“黑色经典”系列之《嵌入式 Linux 应用程序开发详解》

第 11 章 嵌入式 Linux 设备驱动开发

本章目标

本书从第 6 章到第 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。经过本章的学习，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
- 能够独立定制 Linux 服务

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，也就是将最基本的核心代码编译在内核中，其他的代码可以选择在内核中，或者编译为内核的模块文件。

常见的驱动程序也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，加载驱动时就是加载内核模块。

这里，首先列举一些模块相关命令。

- 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，第三列则是该模块使用的数量。如下所示：

```
[root@www root]# lsmod
Module                  Size  Used by
autofs                 12068   0  (autoclean) (unused)
eepro100               18128   1
iptable_nat            19252   0  (autoclean) (unused)
ip_conntrack           18540   1  (autoclean) [iptable_nat]
iptable_mangle          2272   0  (autoclean) (unused)
iptable_filter          2272   0  (autoclean) (unused)
ip_tables              11936   5  [iptable_nat iptable_mangle iptable_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]
```

- rmmod 是用于将当前模块卸载。
- insmod 和 modprobe 是用于加载当前模块，但 insmod 不会自动解决依存关系，而 modprobe 则可以根据模块间依存关系以及/etc/modules.conf 文件中的内容自动插入模块。
- mknod 是用于创建相关模块。

11.1.2 设备文件分类

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 系统的设备文件分为三类：块设备文件、字符设备文件和网络设备文件。

- 块设备文件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。
- 字符型设备文件通常指可以直接读写，没有缓冲区的设备，如并口、虚拟控制台等。
- 网络设备文件通常是指网络设备访问的 BSD socket 接口，如网卡等。

对这三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。主设备号表明某一类设备，一般对应着确定的驱动程序；次设备号一般是用于区分标明不同属性，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。高字节为主设备号和底字节为次设备号。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……

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，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。

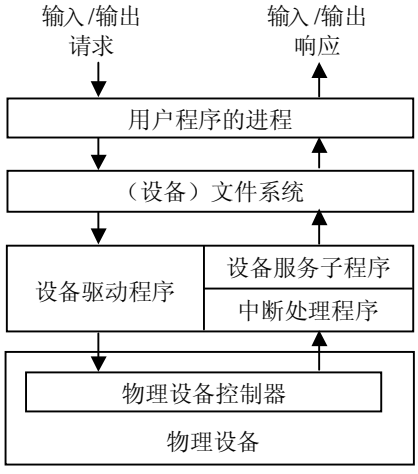

图 11.1 Linux 输入/输出系统层次结构和功能

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。

设备控制器需要获得系统服务时有两种方式：查询和中断。因为 Linux 下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三个部分。

- 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。
- 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。
- 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下图 11.2 所示。

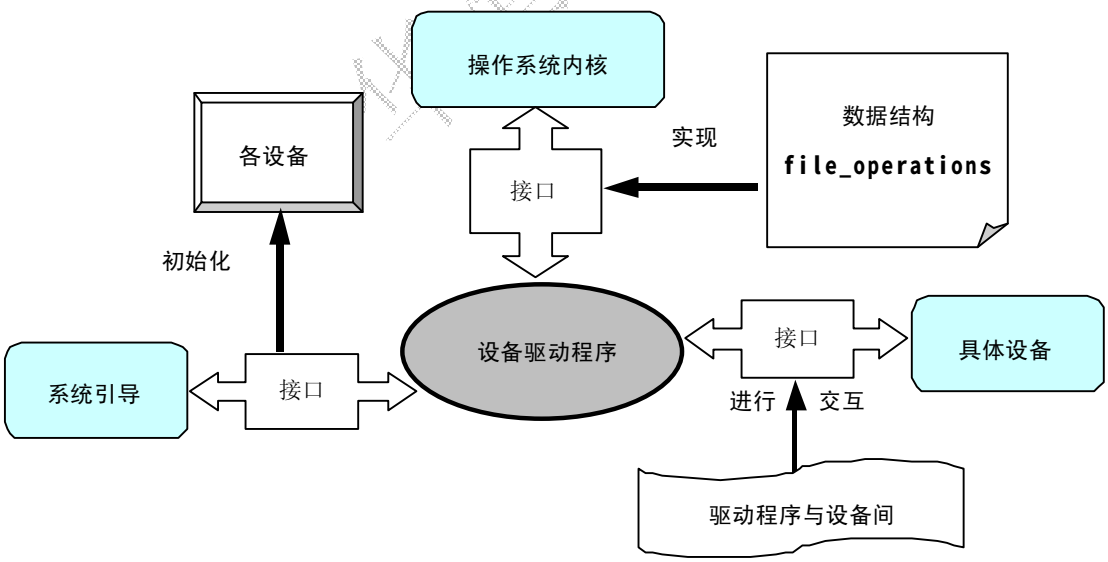

图 11.2 设备驱动程序与外界的接口

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 字符设备驱动编写

字符设备驱动编写流程

1．流程说明

在上一节中已经提到，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 rmmod 函数时被卸载，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 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。

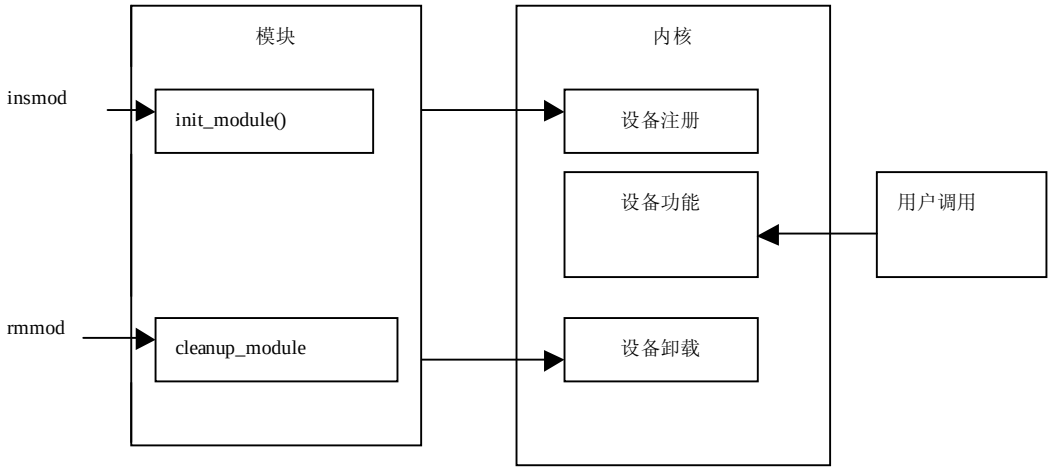

图 11.3　设备驱动程序流程图

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，它是一个在<linux/fs.h>中定义的 struct file 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读者在第 6 章中已经见到过了，当时是调用这些函数，而在这里我们将学习如何实现这些函数。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。

其中，struct inode 提供了关于设备文件/dev/driver（假设此设备名为 driver）的信息。struct file 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 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）设备注册

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。

register_chrdev 函数格式如表 11.1 所示。

表 11.1　　register_chrdev 等函数语法要点

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地分配一个主设备号
	name：设备名
	fops：对各个调用的入口点
函数返回值	成功：如果是动态分配主设备号，此返回所分配的主设备号。且设备名就会出现在/proc/devices 文件里
	出错：–1

（2）设备解除注册

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。

unregister_chrdev 函数格式如下表 11.2 所示：

表 11.2　　unregister_chrdev 等函数语法要点

所需头文件	#include <linux/fs.h>

函数原型	int unregister_chrdev(unsigned int major, const char *name)
函数传入值	major：设备的主设备号，必须和注册时的主设备号相同。
	name：设备名
函数返回值	成功：0，且设备名从/proc/devices 文件里消失。
	出错：–1

（3）打开设备

打开设备的接口函数是 open，根据设备的不同，open 函数完成的功能也有所不同，但通常情况下在 open 函数中要完成如下工作。

- 递增计数器。
- 检查特定设备的特殊情况。
- 初始化设备。
- 识别次设备号。

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较多次数，也可以由不同的进程所使用，所以若有一进程想要关闭该设备，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用在<linux/module.h>中定义的 3 个宏如下。

- MOD_INC_USE_COUNT：计数器加一。
- MOD_DEC_USE_COUNT：计数器减一。
- 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。

（4）释放设备

释放设备的接口函数是 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。

释放设备时要完成的工作如下。

- 递减计数器 MOD_DEC_USE_COUNT。
- 在最后一次释放设备操作时关闭设备。

（5）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 11.3 所示。

表 11.3　　read、write 函数语法要点

所需头文件	#include <linux/fs.h>
函数原型	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)
函数传入值	filp：文件指针
	buff：指向用户缓冲区
	count：传入的数据长度
	offp：用户在文件中的位置
函数返回值	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，其中之一就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。所以就不能使用诸如 memcpy 之类的函数来完成这样的操作。在这里就要使用 copy_to_user 或 copy_from_user 函数，它们就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 11.4 所示。

表 11.4　　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。

（6）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存可以有基于内存地址和基于页面为单位两类。其中，基于内存地址的函数有 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。

基于页为单位的内存有函数族有如下。

- get_zeroed_page：获得一个已清零页面。
- get_free_page：获得一个或几个连续页面。
- 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_pages 族。

表 11.5 给出了 kmalloc 函数的语法格式。

表 11.5 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
		_GFP_HIGHMEN：请求高端内存
函数返回值	成功：写入的数据长度 失败：-EFAULT	

表 11.6 给出了 kfree 函数的语法格式。

表 11.6 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 11.7 给出了基于页的分配函数 get_free_ page 族函数的语法格式。

表 11.7 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：写入的数据长度 失败：-EFAULT

表 11.8 给出了基于页的内存释放函数 free_ page 族函数的语法格式。

表 11.8　　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_page(unsigned long addr)
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：写入的数据长度 失败：-EFAULT

（7）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 11.9 给出了 printk 函数的语法格式。

表 11.9　　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
	…：如 printf 一样的格式说明	
函数返回值	成功：0 失败：–1	

这些不同优先级的信息可以输出到控制台上、/var/log/messages 里。其中，对输出给控制台的信息有一个特定的优先级 console_loglevel。若优先级小于这个整数值时，则消息才能显示到控制台上，否则，消息会显示在/var/log/messages 里。若不加任何优先级选项，则消息默认输出到/var/log/messages 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，获取有关进程的有用信息，在运行时通过改变内核参数改变设置。与其他文件系统不同，/proc 存在于内存之中而不是硬盘上。读者可以通过“ls”查看/proc 文件系统的内容。

表 11.10 列出了/proc 文件系统的主要目录内容。

表 11.10 /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	关于 CPU 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	mounts	加载的文件系统
filesystems	支持的文件系统	partitions	系统识别的分区表
interrupts	中断的使用	rtc	实时时钟
ioports	I/O 端口的使用	slabinfo Slab	池信息
kcore	内核核心印象	stat	全面统计状态表
kmsg	内核消息	swaps	对换空间的利用情况
ksyms	内核符号表	version	内核版本
loadavg	负载均衡	uptime	系统正常运行时间

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 11.11 所示。

表 11.11 /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，读者可以使用查看/proc/device 文件获得相关设备的主设备号。

11.3 LCD 驱动编写实例

11.3.1 LCD 工作原理

S3C2410LCD 控制器用于传输视频数据和产生必要的控制信号，如 VFRAME、VLINE、

VCLK、VM 等。除了控制信号，S3C2410 还有输出视频数据的端口 VD[23:0]，如图 11.4 所示。

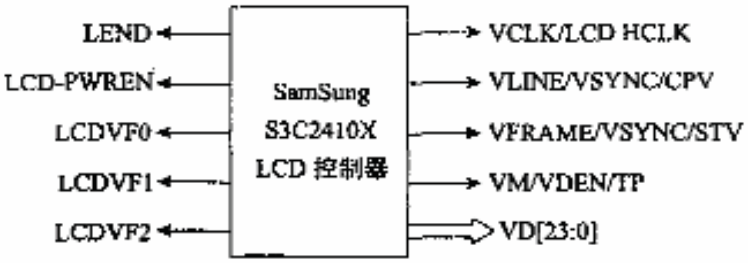

图 11.4　S3C2410 LCD 控制器

（1）寄存器介绍

LCD 的寄存器主要有：LCDCON1 寄存器、LCDCON2 寄存器、LCDCON3 寄存器、LCDC-ON4 寄存器和 LCDCON5 寄存器。

（2）控制流程

LCD 控制器由 REGBANK、LCDCDMA、VIDPRCS 和 LPC3600 组成（如图 11.5 所示）。REGBANK 有 17 个可编程寄存器组和 256*16 的调色板存储器，用来设定 LCD 控制器。LCDCDMA 是一个专用 DMA，自动从帧存储器传输视频数据到 LCD 控制器，用这个特殊的 DMA，视频数据可不经过 CPU 干涉就显示在屏幕上。IDPRCS 接受从 LCDCDMA 来的视频数据并在将其改变到合适数据格式后经 VD[23：0]将之送到 LCD 驱动器，如 4/8 单扫描或 4 双扫描显示模式。TIMEGEN 由可编程逻辑组成，以支持不同 LCD 驱动器的接口时序和速率的不同要求。TIMEGEN 产生 VFRAME、VLINE、VCLK、VM 信号等。

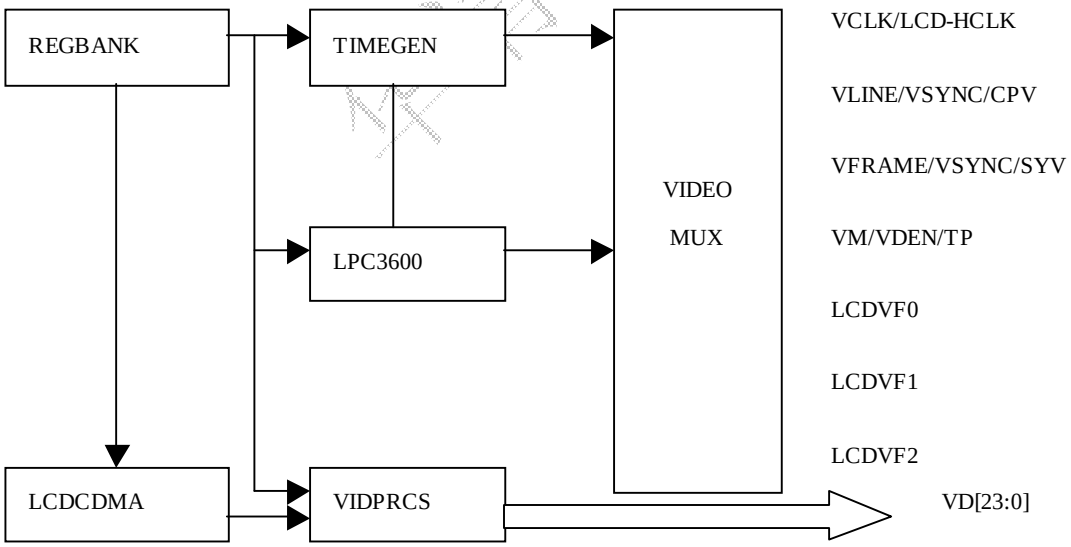

图 11.5　S3C2410 LCD 控制器内部方框图

（3）数据流描述

FIFO 存储器位于 LCDCDMA。当 FIFO 空或部分空时，LCDCDMA 要求从基于突发传输模式的帧存储器中取来数据，存入要显示的图像数据，而这个帧存储器是 LCD 控制器在 RAM 中开辟的一片缓冲区。当这个传送请求被存储控制器中的总线仲裁器接收到后，从系统存储器到内部 FIFO

就会成功传送 4 个字。FIFO 的总大小是 28 个字，其中低位 FIFOL 是 12 个字，高位 FIFOH 是 16 个字。S3C2410 有两个 FIFO 来支持双扫描显示模式。在单扫描模式下，只使用一个 FIFO（FIFOH）。

（4）TFT 控制器操作

S3C2410 支持 STN-LCD 和 TFT-LCD。TIMEGEN 产生 LCD 驱动器的控制信号，如 VSYNC、HSYNC、VCLK、VDEN 和 LEND 等。这些控制信号与 REGBANK 寄存器组中的 LCDCONl/2/3/4/5 寄存器的配置关系相当密切，基于 LCD 控制寄存器中的这些可编程配置，TIMEGEN 产生可编程控制信号来支持不同类型的 LCD 驱动器。

VSYNC 和 HSYNC 脉冲的产生依赖于 LCDCON2/3 寄存器的 HOZVAl 域和 LINEVAL 域的配置。HOZVAL 和 LINEVAL 的值由 LCD 屏的尺寸决定，如下公式：

HOZVAL=水平显示尺寸–1 （1）

LINEVAL=垂直显示尺寸–1 （2）

VCLK 信号的频率取决于 LCDCONl 寄存器中的 CLKVAL 域。VCLK 和 CLKVAL 的关系如下，其中 CLKVAl 的最小值是 0：

VCLK(Hz)＝HCLK/L(CLKVAL+1)×2] （3）

帧频率是 VSYNC 信号的频率，它与 LCDCONl 和 LCDCON2/3/4 寄存器的 VSYNC、VD-PD、VFPD、LINEVAL、HSYNC、HBPD、HFPD、HOZVAL 和 CLKVAL 都有关系。大多数 LCD 驱动器都需要与显示器相匹配的帧频率，帧频率计算公式如下：

FrameRate=1/{[(VSPW+1)+(VBPD+1)+(LINEVAL+1)+(VFPD+1)]*[(HSPW+1)+(HBPD+1)+(HFPD+l)+(HOZVAL+1)*[2*(CLKVAL+1)/(HCLK)]}

11.3.2 LCD 驱动实例

LCD 驱动代码如下所示：

```
#include <linux/kernel.h>
#include <linux/init. h>
#include <linux/module. h>
#include <linuxJfs. h>
#include <linux/delay.h>
#include <asm/fcntl.h>
#include <asm/unistd.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include "lcdexp.h"
static unsigned char*, lcd base;
/* LCD 配置函数 */
static void setup_lcd(void)
{
/*在设置 LCD 寄存器之前关闭 LCD*/
     LCDEN[12] = 0;
```

```
        SYSCON1 &= ~0x00001000;
    /* 设置 LCD 控制寄存器
        * Video Buffer Size[0:12]: 320'240'12 / 128 = 0xlclf
        * Line Length[13:18]: 320 / 16 -1 = 0x13
        * Pixel Prescale[19:24]: 0x01
        * ACPmscale[25:29]: 0x13
        * GSEN[30]: =1,Enables gray scale output to LCD
        * GSMD[31]: =1,4 bpp ( 16-gray scale )
        */
        LCDCON = 0xe60f7clf;
        /* 设置 LCD Palette 寄存器 */
        PALLSW = 0x76543210;
        PALMSW = 0xfedcba98;
        /*
        * 设置 LCD frame buffer Sets 的起始位置
        * 这样, frame buffer 就从 0xc0000000 起始
        */
        FBADDR = 0xc;
    /*使能 LCD 使之改变 LCD 的配置*/
        LCDEN[12] = 1;
        SYSCON1 = 0x00001000;
        return;
    }

    /*在 LCD 中画一个点
     * x,y: 亮点的坐标
     * color:点的颜色
     */
    static void lcd__pixel_set(int x, int y, COLOR color)
    {
          unsigned char* fb_ptr;
          COLOR pure_color = OxO000;
          /* if the dot is out of the LCD, return */
          If (x<0 II x>=32011 y<0 II y>=240){
    /*计算点的地址 */
          fb_.ptr =  lcd base + (x+y*320)*12/8;
          /*把版面上的点映射到帧缓冲中（frame buffer）*/
          if (x & 0xl ) (
                pure_color = ( color & 0x000f ) < < 4;
```

```
                *fb_ptr &= 0x0f;
                *fb_ptr I= pure_color;
                pure_color = ( color & 0x0ff0 ) >> 4;
                *(fb_ptr+l) = 0xff & pure_color;
        } else {
                pure_color = color & 0x00ff;
                *fb_ptr = 0xff & pure_color;
                pure_color = (color & 0x0f00 ) >> 8;
                *(fb_ptr+l) &= OxfO;
                *(fb_ptr+l) |= pure_color;
                }
        return;
}
/*
        把所有 LCD 图片清零
*/
void clear_lcd(void)
{
        int x;
        int y;
        for (y=0;y<240; y++) {
              for (x=0; x<320; x++) {
                    //lcd_disp.x = x;
                    //lcd_disp.y = y;
                    Lcd_plxel_set(x,y,0x0000);
               }
        }
        Return;
}

/* (start x, start_y): 矩形最左边的坐标
 * (end_x, end_y): 矩形最右边的坐标
 */
static void draw_rectangle(int start_x,int start_y,int end_x,int end_y,COLOR
color)
{
        draw__vline(start_x, start_y, end_y, color);
        draw_vline(end_x, start_y, end_y, color);
        draw_hline(start_x, end_x, start_y, color),
```

```
        draw_hline(start_x, end_x, end_y, color);
        return;
    }
    /*
     * (start x, start_y):矩形最上边的坐标
     * (end_x, end_y): 矩形最下边的坐标
     */
    static void draw_full_rectangle(int start_x, int start_y,int end_x,int
end_y,COLOR color)
    {
        int i = 0;
        int tmp= 0;
        tmp= end_x - start_x;
        for ( i=0;i<tmp;++i ) {
            draw_vline(start_x+i, start_y,end_y, color);
        }
        return;
    }
    /*显示一个 ascii 符号
     * x,y: 符号起始坐标
     * codes: 要显示的字节数
    */
    static void write_en(int x, int y, unsigned char* codes, COLOR color)
    {
        inti = 0;
        /* total 16 bytes codes*/
        for (i=0;i<16;++i) {
            intj = 0;
        x += 8;
        for ( j=0;j<8;++j ){
            --x;
            if ((codes[i]>>j) 8, 0x1 ) {
                lcd_pixel_set(x, y, color);
            }
        }
        /*移到下一行，x 轴不变，y 轴加一*/
        ++y;
        }
        return;
```

```
    }
    /*显示一个中文字符
     * x,y: 字符起始点
     * codes: 要显示的字节数
     * color: 要显示的字符颜色
     */
    static void write_cn(int x, iht y, unsigned char* codes, COLOR color)
    {
           int i;
           /* total 2'16 bytes codes */
           for(i=0;i< 16;i++) {
                 int j = 0;
                 for (j=0;j<2;++j) {
                       int k = 0;
                       x += 8(j+1);
                       for ( k=O;k<8;++k ){
                             --x;
                             if ( ( codes[2*i+j] >> k) &0xl ) {
                                    Icd_pixel_set(x,y,color);
                             }
                       }
                 }
                 x-= 8;
                ++y;
           }
           return;
    }
    static int lcdexp_open(struct inode *node, struct file *file)
    {
           return 0;
    }
    static int lcdexp_read(struct file *file, char *buff, size_t count, Ioff_t *offp)
    {
           return 0;
    }
    static int lcdexp_write(struct file *file, const char *buff, size_t count,
Ioff_t *offp)
    {
           return 0;
```

```
    }
    /*lcd ioctl 驱动函数，分类处理 lcd 的各项动作，在每种情况下都会调用前述的关键函数*/
    static int lcdex_ioctl(struct inode *inode, struct file *file, unsigned int
cmd, unsigned long
    arg)
    {
           switch ( cmd ) {
           case LCD_Clear:/*lcd 清屏*/
           {
                 clear_lcd();
                 break;
           }
           case LCD_Pixel_Set /*lcd 象素设置*/
           {
                 struct lcd_display    pixel_display;
                 if(copy_from_user(&pixel_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))     {
                       printk("copy_frorn_user error!\n"),
                       return -1;
                 }
                 lcd_pixel_set(pixeLdisplay.xl, pixel_display.yl, pixel_display.
color);
                 break;
           }
           case LCD_Big_Pixel_Set:/*lcd 高级象素设置*/
           {
                struct lcd_display   b_pixel_display;
                if(copy_from_user(&b_pixel_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))                {
                printk("copy_from_user error!\n");
                return -1;
           }
           lcd_big_pixel_set(b_pixel_display.xl, b_pixel display.y1, b_pixel_display.
color);
           break;
        }
        case LCD_Draw_Vline:/*lcd 中显示水平线*/
        {
              struct lcd_display    vline_display;
```

```
             if(copy_from_user(&vline_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))
             {
                  printk("copy_from_user error!\n");
                  return -1;
             }
             draw_vline(vline_display.xl, vline_display.yl, vline_display.y2,
vline_display.color);
        }
        case LCD_Draw_HLine:/*lcd 中显示垂直线*/
        {
             struct lcddisplay    hline_display;
             if ( copy_from_user(Ehline_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))     {
                   printk("copy_from_user error!\n");
                   return -1;
             }
        draw_hline(hline_display.xl,   hline   display.x2,   hline_display.yl,
hline_display.color);
        break;
        }
        Case LCD_Draw_Vdashed:/*lcd 中显示水平随意线*/
        {
             struct lcd-_display    vdashed display;
             if(copy_from_user(&vdashed_display,(structlcd_display*)arg,sizeof
(struct lcd_display)))     {
                  printk("copy_from_user error!\n");
                  return -1;
             }
             draw hdashed(hdashed-display.xl, hdashed_display.x2, hdashed_
display.yl,
    vdashed_display.color);
             break;
        }
        Case LCD_Draw_HDdashed:/*lcd 中显示垂直随意线*/
        {
             struct lcd_display    hdashed display;
             if(copy_from_user(&hdashed_display,(structlcd_display*)arg,sizeof
(struct lcd_display)))     {
```

```
                printk("copy_from_user error!\n");
                return -1;
            }
            draw hdashed(hdashed-display.xl, hdashed_display.x2, hdashed_
display.yl,
    vdashed_display.color);
            break;
        }
        case LCD_Draw_Rectangle:/*lcd 中显示矩阵*/
        {
            struct/cd-display    rect display;
             if ( copy_from_user(&rect_display,(struct Icd_display*)arg,sizeof
(struct lcd_display))) {
                   printk("copy_from_user error!\n");
                   return -1;
             }
             draw_rectangle(rect_display.xl,rect_display.yl,rect_display.x2,
rect_display.y2,rect_display.color);
        break;
        }
        case LCD_Draw_Full_Rectangle:/*lcd 中显示填充矩阵*/
        {
            Struct xlcd_display    frect_display;
            if ( copy_from_user(&frect_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))                {
                 printk("copy_from_user error!\n");
            }
            draw_full_rectangle(frect_display.xl,
frect_display.yl,frect_display.x2,frect_display.y2, rect_display.color);
            break;
        }
        case LCD Write_EN:/*lcd 英文显示*/
        {
            Struct lcd_display    en_display;
             if ( copy_from_user(&en_display,(struct Icd_display*)arg,sizeof
(struct lcd_display)))         {
                   printk("copy_from_user error!\n");
             }
             return -1;
```

```
            write_en(en_display.xl, en_.display.yl, en_display.buf, en_display.
color);
            break;
        }
        case LCD Write_CN:/*lcd 中文显示*/
        {
            struct lcd_display    cn_display;
            if ( copy_from_user(&cn_display,(struct Icd_display*)arg,sizeof
(struct lcd_display))) {
                printk("copy_ffom_user errod\n");
                return -1;
            }
            write_cn(cn_display.xl, cn_display.yl, cn_display.buf, cn_display.
color);
            break;
    }
        default:
            printk("unknown cmd\n");
            break;
    return 0;
    }

    static struct file_operations lcdexp_fops = {
          open:             lcdexp_open,
          read:              lcdexp_read,
          ioctl:            lcdexp_joctl,
          write:             lcdexp_write,
          release:          lcdexp_release,
    };
    int lcdexp_init(void)
    {
         int result;
         lcd base = (unsigned char*)0xc0000000;
         result = register_chrdev(DEV_MA)OR,"lcdexp",&lcdexp_fops);
         if ( result < 0 ) {
              printk( KERN_INFO "lcdexp:register Icdexp failed !\n'
              return result;
         setup_lcd();
         for ( i=0;i<320*240*12/8;i++ )
```

```
            lcd base++ = 0x77;
        _lcd_base = (unsigned char*)0xc0000000;
        printk("LCD  .............  support.\n");
return 0;
}
static void _exit lcdexp_exit(void)
{
        /* clear LCD */
        unregister_chrdev(DEV_MA.lOR,"lcdexp");
}
module_init(lcdexp_init);
module_exit(lcclexp_exit);
```

11.4 块设备驱动编写

11.4.1 块设备驱动程序描述符

块设备文件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。块设备驱动程序描述符是一个包含在<linux/blkdev.h>中的 blk_dev_struct 类型的数据结构，其定义如下所示：

```
struct blk_dev_struct {
        request_queue_t  request_queue;
        queue_proc   *queue;
        void  *date;
};
```

在这个结构中，请求队列 request_queue 是主体，包含了初始化之后的 I/O 请求队列。对于函数指针 queue，当其为非 0 时，就调用这个函数来找到具体设备的请求队列，这是为考虑具有同一主设备号的多种同类设备而设的一个域，该指针也在初始化时就设置好。指针 data 是辅助 queue 函数找到特定设备的请求队列，保存一些私有的数据。

所有块设备的描述符都存放在 blk_dev 表 struct blk_dev_struct blk_dev[MAX_BLKDEV]中；每个块设备都对应着数组中的一项，可以使用主设备号进行检索。每当用户进程对一个块设备发出一个读写请求时，首先调用块设备所公用的函数 generic_file_read()和 generic_file_write()。如果数据存在在缓冲区中或缓冲区还可以存放数据，那么就同缓冲区进行数据交换。否则，系统会将相应的请求队列结构添加到其对应项的 blk_dev_struct 中，如下图 11.6 所示。

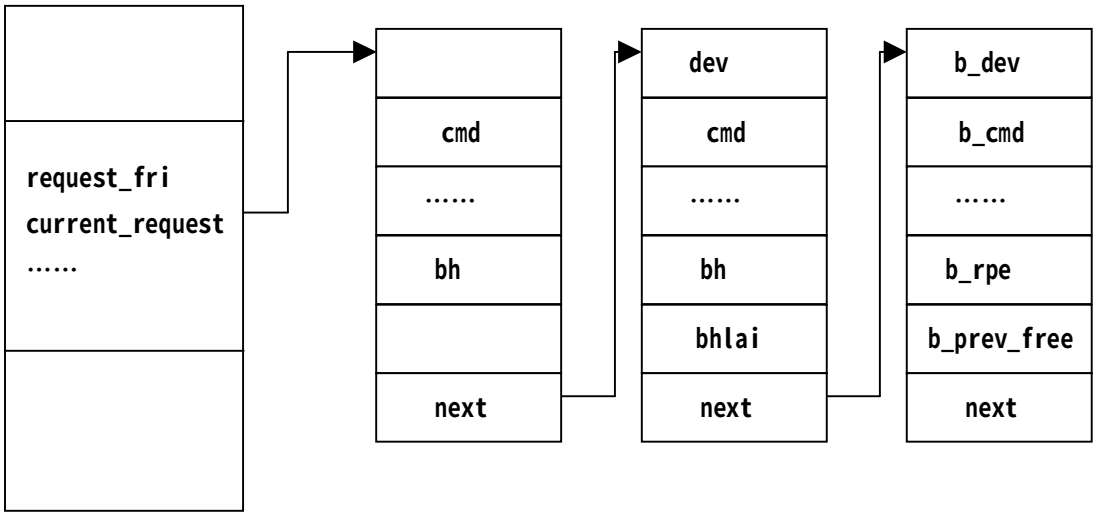

图 11.6　块设备请求队列

11.4.2　块设备驱动编写流程

1．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，用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，下图 11.7 给出了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。

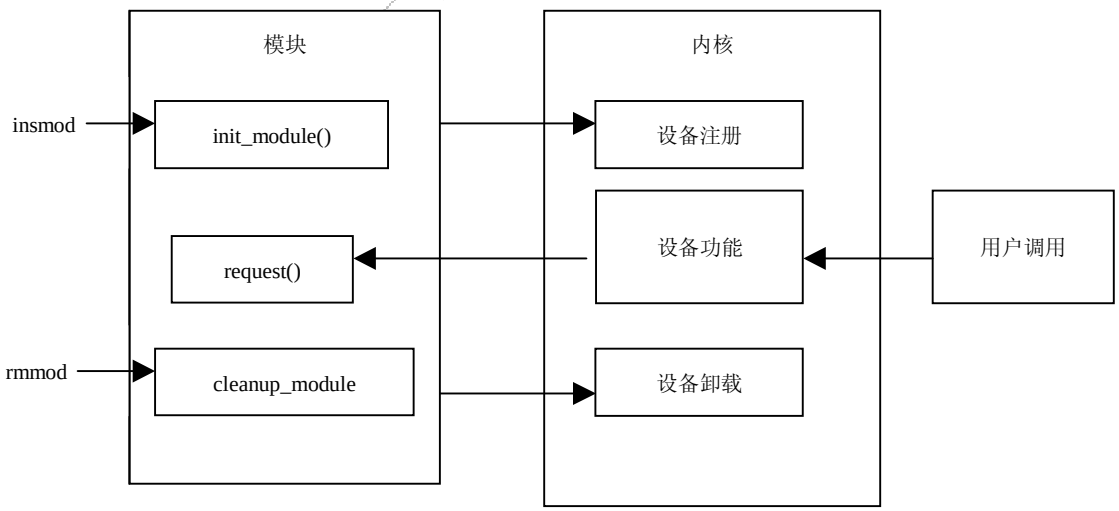

图 11.7　块设备驱动程序流程图

2．重要数据结构

由于块设备驱动程序的绝大部分都与设备无关的，故内核的开发者通过把大部分相同的代码放在一个头文件<linux/blk.h>中来简化驱动程序的代码。从而每个块设备驱动程序都必须包含这个头文件。先给出块设备驱动程序要用到的数据结构定义：

```
struct device_struct {
       const char *name;
       struct file_operations *chops;
};
static struct device_struct blkdevs[MAX_BLKDEV];
struct sbull_dev {
       void **data;
       int quantum; // the current quantum size
       int qset; // the current array size
       unsigned long size;
       unsigned int access_key; // used by sbulluid and sbullpriv
       unsigned int usage; // lock the device while using it
       unsigned int new_msg;
       struct sbull_dev *next; // next listitem
};
```

与字符设备驱动程序一样，块设备驱动程序也包含一个 file_operation 结构，其结构定义一般如下所示：

```
struct file_operation blk_fops = {
       NULL，//seek
       block_read，//内核函数
       block_write，//内核函数
       NULL，//readdir
       NULL，//poll
       sbull_ioctl，// ioctl
       NULL，//mmap
       sbull_open，//open
       NULL，//flush
       sbull_release，//release
       block_fsync，//内核函数
       NULL，//fasync
       sbull_check_media_change，//check media change
       NULL，//revalidate
       NULL，//lock
```

```
};
```

从上面结构中可以看出，所有的块驱动程序都调用内核函数 block_read()、block_write()，block_fsync()函数，所以在块设备驱动程序入口中不包含这些函数，只需包括 ioctl()、open()和 release()函数即可。

（1）设备初始化

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引导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。

块设备驱动程序初始化的工作主要包括：

- 检查硬件是否存在；
- 登记主设备号；
- 将 fops 结构的指针传递给内核；
- 利用 register_blkdev()函数对设备进行注册：

```
if(register_blkdev(sbull_MAJOR, “sbull”, &sbull_fops)) {
      printk( “Registering block device major: %d failed\n”, sbull_MAJOR);
      return-EIO;
};
```

- 将 request()函数的地址传递给内核：

```
blk_dev[sbull_MAJOR].request_fn = DEVICE_REQUEST;
```

- 将块设备驱动程序的数据容量传递给缓冲区：

```
#define  sbull_HARDS_SIZE  512
#define  sbull_BLOCK_SIZE  1024
static  int  sbull_hard = sbull_HARDS_SIZE;
static  int  sbull_soft = sbull_BLOCK_SIZE;
hardsect_size[sbull_MAJOR] = &sbull_hard;
blksize_size[sbull_MAJOR] = &sbull_soft;
```

在块设备驱动程序内核编译时，应把下列宏加到 blk.h 文件中：

```
#define  MAJOR_NR  sbull_MAJOR
#define  DEVICE_NAME  “sbull”
#define  DEVICE_REQUEST  sbull_request
#define  DEVICE_NR(device)  (MINOR(device))
#define  DEVICE_ON(device)
#define  DEVICE_OFF(device)
```

（2）request 操作

Request 操作涉及一个重要的数据结构如下。

```
struct request {
      kdev_t rq_dev;
```

```
        int cmd;    // 读或写
        int errors;
        unsigned long sector;
        char *buffer;
        struct request *next;
};
```

对于具体的块设备，函数指针 request_fn 当然是不同的。块设备的读写操作都是由 request()函数完成。所有的读写请求都存储在 request 结构的链表中。request()函数利用 CURRENT 宏检查当前的请求：

```
#define CURRENT (blk_dev[MAJOR_NR].current_request)
接下来看一看 sbull_request 的具体使用:
extern struct request *CURRENT;
void sbull_request(void) {
     unsigned long offset, total;
Begin:
     INIT_REQUEST:
          offset = CURRENT -> sector * sbull_hard;
          total = CURRENT -> current_nr_sectors * sbull_hard;
/*超出设备的边界*/
    if(total + offset > sbull_size * 1024) {
/*请求错误*/
         end_request(0);
         goto Begin;
     }
     if(CURRENT -> cmd == READ) {
          memcpy(CURRENT -> buffer, sbull_storage + offset, total);
     }
     else if(CURRENT -> cmd == WRITE) {
          memcpy(sbull_storage + offset, CURRENT -> buffer, total);
          }
          else {
               end_request(0);
          }
/*成功*/
     end_request(1);
/*当请求做完时让 INIT_REQUEST 返回*/
     goto Begin;
}
```

request()函数从 INIT_REQUEST 宏命令开始（它也在 blk.h 中定义），它对请求队列进行检查，保证请求队列中至少有一个请求在等待处理。如果没有请求（即 CURRENT = 0），则 INIT_REQUEST 宏命令将使 request()函数返回，任务结束。

假定队列中至少有一个请求，request()函数现在应处理队列中的第一个请求，当处理完请求后，request()函数将调用 end_request()函数。如果成功地完成了读写操作，那么应该用参数值 1 调用 end_request()函数；如果读写操作不成功，那么以参数值 0 调用 end_request()函数。如果队列中还有其他请求，那么将 CURRENT 指针设为指向下一个请求。执行 end_request()函数后，request()函数回到循环的起点，对下一个请求重复上面的处理过程。

（3）打开操作

打开操作要完成的流程图如下图 11.8 所示。

典型实现代码如下所示：

```
int sbull_open(struct inode *inode, struct file *filp) {
      int num = MINOR(inode -> i_rdev);
      if(num >= sbull -> size)
           return   ENODEV;
      sbull -> size = sbull -> size + num;
      if(!sbull -> usage) {
            check_disk_change(inode -> i_rdev);
            if(!*(sbull -> data))
                  return -ENOMEM;
            }
      sbull -> usage++;
      MOD_INC_USE_COUNT;
      return 0;
}
```

（4）释放设备操作

释放设备操作要完成的流程图如图 11.9 所示。

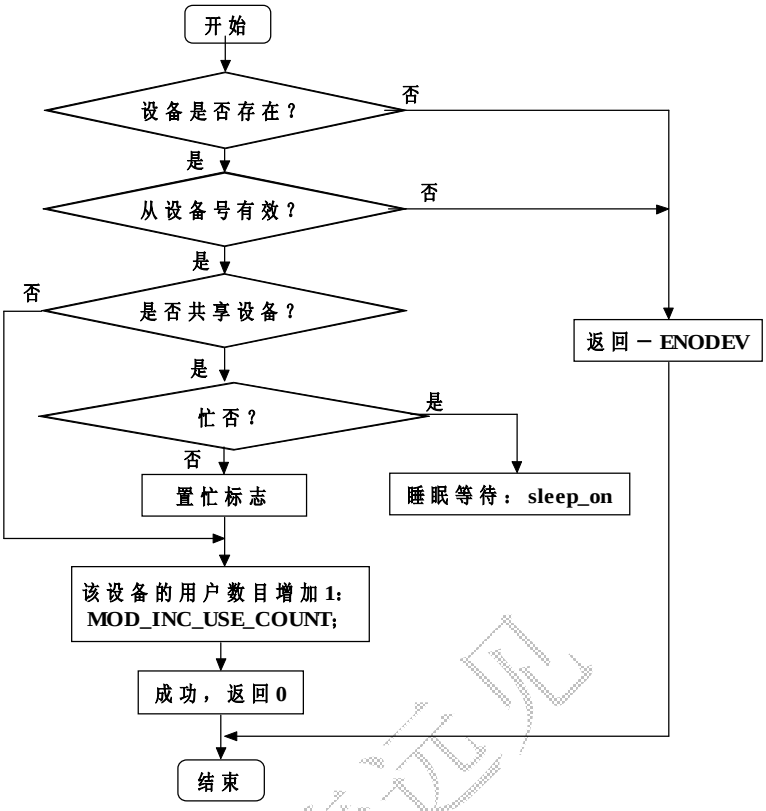

图 11.8　块设备打开操作流程图

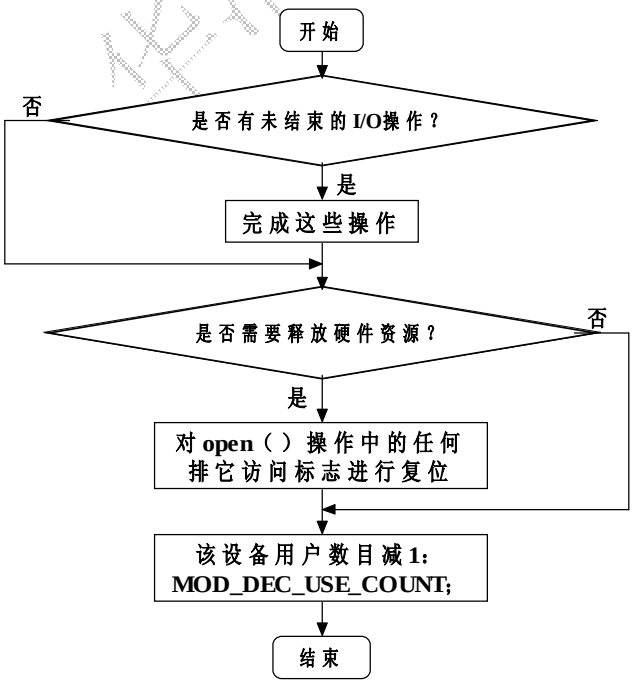

图 11.9 释放设备操作流程图

典型实现代码如下所示：

```
void sbull_release(struct inode *inode, struct file *filp) {
      sbull -> size = sbull -> size + MINOR(inode -> i_rdev);
      sbull -> usage--;
      MOD_DEC_USE_COUNT;
      printk("This blkdev is in release!\n");
      return 0;
}
```

（5）ioctl 操作

ioctl 操作要完成的流程图如图 11.10 所示。

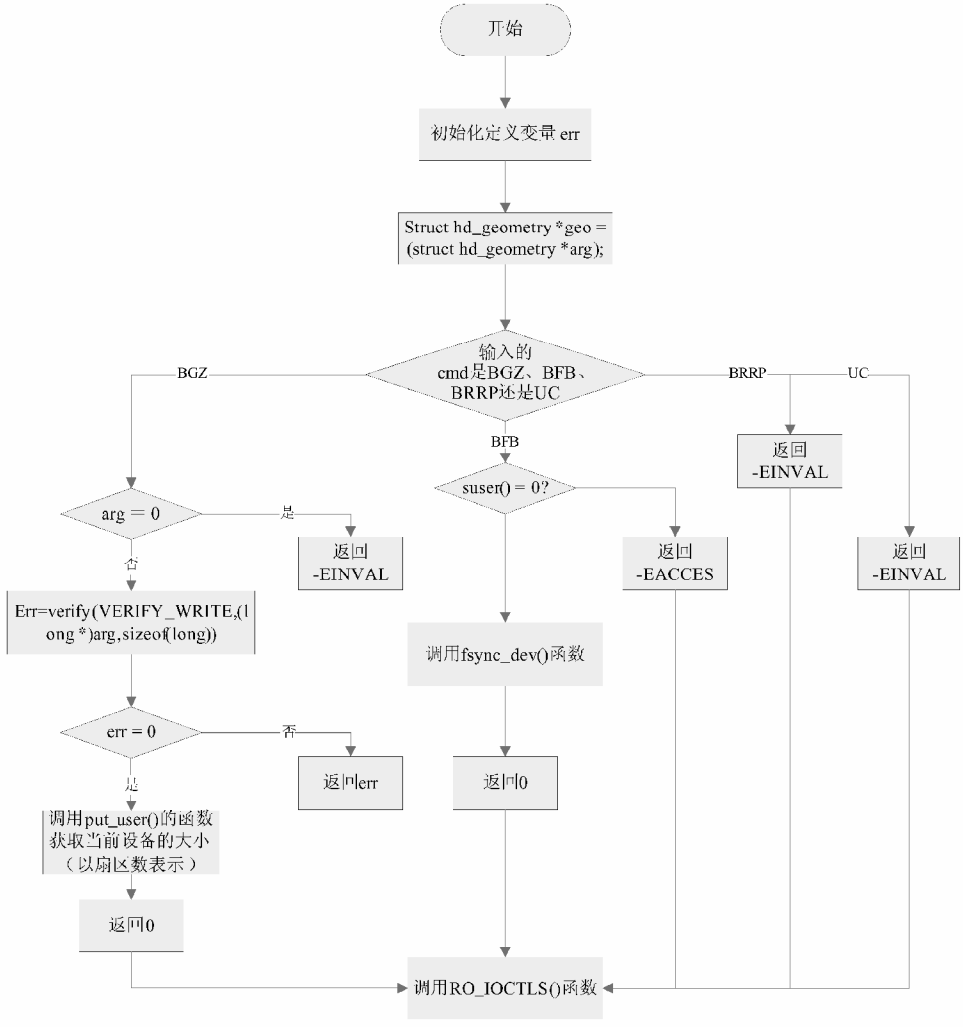

图 11.10 ioctl 操作要完成的流程图

其典型实现代码如下所示：

```
    #include <linux/ioctl.h>
    #include <linux/fs.h>
    int  sbull_ioctl(struct inode *inode，struct file *filp，unsigned int cmd，
unsigned long arg) {
    int err;
    struct hd_geometry *geo = (struct hd_geometry *)arg;
    PDEBUG("ioctl  0x%x  0x%lx\n"，cmd，arg);
    switch(cmd) {
          case  BLKGETSIZE:
    / *返回设备大小*/
          if(!arg)
                return –EINVAL;   // NULL pointer: not valid
          err = verify_area(VERIFY_WRITE，(long *)arg，sizeof(long));
          if(err)
          return err;
          put_user(1024*sbull_sizes[MINOR(inode -> i_rdev)/sbull_hardsects
[MINOR(inode -> i_rdev)]，(long*)arg];
          return 0;
          case  BLKFLSBUF: // flush
          if(!suser())
                return –EACCES;   // only root
          fsync_dev(inode -> i_rdev);
                return 0;
          case  BLKRRPART:   // re-read partition table: can't do it
               return –EINVAL;
          RO_IOCTLS(inode -> i_rdev，arg);
    // the default RO operations，宏 RO_IOCTLS(kdev_t dev，unsigned long where)
在 blk.h 中定义
          }
    return –EINVAL;   // unknown command
    }
```

11.5　中断编程

前面所讲述的驱动程序中都没有涉及到中断处理，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。

这是驱动程序申请中断和释放中断的调用。在 include/linux/sched.h 里声明。

request_irq()调用的定义：

```
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);
```

irq 是要申请的硬件中断号。在 Intel 平台，范围是 0～15。handler 是向系统登记的中断处理函数。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，device id，寄存器值。dev_id 就是下面的 request_irq 时传递给系统的参数 dev_id。irqflags 是中断处理的一些属性。比较重要的有 SA_INTERRUPT，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。慢速处理程序不屏蔽。还有一个 SA_SHIRQ 属性，设置了以后运行多个设备共享中断。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。void free_irq(unsigned int irq,void *dev_id);

11.6 键盘驱动实现

11.6.1 键盘工作原理

1．原理简介

在键盘产生按键动作之后，键盘上的扫描芯片（一般为 8048）获得键盘的扫描码，并将其发送到主机端。在主机端的处理过程为是端口读取扫描码之后，对键盘模式作一个判断，如果是 RAW 模式，则直接将键盘扫描码发送给应用程序；如果是其他模式，则就将扫描码转化成为键盘码，然后再判断模式以决定是否将键盘码直接发送给应用程序；如果是 XLATE 或 Unicod。模式，则将键盘码再次转化成为符号码，然后根据对符号码解析，获得相应的处理函数，并将其送到 TY 设备的缓存中。模式判断的对应关系如图 11.11 所示。

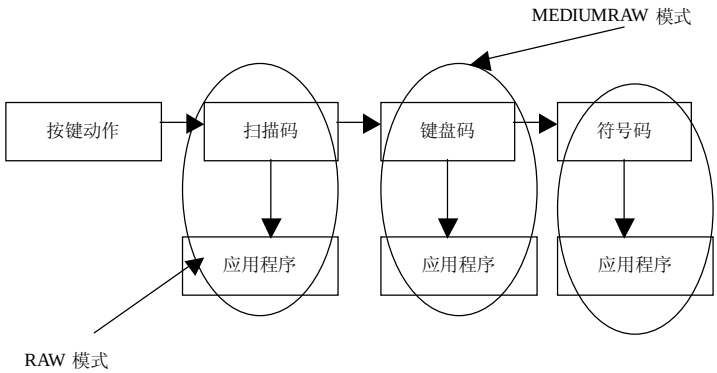

图 11.11 模式判断的对应关系

键盘模式有 4 种，这 4 种模式的对应关系如图 1 所示。

- Scancode mode（RAW）模式：将键盘端口上读出的扫描码放入缓冲区，通过参数 s 可以设置。
- Keycode mode（MEDIUMRAW）模式：将扫描码过滤为键盘码放入缓冲区，通过参数 k 可以设置。
- ASCII mode（XLATE）模式：识别各种键盘码的组合，转换为 TTY 终端代码放入缓冲区，通过参数 a 可以设置。
- UTF-8 mode（Unicode）模式：Unicode 模式基本上与 XLATE 相同，只不过可以通过数字小键盘 I 旬接枪入 Unicode 代码，通过参数 u 可以设置。

2．扫描码

一个基本按键的扫描码由 3 个字节组成：1 个字节的接通扫描码和 2 个字节的断开扫描码。其中第 1 和第 2 个字节相同，中间字节是断开标志 FOH。例如 B 键的接通扫描码是 32H，断开扫描码是 FOH 32H，B 键被按下时，32H 被发送出去，如果移植按住不放，则键盘将以按键重复率不停地发送 32H，直到该键释放，才发出断开扫描码 FOH 32H。扫描码与按键的位置有关，与该键的 ASCII 码并无对应关系。键盘上还有部分扩展键（功能键和控制键等），这些键的扫描码由 5 个字节组成，与基本键的扫描码相比，接通扫描码与断开扫描码前各多了一个固定值字节 EOH。例如 Home 键的接通扫描码是 EOH 70H，断开扫描码是 EOH FOH 70H。还有两个特殊键，PrintScreen 键的接通扫描码是 EOH 12H EOH 7CH；断开扫描码是 EOH FOH 7CH EOH FOH 77H，无断开扫描树。

3．键盘码

由前面的分析可见，单单一个键的按下与断开，键盘最多要产生一系列多达 6 个字节的扫描码序列，而内核必须解析扫描码序列从而定位某个键被按下与释放的事件。为达到这个目的，每一个键被分配一个键盘码 k（k 的范围 1-127）0 如果按键按下产生的键盘码为 k，则释放该键产生的键盘码为 k+128。按照键盘码的分配规则，对于产生单个扫描码范围 OxOl～Ox58 的键，其键盘码与扫描码相同。而对于 0x59～0x71 范围的键，可以查表获得其扫描码与键盘码对应。

4．符号码

符号码（keysym）最终是用来标志一个按键事件的惟一值；根据上面的分析，它由键盘码经过 Keymap 表映射而来。它包括 2 个字节，高 8 位表示 type，根据 type 的不同，我们最终选择不同的处理函数来处理不同类型的事件，type 相同的事件由同一个函数来处理。Type 包括一般键、方向键、字母键、函数键等。在<linux/keyboard.h>中可以找到 13 种键类型的宏定义。

5．Keymap 表

在使用键盘时常常使用组合键，而组合键的意义通常是系统另外赋予的，所以从键盘码向 TY 输入符的转换需要借助 Keymap 表来索引。

```
intshift_ final = shift state ^ kbd-Aockstate;
ushort'key map=key maps[shift finall];
```

```
keysym=key_map[keycodel];
```

由于共计有 8 个修饰符（modifier），即 Shift，AitGr，Control，Alt，ShiftL，ShiftR，CtrIL 和 CtrlR，因此共有 256 张可能的 Keymap，而在实际使用时，内核缺省只分配 7 张 Keymap:plain, Shift, AltR, Ctrl, Ctrl+Shift, AItL 和 Ctrl+AitL。

Keymap 表是一张二维表，结构图如图 11.12 所示。通过这张 Keymap 表，就能完成键盘码到符号码的转化，获得相应的符号码。

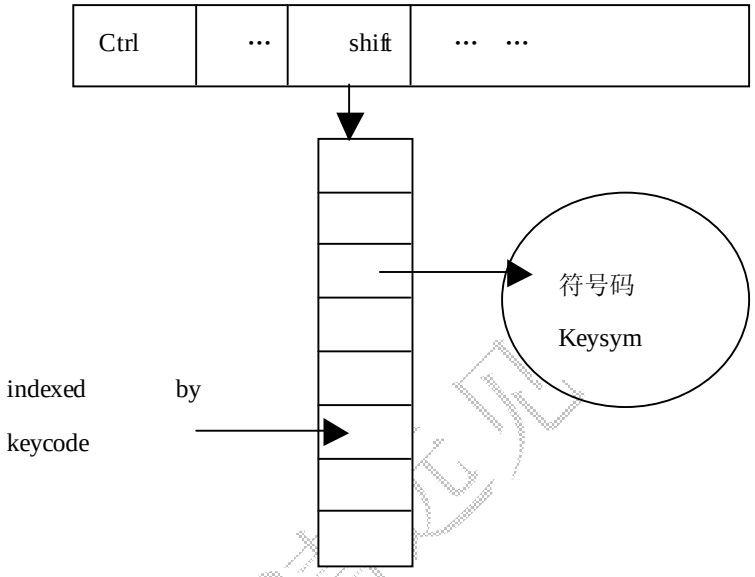

图 11.12 二维表结构图

11.6.2 键盘驱动综述

Linux 中的大多数驱动程序都采用了层次型的体系结构，键盘驱动程序也不例外。在 Linux 中，键盘驱动被划分成两层来实现。其中，上层是一个通用的键盘抽象层，完成键盘驱动中不依赖于底层具体硬件的一些功能，并且负责为底层提供服务；下层则是硬件处理层，与具体硬件密切相关，主要负责对硬件进行直接操作。键盘驱动程序的上层公共部分都在 driver/keyboard.c 中。在 keyboard.c 中，不涉及底层操作，也不涉及到任何体系结构，主要负责键盘初始化、键盘 tasklet 的挂入、按键盘后的处理、Keymap 表的装入、Scancode 的转化、与 TTY 设备的通信。

在 pc_keyb.c 中，主要负责一些底层操作，跟具体的体系结构相关，它完成的功能有：键盘的 I/O 端口和中断号的分配，键盘的硬件初始化，扫描码到键盘码的转化，键盘中断处理。

11.6.3 键盘驱动流程

（1）初始化

kbd_init()函数是键盘代码执行的入口点。kbd_init()在对键盘的工作模式及其他参数进行配置后，调用 kbd_init hw()函数。对于上层来说，此函数是一个统一的接口，对于不同体系结构或同体系下的不同开发板，它们的 kbd_init_hw()的实现代码是不同的（通过

CONFIG_ARCHXXX 的值来确定），它就是进行键盘的硬件初始化功能。然后将 keyboard tasklet 加入到 tasklet 链表中。至此键盘驱动的初始化工作已经完成。

键盘驱动的初始化代码是 keyboard.c 中的 kbd_init，其源码及分析如下所示：

```
int __init kbd_init(void)
{
      int i;
      struct kbd_struct kbd0;
/*维护 tty/console 对象，承担 tty 对外的输入和输出*/
      extern struct tty_driver console_driver;
/*缺省不亮灯*/
      kbd0.ledflagstate = kbd0.default_ledflagstate = KBD_DEFLEDS;
/*用于显示 flag*/
      kbd0.ledmode = LED_SHOW_FLAGS;
/*表示用 key_map 的第一个表，没有 lock 键*/
      kbd0.lockstate = KBD_DEFLOCK;
/*没有粘键*/
      kbd0.slockstate = 0;
      kbd0.modeflags = KBD_DEFMODE;
      kbd0.kbdmode = VC_XLATE;
 /*为每个控制台分配一个 KBD 结构*/
      for (i = 0 ; i < MAX_NR_CONSOLES ; i++)
            kbd_table[i] = kbd0;
/*维护当前各个控制台的 tty_struct 表*/
      ttytab = console_driver.table;
      kbd_init_hw();
/*把 keyboard_tasklet 挂到 CPU 的运行队列中去*/
      tasklet_enable(&keyboard_tasklet);
      tasklet_schedule(&keyboard_tasklet);

/*注册电源管理的 KEB 设备*/
      pm_kbd = pm_register(PM_SYS_DEV, PM_SYS_KBC, pm_kbd_request_override);

      return 0;
}
```

Kbd_init_hw()，包含了为 keyboard 分配 1/O 端口、分配中断号及对应处理函数、为进行基本保证测试（BAT）初始化寄存器，然后调用在 pc_keyb.c 的 intialize_kbd()进行硬件初始化，这是一个非常重要的函数，它的初始化过程的流程如图 11.13 所示。

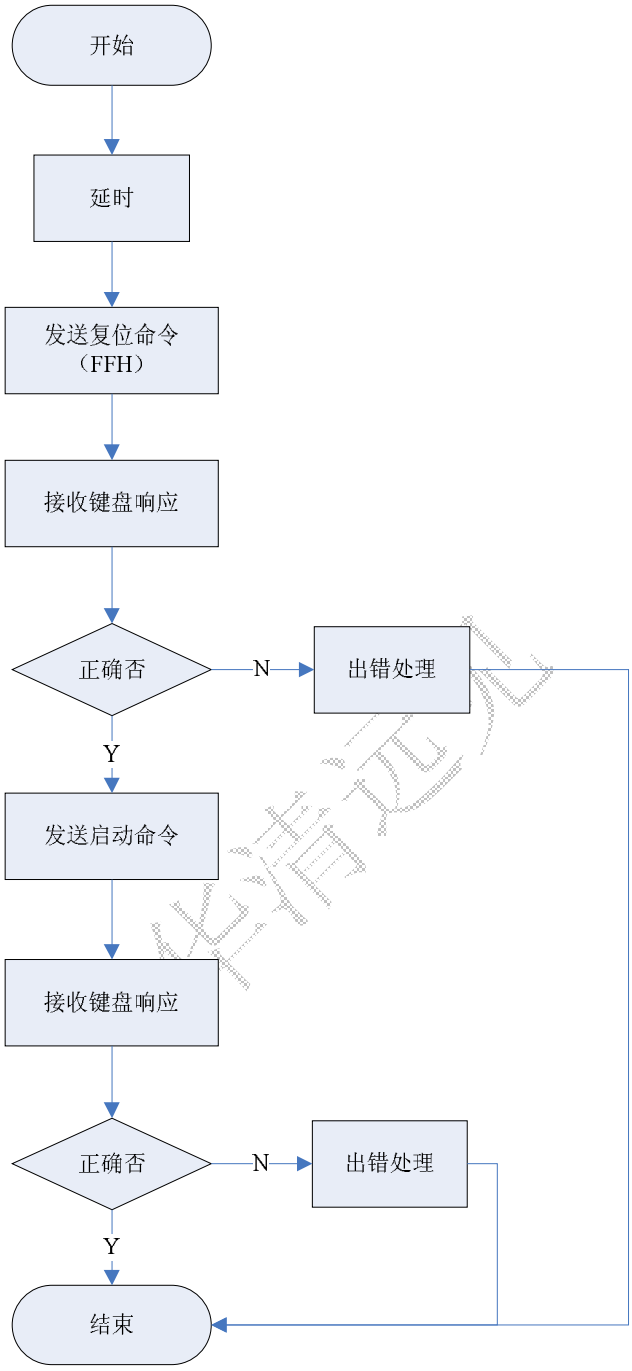

图 11.13　intialize_kbd()函数流程

```
static char * __init initialize_kbd(void)
{
       int status;
```

```
        /*
         * 测试键盘接口
         * 如果测试成功，那么将会有一个 x55 放在缓冲区中
         */
        kbd_write_command_w(KBD_CCMD_SELF_TEST);
        if (kbd_wait_for_input() != 0x55)
            return "Keyboard failed self test";

        /*
         * 启动一个键盘接口测试，这时会启动控制器来测试键盘的时钟和数据线，测试结果
        放在输入缓冲区中
         */
        kbd_write_command_w(KBD_CCMD_KBD_TEST);
        if (kbd_wait_for_input() != 0x00)
              return "Keyboard interface failed self test";

        /*
         * 通过启动键盘时钟使能键盘
         */
        kbd_write_command_w(KBD_CCMD_KBD_ENABLE);

        /*
         * 重启键盘。如果读取时间超时，就会认为在该机器里没有键盘
         * 如果键盘要求再次发送，则使能键盘重发机制
         */
        do {
               kbd_write_output_w(KBD_CMD_RESET);
               status = kbd_wait_for_input();
               if (status == KBD_REPLY_ACK)
                      break;
               if (status != KBD_REPLY_RESEND)
                      return "Keyboard reset failed, no ACK";
        } while (1);

        if (kbd_wait_for_input() != KBD_REPLY_POR)
               return "Keyboard reset failed, no POR";

        /*
         * 设置键盘控制模式，在这期间，键盘应该设置为关闭状态
```

```
         */
        do {
               kbd_write_output_w(KBD_CMD_DISABLE);
               status = kbd_wait_for_input();
               if (status == KBD_REPLY_ACK)
                      break;
               if (status != KBD_REPLY_RESEND)
                      return "Disable keyboard: no ACK";
        } while (1);

        kbd_write_command_w(KBD_CCMD_WRITE_MODE);
        kbd_write_output_w(KBD_MODE_KBD_INT
                        | KBD_MODE_SYS
                        | KBD_MODE_DISABLE_MOUSE
                        | KBD_MODE_KCC);

        if (!(kbd_write_command_w_and_wait(KBD_CCMD_READ_MODE) & KBD_MODE_KCC))
        {
               /*
                * If the controller does not support conversion,
                * Set the keyboard to scan-code set 1.
                */
               kbd_write_output_w(0xF0);
               kbd_wait_for_input();
               kbd_write_output_w(0x01);
               kbd_wait_for_input();
        }

        if (kbd_write_output_w_and_wait(KBD_CMD_ENABLE) != KBD_REPLY_ACK)
               return "Enable keyboard: no ACK";

        /*
         *最后，把键盘读取率设置为最高
         */
        if (kbd_write_output_w_and_wait(KBD_CMD_SET_RATE) != KBD_REPLY_ACK)
               return "Set rate: no ACK";
        if (kbd_write_output_w_and_wait(0x00) != KBD_REPLY_ACK)
               return "Set rate: no 2nd ACK";
```

```
        return NULL;
}
```

（2）按键处理

按键处理是键盘驱动中最为重要的一部分。当有按键事件产生时，则调用键盘中断处理函数，也就是 keyboard interrupt()，它会调用到 handle_kbd_event()并调用 handle_scancode()函数。handle_scancode()这个函数完成按键处理的过程，它的功能是与 TY 设备通信，Keymap 表装入，按键处理。handle scancode()处理的结果就是把按键发给相应的处理函数，这些函数基本上都会调用 put_queue()函数。这个函数就是将处理函数的结果发送到 TY 或者 Console 进行显示。

下面是 handle_kbd_event 和 handle_scancode 函数源代码：

```
static unsigned char handle_kbd_event(void)
{
        unsigned char status = kbd_read_status();
        unsigned int work = 10000;

        while ((--work > 0) && (status & KBD_STAT_OBF)) {
                unsigned char scancode;

                scancode = kbd_read_input();

                /*错误字节必须被忽略*/
#if 1
                /* 忽略错误字节 */
                if (!(status & (KBD_STAT_GTO | KBD_STAT_PERR)))
#endif
                {
                        if (status & KBD_STAT_MOUSE_OBF)
                                handle_mouse_event(scancode);
                        else
                                handle_keyboard_event(scancode);
                }

                status = kbd_read_status();
        }

        if (!work)
                printk(KERN_ERR "pc_keyb: controller jammed (0x%02X).\n", status);
```

```
        return status;
}

static inline void handle_keyboard_event(unsigned char scancode)
{
#ifdef CONFIG_VT
        kbd_exists = 1;
        if (do_acknowledge(scancode))
                handle_scancode(scancode, !(scancode & 0x80));
#endif
        tasklet_schedule(&keyboard_tasklet);
}
```

（3）转化键盘扫描码

在完成键盘的初始化之后，就需要完成对键盘扫描码的转化。这里调用函数 pckdb_translate，实现了 scancode 和 keycode 之间的转换。

```
int pckbd_translate(unsigned char scancode, unsigned char *keycode,char raw_mode)
{
     static int prev_scancode;

     /* special prefix scancodes.. */
     if (scancode == 0xe0 || scancode == 0xe1) {
          prev_scancode = scancode;
          return 0;
     }

     /* 0xFF 很少被发送，故可以忽略，0x00 是错误码。*/
     if (scancode == 0x00 || scancode == 0xff) {
          prev_scancode = 0;
          return 0;
     }

     scancode &= 0x7f;

     if (prev_scancode) {
        /*
         通常是 0xe0，但是一个暂停键可以产生 e1 1d 45 e1 9d c5 字符
         */
        if (prev_scancode != 0xe0) {
```

```
                if (prev_scancode == 0xe1 && scancode == 0x1d) {
                prev_scancode = 0x100;
                return 0;
               } else if (prev_scancode == 0x100 && scancode == 0x45) {
                *keycode = E1_PAUSE;
                prev_scancode = 0;
                } else {
   #ifdef KBD_REPORT_UNKN
                if (!raw_mode)
                    printk(KERN_INFO "keyboard: unknown e1 escape sequence\n");
   #endif
                prev_scancode = 0;
                return 0;
                }
          } else {
               prev_scancode = 0;
               /*
                *  键盘保持了它自己的内部总线锁和锁状态。在总线锁中，状态 E0 AA 会生成代
码，而状态 E0 2A 会跟随停止码
               if (scancode == 0x2a || scancode == 0x36)
                    return 0;

               if (e0_keys[scancode])
               *keycode = e0_keys[scancode];
               else {
   #ifdef KBD_REPORT_UNKN
              if (!raw_mode)
                  printk(KERN_INFO "keyboard: unknown scancode e0 %02x\n",
                      scancode);
   #endif
              return 0;
              }
        }
      } else if (scancode >= SC_LIM) {
         *keycode = high_keys[scancode - SC_LIM];

         if (!*keycode) {
              if (!raw_mode) {
   #ifdef KBD_REPORT_UNKN
```

```
                printk(KERN_INFO "keyboard: unrecognized scancode (%02x)"
                     " - ignored\n", scancode);
#endif
                }
                return 0;
           }
         } else
           *keycode = scancode;
           return 1;
}
```

（4）按键处理

在完成键盘扫描码转换之后就可以开始进行按键处理，这里用到了 keyboard.c 中的重要函数 kbd_processkeycode。源码如下所示：

```
static void
kbd_processkeycode(unsigned char keycode, char up_flag, int autorepeat)
{
     char raw_mode = (kbd->kbdmode == VC_RAW);

     if (up_flag) {
          rep = 0;
          if(!test_and_clear_bit(keycode, key_down))
               up_flag = kbd_unexpected_up(keycode);
     } else {
          rep = test_and_set_bit(keycode, key_down);
          /*如果键盘自动重复，那么就要把它忽略，我们会使用自己的自动重复机制*/
          if (rep && !autorepeat)
               return;
     }

     if (kbd_repeatkeycode == keycode || !up_flag || raw_mode) {
          kbd_repeatkeycode = -1;
          del_timer(&key_autorepeat_timer);
     }

#ifdef CONFIG_MAGIC_SYSRQ      /* Handle the SysRq Hack */
     if (keycode == SYSRQ_KEY) {
          sysrq_pressed = !up_flag;
          goto out;
```

```
        } else if (sysrq_pressed) {
              if (!up_flag) {
                    handle_sysrq(kbd_sysrq_xlate[keycode], kbd_pt_regs, kbd, tty);
                    goto out;
              }
        }
    #endif

        /*
         *计算下一次需要自动重复的时间
         */
        if (!up_flag && !raw_mode) {
              kbd_repeatkeycode = keycode;
              if (vc_kbd_mode(kbd, VC_REPEAT)) {
                    if (rep)
                          key_autorepeat_timer.expires = jiffies + kbd_repeati-
nterval;
                    else
                          key_autorepeat_timer.expires = jiffies + kbd_repea-
ttimeout;
                    add_timer(&key_autorepeat_timer);
              }
        }

        if (kbd->kbdmode == VC_MEDIUMRAW) {
              /* soon keycodes will require more than one byte */
              put_queue(keycode + up_flag);
              raw_mode = 1;    /* Most key classes will be ignored */
        }

        if (!rep ||
              (vc_kbd_mode(kbd,VC_REPEAT) && tty &&
               (L_ECHO(tty) || (tty->driver.chars_in_buffer(tty) == 0)))) {
              u_short keysym;
              u_char type;

              /* the XOR below used to be an OR */
              int shift_final = (shift_state | kbd->slockstate) ^
                   kbd->lockstate;
```

```
            ushort *key_map = key_maps[shift_final];

            if (key_map != NULL) {
                  keysym = key_map[keycode];
                  type = KTYP(keysym);

             if (type >= 0xf0) {
                   type -= 0xf0;
                   if (raw_mode && !(TYPES_ALLOWED_IN_RAW_MODE & (1 << type)))
                    goto out;
                   if (type == KT_LETTER) {
                    type = KT_LATIN;
                    if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
                         key_map = key_maps[shift_final ^ (1<<KG_SHIFT)];
                         if (key_map)
                             keysym = key_map[keycode];
                    }
                  }
                  (*key_handler[type])(keysym & 0xff, up_flag);
                  if (type != KT_SLOCK)
                     kbd->slockstate = 0;
             } else {
                  /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
                  if (!up_flag && !raw_mode)
                     to_utf8(keysym);
             }
        } else {
             /* 我们至少需要更新移动状态*/
#if 1
            compute_shiftstate();
            kbd->slockstate = 0; /* play it safe */
#else
            keysym = U(key_maps[0][keycode]);
            type = KTYP(keysym);
            if (type == KT_SHIFT)
               (*key_handler[type])(keysym & 0xff, up_flag);
#endif
        }
     }
```

```
        rep = 0;
    out:
        return;
    }
```

11.7 实验内容——skull 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。读者可以了解到整个驱动的编写流程。

2. 实验内容

该实验要求实现对一段内存的打开、关闭、读写的操作，并要通过编写测试程序来测试驱动安装是否成功。

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：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/malloc.h>
#include <asm/uaccess.h>
#include <linux/errno.h>
/*全局变量*/
unsigned int fs_major =0;
static char *data;
/*关键数据类型，注意每行结尾是逗号*/
static struct file_operations chr_fops={
          read:   test_read,
          write:  test_write,
          open:   test_open,
          release:      test_release,
};
```

```
    /*函数声明*/
    static ssize_t test_read(struct file *file, char *buf, size_t count, loff_t *f_pos);
    static ssize_t test_write(struct file *file,const char *buffer, size_t
count,loff_t *f_pos);
    static int test_open(struct inode *inode, struct file *file);
    static int test_release(struct inode *inode,struct file *file);
    int init_module(void);
    void cleanup_module(void);
    /*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);
              if(len < count)
                         count = len;
              copy_to_user(buf,data,count+1);
              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;
              kfree(data);
              data = (char *)kmalloc(sizeof(char)*(count+1),GFP_KERNEL);
              if(!data)
                         return -ENOMEM;
              copy_from_user(data,buffer,count+1);
              return count;
    }
    /*打开函数*/
    static int test_open(struct inode *inode, struct file *file)
    {
               MOD_INC_USE_COUNT;
               printk("This is open\n");
               return 0;
```

```
}
/*释放函数*/
static int test_release(struct inode *inode,struct file *file)
{
           MOD_DEC_USE_COUNT;
           printk("this is released\n");
           return 0;
}
/*模块注册入口*/
int init_module(void)
{
          int res;
          res=register_chrdev(0,"fs",&chr_fops);
          if(res<0)
          {
                    printk("can't get major name!\n");
                    return res;
          }
          if(fs_major == 0)
                   fs_major = res;
          return 0;
}
/*撤销模块入口*/
void cleanup_module(void)
{
unregister_chrdev(fs_major,"fs");
}
```

（2）编译代码

要注意在此处要加上-DMODULE -D__KERNEL__选项，如下所示：

```
arm-linux-gcc -DMODULE -D__KERNEL__ -c kernel.c
```

（3）加载模块

```
insmod ./kernel.o
```

（4）查看设备号

```
vi /proc/device
```

（5）映射为设备文件

接下来就要将相应的设备映射为设备文件，这里可以使用命令 mknod，如下所示：

```
mknod /dev/fs c 254 0
```

这里的/dev/fs 就是相应的设备文件，c 代表字符文件，254 代表主设备号（与/proc/devices 中一样），0 为次设备号。

（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正确性。上面的实例只实现了简单的读写功能，测试代码如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/ioctl.h>
int main()
{
          int fd,i,nwrite,nread;
          char *buf ="hello\n";
          char read_buf[6]={0};
          fd=open("/dev/fs",O_RDWR);
          if(fd<=0)
          {
                    perror("open");
                    exit(1);
          }
          else
                    printf("open success\n");
          nwrite = write(fd,buf,strlen(buf));
          if(nwrite<0)
          {
                    perror("write");
                    exit(1);
          }
          nread = read(fd,read_buf,6);
          if(nread<0)
          {
                    perror("read");
```

```
            exit(1);
        }
        else
                printf("read is %s\n",read_buf);
        close(fd);
        exit(0);
}
```

4．实验结果

在加载模块后可以查看/var/log/messages 是否有程序中相应的信息输出：

```
Feb 21 09:49:10  kernel: This is open
```

查看设备号时有类似如下信息：

```
254 fs
```

这代表 fs 设备的主设备号是 254。
最后运行测试程序，结果如下所示：

```
[root@(none) tmp]# ./testing
open success
read is hello
```

查看/var/log/messages，有输出信息如下所示：

```
Feb 21 12:57:06 kernel: This is open
Feb 21 12:57:06 kernel: this is released
Feb 21 09:43:40 kernel: Goodbye world
```

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 LCD 驱动为例介绍了一个较为大型的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，并以键盘驱动为例进行讲解。

本章的实验安排的是 skull 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整个流程。

思考与练习

将本章中所述的 lcd 驱动程序运行编译，并通过模块加载在开发板上测试实验。