第32卷 第23期 计 算 机 工 程 2006年12月
Vol.32 No.23 Computer Engineering December 2006

·开发研究与设计技术· 文章编号：1000—3428(2006)23—0278—03 文献标识码：A 中图分类号：TP311.52

基于 Linux 字符设备驱动程序的设计与实现

王粉花

（北京科技大学信息工程学院，北京 100083）

摘 要：描述了 Linux 系统设备驱动程序的概念、分类及关键技术，包括设备的设备号、设备的操作及设备的注册和卸载等。以 LED 设备驱动程序为例，分析了 Linux 系统下字符设备驱动程序的设计方法。该文列出的 LED 驱动程序已在 Samsung ARM9 2410 开发板上调试通过。

关键词：Linux；驱动程序；LED；嵌入式系统

Design & Implementation of Char Drive Procedures Based on Linux

WANG Fenhua

(College of Information Engineering, Univ. of Science and Technology Beijing, Beijing 100083)

【**Abstract**】This paper describes the conceptions, classification and key technology of the drivers on Linux, including numbers, operations, register and uninstall of the devices. By taking LED drivers as the example, the design method of char drivers on Linux is discussed. The LED driving procedures are debugged on Samsung ARM9 2410 development board.

【**Key words**】Linux; Driver; LED; Embedded system

Linux 设备驱动程序是为特定的硬件提供给用户程序的一组标准化接口，它隐藏了设备工作的细节。Linux 系统下驱动程序是运行在内核态的，是和内核连接在一起的程序。如果运行在用户态的应用程序想控制硬件设备，必须通过驱动程序来控制。在 Linux 系统里，任何设备都以文件的形式来表示，也就是说，对设备文件的操作实质反映的是对设备的操作。Linux 系统的设备分为 3 种类型，分别是字符设备、块设备和网络设备。

1 Linux 设备的主设备号和次设备号

Linux 系统为每一个设备分配了一个主设备号和次设备号，主设备号标识设备对应驱动程序，次设备号标识具体设备的实例。例如一块开发板上有 2 个串口终端（/dev/tty0，/dev/tty1），它们的主设备号都是 4，次设备号分别为 0 和 1。

每一类设备使用的主设备号是独一无二的，系统增加一个驱动程序就要赋予它一个主设备号。这一赋值过程在驱动程序的初始化过程中进行。

2 Linux 设备的操作

Linux 系统访问设备就像访问文件一样，例如打开设备使用系统调用 open( )，关闭设备使用系统调用 close( )。在 Linux 内核中，字符设备使用 struct file_operations 结构来定义设备的各种操作集合。编写字符设备驱动程序，主要是实现 struct file_operations 结构中的各个函数。对于结构中没有实现的操作函数，函数指针变量设置为 NULL。struct file_operations 结构在/include/linux/fs.h 文件中定义，下面介绍该结构体所包含的函数操作。

```
Struct file_operations {
  struct module *owner;
   //owner 结构指针指向结构体所属于的驱动程序模块
  loff_t (*llseek) (struct file *, loff_t, int); //改变文件的当前读写位置
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
     //从设备读取数据
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
     //向设备写入数据
    int (*readdir) (struct file *, void *, filldir_t);
     //用于读取目录，只对文件系统有用
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
     //用来查询设备状态
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long) ;//提供用户程序，对设备执行特定的命令的方法
    int (*mmap) (struct file *, struct vm_area_struct *);
     //将设备的内存映射到用户进程的地址空间中去
    int (*open) (struct inode *, struct file *); //打开设备
  int (*flush) (struct file *); /刷新设备缓存区
  int (*release) (struct inode *, struct file *); //释放结构体
  int (*fsync) (struct file *, struct dentry *, int datasync);
     //刷新待处理的数据
  int (*fasync) (int, struct file *, int); //通知设备
  int (*lock) (struct file *, int, struct file_lock *); //实现文件的锁定
  ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int) ;//实现“分散/聚集”型的读写操作
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);}
```

作者简介：王粉花(1971－)，女，博士、副教授，主研方向：智能控制，单片机及嵌入式技术应用

收稿日期：2006-01-25 **E-mail**：wfh_2001@sohu.com

3 Linux 设备的注册和卸载

设备驱动程序所提供的入口点，在设备驱动程序初始化的时候向系统进行注册，以便系统在适当的时候调用。在 Linux 系统中，通过调用 register_chrdev 向系统注册字符型设备驱动程序。register_chrdev 定义为

```
# include <linux/fs.h>
# include <linux/errno.h>
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops);
```

其中，major 是主设备号，如果为 0 则系统为此驱动程序动态地分配一个主设备号。Name 是设备名称，本文以 LED_DEV 为设备名称。设备是以文件的形式存在的，设备注册时需要使用一个文件结构 struct file_operations 定义，本文使用的设备文件结构是 led_ops。

类似地，字符设备卸载函数定义为

```
int unregister_chrdev(unsigned int major, const char * name);
```

4 LED 设备驱动程序的设计

4.1 LED 电路设计

开发板配套 S3C2410 处理器核心子板的 LED 电路接线如图 1 所示。其中 LED1、LED2、LED3 及 LED4 分别接 S3C2410 芯片的 I/O 口 GPF4、GPF5、GPF6 及 GPF7，这样就可以通过读写 GPF I/O 口来控制 LED 的状态。由图 1 可以看出，4 个 LED 为共阳极接法，若想让哪一个 LED 发光，则只要将对应的 GPF 口输出“ 0 ”即可。

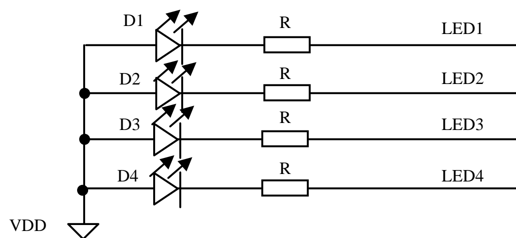

图 1 LED 电路设计

4.2 LED 驱动程序设计

LED 设备驱动代码如下：

```
#include <linux/module.h>
……
static int nLedMajor=0; //可自动分配主设备号
#define LED_DEV   "led"; //定义 LED 设备名称
#define GPF_MASK; (0xF<<4)
//定义 GPF4~7 分别对应 LED1~4
#define GPF_CTL_BASE   io_p2v ( 0x56000050);
//设置 GPF 的虚拟地址
#define LED_LOCK(u)   down (&u->lock);
#define LED_UNLOCK(u)   up (&u->lock)
```

设计 LED 驱动程序，首先定义一个 struct unit 的结构，这个结构作为一个 LED 控制器的抽象对象。在驱动程序中，对 LED 控制器的访问通过读写结构对象中的数据来实现，结构定义如下：

```
struct unit {
struct semaphore lock; //内核信号量，当多个用户程序同时访问一个 LED 控制器时，用 lock 进行同步
u32 *GPF_CON; //指向通用 IO 端口 F 的配置寄存器，初始化时将 GPF4、GPF5、GPF6 及 GPF7 都配置为输出口
u32 *GPF_DAT; //指向通用 IO 端口 F 的数据寄存器，与 LED 连接关系为：GPF4—LED1、GPF5—LED2、GPF6—LED3、GPF7—LED4
u32 *GPF_UP; //指向通用 IO 端口 F 的上拉电阻配置寄存器
u32 *f; //保存 GPF4~GPF7 的值};
static struct unit led_unit={
.GPF_CON   = ( u32 *) S3C2410_GPFCON;
.GPF_DAT   = ( u32 *) S3C2410_GPFDAT;
.GPF_UP   = ( u32 *) S3C2410_GPFUP;
.f= 0x00;
}; //初始化 4 个 LED，使其全部发光}
}
```

LED 设备驱动程序的 struct file_operation 结构声明如下所示，在此根据实际需要，做了适当的精简，只定义了与用户的 LED 应用程序里对设备文件操作的函数相对应的驱动函数。用户可根据自己的需要作相应的定义，使用该方法来提高驱动程序的可移植性。

```
static struct file_operations led_ops = {
    owner:   THIS_MODULE,
    read:    led_read,
    write:   led_write,
    open:    led_open,
    release:   led_release_f};
```

在用户程序打开设备时，LED 字符设备驱动程序实现的函数 led_open 把 LED 控制单元赋值给文件描述符的私有数据。MOD_INC_USE_COUNT 是内核提供的一个宏，它使模块的使用次数加 1，目的是防止当模块被使用的同时又被卸载，只有当模块使用计数为 0 时，才能被卸载。

```
static int led_open (struct inode *inode,struct file *file){
leds_event (led_stop)   //关闭内核对 LED 的操作
file->private_deta = &led_unit   //把私有数据设为符合 unit 结构的结构变量
MOD_INC_USE_COUNT   //使模块的使用次数加 1
return 0};
```

当用户关闭设备时，先使模块的使用计数减 1，再还原内核对 LED 的控制。

```
static int led_relaese (struct inode *inode,struct file *file){
MOD_DEC_USE_COUNT; //使模块的使用次数减 1
leds_event (led_start); //恢复内核对 LED 的操作
return 0}
```

读取 LED 状态时，先锁定 LED 控制单元，防止另一进程同一时刻改变 LED 状态。然后从控制单元获得状态值，并拷贝到用户缓冲区。操作完成后解锁，并返回读取字节数。其中 copy_to_user 函数实现将内核空间的数据 COPY 到用户空间里。

```
static ssize_t   led_read (struct file *file, char *buf, size_t count, loff_t *offset ){
u8 temp;
int ret;
struct unit *unit = (struct unit * )file->private_data;
if (count>1)
count = 1;
LED_LOCK(unit);   //上锁
temp = led_get_value (unit);   //读取 LED 状态，并赋值给局部变量 temp
ret = copy_to_user(buf, &temp, count) ? –EFAULT: count;
//把 temp 数据 copy 到 buf
LED_UNLOCK(unit); //开锁
return ret; }
```

写 LED 状态与读取 LED 状态的操作大同小异，其中 copy_from_user 函数实现将用户空间的数据 COPY 到内核空间里，程序如下：

```
static ssize_t  led_write (struct file *file, const char *buf, size_t count, loff_t *offset ){
u8 temp;
int ret;
char *temp;
struct unit *unit = (struct unit * )file->private_data;
if (count>1)
count = 1;
LED_LOCK(unit);  //上锁
ret = copy_from_user(&temp, buf, count) ? –EFAULT: count ;
//把 buf 数据写到 temp
if (ret)
led_set_value (unit,temp);
LED_UNLOCK(unit) ; //开锁
return  ret;}
```

在内核启动时，使用 module_init 宏指定驱动程序初始化函数。led_init_module 为设备模块初始化函数。

```
int _ _init led_init_module (void)
{
  int res;
  # ifdef  CONFIG_DEVFS_FS
res = devfs_register_chrdev (0,LED_DEV, &led_ops);
//调用字符设备驱动程序
if (res<0) {
printk("unable to get major for led device.\n");
return res;}
devfs_led_dir = devfs_mk_dir ( NULL,LED_DEV, NULL);
devfs_handle = devfs_register ( devfs_led_dir, "0",DEVFS_FL_DEFAULT, res, 0,
S_IFCHR | S_IRUSR | S_IWUSR, &led_ops,NULL);
//在/dev 目录下动态生成设备对应的文件
# else
res = register_chrdev (nLedMajor, LED_DEV, &led_ops);
# endif
# ifdef  CONFIG_PROC_FS
create_proc_read_entry ("driver/led",0,0,led_procNULL);
# end if
led_init (&led_unit);
return 0;}
module_init (led_init_module) //内核显式声明
```

使用 module_exit 宏指定驱动程序的卸载函数。模块卸载时需要注销设备，包括删除设备安装目录以及该设备的信息文件，注销设备注册时的主设备号和设备名称为：

```
void _ _exit led_cleanup(void){
int res;
res = unregister_chrdev (nLedMajor,LED_DEV);
if (res<0)
printk("unable to release major %d for led device.\n",nLedMajor);
# ifdef  CONFIG_DEVFS_FS
devfs_unregister ( devfs_handle);
devfs_unregister ( devfs_led_dir);
# endif
# ifdef  CONFIG_PROC_FS
remove_proc_entry ("driver/led", NULL);
# endif}
module_exit (led_cleanup)
```

5 结束语

本文以 LED 设备驱动程序的设计为例，阐述了 Linux 系统下，字符设备驱动程序的设计方法和关键技术。在本文所设计的驱动程序的基础上，再编制一个简单的测试程序，实现了对 4 个 LED 的读写操作。

参考文献

1 毛德操, 胡希明. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2001.

2 彭晓明, 王　强. Linux 核心源代码分析[M]. 北京: 人民邮电出版社, 2000.

3 Andrew S T, Albert S W. Operating Systems Designed and Implementation(2nd Edition)[M]. 北京: 清华大学出版社, 1997.

（上接第 274 页）

浏览器，它直接针对嵌入式 Linux 环境。通过包含开放源代码的核心部件，已经能够在占用少量 RAM 和 ROM 资源的情况下使用一个高品质的显示引擎。ViewML 界面不够友好的缺点也已经由 Century Software 公司做了改进，使之能够适应 PDA 的显示特点。但是 ViewML 还是存在一些缺点和不足，最突出的一点就是不支持文件的下载。因为我们在开发板上实现了 J2ME 应用平台，Java 程序的下载也是其中的任务之一，所以需要对 ViewML 进行修改，使之能够和 MIDP 合作实现 Java 程序的选择下载。

修改是在浏览器的应用层实现的，浏览器只是下载了 JAD 类型的 Java 程序描述文件，JAR 类型的 Java 程序则调用 MIDP 下载。过程如下：当 URL 指向 JAD 文件时，ViewML 浏览器将下载该 JAD 文件并保存为一个临时文件。然后分析 JAD 文件中的 Java 程序相关信息，并弹出提示框显示 Java 程序的大小等信息，并询问是否要下载，如果选择下载，程序会调用 MIDP 下载 Java 程序并启动 MIDP 图形界面，如果选择取消则直接回到浏览器。具体编程实现使用了 Linux 中复制进程的用法，当 MIDP 被启动之后，浏览器进入等待状态。修改后的浏览器在测试中使用情况良好，能够有效地配合 MIDP 进行 Java 程序的选择下载。

4 结论

无线上网是智能移动终端的一个必备功能，而提供网络连接和基本的人机界面则是无线上网的基础。本文在实践基础上，就上述问题在嵌入式系统的实现，提出了实用性的方案。实现了无线上网后，系统应用的主体——J2ME 平台才能发挥最大限度的作用。

参考文献

1 Haerr G. The Design and Evolution of the ViewML Internet Browser for Embedded Linux[EB/OL]. 2001. http://www.pixil.org/docs/viewml-overview.doc.

2 Haree G. Microwindows ArchitecturepEB/OL]. 2001. http://www.viewml.com.

3 朱奇波, 郑扣根, 潘云鹤. 一个基于新型嵌入式系统的浏览器的研究和实现[J]. 计算机应用研究, 2005, 19(5): 101.

4 孙先虎, 张曦煌. 基于 Linux 的嵌入式浏览器的特点和实现[J]. 计算机应用与软件, 2004, 15(21): 112.

5 周世杰, 秦志光. 文件传输协议分析及应用[J]. 计算机应用, 2001, 21(增刊).

6 刘峥嵘, 张智超, 许振山等. 嵌入式 Linux 应用开发详解[M]. 北京: 机械工业出版社, 2004.