调用register_chrdev。在本例里，是memory_open函数。它又几个参数：一个inode结构，该结构向内核发送主设备号和从设备号的信息；另外是一个file结构，用于说明，该设备文件允许哪些操作。所有这些函数在本文中都未做深入的讲解。

当设备文件被打开后，通常就需要初始化驱动的各个变量，对设备进行复位。但在本例中，这些操作都没进行。

memory_open函数定义如下：

<memory open> =

```
int memory_open(struct inode *inode, struct file *filp)
{

  /* Success */
  return 0;
}
```

表5

设备驱动事件和在内核空间和用户空间之间实现该功能的函数

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device		
Write device		
Close device		
Remove module	rmmod	module_exit()

“monory”驱动：像文件一样关闭设备

在内核空间里，和用户空间里关闭文件的fclose对应的函数是release：它也是file_operations结构体的成员，用于调用register_chrdev。本例中，它是函数memory_release,和上面的相似，它也有inode和file两个参数。

当设备文件关闭后，通常需要释放该设备使用的内存，释放各种操作该设备相关的变量。但是，为简单起见，例子里没有进行这些操作。

memory_release函数定义如下：

<*memory release*> =

```
int memory_release(struct inode *inode, struct file
*filp) {

  /* Success */
  return 0;
}
```

表6.

设备驱动事件和在内核空间和用户空间之间实现该功能的函数

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device		
Write device		
Close device	fclose	file_operations: release
Remove module	rmmod	module_exit()

"memory"驱动：读取设备

和用户空间函数fread类似，内核空间里，读取设备文件使用read函数：read是file_operations的成员，用于调用register_chrdev。本例中，是memory_read函数。它的参数有：一个file结构；一个缓冲区(buf)，用户空间的fread函数将从该缓冲区读数据；一个记录要传输的字节数量的计数器（count），它和用户空间的fread使用的计数器值相同；最后一个参数（f_pos）指示从哪里开始读取该设备文件。

本例中，memory_read函数通过copy_to_user函数从驱动的缓冲区（memory_buffer）向用户空间传送一个简单的字节：

<memory read> =

```
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos) {

  /* Transfering data to user space */
  copy_to_user(buf,memory_buffer,1);

  /* Changing reading position as best suits */
  if (*f_pos == 0) {
    *f_pos+=1;
    return 1;
  } else {
    return 0;
  }
}
```

设备文件的读取位置(f_pos)也改变了。如果起始点是文件的开头，那么f_pos的值将增加1，如果要读取的字节读取正常，则返回值为1。如果读取位置不是文件开头，则是文件的末尾，返回

值将是0，（因为文件只存储了1个字节）。

表7. 设备驱动事件和在内核空间和用户空间之间实现该功能的函数

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device	fread	file_operations: read
Write device		
Close device	fclose	file_operations: release
		module_exit()
Remove modules	rmmod	

“memory”驱动：向设备写数据

和用户空间里写文件的fwrite对应，内核空间里是write:write是file_operations的成员，用于调用register_chrdev。本例中是memory_write函数，它有如下几个参数：一个file结构；buf，一个缓冲区，用户空间函数fwrite将向该该缓冲区写数据；count，统计将传送的字节数的计数器，和用户空间函数fwrite的计数器有相同的数值；最后是f_pos，指示从哪里开始写文件。

<memory write> =

```
ssize_t memory_write( struct file *filp, char *buf,
                      size_t count, loff_t *f_pos) {

  char *tmp;

  tmp=buf+count-1;
  copy_from_user(memory_buffer,tmp,1);
  return 1;
}
```

本例中，函数copy_from_user从用户空间传送数据到内核空间。

表8

设备驱动事件和在内核空间和用户空间之间实现该功能的函数

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Close device	fread	file_operations: read
Write device	fwrite	file_operations: write
Close device	fclose	file_operations: release
Remove module	rmmod	module_exit()

完整的“memory”驱动

把以上各部分代码整合起来，一个完整的驱动就完成了：

<memory.c> =

```
<memory initial>
<memory init module>
<memory exit module>
<memory open>
<memory release>
<memory read>
<memory write>
```

在该模块使用之前，你需要和刚才那个模块一样，进行模块编译。编译好后，用以下命令进行加载：

```
# insmod memory.ko
```

并且最后是取出设备文件的保护：

```
# chmod 666 /dev/memory
```

如果以上步骤一切正常，此时你就可以向设备/dev/memory写一串字符，并且它将把你写入的最后一个字符存储起来。你可以按下例操作：

```
$ echo -n abcdef >/dev/memory
```

使用cat检查设备的内容：

```
$ cat /dev/memory
```

存储的字符将不会改变，直到该字符被覆盖，或者是该模块被卸载。

真实的“并口”驱动：描述并口

接下来，将修改刚刚写的memory驱动，来在一个真实设备上进行真实的操作。使用简单并且常见的计算机并口作为例子，新驱动的名称叫做：parlelport。

并口实际上是一个允许输入输出数字信息的设备。它有一个母的D-25连接头，有25针。从内部看，从CPU视图看，并口有3字节的存储,在PC上，基地址（设备的起始地址）通常是0x378。在本例中，我们仅使用包含完整数字输出的第一个字节。

上面提到的字节和外部接头针脚之间的连接情况不下图：

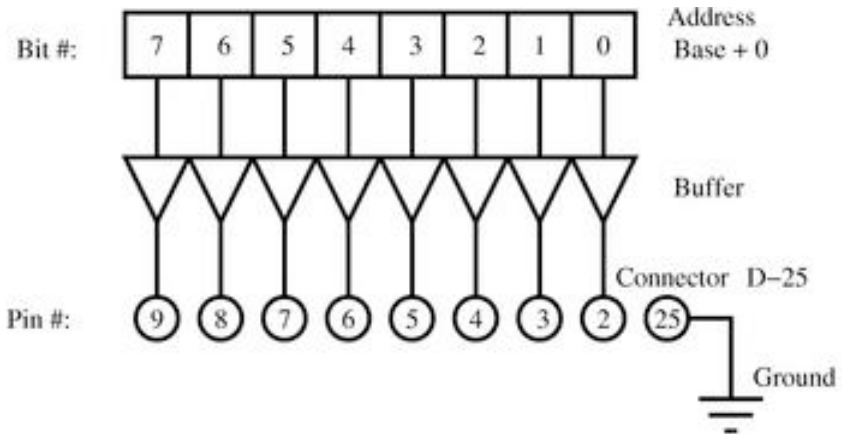

图2: 并口的第一个字节和它在D-25连接头上对应的针脚

“并口”驱动：模块初始化

刚才的memory_init函数需要进行修改－－指定RAM地址为保留的并口的内存地址（0x378）。check_region函数用于检查一个内存区域是否可用，并且，用request_region函数保留指定的内存区域给当前设备。这两个函数都有两个参数，内存区域的基地址以及长度。另外request_region函数还需要一个指定模块名称的字符串。

<parlelport modified init module> =

```
  /* Registering port */
  port = check_region(0x378, 1);
  if (port) {
    printk("<1>parlelport: cannot reserve 0x378\n");
    result = port;
    goto fail;
  }
  request_region(0x378, 1, "parlelport");
```

“并口”驱动：卸载驱动

这部分和memory模块很相似，只是把释放内存，换成了释放并口保留的内存。这各功能通过release_region函数进行，它和check_region函数参数相同。

<*parlelport modified exit module*> =

```
  /* Make port free! */
  if (!port) {
    release_region(0x378,1);
  }
```

“并口”驱动：读取设备

本例中，需要增加一个真实设备的读取动作，以允许向用户空间传送信息。inb函数就是干这活的。它的参数是并口的内存地址，它的返回值是端口的内容。

<*parlelport inport*> =

```
/* Reading port */
parlelport_buffer = inb(0x378);
```

表 9 (和表2一样) 展示新函数

设备驱动事件和在内核空间和硬件之间实现该功能的函数

Events	Kernel functions
Read data	inb
Write data	

“并口”驱动：向设备写数据

同样的，需要增加一个向设备写数据的函数，以使过后，向用户空间传送信息成为可能。函数outb可以完成此功能；它的参数是要向端口写的数据以及端口的内存地址。

<*parlelport outport*> =

```
/* Writing to the port */
outb(parlelport_buffer,0x378);
```

表 10

设备驱动事件和在内核空间和硬件之间实现该功能的函数

Events	Kernel functions
Read data	inb
Write data	outb

完整的“并口”驱动

接下来，列除完整的并口模块代码。你需要把memory驱动里的单词memory 用parlelport替代。替代的结果如下：

<*parlelport.c*> =

```
<parlelport initial>
<parlelport init module>
```

```
<parlelport exit module>
<parlelport open>
<parlelport release>
<parlelport read>
<parlelport write>
```

初始化

在驱动初始化部分，并口使用了另外一个主设备号61。并且全局变量memory_buffer变成了port，还有，多了两个#include语句：ioport.h 和io.h。

<parlelport initial> =

```
/* Necessary includes for drivers */
#include <linux/init.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/ioport.h>
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */
#include <asm/io.h> /* inb, outb */

MODULE_LICENSE("Dual BSD/GPL");

/* Function declaration of parlelport.c */
int parlelport_open(struct inode *inode, struct file
*filp);
int parlelport_release(struct inode *inode, struct file
```

```
*filp);
ssize_t parlelport_read(struct file *filp, char *buf,
                        size_t count, loff_t *f_pos);
ssize_t parlelport_write(struct file *filp, char *buf,
                        size_t count, loff_t *f_pos);
void parlelport_exit(void);
int parlelport_init(void);

/* Structure that declares the common */
/* file access fcuntions */
struct file_operations parlelport_fops = {
  read: parlelport_read,
  write: parlelport_write,
  open: parlelport_open,
  release: parlelport_release
};

/* Driver global variables */
/* Major number */
int parlelport_major = 61;

/* Control variable for memory */
/* reservation of the parallel port*/
int port;

module_init(parlelport_init);
module_exit(parlelport_exit);
```

模块初始化

模块初始化，涉及了之前讲的并口数据存储方式。

<parlelport init module> =

```
int parlelport_init(void) {
```

```
716:   int result;
717:
718:   /* Registering device */
719:   result = register_chrdev(parlelport_major,
720: "parlelport",
721:       &parlelport_fops);
722:   if (result < 0) {
723:     printk(
724:       "<1>parlelport: cannot obtain major number %d\n",
725:       parlelport_major);
726:     return result;
727:   }
728:
729:   <parlelport modified init module>
730:
731:   printk("<1>Inserting parlelport module\n");
732:   return 0;
733:
734:   fail:
735:     parlelport_exit();
736:     return result;
737: }
```

738:

739: 卸载模块

740: 卸载方式和之前的memory驱动类似。

741: *<parlelport exit module> =*

```
742: void parlelport_exit(void) {
743:
744:   /* Make major number free! */
745:   unregister_chrdev(parlelport_major, "parlelport");
746:
747:   <parlelport modified exit module>
748:
749:   printk("<1>Removing parlelport module\n");
750: }
```

751:

752:

753: 打开设备

754: 这部分和memory驱动的一样

755: *<parlelport open> =*

```
756: int parlelport_open(struct inode *inode, struct file
757: *filp) {
758:
759:   /* Success */
760:   return 0;
761:
762: }
763:
```

764:

765: 关闭设备

766: 同样的，和上面的匹配，对应。

767: *<parlelport release> =*

```
768: int parlelport_release(struct inode *inode, struct file
769: *filp) {
770:
771:   /* Success */
772:   return 0;
773: }
```

774:

775: 读取设备

776: 读取函数和memory的相似，但被修改成读取设备的一个端口。

777: *<parlelport read> =*

```
778: ssize_t parlelport_read(struct file *filp, char *buf,
779:   size_t count, loff_t *f_pos) {
780:
781:   /* Buffer to read the device */
782:   char parlelport_buffer;
783:
784:   <parlelport inport>
785:
786:   /* We transfer data to user space */
787:   copy_to_user(buf,&parlelport_buffer,1);
788:
789:   /* We change the reading position as best suits */
790:   if (*f_pos == 0) {
791:     *f_pos+=1;
792:     return 1;
793:   } else {
794:     return 0;
795:   }
796: }
797:
```

798:

799: 向设备写数据

800: 和memory例子类似，向设备写数据。

801: *<parlelport write> =*

```
802: ssize_t parlelport_write( struct file *filp, char *buf,
803:   size_t count, loff_t *f_pos) {
804:
805:   char *tmp;
806:
807:   /* Buffer writing to the device */
808:   char parlelport_buffer;
```

```
809:
810:   tmp=buf+count-1;
811:   copy_from_user(&parlelport_buffer,tmp,1);
812:
813:   <parlelport outport>
814:
815:   return 1;
816: }
817:
```

818:

819: 使用LEDs测试“并口”驱动

820: 在这一节里，将介绍如何用几个简单的LED灯以直观的显示并口
821: 的状态。

822: 警告：连接设备到并口可能伤害你的计算机。请确认电路接地，
823: 并且在设备连接电脑时，电脑是关闭的。你要为该实验可能引起
824: 的任何问题承担责任。

825: 按图3所示电路建立实验设备。

826:

827: 需要首先确认所有硬件连接正确。接下来，关闭PC，把所建的
828: 设备连接到并口。然后打开PC，并且，所有卸载所有并口相关
829: 的模块（如,lp, parport, parport_pc等）。Debian Sarge的hotplug
830: 模块可能引起麻烦，所以也需要卸载。如果文件/dev/
831: parlelport不存在，请首先用以下命令建立该文件：

832: `# mknod /dev/parlelport c 61 0`

833: 改变并口的权限，是任何人都可以读或写：

834: `# chmod 666 /dev/parlelport`

模块parlelport现在可以安装了。可以通过以下命令检查它分配到的输入输出端口地址是否是0x378：

```
$ cat /proc/ioports
```

执行以下命令，可打开LED并且检查系统是否工作正常：

```
$ echo -n A >/dev/parlelport
```

0号和6号LED应该亮了，其它的则该是熄的。

可以通过以下命令检查并口状态：

```
$ cat /dev/parlelport
```

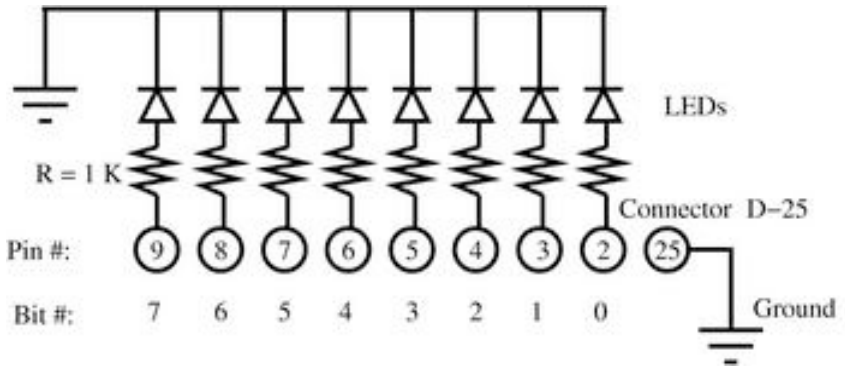

图 3: 监控并口的LED的电路图

最终的应用程序：闪光灯

最后，我开发了一个有趣的应用程序：它可以让LED不停的闪烁。要实现这个功能需要用户空间应用程序，一次只向/dev/parlelport设备写一个字位(bit)。

<lights.c> =

```
#include <stdio.h>
#include <unistd.h></p>

int main() {
```

```
  unsigned char byte,dummy;
  FILE * PARLELPORT;

  /* Opening the device parlelport */
  PARLELPORT=fopen("/dev/parlelport","w");
  /* We remove the buffer from the file i/o */
  setvbuf(PARLELPORT,&dummy,_IONBF,1);

  /* Initializing the variable to one */
  byte=1;

  /* We make an infinite loop */
  while (1) {
    /* Writing to the parallel port */
    /* to turn on a LED */
    printf("Byte value is %d\n",byte);
    fwrite(&byte,1,1,PARLELPORT);
    sleep(1);

    /* Updating the byte value */
    byte<<=1;
    if (byte == 0) byte = 1;
  }

  fclose(PARLELPORT);

}
```

编译:

```
$ gcc -o lights lights.c
```

执行:

```
$ lights
```

LED灯将一个接一个的闪烁。图4是闪烁的LED灯和运行该程序

的Linux系统。

总结

在跟着这份手册一步步学习过来，你该有能力为些简单硬件写驱动了，比如一个简单的继电器盒（见附录C），或者复杂硬件的最小限度的设备驱动。学习理解Linux内核内部的一些简单原理、概念，可以快速提高写设备驱动的能力。并且，这将使你离成为真正的Linux内核开发人员越来越近。

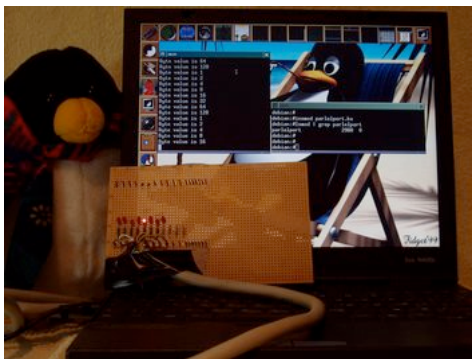

图 4: 闪烁的LED灯固定在线路板上。计算机真在运行Linux，系统开了两个终端：一个显示“parlelport”模块已被加载，另外一个显示“lights”程序正在运行。Linux很直观的显示什么正在运行。

参考文献

A. Rubini, J. Corbert. 2001. Linux device drivers (second edition). Ed. O’Reilly. This book is available for free on the internet.

Jonathan Corbet. 2003/2004. Porting device drivers to the 2.6 kernel. This is a very valuable resource for porting drivers to the new 2.6 Linux kernel and also for learning about Linux device drivers.

B. Zoller. 1998. PC & Electronics: Connecting Your PC to the Outside World (Productivity Series). Nowadays it is probably easier to surf the web for hardware projects like this one.

M. Waite, S. Prata. 1990. C Programming. Any other good book on C

programming would suffice.

附录A. 完整的Makefile

<*Makefile*> =

```
obj-m := nothing.o hello.o memory.o parlelport.o
```

附录B. 在Debian Sarge系统上编译内核

在Debian Sarge系统下编译2.6.x内核的步骤（所有步骤都需以root权限执行）：

1. 安装“kernel-image-2.6.x”软件包。
2. 从新启动系统，以使用新的内核。这个步骤Debian可以自动完成。你可能还需要修改下/etc/lilo.conf文件然后执行lilo命令。（如果使用grub引导系统，则不需要休息lilo）
3. 安装“kernel-source-2.6.x”软件包。
4. 进入源代码目录：`cd /usr/src`，并解开源代码：`bunzip2 kernel-source-2.6.x.tar.bz2`；`tar xvf kernel-source-2.6.x.tar`。进入内核源代码目录：`cd /usr/src/kernel-source-2.6.x`
5. 拷贝Debian内核的默认配置文件到当前的内核源码目录：

   ```
   cp /boot/config-2.6.x .config.
   ```

6. 编译内核及模块：`make`；`make modules.`

附录 C. 练习

如果你想接受一些更大的挑战，这里有一些练习你可以做下:

1. 我曾经为两个ISA接口的 Meilhaus板写了两个驱动程序，一

个数字转接头（ME26）和一个继电控制器板（ME53）。这两个驱动可以从 ADQ 项目下载到。以我的ISA接口驱动为基础，为新的PCI接口的Meilhaus板开发驱动。

2. 找出一些目前还不能在Linux下正常工作（有另外一个使用相似芯片的设备且提供了Linux驱动）的设备。试着修改已有驱动，使它能够支持你的新设备。如果你成功了，你可以提交你的代码，并且使自己成为一个内核开发者。

后记

从本文第一版推出，已经三年过去了。最初，打算用西班牙文写的，针对2.2内核，但是当时2.4内核已经开始可用了。写这份文档的 原因是写设备驱动的好文档——《**Linux device drivers**》一书，比内核的发行延时好几个月。本文档新版本同样在新的2.6内核推出不久后出来了。目前最及时的文档可以在 Linux Weekly News 里看到，它可是使本文适用与最新的内核。

非常幸运的是，PC仍然内建了并口，使得本文所讲的并口的例子可以实际操作。我们期望PC在以后的日子里，继续内建并口，或者，至少，仍然有PCI接口的并口在卖。

本文使用文本编辑器（emacs）写成，使用noweb格式。然后写成的文档经过noweb工具处理生成LaTex文件（.tex）和源代码文件(.c)所有这些可以通过提供的`makefile.document`文件使用`make -f makefile.document`命令完成。

我要感谢 “Instituto Politécnico de Bragança”, “Núcleo Estudantil de Linux del Instituto Politécnico de Bragança (NUX)”, “Asociación de Software Libre de León (SLeón)” 还有 “Núcleo de Estudantes de Engenharia Informática da Universidade de Évora”的帮助。