第十一章　设备驱动程序

操作系统的主要任务之一是控制所有的输入/输出设备。它必须向设备发布命令，捕获中断并进行错误处理，它还要提供一个设备与系统其余部分的简单易用的界面，该界面应该对所有的设备尽可能的一致，从而将系统硬件设备细节从用户视线中隐藏起来，例如虚拟文件系统对各种已安装的文件系统类型提供了统一的视图而屏蔽了具体底层细节，具体细节都是由设备驱动程序来完成的，对于驱动程序，在 Linux 中可以按照模块的形式进行编译和加载。

11.1　概述

在 Linux 中输入/输出设备被分为 3 类：块设备，字符设备和网络设备。这种分类的使用方法，可以将控制不同输入/输出设备的驱动程序和其他操作系统软件成分分离开来。例如文件系统仅仅控制抽象的块设备，而将与设备有关的部分留给低层软件，即驱动程序。字符设备指那些无需缓冲区可以直接读写的设备，如系统的串口设备/dev/cua0 和/dev/cua1。块设备则仅能以块为单位进行读写的设备，如软盘、硬盘、光盘等，典型块的大小为 512 或 1024 字节。从名称使人想到，字符设备在单个字符的基础上接收和发送数据。为了改进传送数据的速度和效率，块设备在整个数据缓冲区填满时才一起传送数据。网络设备可以通过 BSD 套接口访问数据，关于这方面的内容我们将在第十二章中进行讨论。

在 Linux 中，对每一个设备的描述是通过主设备号和从设备号，其中主设备号描述控制这个设备的驱动程序，也就是说驱动程序和主设备号是一一对应的，从设备号是用来区分同一个驱动程序控制的不同设备。例如主 IDE 硬盘的每个分区的从设备号都不相同，/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主、从设备号将包含在系统调用中的设备特殊文件映射到设备的管理程序，以及大量系统表格中，如字符设备表—chrdevs。 块（磁盘）设备和字符设备的设备特殊文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但 Linux 寻找和初始化网络设备时才建立这种文件。

11.1.1　I/O 软件

I/O 软件的总体目标就是将软件组织成一种层次结构，低层软件用来屏蔽具体设备细节，高层软件则为用户提供一个简洁规范的界面。这种层次结构很好地体现了 I/O 设计的一个关键的概念：设备无关性，其含义就是程序员写的软件无需须修改就能读出软盘，硬盘以及

CD-ROM 等不同设备上的文件。

输入/输出系统的层次结构及各层次的功能如图 11.1 所示。

从图可以看出，用户进程的下层是设备无关的软件，在 Linux 中，设备无关软件的功能大部分由文件系统去完成，其基本功能就是执行适用于所有设备的常用的输入/输出功能，向用户软件提供一个一致的接口。其结构如图 11.2 所示。

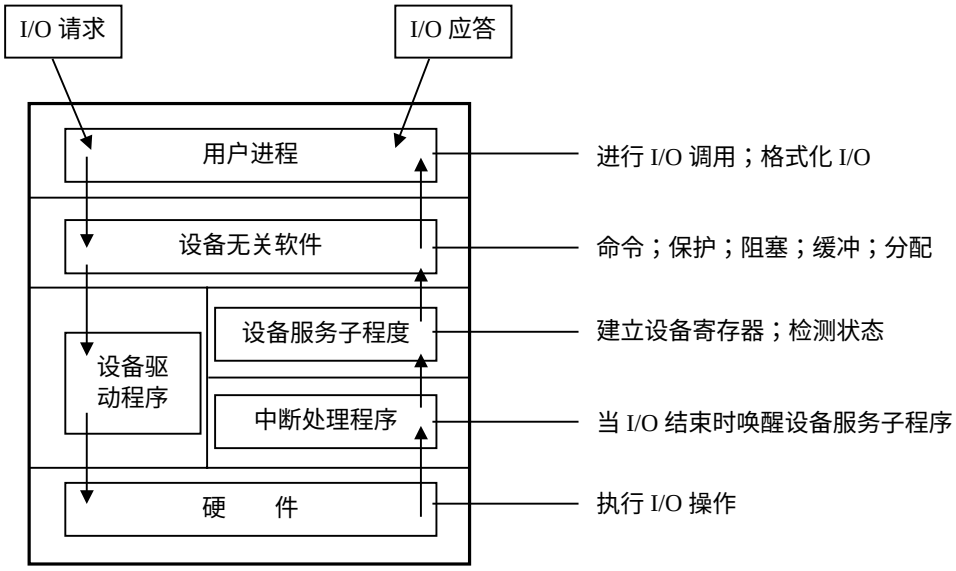

图 11.1　输入/输出系统的层次结构及各层次的功能

对设备程序的统一接口
设备命名
设备保护
提供一个独立于设备的块
缓冲
块设备的存储分配
分配和释放独占设备
错误报告

图 11.2　设备无关软件的功能

设备无关的软件具有以下特点。

- 文件和设备采用统一命名。设备无关软件负责将设备名映射到相应的驱动程序，一个设备名唯一地确定一个索引节点，索引节点中包含了主设备号和从设备号，通过主设备号可以找到相应的设备驱动程序，通过从设备号确定具体的物理设备。

- 对设备提供的保护机制同文件系统一样都采用 rwx 权限。
- 数据块的大小可能对于不同的设备其大小不一样，但操作系统屏蔽这一事实，向高层软件提供了统一的逻辑块的大小。
- 为了解决数据交换速度的匹配问题，采用了缓冲技术，对于缓冲区的管理由文件系统去完成。
- 块设备的存储分配也是由文件系统去处理。
- 对于独占设备的分配和释放属于对临界资源的管理。

11.1.2　设备驱动程序

CPU 并不是系统中唯一的智能设备，每个物理设备都拥有自己的控制器。键盘、鼠标和串行口由一个高级 I/O 芯片统一管理，IDE 控制器控制 IDE 硬盘，而 SCSI 控制器控制 SCSI 硬盘等等。每个硬件控制器都有各自的控制状态寄存器（CSR）并且各不相同。例如 Adaptec 2940 SCSI 控制器的 CSR 与 NCR 810 SCSI 控制器完全不一样。这些寄存器用来启动、停止、初始化设备以及对设备进行诊断。在 Linux 中管理硬件设备控制器的代码并没有放置在每个应用程序中而是由内核统一管理，这些处理和管理硬件控制器的软件就是设备驱动程序。Linux 内核的设备管理是由一组运行在特权级上，驻留在内存以及对底层硬件进行处理的共享库的驱动程序来完成的。

设备管理的一个基本特征是设备处理的抽象性，即所有硬件设备都被看成普通文件，可以通过用操纵普通文件相同的系统调用来打开、关闭、读取和写入设备。系统中每个设备都用一种设备特殊文件来表示，例如系统中第一个 IDE 硬盘被表示成/dev/hda。

那么，系统是如何将设备在用户视野中屏蔽起来的呢？图 11.3 说明了用户进程请求设备进行输入输出的简单流程。

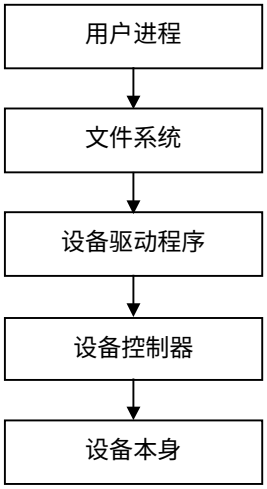

图 11.3　用户进程请求设备服务的流程

首先当用户进程发出输入输出时，系统把请求处理的权限放在文件系统，文件系统通过驱动程序提供的接口将任务下放到驱动程序，驱动程序根据需要对设备控制器进行操作，设

备控制器再去控制设备本身。

这样通过层层隔离，对用户进程基本上屏蔽了设备的各种特性，使用户的操作简便易行，不必去考虑具体设备的运作，就像对待文件操作一样去操作设备，因为实际上在驱动程序向文件系统提供的接口已经屏蔽掉了设备的电器特性。

设备控制器对设备本身的控制是电器工程师所关心的事情，操作系统对输入/输出设备的管理只是通过文件系统和驱动程序来完成的。也就是说在操作系统中，输入/输出系统所关心的只是驱动程序。

Linux 设备驱动程序的主要功能有：

- 对设备进行初始化；
- 使设备投入运行和退出服务；
- 从设备接收数据并将它们送回内核；
- 将数据从内核送到设备；
- 检测和处理设备出现的错误。

在 Linux 中，设备驱动程序是一组相关函数的集合。它包含设备服务子程序和中断处理程序。设备服务子程序包含了所有与设备相关的代码，每个设备服务子程序只处理一种设备或者紧密相关的设备。其功能就是从与设备无关的软件中接受抽象的命令并执行之。当执行一条请求时,具体操作是根据控制器对驱动程序提供的接口(指的是控制器中的各种寄存器)，并利用中断机制去调用中断服务子程序配合设备来完成这个请求。设备驱动程序利用结构 file_operations 与文件系统联系起来，即设备的各种操作的入口函数存在 file_operation 中。对于特定的设备来说有一些操作是不必要的，其入口置为 NULL。

Linux 内核中虽存在许多不同的设备驱动程序但它们具有一些共同的特性，如下所述。

1．驱动程序属于内核代码

设备驱动程序是内核的一部分，它像内核中其他代码一样运行在内核模式，驱动程序如果出错将会使操作系统受到严重破坏,甚至能使系统崩溃并导致文件系统的破坏和数据丢失。

2．为内核提供统一的接口

设备驱动程序必须为 Linux 内核或其他子系统提供一个标准的接口。例如终端驱动程序为 Linux 内核提供了一个文件 I/O 接口。

3．驱动程序的执行属于内核机制并且使用内核服务

设备驱动可以使用标准的内核服务如内存分配、中断发送和等待队列等。

4．动态可加载

多数 Linux 设备驱动程序可以在内核模块发出加载请求时加载，而不再使用时将其卸载。这样内核能有效地利用系统资源。

5．可配置

Linux 设备驱动程序可以连接到内核中。当内核被编译时，被连入内核的设备驱动程序是可配置的。

11.2　设备驱动基础

11.2.1　I/O 端口

每个连接到 I/O 总线上的设备都有自己的 I/O 地址集，即所谓的 I/O 端口（I/O port)。在 IBM PC 体系结构中，I/O 地址空间一共提供了 65,536 个 8 位的 I/O 端口。可以把两个连续的 8 位端口看成一个 16 位端口，但是这必须是从偶数地址开始。同理，也可以把两个连续的 16 位端口看成一个 32 位端口，但是这必须是从 4 的整数倍地址开始。有 4 条专用的汇编语言指令可以允许 CPU 对 I/O 端口进行读写：它们分别是 in、ins、out 和 outs。在执行其中的一条指令时，CPU 使用地址总线选择所请求的 I/O 端口，使用数据总线在 CPU 寄存器和端口之间传送数据。

I/O 端口还可以被映射到物理地址空间，因此，处理器和 I/O 设备之间的通信就可以直接使用对内存进行操作的汇编语言指令（例如，mov、and、or 等等)。现代的硬件设备更倾向于映射 I/O，因为这样处理的速度较快，并可以和 DMA 结合起来使用。

系统设计者的主要目的是提供对 I/O 编程的统一方法，但又不牺牲性能。为了达到这个目的，每个设备的 I/O 端口都被组织成如图 11.4 所示的一组专用寄存器。CPU 把要发给设备的命令写入控制寄存器（Control Register)，并从状态寄存器（Status Register）中读出表示设备内部状态的值。CPU 还可以通过读取输入寄存器（Input Register）的内容从设备取得数据，也可以通过向输出寄存器（Output Register）中写入字节而把数据输出到设备。

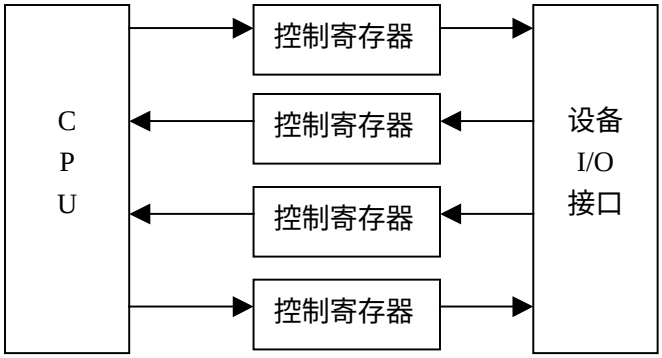

图 11.4　专用 I/O 端口

为了降低成本，通常把同一 I/O 端口用于不同目的。例如，某些位描述设备的状态，而其他位指定发布给设备的命令。同理，也可以把同一 I/O 端口用作输入寄存器或输出寄存器。

那么如何访问 I/O 端口? in、out、ins 和 outs 汇编语言指令都可以访问 I/O 端口。Linux 内核中定义了以下辅助函数来简化这种访问。

1. inb( )、inw( )、inl( )函数

分别从 I/O 端口读取 1、2 或 4 个连续字节。后缀“ b ”、“ w ”、“ l ”分别代表一个字节（8 位）、一个字（16 位）以及一个长整型（32 位）。

2. inb_p( )、inw_p( )、inl_p( )

分别从 I/O 端口读取 1、2 或 4 个连续字节，然后执行一条“ 哑元（dummy，即空指令）”指令使 CPU 暂停。

3. outb( )、outw( )、outl( )

分别向一个 I/O 端口写入 1、2 或 4 个连续字节。

4. outb_p( )、outw_p( )、outl_p( )

分别向一个 I/O 端口写入 1、2 或 4 个连续字节，然后执行一条“ 哑元 ”指令使 CPU 暂停。

5. insb( )、insw( )、insl( )

分别从 I/O 端口读入以 1、2 或 4 个字节为一组的连续字节序列。字节序列的长度由该函数的参数给出。

6. outsb( )、outsw( )、outsl( )

分别向 I/O 端口写入以 1、2 或 4 个字节为一组的连续字节序列。

虽然访问 I/O 端口非常简单，但是检测哪些 I/O 端口已经分配给 I/O 设备可能就不这么简单，特别是对基于 ISA 总线的系统来说更是如此。通常，I/O 设备驱动程序为了侦探硬件设备，需要盲目地向某一 I/O 端口写入数据；但是，如果其他硬件设备已经使用这个端口，那么系统就会崩溃。为了防止这种情况的发生，内核必须使用 iotable 表来记录分配给每个硬件设备的 I/O 端口。任何设备驱动程序都可以使用下面 3 个函数。

request_region( )

把一个给定区间的 I/O 端口分配给一个 I/O 设备。

check_region( )

检查一个给定区间的 I/O 端口是否空闲，或者其中一些是否已经分配给某个 I/O 设备。

release_region( )

释放以前分配给一个 I/O 设备的给定区间的 I/O 端口。

当前分配给 I/O 设备的 I/O 地址可以从/proc/ioports 文件中获得。

11.2.2　I/O 接口及设备控制器

I/O 接口是处于一组 I/O 端口和对应的设备控制器之间的一种硬件电路。它起翻译器的作用，即把 I/O 端口中的值转换成设备所需要的命令和数据。从另一个角度来看，它检测设备状态的变化，并对起状态寄存器作用的 I/O 端口进行相应地更新。还可以通过一条 IRQ 线把这种电路连接到可编程中断控制器上，以使它代表相应的设备发出中断请求。

有两类类型的接口，如下所述。

1．专用 I/O 接口

专门用于一个特定的硬件设备。在一些情况下，设备控制器与这种 I/O 接口处于同一块卡中，连接到专用 I/O 接口上的设备可以是内部设备（位于 PC 机箱内部的设备），也可以是外部设备（位于 PC 机箱外部的设备）。例如键盘接口、图形接口、磁盘接口、总线鼠标接口及网络接口都属于专用 I/O 接口。

2．通用 I/O 接口

用来连接多个不同的硬件设备。连接到通用 I/O 接口上的设备通常都是外部设备。例如并口、串口、通用串行总线（USB）、PCMCIA 接口及 SCSI 接口都属于通用 I/O 接口。

复杂的设备可能需要一个设备控制器来驱动。控制器具有两方面的作用，一是对从 I/O 接口接收到的高级命令进行解释，并通过向设备发送适当的电信号序列强制设备执行特定的操作；二是对从设备接收到的电信号进行转换和解释，并通过 I/O 接口修改状态寄存器的值。

磁盘控制器是一种比较典型的设备控制器，它通过 I/O 接口从微处理器接收诸如“写这个数据块”之类的高级命令，并将其转换成诸如“把磁头定位在正确位置的磁道”上和“把数据写入这个磁道”之类的低级磁盘操作。现在的磁盘控制器相当复杂，因为它们可以把磁盘数据快速保存到内存的缓存区中，还可以根据实际磁盘的几何结构重新安排 CPU 的高级请求，使其优化。

11.2.3　设备文件

设备文件是用来表示 Linux 所支持的大多数设备的，每个设备文件除了设备名，还有 3 个属性：即类型、主设备号、从设备号。

设备文件是通过 mknod 系统调用创建的。其原型为：

```
mknod(const char * filename, int mode, dev_t dev)
```

其参数有设备文件名、操作模式、主设备号及从设备号。最后两个参数合并成一个 16 位的 dev_t 无符号短整数，高 8 位用于主设备号，低 8 位用于从设备号。内核中定义了 3 个宏来处理主、从设备号：MAJOR 和 MINOR 宏可以从 16 位数中提取出主、从设备号，而 MKDEV 宏可以把主、从号合并为一个 16 位数。实际上，dev_t 是专用于应用程序的一个数据类型；在内核中使用 kdev_t 数据类型。在 Linux 2.4 及以前的版本中，这两个类型都会是一个无符号短整型，但是在以后的 Linux 版本中，kdev_t 会成为一个完整的设备文件描述符，也就说，也许会扩成 32 位的长整数。

分配给设备号的正式注册信息及/dev 目录索引节点存放在 documentation/devices.txt 文件中。也可以在 include/linux/major.h 文件中找到所支持的主设备号。

设备文件通常位于/dev 目录下。表 11.1 显示了一些设备文件的属性。注意同一主设备号既可以标识字符设备，也可以标识块设备。

表 11.1　　设备文件的例子

设备名	类型	主设备号	从号	说明
/dev/fd0	块设备	2	0	软盘
/dev/hda	块设备	3	0	第 1 个 IDE 磁盘
/dev/hda2	块设备	3	2	第 1 个 IDE 磁盘上的第 2 个主分区
/dev/hdb	块设备	3	64	第 2 个 IDE 磁盘
/dev/hdb3	块设备	3	67	第 2 个 IDE 磁盘上的第 3 个主分区
/dev/ttyp0	字符设备	3	0	终端
/dev/console	字符设备	5	1	控制台
/dev/lp1	字符设备	6	1	并口打印机
/dev/ttyS0	字符设备	4	64	第 1 个串口
/dev/rtc	字符设备	10	135	实时时钟
/dev/null	字符设备	1	3	空设备（黑洞）

一个设备文件通常与一个硬件设备（如硬盘，/dev/hda）相关连， 或硬件设备的某一物理或逻辑分区（如磁盘分区，/dev/hda2）相关联。但在某些情况下，设备文件不会和任何实际的硬件关联，而是表示一个虚拟的逻辑设备。例如，/dev/null 就是对应于一个“黑洞”的设备文件：所有写入这个文件的数据都被简单地丢弃，因此，该文件看起来总为空。

就内核所关心的内容而言，设备文件名是无关紧要的。如果你建立了一个名为/tmp/disk 的设备文件，类型为“块”，主设备号是 3，从设备号为 0，那么这个设备文件就和表中的/dev/hda 等价。另一方面，对某些应用程序来说，设备文件名可能就很有意义。例如，通信程序可以假设第 1 个串口和/dev/ttyS0 设备文件关联。

1．块设备和字符设备的比较

块设备具有以下特点。

- 可以在一次 I/O 操作中传送固定大小的数据块。
- 可以随机访问设备中所存放的块：传送数据块所需要的时间独立于块在设备中的位置，也独立于当前设备的状态。

块设备典型的例子是硬盘、软盘及 CD-ROM。也可以把 RAM 磁盘当作块设备来对待，这是通过把部分 RAM 配置成快速硬盘而获得的，因此，可以把这部分 RAM 作为应用程序高效存取数据的临时存储器。

字符设备具有以下特点。

- 可以在一次 I/O 操作中传送任意大小的数据。实际上，诸如打印机之类的字符设备可以一次传送一个字节，而诸如磁带之类的设备可以一次传送可变大小的数据块。
- 通常访问连续的字符。

2．网卡

有些 I/O 设备没有对应的设备文件。最明显的一个例子是网卡。实际上，网卡把向外发送的数据放入通往远程计算机系统的一条线上,把从远程系统中接收到的报文装入内核内存。

从 BSD 开始，所有的 UNIX 类系统为计算机中的每个网卡都分配一个不同的符号名。例如，第一个以太网卡名为 eth0。然而，这个名字并没有对应的设备文件，也没有对应的索引节点。

由于没有使用文件系统，所以系统管理员必须建立设备名和网络地址之间的联系。因此，应用程序和网络接口之间的数据通信不是基于标准的有关文件的系统调用的，而是基于 socket( )、bind( )、listen( )、accept()和 connect( )系统调用的，这些系统调用对网络地址进行操作。这组系统调用是在 UNIX BSD 中首先引入的，现在已经成为网络设备的标准编程模型。

11.2.4　VFS 对设备文件的处理

虽然设备文件也在系统的目录树中，但是它们和普通文件以及目录有根本的不同。当进程访问普通文件（即磁盘文件）时，它会通过文件系统访问磁盘分区中的一些数据块。而在进程访问设备文件时，它只要驱动硬件设备就可以了。例如，进程可以访问一个设备文件以从连接到计算机的温度计读取房间的温度。VFS 的责任是为应用程序隐藏设备文件与普通文件之间的差异。

为了做到这点，VFS 改变打开的设备文件的缺省文件操作。因此，可以把对设备文件的任一系统调用转换成对设备相关的函数的调用，而不是对主文件系统相应函数的调用。设备相关的函数对硬件设备进行操作以完成进程所请求的操作。

控制 I/O 设备的一组设备相关的函数称为设备驱动程序。由于每个设备都有一个唯一的 I/O 控制器，因此也就有唯一的命令和唯一的状态信息，所以大部分 I/O 设备类型都有自己的驱动程序。

11.2.5　中断处理

设备一般都比 CPU 慢得多。因此一般情况下，当一个进程通过设备驱动程序向设备发出读写请求后，CPU 并不等待 I/O 操作的完成，而是让正在执行的进程去睡眠，CPU 自己做别的事情，例如唤醒另一个进程执行。当设备完成 I/O 操作需要通知 CPU 时，会向 CPU 发出一个中断请求；然后 CPU 根据中断请求来决定调用相应的设备驱动程序。

当设备执行某个命令时，如“ 将读取磁头移动到软盘的第 42 扇区上 ”，设备驱动程序可以从查询方式和中断方式中选择一种来判断设备是否已经完成此命令。

查询方式意味着需要经常读取设备的状态，一直到设备状态表明请求已经完成为止。如

果设备驱动程序被连接进内核，这时使用查询方式将会带来灾难性后果：内核将在此过程中无所事事，直到设备完成目前的请求。有一种方法可以有效地改善这一弊端，就是通过使用系统定时器，使内核周期性调用设备驱动程序中的某个例程来检查设备状态。 使用定时器是查询方式中最好的一种，但更有效的方法是使用中断。

基于中断的设备驱动程序，指的是在硬件设备需要服务时向 CPU 发一个中断信号，引发中断服务子程序执行 。这样就大大地提高了系统资源的利用率，使内核不必一直等到设备执行完任务后才开始有事可干，而是在设备工作期间内核就可以转去处理其他的事务，收到中断请求信号时再回头响应设备。

1. Linux 对中断的管理

Linux 内核为了将来自硬件设备的中断传递到相应的设备驱动程序，在驱动程序初始化的时候就将其对应的中断程序进行了登记，即通过调用函数 request_irq ( ) 将其中断信息添加到结构为 irqaction 的数组中，从而使中断号和中断服务程序联系起来。请参见第四章。

request_irq ( )函数原形如下：

```
int request_irq(unsigned int irq,              /*  中断请求号  */
 void (*handler)(int, void *, struct pt_regs *),/*  指向中断服务子程序 */
      unsigned long irqflags,                   /*  中断类型 */
       const char * devname,                    /*  设备的名字 */
       void *dev_id)；
```

另外，irqaction 的数据结构如下，其图示如图 11.5 所示。

```
struct irqaction {
     void (*handler)(int, void *, struct pt_regs *);
     unsigned long flags;
     unsigned long mask;
     const char *name;
     void *dev_id;
    struct irqaction *next;
};
   static struct irqaction *irq_action[NR_IRQS+1]
```

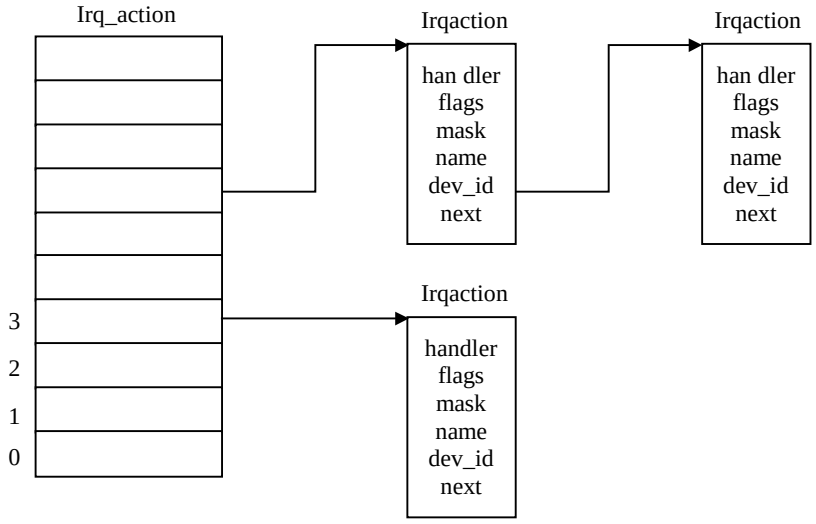

图 11.5　irqaction 的数据结构

根据设备的中断号可以在数组 irq_action 检索到设备的中断信息。对中断资源的请求在驱动程序初始化时就已经完成。

在传统的 PC 体系结构中 ，有些中断已经被固定下来。软盘设备正是这种情况，它的中断号总为 6。有时设备驱动程序可能不知道设备使用的中断号，对 PCI 设备来说这不是什么大问题，它们总是可以通过设备配置头知道其中断号。但对于 ISA 设备则没有取得中断号的方便方式，Linux 通过让设备驱动程序检测它们的中断号来解决这个问题。

让我们来看一下对 ISA 设备中断号的检测过程。设备驱动程序首先迫使 ISA 设备引起一个中断，系统中所有未被分配的中断都被打开。此时设备引发的中断可以通过可编程中断控制器来发送出去，在它接受到 CPU 的响应信号以后将中断号放置在数据线上，Linux 读取此数据并将其内容返回给设备驱动程序。非 0 结果则表示在此次检测中有中断发生，设备驱动程序然后将关闭检测并将所有未分配中断屏蔽掉，这样 ISA 设备驱动程序就成功地找到了设备的 IRQ 号。

基于 PCI 系统比基于 ISA 系统有更多的动态性。ISA 设备使用的中断引脚通常是通过硬件设备上的跳线来设置的。而每个 PCI 设备都对应一个配置头， PCI 设备在系统启动与初始化 PCI 时由 PCI BIOS 或 PCI 子系统来分配中断，将其放入配置头中，故而驱动程序可以方便获得 PCI 设备使用的中断号。

系统中可能存在许多 PCI 中断源，比如在使用 PCI-PCI 桥接器时。这些中断源的个数可能将超出系统可编程中断控制器的引脚数。此时 PCI 设备必须共享中断号，中断控制器上的一个引脚可能被多个 PCI 设备同时使用。Linux 让中断的第 1 个请求者申明此中断是否可以共享，中断的共享将导致 irq_action 数组中的一个入口同时指向几个 irqaction 数据结构，如图 11.5 所示。当共享中断发生时 Linux 将调用对应此中断源的所有中断处理过程。

2．Linux 对中断的处理

Linux 中断处理子系统的一个基本任务是将中断正确联系到中断处理代码中的正确位

置。这些代码必须了解系统的中断拓扑结构。例如在中断控制器上引脚 6 上发生的软盘控制器中断必须被辨认出的确来自软盘并同系统的软盘设备驱动的中断服务子程序联系起来。

中断发生时，Linux 首先读取系统可编程中断控制器中中断状态寄存器，判断出中断源，将其转换成 irq_action 数组中偏移值（例如来自软盘控制器引脚 6 的中断将被转换成对应于 irq_action 数组中的第 7 个指针），然后调用其相应的中断处理程序。

当 Linux 内核调用设备驱动程序的中断服务子程序时，必须找出中断产生的原因以及相应的解决办法，这是通过读取设备上的状态寄存器的内容来完成的。

下面我们结合输入/输出系统的层次结构来看一下中断在驱动程序工作的过程中的作用。

(1) 用户发出某种输入/输出请求。

(2) 调用驱动程序的 read() 函数或 request() 函数，将完成的输入/输出的指令送给设备控制器，现在设备驱动程序等待操作的发生。

(3) 一小段时间以后，硬设备准备好完成指令的操作，并产生中断信号标志事件的发生。

(4) 中断信号导致调用驱动程序的中断服务子程序，它将所要的数据从硬设备复制到设备驱动程序的缓冲区中，并通知正在等待的 read() 函数和 request()函数，现在数据可供使用。

(5) 在数据可供使用时，read()或 request()函数现在可将数据提供给用户进程。

上述过程是经过了简化了的，但却反映了中断的主要过程的主要方面。

11.2.6 驱动 DMA 工作

所有的 PC 都包含一个称为直接内存访问控制器或 DMAC 的辅助处理器，它可以用来控制在 RAM 和 I/O 设备之间数据的传送。DMAC 一旦被 CPU 激活，就可以自行传送数据；当数据传送完成之后，DMAC 发出一个中断请求。当 CPU 和 DMAC 同时访问同一内存单元时，所产生的冲突由一个称为内存仲裁器的硬件电路来解决。

使用 DMAC 最多的是磁盘驱动器和其他需要一次传送大量字节的慢速设备。因为 DMAC 的设置时间相当长，所以在传送数量很少的数据时直接使用 CPU 效率更高。

原来的 ISA 总线所使用的第一个 DMAC 非常复杂，难于对其进行编程。PCI 和 SCSI 总线所使用的最新 DMAC 依靠总线中的专用硬件电路,这就使设备驱动程序开发人员的开发工作变得简单。

到现在为止，我们已区分了 3 类内存地址：逻辑地址、线性地址以及物理地址，前两个在 CPU 内部使用，最后一个是 CPU 从物理上驱动数据总线所用的内存地址。但是，还有第 4 种内存地址，称为总线地址：它是除 CPU 之外的硬件设备驱动数据总线所用的内存地址。在 PC 体系结构中，总线地址和物理地址是一致的；但是在其他体系结构中，例如 Sun 的 SPARC 和 Compaq 的 Alpha 体系结构中，这两种地址是不同的。

从根本上说,内核为什么应该关心总线地址呢？这是因为在 DMA 操作中数据传送不用 CPU 的参与：I/O 设备和 DMAC 直接驱动数据总线。因此，在内核开始 DMA 操作时，必须把所涉及的内存缓冲区总线地址或写入 DMAC 适当的 I/O 端口、或写入 I/O 设备适当的 I/O 端口。

很多 I/O 驱动程序都使用直接内存访问控制器（DMAC）来加快操作的速度。DMAC 与设备的 I/O 控制器相互作用共同实现数据传送。后文中我们还会看到，内核中包含一组易用的例程来对 DMAC 进行编程。当数据传送完成时，I/O 控制器通过 IRQ 向 CPU 发出信号。

当设备驱动程序为某个 I/O 设备建立 DMA 操作时，必须使用总线地址指定所用的内存缓冲区。内核提供两个宏 virt_to_bus 和 bus_to_virt，分别把虚拟地址转换成总线地址或把总线地址转换成虚拟地址。

与 IRQ 一样，DMAC 也是一种资源，必须把这种资源动态地分配给需要它的设备驱动程序。驱动程序开始和结束 DMA 操作的方法依赖于总线的类型。

1．ISA 总线的 DMA

每个 ISA DMAC 只能控制有限个通道。每个通道都包括一组独立的内部寄存器，所以，DMAC 就可以同时控制几个数据的传送。

设备驱动程序通常使用下面的方式来申请和释放 ISA DMAC。设备驱动程序照样要靠一个引用计数器来检测什么时候任何进程都不再访问设备文件。驱动程序执行以下操作。

（1）在设备文件的 open( )方法中把设备的引用计数器加 1。如果原来的值是 0，那么，驱动程序执行以下操作：

- 调用 request_irq( )来分配 ISA DMAC 所使用的 IRQ 中断号；
- 调用 request_dma( )来分配 DMA 通道；
- 通知硬件设备应该使用 DMA 并产生中断。
- 如果需要，为 DMA 缓冲区分配一个存储区域

（2）当必须启动 DMA 操作时，在设备文件的 read（）和 write（）方法中执行以下操作：

- 调用 set_dma_mode( )把通道设置成读/写模式；
- 调用 set_dma_addr( )来设置 DMA 缓冲区的总线地址。（因为只有最低的 24 位地址会发给 DMAC，所以缓冲区必须在 RAM 的前 16MB 中）；
- 调用 set_dma_count( )来设置要发送的字节数；
- 调用 set_dma_dma( )来启用 DMA 通道；
- 把当前进程加入该设备的等待队列，并把它挂起，当 DMAC 完成数据传送操作时，设备的 I/O 控制器就发出一个中断，相应的中断处理程序会唤醒正在睡眠的进程；
- 进程一旦被唤醒，就立即调用 disable_dma( )来禁用这个 DMA 通道；
- 调用 get_dma_residue( )来检查是否所有的数据都已被传送。

（3）在设备文件的 release 方法中，减少设备的引用计数器。如果该值变成 0，就执行以下操作：

- 禁用 DMA 和对这个硬件设备上的相应中断；
- 调用 free_dma( )来释放 DMA 通道；
- 调用 free_irq( )来释放 DMA 所使用的 IRQ 线。

2．PCI 总线的 DMA

PCI 总线对于 DMA 的使用要简单得多，因为 DMAC 是集成到 I/O 接口内部的。在 open() 方法中，设备驱动程序照样必须分配一条 IRQ 线来通知 DMA 操作的完成。但是，并没有必要

分配一个 DMA 通道，因为每个硬件设备都直接控制 PCI 总线的电信号。要启动 DMA 操作，设备驱动程序在硬件设备的某个 I/O 端口中简单地写入 DMA 缓冲区的总线地址、传送方向以及数据大小，然后驱动程序就挂起当前进程。在最后一个进程关闭这个文件对象时，release 方法负责释放这条 IRQ 线。

11.2.7 I/O 空间的映射

很多硬件设备都有自己的内存，通常称之为 I/O 空间。例如，所有比较新的图形卡都有几 MB 的 RAM，称为显存，用它来存放要在屏幕上显示的屏幕影像。

1．地址映射

根据设备和总线类型的不同，PC 体系结构中的 I/O 空间可以在 3 个不同的物理地址范围之间进行映射。

（1）对于连接到 ISA 总线上的大多数设备

I/O 空间通常被映射到从 0xa0000 到 0xfffff 的物理地址范围，这就在 640K 和 1MB 之间留出了一段空间，这就是所谓的“ 洞 ”。

（2）对于使用 VESA 本地总线（VLB）的一些老设备

这主要是由图形卡使用的一条专用总线：I/O 空间被映射到从 0xe00000 到 0xffffff 的地址范围中，也就是 14MB 到 16MB 之间。因为这些设备使页表的初始化更加复杂，因此已经不生产这种设备了。

（3）对于连接到 PCI 总线的设备

I/O 空间被映射到很大的物理地址区间，位于 RAM 物理地址的顶端。这种设备的处理比较简单。

2．访问 I/O 空间

内核如何访问一个 I/O 空间单元？让我们从 PC 体系结构开始入手，这个问题很容易就可以解决，之后我们再进一步讨论其他体系结构。

不要忘了内核程序作用于虚拟地址，因此 I/O 空间单元必须表示成大于 PAGE_OFFSET 的地址。在后面的讨论中，我们假设 PAGE_OFFSET 等于 0xc0000000，也就是说，内核虚拟地址是在第 4GB。

内核驱动程序必须把 I/O 空间单元的物理地址转换成内核空间的虚拟地址。在 PC 体系结构中，这可以简单地把 32 位的物理地址和 0xc0000000 常量进行或运算得到。例如，假设内核需要把物理地址为 0x000b0fe4 的 I/O 单元的值存放在 t1 中，把物理地址为 0xfc000000 的 I/O 单元的值存放在 t2 中，就可以使用下面的表达式来完成这项功能：

```
t1 = *((unsigned char *)(0xc00b0fe4));
t2 = *((unsigned char *)(0xfc000000));
```

在第六章我们已经介绍过，在初始化阶段，内核已经把可用的 RAM 物理地址映射到虚拟地址空间第 4GB 的最初部分。因此，分页机制把出现在第 1 个语句中的虚拟地址 0xc00b0fe4 映射回到原来的 I/O 物理地址 0x000b0fe4，正好落在从 640K 到 1MB 的这段“ ISA 洞 ”中。这

正是我们所期望的。

但是，对于第 2 个语句来说，这里有一个问题，因为其 I/O 物理地址超过了系统 RAM 的最大物理地址。因此，虚拟地址 0xfc000000 就不需要与物理地址 0xfc000000 相对应。在这种情况下，为了在内核页表中包括对这个 I/O 物理地址进行映射的虚拟地址，必须对页表进行修改：这可以通过调用 ioremap( )函数来实现。ioremap( )和 vmalloc( )函数类似，都调用 get_vm_area( ) 建立一个新的 vm_struct 描述符，其描述的虚拟地址区间为所请求 I/O 空间区的大小。然后，ioremap( )函数适当地更新所有进程的对应页表项。

因此，第 2 个语句的正确形式应该为：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

第 1 条语句建立一个 2MB 的虚拟地址区间，从 0xfb000000 开始；第 2 条语句读取地址 0xfc000000 的内存单元。驱动程序以后要取消这种映射，就必须使用 iounmap( )函数。

现在让我们考虑一下除 PC 之外的体系结构。在这种情况下，把 I/O 物理地址加上 0xc0000000 常量所得到的相应虚拟地址并不总是正确的。为了提高内核的可移植性，Linux 特意包含了下面这些宏来访问 I/O 空间。

```
readb, readw, readl
```

分别从一个 I/O 空间单元读取 1、2 或者 4 个字节。

```
writeb, writew, writel
```

分别向一个 I/O 空间单元写入 1、2 或者 4 个字节。

```
memcpy_fromio, memcpy_toio
```

把一个数据块从一个 I/O 空间单元拷贝到动态内存中，另一个函数正好相反，把一个数据块从动态内存中拷贝到一个 I/O 空间单元。

```
memset_io
```

用一个固定的值填充一个 I/O 空间区域。

对于 0xfc000000 I/O 单元的访问推荐使用如下方法：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

使用这些宏，就可以隐藏不同平台访问 I/O 空间所用方法的差异。

11.2.8　设备驱动程序框架

由于设备种类繁多，相应的设备驱动程序也非常之多。尽管设备驱动程序是内核的一部分，但设备驱动程序的开发往往由很多人来完成，如业余编程高手、设备厂商等。为了让设备驱动程序的开发建立在规范的基础上，就必须在驱动程序和内核之间有一个严格定义和管理的接口，例如 SVR4 提出了 DDI/DDK 规范，其含义就是设备与驱动程序接口／设备驱动程序与内核接口（Device-Driver Interface／Driver-Kernel Interface)。通过这个规范，可以规范设备驱动程序与内核之间的接口。

Linux 的设备驱动程序与外接的接口与 DDI/DKI 规范相似，可以分为以下 3 部分。

(1) 驱动程序与内核的接口，这是通过数据结构 file_operations 来完成的。

(2) 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。

(3) 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。

根据功能，驱动程序的代码可以分为如下几个部分。

(1) 驱动程序的注册和注销。

(2) 设备的打开与释放。

(3) 设备的读和写操作。

(4) 设备的控制操作。

(5) 设备的中断和查询处理。

前三点我们已经给予了简单说明，后面我们还会结合具体程序给出进一步的说明。关于设备的控制操作可以通过驱动程序中的 ioctl()来完成,例如,对光驱的控制可以使用 cdrom_ioctl()。

与读写操作不同，ioctl()的用法与具体设备密切相关，例如，对于软驱的控制可以使用 floppy_ioctl()，其调用形式为：

```
static int floppy_ioctl(struct inode *inode, struct file *filp,
                        unsigned int cmd, unsigned long param)
```

其中 cmd 的取值及含义与软驱有关，例如，FDEJECT 表示弹出软盘。

除了 ioctl()，设备驱动程序还可能有其他控制函数，如 lseek（）等。

对于不支持中断的设备，读写时需要轮流查询设备的状态，以便决定是否继续进行数据传输，例如，打印机驱动程序在缺省时轮流查询打印机的状态。

如果设备支持中断，则可按中断方式进行。

11.3 块设备驱动程序

对于块设备来说，读写操作是以数据块为单位进行的，为了使高速的 CPU 同低速块设备能够协调工作，提高读写效率，操作系统设置了缓冲机制。当进行读写的时候，首先对缓冲区读写，只有缓冲区中没有需要读的数据或是需要写的数据没有地方写时，才真正地启动设备控制器去控制设备本身进行数据交换，而对于设备本身的数据交换同样也是同缓冲区打交道。

11.3.1 块设备驱动程序的注册

对于块设备来说，驱动程序的注册不仅在其初始化的时候进行而且在编译的时候也要进行注册。在初始化时通过 register_blkdev( ) 函数将相应的块设备添加到数组 blkdevs 中，该数组在 fs/block_dev.c 中定义如下：

```
static struct {
        const char *name;
        struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];
```

从 Linux 2.4 开始，块设备表的定义与下一节要介绍的字符设备表的定义有所不同。因为每种具体的块设备都有一套具体的操作，因而各自有一个类似于 file_operations 那样的

数据结构，称为 block_device_operations 结构，其定义为：

```
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);
        int (*check_media_change) (kdev_t);
        int (*revalidate) (kdev_t);
         struct module *owner;
};
```

如果说 file_operation 结构是连接虚拟的 VFS 文件的操作与具体文件系统的文件操作之间的枢纽，那么 block_device_operations 就是连接抽象的块设备操作与具体块设备操作之间的枢纽。

具体的块设备是由主设备号唯一确定的，因此，主设备号唯一地确定了一个具体的 block_device_operations 数据结构。

下面我们来看 register_blkdev（ ）函数的具体实现，其代码在 fs/block_dev.c 中：

```
int register_blkdev(unsigned int major, const char * name, struct block_device_operations
*bdops)
{
         if (major == 0) {
                 for (major = MAX_BLKDEV-1; major > 0; major--) {
                        if (blkdevs[major].bdops == NULL) {
                                blkdevs[major].name = name;
                                blkdevs[major].bdops = bdops;
                                return major;
                        }
                 }
                return -EBUSY;
         }
         if (major >= MAX_BLKDEV)
                 return -EINVAL;
        if (blkdevs[major].bdops && blkdevs[major].bdops != bdops)
                 return -EBUSY;
         blkdevs[major].name = name;
         blkdevs[major].bdops = bdops;
        return 0;
}
```

这个函数的第 1 个参数是主设备号，第 2 个参数是设备名称的字符串，第 3 个参数是指向具体设备操作的指针。如果一切顺利则返回 0，否则返回负值。如果指定的主设备号为 0，此函数将会搜索空闲的主设备号分配给该设备驱动程序并将其作为返回值。

那么，块设备注册到系统以后，怎样与文件系统联系起来呢，也就是说，文件系统怎么调用已注册的块设备，这还得从 file_operations 结构说起。

我们先来看一下块设备的 file_operations 结构的定义，其位于 fs/block_dev.c 中：

```
struct file_operations def_blk_fops = {
        open:           blkdev_open,
        release:        blkdev_close,
        llseek:         block_llseek,
         read:           generic_file_read,
```

```
        write:          generic_file_write,
        mmap:           generic_file_mmap,
        fsync:          block_fsync,
        ioctl:          blkdev_ioctl,
};
```

下面以 open()系统调用为例，说明用户进程中的一个系统调用如何最终与物理块设备的操作联系起来。在此，我们仅仅给出几个 open（）函数的调用关系，如图 11.6 所示。

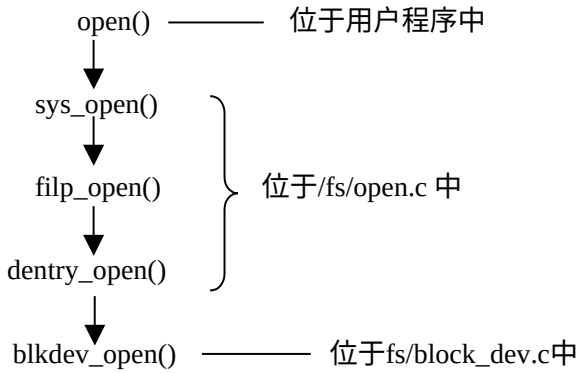

图 11.6　几个 open()函数的调用关系

如图 11.6 所示，当调用 open（）系统调用时，其最终会调用到 def_blk_fops 的 blkdev_open（）函数。blkdev_open（）函数的任务就是根据主设备号找到对应的 block_device_operations 结构，然后再调用 block_device_operations 结构中的函数指针 open 所指向的函数，如果 open 所指向的函数非空，就调用该函数打开最终的物理块设备。这就简单地说明了块设备注册以后，从最上层的系统调用到具体地打开一个设备的过程。

另外要说明的是，如果选择了通过设备文件系统 DevFS 进行注册，则调用 devfs_register_blkdev()函数，该函数的说明及代码在 fs/devfs/base.c 中定义如下：

```
/**
*      devfs_register_blkdev - Optionally register a conventional block driver.
  *      @major: The major number for the driver.
  *      @name: The name of the driver (as seen in /proc/devices).
  *      @bdops: The &block_device_operations structure pointer.
  *
  *      This function will register a block driver provided the "devfs=only"
  *      option was not provided at boot time.
  *      Returns 0 on success, else a negative error code on failure.
  */

int devfs_register_blkdev (unsigned int major, const char *name,
                           struct block_device_operations *bdops)
{
     if (boot_options & OPTION_ONLY) return 0;
     return register_blkdev (major, name, bdops);
}   /*  End Function devfs_register_blkdev  */
```

11.3.2　块设备基于缓冲区的数据交换

关于块缓冲区的管理在第八章虚拟文件系统中已有所描述，在这里我们从交换数据的角度来看一下基于缓冲区的数据交换的实现。

1．扇区及块缓冲区

块设备的每次数据传送操作都作用于一组相邻字节，我们称之为扇区。在大部分磁盘设备中，扇区的大小是 512 字节，但是现在新出现的一些设备使用更大的扇区（1024 和 2014 字节）。注意，应该把扇区作为数据传送的基本单元：不允许传送少于一个扇区的数据，而大部分磁盘设备都可以同时传送几个相邻的扇区。

所谓块就是块设备驱动程序在一次单独操作中所传送的一大块相邻字节。注意不要混淆块（block）和扇区（sector）：扇区是硬件设备传送数据的基本单元，而块只是硬件设备请求一次 I/O 操作所涉及的一组相邻字节。

在 Linux 中，块大小必须是 2 的幂，而且不能超过一个页面。此外，它必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此，在 PC 体系结构中，允许块的大小为 512、1024、2048 和 4096 字节。同一个块设备驱动程序可以作用于多个块大小，因为它必须处理共享同一主设备号的一组设备文件，而每个块设备文件都有自己预定义的块大小。例如，一个块设备驱动程序可能会处理有两个分区的硬盘，一个分区包含 Ext2 文件系统，另一个分区包含交换分区。

内核在一个名为 blksize_size 的表中存放块的大小；表中每个元素的索引就是相应块设备文件的主设备号和从设备号。如果 blksize_size[M]为 NULL，那么共享主设备号 M 的所有块设备都使用标准的块大小，即 1024 字节。

每个块都需要自己的缓冲区，它是内核用来存放块内容的 RAM 内存区。当设备驱动程序从磁盘读出一个块时，就用从硬件设备中所获得的值来填充相应的缓冲区；同样，当设备驱动程序向磁盘中写入一个块时，就用相关缓冲区的实际值来更新硬件设备上相应的一组相邻字节。缓冲区的大小一定要与块的大小相匹配。

2．块驱动程序的体系结构

下面我们说明通用块驱动程序的体系结构，以及在为缓冲区 I/O 操作时所涉及的主要成分。

块设备驱动程序通常分为两部分，即高级驱动程序和低级驱动程序，前者处理 VFS 层，后者处理硬件设备，如图 11.7 所示。

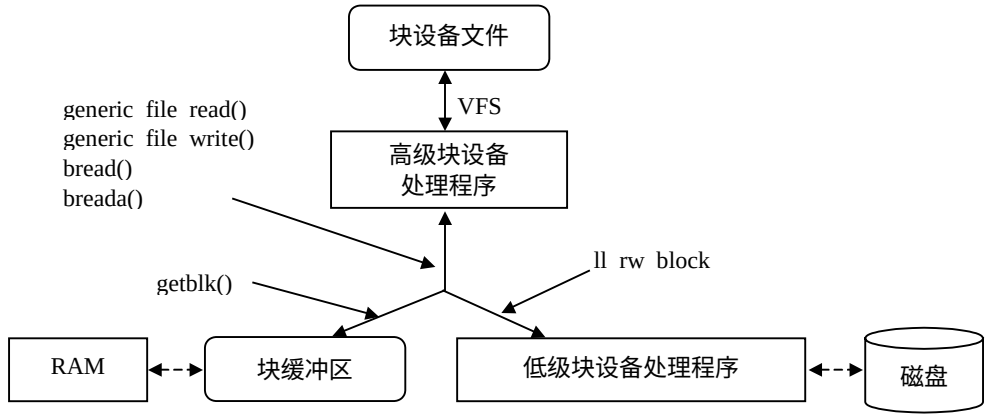

图 11.7　块设备驱动程序的体系结构

假设进程对一个设备文件发出 read( )或 write( )系统调用。VFS 执行对应文件对象的 read 或 write 方法，由此就调用高级块设备处理程序中的一个过程。这个过程执行的所有操作都与对这个硬件设备的具体读写请求有关。内核提供两个名为 generic_file_read ( )和 generic_file_write ( )通用函数来留意所有事件的发生。因此，在大部分情况下，高级硬件设备驱动程序不必做什么，而设备文件的 read 和 write 方法分别指向 generic_file_read ( )和 generic_file_write ( )方法。

但是，有些块设备的处理程序需要自己专用的高级设备驱动程序。典型的例子是软驱的设备驱动程序：它必须检查从上次访问磁盘以来，用户有没有改变驱动器中的磁盘；如果已插入一张新磁盘，那么设备驱动程序必须使缓冲区中所包含的旧数据无效。

即使高级设备驱动程序有自己的 read 和 write 方法，但是这两个方法通常最终还会调用 generic_file_read ( )和 generic_file_write ( )函数。这些函数把对 I/O 设备文件的访问请求转换成对相应硬件设备的块请求。所请求的块可能已在主存，因此 generic_file_read ( )和 generic_file_write ( )函数调用 getblk( )函数来检查缓冲区中是否已经预取了块，还是从上次访问以来缓冲区一直都没有改变。如果块不在缓冲区中，getblk( )就必须调用 ll_rw_block( )继续从磁盘中读取这个块，后面这个函数激活操纵设备控制器的低级驱动程序，以执行对块设备所请求的操作。

在 VFS 直接访问某一块设备上的特定块时，也会触发缓冲区 I/O 操作。例如，如果内核必须从磁盘文件系统中读取一个索引节点，那么它必须从相应磁盘分区的块中传送数据 。对于特定块的直接访问是由 bread( )和 breada( )函数来执行的，这两个函数又会调用前面提到过的 getblk( )和 ll_rw_block( )函数。

由于块设备速度很慢，因此缓冲区 I/O 数据传送通常都是异步处理的：低级设备驱动程序对 DMAC 和磁盘控制器进行编程来控制其操作，然后结束。当数据传送完成时，就会产生一个中断，从而第 2 次激活这个低级设备驱动程序来清除这次 I/O 操作所涉及的数据结构。

3．块设备请求

虽然块设备驱动程序可以一次传送一个单独的数据块，但是内核并不会为磁盘上每个被访问的数据块都单独执行一次 I/O 操作：这会导致磁盘性能的下降，因为确定磁盘表面块的

物理位置是相当费时的。取而代之的是，只要可能，内核就试图把几个块合并在一起，并作为一个整体来处理，这样就减少了磁头的平均移动时间。

当进程、VFS 层或者任何其他的内核部分要读写一个磁盘块时，就真正引起一个块设备请求。从本质上说，这个请求描述的是所请求的块以及要对它执行的操作类型（读还是写）。然而，并不是请求一发出，内核就满足它，实际上，块请求发出时 I/O 操作仅仅被调度，稍后才会被执行。这种人为的延迟有悖于提高块设备性能的关键机制。当请求传送一个新的数据块时，内核检查能否通过稍微扩大前一个一直处于等待状态的请求而满足这个新请求，也就是说，能否不用进一步的搜索操作就能满足新请求。由于磁盘的访问大都是顺序的，因此这种简单机制就非常高效。

延迟请求复杂化了块设备的处理。例如，假设某个进程打开了一个普通文件，然后，文件系统的驱动程序就要从磁盘读取相应的索引节点。高级块设备驱动程序把这个请求加入一个等待队列，并把这个进程挂起，直到存放索引节点的块被传送为止。

因为块设备驱动程序是中断驱动的，因此，只要高级驱动程序一发出块请求，它就可以终止执行。在稍后的时间低级驱动程序才被激活，它会调用一个所谓的策略程序从一个队列中取得这个请求，并向磁盘控制器发出适当的命令来满足这个请求。当 I/O 操作完成时，磁盘控制器就产生一个中断，如果需要，相应的处理程序会再次调用这个策略程序来处理队列中进程的下一个请求。

每个块设备驱动程序都维护自己的请求队列；每个物理块设备都应该有一个请求队列，以提高磁盘性能的方式对请求进行排序。因此策略程序就可以顺序扫描这种队列，并以最少地移动磁头而为所有的请求提供服务。

每个块设备请求都是由一个 request 结构来描述的，其定义于 include/linux/blkdev.h：

```
/*
  * Ok, this is an expanded form so that we can use the same
  * request for paging requests.
  */
struct request {
         struct list_head queue;
         int elevator_sequence;

         volatile int rq_status; /* should split this into a few status bits */
#define RQ_INACTIVE             (-1)
#define RQ_ACTIVE               1
#define RQ_SCSI_BUSY            0xffff
#define RQ_SCSI_DONE            0xfffe
#define RQ_SCSI_DISCONNECTING   0xffe0

         kdev_t rq_dev;
         int cmd;                /* READ or WRITE */
         int errors;
         unsigned long sector;
         unsigned long nr_sectors;
         unsigned long hard_sector, hard_nr_sectors;
         unsigned int nr_segments;
         unsigned int nr_hw_segments;
         unsigned long current_nr_sectors;
```

```
        void * special;
         char * buffer;
         struct completion * waiting;
         struct buffer_head * bh;
         struct buffer_head * bhtail;
         request_queue_t *q;
};
```

从代码注释可以知道，在 2.2 以前的版本中没有这么多域，很多域是为分页请求而增加的，我们暂且不予考虑。在此，我们只说明与块传送有关的域。为了描述方便起见，我们把 struct request 叫做请求描述符。

数据传送的方向存放在 cmd 域中：该值可能是 READ（把数据从块设备读到 RAM 中）或者 WRITE（把数据从 RAM 写到块设备中）。rq_status 域用来定义请求的状态：对于大部分块设备来说，这个域的值可能为 RQ_INACTIVE（请求描述符还没有使用）或者 RQ_ACTIVE（有效的请求，低级设备驱动程序要对其服务或正在对其服务）。

一次请求可能包括同一设备中的很多相邻块。rq_dev 域指定块设备，而 sector 域说明请求中第一个块对应的第一个扇区的编号。nr_sector 和 current_nr_sector 给出要传送数据的扇区数。sector、nr_sector 和 current_nr_sector 域都可以在请求得到服务的过程中而被动态修改。

请求块的所有缓冲区首部都被集中在一个简单链表中。每个缓冲区首部的 b_reqnext 域指向链表中的下一个元素，而请求描述符的 bh 和 bhtail 域分别指向链表的第一个元素和最后一个元素。

请求描述符的 buffer 域指向实际数据传送所使用的内存区。如果只请求一个单独的块，那么缓冲区只是缓冲区首部的 b_data 域的一个拷贝。然而，如果请求了多个块，而这些块的缓冲区在内存中又不是连续的，那么就使用缓冲区首部的 b_reqnext 域把这些缓冲区链接在一起。对于读操作来说，低级设备驱动程序可以选择先分配一个大的内存区来立即读取请求的所有扇区，然后再把这些数据拷贝到各个缓冲区。同样，对于写操作来说，低级设备驱动程序可以把很多不连续缓冲区中的数据拷贝到一个单独内存区的缓冲区中，然后再立即执行整个数据的传送。

另外，在严重负载和磁盘操作频繁的情况下，固定数目的请求描述符就可能成为一个瓶颈。空闲描述符的缺乏可能会强制进程等待直到正在执行的数据传送结束。因此，request_queue_t 类型（见下面）中的 wait_for_request 等待队列就用来对正在等待空闲请求描述符的进程进行排队。get_request_wait( )试图获取一个空闲的请求描述符，如果没有找到，就让当前进程在等待队列中睡眠；get_request( )函数与之类似，但是如果没有可用的空闲请求描述符，它只是简单地返回 NULL。

4．请求队列

请求队列只是一个简单的链表，其元素是请求描述符。每个请求描述符中的 next 域都指向请求队列的下一个元素，最后一个元素为空。这个链表的排序通常是：首先根据设备标识符，其次根据最初的扇区号。

如前所述，对于所服务的每个硬盘，设备驱动程序通常都有一个请求队列。然而，一些

设备驱动程序只有一个请求队列，其中包括了由这个驱动器处理的所有物理设备的请求。这种方法简化了驱动程序的设计，但是损失了系统的整体性能，因为不能对队列强制使用简单排序的策略。请求队列定义如下：

```
struct request_queue
{
        /*
         * the queue request freelist, one for reads and one for writes
         */
        struct request_list     rq[2];

        /*
         * Together with queue_head for cacheline sharing
         */
        struct list_head        queue_head;
        elevator_t              elevator;

        request_fn_proc         * request_fn;
        merge_request_fn        * back_merge_fn;
        merge_request_fn        * front_merge_fn;
        merge_requests_fn       * merge_requests_fn;
        make_request_fn         * make_request_fn;
        plug_device_fn          * plug_device_fn;
        /*
         * The queue owner gets to use this for whatever they like.
        * ll_rw_blk doesn't touch it.
         */
        void                    * queuedata;

        /*
         * This is used to remove the plug when tq_disk runs.
         */
        struct tq_struct        plug_tq;

        /*
         * Boolean that indicates whether this queue is plugged or not.
         */
        char                    plugged;

        /*
         * Boolean that indicates whether current_request is active or
         * not.
         */
        char                    head_active;

        /*
         * Is meant to protect the queue in the future instead of
         * io_request_lock
         */
        spinlock_t              queue_lock;

        /*
```

```
         * Tasks wait here for free request
         */
        wait_queue_head_t       wait_for_request;
  };
typedef struct request_queue request_queue_t;
```

其中，request_list 为请求描述符组成的空闲链表，其定义如下：

```
  struct request_list {
        unsigned int count;
         struct list_head free;
  };
```

有两个这样的链表，一个用于读，一个用于写。

elevator_t 结构描述的是为磁盘的电梯调度算法而设的数据结构。从 request_fn_proc 到 plug_device_fn 都是一些函数指针。例如 request_fn 是一个指针，指向类型为 request_fn_proc 的对象。而 request_fn_proc 则通过#typedef 定义为一种函数：

```
typedef void (request_fn_proc) (request_queue_t *q)
```

其余的函数也与此类似，这些指针（连同其他域）都是在相应设备初始化时设置好的。需要对一个块设备进行操作时，就为之设置好一个数据结构 request_queue。并将其挂入相应的请求队列中。

这里要说明的是，request_fn( )域包含驱动程序的策略程序的地址，策略程序是低级块设备驱动程序的关键函数，为了开始传送队列中的一个请求所指定的数据，它与物理块设备（通常是磁盘控制器）真正打交道。

5．块设备驱动程序描述符

驱动程序描述符是一个 blk_dev_struct 类型的数据结构，其定义如下：

```
   struct blk_dev_struct {
        /*
        * queue_proc has to be atomic
        */
         request_queue_t         request_queue;
         queue_proc              *queue;
         void                    *data;
   };
```

在这个结构中，其主体是请求队列 request_queue；此外，还有一个函数指针 queue，当这个指针为非 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。这个指针也在设备初始化时就设置好，另一个指针 data 是辅助 queue 函数找到特定设备的请求队列。

所有块设备的描述符都存放在 blk_dev 表中：

```
struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

每个块设备都对应着数组中的一项，可以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 ( )，如果数据存在缓冲区中或缓冲区还可以存放数据，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，如图 11.8 所示。如果在加入请求队列结构的时候该设备没有请求，则马上响应该请求，否则

将其追加到请求任务队列尾顺序执行。

图 11.8 表示每个请求有指向一个或多个 buffer_hear 结构的指针，每个请求读写一块数据。如果系统对 buffer_head 结构上锁，则进程会等待到对此缓冲区的块操作完成。一旦设备驱动程序完成了请求则它必须将每个 buffer_heard 结构从 request 结构中清除，将它们标记成已更新状态并对它们解锁。对 buffer_head 的解锁将唤醒所有等待此块操作完成的睡眠进程，然后 request 数据结构被标记成空闲以便被其他块请求使用。

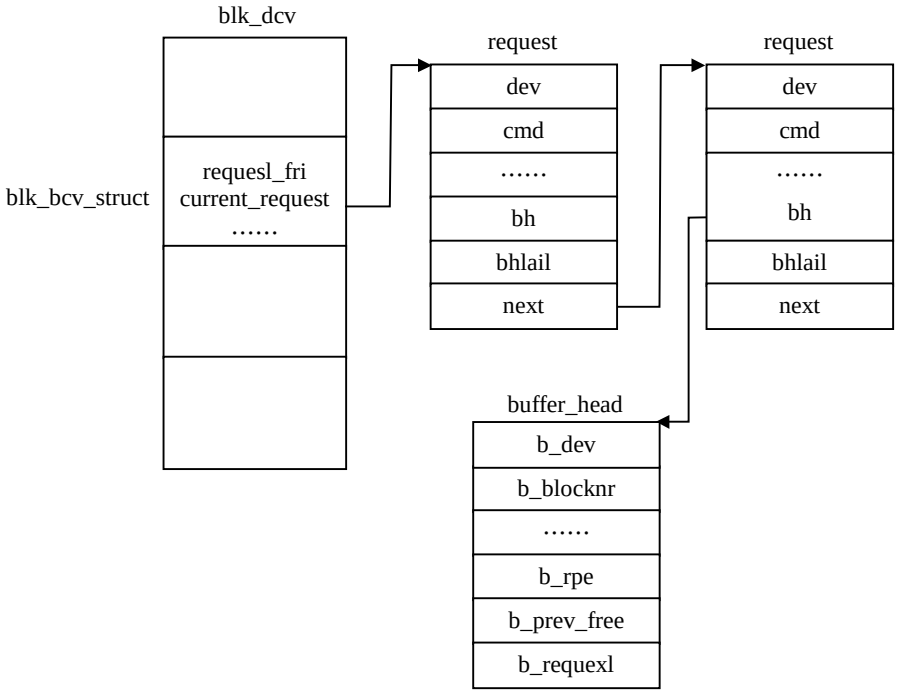

图 11.8　块设备读写请求

11.3.3　块设备驱动程序的几个函数

所有对块设备的读写都是调用 generic_file_read ( )和 generic_file_write ( ) 函数，这两个函数的原型如下：

```
ssize_t generic_file_read(struct file * filp, char * buf, size_t count, loff_t    *ppos)
ssize_t    generic_file_write(struct file *file,const char *buf,size_t count, loff_t  *ppos)
```

其参数的含义如下。

filp：和这个设备文件相对应的文件对象的地址。

Buf：用户态地址空间中的缓冲区的地址。generic_file_read（）把从块设备中读出的数据写入这个缓冲区；反之，generic_file_write（）从这个缓冲区中读取要写入块设备的数据。

Count：要传送的字节数。

ppos：设备文件中的偏移变量的地址；通常，这个参数指向 filp->f_pos，也就是说，

指向设备文件的文件指针。

只要进程对设备文件发出读写操作，高级设备驱动程序就调用这两个函数。例如，superformat 程序通过把块写入/dev/fd0 设备文件来格式化磁盘，相应文件对象的 write 方法就调用 generic_file_write（）函数。这两个函数所做的就是对缓冲区进行读写，如果缓冲区不能满足操作要求则返回负值，否则返回实际读写的字节数。每个块设备在需要读写时都调用这两个函数。

下面介绍几个低层被频繁调用的函数。

bread( )和 breada( ) 函数

bread( )函数检查缓冲区中是否已经包含了一个特定的块；如果还没有，该函数就从块设备中读取这个块。文件系统广泛使用 bread( )从磁盘位图、索引节点以及其他基于块的数据结构中读取数据（注意当进程要读块设备文件时是使用 generic_file_read（）函数，而不是使用 bread( )函数）。该函数接收设备标志符、块号和块大小作为参数，其代码在 fs/buffer.c 中：

```
/**
 *      bread() - reads a specified block and returns the bh
 *      @block: number of block
 *      @size: size (in bytes) to read
 *
  *      Reads a specified block, and returns buffer head that
  *      contains it. It returns NULL if the block was unreadable.
  */
struct buffer_head * bread(kdev_t dev, int block, int size)
{
        struct buffer_head * bh;

       bh = getblk(dev, block, size);
       touch_buffer(bh);
       if (buffer_uptodate(bh))
                return bh;
        ll_rw_block(READ, 1, &bh);
         wait_on_buffer(bh);
         if (buffer_uptodate(bh))
                 return bh;
        brelse(bh);
        return NULL;
}
```

对该函数解释如下。

- 调用 getblk( )函数来查找缓冲区中的一个块；如果这个块不在缓冲区中，那么 getblk( )就为它分配一个新的缓冲区。
- 调用 buffer_uptodate（）宏来判断这个缓冲区是否已经包含最新数据，如果是，则 getblk( )结束。
- 如果缓冲区中没有包含最新数据，就调用 ll_rw_block( )函数启动读操作。
- 等待，直到数据传送完成为止。这是通过调用一个名为 wait_on_buffer( )的函数来实现的，该函数把当前进程插入 b_wait 等待队列中，并挂起当前进程直到这个缓冲区被开锁

为止。

breada( )和 bread( )十分类似，但是它除了读取所请求的块之外，还要另外预读一些其他块。注意不存在把块直接写入磁盘的函数。写操作永远都不会成为系统性能的瓶颈，因为写操作通常都会延时。

2. ll_rw_block( )函数

ll_rw_block( )函数产生块设备请求；内核和设备驱动程序的很多地方都会调用这个函数。该函数的原型如下：

```
void  ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
```

其参数的含义如下。

- 操作类型 rw，其值可以是 READ、WRITE、READA 或者 WRITEA。最后两种操作类型和前两种操作类型之间的区别在于，当没有可用的请求描述符时后两个函数不会阻塞。
- 要传送的块数 nr。
- 一个 bhs 数组，有 nr 个指针，指向说明块的缓冲区首部（这些块的大小必须相同，而且必须处于同一个块设备）。

该函数的代码在 block/ll_rw_blk.c 中：

```
    void ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
{
        unsigned int major;
        int correct_size;
        int i;

        if (!nr)
                return;

         major = MAJOR(bhs[0]->b_dev);

         /* Determine correct block size for this device. */
        correct_size = get_hardsect_size(bhs[0]->b_dev);

         /* Verify requested block sizes. */
         for (i = 0; i < nr; i++) {
                struct buffer_head *bh = bhs[i];
                if (bh->b_size % correct_size) {
                       printk(KERN_NOTICE "ll_rw_block: device %s: "
                              "only %d-char blocks implemented (%u)\n",
                              kdevname(bhs[0]->b_dev),
                              correct_size, bh->b_size);
                        goto sorry;
                }
        }

         if ((rw & WRITE) && is_read_only(bhs[0]->b_dev)) {
                printk(KERN_NOTICE "Can't write to read-only device %s\n",
                      kdevname(bhs[0]->b_dev));
               goto sorry;
         }
```

```
        for (i = 0; i < nr; i++) {
                struct buffer_head *bh = bhs[i];

               /* Only one thread can actually submit the I/O. */
               if (test_and_set_bit(BH_Lock, &bh->b_state))
                        continue;

                /* We have the buffer lock */
                atomic_inc(&bh->b_count);
                bh->b_end_io = end_buffer_io_sync;

                switch(rw) {
                case WRITE:
                       if (!atomic_set_buffer_clean(bh))
                                /* Hmmph! Nothing to write */
                                goto end_io;
                        __mark_buffer_clean(bh);
                       break;

                case READA:
                case READ:
                       if (buffer_uptodate(bh))
                                /* Hmmph! Already have it */
                               goto end_io;
                       break;
                default:
                        BUG();
        end_io:
                        bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
                       continue;
               }

                submit_bh(rw, bh);
        }
        return;

sorry:
        /* Make sure we don't get infinite dirty retries.. */
       for (i = 0; i < nr; i++)
              mark_buffer_clean(bhs[i]);
}
```

下面对该函数给予解释。

进入 ll_rw_block()以后，先对块大小作一些检查；如果是写访问，则还要检查目标设备是否可写。内核中有个二维数组 ro_bits，定义于 drivers/block/ll_rw_blk.c 中：

```
static long ro_bits[MAX_BLKDEV][8];
```

每个设备在这个数组中都有个标志，通过系统调用 ioctl()可以将一个标志位设置成 1 或 0，表示相应设备为只读或可写，而 is_read_only（）就是检查这个数组中的标志位是否为 1。

接下来，就通过第 2 个 for 循环依次处理对各个缓冲区的读写请求了。对于要读写的每个块，首先将其缓冲区加上锁，还要将其 buffer_head 结构中的函数指针 b_end_io 设置成指向 end_buffer_io_sync，当完成对给定块的读写时，就调用该函数。此外，对于待写的缓冲区，其 BH_Dirty 标志位应该为 1，否则就不需要写了，而既然写了，就要把它清 0，并通过 __mark_buffer_clean(bh)将缓冲区转移到干净页面的 LRU 队列中。反之,对于待读的缓冲区，其 buffer_uptodate () 标志位为 0，否则就不需要读了。每个具体的设备就好像是个服务器，所以最后具体的读写是通过 submit_bh （） 将读写请求提交各“ 服务器 ”完成的，每次读写一个块，该函数的代码也在同一文件中，读者可以自己去读。

11.3.4　RAM 盘驱动程序的实现

1．RAM 盘的硬件

利用 RAM 盘的驱动程序可以访问内存的任何部分，它的主要用途是保留一部分内存并象普通磁盘一样来使用它。

RAM 盘的思想很简单，块设备是有两个操作的命令的存储介质：即写数据块和读数据块。通常这些数据存储于旋转存储设备上如软盘和硬盘，RAM 盘则简单得多，它利用预先分配的主存来存储数据块。因此不存在像磁盘那样的寻道操作，其读写操作只是在内存间进行的。RAM 盘具有快速存取的优点（没有寻道和旋转延迟的时间），适合于存储需要频繁存取的数据。

操作系统根据对 RAM 盘的需求为它分配内存的大小，RAM 盘被分成几块，每块的大小同实际磁盘的块的大小相同。一个 RAM 盘驱动程序支持将存储器中的若干区域当作 RAM 盘来使用，不同的 RAM 盘用从设备号来区分。

2．Linux 中 RAM 盘的驱动程序

RAM 盘的驱动程序同其他所有的驱动程序一样都是由一组函数组成，对 RAM 盘的操作实际上是对内存的操作，它不需要中断机制，故 RAM 盘的驱动程序不包括中断服务子程序.。一般我们对于一个驱动程序的分析是在了解硬件的基础上从该设备所提供的操作入手的，相应的写驱动程序也应该是这样的。

下面是 RAM 盘操作的结构：

```
s static struct block_device_operations rd_bd_op = {
        owner:          THIS_MODULE,
        open:           rd_open,
        ioctl:          rd_ioctl,
 };
```

在 Linux 中，RAM 盘的主设备号是 1。在 rd_open （ ） 函数中,它首先检测设备号 INITRD_MINOR，由于 INITRD 是在系统一启动的时候就已经创建，其中映像的是操作系统从偏移地址 0 开始的内容，即内核空间，如果是内核空间，其接口需要相应的发生变换即：filp->f_op = &initrd_fops。

```
static struct file_operations initrd_fops = {
```

```
        read:           initrd_read,
        release:        initrd_release,
};
```

对于 INITRD 盘的操作用户只有读和释放的权限而无写的权限。initrd_read ( ) 函数执行的是从内核区进行的读操作，故而是利用 memcpy_tofs (buf， (char *) initrd_start+file->f_pos，count) 去完成的。initrd_release ( ) 函数在判断没有用户操作这个设备之后，以页的方式把 INITRD 盘所占的内存释放掉。

在普通 RAM 盘接口中的另一个函数为 rd_ioctl ( )，同其他设备驱动程序一样是执行一些输入/输出的控制操作。

RAM 盘的驱动程序可以以模块的形式进行编译，所以驱动程序中还有一些关于模块的操作，关于模块的知识请参见上一章。

```
int init_module(void)；          /*执行 rd_init()*/
```

void cleanup_module(void)　释放模块的时候首先要把保护的缓冲区标志为无效，然后取消 ramdisk 的注册。

RAM 盘中还有 3 个函数比较重要，如下所述。

（1） int　identify_ramdisk_image(kdev_t device, struct file *fp, int start_block)；

检测设备中被映像文件的文件系统的类型，返回被映像的最大块数。

(2) static void rd_load_image(kdev_t device,int offset)

把文件映像到 RAM 盘，从偏移地址 offset 开始。

(3) void rd_load ( )

用软盘启动的时候装载映像文件到 ROOT_DEV 中。

至此，我们对于 Linux 中关于 RAM 的实现有一个大体的了解，下面我们再看一个较复杂的驱动程序——硬盘的驱动程序。

11.3.5　硬盘驱动程序的实现

1．磁盘硬件

所有实际的磁盘都组织成许多柱面，每个柱面上的磁道数和磁头数相同。磁道又被划分成许多扇区。如果每条磁道上的扇区数相同，则外圈磁道的数据的密度就会小一些，这就意味着会牺牲一些磁盘容量，也意味着必须存在更复杂的系统。现代大容量的硬盘中外圈磁道有的扇区数比内圈多，这就是 IDE（Integrated Drive Electronics）驱动器，它内置的电子器件屏蔽了复杂的细节，对于操作系统来说仍呈现出简单的几何结构，每条磁道具有相同的扇区。

下面我们看一下硬盘控制器的硬件结构，以便于我们进一步了解硬盘驱动程序。对于驱动程序，了解硬盘的各种寄存器以及寄存器的各个位是重要的，表 11.1(a)～(e)给出各个寄存器的具体描述。

表 11.1　　(a) IDE 硬盘的控制寄存

A2	A1	A0	读功能	写功能
0	0	0	数据	数据
0	0	0	出错	写预补偿
0	0	0	扇区计数	扇区计数
0	1	1	扇区号（0～7）	扇区号（0～7）
1	0	0	柱面号低位（8～15）	柱面号低位（8～15）
1	0	0	柱面号高位（16～23）	柱面号高位（16～23）
1	1	0	选择驱动器磁头（24～27）	选择驱动器磁头（24～27）
1	1	1	状态	状态

表 11.1　　(b)驱动器/磁头寄存器的选择域

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA：0 = 柱面数/磁头/扇区模式
　　1 = 逻辑寻块模式
D：　0 = 主驱动器
　　1 = 从驱动器
HSn：CHS 模式：在 CHS 模式下磁头选择
　　LBA 模式：块选择位 24～27

表 11.1　　(c)出错标志寄存器各个域的功能

位	功能
D0	当它置 1 时，表明找到了扇区但不能找到数据地址的标记
D1	当它置 1 时，表明回零道命令时，发生了 1024 个脉冲仍未找到 0 道
D2	当它置 1 时，表明无效命令
D4	当它置 1 时，表示磁盘转了 8 圈仍未找到所要求的参数或出现 ID 段 CRC 错
D6	当它置 1 时，表明数据段 CRC 错或未找到数据地址标志
D7	当它置 1 时，表明在 ID 段有坏块标志

表 11.1　　(d)状态寄存器各个域的含义

位	功能
D0	当它置位时，,标志错误寄存器存在错误标志
D1	当它置位时，表示正在进行命令不能接受新的命令
D2	未用
D3	当它置位时，表明 WDC 请求与主机交换数据
D4	寻找完成标志
D5	当它置位时，表示有写错误
D6	当它置位时，表示盘准备好
D7	命令写入时将它置位，命令结束时将它清除

表 11.1　　(e)命令寄存器的功能

命令	D7D6D5D4D3D2D1D0
回零道	0001R3R2R1R0
寻道	1111R3R2R1R0
读扇区	0010IM0T
写扇区	0011OM0T
扫描 ID	01000000
格式化	01010000

R3～R0 的编码规定了步进脉冲的间隔

I 是中断允许位当 I=0 时，将在 BDRQ（BDRQ 为扇区缓冲区数据请求输出信号）有效时产生中断
　　当 I=1 时，是在命令结束时产生中断

M 是多扇区读写标志 当 M=0 时，只写一个扇区
　　当 M=1 时，传送多个扇区

T　是允许重试命令　当 T=0 时，允许重试
　　当 T=1 时，不允许重试

上面对于硬盘控制器的几个寄存器的功能作了简要的说明，由此我们可以了解到，对于硬盘的很大一部分工作，由硬盘控制器就可以完成。而对于软盘来说，其控制器则简单得多，我们需要编程去完成各种功能，这样软盘驱动程序就变得比较复杂。在这儿我们只讨论硬盘驱动程序。

2．Linux 中硬盘驱动程序的实现

接下来我们将要讨论的驱动程序在 drivers/ide/hd.c 中，在文件为 include/linux/hdreg.h 中，定义了控制器寄存器、状态位和命令、数据结构和原形。这些宏定义可以根据其名字并结合上面所说的硬件内容去理解。

Linux 中，硬盘被认为是计算机的最基本的配置，所以在装载内核的时候，硬盘驱动程序必须就被编译进内核，不能作为模块编译。硬盘驱动程序提供内核的接口为：

```
static struct block_device_operations hd_fops = {
      open:           hd_open,
      release:        hd_release,
      ioctl:          hd_ioctl,
};
```

对硬盘的操作只有 3 个函数。我们来看一下 hd_open（）和 hd_release（）函数，打开操作首先检测了设备的有效性，接着测试了它的忙标志，最后对请求硬盘的总数加 1，来标识对硬盘的请求个数，hd_release（）函数则将请求的总数减 1。

前面说过，对于块设备的读写操作是先对缓冲区操作，但是当需要真正同硬盘交换数据的时候，驱动程序又干了些什么？在 hd.c 中有一个函数 hd_out（），可以说它在实际的数据交换中起着主要的作用。它的原形是：

```
static void  hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
                    unsigned int head,unsigned int cyl,unsigned int cmd,
```

```
                    void (*intr_addr)(void));
```

其中参数 drive 是进行操作的设备号；nsect 是每次读写的扇区数；sect 是读写的开始扇区号；head 是读写的磁头号；cmd 是操作命令控制命令字。

通过这个函数向硬盘控制器的寄存器中写入数据，启动硬盘进行实际的操作。同时这个函数也配合完成 cmd 命令相应的中断服务子程序，通过 SET_INIT(intr_addr) 宏定义将其地址赋给 DEVICE_INTR。

hd_request () 函数就是通过这个函数进行实际的数据交换，同其他驱动程序不同的是该函数还要根据每个命令的不同来确定一些参数，最基本的是读写方式的确定，关于硬盘的读写方式有两种，一种是单扇区的读写，另一种是多扇区的读写，单扇区的读写是指每次操作只对一个扇区操作，而多扇区则指每次对多个扇区进行操作，不同的方式其中断服务子程序不同，其相应的地址就作为参数传给 hd_out ( ) ，由它设置 DEVICE_INIT。hd_request( ) 函数确定的其他参数也就是 hd_out()所需要的参数。

我们知道块设备的实际数据交换需要中断服务子程序的配合，在本驱动程序中的中断服务子程序有以下几个主要函数。

(1) void unexpected_hd_interrupt(void)

功能：对不期望的中断进行处理(设置 SET_TIMER)。

(2) static void bad_rw_intr(void)

功能：当硬盘的读写操作出现错误时进行处理。

- 每重复 4 次磁头复位；
- 每重复 8 次控制器复位；
- 每重复 16 次放弃操作。

(3) static void recal_intr(void)

功能：重新进行硬盘的本次操作。

(4) static void read_intr(void)

功能：从硬盘读数据到缓冲区。

(5) static void write_intr(void)

功能：从缓冲区读数据到硬盘。

(6) static void hd_interupt(void)

功能：决定硬盘中断所要调用的中断程序。

在注册的时候，同硬盘中断联系的是 hd_interupt ( )，也就是说当硬盘中断到来的时候，执行的函数是 hd_interupt ( )，在此函数中调用 DEVICE_INTR 所指向的中断函数，如果 DEVICE_INTR 为空，则执行 unexpected_hd_interrupt()函数。

对硬盘的操作离不开控制寄存器，为了控制磁盘要经常去检测磁盘的运行状态，在本驱动程序中有一系列的函数是完成这项工作的，check_status ( ) 检测硬盘的运行状态，如果出现错误则进行处理。contorller_ready ( ) 检测控制器是否准备好。drive_busy ( ) 检测硬盘设备是否处于忙态。当出现错误的时候，由 dump_status ( ) 函数去检测出错的原因。wait_DRQ ( ) 对数据请求位进行测试。

当硬盘的操作出现错误的时候，硬盘驱动程序会把它尽量在接近硬件的地方解决掉，其方法是进行重复操作，这些在 bad_rw_intr ( ) 中进行，与其相关的函数有

reset_controller ( ) 和 reset_hd ( ) 。

函数 hd_init ( ) 是对硬盘进行初始化的，这个函数的过程同其他块设备基本一致。还有一些对硬盘的参数进行测试和确定的函数，这里就不一一说明。

通过对 RAM 盘和硬盘驱动程序的分析我们对一个块设备的驱动程序应已经有一个大概的认识，那么对于其他的块设备驱动程序我们就可以根据硬盘驱动程序的分析方法去分析，在分析的时候要切记不能脱离硬件控制器，也不能一开始就扎入技术细节，那样我们就会陷入一个不可自拔的泥潭。如果要写一个块设备的驱动程序，我们除了要了解硬件寄存器外还要弄清具体驱动程序所要解决的问题，然后再根据驱动程序的写法去做进一步的工作。

11.4 字符设备驱动程序

我们说，传统的 UNIX 设备驱动是以主／从设备号为主的，每个设备都有唯一的主设备号和从设备号，而内核中则有块设备表和字符设备表，根据设备的类型和主设备号便可在设备表中找到相应的驱动函数，而从设备号则一般只用作同类型设备中具体设备项的编号。但是，由于字符设备的多样性，有时候也用从设备号作进一步的归类。这方面典型的例子就是终端设备 TTY。TTY 设备是字符设备，主设备号为 4，但是当从设备号为 0 时就表示当前虚拟控制终端，而 1～63 表示 63 个可能的虚拟控制终端，64～255 则表示 192 个可能的串口 URAT（通用异步收发器）和连接在上面的实际终端设备。这里所谓实际终端设备通常是指老式的 CRT 终端，或“ 笨终端 ”，而虚拟控制终端则通常是建立在 PC 机的显示器和图形接口卡基础上的。显然，这里相同的主设备号并不意味着相同的驱动程序。另一个例子是主设备号为 1 的“ 字符设备 ”，当从设备号为 1 时表示物理内存/dev/mem，为 2 时则表示内核的虚存空间／dev/kmem/，为 3 时表示“ 空设备 ”/dev/null，而从设备号 8 则表示随机数生成器/dev/random。类似的情况在块设备中也有（例如软盘设备），但是很少。随同 Linux 内核发布的 Documentation/devices.txt 文件中列举了对块设备和字符设备两种主设备号和从设备号的分配和指定，读者可以参阅。

11.4.1 简单字符设备驱动程序

我们来看一个最简单的字符设备，即“ 空设备 ”/dev/null。大家知道，应用程序在运行的过程中，一般都要通过其预先打开的标准输出通道或标准出错通道在终端显示屏上输出一些信息，但是有时候（特别是在批处理中）不宜在显示屏上显示信息，又不宜将这些信息重定向到一个磁盘文件中，而要求直接使这些信息流入“ 下水道 ”而消失，这时候就可以用／dev/null 来起这个“ 下水道 ”的作用，这个设备的主设备号为 1。

如前所述，主设备号为 1 的设备其实不是“ 设备 ”，而都是与内存有关，或是在内存中（不必通过外设）就可以提供的功能，所以其主设备号标识符为 MEM_MAJOR，其定义于 include/linux/major.h 中：

```
#define MEM_MAJOR       1
```

其 file_operatins 结构为 memory_fops，定义于 dreivers/char/mem.c 中：

```
static struct file_operations memory_fops = {
        open:           memory_open,    /* just a selector for the real open */
};
```

因为主设备号为 1 的字符设备并不能唯一地确定具体的设备驱动程序，因此需要根据从设备号来进行进一步的区分，所以 memory_fops 还不是最终的 file_operations 结构，还需要由 memory_open（）进一步加以确定和设置，其代码在同一文件中：

```
static int memory_open(struct inode * inode, struct file * filp)
{
         switch (MINOR(inode->i_rdev)) {
                 case 1:
                        filp->f_op = &mem_fops;
                         break;
                case 2:
                       filp->f_op = &kmem_fops;
                        break;
                 case 3:
                        filp->f_op = &null_fops;
                         break;
                       …
                }
      if (filp->f_op && filp->f_op->open)
                  return filp->f_op->open(inode,filp);
            return 0;
    }
```

因为/dev/null 的从设备号为 3，所以其 file_operations 结构为 null_fops：

```
    static struct file_operations null_fops = {
         llseek:         null_lseek,
         read:           read_null,
         write:          write_null,
  };
```

由于这个结构中函数指针 open 为 NULL，因此在打开这个文件时没有任何附加操作。当通过 write()系统调用写这个文件时，相应的驱动函数为 write_null()，其代码为：

```
static ssize_t write_null(struct file * file, const char * buf,
                          size_t count, loff_t *ppos)
{
         return count;
 }
```

从中可以看出，这个函数什么也没做，仅仅返回 count，假装要求写入的字节已经写好了，而实际把写的内容丢弃了。

再来看一下读操作又做了些什么，read_null（）的代码为：

```
static ssize_t read_null(struct file * file, char * buf,
                         size_t count, loff_t *ppos)
{
         return 0;
}
```

返回 0 表示从这个文件读了 0 个字节，但是并没有到达（永远也不会到达）文件的末尾。

当然，字符设备的驱动程序不会都这么简单，但是总的框架是一样的。

11.4.2 字符设备驱动程序的注册

具有相同主设备号和类型的每类设备文件都是由 device_struct 数据结构来描述的，该结构定义于 fs/devices.c：

```
struct device_struct {
        const char * name;
        struct file_operations * fops;
};
```

其中，name 是某类设备的名字，fops 是指向文件操作表的一个指针。所有字符设备文件的 device_struct 描述符都包含在 chrdevs 表中：

```
static struct device_struct chrdevs[MAX_CHRDEV];
```

该表包含有 255 个元素，每个元素对应一个可能的主设备号，其中主设备号 255 为将来的扩展而保留的。表的第一项为空，因为没有一个设备文件的主设备号是 0。

chrdevs 表最初为空。register_chrdev( )函数用来向其中的一个表中插入一个新项，而 unregister_chrdev( )函数用来从表中删除一个项。我们来看一下 register_chrdev ()的具体实现：

```
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
 {
        if (major == 0) {
                write_lock(&chrdevs_lock);
                for (major = MAX_CHRDEV-1; major > 0; major--) {
                        if (chrdevs[major].fops == NULL) {
                                chrdevs[major].name = name;
                                chrdevs[major].fops = fops;
                                write_unlock(&chrdevs_lock);
                                return major;
                        }
                }
                write_unlock(&chrdevs_lock);
               return -EBUSY;
        }
        if (major >= MAX_CHRDEV)
               return -EINVAL;
        write_lock(&chrdevs_lock);
        if (chrdevs[major].fops && chrdevs[major].fops != fops) {
                write_unlock(&chrdevs_lock);
               return -EBUSY;
        }
        chrdevs[major].name = name;
        chrdevs[major].fops = fops;
        write_unlock(&chrdevs_lock);
        return 0;
}
```

从代码可以看出，如果参数 major 为 0，则由系统自动分配第 1 个空闲的主设备号，并把设备名和文件操作表的指针置于 chrdevs 表的相应位置。

例如，可以按如下方式把并口打印机驱动程序的相应结构插入到 chrdevs 表中：

```
register_chrdev(6, "lp", &lp_fops);
```

该函数的第 1 个参数表示主设备号，第 2 个参数表示设备类名，最后一个参数是指向文件操作表的一个指针。

如果设备驱动程序被静态地加入内核，那么，在系统初始化期间就注册相应的设备文件类。但是，如果设备驱动程序作为模块被动态装入内核，那么，对应的设备文件在装载模块时被注册，在卸载模块时被注销。

字符设备被注册以后，它所提供的接口，即 file_operations 结构在 fs/devices.c 中定义如下：

```
/*
  * Dummy default file-operations: the only thing this does
  * is contain the open that then fills in the correct operations
  * depending on the special file...
  */
 static struct file_operations def_chr_fops = {
        open:           chrdev_open,
};
```

由于字符设备的多样性，因此，这个缺省的 file_operations 仅仅提供了打开操作，具体字符设备文件的 file_operations 由 chrdev_open（）函数决定：

```
  /*
  * Called every time a character special file is opened
  */
int chrdev_open(struct inode * inode, struct file * filp)
{
        int ret = -ENODEV;

        filp->f_op=get_chrfops(MAJOR(inode->i_rdev), MINOR(inode->i_rdev));
        if (filp->f_op) {
                ret = 0;
                if (filp->f_op->open != NULL) {
                        lock_kernel();
                         ret = filp->f_op->open(inode,filp);
                        unlock_kernel();
                }
        }
        return ret;
 }
```

首先调用 MAJOR（）和 MINOR（）宏从索引节点对象的 i_rdev 域中取得设备驱动程序的主设备号和从设备号，然后调用 get_chrfops（）函数为具体设备文件安装合适的文件操作。如果文件操作表中定义了 open 方法，就调用它。

注意，最后一次调用的 open( )方法就是对实际设备操作，这个函数的工作是设置设备。通常，open(　)函数执行如下操作。

- 如果设备驱动程序被包含在一个内核模块中，那么把引用计数器的值加 1，以便只有把设备文件关闭之后才能卸载这个模块。
- 如果设备驱动程序要处理多个同类型的设备，那么，就使用从设备号来选择合适的驱动程序，如果需要，还要使用专门的文件操作表选择驱动程序。

- 检查该设备是否真正存在，现在是否正在工作。
- 如果必要，向硬件设备发送一个初始化命令序列。
- 初始化设备驱动程序的数据结构。

11.4.3 一个字符设备驱动程序的实例

下面我们通过一个实例对字符设备以及编写驱动程序的方法进行说明，通过下面的分析我们可以了解一个设备驱动程序的编写过程以及注意事项。虽然这个驱动程序没有什么实用价值,但是我们也可以通过它对一个驱动程序的编写特别是字符设备驱动程序有一定的认识。

一个设备驱动程序在结构上是非常相似的，在 Linux 中，驱动程序一般用 C 语言编写，有时也支持一些汇编和 C++语言。

1．头文件、宏定义和全局变量

一个典型的设备驱动程序一般都包含有一个专用头文件，这个头文件中包含一些系统函数的声明、设备寄存器的地址、寄存器状态位和控制位的定义以及用于此设备驱动程序的全局变量的定义，另外大多数驱动程序还使用以下一些标准的头文件。

param.h	包含一些内核参数
dir.h	包含一些目录参数
user.h	用户区域的定义
tty.h	终端和命令列表的定义
fs.h	其中包括 Buffer header 信息

下面是一些必要的头文件

```
#include <linux/kernel.h>
#include <linux/module.h>
#if CONFIG_MODVERSIONS==1  /* 处理 CONFIG_MODVERSIONS */
#define MODVERSIONS
#include <linux/modversions.h>
#endif
/* 下面是针对字符设备的头文件 */
#include <linux/fs.h>
#include <linux/wrapper.h>

/* 对于不同的版本我们需要做一些必要的事情*/
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,0)
#include <asm/uaccess.h>  /* for copy_to_user */
#endif

#define SUCCESS 0

/* 声明设备 */
/* 这是本设备的名字，它将会出现在 /proc/devices */
```

```
#define DEVICE_NAME "char_dev"

/* 定义此设备消息缓冲的最大长度 */
#define BUF_LEN  100

/* 为了防止不同的进程在同一个时间使用此设备，定义此静态变量跟踪其状态 */
static int Device_Open = 0；

/* 当提出请求的时候，设备将读写的内容放在下面的数组中 */
static char Message[BUF_LEN]；

/* 在进程读取这个内容的时候，这个指针是指向读取的位置*/
static char *Message_Ptr ；

/* 在这个文件中，主设备号作为全局变量以便于这个设备在注册和释放的时候使用*/
static int Major;
```

2．open ( ) 函数

功能：无论一个进程何时试图去打开这个设备都会调用这个函数。

```
static int device_open(struct inode *inode,
                struct file *file)
{
  static int counter = 0;

#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif

  printk("Device: %d.%d\n",
      inode->i_rdev >> 8, inode->i_rdev & 0xFF);

/* 这个设备是一个独占设备，为了避免同时有两个进程使用这一个设备我们需要采取一定的措施*/
  if (Device_Open)
    return -EBUSY;

  Device_Open++;

/* 下面是初始化消息，注意不要使读写内容的长度超出缓冲区的长度，特别是运行在内核模式时，否则如果出现缓冲上溢则可能导致系统的崩溃*/
  sprintf(Message,
 "If I told you once, I told you %d times - %s",
   counter++,
    "Hello, world\n");

  Message_Ptr = Message;

/*当这个文件被打开的时候，我们必须确认该模块还没有被移走并且增加此模块的用户数目（在移走一个模块的时候会根据这个数字去决定可否移去，如果不是 0 则表明还有进程正在使用这个模块，不能移走）*/
  MOD_INC_USE_COUNT;
```

```
  return SUCCESS;
}
```

3. release ( ) 函数

功能：当一个进程试图关闭这个设备特殊文件的时候调用这个函数。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static int device_release(struct inode *inode,
                struct file *file)
#else
static void device_release(struct inode *inode,
                 struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

  /* 为下一个使用这个设备的进程做准备*/
  Device_Open --;

/* 减少这个模块使用者的数目，否则一旦你打开这个模块以后，你永远都不能释放掉它*/
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
  return 0;
#endif
}
```

4. read ( ) 函数

功能：当一个进程已经打开此设备文件以后并且试图去读它的时候调用这个函数。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* 把读出的数据放到这个缓冲区*/
    size_t length,   /* 缓冲区的长度*/
    loff_t *offset)  /* 文件中的偏移 */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,       int length)
#endif
{
  /* 实际上读出的字节数 */
  int bytes_read = 0;
  /* 如果读到缓冲区的末尾，则返回 0 ，类似文件的结束*/
  if (*Message_Ptr == 0)
    return 0;
  /* 将数据放入缓冲区中*/
  while (length && *Message_Ptr)  {
 /* 由于缓冲区是在用户空间而不是内核空间，所以我们必须使用 copu_to_user()函数将内核空间中
```

```
的数据拷贝到用户空间*/
        copy_to_user(buffer++,*(Message_Ptr++), length--);
        bytes_read ++;
      }

    #ifdef DEBUG
       printk ("Read %d bytes, %d left\n",
         bytes_read, length);
    #endif

       /* Read 函数返回一个真正读出的字节数*/
      return bytes_read;
    }
```

5. write ( ) 函数

功能：当试图将数据写入这个设备文件的时侯，这个函数被调用。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static ssize_t device_write(struct file *file,
    const char *buffer,
    size_t length,
    loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)", file, buffer, length);
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
    copy_from_user(Message, buffer, length);

  Message_Ptr = Message;

  /* 返回写入的字节数 */
  return i;
}
```

6. 这个设备驱动程序提供给文件系统的接口

当一个进程试图对我们生成的设备进行操作的时候就利用下面这个结构，这个结构就是我们提供给操作系统的接口，它的指针保存在设备表中，在 init_module () 中被传递给操作系统。

```
struct file_operations Fops = {
  read:    device_read,
```

```
  write:    device_write,
  open:     device_open,
  release: device_release
  };
```

7．模块的初始化和模块的卸载

init_module 函数用来初始化这个模块——注册该字符设备。init_module （）函数调用 module_register_chrdev，把设备驱动程序添加到内核的字符设备驱动程序表中，它返回这个驱动程序所使用的主设备号。

```
int init_module()
{
  /* 试图注册设备*/
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);

  /* 失败的时候返回负值*/
  if (Major < 0) {
    printk ("%s device failed with %d\n",
         "Sorry, registering the character",
         Major);
    return Major;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success.",
          Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
          "and see what happens.\n");

  return 0;
}
```

以下这个函数的功能是卸载模块，主要是从 /proc 中 取消注册的设备特殊文件。

```
void cleanup_module()
{
  int ret;

  /* 取消注册的设备*/
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

  /* 如果出错则显示出错信息 */
  if (ret < 0)
    printk("Error in unregister_chrdev: %d\n", ret);
}
```

11.4.4　驱动程序的编译与装载

写完了设备驱动程序，下一项任务就是对驱动程序进行编译和装载。在 Linux 里，除了直接修改系统内核的源代码，把设备驱动程序加进内核外，还可以把设备驱动程序作为可加载的模块，由系统管理员动态地加载它，使之成为内核的一部分。也可以由系统管理员把已加载的模块动态地卸载下来。Linux 中，模块可以用 C 语言编写，用 gcc 编译成目标文件（不进行链接，作为*.o 文件存盘），为此需要在 gcc 命令行里加上-c 的参数。在编译时，还应该在 gcc 的命令行里加上这样的参数：-D__KERNEL__ -DMODULE。由于在不链接时，gcc 只允许一个输入文件，因此一个模块的所有部分都必须在一个文件里实现。

编译好的模块*.o 放在/lib/modules/xxxx/misc 下(xxxx 表示内核版本),然后用 depmod -a 使此模块成为可加载模块。模块用 insmod 命令加载,用 rmmod 命令来卸载,并可以用 lsmod 命令来查看所有已加载的模块的状态。

编写模块程序的时候，必须提供两个函数，一个是 int init_module(void)，供 insmod 在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。init_module 返回 0 以表示初始化成功，返回负数表示失败。另一个函数是 void cleanup_module (void)，在模块被卸载时调用，负责进行设备驱动程序的清除工作。

在成功地向系统注册了设备驱动程序后(调用 register_chrdev 成功后),就可以用 mknod 命令来把设备映射为一个特别文件，其他程序使用这个设备的时候，只要对此特别文件进行操作就行了。