

# 1: Linux设备驱动开发入门

2: 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  
3: 开发linux设备驱动

4:

5: 源作者: [Xavier Calbet](#)

6: 版权: GNU Free Documentation License

7:

8: 翻译: 顾宏军 (<http://www.oss-cn.com>)

9:

10: 中文版权: 创作共用.署名-非商业用途-保持一致

11:

## 12: 知识准备

13: 要开发Linux设备驱动, 需要掌握以下知识:

- 14: • C编程 需要掌握深入一些的C语言知识, 比如, 指针的使  
15: 用, 位处理函数, 等。
- 16: • 微处理器编程 需要理解微机的内部工作原理: 存储器地  
17: 址, 中断, 等。这些内容对一个汇编程序员应该比较熟  
18: 悉。

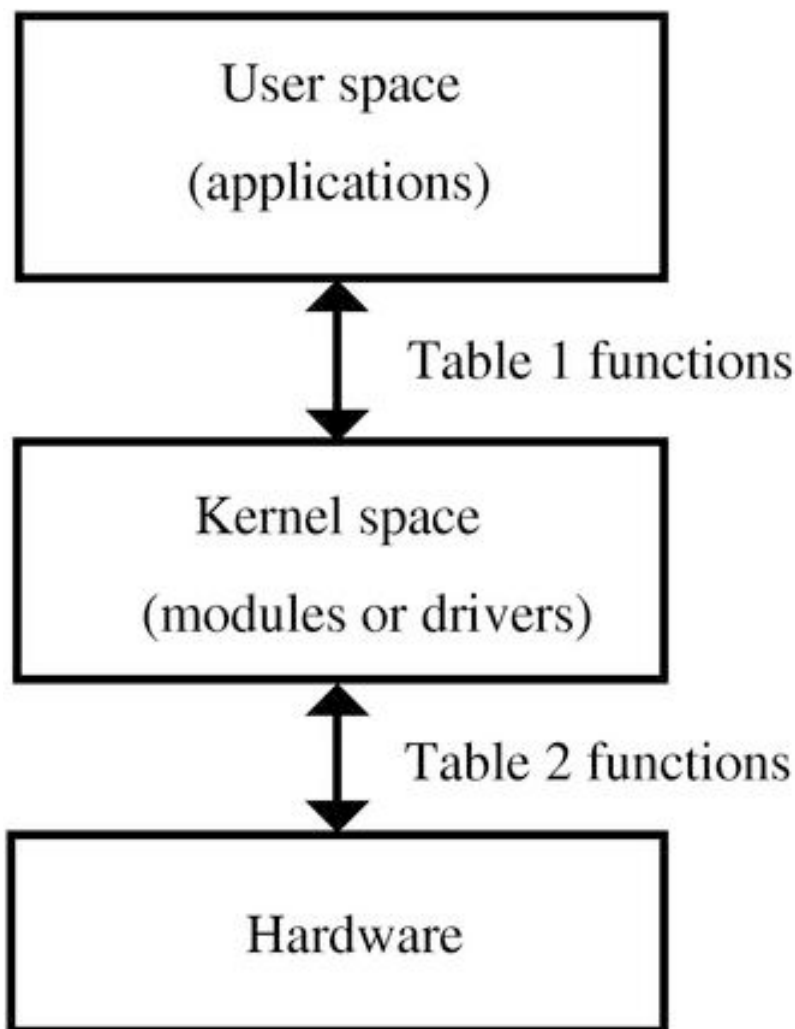
19: Linux下有好几种不同的设备。为简单起见, 本文只涉及以模块  
20: 形式加载的字符设备。使用2.6.x的内核。(特别是Debian Sarge  
21: 使用的2.6.8内核。)

## 22: 用户空间和内核空间

23: 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  
24: 区别。

25:

- 26: • 内核空间：Linux操作系统，特别是它的内核，用一种简单  
27: 而有效的方法管理机器的硬件，给用户提供一个简捷而统  
28: 一的编程接口。同样的，内核，特别是它的设备驱动程  
29: 序，是连接最终用户/程序员和硬件的一坐桥或者说是接  
30: 口。任何子程序或者函数只要是内核的一部分（例如：模  
31: 块，和设备驱动），那它也就是内核空间的一部分。  
32:
- 33: • 用户空间. 最终用户的应用程序，像UNIX的shell或者其它的  
34: GUI的程序(例如，gedit),都是用户空间的一部分。很显然，  
35: 这些应用程序需要和系统的硬件进行交互。但是，他们不  
36: 是直接进行，而是通过内核支持的函数进行。  
37: 它们的关系可以通过下图表示：



39: 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

40:

## 41: 用户空间和内核空间之间的接口函数

42:

43: 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  
 44: 程序员和硬件进行交互。通常，在UNIX或者Linux系统中，这种  
 45: 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  
 46: 因为从用户的视角看，UNIX的设备就是一个个文件。

47: 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  
 48: 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  
 49: 传递信息。

50: 通常，用户空间的每个函数（用于使用设备或者文件的），在内  
 51: 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  
 52: 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  
 53: 们后面每个表项都会填入对应的函数。

54: 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

56: 事件	用户函数	内核函数
60: 加载模块		
62: 打开设备		
66: 读设备		
68: 写设备		
70: 关闭设备		
78: 卸载模块		

76:

77:

## 78: 内核空间和硬件设备之间的接口函数

79: 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  
 80: 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  
 81: 到相应内容时填写上。

82:

83: 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

84: 事件	内核函数
--------	------

ddd

```
89: 读数据  
88: 写数据
```

90:

91:

## 92: 第一个驱动：在用户空间加载和卸载驱动

93:

94: 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  
95: 在。

96: 首先，写一个文件名为nothing.c的文件，代码如下：

97: `<nothing.c> =`

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

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

100:

101: 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  
102: Sarge系统，你可以按照附录B（在本文末尾）的步骤进行操作。  
103: 在以下的内容里，假设你使用的是2.6.8内核。

104:

105: 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：

106: `<Makefile1> =`

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

108:

109: 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  
110: 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：

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

112: 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

ddd

116: 的一部分。

117: 在用户空间，你可以以root账号加载该模块，命令如下：

118: # insmod nothing.ko

119: 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  
120: 用。

121: 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  
122: 进行：

123: # lsmod

124: 最后，模块可以通过以下命令从内核中移除：

125: # rmmmod nothing

126: 同样的，使用lsmod命令，可以用于验证该模块已不在内核中。

127: 主要内容整理在如下表格里。

128: 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	rmmmod	

130:  
131:  
132:  
133:  
134:  
135:  
136:  
137:  
138:  
139:  
140:  
141:  
142:  
143:  
144:  
145:  
146:  
147:  
148:  
149:  
150:

151:  
152:

## 153: “hello world”驱动：在内核空间加载和 154: 移除驱动

155:

156: 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  
157: 括：设备复位，初始化RAM，初始化中断，初始化输入/输出端

ddd

158: 口，等。

159:

160: 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：  
161: `module_init` 和 `module_exit`；它们和用户空间的用于安装和卸载模  
162: 块的命令 `insmod` 和 `rmmod` 对应。也可以说，用户空间的命令  
163: `insmod` 和 `rmmod` 使用内核空间的函数 `module_init` 和 `module_exit` 进  
164: 行。

165:

166: 我们通过一个最基本的hello world 程序，看实际的例子：

167: `<hello.c> =`

```
168: #include <linux/init.h>
169: #include <linux/module.h>
170: #include <linux/kernel.h>
171: MODULE_LICENSE("Dual BSD/GPL");
172: static int hello_init(void) {
173:     printk("<1> Hello world!\n");
174:     return 0;
175: }
176: static void hello_exit(void) {
177:     printk("<1> Bye, cruel world\n");
178: }
179: module_init(hello_init);
180: module_exit(hello_exit);
```

181:

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

ddd

183: 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  
184: module\_init和module\_exit的参数。

185:

186: 以上代码里还包括了printk函数。它和我们非常熟悉的printf函数  
187: 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  
188: (数字)。这样就可以通过内核的日志文件里看到该消息，该消  
189: 息也会在系统控制台中显示。

190:

191: 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  
192: 把它的名字加在Makefile文件里。

193: <Makefile2> =

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

195:

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

198:

199: 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  
200: 控制台上。如果这个消息没有在控制台上显示，可以通过dmesg  
201: 命令，或者查看系统的日志文件cat /var/log/syslog命令看到。

202: 表4 填入了两个新函数。

203: 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()

ddd

226:

227:

228: 一个完整的驱动 “memory” :驱动的初始  
229: 化

230:

231: 现在我开始构建一个完整的设备驱动: memory.c。可以从这个设  
232: 备读取和写入一个字符。虽然这个设备没什么用途,但提供了个  
233: 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  
234: 实际的硬件设备(它是电脑内部模拟的硬件)。

235:

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

238: *<memory initial> =*

239: /\* Necessary includes for device drivers \*/

240: #include <linux/init.h>

241: #include <linux/config.h>

242: #include <linux/module.h>

243: #include <linux/kernel.h> /\* printk() \*/

244: #include <linux/slab.h> /\* kmalloc() \*/

245: #include <linux/fs.h> /\* everything... \*/

246: #include <linux/errno.h> /\* error codes \*/

247: #include <linux/types.h> /\* size\_t \*/

248: #include <linux/proc\_fs.h>

249: #include <linux/fcntl.h> /\* O\_ACCMODE \*/

250: #include <asm/system.h> /\* cli(), \*\_flags \*/

251: #include <asm/uaccess.h> /\* copy\_from/to\_user \*/

252:

253: MODULE\_LICENSE("Dual BSD/GPL");

254:

255: /\* Declaration of memory.c functions \*/

256: int memory\_open(struct inode \*inode, struct file \*filp);

257: int memory\_release(struct inode \*inode, struct file

258: \*filp);



ddd

```
259: ssize_t memory_read(struct file *filp, char *buf, size_t
260: count, loff_t *f_pos);
261: ssize_t memory_write(struct file *filp, char *buf,
262: size_t count, loff_t *f_pos);
263: void memory_exit(void);
264: int memory_init(void);
265:
266: /* Structure that declares the usual file */
267: /* access functions */
268: struct file_operations memory_fops = {
269:     read: memory_read,
270:     write: memory_write,
271:     open: memory_open,
272:     release: memory_release
273: };
274:
275: /* Declaration of the init and exit functions */
276: module_init(memory_init);
277: module_exit(memory_exit);
278:
279: /* Global variables of the driver */
280: /* Major number */
281: int memory_major = 60;
282: /* Buffer to store data */
283: char *memory_buffer;
284:
```

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

290:

## 291: “memory”驱动：连接到设备

292: 在UNIX和Linux中，设备可以用和文件一样的方式从用户空间访

293: 问。这些设备文件通常在/dev目录下。

294: 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  
295: 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  
296: 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  
297: 释。

ddd

298:

299: 需要创建一个文件（该设备文件用于和设备驱动操作），

300: # mknod /dev/memory c 60 0

301: 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。

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

305: *<memory init module> =*

```
306: int memory_init(void) {
307:     int result;
308:
309:     /* Registering device */
310:     result = register_chrdev(memory_major, "memory",
311: &memory_fops);
312:     if (result < 0) {
313:         printk(
314:             "<1>memory: cannot obtain major number %d\n",
315: memory_major);
316:         return result;
317:     }
318:
319:     /* Allocating memory for the buffer */
320:     memory_buffer = kmalloc(1, GFP_KERNEL);
321:     if (!memory_buffer) {
322:         result = -ENOMEM;
323:         goto fail;
324:     }
325:     memset(memory_buffer, 0, 1);
326:
327:     printk("<1>Inserting memory module\n");
328:     return 0;
329:
330: fail:
331:     memory_exit();
```

ddd

```
332:     return result;
333: }
```

334:

335: 以上代码使用了kmalloc函数。这个函数工作在内核空间，用于为  
336: 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  
337: 似。最后，如果注册主设备号或者分配内存失败，模块将退出。

338:

## 339: “memory”驱动：卸载驱动

340: 为通过memory\_exit函数卸载模块，需要定义unregister\_chrdev函  
341: 数。这将释放驱动之前向内核申请的主设备号。

```
342: <memory exit module> =
```

```
343: void memory_exit(void) {
344:     /* Freeing the major number */
345:     unregister_chrdev(memory_major, "memory");
346:
347:     /* Freeing buffer memory */
348:     if (memory_buffer) {
349:         kfree(memory_buffer);
350:     }
351:
352:     printk("<1>Removing memory module\n");
353:
354: }
```

355:

356: 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。

## 357: “momory”驱动：像文件一样打

# 开设备

358:

359: 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  
 360: fopen对应：在file\_operations结构里，用于调用register\_chrdev。在本例  
 361: 里，是memory\_open函数。它有几个参数：一个inode结构，该结构向内核发送  
 362: 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  
 363: 哪些操作。所有这些函数在本文中从未做深入的讲解。

364: 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  
 365: 备进行复位。但在本例中，这些操作都没进行。

366: memory\_open函数定义如下：

367: *<memory open> =*

```
368: int memory_open(struct inode *inode, struct file *filp)
369: {
370:
371:     /* Success */
372:     return 0;
373: }
```

374:

375: 表5

376: 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

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

398:

## 399: “monory”驱动：像文件一样关 400: 闭设备

401:

402: 在内核空间里，和用户空间里关闭文件的fclose对应的函数是  
403: release：它也是file\_operations结构体的成员，用于调用  
404: register\_chrdev。本例中，它是函数memory\_release，和上面的相似，  
405: 它也有inode和file两个参数。

406: 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  
407: 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  
408: 些操作。

409: memory\_release函数定义如下：

410: *<memory release> =*

```
411: int memory_release(struct inode *inode, struct file
412: *filp) {
413:
414:     /* Success */
415:     return 0;
416: }
```

417:

418:

419: 表6.

420: 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

421: Events	User functions	Kernel functions
422: Load module	insmod	module_init()
423: Open device	fopen	file_operations: open
424: Read device		
425: Write device		
426: Close device	fclose	file_operations: release
427: Remove module	rmmod	module_exit()

442:

## 443: "memory" 驱动：读取设备

444: 和用户空间函数fread类似，内核空间里，读取设备文件使用read  
 445: 函数：read是file\_operations的成员，用于调用register\_chrdev。  
 446: 本例中，是memory\_read函数。它的参数有：一个file结构；一个缓冲区  
 447: (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  
 448: 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  
 449: 参数（f\_pos）指示从哪里开始读取该设备文件。

450: 本例中，memory\_read函数通过copy\_to\_user函数从驱动的缓冲区  
 451: （memory\_buffer）向用户空间传送一个简单的字节：

452: <memory read> =

```
453: ssize_t memory_read(struct file *filp, char *buf,
454:                    size_t count, loff_t *f_pos) {
455:
456:     /* Transferring data to user space */
457:     copy_to_user(buf, memory_buffer, 1);
458:
459:     /* Changing reading position as best suits */
460:     if (*f_pos == 0) {
461:         *f_pos += 1;
462:         return 1;
463:     } else {
464:         return 0;
465:     }
466: }
```

468:

469: 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  
 470: 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  
 471: 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

ddd

472: 值将是0，（因为文件只存储了1个字节）。

473: 表7. 设备驱动事件和在内核空间 and 用户空间之间实现该功能的函数  
474:

476:	Events	User functions	Kernel functions
480:	Load module	insmod	module_init()
483:	Open device	fopen	file_operations: open
486:	Read device	fread	file_operations: read
488:	Write device		
490:	Close device	fclose	file_operations: release
495:			module_exit()
493:	Remove modules	rmmod	
496:			

## 497: “memory”驱动：向设备写数据

498:

499: 和用户空间里写文件的fwrite对应，内核空间里是write:write是  
500: file\_operations的成员，用于调用register\_chrdev。本例中是  
501: memory\_write函数，它有如下几个参数：一个file结构；buf，一个缓冲区，  
502: 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  
503: 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  
504: 里开始写文件。

505: *<memory write> =*

```
506: ssize_t memory_write( struct file *filp, char *buf,  
507:                       size_t count, loff_t *f_pos) {  
508:  
509:     char *tmp;  
510:  
511:     tmp=buf+count-1;  
512:     copy_from_user(memory_buffer,tmp,1);  
513:     return 1;  
514: }
```



ddd

515:

516: 本例中，函数`copy_from_user`从用户空间传送数据到内核空间。

517: 表8

518: 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

520:	Events	User functions	Kernel functions
522:	Load module	<code>insmod</code>	<code>module_init()</code>
526:	Open device	<code>fopen</code>	<code>file_operations: open</code>
528:	Close device	<code>fread</code>	<code>file_operations: read</code>
532:	Write device	<code>fwrite</code>	<code>file_operations: write</code>
536:	Close device	<code>fclose</code>	<code>file_operations: release</code>
538:	Remove module	<code>rmmod</code>	<code>module_exit()</code>

540:

## 541: 完整的“memory”驱动

542: 把以上各部分代码整合起来，一个完整的驱动就完成了：

543: `<memory.c> =`

544: `<memory initial>`

545: `<memory init module>`

546: `<memory exit module>`

547: `<memory open>`

548: `<memory release>`

549: `<memory read>`

550: `<memory write>`

551:

552:

553: 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。  
554: 编译好后，用以下命令进行加载：

ddd

555: # insmod memory.ko

556: 并且最后是取出设备文件的保护:

557: # chmod 666 /dev/memory

558: 如果以上步骤一切正常, 此时你就可以向设备/dev/memory写一串  
559: 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:

560: \$ echo -n abcdef >/dev/memory

561: 使用cat检查设备的内容:

562: \$ cat /dev/memory

563: 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  
564: 载。

565:

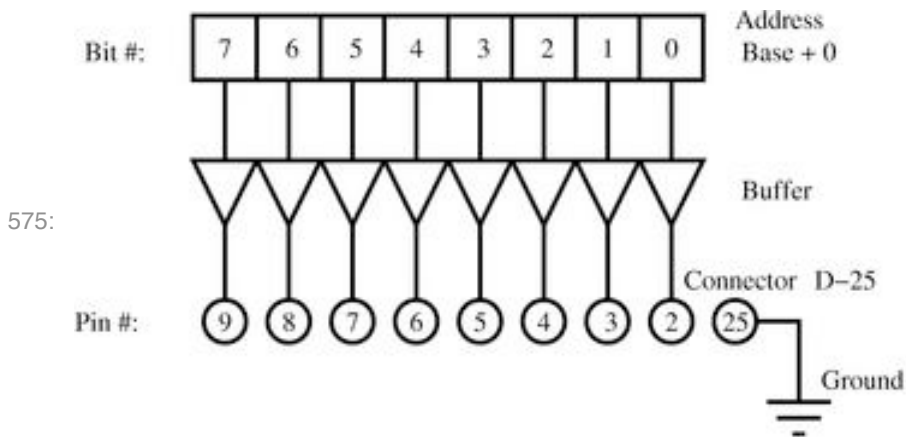
## 566: 真实的“并口”驱动: 描述并口

567: 接下来, 将修改刚刚写的memory驱动, 来在一个真实设备上进  
568: 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  
569: 动的名称叫做: parlepport。

570: 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  
571: D-25接头, 有25针。从内部看, 从CPU视图看, 并口有3字节  
572: 的存储, 在PC上, 基地址 (设备的起始地址) 通常是0x378。在本  
573: 例中, 我们仅使用包含完整数字输出的第一个字节。

574: 上面提到的字节和外部接头针脚之间的连接情况不下图:

ddd



576: 图2: 并口的第一个字节和它在D-25连接头上对应的针脚

577:

## 578: “并口” 驱动：模块初始化

579: 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  
580: 存地址 (0x378)。check\_region函数用于检查一个内存区域是否可用，并  
581: 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  
582: 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  
583: 一个指定模块名称的字符串。

```
584: <parlelport modified init module> =  
  
585:  /* Registering port */  
586:  port = check_region(0x378, 1);  
587:  if (port) {  
588:      printk("<1>parlelport: cannot reserve 0x378\n");  
589:      result = port;  
590:      goto fail;  
591:  }  
592:  request_region(0x378, 1, "parlelport");  
593:
```

594:

## “并口”驱动：卸载驱动

595:

596: 这部分和memory模块很相似，只是把释放内存，换成了释放并  
597: 口保留的内存。这各功能通过release\_region函数进行，它和  
598: check\_region函数参数相同。

599: *<parlelport modified exit module> =*

```
600:  /* Make port free! */
601:  if (!port) {
602:      release_region(0x378,1);
603:  }
604:
```

605:

## “并口”驱动：读取设备

606:

607: 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  
608: 传送信息。inb函数就是干这活的。它的参数是并口的内存地  
609: 址，它的返回值是端口的内容。

610: *<parlelport inport> =*

```
611:  /* Reading port */
612:  parlelport_buffer = inb(0x378);
613:
614:
```

615: 表 9 (和表2一样) 展示新函数

616: 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

617: Events    Kernel functions

620: Read data inb

621: Write data

ddd

622:

## 623: “并口”驱动：向设备写数据

624: 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  
625: 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  
626: 要向端口写的的数据以及端口的内存地址。

627: *<parlelport outport> =*

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

630:

631:

632:

633: 表 10

634: 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

636: Events      Kernel functions

638: Read data inb

639: Write data outb

641:

## 642: 完整的“并口”驱动

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

645: *<parlelport.c> =*

```
646: <parlelport initial>
```

```
647: <parlelport init module>
```

ddd

```
648: <parlelport exit module>
649: <parlelport open>
650: <parlelport release>
651: <parlelport read>
652: <parlelport write>
653:
```

654:

## 初始化

655:

656: 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  
657: 变量memory\_buffer变成了port，还有，多了两个#include语句：  
658: ioport.h 和io.h。

```
659: <parlelport initial> =
```

```
660: /* Necessary includes for drivers */
661: #include <linux/init.h>
662: #include <linux/config.h>
663: #include <linux/module.h>
664: #include <linux/kernel.h> /* printk() */
665: #include <linux/slab.h> /* kcalloc() */
666: #include <linux/fs.h> /* everything... */
667: #include <linux/errno.h> /* error codes */
668: #include <linux/types.h> /* size_t */
669: #include <linux/proc_fs.h>
670: #include <linux/fcntl.h> /* O_ACCMODE */
671: #include <linux/ioport.h>
672: #include <asm/system.h> /* cli(), *_flags */
673: #include <asm/uaccess.h> /* copy_from/to_user */
674: #include <asm/io.h> /* inb, outb */
675:
676: MODULE_LICENSE("Dual BSD/GPL");
677:
678: /* Function declaration of parlelport.c */
679: int parlelport_open(struct inode *inode, struct file
680: *filp);
681: int parlelport_release(struct inode *inode, struct file
```

ddd

```
682: *filp);
683: ssize_t parlepport_read(struct file *filp, char *buf,
684:                         size_t count, loff_t *f_pos);
685: ssize_t parlepport_write(struct file *filp, char *buf,
686:                          size_t count, loff_t *f_pos);
687: void parlepport_exit(void);
688: int parlepport_init(void);
689:
690: /* Structure that declares the common */
691: /* file access fcuntions */
692: struct file_operations parlepport_fops = {
693:     read: parlepport_read,
694:     write: parlepport_write,
695:     open: parlepport_open,
696:     release: parlepport_release
697: };
698:
699: /* Driver global variables */
700: /* Major number */
701: int parlepport_major = 61;
702:
703: /* Control variable for memory */
704: /* reservation of the parallel port*/
705: int port;
706:
707: module_init(parlepport_init);
708: module_exit(parlepport_exit);
709:
```

710:

## 711: 模块初始化

711:

712: 模块初始化，涉及了之前讲的并口数据存储方式。

712:

713: *<parlepport init module>* =

714:

```
715: int parlepport_init(void) {
```

ddd

```
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:
```

## 卸载模块

739:

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: }
```



ddd

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

ddd

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;
```

ddd

```
809:
810:     tmp=buf+count-1;
811:     copy_from_user(&parlelport_buffer,tmp,1);
812:
813:     <parlelport output>
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
```

ddd

835: 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:

837: `$ cat /proc/iports`

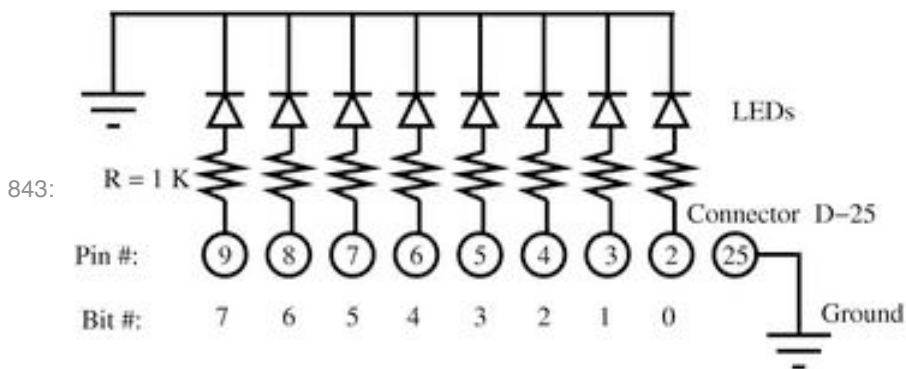
838: 执行以下命令，可打开LED并且检查系统是否工作正常:

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

840: 0号和6号LED应该亮了，其它的则该是熄的。

841: 可以通过以下命令检查并口状态:

842: `$ cat /dev/parlelport`



844: 图 3: 监控并口的LED的电路图

845:

## 846: 最终的应用程序：闪光灯

847: 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)。

850: `<lights.c> =`

851: `#include <stdio.h>`

852: `#include <unistd.h></p>`

853:

854: `int main() {`

ddd

```
855: unsigned char byte,dummy;
856: FILE * PARLELPORT;
857:
858: /* Opening the device parlelport */
859: PARLELPORT=fopen("/dev/parlelport","w");
860: /* We remove the buffer from the file i/o */
861: setvbuf(PARLELPORT,&dummy,_IONBF,1);
862:
863: /* Initializing the variable to one */
864: byte=1;
865:
866: /* We make an infinite loop */
867: while (1) {
868:     /* Writing to the parallel port */
869:     /* to turn on a LED */
870:     printf("Byte value is %d\n",byte);
871:     fwrite(&byte,1,1,PARLELPORT);
872:     sleep(1);
873:
874:     /* Updating the byte value */
875:     byte<<=1;
876:     if (byte == 0) byte = 1;
877: }
878:
879: fclose(PARLELPORT);
880:
881: }
882:
883:
```

884: 编译:

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

886: 执行:

```
887: $ lights
```

888: 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

889: 的Linux系统。

## 890: 总结

891: 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  
892: 动了，比如一个简单的继电器盒（见附录C），或者复杂硬件的  
893: 最小限度的设备驱动。学习理解Linux内核内部的一些简单原  
894: 理、概念，可以快速提高写设备驱动的能力。并且，这将使你离  
895: 成为真正的Linux内核开发人员越来越近。



896:   
897: 图 4: 闪烁的LED灯固定在线路板上。计算机真在运行Linux，系  
898: 统开了两个终端：一个显示“parlelport”模块已被加载，另外一个  
899: 显示“lights”程序正在运行。Linux很直观的显示什么正在运行。

## 900: 参考文献

901: A. Rubini, J. Corbert. 2001. [Linux device drivers \(second edition\)](#). Ed.  
902: O'Reilly. This book is available for free on the internet.

903: Jonathan Corbet. 2003/2004. [Porting device drivers to the 2.6 kernel](#).  
904: This is a very valuable resource for porting drivers to the new 2.6  
905: Linux kernel and also for learning about Linux device drivers.

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

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

ddd

910: programming would suffice.

## 911: 附录A. 完整的Makefile

912: *<Makefile>* =

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

914:

915:

## 916: 附录B. 在Debian Sarge系统上编译内核

917:

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

- 920: 1. 安装“kernel-image-2.6.x”软件包。
- 921: 2. 从新启动系统，以使用新的内核。这个步骤Debian可以自动  
922: 完成。你可能还需要修改下/etc/lilo.conf文件然后执行lilo命  
923: 令。（如果使用grub引导系统，则不需要休息lilo）
- 924: 3. 安装“kernel-source-2.6.x”软件包。
- 925: 4. 进入源代码目录：cd /usr/src，并解开源代码： bunzip2  
926: kernel-source-2.6.x.tar.bz2；tar xvf kernel-  
927: source-2.6.x.tar。进入内核源代码目录：cd /usr/src/  
928: kernel-source-2.6.x
- 929: 5. 拷贝Debian内核的默认配置文件到当前的内核源码目录：  
930: cp /boot/config-2.6.x .config.
- 931: 6. 编译内核及模块：make；make modules.

## 932: 附录C. 练习

933: 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：

- 934: 1. 我曾经为两个ISA接口的 [Meilhaus](#)板写了两个驱动程序，一

935: 个数字转接头 (ME26) 和一个继电器控制板 (ME53)。  
936: 这两个驱动可以从 [ADQ](#) 项目下载到。以我的ISA接口驱动  
937: 为基础, 为新的PCI接口的[Meilhaus](#)板开发驱动。  
938: 2. 找出一些目前还不能在Linux下正常工作 (有另外一个使用  
939: 相似芯片的设备且提供了Linux驱动) 的设备。试着修改已  
940: 有驱动, 使它能够支持你的新设备。如果你成功了, 你可  
941: 以提交你的代码, 并且使自己成为一个内核开发者。

942:

## 后记

943:

944: 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  
945: 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  
946: 档的原因是写设备驱动的好文档——《**Linux device drivers**》一  
947: 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  
948: 内核推出不久后出来了。目前最及时的文档可以在 [Linux Weekly](#)  
949: [News](#) 里看到, 它可是使本文适用与最新的内核。

950: 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  
951: 子可以实际操作。我们期望PC在以后的日子里, 继续内建并  
952: 口, 或者, 至少, 仍然有PCI接口的并口在卖。

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

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