

嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发 标准教程 (第2版)

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embedded Linux Application Development




光盘内容
本书源代码
本书配套PPT
嵌入式专家讲座视频

 **人民邮电出版社**
POSTS & TELECOM PRESS



第 11 章 嵌入式 Linux 设备驱动开发

本章目标

本书从第 6 章~第 10 章详细讲解了嵌入式 Linux 应用程序的开发，这些都是处于用户空间的内容。本章将进入到 Linux 的内核空间，初步介绍嵌入式 Linux 设备驱动的开发。驱动的开发流程相对于应用程序的开发是全新的，与读者以前的编程习惯完全不同，希望读者能尽快地熟悉现在环境。通过本章的学习，读者将会掌握以下内容。

- Linux 设备驱动的基本概念
- Linux 设备驱动程序的基本功能
- Linux 设备驱动的运作过程
- 常见设备驱动接口函数
- 掌握 LCD 设备驱动程序编写步骤
- 掌握键盘设备驱动程序编写步骤

11.1 设备驱动概述

11.1.1 设备驱动简介及驱动模块

操作系统是通过各种驱动程序来驾驭硬件设备的，它为用户屏蔽了各种各样的设备，驱动硬件是操作系统最基本的功能，并且提供统一的操作方式。设备驱动程序是内核的一部分，硬件驱动程序是操作系统最基本的组成部分，在 Linux 内核源程序中也占有 60% 以上。因此，熟悉驱动的编写是很重要的。

在第 2 章中已经提到过，Linux 内核中采用可加载的模块化设计 (LKMs, Loadable Kernel Modules)，一般情况下编译的 Linux 内核是支持可插入式模块的，也就是将最核心的代码编译在内核中，其他的代码可以编译到内核中，或者编译为内核的模块文件（在需要时动态加载）。

常见的驱动程序是作为内核模块动态加载的，比如声卡驱动和网卡驱动等，而 Linux 最基础的驱动，如 CPU、PCI 总线、TCP/IP 协议、APM（高级电源管理）、VFS 等驱动程序则直接编译在内核文件中。有时也把内核模块叫做驱动程序，只不过驱动的内容不一定是硬件罢了，比如 ext3 文件系统的驱动。因此，加载驱动就是加载内核模块。

这里，首先列举一些模块相关的命令。

n **lsmod** 列出当前系统中加载的模块，其中左边第一列是模块名，第二列是该模块大小，第三列则是使用该模块的对象数目。如下所示：

```
$ lsmod
Module                Size  Used by
Autofs                 12068  0 (autoclean) (unused)
eepro100              18128  1
iptables_nat          19252  0 (autoclean) (unused)
ip_conntrack          18540  1 (autoclean) [iptables_nat]
iptables_mangle        2272  0 (autoclean) (unused)
iptables_filter        2272  0 (autoclean) (unused)
ip_tables              11936  5 [iptables_nat iptables_mangle
iptables_filter]
usb-ohci               19328  0 (unused)
usbcore                54528  1 [usb-ohci]
ext3                   67728  2
jbd                    44480  2 [ext3]
aic7xxx                114704  3
sd_mod                 11584  3
scsi_mod               98512  2 [aic7xxx sd_mod]
```

n **rmmod** 是用于将当前模块卸载。

n **insmod** 和 **modprobe** 是用于加载当前模块，但 **insmod** 不会自动解决依存关系，

即如果要加载的模块引用了当前内核符号表中不存在的符号，则无法加载，也不会去查在其他尚未加载的模块中是否定义了该符号；`modprobe` 可以根据模块间依存关系以及 `/etc/modules.conf` 文件中的内容自动加载其他有依赖关系的模块。

11.1.2 设备分类

本书在前面也提到过，Linux 的一个重要特点就是将所有的设备都当做文件进行处理，这一类特殊文件就是设备文件，它们可以使用前面提到的文件、I/O 相关函数进行操作，这样就大大方便了对设备的处理。它通常在 `/dev` 下面存在一个对应的逻辑设备节点，这个节点以文件的形式存在。

Linux 系统的设备分为 3 类：字符设备、块设备和网络设备。

- n 字符设备通常指像普通文件或字节流一样，以字节为单位顺序读写的设备，如并口设备、虚拟控制台等。字符设备可以通过设备文件节点访问，它与普通文件之间的区别在于普通文件可以被随机访问（可以前后移动访问指针），而大多数字符设备只能提供顺序访问，因为对它们的访问不会被系统所缓存。但也有例外，例如帧缓存(`framebuffer`)是一个可以被随机访问的字符设备。
- n 块设备通常指一些需要以块为单位随机读写的设备，如 IDE 硬盘、SCSI 硬盘、光驱等。块设备也是通过文件节点来访问，它不仅可以提供随机访问，而且可以容纳文件系统（例如硬盘、闪存等）。Linux 可以使用户态程序像访问字符设备一样每次进行任意字节的操作，只是在内核态内部中的管理方式和内核提供的驱动接口上不同。

通过文件属性可以查看它们是哪种设备文件(字符设备文件或块设备文件)。

```
$ ls -l /dev
crw-rw---- 1 root uucp 4, 64 08-30 22:58 ttyS0 /*串口设备, c 表示字符设备
*/
brw-r----- 1 root floppy 2, 0 08-30 22:58 fd0/*软盘设备, b 表示块设备
*/
```

- n 网络设备通常是指通过网络能够与其他主机进行数据通信的设备，如网卡等。

内核和网络设备驱动程序之间的通信调用一套数据包处理函数，它们完全不同于内核和字符以及块设备驱动程序之间的通信（`read()`、`write()`等函数）。Linux 网络设备不是面向流的设备，因此不会将网络设备的名字（例如 `eth0`）映射到文件系统中去。

对这 3 种设备文件编写驱动程序时会有一定的区别，本书在后面会有相关内容的讲解。

11.1.3 设备号

设备号是一个数字，它是设备的标志。就如前面所述，一个设备文件（也就是设备节点）可以通过 `mknod` 命令来创建，其中指定了主设备号和次设备号。主设备号表明设备的类型（例如串口设备、SCSI 硬盘），与一个确定的驱动程序对应；次设备号

通常是用于标明不同的属性，例如不同的使用方法、不同的位置、不同的操作等，它标志着某个具体的物理设备。高字节为主设备号，底字节为次设备号。

例如，在系统中的块设备 IDE 硬盘的主设备号是 3，而多个 IDE 硬盘及其各个分区分别赋予次设备号 1、2、3...

```
$ ls -l /dev
```

```
crw-rw---- 1 root uucp 4, 64 08-30 22:58 ttyS0 /* 主设备号 4，此设备号
64 */
```

11.1.4 驱动层次结构

Linux 下的设备驱动程序是内核的一部分，运行在内核模式下，也就是说设备驱动程序为内核提供了一个 I/O 接口，用户使用这个接口实现对设备的操作。图 11.1 显示了典型的 Linux 输入/输出系统中各层次结构和功能。

Linux 设备驱动程序包含中断处理程序和设备服务子程序两部分。

设备服务子程序包含了所有与设备操作相关的处理代码。它从面向用户进程的设备文件系统中接受用户命令，并对设备控制器执行操作。这样，设备驱动程序屏蔽了设备的特殊性，使用户可以像对待文件一样操作设备。

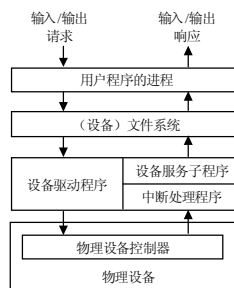


图 11.1 Linux 输入/输出层次结构和功能

设备控制器获得系统服务有两种方式：查询和中断。因为 Linux 的设备驱动程序是内核的一部分，在设备查询期间系统不能运行其他代码，查询方式的工作效率比较低，所以只有少数设备如软盘驱动程序采取这种方式，大多设备以中断方式向设备驱动程序发出输入/输出请求。

11.1.5 设备驱动程序与外界的接口

每种类型的驱动程序，不管是字符设备还是块设备都为内核提供相同的调用接口，因此内核能以相同的方式处理不同的设备。Linux 为每种不同类型的设备驱动程序维护相应的数据结构，以便定义统一的接口并实现驱动程序的可装载性和动态性。

Linux 设备驱动程序与外界的接口可以分为如下 3 个部分。

- n 驱动程序与操作系统内核的接口：这是通过数据结构 `file_operations`（在本书后面会有详细介绍）来完成的。
- n 驱动程序与系统引导的接口：这部分利用驱动程序对设备进行初始化。
- n 驱动程序与设备的接口：这部分描述了驱动程序如何与设备进行交互，这与具体设备密切相关。

它们之间的相互关系如图 11.2 所示。

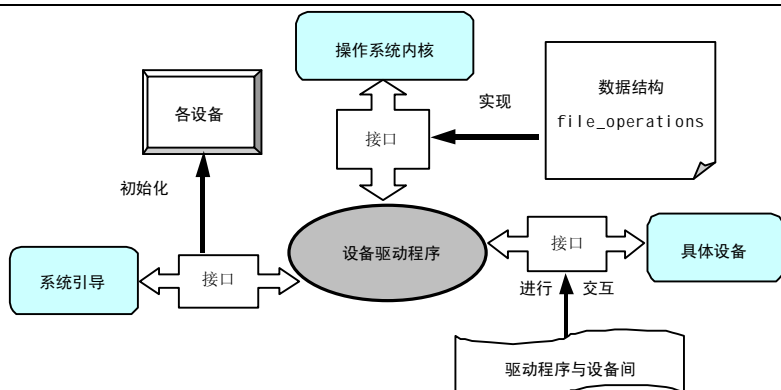


图 11.2 设备驱动程序与外界接口

11.1.6 设备驱动程序的特点

综上所述，Linux 中的设备驱动程序有如下特点。

(1) 内核代码：设备驱动程序是内核的一部分，如果驱动程序出错，则可能导致系统崩溃。

(2) 内核接口：设备驱动程序必须为内核或者其子系统提供一个标准接口。比如，一个终端驱动程序必须为内核提供一个文件 I/O 接口；一个 SCSI 设备驱动程序应该为 SCSI 子系统提供一个 SCSI 设备接口，同时 SCSI 子系统也必须为内核提供文件的 I/O 接口及缓冲区。

(3) 内核机制和服务：设备驱动程序使用一些标准的内核服务，如内存分配等。

(4) 可装载：大多数的 Linux 操作系统设备驱动程序都可以在需要时装载进内核，在不需要时从内核中卸载。

(5) 可设置：Linux 操作系统设备驱动程序可以集成为内核的一部分，并可以根据需要把其中的某一部分集成到内核中，这只需要在系统编译时进行相应的设置即可。

(6) 动态性：在系统启动且各个设备驱动程序初始化后，驱动程序将维护其控制的设备。如果该设备驱动程序控制的设备不存在也不影响系统的运行，那么此时的设备驱动程序只是多占用了一点系统内存罢了。

11.2 字符设备驱动编程

1. 字符设备驱动编写流程

设备驱动程序可以使用模块的方式动态加载到内核中去。加载模块的方式与以往的应用程序开发有很大的不同。以往在开发应用程序时都有一个 `main()` 函数作为程序的入口点，而在驱动开发时却没有 `main()` 函数，模块在调用 `insmod` 命令时被加载，此时的入口点是 `init_module()` 函数，通常在该函数中完成设备的注册。同样，模块在调用 `rmmmod` 命令时被卸载，此时的入口点是 `cleanup_module()` 函数，在该函数中完成设备的卸载。在设备完成注册加载之后，用户的应用程序

就可以对该设备进行一定的操作，如 `open()`、`read()`、`write()` 等，而驱动程序就是用于实现这些操作，在用户应用程序调用相应入口函数时执行相关的操作，`init_module()` 入口点函数则不需要完成其他如 `read()`、`write()` 之类功能。

上述函数之间的关系如图 11.3 所示。

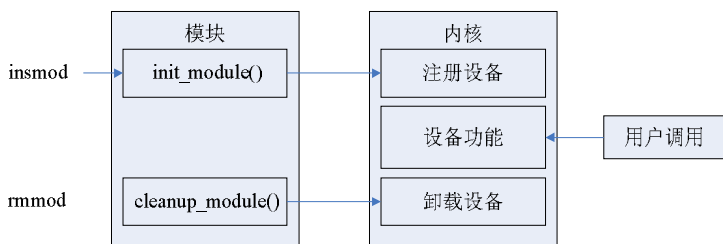


图 11.3 设备驱动程序流程图

2. 重要数据结构

用户应用程序调用设备的一些功能是在设备驱动程序中定义的，也就是设备驱动程序的入口点，它是在一个在 `<linux/fs.h>` 中定义的 `struct file_operations` 结构，这是一个内核结构，不会出现在用户空间的程序中，它定义了常见文件 I/O 函数的入口，如下所示：

```
struct file_operations
{
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *filp,
                     char *buff, size_t count, loff_t *offp);
    ssize_t (*write) (struct file *filp,
                     const char *buff, size_t count, loff_t
*offp);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *,
                  struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *);
    int (*fasync) (int, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
    int (*lock) (struct file *, int, struct file_lock *);
};
```

这里定义的很多函数是否跟第 6 章中的文件 I/O 系统调用类似？其实当时的系统调用函数通过内核，最终调用对应的 struct file_operations 结构的接口函数（例如，open()文件操作是通过调用对应文件的 file_operations 结构的 open 函数接口而被实现）。当然，每个设备的驱动程序不一定要实现其中所有的函数操作，若不需要定义实现时，则只需将其设为 NULL 即可。

struct inode 结构提供了关于设备文件/dev/driver（假设此设备名为 driver）的信息，struct file 结构提供关于被打开的文件信息，主要用于与文件系统对应的设备驱动程序使用。struct file 结构较为重要，这里列出了它的定义：

```
struct file
{
    mode_t f_mode; /*标识文件是否可读或可写，FMODE_READ 或 FMODE_WRITE*/
    dev_t f_rdev; /* 用于/dev/tty */
    off_t f_pos; /* 当前文件位移 */
    unsigned short f_flags; /* 文件标志，如 O_RDONLY、O_NONBLOCK 和 O_SYNC
*/

    unsigned short f_count; /* 打开的文件数目 */
    unsigned short f_reada;
    struct inode *f_inode; /*指向 inode 的结构指针 */
    struct file_operations *f_op; /* 文件索引指针 */
};
```

3. 设备驱动程序主要组成

(1) 早期版本的字符设备注册。

早期版本的设备注册使用函数 register_chrdev(), 调用该函数后就可以向系统申请主设备号，如果 register_chrdev()操作成功，设备名就会出现在/proc/devices 文件里。在关闭设备时，通常需要解除原先的设备注册，此时可使用函数 unregister_chrdev(), 此后该设备就会从/proc/devices 里消失。其中主设备号和次设备号不能大于 255。

当前不少的字符设备驱动代码仍然使用这些早期版本的函数接口，但在未来内核的代码中，将不会出现这种编程接口机制。因此应该尽量使用后面讲述的编程机制。

register_chrdev()函数格式如表 11.1 所示。

表 11.1 register_chrdev() 函数语法要点

所需头文件	#include <linux/fs.h>
函数原型	int register_chrdev(unsigned int major, const char *name,struct file_operations *fops)
函数传入值	major: 设备驱动程序向系统申请的主设备号，如果为 0 则系统为此驱动程序动态配一个主设备号
	name: 设备名
	fops: 对各个调用的入口点

函数返回值	成功：0（只限于两种注册函数）
	出错：-1（只限于两种注册函数）

（3）最新版本的字符设备注册。

在获得了系统分配的设备号之后，通过注册设备才能实现设备号和驱动程序之间的关联。这里讲解 2.6 内核中的字符设备的注册和注销过程。

在 Linux 内核中使用 struct cdev 结构来描述字符设备，我们在驱动程序中必须将已分配到的设备号以及设备操作接口（即为 struct file_operations 结构）赋予 struct cdev 结构变量。首先使用 cdev_alloc()函数向系统申请分配 struct cdev 结构，再用 cdev_init()函数初始化已分配到的结构并与 file_operations 结构关联起来。最后调用 cdev_add()函数将设备号与 struct cdev 结构进行关联并向内核正式报告新设备的注册，这样新设备可以被用起来了。

如果要从系统中删除一个设备，则要调用 cdev_del()函数。具体函数格式如表 11.4 所示。

表 11.4 最新版本的字符设备注册

所需头文件	#include <linux/cdev.h>
函数原型	struct cdev *cdev_alloc(void) void cdev_init(struct cdev *cdev, struct file_operations *fops) int cdev_add (struct cdev *cdev, dev_t num, unsigned int count) void cdev_del(struct cdev *dev)
函数传入值	cdev: 需要初始化/注册/删除的 struct cdev 结构 fops: 该字符设备的 file_operations 结构 num: 系统给该设备分配的第一个设备号 count: 该设备对应的设备号数量
函数返回值	成功: cdev_alloc: 返回分配到的 struct cdev 结构指针 cdev_add: 返回 0
	出错: cdev_alloc: 返回 NULL cdev_add: 返回 -1

2.6 内核仍然保留早期版本的 register_chrdev()等字符设备相关函数，其实从内核代码中可以发现，在 register_chrdev()函数的实现中用到 cdev_alloc()和 cdev_add()函数，而在 unregister_chrdev()函数的实现中调用 cdev_del()函数。因此很多代码仍然使用早期版本接口，但这种机制将来会从内核中消失。

前面已经提到字符设备的实际操作在 struct file_operations 结构的一组函数中定义，并在驱动程序中需要与字符设备结构关联起来。下面讨论 struct file_operations 结构中最主要的成员函数和它们的用法。

（4）打开设备。

打开设备的函数接口是 open，根据设备的不同，open 函数接口完成的功能也有

所不同，但通常情况下在 `open` 函数接口中要完成如下工作。


- n 递增计数器，检查错误。
- n 如果未初始化，则进行初始化。
- n 识别次设备号，如果必要，更新 `f_op` 指针。
- n 分配并填写被置于 `filp->private_data` 的数据结构。

其中递增计数器是用于设备计数的。由于设备在使用时通常会打开多次，也可以由不同的进程所使用，所以若有一进程想要删除该设备，则必须保证其他设备没有使用该设备。因此使用计数器就可以很好地完成这项功能。

这里，实现计数器操作的是在 2.6 内核早期版本的 `<linux/module.h>` 中定义的 3 个宏，它们在最新版本里早就消失了，在下面列出只是为了帮读者理解老版本中的驱动代码。

- n `MOD_INC_USE_COUNT`: 计数器加 1。
- n `MOD_DEC_USE_COUNT`: 计数器减 1。
- n `MOD_IN_USE`: 计数器非零时返回真。

另外，当有多个物理设备时，就需要识别次设备号来对各个不同的设备进行不同的操作，在有些驱动程序中并不需要用到。

 **注意** 虽然这是对设备文件执行的第一个操作，但却不是驱动程序一定要声明的操作。若这个函数的入口为 `NULL`，那么设备的打开操作将永远成功，但系统不会通知驱动程序。

(5) 释放设备。

释放设备的函数接口是 `release()`。要注意释放设备和关闭设备是完全不同的。当一个进程释放设备时，其他进程还能继续使用该设备，只是该进程暂时停止对该设备的使用；而当一个进程关闭设备时，其他进程必须重新打开此设备才能使用它。

释放设备时要完成的工作如下。

- n 递减计数器 `MOD_DEC_USE_COUNT`（最新版本已经不再使用）。
- n 释放打开设备时系统所分配的内存空间（包括 `filp->private_data` 指向的内存空间）。
- n 在最后一次释放设备操作时关闭设备。

(6) 读写设备。

读写设备的主要任务就是把内核空间的数据复制到用户空间，或者从用户空间复制到内核空间，也就是将内核空间缓冲区里的数据复制到用户空间的缓冲区中或者相反。这里首先解释一个 `read()` 和 `write()` 函数的入口函数，如表 11.5 所示。

表 11.5 read、write 函数接口语法要点

所需头文件	<code>#include <linux/fs.h></code>
函数原型	<code>ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp)</code> <code>ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t *offp)</code>
函数传入值	<code>filp</code> : 文件指针
	<code>buff</code> : 指向用户缓冲区

函数返回值	count: 传入的数据长度
	offp: 用户在文件中的位置
	成功: 写入的数据长度

虽然这个过程看起来很简单，但是内核空间地址和应用空间地址是有很大区别的，其中一个区别是用户空间的内存是可以被换出的，因此可能会出现页面失效等情况。所以不能使用诸如 `memcpy()` 之类的函数来完成这样的操作。在这里要使用 `copy_to_user()` 或 `copy_from_user()` 等函数，它们是用来实现用户空间和内核空间的数据交换的。

`copy_to_user()` 和 `copy_from_user()` 的格式如表 11.6 所示。

表 11.6 `copy_to_user()/copy_from_user()` 函数语法要点

所需头文件	#include <asm/uaccess.h>
函数原型	unsigned long copy_to_user(void *to, const void *from, unsigned long count) unsigned long copy_from_user(void *to, const void *from, unsigned long count)
函数传入值	to: 数据目的缓冲区
	from: 数据源缓冲区
	count: 数据长度
函数返回值	成功: 写入的数据长度 失败: -EFAULT

要注意，这两个函数不仅实现了用户空间和内核空间的数据转换，而且还会检查用户空间指针的有效性。如果指针无效，那么就不进行复制。

(7) `ioctl`。

大部分设备除了读写操作，还需要硬件配置和控制（例如，设置串口设备的波特率）等很多其他操作。在字符设备驱动中 `ioctl` 函数接口给用户提供对设备的非读写操作机制。

`ioctl` 函数接口的具体格式如表 11.7 所示。

表 11.7 `ioctl` 函数接口语法要点

所需头文件	#include <linux/fs.h>
函数原型	int(*ioctl)(struct inode* inode, struct file* filp, unsigned int cmd, unsigned long arg)
函数传入值	inode: 文件的内核内部结构指针
	filp: 被打开的文件描述符
	cmd: 命令类型
	arg: 命令相关参数

下面列出其他在驱动程序中常用的内核函数。

(8) 获取内存。

在应用程序中获取内存通常使用函数 `malloc()`，但在设备驱动程序中动态开辟内

《嵌入式 Linux 应用程序开发标准教程》——第 11 章、嵌入式 Linux 设备驱动开发

存可以以字节或页面为单位。其中，以字节为单位分配内存的函数有 `kmalloc()`，注意的是，`kmalloc()`函数返回的是物理地址，而 `malloc()`等返回的是线性虚拟地址，因此在驱动程序中不能使用 `malloc()`函数。与 `malloc()`不同，`kmalloc()`申请空间有大小限制。长度是 2 的整次方，并且不会对所获取的内存空间清零。

- 以页为单位分配内存的函数如下所示。
- n `get_zeroed_page()`: 获得一个已清零页面。
 - n `get_free_page()`: 获得一个或几个连续页面。
 - n `get_dma_pages()`: 获得用于 DMA 传输的页面。
- 与之相对应的释放内存用也有 `kfree()`或 `free_page` 函数族。
- 表 11.8 给出了 `kmalloc()`函数的语法格式。

表 11.8 `kmalloc()` 函数语法要点

所需头文件	#include <linux/malloc.h>	
函数原型	void *kmalloc(unsigned int len,int flags)	
函数传入值	len:	希望申请的字节数
	flags	GFP_KERNEL: 内核内存的通常分配方法，可能引起睡眠
		GFP_BUFFER: 用于管理缓冲区高速缓存
		GFP_ATOMIC: 为中断处理程序或其他运行于进程上下文之外的代码分配内存，且不会引起睡眠
		GFP_USER: 用户分配内存，可能引起睡眠
		GFP_HIGHUSER: 优先高端内存分配
		__GFP_DMA: DMA 数据传输请求内存
函数返回值	成功: 写入的数据长度	
	失败: -EFAULT	

表 11.9 给出了 `kfree()`函数的语法格式。

表 11.9 `kfree()` 函数语法要点

所需头文件	#include <linux/malloc.h>	
函数原型	void kfree(void * obj)	
函数传入值	obj: 要释放的内存指针	
函数返回值	成功: 写入的数据长度	
	失败: -EFAULT	

表 11.10 给出了以页为单位的分配函数 `get_free_page` 类函数的语法格式。

表 11.10 **get_free_page** 类函数语法要点

所需头文件	#include <linux/malloc.h>
函数原型	unsigned long get_zeroed_page(int flags) unsigned long __get_free_page(int flags) unsigned long __get_free_page(int flags,unsigned long order) unsigned long __get_dma_page(int flags,unsigned long order)
函数传入值	flags: 同 kmalloc()
	order: 要请求的页面数, 以 2 为底的对数
函数返回值	成功: 返回指向新分配的页面的指针 失败: -EFAULT

表 11.11 给出了基于页的内存释放函数 free_page 族函数的语法格式。

表 11.11 **free_page** 类函数语法要点

所需头文件	#include <linux/malloc.h>
函数原型	unsigned long free_page(unsigned long addr) unsigned long free_pages(unsigned long addr, unsigned long order)
函数传入值	addr: 要释放的内存起始地址
	order: 要请求的页面数, 以 2 为底的对数
函数返回值	成功: 写入的数据长度 失败: -EFAULT

(9) 打印信息。

就如同在编写用户空间的应用程序, 打印信息有时是很好的调试手段, 也是在代码中很常用的组成部分。但是与用户空间不同, 在内核空间要用函数 printk()而不能用平常的函数 printf()。printk()和 printf()很类似, 都可以按照一定的格式打印消息, 所不同的是, printk()还可以定义打印消息的优先级。

表 11.12 给出了 printk()函数的语法格式。

表 11.12 **printk** 类函数语法要点

所需头文件	#include <linux/kernel>		
函数原型	int printk(const char * fmt, ...)		
函数传入值	fmt: 日志级别	KERN_EMERG:	紧急时间消息
		KERN_ALERT:	需要立即采取动作的情况
		KERN_CRIT:	临界状态，通常涉及严重的硬件或软件操作失败
		KERN_ERR:	错误报告
		KERN_WARNING:	对可能出现的问题提出警告
		KERN_NOTICE:	有必要进行提示的正常情况
		KERN_INFO:	提示性信息
		KERN_DEBUG:	调试信息
	…: 与 printf()相同		
函数返回值	成功：0 失败：-1		

这些不同优先级的信息输出到系统日志文件（例如：“/var/log/messages”），有时

《嵌入式 Linux 应用程序开发标准教程》——第 11 章、嵌入式 Linux 设备驱动开发

也可以输出到虚拟控制台上。其中，对输出给控制台的信息有一个特定的优先级 console_loglevel。只有打印信息的优先级小于这个整数值，信息才能被输出到虚拟控制台上，否则，信息仅仅被写入到系统日志文件中。若不加任何优先级选项，则消息默认输出到系统日志文件中。

 **注意** 要开启 klogd 和 syslogd 服务，消息才能正常输出。

4. proc 文件系统

/proc 文件系统是一个伪文件系统，它是一种内核和内核模块用来向进程发送信息的机制。这个伪文件系统让用户可以和内核内部数据结构进行交互，获取有关系统和进程的有用信息，在运行时通过改变内核参数来改变设置。与其他文件系统不同，/proc 存在于内存之中而不是在硬盘上。读者可以通过“ls”查看/proc 文件系统的内容。

表 11.13 列出了/proc 文件系统的主要目录内容。

表 11.13 /proc 文件系统主要目录内容

目 录 名 称	目 录 内 容	目 录 名 称	目 录 内 容
apm	高级电源管理信息	locks	内核锁
cmdline	内核命令行	meminfo	内存信息
cpuinfo	CPU 相关信息	misc	杂项
devices	设备信息（块设备/字符设备）	modules	加载模块列表
dma	使用的 DMA 通道信息	mounts	加载的文件系统
filesystems	支持的文件系统信息	partitions	系统识别的分区表
interrupts	中断的使用信息	rtc	实时时钟
ioports	I/O 端口的使用信息	stat	全面统计状态表
kcore	内核映像	swaps	对换空间的利用情况
kmsg	内核消息	version	内核版本
ksyms	内核符号表	uptime	系统正常运行时间
loadavg	负载均衡

除此之外，还有一些是以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口。进程目录的结构如表 11.14 所示。

表 11.14 /proc 中进程目录结构

目 录 名 称	目 录 内 容	目 录 名 称	目 录 内 容
cmdline	命令行参数	cwd	当前工作目录的链接
environ	环境变量值	exe	指向该进程的执行命令文件
fd	一个包含所有文件描述符的目录	maps	内存映像
mem	进程的内存被利用情况	statm	进程内存状态信息
stat	进程状态	root	链接此进程的 root 目录
status	进程当前状态，以可读的方式显示出来

用户可以使用 cat 命令来查看其中的内容。

可以看到，/proc 文件系统体现了内核及进程运行的内容，在加载模块成功后，读者可以通过查看/proc/device 文件获得相关设备的主设备号。

11.3 GPIO 驱动程序实例

11.3.1 GPIO 工作原理

FS2410 开发板的 S3C2410 处理器具有 117 个多功能通用 I/O（GPIO）端口管脚，包括 GPIO 8 个端口组，分别为 GPA（23 个输出端口）、GPB（11 个输入/输出端口）、GPC（16 个输入/输出端口）、GPD（16 个输入/输出端口）、GPE（16 个输入/输出端口）、GPF（8 个输入/输出端口）、GPH（11 个输入/输出端口）。根据各种系统设计的需求，通过软件方法可以将这些端口配置成具有相应功能（例如：外部中断或数据总线）的端口。

为了控制这些端口，S3C2410 处理器为每个端口组分别提供几种相应的控制寄存器。其中最常用的有端口配置寄存器（GPACON ~ GPHCON）和端口数据寄存器（GPADAT ~ GPHDAT）。因为大部分 I/O 管脚可以提供多种功能，通过配置寄存器（PnCON）设定每个管脚用于何种目的。数据寄存器的每位将对应于某个管脚上的输入或输出。所以通过对数据寄存器（PnDAT）的位读写，可以进行对每个端口的输入或输出。

在此主要以发光二极管（LED）和蜂鸣器为例，讨论 GPIO 设备的驱动程序。它们的硬件驱动电路的原理图如图 11.4 所示。

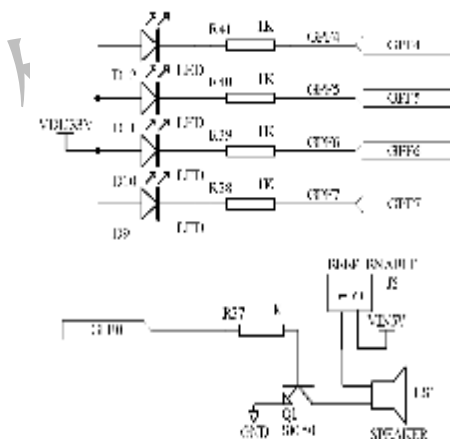


图 11.4 LED（左）和蜂鸣器（右）的驱动电路原理图

在图 11.4 中，可知使用 S3C2410 处理器的通用 I/O 口 GPF4、GPF5、GPF6 和 GPF7 分别直接驱动 LED D12、D11、D10 以及 D9，而使用 GPB0 端口驱动蜂鸣器。4 个 LED 分别在对应端口（GPF4~GPF7）为低电平时发亮，而蜂鸣器在 GPB0 为高电平时发声。这 5 个端口的数据流方向均为输出。

在表 11.15 中，详细描述了 GPF 的主要控制寄存器。GPB 的相关寄存器的描述与此类似，具体可以参考 S3C2410 处理器数据手册。

表 11.15 GPF 端口（GPF0-GPF7）的主要控制寄存器

寄存器	地址	R/W	功能	初始值
GPFCON	0x56000050	R/W	配置 GPF 端口组	0x0
GPFDAT	0x56000054	R/W	GPF 端口的数据寄存器	未定义
GPFUP	0x56000058	R/W	GPF 端口的取消上拉寄存器	0x0
GPFCON	位	描述		
GPF7	[15:14]	00 = 输入 01 = 输出 10 = EINT7 11 = 保留		
GPF6	[13:12]	00 = 输入 01 = 输出 10 = EINT6 11 = 保留		
GPF5	[11:10]	00 = 输入 01 = 输出 10 = EINT5 11 = 保留		
GPF4	[9:8]	00 = 输入 01 = 输出 10 = EINT4 11 = 保留		
GPF3	[7:6]	00 = 输入 01 = 输出 10 = EINT3 11 = 保留		
GPF2	[5:4]	00 = 输入 01 = 输出 10 = EINT2 11 = 保留		
GPF1	[3:2]	00 = 输入 01 = 输出 10 = EINT1 11 = 保留		
GPF0	[1:0]	00 = 输入 01 = 输出 10 = EINT0 11 = 保留		
GPFDAT	位	描述		
GPF[7:0]	[7:0]	每位对应于相应的端口，若端口用于输入，则可以通过相应的位读取数据；若端口用于输出，则可以通过相应的位输出数据；若端口用于其他功能，则其值无法确定。		
GPFUP	位	描述		
GPF[7:0]	[7:0]	0：向相应端口管脚赋予上拉(pull-up)功能 1：取消上拉功能		

为了驱动 LED 和蜂鸣器，首先通过端口配置寄存器将 5 个相应寄存器配置为输出模式。然后通过对端口数据寄存器的写操作，实现对每个 GPIO 设备的控制（发亮或发声）。在下一个小节中介绍的驱动程序中，s3c2410_gpio_cfgpin() 函数和 s3c2410_gpio_pullup() 函数将进行对某个端口的配置，而 s3c2410_gpio_setpin() 函数实现向数据寄存器的某个端口的输出。

11.3.2 GPIO 驱动程序

GPIO 驱动程序代码如下所示：

```
/* gpio_drv.h */
#ifndef      FS2410_GPIO_SET_H
#define      FS2410_GPIO_SET_H
#include     <linux/ioctl.h>
#define      GPIO_DEVICE_NAME      "gpio"
#define      GPIO_DEVICE_FILENAME  "/dev/gpio"
```

```

#define LED_NUM 4
#define GPIO_IOCTL_MAGIC 'G'
#define LED_D09_SWT _IOW(GPIO_IOCTL_MAGIC, 0, unsigned
int)
#define LED_D10_SWT _IOW(GPIO_IOCTL_MAGIC, 1, unsigned
int)
#define LED_D11_SWT _IOW(GPIO_IOCTL_MAGIC, 2, unsigned
int)
#define LED_D12_SWT _IOW(GPIO_IOCTL_MAGIC, 3, unsigned
int)
#define BEEP_SWT _IOW(GPIO_IOCTL_MAGIC, 4, unsigned
int)
#define LED_SWT_ON 0
#define LED_SWT_OFF 1
#define BEEP_SWT_ON 1
#define BEEP_SWT_OFF 0
#endif /* FS2410_GPIO_SET_H */

/* gpio_drv.c */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.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/mm.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/arch-s3c2410/regs-gpio.h>
#include "gpio_drv.h"

static int major = 0; /* 采用字符设备号的动态分配 */
module_param(major, int, 0); /* 以参数的方式可以指定设备的主设备号 */

```

```
void s3c2410_gpio_cfgpin(unsigned int pin, unsigned int function)
{ /* 对某个管脚进行配置（输入/输出/其他功能）*/
    unsigned long base = S3C2410_GPIO_BASE(pin); /* 获得端口的组基地址
*/

    unsigned long shift = 1;
    unsigned long mask = 0x03; /* 通常用配置寄存器的两位表示一个端口*/
    unsigned long con;
    unsigned long flags;

    if (pin < S3C2410_GPIO_BANKB)
    {
        shift = 0;
        mask = 0x01; /* 在 GPA 端口中用配置寄存器的一位表示一个端口*/
    }
    mask <= (S3C2410_GPIO_OFFSET(pin) << shift);
    local_irq_save(flags); /* 保存现场，保证下面一段是原子操作 */
    con = __raw_readl(base + 0x00);
    con &= ~mask;
    con |= function;
    __raw_writel(con, base + 0x00); /* 向配置寄存器写入新配置数据 */
    local_irq_restore(flags); /* 恢复现场 */
}

void s3c2410_gpio_pullup(unsigned int pin, unsigned int to)
{ /* 配置上拉功能 */
    unsigned long base = S3C2410_GPIO_BASE(pin); /* 获得端口的组基地址*/
    unsigned long offs = S3C2410_GPIO_OFFSET(pin); /* 获得端口的组内偏移地址
*/

    unsigned long flags;
    unsigned long up;

    if (pin < S3C2410_GPIO_BANKB)
    {
        return;
    }

    local_irq_save(flags);
    up = __raw_readl(base + 0x08);
    up &= ~(1 << offs);
```

```

up |= to << offs;
__raw_writel(up, base + 0x08); /* 向上拉功能寄存器写入新配置数据*/
local_irq_restore(flags);
}

```

```

void s3c2410_gpio_setpin(unsigned int pin, unsigned int to)
{ /* 向某个管脚进行输出 */
    unsigned long base = S3C2410_GPIO_BASE(pin);
    unsigned long offs = S3C2410_GPIO_OFFSET(pin);
    unsigned long flags;
    unsigned long dat;

    local_irq_save(flags);
    dat = __raw_readl(base + 0x04);
    dat &= ~(1 << offs);
    dat |= to << offs;
    __raw_writel(dat, base + 0x04); /* 向数据寄存器写入新数据*/
    local_irq_restore(flags);
}

```

```

int gpio_open (struct inode *inode, struct file *filp)
{ /* open 操作函数: 进行寄存器配置*/
    s3c2410_gpio_pullup(S3C2410_GPB0, 1); /* BEEP*/
    s3c2410_gpio_pullup(S3C2410_GPF4, 1); /* LED D12 */
    s3c2410_gpio_pullup(S3C2410_GPF5, 1); /* LED D11 */
    s3c2410_gpio_pullup(S3C2410_GPF6, 1); /* LED D10 */
    s3c2410_gpio_pullup(S3C2410_GPF7, 1); /* LED D9 */
    s3c2410_gpio_cfgpin(S3C2410_GPB0, S3C2410_GPB0_OUTP);
    s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF4_OUTP);
    s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF5_OUTP);
    s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF6_OUTP);
    s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF7_OUTP);
    return 0;
}

ssize_t gpio_read(struct file *file, char __user *buff,
                  size_t count, loff_t
*offp)
{ /* read 操作函数: 没有实际功能*/
    return 0;
}

```




```
ssize_t gpio_write(struct file *file, const char __user *buff,
                    size_t count, loff_t
*offp)
{ /* write 操作函数: 没有实际功能*/
    return 0;
}

int switch_gpio(unsigned int pin, unsigned int swt)
{ /* 向 5 个端口中的一个输出 ON/OFF 值 */
    if (!(pin <= S3C2410_GPF7) && (pin >= S3C2410_GPF4))
        && (pin != S3C2410_GPB0))
    {
        printk("Unsupported pin");
        return 1;
    }
    s3c2410_gpio_setpin(pin, swt);
    return 0;
}

static int gpio_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg)
{ /* ioctl 函数接口: 主要接口的实现。对 5 个 GPIO 设备进行控制 (发亮或发声) */
    unsigned int swt = (unsigned int)arg;
    switch (cmd)
    {
        case LED_D09_SWT:
        {
            switch_gpio(S3C2410_GPF7, swt);
        }
        break;

        case LED_D10_SWT:
        {
            switch_gpio(S3C2410_GPF6, swt);
        }
        break;

        case LED_D11_SWT:
        {
            switch_gpio(S3C2410_GPF5, swt);
        }
    }
}
```

```
    }
    break;
    case LED_D12_SWT:
    {
        switch_gpio(S3C2410_GPF4, swt);
    }
    break;

    case BEEP_SWT:
    {
        switch_gpio(S3C2410_GPB0, swt);
        break;
    }

    default:
    {
        printk("Unsupported command\n");
        break;
    }
}
return 0;
}

static int gpio_release(struct inode *node, struct file *file)
{ /* release 操作函数，熄灭所有灯和关闭蜂鸣器 */
    switch_gpio(S3C2410_GPB0, BEEP_SWT_OFF);
    switch_gpio(S3C2410_GPF4, LED_SWT_OFF);
    switch_gpio(S3C2410_GPF5, LED_SWT_OFF);
    switch_gpio(S3C2410_GPF6, LED_SWT_OFF);
    switch_gpio(S3C2410_GPF7, LED_SWT_OFF);
    return 0;
}

static void gpio_setup_cdev(struct cdev *dev, int minor,
    struct file_operations *fops)
{ /* 字符设备的创建和注册 */
    int err, devno = MKDEV(major, minor);
    cdev_init(dev, fops);
    dev->owner = THIS_MODULE;
    dev->ops = fops;
```



```
err = cdev_add (dev, devno, 1);
if (err)
{
    printk (KERN_NOTICE "Error %d adding gpio %d", err, minor);
}
}

static struct file_operations gpio_fops =
{ /* gpio 设备的 file_operations 结构定义 */
    .owner    = THIS_MODULE,
    .open     = gpio_open,      /* 进行初始化配置*/
    .release  = gpio_release,   /* 关闭设备*/
    .read     = gpio_read,
    .write    = gpio_write,
    .ioctl    = gpio_ioctl,     /* 实现主要控制功能*/
};

static struct cdev gpio_devs;
static int gpio_init(void)
{
    int result;
    dev_t dev = MKDEV(major, 0);

    if (major)
    { /* 设备号的动态分配 */
        result = register_chrdev_region(dev, 1, GPIO_DEVICE_NAME);
    }
    else
    { /* 设备号的动态分配 */
        result = alloc_chrdev_region(&dev, 0, 1, GPIO_DEVICE_NAME);
        major = MAJOR(dev);
    }
    if (result < 0)
    {
        printk(KERN_WARNING "Gpio: unable to get major %d\n", major);
        return result;
    }
    gpio_setup_cdev(&gpio_devs, 0, &gpio_fops);
    printk("The major of the gpio device is %d\n", major);
    return 0;
}
```

```

}

static void gpio_cleanup(void)
{
    cdev_del(&gpio_devs); /* 字符设备的注销 */
    unregister_chrdev_region(MKDEV(major, 0), 1); /* 设备号的注销 */
    printk("Gpio device uninstalled\n");
}

module_init(gpio_init);
module_exit(gpio_cleanup);
MODULE_AUTHOR("David");
MODULE_LICENSE("Dual BSD/GPL");

```

下面列出 GPIO 驱动程序的测试用例：

```

/* gpio_test.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "gpio_drv.h"

int led_timer(int dev_fd, int led_no, unsigned int time)
{ /*指定 LED 发亮一段时间之后熄灭它*/
    led_no %= 4;
    ioctl(dev_fd, LED_D09_SWT + led_no, LED_SWT_ON); /* 发亮*/
    sleep(time);
    ioctl(dev_fd, LED_D09_SWT + led_no, LED_SWT_OFF); /* 熄灭 */
}

int beep_timer(int dev_fd, unsigned int time)
{ /* 开蜂鸣器一段时间之后关闭*/
    ioctl(dev_fd, BEEP_SWT, BEEP_SWT_ON); /* 发声*/
    sleep(time);
    ioctl(dev_fd, BEEP_SWT, BEEP_SWT_OFF); /* 关闭 */
}

```

```
int main()
{
    int i = 0;
    int dev_fd;

    /* 打开 gpio 设备 */
    dev_fd = open(GPIO_DEVICE_FILENAME, O_RDWR | O_NONBLOCK);
    if ( dev_fd == -1 )
    {
        printf("Cann't open gpio device file\n");
        exit(1);
    }

    while(1)
    {
        i = (i + 1) % 4;
        led_timer(dev_fd, i, 1);
        beep_timer(dev_fd, 1);
    }
    close(dev_fd);
    return 0;
}
```

具体运行过程如下所示。首先编译并加载驱动程序：

```
$ make clean;make /* 驱动程序的编译*/
$ insmod gpio_drv.ko /* 加载 gpio 驱动 */
$ cat /proc/devices /* 通过这个命令可以查到 gpio 设备的主设备号 */
$ mknod /dev/gpio c 252 0 /* 假设主设备号为 252，创建设备文件节点*/
```

然后编译并运行驱动测试程序：

```
$ arm-linux-gcc -o gpio_test gpio_test.c
$ ./gpio_test
```

运行结果为 4 个 LED 轮流闪烁，同时蜂鸣器以一定周期发出声响。

11.4 块设备驱动编程

块设备通常指一些需要以块（如 512 字节）的方式写入的设备，如 IDE 硬盘、SCSI 硬盘、光驱等。它的驱动程序的编写过程与字符型设备驱动程序的编写有很大的区别。

块设备驱动编程接口相对复杂，不如字符设备明晰易用。块设备驱动程序对整个系统的性能影响较大，速度和效率是设计块设备驱动程序要重点考虑的问题。系统中使

1. 编程流程说明

块设备驱动程序的编写流程同字符设备驱动程序的编写流程很类似，也包括了注册和使用两部分。但与字符驱动设备所不同的是，块设备驱动程序包括一个 request 请求队列。它是当内核安排一次数据传输时在列表中的一个请求队列，以最大化系统性能为原则进行排序。在后面的读写操作时会详细讲解这个函数，图 11.5 为块设备驱动程序的流程图，请读者注意与字符设备驱动程序的区别。

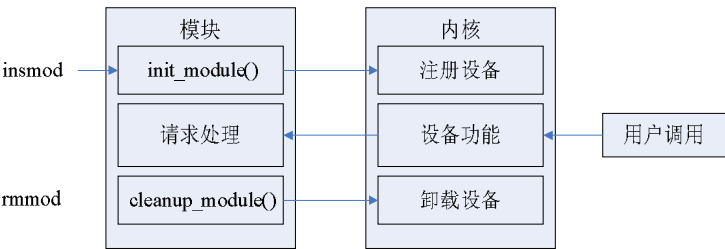


图 11.5 块设备驱动程序流程图

2. 重要数据结构

每个块设备物理实体由一个gendisk结构体来表示（在</linux/genhd.h>中定义），每个gendisk可以支持多个分区。

每个gendisk中包含了本物理实体的全部信息以及操作函数接口。整个块设备的注册过程是围绕gendisk来展开的。在驱动程序中需要初始化的gendisk的一些成员如下所示。

```
struct gendisk
{
    int major;           /* 主设备号 */
    int first_minor;     /* 第一个次设备号 */
    int minors;          /* 次设备号个数，一个块设备至少需要使用一个次设备号，而且块
                        设备的每个分区都需要一个次设备号，因此这个成员等于1，则表明该块
                        设备是不可被分区的，否则可以包含minors - 1 个分区。*/
    char disk_name[32];  /* 块设备名称，在/proc/partitions中显示 */
    struct hd_struct **part; /* 分区表 */
    struct block_device_operations *fops; /* 块设备操作接口，与字符设备
                        的
                        file_operations 结构对应 */
    struct request_queue *queue; /* I/O 请求队列 */
    void *private_data; /* 指向驱动程序私有数据 */
    sector_t capacity; /* 块设备可包含的扇区数 */
    ..... /* 其他省略 */
}
```


};

与字符设备驱动程序一样，块设备驱动程序也包含一个在<linux/fs.h>中定义的 block_device_operations 结构，其定义如下所示。

```
struct block_device_operations
{
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t, unsigned long
*);

    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    struct module *owner;
};
```

从该结构的定义中，可以看出块设备并不提供 read()、write()等函数接口。对块设备的读写请求都是以异步方式发送到设备相关的 request 队列之中。

3. 块设备注册和初始化

块设备的初始化过程要比字符设备复杂，它既需要像字符设备一样在加载内核时完成一定的工作，还需要在内核编译时增加一些内容。块设备驱动程序初始化时，由驱动程序的 init()完成。

块设备的初始化过程如图 11.6 所示。

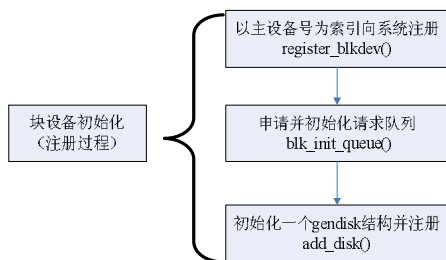


图 11.6 块设备驱动程序初始化过程

(1) 向内核注册。

使用 register_blkdev() 函数对设备进行注册。

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

其中参数 major 为要注册的块设备的主设备号，如果其值等于 0，则系统动态分配并返回主设备号。参数 name 为设备名，在 /proc/devices 中显示。如果出错，则该函

数返回负值。

与其对应的块设备的注销函数为 `unregister_blkdev()`，其格式如下所示。

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

其参数必须与注册函数中的参数相同。如果出错则返回负值。

(2) 申请并初始化请求队列。

这一步要调用 `blk_init_queue()`函数来申请并初始化请求队列，其格式如下所示。

```
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
```

其中参数 `rfn` 是请求队列的处理函数指针，它负责执行块设备的读、写请求。参数 `lock` 为自旋锁，用于控制对所分配的队列的访问。

(3) 初始化并注册 `gendisk` 结构。

内核提供的 `gendisk` 结构相关函数如表 11-16 所示。

表 11-16 `gendisk` 结构相关函数

函数格式	说明
<code>struct gendisk *alloc_disk(int minors)</code>	动态分配 <code>gendisk</code> 结构，参数为次设备号的个数
<code>void add_disk(struct gendisk *disk)</code>	向系统注册 <code>gendisk</code> 结构
<code>void del_gendisk(struct gendisk *disk)</code>	从系统注销 <code>gendisk</code> 结构

首先使用 `alloc_disk()`函数动态分配 `gendisk` 结构，接下来，对 `gendisk` 结构的主设备号（`major`）、次设备号相关成员（`first_minor` 和 `minors`）、块设备操作函数（`fops`）、请求队列（`queue`）、可包含的扇区数（`capacity`）以及设备名称（`disk_name`）等成员进行初始化。

在完成对 `gendisk` 的分配和初始化之后，调用 `add_disk（）`函数向系统注册块设备。在卸载 `gendisk` 结构的时候，要调用 `del_gendisk()`函数。

4. 块设备请求处理

块设备驱动中一般要实现一个请求队列处理函数来处理队列中的请求。从块设备的运行流程，可知请求处理是块设备的基本处理单位，也是最核心的部分。对块设备的读写操作被封装到了每一个请求中。

已经提过调用 `blk_init_queue()`函数来申请并初始化请求队列。表 11-17 列出了一些与请求处理相关的函数。

表 11-17 请求处理相关函数

函数格式	说明
<code>request_queue_t *blk_alloc_queue(int gfp_mask)</code>	分配请求队列
<code>request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)</code>	分配并初始化请求队列
<code>struct request *blk_get_request(request_queue_t *q, int rw, int gfp_mask)</code>	从队列中获取一个请求

<code>void blk_requeue_request(request_queue_t *q, struct request *rq)</code>	将请求再次加入队列
<code>void blk_queue_max_sectors(request_queue_t *q, unsigned short max_sectors)</code>	设置最大访问扇区数
<code>void blk_queue_max_phys_segments(request_queue_t *q, unsigned short max_segments)</code>	设置最大物理段数
<code>void end_request(struct request *req, int uptodate)</code>	结束本次请求处理
<code>void blk_queue_hardsect_size(request_queue_t *q, unsigned short size)</code>	设置物理扇区大小

以上简单地介绍了块设备驱动编程的最基本的概念和流程。更深入的内容不是本书的重点，有兴趣的读者可以参考其他书籍。

11.5 中断编程

前面所讲述的驱动程序中都没有涉及中断处理，而实际上，有很多 Linux 的驱动都是通过中断的方式来进行内核和硬件的交互。中断机制提供了硬件和软件之间异步传递信息的方式。硬件设备在发生某个事件时通过中断通知软件进行处理。中断实现了硬件设备按需获得处理器关注的机制，与查询方式相比可以大大节省 CPU 资源的开销。

在此将介绍在驱动程序中用于申请中断的 `request_irq()` 调用，和用于释放中断的 `free_irq()` 调用。`request_irq()` 函数调用的格式如下所示：

```
int request_irq(unsigned int irq,
                void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
                unsigned long irqflags, const char * devname, oid *dev_id);
```

其中 `irq` 是要申请的硬件中断号。在 Intel 平台，范围是 0~15。

参数 `handler` 为将要向系统注册的中断处理函数。这是一个回调函数，中断发生时，系统调用这个函数，传入的参数包括硬件中断号、设备 id 以及寄存器值。设备 id 就是在调用 `request_irq()` 时传递给系统的参数 `dev_id`。

参数 `irqflags` 是中断处理的一些属性，其中比较重要的有 `SA_INTERRUPT`。这个参数用于标明中断处理程序是快速处理程序（设置 `SA_INTERRUPT`）还是慢速处理程序（不设置 `SA_INTERRUPT`）。快速处理程序被调用时屏蔽所有中断。慢速处理程序只屏蔽正在处理的中断。还有一个 `SA_SHIRQ` 属性，设置了以后运行多个设备共享中断，在中断处理程序中根据 `dev_id` 区分不同设备产生的中断。

参数 `devname` 为设备名，会在 `/dev/interrupts` 中显示。

参数 `dev_id` 在中断共享时会用到。一般设置为这个设备的 `device` 结构本身或者 `NULL`。中断处理程序可以用 `dev_id` 找到相应的控制这个中断的设备，或者用 `irq2dev_map()` 找到中断对应的设备。

释放中断的 `free_irq()` 函数调用的格式如下所示。该函数的参数与 `request_irq()` 相同。

```
void free_irq(unsigned int irq, void *dev_id);
```

11.6 按键驱动程序实例

11.6.1 按键工作原理

LED 和蜂鸣器是最简单的 GPIO 的应用，都不需要任何外部输入或控制。按键同样使用 GPIO 接口，但按键本身需要外部的输入，即在驱动程序中要处理外部中断。按键硬件驱动原理图如图 11-7 所示。在图 11-7 的 4×4 矩阵按键（K1~K16）电路中，使用 4 个输入/输出端口（EINT0、EINT2、EINT11 和 EINT19）和 4 个输出端口（KSCAN0~KSCAN3）。

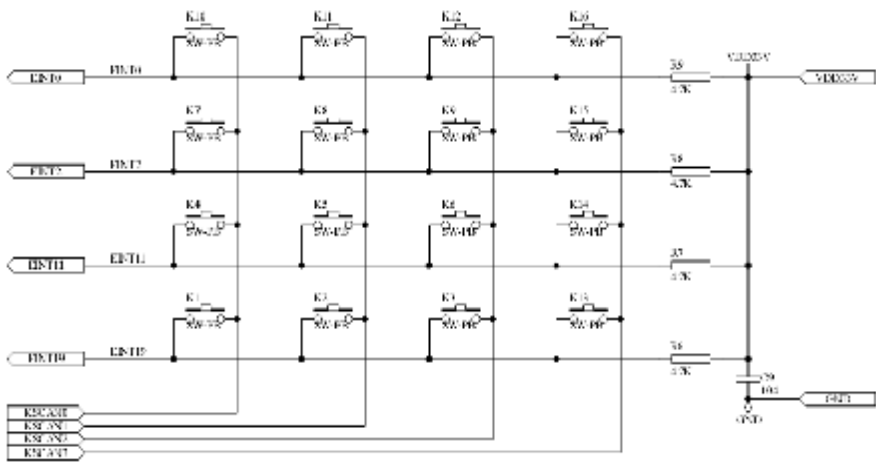


图 11.7 按键驱动电路原理图

按键驱动电路使用的端口和对应的寄存器如表 11-18 所示。

表 11.18 按键电路的主要端口

管脚	端口	输入/输出	管脚	端口	输入/输出
KEYSCAN0	GPE11	输出	EINT0	EINT0/GPF0	输入/输出
KEYSCAN1	GPG6	输出	EINT2	EINT2/GPF2	输入/输出
KEYSCAN2	GPE13	输出	EINT11	EINT11/GPG3	输入/输出
KEYSCAN3	GPG2	输出	EINT19	EINT19/GPG11	输入/输出

因为通常中断端口是比较珍贵且有限的资源，所以在本电路设计中，16 个按键复用了 4 个中断线。那怎么样才能及时而准确地对矩阵按键进行扫描呢？

某个中断的产生表示，与它所对应的矩阵行的 4 个按键中，至少有一个按键被按住了。因此可以通过查看产生了哪个中断，来确定在矩阵的哪一行中发生了按键操作（按住或释放）。例如，如果产生了外部 2 号线上中断（EINT2 变为低电平），则表示 K7、K8、K9 和 K15 中至少有一个按键被按住了。这时候 4 个 EINT 端口应该通过 GPIO

配置寄存器被设置为外部中断端口，而且 4 个 KSCAN 端口的输出必须为低电平。

在确定按键操作所在行的位置之后，我们还得查看按键操作所在列的位置。此时要使用 KSCAN 端口组，同时将 4 个 EINT 端口配置为通用输入端口（而不是中断端口）。在 4 个 KSCAN 端口中，轮流将其中某一个端口的输出置为低电平，其他 3 个端口的输出置为高电平。这样逐列进行扫描，直到按键所在列的 KSCAN 端口输出为低电平，此时按键操作所在行的 EINT 管脚的输入端口的值会变成低电平。例如，在确认产生了外部 2 号中断之后，进行逐列扫描。若发现在 KSCAN1 为低电平时（其他端口输出均为高电平），GPF2（EINT2 管脚的输入端口）变为低电平，则可以断定按键 K8 被按住了。

以上的讨论都是在按键的理想状态下进行的，但实际的按键动作会在短时间（几毫秒至几十毫秒）内产生信号抖动。例如，当按键被按下时，其动作就像弹簧的若干次往复运动，将产生几个脉冲信号。一次按键操作将会产生若干次按键中断，从而会产生抖动现象。因此驱动程序中必须要解决去除抖动所产生的毛刺信号的问题。

11.6.2 按键驱动程序

首先按键设备相关的数据结构的定义如下所示：

```
/* butt_drv.h */
.....
typedef struct _st_key_info_matrix          /* 按键数据结构 */
{
    unsigned char    key_id;                /* 按键 ID */
    unsigned int     irq_no;                /* 对应的中断号 */
    unsigned int     irq_gpio_port;         /* 对应的中断线的输入端口地址 */
} st_key_info_matrix;

typedef struct _st_key_buffer               /* 按键缓冲数据结构 */
{
    unsigned long    jiffy[MAX_KEY_COUNT]; /* 按键时间，5s 以前的按键作废 */
    unsigned char    buf[MAX_KEY_COUNT];   /* 按键缓冲区 */
    unsigned int     head,tail;             /* 按键缓冲区头和尾 */
} st_key_buffer;
.....
```

下面是矩阵按键数组的定义，数组元素的信息（一个按键信息）按照 0 行 0 列，0 行 1 列，...，3 行 2 列，3 行 3 列的顺序逐行排列。

```
static st_key_info_matrix key_info_matrix[MAX_COLUMN][MAX_ROW] =
{
```

```

    {{10,      IRQ_EINT0,  S3C2410_GPF0,  S3C2410_GPE11}},      /* 0 行 0 列 */
    /*
    {{11,      IRQ_EINT0,  S3C2410_GPF0,  S3C2410_GPG6}},
    {{12,      IRQ_EINT0,  S3C2410_GPF0,  S3C2410_GPE13}},
    {{16,      IRQ_EINT0,  S3C2410_GPF0,  S3C2410_GPG2}},

    {{7,       IRQ_EINT2,  S3C2410_GPF2,  S3C2410_GPE11}},      /* 1 行 0 列 */
    {{8,       IRQ_EINT2,  S3C2410_GPF2,  S3C2410_GPG6}},
    {{9,       IRQ_EINT2,  S3C2410_GPF2,  S3C2410_GPE13}},
    {{15,      IRQ_EINT2,  S3C2410_GPF2,  S3C2410_GPG2}},

    {{4,       IRQ_EINT11, S3C2410_GPG3,  S3C2410_GPE11}},      /* 2 行 0 列 */
    /*
    {{5,       IRQ_EINT11, S3C2410_GPG3,  S3C2410_GPG6}},
    {{6,       IRQ_EINT11, S3C2410_GPG3,  S3C2410_GPE13}},
    {{14,      IRQ_EINT11, S3C2410_GPG3,  S3C2410_GPG2}},

    {{1,       IRQ_EINT19, S3C2410_GPG11, S3C2410_GPE11}},      /* 3 行 0 列 */
    {{2,       IRQ_EINT19, S3C2410_GPG11, S3C2410_GPG6}},
    {{3,       IRQ_EINT19, S3C2410_GPG11, S3C2410_GPE13}},
    {{13,      IRQ_EINT19, S3C2410_GPG11, S3C2410_GPG2}},
    };

```

下面是与按键相关的端口的初始化函数。这些函数已经在简单的 GPIO 字符设备驱动程序里被使用过。此外，set_irq_type() 函数用于设定中断线的类型，在本实例中通过该函数将 4 个中断线的类型配置为下降沿触发式。

```

static void init_gpio(void)
{
    s3c2410_gpio_cfgpin(S3C2410_GPE11, S3C2410_GPE11_OUTP); /* GPE11 */
    /*
    s3c2410_gpio_setpin(S3C2410_GPE11, 0);
    s3c2410_gpio_cfgpin(S3C2410_GPE13, S3C2410_GPE13_OUTP); /* GPE13 */
    /*
    s3c2410_gpio_setpin(S3C2410_GPE13, 0);
    s3c2410_gpio_cfgpin(S3C2410_GPG2, S3C2410_GPG2_OUTP); /* GPG2 */
    s3c2410_gpio_setpin(S3C2410_GPG2, 0);
    s3c2410_gpio_cfgpin(S3C2410_GPG6, S3C2410_GPG6_OUTP); /* GPG6 */
    s3c2410_gpio_setpin(S3C2410_GPG6, 0);

    s3c2410_gpio_cfgpin(S3C2410_GPF0, S3C2410_GPF0_EINT0); /* GPF0 */

```




```
s3c2410_gpio_cfgpin(S3C2410_GPF2, S3C2410_GPF2_EINT2); /* GPF2 */
s3c2410_gpio_cfgpin(S3C2410_GPG3, S3C2410_GPG3_EINT11); /* GPG3 */
s3c2410_gpio_cfgpin(S3C2410_GPG11, S3C2410_GPG11_EINT19); /* GPG11
*/

set_irq_type(IRQ_EINT0, IRQT_FALLING);
set_irq_type(IRQ_EINT2, IRQT_FALLING);
set_irq_type(IRQ_EINT11, IRQT_FALLING);
set_irq_type(IRQ_EINT19, IRQT_FALLING);
}
```

下面讲解按键驱动的主要接口，以下为驱动模块的入口和卸载函数。

```
/* 初始化并添加 struct cdev 结构到系统之中 */
static void button_setup_cdev(struct cdev *dev,
                             int minor, struct file_operations *fops)
{
    int err;
    int devno = MKDEV(button_major, minor);
    cdev_init(dev, fops); /* 初始化结构体 struct cdev */
    dev->owner = THIS_MODULE;
    dev->ops = fops; /* 关联到设备的 file_operations 结构 */
    err = cdev_add(dev, devno, 1); /* 将 struct cdev 结构添加到系统之中 */
    if (err)
    {
        printk(KERN_INFO "Error %d adding button %d\n", err, minor);
    }
}

.....
/* 驱动初始化 */
static int button_init(void)
{
    int ret;
    /* 将主设备号和次设备号定义到一个 dev_t 数据类型的结构体之中 */
    dev_t dev = MKDEV(button_major, 0);
    if (button_major)
    {
        /* 静态注册一个设备，设备号先前指定好，并设定设备名，用 cat /proc/devices 来查看 */
        ret = register_chrdev_region(dev, 1, BUTTONS_DEVICE_NAME);
    }
    else
}
```

```

{ /* 由系统动态分配主设备号 */
    ret = alloc_chrdev_region(&dev, 0, 1, BUTTONS_DEVICE_NAME);
    button_major = MAJOR(dev); /* 获得主设备号 */
}

if (ret < 0)
{
    printk(KERN_WARNING"Button:unable          to          get          major
%d\n",button_major);
    return ret;
}
/* 初始化和添加结构体 struct cdev 到系统之中 */
button_setup_cdev(&button_dev, 0, &button_fops);
printk("Button driver initialized.\n");
return 0;
}
/* 驱动卸载 */
static void __exit button_exit(void)
{
    cdev_del(&button_dev); /* 删除结构体 struct cdev */
    /* 卸载设备驱动所占有的资源 */
    unregister_chrdev_region(MKDEV(button_major, 0), 1);
    printk("Button driver uninstalled\n");
}
module_init(button_init); /* 初始化设备驱动程序的入口 */
module_exit(button_exit); /* 卸载设备驱动程序的入口 */
MODULE_AUTHOR("David");
MODULE_LICENSE("Dual BSD/GPL");

```

按键字符设备的 file_operations 结构定义为：

```

static struct file_operations button_fops =
{
    .owner = THIS_MODULE,
    .ioctl = button_ioctl,
    .open = button_open,
    .read = button_read,
    .release = button_release,
};

```

以下为 open 和 release 函数接口的实现。

```

/* 打开文件, 申请中断 */
static int button_open(struct inode *inode, struct file *filp)
{
    int ret = nonseekable_open(inode, filp);
    if (ret < 0)
    {
        return ret;
    }

    init_gpio();                /* 相关 GPIO 端口的初始化 */
    ret = request_irqs();        /* 申请 4 个中断 */
    if (ret < 0)
    {
        return ret;
    }
    init_keybuffer();           /* 初始化按键缓冲数据结构 */
    return ret;
}

/* 关闭文件, 屏蔽中断 */
static int button_release(struct inode *inode, struct file *filp)
{
    free_irqs();                /* 屏蔽中断 */
    return 0;
}

```

在 `open` 函数接口中, 进行了 GPIO 端口的初始化、申请硬件中断以及按键缓冲的初始化等工作。在以前的章节中提过, 中断端口是比较宝贵而且数量有限的资源。因此需要注意, 最好要在第一次打开设备时申请 (调用 `request_irq` 函数) 中断端口, 而不是在驱动模块加载的时候申请。如果已加载的设备驱动占用而在一定时间段内不使用某些中断资源, 则这些资源不会被其他驱动所使用, 只能白白浪费掉。而在打开设备的时候 (调用 `open` 函数接口) 申请中断, 则不同的设备驱动可以共享这些宝贵的中断资源。

以下为中断申请和释放的部分以及中断处理函数。

```

/* 中断处理函数, 其中 irq 为中断号 */
static irqreturn_t button_irq(int irq, void *dev_id, struct pt_regs
*regs)
{
    unsigned char ucKey = 0;

```

```

disable_irqs();           /* 屏蔽中断 */
/* 延迟 50ms, 屏蔽按键毛刺 */
udelay(50000);

ucKey = button_scan(irq); /* 扫描按键, 获得进行操作的按键的 ID */
if ((ucKey >= 1) && (ucKey <= 16))
{
    /* 如果缓冲区已满, 则不添加 */
    if (((key_buffer.head + 1) & (MAX_KEY_COUNT - 1)) !=
key_buffer.tail)
    {
        spin_lock_irq(&buffer_lock);
        key_buffer.buf[key_buffer.tail] = ucKey;
        key_buffer.jiffy[key_buffer.tail] = get_tick_count();
        key_buffer.tail ++;
        key_buffer.tail &= (MAX_KEY_COUNT - 1);
        spin_unlock_irq(&buffer_lock);
    }
}

init_gpio();             /* 初始化 GPIO 端口, 主要是为了恢复中断端口配置 */
enable_irqs();          /* 开启中断 */
return IRQ_HANDLED; /* 2.6 内核返回值一般是这个宏 */
}

/* 申请 4 个中断 */
static int request_irqs(void)
{
    int ret, i, j;
    for (i = 0; i < MAX_COLUMN; i++)
    {
        ret = request_irq(key_info_matrix[i][0].irq_no,
button_irq, SA_INTERRUPT, BUTTONS_DEVICE_NAME, NULL);
        if (ret < 0)
        {
            for (j = 0; j < i; j++)
            {
                free_irq(key_info_matrix[j][0].irq_no, NULL);
            }
            return -EFAULT;
        }
    }
}

return 0;

```



```

}

/* 释放中断 */
static __inline void free_irqs(void)
{
    int i;
    for (i = 0; i < MAX_COLUMN; i++)
    {
        free_irq(key_info_matrix[i][0].irq_no, NULL);
    }
}

```

中断处理函数在每次中断产生的时候会被调用，因此它的执行时间要尽可能得短。通常中断处理函数只是简单地唤醒等待资源的任务，而复杂且耗时的工作则让这个任务去完成。中断处理函数不能向用户空间发送数据或者接收数据，不能做任何可能发生睡眠的操作，而且不能调用 `schedule()` 函数。

为了简单起见，而且考虑到按键操作的时间比较长，在本实例中的中断处理函数 `button_irq()` 里，通过调用睡眠函数来消除毛刺信号。读者可以根据以上介绍的对中断处理函数的要求改进该部分代码。

按键扫描函数如下所示。首先根据中断号确定操作按键所在行的位置，然后采用逐列扫描法最终确定操作按键所在的位置。

```

/*
** 进入中断后，扫描按键码
** 返回：按键码(1~16)，0xff 表示错误
*/
static __inline unsigned char button_scan(int irq)
{
    unsigned char key_id = 0xff;
    unsigned char column = 0xff, row = 0xff;

    s3c2410_gpio_cfgpin(S3C2410_GPF0, S3C2410_GPF0_INP); /* GPF0 */
    s3c2410_gpio_cfgpin(S3C2410_GPF2, S3C2410_GPF2_INP); /* GPF2 */
    s3c2410_gpio_cfgpin(S3C2410_GPG3, S3C2410_GPG3_INP); /* GPG3 */
    s3c2410_gpio_cfgpin(S3C2410_GPG11, S3C2410_GPG11_INP); /* GPG11 */

    switch (irq)
    { /* 根据 irq 值确定操作按键所在行的位置 */
        case IRQ_EINT0:
        {
            column = 0;
        }
    }
}

```

```
break;
case IRQ_EINT2:
{
    column = 1;
}
break;
case IRQ_EINT11:
{
    column = 2;
}
break;
case IRQ_EINT19:
{
    column = 3;
}
break;
}
if (column != 0xff)
{ /* 开始逐列扫描, 扫描第 0 列 */
    s3c2410_gpio_setpin(S3C2410_GPE11, 0); /* 将 KSCAN0 置为低电平 */
    s3c2410_gpio_setpin(S3C2410_GPG6, 1);
    s3c2410_gpio_setpin(S3C2410_GPE13, 1);
    s3c2410_gpio_setpin(S3C2410_GPG2, 1);

    if(!s3c2410_gpio_getpin(key_info_matrix[column][0].irq_gpio_port))
    { /* 观察对应的中断线的输入端口值 */
        key_id = key_info_matrix[column][0].key_id;
        return key_id;
    }

    /* 扫描第 1 列*/
    s3c2410_gpio_setpin(S3C2410_GPE11, 1);
    s3c2410_gpio_setpin(S3C2410_GPG6, 0); /* 将 KSCAN1 置为低电平 */
    s3c2410_gpio_setpin(S3C2410_GPE13, 1);
    s3c2410_gpio_setpin(S3C2410_GPG2, 1);

    if(!s3c2410_gpio_getpin(key_info_matrix[column][1].irq_gpio_port))
    {
        key_id = key_info_matrix[column][1].key_id;
        return key_id;
    }
}
```



```

/* 扫描第 2 列 */
s3c2410_gpio_setpin(S3C2410_GPE11, 1);
s3c2410_gpio_setpin(S3C2410_GPG6, 1);
s3c2410_gpio_setpin(S3C2410_GPE13, 0); /* 将 KSCAN2 置为低电平 */
s3c2410_gpio_setpin(S3C2410_GPG2, 1);

if(!s3c2410_gpio_getpin(key_info_matrix[column][2].irq_gpio_port))
{
    key_id = key_info_matrix[column][2].key_id;
    return key_id;
}

/* 扫描第 3 列 */
s3c2410_gpio_setpin(S3C2410_GPE11, 1);
s3c2410_gpio_setpin(S3C2410_GPG6, 1);
s3c2410_gpio_setpin(S3C2410_GPE13, 1);
s3c2410_gpio_setpin(S3C2410_GPG2, 0); /* 将 KSCAN3 置为低电平 */

if(!s3c2410_gpio_getpin(key_info_matrix[column][3].irq_gpio_port))
{
    key_id = key_info_matrix[column][3].key_id;
    return key_id;
}
}
return key_id;
}
}

```

以下是 read 函数接口的实现。首先在按键缓冲中删除已经过时的按键操作信息，接下来，从按键缓冲中读取一条信息（按键 ID）并传递给用户层。

```

/* 从缓冲删除过时数据(5s 前的按键值) */
static void remove_timeoutkey(void)
{
    unsigned long tick;
    spin_lock_irq(&buffer_lock); /* 获得一个自旋锁 */
    while(key_buffer.head != key_buffer.tail)
    {
        tick = get_tick_count() - key_buffer.jiffy[key_buffer.head];
        if (tick < 5000) /* 5s */
            break;
        key_buffer.buf[key_buffer.head] = 0;
        key_buffer.jiffy[key_buffer.head] = 0;
    }
}

```



```

        key_buffer.head ++;
        key_buffer.head &= (MAX_KEY_COUNT - 1);
    }
    spin_unlock_irq(&buffer_lock); /* 释放自旋锁 */
}

/* 读键盘 */
static ssize_t button_read(struct file *filp,
                           char *buffer, size_t count, loff_t *f_pos)
{
    ssize_t ret = 0;
    remove_timeoutkey(); /* 删除过时的按键操作信息 */
    spin_lock_irq(&buffer_lock);
    while((key_buffer.head != key_buffer.tail) && (((size_t)ret) <
count))
    {
        put_user((char)(key_buffer.buf[key_buffer.head]),
&buffer[ret]);
        key_buffer.buf[key_buffer.head] = 0;
        key_buffer.jiffy[key_buffer.head] = 0;
        key_buffer.head ++;
        key_buffer.head &= (MAX_KEY_COUNT - 1);
        ret ++;
    }
    spin_unlock_irq(&buffer_lock);
    return ret;
}

```

以上介绍了按键驱动程序中的主要内容。

11.6.3 按键驱动的测试程序

按键驱动程序的测试程序如下所示。在测试程序中，首先打开按键设备文件和 gpio 设备（包括 4 个 LED 和蜂鸣器）文件，接下来，根据按键的输入值（按键 ID）的二进制形式，LED D9~D12 发亮（例如，按下 11 号按键，则 D9、D10 和 D12 会发亮），而蜂鸣器当每次按键时发出声响。

```

/* butt_test.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>

```



```
#include <unistd.h>
#include <asm/delay.h>
#include "butt_drv.h"
#include "gpio_drv.h"

main()
{
    int butt_fd, gpios_fd, i;
    unsigned char key = 0x0;
    butt_fd = open(BUTTONS_DEVICE_FILENAME, O_RDWR); /* 打开按钮设备 */
    if (butt_fd == -1)
    {
        printf("Open button device button errr!\n");
        return 0;
    }

    gpios_fd = open(GPIO_DEVICE_FILENAME, O_RDWR); /* 打开 GPIO 设备 */
    if (gpios_fd == -1)
    {
        printf("Open button device button errr!\n");
        return 0;
    }

    ioctl(butt_fd, 0); /* 清空键盘缓冲区, 后面参数没有意义 */
    printf("Press No.16 key to exit\n");
    do
    {
        if (read(butt_fd, &key, 1) <= 0) /* 读键盘设备, 得到相应的键值 */
        {
            continue;
        }

        printf("Key Value = %d\n", key);
        for (i = 0; i < LED_NUM; i++)
        {
            if ((key & (1 << i)) != 0)
            {
                ioctl(gpios_fd, LED_D09_SWT + i, LED_SWT_ON); /* LED 发亮
*/
            }
        }
    }
```



```

    }

    ioctl(gpios_fd, BEEP_SWT, BEEP_SWT_ON); /* 发声*/

    sleep(1);
    for (i = 0; i < LED_NUM; i++)
    {
        ioctl(gpios_fd, LED_D09_SWT + i, LED_SWT_OFF); /* LED 熄灭 */
    }

    ioctl(gpios_fd, BEEP_SWT, BEEP_SWT_OFF);

    } while(key != 16); /* 按 16 号键则退出 */
    close(gpios_fd);
    close(butt_fd);
    return 0;
}

```

首先编译和加载按键驱动程序，而且要创建设备文件节点。

```

$ make clean;make /* 驱动程序的编译*/
$ insmod butt_dev.ko /* 加载 buttons 设备驱动 */
$ cat /proc/devices /* 通过这个命令可以查到 buttons 设备的主设备号 */
$ mknod /dev/buttons c 252 0 /* 假设主设备号为 252，创建设备文件节点*/

```

接下来，编译和加载 GPIO 驱动程序，而且要创建设备文件节点。

```

$ make clean;make /* 驱动程序的编译*/
$ insmod gpio_drv.ko /* 加载 GPIO 驱动 */
$ cat /proc/devices /* 通过这个命令可以查到 GPIO 设备的主设备号 */
$ mknod /dev/gpio c 251 0 /* 假设主设备号为 251，创建设备文件节点*/

```

然后编译并运行驱动测试程序。

```

$ arm-linux-gcc -o butt_test butt_test.c
$ ./butt_test

```

11.7 实验内容——test 驱动

1. 实验目的

该实验是编写最简单的字符驱动程序，这里的设备也就是一段内存，实现简单的读写功能，并列出常用格式的 Makefile 以及驱动的加载和卸载脚本。读者可以熟悉字

2. 实验内容

该实验要求实现对虚拟设备（一段内存）的打开、关闭、读写的操作，并要通过编写测试程序来测试虚拟设备及其驱动运行是否正常。

3. 实验步骤

(1) 编写代码。

这个简单的驱动程序的源代码如下所示：

```
/* test_drv.c */
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>

#define TEST_DEVICE_NAME "test_dev"
#define BUFF_SZ 1024

/*全局变量*/
static struct cdev test_dev;
unsigned int major =0;
static char *data = NULL;

/*读函数*/
static ssize_t test_read(struct file *file,
                        char *buf, size_t count, loff_t *f_pos)
{
    int len;
    if (count < 0 )
    {
        return -EINVAL;
    }
    len = strlen(data);
    count = (len > count)?count:len;
    if (copy_to_user(buf, data, count)) /* 将内核缓冲的数据拷贝到用户空间
```

```

*/
    {
        return -EFAULT;
    }
    return count;
}

/*写函数*/
static ssize_t test_write(struct file *file, const char *buffer,
                          size_t count, loff_t *f_pos)
{
    if(count < 0)
    {
        return -EINVAL;
    }
    memset(data, 0, BUFF_SZ);
    count = (BUFF_SZ > count)?count:BUFF_SZ;
    if (copy_from_user(data, buffer, count)) /* 将用户缓冲的数据复制到内核空间
*/
    {
        return -EFAULT;
    }
    return count;
}

/*打开函数*/
static int test_open(struct inode *inode, struct file *file)
{
    printk("This is open operation\n");
    /* 分配并初始化缓冲区*/
    data = (char*)kmalloc(sizeof(char) * BUFF_SZ, GFP_KERNEL);
    if (!data)
    {
        return -ENOMEM;
    }
    memset(data, 0, BUFF_SZ);
    return 0;
}

/*关闭函数*/
static int test_release(struct inode *inode, struct file *file)
{
    printk("This is release operation\n");

```



```
if (data)
{
    kfree(data); /* 释放缓冲区*/
    data = NULL; /* 防止出现野指针 */
}
return 0;
}

/* 创建、初始化字符设备，并且注册到系统*/
static void test_setup_cdev(struct cdev *dev, int minor,
    struct file_operations *fops)
{
    int err, devno = MKDEV(major, minor);
    cdev_init(dev, fops);
    dev->owner = THIS_MODULE;
    dev->ops = fops;
    err = cdev_add (dev, devno, 1);
    if (err)
    {
        printk (KERN_NOTICE "Error %d adding test %d", err, minor);
    }
}

/* 虚拟设备的 file_operations 结构 */
static struct file_operations test_fops =
{
    .owner    = THIS_MODULE,
    .read     = test_read,
    .write    = test_write,
    .open     = test_open,
    .release  = test_release,
};

/*模块注册入口*/
int init_module(void)
{
    int result;
    dev_t dev = MKDEV(major, 0);

    if (major)
    {
        /* 静态注册一个设备，设备号先前指定好，并设定设备名，用 cat /proc/devices 来查
```

```

看 */
    result = register_chrdev_region(dev, 1, TEST_DEVICE_NAME);
}
else
{
    result = alloc_chrdev_region(&dev, 0, 1, TEST_DEVICE_NAME);
}

if (result < 0)
{
    printk(KERN_WARNING "Test device: unable to get major %d\n",
major);
    return result;
}
test_setup_cdev(&test_dev, 0, &test_fops);
printk("The major of the test device is %d\n", major);
return 0;
}

/*卸载模块*/
void cleanup_module(void)
{
    cdev_del(&test_dev);
    unregister_chrdev_region(MKDEV(major, 0), 1);
    printk("Test device uninstalled\n");
}

```

(2) 编译代码。

虚拟设备的驱动程序的 Makefile 如下所示：

```

ifeq ($(KERNELRELEASE),)
KERNELDIR ?= /lib/modules/$(shell uname -r)/build /*内核代码编译路径*/
PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
modules_install:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
clean:
    rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .tmp_versions
.PHONY: modules modules_install clean
else

```




```
obj-m := test_drv.o    /* 将生成的模块为 test_drv.ko */
endif
```

(3) 加载和卸载模块。

通过下面两个脚本代码分别实现驱动模块的加载和卸载。

加载脚本 `test_drv_load` 如下所示：

```
#!/bin/sh
# 驱动模块名称
module="test_drv"
# 设备名称。在 /proc/devices 中出现
device="test_dev"
# 设备文件的属性
mode="664"
group="david"

# 删除已存在的设备节点
rm -f /dev/${device}
# 加载驱动模块
/sbin/insmod -f ./${module}.ko $* || exit 1
# 查到创建设备的主设备号
major=`cat /proc/devices | awk "\\$2==\"$device\" {print \\$1}"`
# 创建设备文件节点
mknod /dev/${device} c $major 0
# 设置设备文件属性
chgrp $group /dev/${device}
chmod $mode /dev/${device}
```

卸载脚本 `test_drv_unload` 如下所示：

```
#!/bin/sh
module="test_drv"
device="test_dev"
# 卸载驱动模块
/sbin/rmmod $module $* || exit 1
# 删除设备文件
rm -f /dev/${device}
exit 0
```

(6) 编写测试代码。

最后一步是编写测试代码，也就是用户空间的程序，该程序调用设备驱动来测试驱动的运行是否正常。以下实例只实现了简单的读写功能，测试代码如下所示：

```

/* test.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#define TEST_DEVICE_FILENAME "/dev/test_dev" /* 设备文件名 */

#define BUFF_SZ 1024 /* 缓冲大小 */

int main()
{
    int fd, nwrite, nread;
    char buff[BUFF_SZ]; /*缓冲区*/
    /* 打开设备文件 */
    fd = open(TEST_DEVICE_FILENAME, O_RDWR);
    if (fd < 0)
    {
        perror("open");
        exit(1);
    }

    do
    {
        printf("Input some words to kernel(enter 'quit' to exit):");
        memset(buff, 0, BUFF_SZ);
        if (fgets(buff, BUFF_SZ, stdin) == NULL)
        {
            perror("fgets");
            break;
        }
        buff[strlen(buff) - 1] = '\0';
        if (write(fd, buff, strlen(buff)) < 0) /* 向设备写入数据 */
        {
            perror("write");
            break;
        }

        if (read(fd, buff, BUFF_SZ) < 0) /* 从设备读取数据 */

```



```
{
    perror("read");
    break;
}
else
{
    printf("The read string is from kernel:%s\n", buff);
}
} while(strncmp(buff, "quit", 4));
close(fd);
exit(0);
}
```

4. 实验结果

首先在虚拟设备驱动源码目录下编译并加载驱动模块。

```
$ make clean;make
$ ./test_drv_load
```

接下来，编译并运行测试程序

```
$ gcc -o test test.c
$ ./test
```

测试程序运行效果如下：

```
Input some words to kernel(enter 'quit' to exit):Hello, everybody!
The read string is from kernel:Hello, everybody! /* 从内核读取的数据 */
Input some words to kernel(enter 'quit' to exit):This is a simple driver
The read string is from kernel: This is a simple driver
Input some words to kernel(enter 'quit' to exit):quit
The read string is from kernel:quit
```

最后，卸载驱动程序

```
$ ./test_drv_unload
```

通过 dmesg 命令可以查看内核打印的信息：

```
$ dmesg|tail -n 10
.....
The major of the test device is 250      /* 当加载模块时打印 */
This is open operation                  /* 当打开设备时打印 */
This is release operation                /* 关闭设备时打印 */
```

11.8 本章小结

本章主要介绍了嵌入式 Linux 设备驱动程序的开发。首先介绍了设备驱动程序的概念及 Linux 对设备驱动的处理，这里要明确驱动程序在 Linux 中的定位。

接下来介绍了字符设备驱动程序的编写，这里详细介绍了字符设备驱动程序的编写流程、重要的数据结构、设备驱动程序的主要组成以及 proc 文件系统。接着又以 GPIO 驱动为例介绍了一个简单的字符驱动程序的编写步骤。

再接下来，本章介绍了块设备驱动程序的编写，主要包括块设备驱动程序描述符和块设备驱动的编写流程。

最后，本章介绍了中断编程，并以编写完整的按键驱动程序为例进行讲解。

本章的实验安排的是简单虚拟设备驱动程序的编写，通过该实验，读者可以了解到编写驱动程序的完整流程。

11.9 思考与练习

根据书上的提示，将本章中所述的按键驱动程序进行进一步的改进，并在开发板上进行测试。

推荐课程：嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章: <http://www.embedu.org/courses/index.htm>
- 课程内容: <http://www.embedu.org/courses/course1.htm>
- 项目实战: <http://www.embedu.org/courses/project.htm>
- 出版教材: <http://www.embedu.org/courses/course3.htm>
- 实验设备: <http://www.embedu.org/courses/course5.htm>

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班:

<http://www.farsight.com.cn/courses/TS-LinuxApp.htm>

- 嵌入式 Linux 系统开发班:

<http://www.farsight.com.cn/courses/TS-LinuxEMB.htm>

- 嵌入式 Linux 驱动开发班:

<http://www.farsight.com.cn/courses/TS-LinuxDriver.htm>

华清远见