







第6章 块设备驱动程序(block driver)

6.1 概述

本章描述内核的块设备驱动程序。在 Linux 0.11 内核中主要支持硬盘和软盘驱动器两种块设备。由于块设备主要与文件系统和高速缓冲有关，因此最好先快速浏览一下文件系统一章的内容。所涉及的源代码文件见列表 6-1 所示。

列表 6-1 linux/kernel/blk_drv 目录

文件名	大小	最后修改时间 (GMT)	说明
 Makefile	1951 bytes	1991-12-05 19:59:42	make 配置文件
 blk.h	3464 bytes	1991-12-05 19:58:01	块设备专用头文件
 floppy.c	11429 bytes	1991-12-07 00:00:38	软盘驱动程序
 hd.c	7807 bytes	1991-12-05 19:58:17	硬盘驱动程序
 ll_rw_blk.c	3539 bytes	1991-12-04 13:41:42	块设备接口程序
 ramdisk.c	2740 bytes	1991-12-06 03:08:06	虚拟盘驱动程序

本章程序代码的功能可分为两类，一类是对应各块设备的驱动程序，这类程序有：

1. 硬盘驱动程序 `hd.c`;
2. 软盘驱动程序 `floppy.c`;
3. 内存虚拟盘驱动程序 `ramdisk.c`;

另一类只有一个程序，是内核中其它程序访问块设备的接口程序 `ll_rw_blk.c`。块设备专用头文件 `blk.h` 为这三种块设备与 `ll_rw_blk.c` 程序交互提供了一个统一的设置方式和相同的设备请求开始程序。

6.2 总体功能

对硬盘和软盘块设备上数据的读写操作是通过中断处理程序进行的。每次读写的数据量以一个逻辑块（1024 字节）为单位，而块设备控制器则是以扇区（512 字节）为单位。在处理过程中，使用了读写请求项等待队列来顺序缓冲一次读写多个逻辑块的操作。

当程序需要读取硬盘上的一个逻辑块时，就会向缓冲区管理程序提出申请，而程序的进程则进入睡眠等待状态。缓冲区管理程序首先在缓冲区中寻找以前是否已经读取过这块数据。如果缓冲区中已经有了，就直接将对应的缓冲区块头指针返回给程序并唤醒该程序进程。若缓冲区中不存在所要求的数据块，则缓冲管理程序就会调用本章中的低级块读写函数 `ll_rw_block()`，向相应的块设备驱动程序发出一个读数据块的操作请求。该函数就会为此创建一个请求结构项，并插入请求队列中。为了提供读写磁盘的效率，减小磁头移动的距离，在插入请求项时使用了电梯移动算法。

当对应的块设备的请求项队列空时，表明此刻该块设备不忙。于是内核就会立刻向该块设备的控制器发出读数据命令。当块设备的控制器将数据读入到指定的缓冲块中后，就会发出中断请求信号，并调用相应的读命令后处理函数，处理继续读扇区操作或者结束本次请求项的过程。例如对相应块设备进行关闭操作和设置该缓冲块数据已经更新标志，最后唤醒等待该块数据的进程。

6.2.1 块设备请求项和请求队列

根据上面描述，我们知道低级读写函数 `ll_rw_block()` 是通过请求项来与各种块设备建立联系并发出读写请求。对于各种块设备，内核使用了一张块设备表 `blk_dev[]` 来进行管理。每种块设备都在块设备表中占有一项。块设备表中每个块设备项的结构为（摘自后面 `blk.h`）：

```
struct blk_dev_struct {
    void (*request_fn)(void);           // 请求项操作的函数指针。
    struct request * current_request;    // 当前请求项指针。
};
extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // 块设备表（数组）（NR_BLK_DEV = 7）。
```

其中，第一个字段是一个函数指针，用于操作相应块设备的请求项。例如，对于硬盘驱动程序，它是 `do_hd_request()`，而对于软盘设备，它就是 `do_floppy_request()`。第二个字段是当前请求项结构指针，用于指明本块设备目前正在处理的请求项，初始化时都被置成 `NULL`。

块设备表将在内核初始化时，在 `init/main.c` 程序调用各设备的初始化函数时被设置。为了便于扩展，Linus 把块设备表建成了一个以主设备号为索引的数组。在 Linux 0.11 中，主设备号有 7 种，见表 6-1 所示。其中，主设备号 1、2 和 3 分别对应块设备：虚拟盘、软盘和硬盘。在块设备数组中其它各项都被默认地置成 `NULL`。

表 6-1 Linux 0.11 内核中的主设备号

主设备号	类型	说明	请求项操作函数
0	无	无。	NULL
1	块/字符	ram,内存设备（虚拟盘等）。	do_rd_request()
2	块	fd,软驱设备。	do_fd_request()
3	块	hd,硬盘设备。	do_hd_request()
4	字符	ttyx 设备。	NULL
5	字符	tty 设备。	NULL
6	字符	lp 打印机设备。	NULL

当内核发出一个块设备读写或其它操作请求时，`ll_rw_block()` 函数即会根据其参数中指明的操作命令和数据缓冲块头中的设备号，利用对应的请求项操作函数 `do_XX_request()` 建立一个块设备请求项，并利用电梯算法插入到请求项队列中。请求项队列由请求项数组中的项构成，共有 32 项，每个请求项的数据结构如下所示：

```
struct request {
    int dev;           // 使用的设备号（若为-1，表示该项空闲）。
    int cmd;           // 命令(READ 或 WRITE)。
    int errors;        // 操作时产生的错误次数。
    unsigned long sector; // 起始扇区。（1 块=2 扇区）
    unsigned long nr_sectors; // 读/写扇区数。
```

```
char * buffer; // 数据缓冲区。
struct task_struct * waiting; // 任务等待操作执行完成的地方。
struct buffer_head * bh; // 缓冲区头指针(include/linux/fs.h, 68)。
struct request * next; // 指向下一请求项。
};
extern struct request request[NR_REQUEST]; // 请求队列数组 (NR_REQUEST = 32)。
```

每个块设备的当前请求指针与请求项数组中该设备的请求项链表共同构成了该设备的请求队列。项与项之间利用字段 `next` 指针形成链表。因此块设备项和相关的请求队列形成如下所示结构。请求项采用数组加链表结构的主要原因是为了满足两个目的：一是利用请求项的数组结构在搜索空闲请求块时可以进行循环操作，因此程序可以编制得很简洁；二是为满足电梯算法插入请求项操作，因此也需要采用链表结构。图 6-1 中示出了硬盘设备当前具有 4 个请求项，软盘设备具有 1 个请求项，而虚拟盘设备目前暂时没有读写请求项。

对于一个当前空闲的块设备，当 `ll_rw_block()` 函数为其建立第一个请求项时，会让该设备的当前请求项指针 `current_request` 直接指向刚建立的请求项，并且立刻调用对应设备的请求项操作函数开始执行块设备读写操作。当一个块设备已经有几个请求项组成的链表存在，`ll_rw_block()` 就会利用电梯算法，根据磁头移动距离最小原则，把新建的请求项插入到链表适当的位置处。

另外，为满足读操作的优先权，在为建立新的请求项而搜索请求项数组时，把建立写操作时的空闲项搜索范围限制在整个请求项数组的前 2/3 范围内，而剩下的 1/3 请求项专门给读操作建立请求项使用。

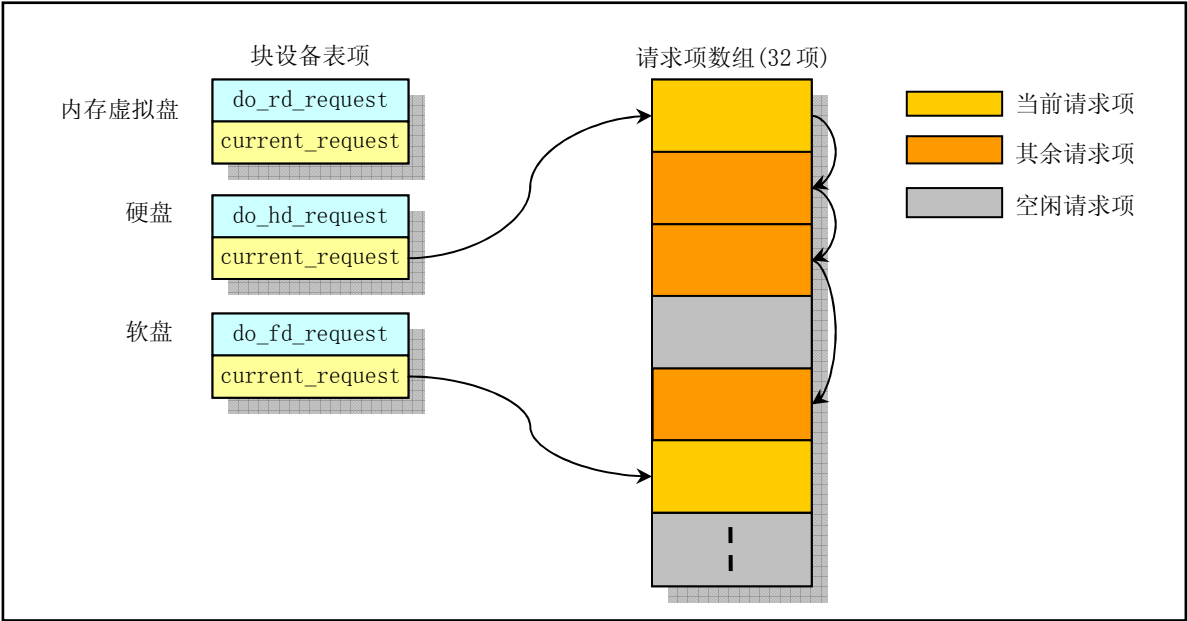


图 6-1 设备表项与请求项

6.2.2 块设备操作方式

在系统（内核）与硬盘进行 IO 操作时，需要考虑三个对象之间的交互作用。它们是系统、控制器和驱动器（例如硬盘或软盘驱动器），见图 6-2 所示。系统可以直接向控制器发送命令或等待控制器发出中断请求；控制器在接收到命令后就会控制驱动器的操作，读/写数据或者进行其它操作。因此我们可以把这里控制器发出的中断信号看作是这三者之间的同步操作信号，所经历的操作步骤为：

首先系统指明控制器在执行命令结束而引发的中断过程中应该调用的 C 函数，然后向块设备控制器发送读、写、复位或其它操作命令；

当控制器完成了指定的命令，会发出中断请求信号，引发系统执行块设备的中断处理过程，并在其中调用指定的 C 函数对读/写或其它命令进行命令结束后的处理工作。

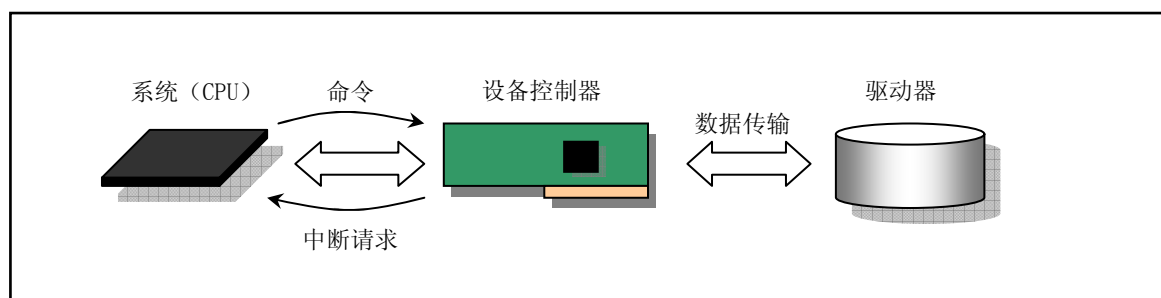


图 6-2 系统、块设备控制器和驱动器

对于写盘操作，系统需要在发出了写命令后（使用 `hd_out()`）等待控制器给予允许向控制器写数据的响应，也即需要查询等待控制器状态寄存器的数据请求服务标志 `DRQ` 置位。一旦 `DRQ` 置位，系统就可以向控制器缓冲区发送一个扇区的数据，同样也使用 `hd_out()` 函数。

当控制器把数据全部写入驱动器（后发生错误）以后，还会产生中断请求信号，从而在中断处理过程中执行前面预设置的 C 函数（`write_intr()`）。这个函数会查询是否还有数据要写。如果有，系统就再把一个扇区的数据传到控制器缓冲区中，然后再次等待控制器把数据写入驱动器后引发的中断，一直这样重复执行。如果此时所有数据都已经写入驱动器，则该 C 函数就执行本次写盘结束后的处理工作：唤醒等待该请求项有关数据的相关进程、唤醒等待请求项的进程、释放当前请求项并从链表中删除该请求项以及释放锁定的相关缓冲区。最后再调用请求项操作函数去执行下一个读/写盘请求项（若还有的话）。

对于读盘操作，系统在向控制器发送出包括需要读的扇区开始位置、扇区数量等信息的命令后，就等待控制器产生中断信号。当控制器按照读命令的要求，把指定的一扇区数据从驱动器传到了自己的缓冲区之后就会发出中断请求。从而会执行到前面为读盘操作预设置的 C 函数（`read_intr()`）。该函数首先把控制器缓冲区中一个扇区的数据放到系统的缓冲区中，调整系统缓冲区中当前写入位置，然后递减需读的扇区数量。若还有数据要读（递减结果值不为 0），则继续等待控制器发出下一个中断信号。若此时所有要求的扇区都已经读到系统缓冲区中，就执行与上面写盘操作一样的结束处理工作。

对于虚拟盘设备，由于它的读写操作不牵涉到与外部设备之间的同步操作，因此没有上述的中断处理过程。当前请求项对虚拟设备的读写操作完全在 `do_rd_request()` 中实现。

6.3 Makefile 文件

6.3.1 功能描述

该 `makefile` 文件用于管理对本目录下所有程序的编译。

6.3.2 代码注释

程序 6-1 `linux/kernel/blk_drv/Makefile`

```

1 #
2 # Makefile for the FREAX-kernel block device drivers.
3 #

```

```

4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX 内核块设备驱动程序的 Makefile 文件
9 # 注意！依赖关系是由 'make dep' 自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
10 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个 .c 文件的信息）。
11 # (Linux 最初的名字叫 FREAX，后来被 ftp.funet.fi 的管理员改成 Linux 这个名字)。
12
13 AR      =gar      # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
14 AS      =gas      # GNU 的汇编程序。
15 LD      =gld      # GNU 的连接程序。
16 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
17 CC      =gcc      # GNU C 语言编译器。
18 # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
19 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
20 # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
21 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linux 自己添加的优化选项，以后不再使用；
22 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../include)。
23 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
24         -finline-functions -mstring-insns -nostdinc -I../include
25 # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
26 # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
27 CPP     =gcc -E -nostdinc -I../include
28
29 # 下面的规则指示 make 利用下面的命令将所有的 .c 文件编译生成 .s 汇编程序。该规则的命令
30 # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止 (-S)，从而产生与
31 # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
32 # 去掉 .c 而加上 .s 后缀。-o 表示其后是输出文件的名称。其中 *.s (或 $@) 是自动目标变量，
33 # $< 代表第一个先决条件，这里即是符合条件 *.c 的文件。
34
35 .c.s:
36     $(CC) $(CFLAGS) \
37     -S -o *.s $<
38 # 下面规则表示将所有 .s 汇编程序文件编译成 .o 目标文件。22 行是实现该操作的具体命令。
39
40 .s.o:
41     $(AS) -c -o *.o $<
42
43 .c.o:
44     # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
45     $(CC) $(CFLAGS) \
46     -c -o *.o $<
47
48 OBJS = ll_rw_blk.o floppy.o hd.o ramdisk.o      # 定义目标文件变量 OBJS。
49
50 # 在有了先决条件 OBJS 后使用下面的命令连接成目标 blk_drv.a 库文件。
51 blk_drv.a: $(OBJS)
52     $(AR) rcs blk_drv.a $(OBJS)
53
54 sync
55
56 # 下面的规则用于清理工作。当执行 'make clean' 时，就会执行 34--35 行上的命令，去除所有编译
57 # 连接生成的文件。'rm' 是文件删除命令，选项 -f 含义是忽略不存在的文件，并且不显示删除信息。
58
59 clean:
60     rm -f core *.o *.a tmp_make
61     for i in *.c;do rm -f `basename $$i .c`.s;done
62
63

```


下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
 # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
 # 文件中'### Dependencies' 行后面的所有行（下面从 44 开始的行），并生成 tmp_make
 # 临时文件（38 行的作用）。然后对 kernel/blk_drv/目录下的每个 C 文件执行 gcc 预处理操作。
 # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
 # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
 # 文件名加上其依赖关系—该源文件中包含的所有头文件列表。把预处理结果都添加到临时
 # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。

```

37 dep:
38     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
39     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,`" "; \
40         $(CPP) -M $$i;done) >> tmp_make
41     cp tmp_make Makefile
42
43 ### Dependencies:
44 floppy.s floppy.o : floppy.c ../../include/linux/sched.h ../../include/linux/head.h \
45     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
46     ../../include/signal.h ../../include/linux/kernel.h \
47     ../../include/linux/fdreg.h ../../include/asm/system.h \
48     ../../include/asm/io.h ../../include/asm/segment.h blk.h
49 hd.s hd.o : hd.c ../../include/linux/config.h ../../include/linux/sched.h \
50     ../../include/linux/head.h ../../include/linux/fs.h \
51     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
52     ../../include/linux/kernel.h ../../include/linux/hdreg.h \
53     ../../include/asm/system.h ../../include/asm/io.h \
54     ../../include/asm/segment.h blk.h
55 ll_rw_blk.s ll_rw_blk.o : ll_rw_blk.c ../../include/errno.h ../../include/linux/sched.h \
56     ../../include/linux/head.h ../../include/linux/fs.h \
57     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
58     ../../include/linux/kernel.h ../../include/asm/system.h blk.h

```

6.4 blk.h 文件

6.4.1 功能描述

这是有关硬盘块设备参数的头文件，因为只用于块设备，所以与块设备代码放在同一个地方。其中主要定义了请求等待队列中项的数据结构 **request**，用宏语句定义了电梯搜索算法，并对内核目前支持的虚拟盘，硬盘和软盘三种块设备，根据它们各自的主设备号分别对应了常数值。

6.4.2 代码注释

程序 6-2 linux/kernel/blk_drv/blk.h

```

1 #ifndef BLK_H
2 #define BLK_H
3
4 #define NR_BLK_DEV      7          // 块设备的数量。
5 /*
6  * NR_REQUEST is the number of entries in the request-queue.

```

```

7  * NOTE that writes may use only the low 2/3 of these: reads
8  * take precedence.
9  *
10 * 32 seems to be a reasonable number: enough to get some benefit
11 * from the elevator-mechanism, but not so much as to lock a lot of
12 * buffers when they are in the queue. 64 seems to be too many (easily
13 * long pauses in reading when heavy writing/syncing is going on)
14 */
/*
* 下面定义的 NR_REQUEST 是请求队列中所包含的项数。
* 注意，读操作仅使用这些项低端的 2/3；读操作优先处理。
*
* 32 项好像是一个合理的数字：已经足够从电梯算法中获得好处，
* 但当缓冲区在队列中而锁住时又不显得是很大的数。64 就看上去
* 去太大了（当大量的写/同步操作运行时很容易引起长时间的暂停）。
*/
15 #define NR_REQUEST      32
16
17 /*
18 * Ok, this is an expanded form so that we can use the same
19 * request for paging requests when that is implemented. In
20 * paging, 'bh' is NULL, and 'waiting' is used to wait for
21 * read/write completion.
22 */
/*
* OK，下面是 request 结构的一个扩展形式，因而当实现以后，我们就可以在分页请求中
* 使用同样的 request 结构。在分页处理中，'bh' 是 NULL，而 'waiting' 则用于等待读/写的完成。
*/
// 下面是请求队列中项的结构。其中如果 dev=-1，则表示该项没有被使用。
23 struct request {
24     int dev;                /* -1 if no request */    // 使用的设备号。
25     int cmd;                /* READ or WRITE */      // 命令 (READ 或 WRITE)。
26     int errors;             // 操作时产生的错误次数。
27     unsigned long sector;   // 起始扇区。(1 块=2 扇区)
28     unsigned long nr_sectors; // 读/写扇区数。
29     char * buffer;          // 数据缓冲区。
30     struct task_struct * waiting; // 任务等待操作执行完成的地方。
31     struct buffer_head * bh;    // 缓冲区头指针(include/linux/fs.h, 68)。
32     struct request * next;     // 指向下一请求项。
33 };
34
35 /*
36 * This is used in the elevator algorithm: Note that
37 * reads always go before writes. This is natural: reads
38 * are much more time-critical than writes.
39 */
/*
* 下面的定义用于电梯算法：注意读操作总是在写操作之前进行。
* 这是很自然的：读操作对时间的要求要比写操作严格得多。
*/
40 #define IN_ORDER(s1,s2) \
41 ((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd && \
42 ((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \

```

```

43 (s1)->sector < (s2)->sector)))
44
45 // 块设备结构。
46 struct blk_dev_struct {
47     void (*request_fn)(void);          // 请求操作的函数指针。
48     struct request * current_request;   // 请求信息结构。
49 };
50 extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // 块设备表（数组），每种块设备占用一项。
51 extern struct request request[NR_REQUEST];        // 请求队列数组。
52 extern struct task_struct * wait_for_request;      // 等待空闲请求的任务结构队列头指针。
53
54 // 在块设备驱动程序（如 hd.c）要包含此头文件时，必须先定义驱动程序对应设备的主设备号。这样
55 // 下面 61 行—87 行就能为包含本文件的驱动程序给出正确的宏定义。
56 #ifdef MAJOR_NR // 主设备号。
57
58 /*
59  * Add entries as needed. Currently the only block devices
60  * supported are hard-disks and floppies.
61  */
62 /*
63  * 需要时加入条目。目前块设备仅支持硬盘和软盘（还有虚拟盘）。
64  */
65
66 #if (MAJOR_NR == 1) // RAM 盘的主设备号是 1。根据这里的定义可以推理内存块主设备号也为 1。
67 /* ram disk */ /* RAM 盘（内存虚拟盘）*/
68 #define DEVICE_NAME "ramdisk" // 设备名称 ramdisk。
69 #define DEVICE_REQUEST do_rd_request // 设备请求函数 do_rd_request()。
70 #define DEVICE_NR(device) ((device) & 7) // 设备号 (0--7)。
71 #define DEVICE_ON(device) // 开启设备。虚拟盘无须开启和关闭。
72 #define DEVICE_OFF(device) // 关闭设备。
73
74 #elif (MAJOR_NR == 2) // 软驱的主设备号是 2。
75 /* floppy */
76 #define DEVICE_NAME "floppy" // 设备名称 floppy。
77 #define DEVICE_INTR do_floppy // 设备中断处理程序 do_floppy()。
78 #define DEVICE_REQUEST do_fd_request // 设备请求函数 do_fd_request()。
79 #define DEVICE_NR(device) ((device) & 3) // 设备号 (0--3)。
80 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device)) // 开启设备函数 floppyon()。
81 #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device)) // 关闭设备函数 floppyoff()。
82
83 #elif (MAJOR_NR == 3) // 硬盘主设备号是 3。
84 /* harddisk */
85 #define DEVICE_NAME "harddisk" // 硬盘名称 harddisk。
86 #define DEVICE_INTR do_hd // 设备中断处理程序 do_hd()。
87 #define DEVICE_REQUEST do_hd_request // 设备请求函数 do_hd_request()。
88 #define DEVICE_NR(device) (MINOR(device)/5) // 设备号 (0--1)。每个硬盘可以有 4 个分区。
89 #define DEVICE_ON(device) // 硬盘一直在工作，无须开启和关闭。
90 #define DEVICE_OFF(device)
91
92 #elif
93 /* unknown blk device */ /* 未知块设备 */
94 #error "unknown blk device"

```



```

90
91 #endif
92
93 #define CURRENT (blk_dev[MAJOR_NR].current_request) // CURRENT 是指定主设备号的当前请求结构。
94 #define CURRENT_DEV DEVICE_NR(CURRENT->dev) // CURRENT_DEV 是 CURRENT 的设备号。
95
96 // 下面申明两个宏定义为函数指针。
97 #ifdef DEVICE_INTR
98 void (*DEVICE_INTR)(void) = NULL;
99 #endif
100 static void (DEVICE_REQUEST)(void);
101
102 // 释放锁定的缓冲区（块）。
103 extern inline void unlock_buffer(struct buffer_head * bh)
104 {
105     if (!bh->b_lock) // 如果指定的缓冲区 bh 并没有被上锁，则显示警告信息。
106         printk(DEVICE_NAME ": free buffer being unlocked\n");
107     bh->b_lock=0; // 否则将该缓冲区解锁。
108     wake_up(&bh->b_wait); // 唤醒等待该缓冲区的进程。
109 }
110
111 // 结束请求处理。
112 // 首先关闭指定块设备，然后检查此次读写缓冲区是否有效。如果有效则根据参数值设置缓冲区数据
113 // 更新标志，并解锁该缓冲区。如果更新标志参数值是 0，表示此次请求项的操作已失败，因此显示
114 // 相关块设备 IO 错误信息。最后，唤醒等待该请求项的进程以及等待空闲请求项出现的进程，释放
115 // 并从请求链表中删除本请求项。
116 extern inline void end_request(int uptodate)
117 {
118     DEVICE_OFF(CURRENT->dev); // 关闭设备。
119     if (CURRENT->bh) { // CURRENT 为指定主设备号的当前请求结构。
120         CURRENT->bh->b_uptodate = uptodate; // 置更新标志。
121         unlock_buffer(CURRENT->bh); // 解锁缓冲区。
122     }
123     if (!uptodate) { // 如果更新标志为 0 则显示设备错误信息。
124         printk(DEVICE_NAME " I/O error\n");
125         printk("dev %04x, block %d\n", CURRENT->dev,
126             CURRENT->bh->b_blocknr);
127     }
128     wake_up(&CURRENT->waiting); // 唤醒等待该请求项的进程。
129     wake_up(&wait_for_request); // 唤醒等待请求的进程。
130     CURRENT->dev = -1; // 释放该请求项。
131     CURRENT = CURRENT->next; // 从请求链表中删除该请求项，并且
132 } // 当前请求项指针指向下一个请求项。
133
134 // 定义初始化请求宏。
135 #define INIT_REQUEST \
136 repeat: \
137     if (!CURRENT) \
138         return; \
139     if (MAJOR(CURRENT->dev) != MAJOR_NR) \
140         panic(DEVICE_NAME ": request list destroyed"); \
141     if (CURRENT->bh) { \
142         if (!CURRENT->bh->b_lock) \
143             // 如果在进行请求操作时缓冲区没锁定则死机。

```

```

135         panic(DEVICE_NAME ": block not locked"); \
136     }
137
138 #endif
139
140 #endif
141

```

6.5 hd.c 程序

6.5.1 功能描述

hd.c 程序是硬盘控制器驱动程序，提供对硬盘控制器块设备的读写驱动和硬盘初始化处理。程序中所有函数按照功能不同可分为 5 类：

- 初始化硬盘和设置硬盘所用数据结构信息的函数，如 `sys_setup()`和 `hd_init()`；
- 向硬盘控制器发送命令的函数 `hd_out()`；
- 处理硬盘当前请求项的函数 `do_hd_request()`；
- 硬盘中断处理过程中调用的 C 函数，如 `read_intr()`、`write_intr()`、`bad_rw_intr()`和 `recal_intr()`。
`do_hd_request()`函数也将在 `read_intr()`和 `write_intr()`中被调用；
- 硬盘控制器操作辅助函数，如 `controler_ready()`、`drive_busy()`、`win_result()`、`hd_out()`和 `reset_controler()`等。

`sys_setup()`函数利用 `boot/setup.s` 程序提供的信息对系统中所含硬盘驱动器的参数进行了设置。然后读取硬盘分区表，并尝试把启动引导盘上的虚拟盘根文件系统映像文件复制到内存虚拟盘中，若成功则加载虚拟盘中的根文件系统，否则就继续执行普通根文件系统加载操作。

`hd_init()`函数用于在内核初始化时设置硬盘控制器中断描述符，并复位硬盘控制器中断屏蔽码，以允许硬盘控制器发送中断请求信号。

`hd_out()`是硬盘控制器操作命令发送函数。该函数带有一个中断过程中调用的 C 函数指针参数，在向控制器发送命令之前，它首先使用这个参数预置好中断过程中会调用的函数指针（`do_hd`），然后它按照规定的方式依次向硬盘控制器 `0x1f0` 至 `0x1f7` 发送命令参数块。除控制器诊断（`WIN_DIAGNOSE`）和建立驱动器参数（`WIN_SPECIFY`）两个命令以外，硬盘控制器在接收到任何其它命令并执行了命令以后，都会向 CPU 发出中断请求信号，从而引发系统去执行硬盘中断处理过程（在 `system_call.s`, 221 行）。

`do_hd_request()`是硬盘请求项的操作函数。其操作流程如下：

- ♦ 首先判断当前请求项是否存在，若当前请求项指针为空，则说明目前硬盘块设备已经没有待处理的请求项，因此立刻退出程序。这是在宏 `INIT_REQUEST` 中执行的语句。否则就继续处理当前请求项。
- ♦ 对当前请求项中指明的设备号和请求的盘起始扇区号的合理性进行验证；
- ♦ 根据当前请求项提供的信息计算请求数据的磁盘磁道号、磁头号和柱面号；
- ♦ 如果复位标志（`reset`）已被设置，则也设置硬盘重新校正标志（`recalibrate`），并对硬盘执行复位操作，向控制器重新发送“建立驱动器参数”命令（`WIN_SPECIFY`）。该命令不会引发硬盘中断；
- ♦ 如果重新校正标志被置位的话，就向控制器发送硬盘重新校正命令（`WIN_RESTORE`），并在发送之前预先设置好该命令引发的中断中需要执行的 C 函数（`recal_intr()`），并退出。`recal_intr()`函数的主要作用是：当控制器执行该命令结束并引发中断时，能重新（继续）执行本函数。
- ♦ 如果当前请求项指定是写操作，则首先设置硬盘控制器调用的 C 函数为 `write_intr()`，向控制器发

送写操作的命令参数块，并循环查询控制器的状态寄存器，以判断请求服务标志（DRQ）是否置位。若该标志置位，则表示控制器已“同意”接收数据，于是接着就把请求项所指缓冲区中的数据写入控制器的数据缓冲区中。若循环查询超时后该标志仍然没有置位，则说明此次操作失败。于是调用 `bad_rw_intr()` 函数，根据处理当前请求项发生的出错次数来确定是放弃继续当前请求项还是需要设置复位标志，以继续重新处理当前请求项。

- ♦ 如果当前请求项是读操作，则设置硬盘控制器调用的 C 函数为 `read_intr()`，并向控制器发送读盘操作命令。

`write_intr()` 是在当前请求项是写操作时被设置成中断过程调用的 C 函数。控制器完成写盘命令后会立刻向 CPU 发送中断请求信号，于是在控制器写操作完成后就会立刻调用该函数。

该函数首先调用 `win_result()` 函数，读取控制器的状态寄存器，以判断是否有错误发生。若在写盘操作时发生了错误，则调用 `bad_rw_intr()`，根据处理当前请求项发生的出错次数来确定是放弃继续当前请求项还是需要设置复位标志，以继续重新处理当前请求项。若没有发生错误，则根据当前请求项中指定的需写扇区总数，判断是否已经把此请求项要求的所有数据写盘了。若还有数据需要写盘，则再把一个扇区的数据复制到控制器缓冲区中。若数据已经全部写盘，则处理当前请求项的结束事宜：唤醒等待本请求项完成的进程、唤醒等待空闲请求项的进程（若有的话）、设置当前请求项所指缓冲区数据已更新标志、释放当前请求项（从块设备链表中删除该项）。最后继续调用 `do_hd_request()` 函数，以继续处理硬盘设备的其它请求项。

`read_intr()` 则是在当前请求项是读操作时被设置成中断过程中调用的 C 函数。控制器在把指定的扇区数据从硬盘驱动器读入自己的缓冲区后，就会立刻发送中断请求信号。而该函数的主要作用就是把控制器中的数据复制到当前请求项指定的缓冲区中。

与 `write_intr()` 开始的处理方式相同，该函数首先也调用 `win_result()` 函数，读取控制器的状态寄存器，以判断是否有错误发生。若在读盘时发生了错误，则执行与 `write_intr()` 同样的处理过程。若没有发生任何错误，则从控制器缓冲区把一个扇区的数据复制到请求项指定的缓冲区中。然后根据当前请求项中指定的欲读扇区总数，判断是否已经读取了所有的数据。若还有数据要读，则退出，以等待下一个中断的到来。若数据已经全部获得，则处理当前请求项的结束事宜：唤醒等待当前请求项完成的进程、唤醒等待空闲请求项的进程（若有的话）、设置当前请求项所指缓冲区数据已更新标志、释放当前请求项（从块设备链表中删除该项）。最后继续调用 `do_hd_request()` 函数，以继续处理硬盘设备的其它请求项。

为了能更清晰的看清楚硬盘读写操作的处理过程，我们可以把这些函数、中断处理过程以及硬盘控制器三者之间的执行时序关系用图 6-3 表示出来。

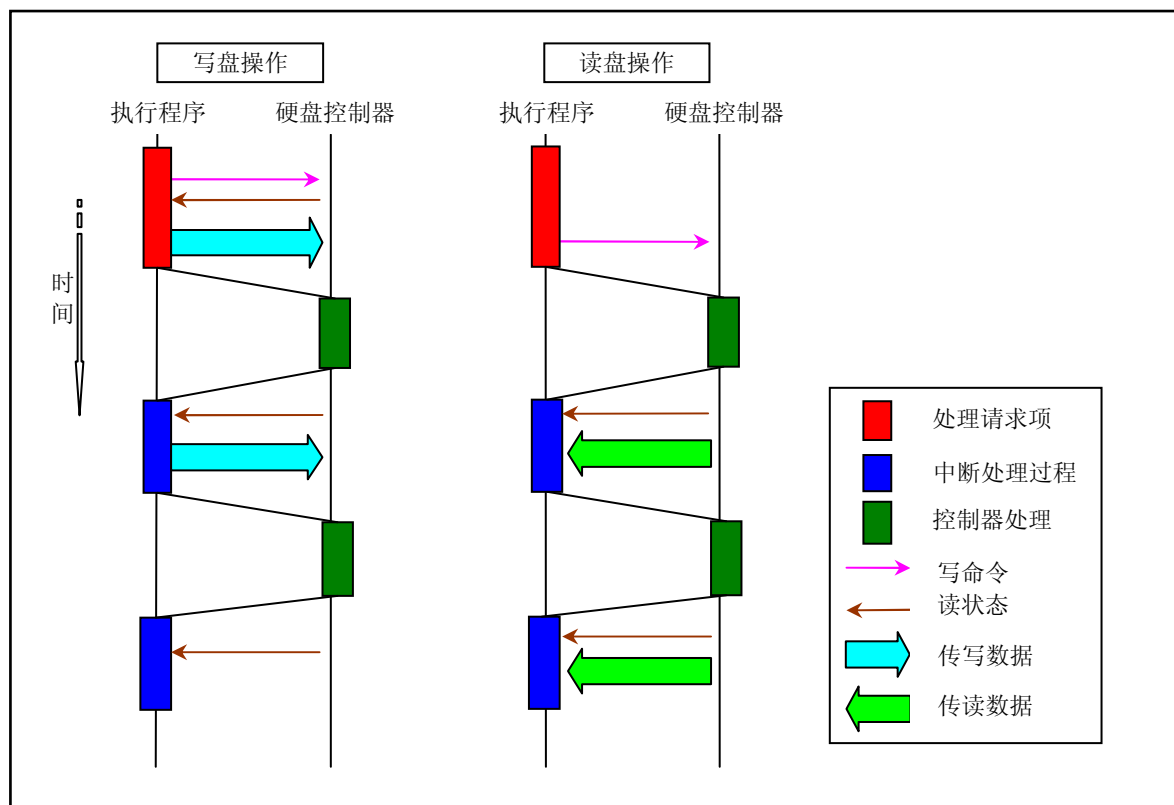


图 6-3 读/写硬盘数据的时序关系

由以上分析可以看出，本程序中最重要的是 4 个函数是 `hd_out()`、`do_hd_request()`、`read_intr()` 和 `write_intr()`。理解了这 4 个函数的作用也就理解了硬盘驱动程序的操作过程☺。

6.5.2 代码注释

程序 6-3 linux/kernel/blk_drv/hd.c

```

1  /*
2  *  linux/kernel/hd.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  This is the low-level hd interrupt support. It traverses the
9  *  request-list, using interrupts to jump between functions. As
10 *  all the functions are called within interrupts, we may not
11 *  sleep. Special care is recommended.
12 *
13 *  modified by Drew Eckhardt to check nr of hd's from the CMOS.
14 */
15
/*
* 本程序是底层硬盘中断辅助程序。主要用于扫描请求列表，使用中断在函数之间跳转。
* 由于所有的函数都是在中断里调用的，所以这些函数不可以睡眠。请特别注意。
* 由 Drew Eckhardt 修改，利用 CMOS 信息检测硬盘数。
*/

```

15

```

16 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型 (HD_TYPE) 可选项。
17 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
18 #include <linux/fs.h> // 文件系统头文件。定义文件表结构 (file, buffer_head, m_inode 等)。
19 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
20 #include <linux/hdreg.h> // 硬盘参数头文件。定义访问硬盘寄存器端口，状态码，分区表等信息。
21 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
22 #include <asm/io.h> // io 头文件。定义硬件端口输入/输出宏汇编语句。
23 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24
    // 必须在 blk.h 文件之前定义下面的主设备号常数，因为 blk.h 文件中要用到该常数。
25 #define MAJOR_NR 3 // 硬盘主设备号是 3。
26 #include "blk.h" // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。
27
28 #define CMOS_READ(addr) ({ \ // 读 CMOS 参数宏函数。
29 outb_p(0x80|addr, 0x70); \
30 inb_p(0x71); \
31 })
32
33 /* Max read/write errors/sector */
34 #define MAX_ERRORS 7 // 读/写一个扇区时允许的最多出错次数。
35 #define MAX_HD 2 // 系统支持的最多硬盘数。
36
37 static void recal_intr(void); // 硬盘中断程序在复位操作时会调用的重新校正函数 (287 行)。
38
39 static int recalibrate = 1; // 重新校正标志。将磁头移动到 0 柱面。
40 static int reset = 1; // 复位标志。当发生读写错误时会设置该标志，以复位硬盘和控制器。
41
42 /*
43 * This struct defines the HD's and their types.
44 */
    /* 下面结构定义了硬盘参数及类型 */
    // 各字段分别是磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆区柱面号、控制字节。
    // 它们的含义请参见程序列表后的说明。
45 struct hd_i_struct {
46     int head, sect, cyl, wpcom, lzone, ctl;
47 };

    // 如果已经在 include/linux/config.h 头文件中定义了 HD_TYPE，就取其中定义好的参数作为
    // hd_info[] 的数据。否则，先默认都设为 0 值，在 setup() 函数中会进行设置。
48 #ifdef HD_TYPE
49 struct hd_i_struct hd_info[] = { HD_TYPE };
50 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct))) // 计算硬盘个数。
51 #else
52 struct hd_i_struct hd_info[] = { {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0} };
53 static int NR_HD = 0;
54 #endif
55
    // 定义硬盘分区结构。给出每个分区的物理起始扇区号、分区扇区总数。
    // 其中 5 的倍数处的项 (例如 hd[0] 和 hd[5] 等) 代表整个硬盘中的参数。
56 static struct hd_struct {
57     long start_sect;
58     long nr_sects;

```

```

59 } hd[5*MAX_HD]={{0,0},{}};
60
// 读端口 port, 共读 nr 字, 保存在 buf 中。
61 #define port_read(port,buf,nr) \
62 __asm__ ("cld;rep;insw::\"d\" (port),\"D\" (buf),\"c\" (nr):\"cx\",\"di\")
63
// 写端口 port, 共写 nr 字, 从 buf 中取数据。
64 #define port_write(port,buf,nr) \
65 __asm__ ("cld;rep;outsw::\"d\" (port),\"S\" (buf),\"c\" (nr):\"cx\",\"si\")
66
67 extern void hd_interrupt(void); // 硬盘中断过程 (system_call.s, 221 行)。
68 extern void rd_load(void); // 虚拟盘创建加载函数 (ramdisk.c, 71 行)。
69
70 /* This may be used only once, enforced by 'static int callable' */
/* 下面该函数只在初始化时被调用一次。用静态变量 callable 作为可调用标志。*/
// 该函数的参数由初始化程序 init/main.c 的 init 子程序设置为指向 0x90080 处,此处存放着 setup.s
// 程序从 BIOS 取得的 2 个硬盘的基本参数表(32 字节)。硬盘参数表信息参见下面列表后的说明。
// 本函数主要功能是读取 CMOS 和硬盘参数表信息,用于设置硬盘分区结构 hd,并加载 RAM 虚拟盘和
// 根文件系统。
71 int sys_setup(void * BIOS)
72 {
73     static int callable = 1;
74     int i,drive;
75     unsigned char cmos_disks;
76     struct partition *p;
77     struct buffer_head *bh;
78
// 初始化时 callable=1,当运行该函数时将其设置为 0,使本函数只能执行一次。
79     if (!callable)
80         return -1;
81     callable = 0;
// 如果没有在 config.h 中定义硬盘参数,就从 0x90080 处读入。
82 #ifndef HD_TYPE
83     for (drive=0 ; drive<2 ; drive++) {
84         hd_info[drive].cyl = *(unsigned short *) BIOS; // 柱面数。
85         hd_info[drive].head = *(unsigned char *) (2+BIOS); // 磁头数。
86         hd_info[drive].wpcom = *(unsigned short *) (5+BIOS); // 写前预补偿柱面号。
87         hd_info[drive].ctl = *(unsigned char *) (8+BIOS); // 控制字节。
88         hd_info[drive].lzone = *(unsigned short *) (12+BIOS); // 磁头着陆区柱面号。
89         hd_info[drive].sect = *(unsigned char *) (14+BIOS); // 每磁道扇区数。
90         BIOS += 16; // 每个硬盘的参数表长 16 字节,这里 BIOS 指向下一个表。
91     }
// setup.s 程序在取 BIOS 中的硬盘参数表信息时,如果只有 1 个硬盘,就会将对应第 2 个硬盘的
// 16 字节全部清零。因此这里只要判断第 2 个硬盘柱面数是否为 0 就可以知道有没有第 2 个硬盘了。
92     if (hd_info[1].cyl)
93         NR_HD=2; // 硬盘数置为 2。
94     else
95         NR_HD=1;
96 #endif
// 设置每个硬盘的起始扇区号和扇区总数。其中编号 i*5 含义参见本程序后的有关说明。
97     for (i=0 ; i<NR_HD ; i++) {
98         hd[i*5].start_sect = 0; // 硬盘起始扇区号。
99         hd[i*5].nr_sects = hd_info[i].head*

```



```

100         hd_info[i].sect*hd_info[i].cyl; // 硬盘总扇区数。
101     }
102
103     /*
104         We query CMOS about hard disks : it could be that
105         we have a SCSI/ESDI/etc controller that is BIOS
106         compatable with ST-506, and thus showing up in our
107         BIOS table, but not register compatable, and therefore
108         not present in CMOS.
109
110         Furthurmore, we will assume that our ST-506 drives
111         <if any> are the primary drives in the system, and
112         the ones reflected as drive 1 or 2.
113
114         The first drive is stored in the high nibble of CMOS
115         byte 0x12, the second in the low nibble. This will be
116         either a 4 bit drive type or 0xf indicating use byte 0x19
117         for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
118
119         Needless to say, a non-zero value means we have
120         an AT controller hard disk for that drive.
121
122     */
123     */
124     /*
125     * 我们对 CMOS 有关硬盘的信息有些怀疑：可能会出现这样的情况，我们有一块 SCSI/ESDI/等的
126     * 控制器，它是以 ST-506 方式与 BIOS 兼容的，因而会出现在我们的 BIOS 参数表中，但却又不
127     * 是寄存器兼容的，因此这些参数在 CMOS 中又不存在。
128     * 另外，我们假设 ST-506 驱动器（如果有的话）是系统中的基本驱动器，也即以驱动器 1 或 2
129     * 出现的驱动器。
130     * 第 1 个驱动器参数存放在 CMOS 字节 0x12 的高半字节中，第 2 个存放在低半字节中。该 4 位字节
131     * 信息可以是驱动器类型，也可能仅是 0xf。0xf 表示使用 CMOS 中 0x19 字节作为驱动器 1 的 8 位
132     * 类型字节，使用 CMOS 中 0x1A 字节作为驱动器 2 的类型字节。
133     * 总之，一个非零值意味着我们有一个 AT 控制器硬盘兼容的驱动器。
134     */
135     // 这里根据上述原理来检测硬盘到底是否是 AT 控制器兼容的。有关 CMOS 信息请参见 4.2.3.1 节。
136     if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
137         if (cmos_disks & 0x0f)
138             NR_HD = 2;
139         else
140             NR_HD = 1;
141     else
142         NR_HD = 0;
143     // 若 NR_HD=0，则两个硬盘都不是 AT 控制器兼容的，硬盘数据结构清零。
144     // 若 NR_HD=1，则将第 2 个硬盘的参数清零。
145     for (i = NR_HD ; i < 2 ; i++) {
146         hd[i*5].start_sect = 0;
147         hd[i*5].nr_sects = 0;
148     }
149     // 读取每一个硬盘上第 1 块数据（第 1 个扇区有用），获取其中的分区表信息。
150     // 首先利用函数 bread() 读硬盘第 1 块数据(fs/buffer.c, 267)，参数中的 0x300 是硬盘的主设备号
151     // （参见列表后的说明）。然后根据硬盘头 1 个扇区位置 0x1fe 处的两个字节是否为'55AA'来判断

```

```

// 该扇区中位于 0x1BE 开始的分区表是否有效。最后将分区表信息放入硬盘分区数据结构 hd 中。
136     for (drive=0 ; drive<NR_HD ; drive++) {
137         if (!(bh = bread(0x300 + drive*5,0))) { // 0x300, 0x305 逻辑设备号。
138             printk("Unable to read partition table of drive %d\n\r",
139                 drive);
140             panic("");
141         }
142         if (bh->b_data[510] != 0x55 || (unsigned char)
143             bh->b_data[511] != 0xAA) { // 判断硬盘信息有效标志'55AA'。
144             printk("Bad partition table on drive %d\n\r",drive);
145             panic("");
146         }
147         p = 0x1BE + (void *)bh->b_data; // 分区表位于硬盘第 1 扇区的 0x1BE 处。
148         for (i=1;i<5;i++,p++) {
149             hd[i+5*drive].start_sect = p->start_sect;
150             hd[i+5*drive].nr_sects = p->nr_sects;
151         }
152         brelse(bh); // 释放为存放硬盘块而申请的内存缓冲区页。
153     }
154     if (NR_HD) // 如果有硬盘存在并且已读入分区表，则打印分区表正常信息。
155         printk("Partition table%s ok. \n\r", (NR_HD>1)? "s": "");
156     rd_load(); // 加载（创建）RAMDISK(kernel/blk_drv/ramdisk.c, 71)。
157     mount_root(); // 安装根文件系统(fs/super.c, 242)。
158     return (0);
159 }
160
//// 判断并循环等待驱动器就绪。
// 读硬盘控制器状态寄存器端口 HD_STATUS(0x1f7)，并循环检测驱动器就绪比特位和控制器忙位。
// 如果返回值为 0，则表示等待超时出错，否则 OK。
161 static int controller_ready(void)
162 {
163     int retries=10000;
164
165     while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
166     return (retries); // 返回等待循环的次数。
167 }
168
//// 检测硬盘执行命令后的状态。(win_表示温切斯特硬盘的缩写)
// 读取状态寄存器中的命令执行结果状态。返回 0 表示正常，1 出错。如果执行命令错，
// 则再读错误寄存器 HD_ERROR(0x1f1)。
169 static int win_result(void)
170 {
171     int i=inb_p(HD_STATUS); // 取状态信息。
172
173     if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
174         == (READY_STAT | SEEK_STAT))
175         return(0); /* ok */
176     if (i&1) i=inb(HD_ERROR); // 若 ERR_STAT 置位，则读取错误寄存器。
177     return (1);
178 }
179
//// 向硬盘控制器发送命令块（参见列表后的说明）。
// 调用参数：drive - 硬盘号(0-1)； nsect - 读写扇区数；

```

```

//      sect - 起始扇区;      head - 磁头号;
//      cyl  - 柱面号;      cmd  - 命令码 (参见控制器命令列表, 表 6.3);
//      *intr_addr() - 硬盘中断发生时处理程序中将调用的 C 处理函数。
180 static void hd\_out(unsigned int drive,unsigned int nsect,unsigned int sect,
181                 unsigned int head,unsigned int cyl,unsigned int cmd,
182                 void (*intr_addr)(void))
183 {
184     register int port asm("dx");    // port 变量对应寄存器 dx。
185
186     if (drive>1 || head>15)          // 如果驱动器号(0,1)>1 或磁头号>15, 则程序不支持。
187         panic("Trying to write bad sector");
188     if (!controller\_ready())        // 如果等待一段时间后仍未就绪则出错, 死机。
189         panic("HD controller not ready");
190     do_hd = intr_addr;              // do_hd 函数指针将在硬盘中断程序中被调用。
191     outb\_p(hd\_info[drive].ctl,HD\_CMD);    // 向控制寄存器(0x3f6)输出控制字节。
192     port=HD\_DATA;                  // 置 dx 为数据寄存器端口(0x1f0)。
193     outb\_p(hd\_info[drive].wpcom>>2,++port); // 参数: 写预补偿柱面号(需除 4)。
194     outb\_p(nsect,++port);            // 参数: 读/写扇区总数。
195     outb\_p(sect,++port);              // 参数: 起始扇区。
196     outb\_p(cyl,++port);               // 参数: 柱面号低 8 位。
197     outb\_p(cyl>>8,++port);           // 参数: 柱面号高 8 位。
198     outb\_p(0xA0|(drive<<4)|head,++port); // 参数: 驱动器号+磁头号。
199     outb(cmd,++port);                // 命令: 硬盘控制命令。
200 }
201
202 // 等待硬盘就绪。也即循环等待主状态控制器忙标志位复位。若仅有就绪或寻道结束标志
203 // 置位, 则成功, 返回 0。若经过一段时间仍为忙, 则返回 1。
204 static int drive\_busy(void)
205 {
206     unsigned int i;
207
208     for (i = 0; i < 10000; i++)      // 循环等待就绪标志位置位。
209         if (READY\_STAT == (inb\_p(HD\_STATUS) & (BUSY\_STAT | READY\_STAT)))
210             break;
211     i = inb(HD\_STATUS);              // 再取主控制器状态字节。
212     i &= BUSY\_STAT | READY\_STAT | SEEK\_STAT; // 检测忙位、就绪位和寻道结束位。
213     if (i == READY\_STAT | SEEK\_STAT)      // 若仅有就绪或寻道结束标志, 则返回 0。
214         return(0);
215     printk("HD controller times out\n\r"); // 否则等待超时, 显示信息。并返回 1。
216     return(1);
217 }
218
219 // 诊断复位 (重新校正) 硬盘控制器。
220 static void reset\_controller(void)
221 {
222     int i;
223
224     outb(4,HD\_CMD);                  // 向控制寄存器端口发送控制字节(4-复位)。
225     for(i = 0; i < 100; i++) nop();    // 等待一段时间 (循环空操作)。
226     outb(hd\_info[0].ctl & 0x0f ,HD\_CMD); // 再发送正常的控制字节(不禁止重试、重读)。
227     if (drive\_busy())                // 若等待硬盘就绪超时, 则显示出错信息。
228         printk("HD-controller still busy\n\r");
229     if ((i = inb(HD\_ERROR)) != 1)    // 取错误寄存器, 若不等于 1 (无错误) 则出错。

```

```

227         printk("HD-controller reset failed: %02x\n\r", i);
228     }
229
230     /// 复位硬盘 nr。首先复位（重新校正）硬盘控制器。然后发送硬盘控制器命令“建立驱动器参数”，
231     /// 其中 recal_intr() 是在硬盘中断处理程序中调用的重新校正处理函数。
232     static void reset_hd(int nr)
233     {
234         reset_controller();
235         hd_out(nr, hd_info[nr].sect, hd_info[nr].sect, hd_info[nr].head-1,
236             hd_info[nr].cyl, WIN_SPECIFY, &recal_intr);
237     }
238
239     /// 意外硬盘中断调用函数。
240     /// 发生意外硬盘中断时，硬盘中断处理程序中调用的默认 C 处理函数。在被调用函数指针为空时
241     /// 调用该函数。参见(kernel/system_call.s, 241 行)。
242     void unexpected_hd_interrupt(void)
243     {
244         printk("Unexpected HD interrupt\n\r");
245     }
246
247     /// 读写硬盘失败处理调用函数。
248     static void bad_rw_intr(void)
249     {
250         if (++CURRENT->errors >= MAX_ERRORS) // 如果读扇区时的出错次数大于或等于 7 次时，
251             end_request(0); // 则结束请求并唤醒等待该请求的进程，而且
252                             // 对应缓冲区更新标志复位（没有更新）。
253         if (CURRENT->errors > MAX_ERRORS/2) // 如果读一扇区时的出错次数已经大于 3 次，
254             reset = 1; // 则要求执行复位硬盘控制器操作。
255     }
256
257     /// 读操作中断调用函数。将在硬盘读命令结束时引发的中断过程中被调用。
258     /// 该函数首先判断此次读命令操作是否出错。若命令结束后控制器还处于忙状态，或者命令执行错误，
259     /// 则处理硬盘操作失败问题，接着请求硬盘作复位处理并执行其它请求项。
260     /// 如果读命令没有出错，则从数据寄存器端口把一个扇区的数据读到请求项的缓冲区中，并递减请求项
261     /// 所需读取的扇区数值。若递减后不等于 0，表示本项请求还有数据没取完，于是直接返回，等待硬盘
262     /// 在读出另一个扇区数据后的中断。否则表明本请求项所需的所有扇区都已读完，于是处理本次请求项
263     /// 结束事宜。最后再次调用 do_hd_request()，去处理其它硬盘请求项。
264     /// 注意：257 行语句中的 256 是指内存字，也即 512 字节。
265     static void read_intr(void)
266     {
267         if (win_result()) { // 若控制器忙、读写错或命令执行错，
268             bad_rw_intr(); // 则进行读写硬盘失败处理
269             do_hd_request(); // 然后再次请求硬盘作相应(复位)处理。
270             return;
271         }
272         port_read(HD_DATA, CURRENT->buffer, 256); // 将数据从数据寄存器口读到请求结构缓冲区。
273         CURRENT->errors = 0; // 清出错次数。
274         CURRENT->buffer += 512; // 调整缓冲区指针，指向新的空区。
275         CURRENT->sector++; // 起始扇区号加 1，
276         if (--CURRENT->nr_sectors) { // 如果所需读出的扇区数还没有读完，则
277             do_hd = &read_intr; // 再次置硬盘调用 C 函数指针为 read_intr()
278             return; // 因为硬盘中断处理程序每次调用 do_hd 时
279         } // 都会将该函数指针置空。参见 system_call.s

```

```

265     end_request(1);           // 若全部扇区数据已经读完，则处理请求结束事宜，
266     do_hd_request();         // 执行其它硬盘请求操作。
267 }
268
269 // 写扇区中断调用函数。在硬盘中断处理程序中被调用。
270 // 在写命令执行后，会产生硬盘中断信号，执行硬盘中断处理程序，此时在硬盘中断处理程序中调用的
271 // C 函数指针 do_hd() 已经指向 write_intr()，因此会在写操作完成（或出错）后，执行该函数。
272 static void write_intr(void)
273 {
274     if (win_result()) {       // 如果硬盘控制器返回错误信息，
275         bad_rw_intr();       // 则首先进行硬盘读写失败处理，
276         do_hd_request();      // 然后再次请求硬盘作相应(复位)处理，
277         return;              // 然后返回（也退出了此次硬盘中断）。
278     }
279     if (--CURRENT->nr_sectors) { // 否则将欲写扇区数减 1，若还有扇区要写，则
280         CURRENT->sector++;    // 当前请求起始扇区号+1，
281         CURRENT->buffer += 512; // 调整请求缓冲区指针，
282         do_hd = &write_intr;  // 置硬盘中断程序调用函数指针为 write_intr()，
283         port_write(HD_DATA, CURRENT->buffer, 256); // 再向数据寄存器端口写 256 字。
284         return;              // 返回等待硬盘再次完成写操作后的中断处理。
285     }
286     end_request(1);           // 若全部扇区数据已经写完，则处理请求结束事宜，
287     do_hd_request();         // 执行其它硬盘请求操作。
288 }
289
290 // 硬盘重新校正（复位）中断调用函数。在硬盘中断处理程序中被调用。
291 // 如果硬盘控制器返回错误信息，则首先进行硬盘读写失败处理，然后请求硬盘作相应(复位)处理。
292 static void recal_intr(void)
293 {
294     if (win_result())
295         bad_rw_intr();
296     do_hd_request();
297 }
298
299 // 执行硬盘读写请求操作。
300 // 若请求项是块设备的第 1 个，则块设备当前请求项指针（参见 ll_rw_blk.c，28 行）会直接指向该
301 // 请求项，并会立刻调用本函数执行读写操作。否则在一个读写操作完成而引发的硬盘中断过程中，
302 // 若还有请求项需要处理，则也会在中断过程中调用本函数。参见 kernel/system_call.s，221 行。
303 void do_hd_request(void)
304 {
305     int i, r;
306     unsigned int block, dev;
307     unsigned int sec, head, cyl;
308     unsigned int nsect;
309
310     // 检测请求项的合法性，若已没有请求项则退出(参见 blk.h, 127)。
311     INIT_REQUEST;
312     // 取设备号中的子设备号(见列表后对硬盘设备号的说明)。子设备号即是硬盘上的分区号。
313     dev = MINOR(CURRENT->dev); // CURRENT 定义为 blk_dev[MAJOR_NR].current_request。
314     block = CURRENT->sector;    // 请求的起始扇区。
315     // 如果子设备号不存在或者起始扇区大于该分区扇区数-2，则结束该请求，并跳转到标号 repeat 处
316     // （定义在 INIT_REQUEST 开始处）。因为一次要求读写 2 个扇区（512*2 字节），所以请求的扇区号
317     // 不能大于分区中最后倒数第二个扇区号。

```

```

304     if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
305         end_request(0);
306         goto repeat;        // 该标号在 blk.h 最后面。
307     }
// 通过加上本分区的起始扇区号, 把将所需读写的块对应到整个硬盘的绝对扇区号上。
308     block += hd[dev].start_sect;
309     dev /= 5;                // 此时 dev 代表硬盘号 (是第 1 个硬盘(0) 还是第 2 个(1))。
// 下面嵌入汇编代码用来从硬盘信息结构中根据起始扇区号和每磁道扇区数计算在磁道中的
// 扇区号(sec)、所在柱面号(cyl)和磁头号(head)。
310     __asm__ ("divl %4": "=a" (block), "=d" (sec): "" (block), "1" (0),
311             "r" (hd_info[dev].sect));
312     __asm__ ("divl %4": "=a" (cyl), "=d" (head): "" (block), "1" (0),
313             "r" (hd_info[dev].head));
314     sec++;
315     nsect = CURRENT->nr_sectors; // 欲读/写的扇区数。
// 如果 reset 标志是置位的, 则执行复位操作。复位硬盘和控制器, 并置需要重新校正标志, 返回。
316     if (reset) {
317         reset = 0;
318         recalibrate = 1;
319         reset_hd(CURRENT_DEV);
320         return;
321     }
// 如果重新校正标志(recalibrate)置位, 则首先复位该标志, 然后向硬盘控制器发送重新校正命令。
// 该命令会执行寻道操作, 让处于任何地方的磁头移动到 0 柱面。
322     if (recalibrate) {
323         recalibrate = 0;
324         hd_out(dev, hd_info[CURRENT_DEV].sect, 0, 0, 0,
325              WIN_RESTORE, &recal_intr);
326         return;
327     }
// 如果当前请求是写扇区操作, 则发送写命令, 循环读取状态寄存器信息并判断请求服务标志
// DRQ_STAT 是否置位。DRQ_STAT 是硬盘状态寄存器的请求服务位, 表示驱动器已经准备好在主机和
// 数据端口之间传输一个字或一个字节的数据。
328     if (CURRENT->cmd == WRITE) {
329         hd_out(dev, nsect, sec, head, cyl, WIN_WRITE, &write_intr);
330         for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
331             /* nothing */;
// 如果请求服务 DRQ 置位则退出循环。若等到循环结束也没有置位, 则表示此次写硬盘操作失败, 去
// 处理下一个硬盘请求。否则向硬盘控制器数据寄存器端口 HD_DATA 写入 1 个扇区的数据。
332         if (!r) {
333             bad_rw_intr();
334             goto repeat;        // 该标号在 blk.h 文件最后面, 也即跳到 301 行。
335         }
336         port_write(HD_DATA, CURRENT->buffer, 256);
// 如果当前请求是读硬盘扇区, 则向硬盘控制器发送读扇区命令。
337     } else if (CURRENT->cmd == READ) {
338         hd_out(dev, nsect, sec, head, cyl, WIN_READ, &read_intr);
339     } else
340         panic("unknown hd-command");
341 }
342
// 硬盘系统初始化。
343 void hd_init(void)

```



```

344 {
345     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // do_hd_request()。
346     set_intr_gate(0x2E, &hd_interrupt); // 设置硬盘中断门向量 int 0x2E(46)。
                                     // hd_interrupt 在(kernel/system_call.s, 221)。
347     outb_p(inb_p(0x21)&0xfb, 0x21); // 复位接联的主 8259A int2 的屏蔽位，允许从片
                                     // 发出中断请求信号。
348     outb(inb_p(0xA1)&0xbf, 0xA1); // 复位硬盘的中断请求屏蔽位（在从片上），允许
                                     // 硬盘控制器发送中断请求信号。
349 }
350

```

6.5.3 其它信息

6.5.3.1 AT 硬盘接口寄存器

AT 硬盘控制器的编程寄存器端口说明见表 6-2 所示。另外请参见 include/linux/hdreg.h 头文件。

表 6-2 AT 硬盘控制器寄存器端口及作用

端口	名称	读操作	写操作
0x1f0	HD_DATA	数据寄存器 -- 扇区数据（读、写、格式化）	
0x1f1	HD_ERROR	错误寄存器（错误状态）	写前预补偿寄存器
0x1f2	HD_NSECTOR	扇区数寄存器 -- 扇区数（读、写、检验、格式化）	
0x1f3	HD_SECTOR	扇区号寄存器 -- 起始扇区（读、写、检验）	
0x1f4	HD_LCYL	柱面号寄存器 -- 柱面号低字节（读、写、检验、格式化）	
0x1f5	HD_HCYL	柱面号寄存器 -- 柱面号高字节（读、写、检验、格式化）	
0x1f6	HD_CURRENT	驱动器/磁头寄存器 -- 驱动器号/磁头号（101dhhhh, d=驱动器号, h=磁头号）	
0x1f7	HD_STATUS	主状态寄存器 (HD_STATUS)	命令寄存器 (HD_COMMAND)
0x3f6	HD_CMD	---	硬盘控制寄存器 (HD_CMD)
0x3f7		数字输入寄存器（与 1.2M 软盘合用）	---

下面对各端口寄存器进行详细说明。

◆数据寄存器（HD_DATA, 0x1f0）

这是一对 16 位高速 PIO 数据传输器，用于扇区读、写和磁道格式化操作。CPU 通过该数据寄存器向硬盘写入或从硬盘读出 1 个扇区的数据，也即要使用命令'rep outsw'或'rep insw'重复读/写 cx=256 字。

◆错误寄存器（读）/写前预补偿寄存器（写）（HD_ERROR, 0x1f1）

在读时，该寄存器存放有 8 位的错误状态。但只有当主状态寄存器(HD_STATUS, 0x1f7)的位 0=1 时该寄存器中的数据才有效。执行控制器诊断命令时的含义与其它命令时的不同。见表 6-3 所示。

表 6-3 硬盘控制器错误寄存器

值	诊断命令时	其它命令时
0x01	无错误	数据标志丢失
0x02	控制器出错	磁道 0 错
0x03	扇区缓冲区错	
0x04	ECC 部件错	命令放弃
0x05	控制处理器错	
0x10		ID 未找到

0x40		ECC 错误
0x80		坏扇区

在写操作时，该寄存器即作为写前预补偿寄存器。它记录写预补偿起始柱面号。对应于与硬盘基本参数表位移 0x05 处的一个字，需除 4 后输出。

◆扇区数寄存器 (HD_NSECTOR, 0x1f2)

该寄存器存放读、写、检验和格式化命令指定的扇区数。当用于多扇区操作时，每完成 1 扇区的操作该寄存器就自动减 1，直到为 0。若初值为 0，则表示传输最大扇区数 256。

◆扇区号寄存器 (HD_SECTOR, 0x1f3)

该寄存器存放读、写、检验操作命令指定的扇区号。在多扇区操作时，保存的是起始扇区号，而每完成 1 扇区的操作就自动增 1。

◆柱面号寄存器 (HD_LCYL, HD_HCYL, 0x1f4, 0x1f5)

该两个柱面号寄存器分别存放有柱面号的低 8 位和高 2 位。

◆驱动器/磁头寄存器 (HD_CURRENT, 0x1f6)

该寄存器存放有读、写、检验、寻道和格式化命令指定的驱动器和磁头号。其位格式为 101dhhhh。其中 101 表示采用 ECC 校验码和每扇区为 512 字节；d 表示选择的驱动器 (0 或 1)；hhhh 表示选择的磁头。

◆主状态寄存器 (读) / 命令寄存器 (写) (HD_STATUS/HD_COMMAND, 0x1f7)

在读时，对应一个 8 位主状态寄存器。反映硬盘控制器在执行命令前后的操作状态。各位的含义见表 6-4 所示。

表 6-4 8 位主状态寄存器

位	名称	屏蔽码	说明
0	ERR_STAT	0x01	命令执行错误。当该位置位时说明前一个命令以出错结束。此时出错寄存器和状态寄存器中的比特位含有引起错误的一些信息。
1	INDEX_STAT	0x02	收到索引。当磁盘旋转遇到索引标志时会设置该位。
2	ECC_STAT	0x04	ECC 校验错。当遇到一个可恢复的数据错误而且已得到纠正，就会设置该位。这种情况不会中断一个多扇区读操作。
3	DRQ_STAT	0x08	数据请求服务。当该位被置位时，表示驱动器已经准备好在主机和数据端口之间传输一个字或一个字节的数据。
4	SEEK_STAT	0x10	驱动器寻道结束。当该位被置位时，表示寻道操作已经完成，磁头已经停在指定的磁道上。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前寻道的完成状态。
5	WRERR_STAT	0x20	驱动器故障 (写出错)。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前写操作的出错状态。
6	READY_STAT	0x40	驱动器准备好 (就绪)。表示驱动器已经准备好接收命令。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前驱动器就绪状态。在开机时，应该复位该比特位，直到驱动器速度达到正常并且能够接收命令。
7	BUSY_STAT	0x80	控制器忙碌。当驱动器正在操作由驱动器的控制器设置该位。此时主机不能发送命令块。而对任何命令寄存器的读操作将返回状态寄存器的值。在下列条件下该位会被置位： 在机器复位信号 RESET 变负或者设备控制寄存器的 SRST 被设置之后 400 纳秒以内。在机器复位之后要求该位置位状态不能超过 30 秒。

		主机在向命令寄存器写重新校正、读、读缓冲、初始化驱动器参数以及执行诊断等命令的 400 纳秒以内。 在写操作、写缓冲或格式化磁道命令期间传输了 512 字节数据的 5 微秒之内。
--	--	--

当执行写操作时，该端口对应命令寄存器，接受 CPU 发出的硬盘控制命令，共有 8 种命令，见表 6-5 所示。其中最后一列用于说明相应命令结束后控制器所采取的动作（引发中断或者什么也不做）。

表 6-5 AT 硬盘控制器命令列表

命令名称		命令码字节		默认值	命令执行结束形式
		高 4 位	D3 D2 D1 D0		
WIN_RESTORE	驱动器重新校正(复位)	0x1	R R R R	0x10	中断
WIN_READ	读扇区	0x2	0 0 L T	0x20	中断
WIN_WRITE	写扇区	0x3	0 0 L T	0x30	中断
WIN_VERIFY	扇区检验	0x4	0 0 0 T	0x40	中断
WIN_FORMAT	格式化磁道	0x5	0 0 0 0	0x50	中断
WIN_INIT	控制器初始化	0x6	0 0 0 0	0x60	中断
WIN_SEEK	寻道	0x7	R R R R	0x70	中断
WIN_DIAGNOSE	控制器诊断	0x9	0 0 0 0	0x90	控制器空闲
WIN_SPECIFY	建立驱动器参数	0x9	0 0 0 1	0x91	控制器空闲

表中命令码字节的低 4 位是附加参数，其含义为：

R 是步进速率。R=0，则步进速率为 35us；R=1 为 0.5ms，以此量递增。程序中默认 R=0。

L 是数据模式。L=0 表示读/写扇区为 512 字节；L=1 表示读/写扇区为 512 加 4 字节的 ECC 码。程序中默认值是 L=0。

T 是重试模式。T=0 表示允许重试；T=1 则禁止重试。程序中取 T=0。

◆硬盘控制寄存器（写）（HD_CMD，0x3f6）

该寄存器是只写的。用于存放硬盘控制字节并控制复位操作。其定义与硬盘基本参数表的位移 0x08 处的字节说明相同，见表 6-6 所示。

表 6-6 硬盘控制字节的含义

位移	大小	说明
0x08	字节	控制字节（驱动器步进选择）
		位 0 未用
		位 1 保留(0) (关闭 IRQ)
		位 2 允许复位
		位 3 若磁头数大于 8 则置 1
		位 4 未用(0)
		位 5 若在柱面数+1 处有生产商的坏区图，则置 1
		位 6 禁止 ECC 重试
		位 7 禁止访问重试。

6.5.3.2 AT 硬盘控制器编程

在对硬盘控制器进行操作控制时，需要同时发送参数和命令。其命令格式见表 6-7 所示。首先发送

6 字节的参数，最后发出 1 字节的命令码。不管什么命令均需要完整输出这 7 字节的命令块，依次写入端口 0x1f1 -- 0x1f7。。

表 6 - 7 命令格式

端口	说明
0x1f1	写预补偿起始柱面号
0x1f2	扇区数
0x1f3	起始扇区号
0x1f4	柱面号低字节
0x1f5	柱面号高字节
0x1f6	驱动器号/磁头号
0x1f7	命令码

首先 CPU 向控制寄存器端口(HD_CMD)0x3f6 输出控制字节，建立相应的硬盘控制方式。方式建立后即可按上面顺序发送参数和命令。步骤为：

检测控制器空闲状态：CPU 通过读主状态寄存器，若位 7 为 0，表示控制器空闲。若在规定时间内控制器一直处于忙状态，则判为超时出错。

检测驱动器是否就绪：CPU 判断主状态寄存器位 6 是否为 1 来看驱动器是否就绪。为 1 则可输出参数和命令。

输出命令块：按顺序输出分别向对应端口输出参数和命令。

CPU 等待中断产生：命令执行后，由硬盘控制器产生中断请求信号（IRQ14 -- 对应中断 int46）或置控制器状态为空闲，表明操作结束或表示请求扇区传输（多扇区读/写）。

检测操作结果：CPU 再次读主状态寄存器，若位 0 等于 0 则表示命令执行成功，否则失败。若失败则可进一步查询错误寄存器(HD_ERROR)取错误码。

6.5.3.3 硬盘基本参数表

中断向量表中，int 0x41 的中断向量位置（4 * 0x41 = 0x0000:0x0104）存放的并不是中断程序的地址而是第一个硬盘的基本参数表，见表 6-8 所示。对于 100%兼容的 BIOS 来说，这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量中。

表 6 - 8 硬盘基本参数信息表

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头数
0x03	字	开始减小写电流的柱面(仅 PC XT 使用，其它为 0)
0x05	字	开始写前预补偿柱面号（乘 4）
0x07	字节	最大 ECC 猝发长度（仅 XT 使用，其它为 0）
0x08	字节	控制字节（驱动器步进选择） 位 0 未用 位 1 保留(0) (关闭 IRQ) 位 2 允许复位 位 3 若磁头数大于 8 则置 1 位 4 未用(0) 位 5 若在柱面数+1 处有生产商的坏区图，则置 1

		位 6 禁止 ECC 重试 位 7 禁止访问重试。
0x09	字节	标准超时值（仅 XT 使用，其它为 0）
0x0A	字节	格式化超时值（仅 XT 使用，其它为 0）
0x0B	字节	检测驱动器超时值（仅 XT 使用，其它为 0）
0x0C	字	磁头着陆(停止)柱面号
0x0E	字节	每磁道扇区数
0x0F	字节	保留。

6.5.3.4 硬盘设备号命名方式

硬盘的主设备号是 3。其它设备的主设备号分别为：

1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道

由于 1 个硬盘中可以存在 1--4 个分区，因此硬盘还依据分区不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成：

设备号=主设备号*256+ 次设备号

也即 $\text{dev_no} = (\text{major} \ll 8) + \text{minor}$

两个硬盘的所有逻辑设备号见表 6-9 所示。

表 6-9 硬盘逻辑设备号

逻辑设备号	对应设备文件	说明
0x300	/dev/hd0	代表整个第 1 个硬盘
0x301	/dev/hd1	表示第 1 个硬盘的第 1 个分区
0x302	/dev/hd2	表示第 1 个硬盘的第 2 个分区
0x303	/dev/hd3	表示第 1 个硬盘的第 3 个分区
0x304	/dev/hd4	表示第 1 个硬盘的第 4 个分区
0x305	/dev/hd5	代表整个第 2 个硬盘
0x306	/dev/hd6	表示第 2 个硬盘的第 1 个分区
0x307	/dev/hd7	表示第 2 个硬盘的第 2 个分区
0x308	/dev/hd8	表示第 2 个硬盘的第 3 个分区
0x309	/dev/hd9	表示第 2 个硬盘的第 4 个分区

其中 0x300 和 0x305 并不与哪个分区对应，而是代表整个硬盘。

从 linux 内核 0.95 版后已经不使用这种烦琐的命名方式，而是使用与现在相同的命名方法了。

6.5.3.5 硬盘分区表

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号和扇区号，见表 6-10 所示。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。

表 6-10 硬盘分区表结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。

			0x00-不从该分区引导操作系统；0x80-从该分区引导操作系统。
0x01	head	字节	分区起始磁头号。
0x02	sector	字节	分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux ...
0x05	end_head	字节	分区的结束磁头号。
0x06	end_sector	字节	结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	结束柱面号低 8 位。
0x08--0x0b	start_sect	长字	分区起始物理扇区号。
0x0c--0x0f	nr_sects	长字	分区占用的扇区数。

6.6 ll_rw_blk.c 程序

6.6.1 功能描述

该程序主要用于执行低层块设备读/写操作，是本章所有块设备与系统其它部分的接口程序。其它程序通过调用该程序的低级块读写函数 `ll_rw_block()` 来读写块设备中的数据。该函数的主要功能是为块设备创建块设备读写请求项，并插入到指定块设备请求队列中。实际的读写操作则是由设备的请求项处理函数 `request_fn()` 完成。对于硬盘操作，该函数是 `do_hd_request()`；对于软盘操作，该函数是 `do_fd_request()`；对于虚拟盘则是 `do_rd_request()`。若 `ll_rw_block()` 为一个块设备建立起一个请求项，并通过测试块设备的当前请求项指针为空而确定设备空闲时，就会设置该新建的请求项为当前请求项，并直接调用 `request_fn()` 对该请求项进行操作。否则就会使用电梯算法将新建的请求项插入到该设备的请求项链表中等待处理。而当 `request_fn()` 结束对一个请求项的处理，就会把该请求项从链表中删除。

由于 `request_fn()` 在每个请求项处理结束时，都会通过中断回调 C 函数（主要是 `read_intr()` 和 `write_intr()`）再次调用 `request_fn()` 自身去处理链表中其余的请求项，因此，只要设备的请求项链表（或者称为队列）中有未处理的请求项存在，都会陆续地被处理，直到设备的请求项链表是空为止。当请求项链表空时，`request_fn()` 将不再向驱动器控制器发送命令，而是立刻退出。因此，对 `request_fn()` 函数的循环调用就此结束。参见图 6-4 所示。

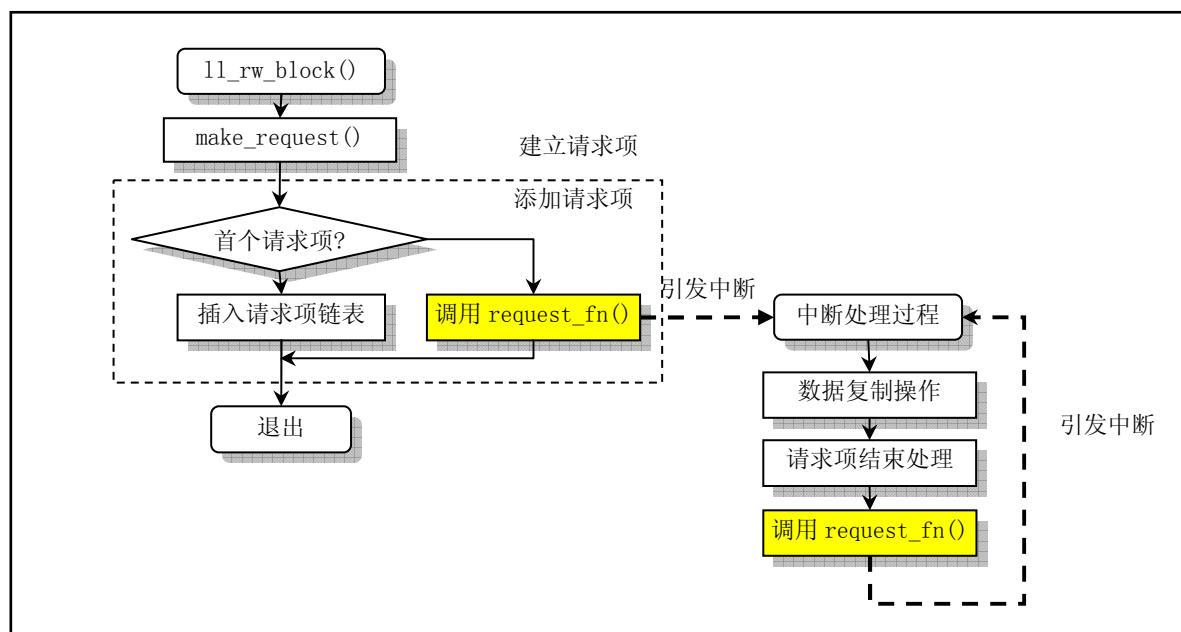


图 6-4 ll_rw_block 调用序列

对于虚拟盘设备，由于它的读写操作不牵涉到上述与外界硬件设备同步操作，因此没有上述的中断处理过程。当前请求项对虚拟设备的读写操作完全在 do_rd_request()中实现。

6.6.2 代码注释

程序 6-4 linux/kernel/blk_drv/ll_rw_blk.c

```

1  /*
2  *  linux/kernel/blk_dev/ll_rw.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
7  /*
8  *  This handles all read/write requests to block devices
9  */
10 /*
11  *  该程序处理块设备的所有读/写操作。
12  */
13 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
14 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
15                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
16 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
17 #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
18 #include "blk.h"            // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。
19
20 /*
21  *  The request-struct contains all necessary data
22  *  to load a nr of sectors into memory
23  */

```

```

/*
 * 请求结构中含有加载 nr 扇区数据到内存中的所有必须的信息。
 */
21 struct request request[NR_REQUEST];
22
23 /*
24  * used to wait on when there are no free requests
25  */
/* 是用于在请求数组没有空闲项时进程的临时等待处 */
26 struct task_struct * wait_for_request = NULL;
27
28 /* blk_dev_struct is:
29  *      do_request-address
30  *      next-request
31  */
/* blk_dev_struct 块设备结构是: (kernel/blk_drv/blk.h, 23)
 *      do_request-address    // 对应主设备号的请求处理程序指针。
 *      current-request      // 该设备的下一个请求。
 */
// 该数组使用主设备号作为索引。实际内容将在各种块设备驱动程序初始化时填入。例如, 硬盘
// 驱动程序进行初始化时 (hd.c, 343 行), 第一条语句即用于设置 blk_dev[3] 的内容。
32 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
33     { NULL, NULL },          /* no_dev */    // 0 - 无设备。
34     { NULL, NULL },          /* dev mem */   // 1 - 内存。
35     { NULL, NULL },          /* dev fd */    // 2 - 软驱设备。
36     { NULL, NULL },          /* dev hd */    // 3 - 硬盘设备。
37     { NULL, NULL },          /* dev ttyx */  // 4 - ttyx 设备。
38     { NULL, NULL },          /* dev tty */   // 5 - tty 设备。
39     { NULL, NULL }           /* dev lp */    // 6 - lp 打印机设备。
40 };
41
// 锁定指定的缓冲区 bh。如果指定的缓冲区已经被其它任务锁定, 则使自己睡眠 (不可中断地等待),
// 直到被执行解锁缓冲区的任务明确地唤醒。
42 static inline void lock_buffer(struct buffer_head * bh)
43 {
44     cli();                      // 清中断许可。
45     while (bh->b_lock)          // 如果缓冲区已被锁定, 则睡眠, 直到缓冲区解锁。
46         sleep_on(&bh->b_wait);
47     bh->b_lock=1;               // 立刻锁定该缓冲区。
48     sti();                      // 开中断。
49 }
50
// 释放 (解锁) 锁定的缓冲区。
51 static inline void unlock_buffer(struct buffer_head * bh)
52 {
53     if (!bh->b_lock)             // 如果该缓冲区并没有被锁定, 则打印出错信息。
54         printk("ll_rw_block.c: buffer not locked\n");
55     bh->b_lock = 0;              // 清锁定标志。
56     wake_up(&bh->b_wait);       // 唤醒等待该缓冲区的任务。
57 }
58
59 /*
60  * add-request adds a request to the linked list.

```

```

61  * It disables interrupts so that it can muck with the
62  * request-lists in peace.
63  */
/*
* add-request() 向链表中加入一项请求。它会关闭中断，
* 这样就能安全地处理请求链表了 */
*/
///// 向链表中加入请求项。参数 dev 指定块设备，req 是请求项结构信息指针。

64 static void add_request(struct blk_dev_struct * dev, struct request * req)
65 {
66     struct request * tmp;
67
68     req->next = NULL;
69     cli();                                // 关中断。
70     if (req->bh)
71         req->bh->b_dirt = 0;                // 清缓冲区“脏”标志。
// 如果 dev 的当前请求(current_request)子段为空，则表示目前该设备没有请求项，本次是第 1 个
// 请求项，因此可将块设备当前请求指针直接指向该请求项，并立刻执行相应设备的请求函数。
72     if (!(tmp = dev->current_request)) {
73         dev->current_request = req;
74         sti();                            // 开中断。
75         (dev->request_fn)();              // 执行设备请求函数，对于硬盘是 do_hd_request()。
76         return;
77     }
// 如果目前该设备已经有请求项在等待，则首先利用电梯算法搜索最佳插入位置，然后将当前请求插入
// 到请求链表中。电梯算法的作用是让磁盘磁头的移动距离最小，从而改善硬盘访问时间。
78     for (; tmp->next; tmp=tmp->next)
79         if ((IN_ORDER(tmp, req) ||
80             !IN_ORDER(tmp, tmp->next)) &&
81             IN_ORDER(req, tmp->next))
82             break;
83     req->next=tmp->next;
84     tmp->next=req;
85     sti();
86 }
87
///// 创建请求项并插入请求队列。参数是：主设备号 major，命令 rw，存放数据的缓冲区头指针 bh。
88 static void make_request(int major, int rw, struct buffer_head * bh)
89 {
90     struct request * req;
91     int rw_ahead;
92
93     /* WRITEA/READA is special case - it is not really needed, so if the */
94     /* buffer is locked, we just forget about it, else it's a normal read */
// WRITEA/READA 是一种特殊情况 - 它们并非必要，所以如果缓冲区已经上锁，*/
// 我们就不管它而退出，否则的话就执行一般的读/写操作。 */
// 这里' READ' 和' WRITE' 后面的' A' 字符代表英文单词 Ahead，表示提前预读/写数据块的意思。
// 对于命令是 READA/WRITEA 的情况，当指定的缓冲区正在使用，已被上锁时，就放弃预读/写请求。
// 否则就作为普通的 READ/WRITE 命令进行操作。
95     if (rw_ahead = (rw == READA || rw == WRITEA)) {
96         if (bh->b_lock)
97             return;

```

```

98         if (rw == READA)
99             rw = READ;
100        else
101            rw = WRITE;
102    }
// 如果命令不是 READ 或 WRITE 则表示内核程序有错, 显示出错信息并死机。
103    if (rw!=READ && rw!=WRITE)
104        panic("Bad block dev command, must be R/W/RA/WA");
// 锁定缓冲区, 如果缓冲区已经上锁, 则当前任务(进程)就会睡眠, 直到被明确地唤醒。
105    lock_buffer(bh);
// 如果命令是写并且缓冲区数据不脏(没有被修改过), 或者命令是读并且缓冲区数据是更新过的,
// 则不用添加这个请求。将缓冲区解锁并退出。
106    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
107        unlock_buffer(bh);
108        return;
109    }
110 repeat:
111 /* we don't allow the write-requests to fill up the queue completely:
112  * we want some room for reads: they take precedence. The last third
113  * of the requests are only for reads.
114  */
// 我们不能让队列中全都是写请求项: 我们需要为读请求保留一些空间: 读操作
// 是优先的。请求队列的后三分之一空间是为读准备的。
// 请求项是从请求数组末尾开始搜索空项填入的。根据上述要求, 对于读命令请求, 可以直接
// 从队列末尾开始操作, 而写请求则只能从队列 2/3 处向队列头处搜索空项填入。
115    if (rw == READ)
116        req = request+NR_REQUEST; // 对于读请求, 将队列指针指向队列尾部。
117    else
118        req = request+((NR_REQUEST*2)/3); // 对于写请求, 队列指针指向队列 2/3 处。
119 /* find an empty request */
// 搜索一个空请求项 */
// 从后向前搜索, 当请求结构 request 的 dev 字段值=-1 时, 表示该项未被占用。
120    while (--req >= request)
121        if (req->dev<0)
122            break;
123 /* if none found, sleep on new requests: check for rw_ahead */
// 如果没有找到空闲项, 则让该次新请求睡眠: 需检查是否提前读/写 */
// 如果没有一项是空闲的(此时 request 数组指针已经搜索越过头部), 则查看此次请求是否是
// 提前读/写 (READA 或 WRITEA), 如果是则放弃此次请求。否则让本次请求睡眠(等待请求队列
// 腾出空项), 过一会再来搜索请求队列。
124    if (req < request) { // 如果请求队列中没有空项, 则
125        if (rw_ahead) { // 如果是提前读/写请求, 则解锁缓冲区, 退出。
126            unlock_buffer(bh);
127            return;
128        }
129        sleep_on(&wait_for_request); // 否则让本次请求睡眠, 过会再查看请求队列。
130        goto repeat;
131    }
132 /* fill up the request-info, and add it to the queue */
// 向空闲请求项中填写请求信息, 并将其加入队列中 */
// 程序执行到这里表示已找到一个空闲请求项。请求结构参见(kernel/blk_drv/blk.h, 23)。
133    req->dev = bh->b_dev; // 设备号。

```

```

134     req->cmd = rw;                                // 命令(READ/WRITE)。
135     req->errors=0;                                // 操作时产生的错误次数。
136     req->sector = bh->b_blocknr<<1;              // 起始扇区。块号转换成扇区号(1块=2扇区)。
137     req->nr_sectors = 2;                          // 读写扇区数。
138     req->buffer = bh->b_data;                      // 数据缓冲区。
139     req->waiting = NULL;                          // 任务等待操作执行完成的地方。
140     req->bh = bh;                                // 缓冲块头指针。
141     req->next = NULL;                             // 指向下一请求项。
142     add_request(major+blk_dev, req);              // 将请求项加入队列中 (blk_dev[major], req)。
143 }
144
145 // 低层读写数据块函数，是块设备与系统其它部分的接口函数。
146 // 该函数在 fs/buffer.c 中被调用。主要功能是创建块设备读写请求项并插入到指定块设备请求队列中。
147 // 实际的读写操作则是由设备的 request_fn() 函数完成。对于硬盘操作，该函数是 do_hd_request();
148 // 对于软盘操作，该函数是 do_fd_request(); 对于虚拟盘则是 do_rd_request()。
149 // 另外，需要读/写块设备的信息已保存在缓冲块头结构中，如设备号、块号。
150 // 参数: rw - READ、READA、WRITE 或 WRITEA 命令; bh - 数据缓冲块头指针。
151 void ll_rw_block(int rw, struct buffer_head * bh)
152 {
153     unsigned int major;                          // 主设备号（对于硬盘是 3）。
154
155     // 如果设备的主设备号不存在或者该设备的读写操作函数不存在，则显示出错信息，并返回。
156     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
157         !(blk_dev[major].request_fn)) {
158         printk("Trying to read nonexistent block-device\n");
159         return;
160     }
161     make_request(major, rw, bh);                  // 创建请求项并插入请求队列。
162 }
163
164 // 块设备初始化函数，由初始化程序 main.c 调用 (init/main.c, 128)。
165 // 初始化请求数组，将所有请求项置为空闲项(dev = -1)。有 32 项(NR_REQUEST = 32)。
166 void blk_dev_init(void)
167 {
168     int i;
169
170     for (i=0 ; i<NR_REQUEST ; i++) {
171         request[i].dev = -1;
172         request[i].next = NULL;
173     }
174 }

```

6.7 ramdisk.c 程序

6.7.1 功能描述

本文件是内存虚拟盘（Ram Disk）驱动程序，由 Theodore Ts'o 编制。虚拟盘设备是一种利用物理内存来模拟实际磁盘存储数据的方式。其目的主要是为了提高对“磁盘”数据的读写操作速度。除了需要

占用一些宝贵的内存资源外，其主要缺点是一旦系统崩溃或关闭，虚拟盘中的所有数据将全部消失。因此虚拟盘中通常存放一些系统命令等常用工具程序或临时数据，而非重要的输入文档。

当在 `linux/Makefile` 文件中定义了常量 `RAMDISK`，内核初始化程序就会在内存中划出一块该常量值指定大小的内存区域用于存放虚拟盘数据。虚拟盘在物理内存中所处的具体位置是在内核初始化阶段确定的（`init/main.c`，123 行），它位于内核高速缓冲区和主内存区之间。若运行的机器含有 16MB 的物理内存，那么虚拟盘区域会被设置在内存 4MB 开始处，虚拟盘容量即等于 `RAMDISK` 的值（KB）。若 `RAMDISK=512`，则此时内存情况见图 6-5 所示。

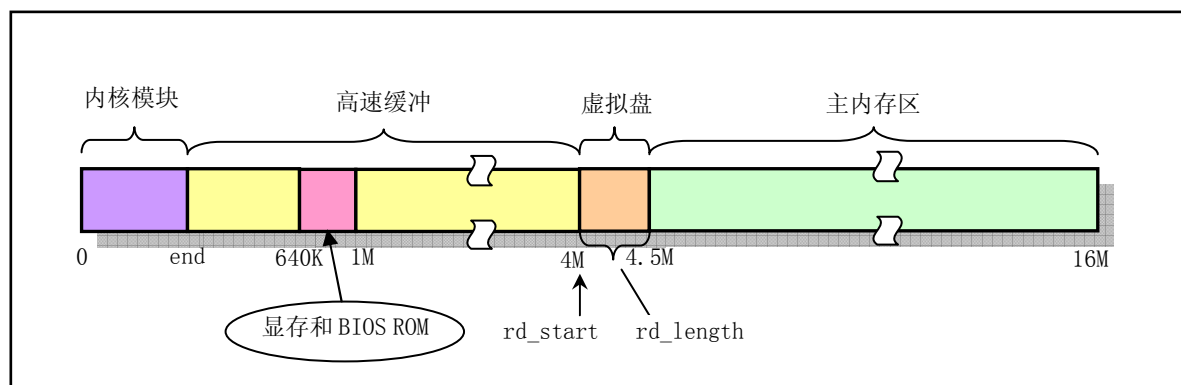


图 6-5 虚拟盘在 16MB 内存系统中所处的具体位置

对虚拟盘设备的读写访问操作原则上完全按照对普通磁盘的操作进行，也需要按照块设备的访问方式对其进行读写操作。由于在实现上不牵涉与外部控制器或设备进行同步操作，因此其实现方式比较简单。对于数据在系统与设备之间的“传送”只需执行内存数据块复制操作即可。

本程序包含 3 个函数。`rd_init()`会在系统初始化时被 `init/main.c` 程序调用，用于确定虚拟盘在物理内存中的具体位置和大小；`do_rd_request()`是虚拟盘设备的请求项操作函数，对当前请求项实现虚拟盘数据的访问操作；`rd_load()`是虚拟盘根文件加载函数。在系统初始化阶段，该函数被用于尝试从启动引导盘上指定的磁盘块位置开始处把一个根文件系统加载到虚拟盘中。在函数中，这个起始磁盘块位置被定为 256。当然你也可以根据自己的具体要求修改这个值，只要保证这个值所规定的磁盘容量能容纳内核映象文件即可。这样一个由内核引导映象文件（`Bootimage`）加上根文件系统映象文件（`Rootimage`）组合而成的“二合一”磁盘，就可以象启动 DOS 系统盘那样来启动 Linux 系统。关于这种组合盘的制作方式留给读者自己考虑。

在进行正常的根文件系统加载之前，系统会首先执行 `rd_load()`函数，试图从磁盘的第 257 块中读取根文件系统超级块。若成功，就把该根文件映象文件读到内存虚拟盘中，并把根文件系统设备标志 `ROOT_DEV` 设置为虚拟盘设备（0x0101），否则退出 `rd_load()`，系统继续从别的设备上执行根文件加载操作。

6.7.2 代码注释

程序 6-5 `linux/kernel/blk_drv/ramdisk.c`

```

1 /*
2  * linux/kernel/blk_drv/ramdisk.c
3  *
4  * Written by Theodore Ts'o, 12/2/91
5  */
/* 由 Theodore Ts'o 编制，12/2/91
*/

```



```

// Theodore Ts'o (Ted Ts'o) 是 linux 社区中的著名人物。Linux 在世界范围内的流行也有他很大的
// 功劳，早在 Linux 操作系统刚问世时，他就怀着极大的热情为 linux 的发展提供了 maillist，并
// 在北美洲地区最早设立了 linux 的 ftp 站点 (tsx-ll.mit.edu)，而且至今仍然为广大 linux 用户
// 提供服务。他对 linux 作出的最大贡献之一是提出并实现了 ext2 文件系统。该文件系统已成为
// linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件系统，大大提高了文件系统的
// 稳定性和访问效率。作为对他的推崇，第 97 期（2002 年 5 月）的 linuxjournal 期刊将他作为
// 了封面人物，并对他进行了采访。目前，他为 IBM linux 技术中心工作，并从事着有关 LSB
// (Linux Standard Base) 等方面的工作。(他的主页: http://thunk.org/tytso/)

6
7 #include <string.h>          // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8
9 #include <linux/config.h>    // 内核配置头文件。定义键盘语言和硬盘类型 (HD_TYPE) 可选项。
10 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/fs.h>       // 文件系统头文件。定义文件表结构 (file, buffer_head, m_inode 等)。
12 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
15 #include <asm/memory.h>     // 内存拷贝头文件。含有 memcpy() 嵌入式汇编宏函数。
16
17 #define MAJOR_NR 1          // RAM 盘主设备号是 1。主设备号必须在 blk.h 之前被定义。
18 #include "blk.h"
19
20 char    *rd_start;          // 虚拟盘在内存中的起始位置。在 52 行初始化函数 rd_init() 中
    // 确定。参见 (init/main.c, 124) (缩写 rd_代表 ramdisk_)。
21 int     rd_length = 0;      // 虚拟盘所占内存大小 (字节)。
22
    // 虚拟盘当前请求项操作函数。程序结构与 do_hd_request() 类似 (hd.c, 294)。
    // 在低级块设备接口函数 ll_rw_block() 建立了虚拟盘 (rd) 的请求项并添加到 rd 的链表之后，
    // 就会调用该函数对 rd 当前请求项进行处理。该函数首先计算当前请求项中指定的起始扇区对应
    // 虚拟盘所处内存的起始位置 addr 和要求的扇区数对应的字节长度值 len，然后根据请求项中的
    // 命令进行操作。若是写命令 WRITE，就把请求项所指缓冲区中的数据直接复制到内存位置 addr
    // 处。若是读操作则反之。数据复制完成后即可直接调用 end_request() 对本次请求项作结束处理。
    // 然后跳转到函数开始处再去处理下一个请求项。
23 void do_rd_request(void)
24 {
25     int     len;
26     char    *addr;
27
    // 检测请求项的合法性，若已没有请求项则退出 (参见 blk.h, 127)。
28     INIT_REQUEST;
    // 下面语句取得 ramdisk 的起始扇区对应的内存起始位置和内存长度。
    // 其中 sector << 9 表示 sector * 512，CURRENT 定义为 (blk_dev[MAJOR_NR].current_request)。
29     addr = rd_start + (CURRENT->sector << 9);
30     len = CURRENT->nr_sectors << 9;
    // 如果子设备号不为 1 或者对应内存起始位置 > 虚拟盘末尾，则结束该请求，并跳转到 repeat 处。
    // 标号 repeat 定义在宏 INIT_REQUEST 内，位于宏的开始处，参见 blk.h, 127 行。
31     if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32         end_request(0);
33         goto repeat;
34     }
    // 如果是写命令 (WRITE)，则将请求项中缓冲区的内容复制到 addr 处，长度为 len 字节。
35     if (CURRENT->cmd == WRITE) {

```

```

36         (void ) memcpy(addr,
37                         CURRENT->buffer,
38                         len);
// 如果是读命令(READ), 则将 addr 开始的内容复制到请求项中缓冲区中, 长度为 len 字节。
39     } else if (CURRENT->cmd == READ) {
40         (void ) memcpy(CURRENT->buffer,
41                         addr,
42                         len);
// 否则显示命令不存在, 死机。
43     } else
44         panic("unknown ramdisk-command");
// 请求项成功后处理, 置更新标志。并继续处理本设备的下一请求项。
45     end_request(1);
46     goto repeat;
47 }
48
49 /*
50  * Returns amount of memory which needs to be reserved.
51  */
// 返回内存虚拟盘 ramdisk 所需的内存量 */
// 虚拟盘初始化函数。确定虚拟盘在内存中的起始地址, 长度。并对整个虚拟盘区清零。
52 long rd_init(long mem_start, int length)
53 {
54     int    i;
55     char   *cp;
56
57     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // do_rd_request()。
58     rd_start = (char *) mem_start;                // 对于 16MB 系统, 该值为 4MB。
59     rd_length = length;
60     cp = rd_start;
61     for (i=0; i < length; i++)
62         *cp++ = '\0';
63     return(length);
64 }
65
66 /*
67  * If the root device is the ram disk, try to load it.
68  * In order to do this, the root device is originally set to the
69  * floppy, and we later change it to be ram disk.
70  */
//
// * 如果根文件系统设备(root device)是 ramdisk 的话, 则尝试加载它。root device 原先是指向
// * 软盘的, 我们将它改成指向 ramdisk。
// */
//// 尝试把根文件系统加载到虚拟盘中。
// 该函数将在内核设置函数 setup() (hd.c, 156 行) 中被调用。另外, 1 磁盘块 = 1024 字节。
71 void rd_load(void)
72 {
73     struct buffer_head *bh;           // 高速缓冲块头指针。
74     struct super_block s;             // 文件超级块结构。
75     int block = 256;                  /* Start at block 256 */
76     int i = 1;                        /* 表示根文件系统映象文件在 boot 盘第 256 磁盘块开始处*/
77     int nblocks;

```

```

78     char                *cp;                /* Move pointer */
79
80     if (!rd_length)                        // 如果 ramdisk 的长度为零, 则退出。
81         return;
82     printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
83            (int) rd_start);                // 显示 ramdisk 的大小以及内存起始位置。
84     if (MAJOR(ROOT_DEV) != 2)              // 如果此时根文件设备不是软盘, 则退出。
85         return;
86     // 读软盘块 256+1, 256, 256+2。breada() 用于读取指定的数据块, 并标出还需要读的块, 然后返回
87     // 含有数据块的缓冲区指针。如果返回 NULL, 则表示数据块不可读(fs/buffer.c, 322)。
88     // 这里 block+1 是指磁盘上的超级块。
89     bh = breada(ROOT_DEV, block+1, block, block+2, -1);
90     if (!bh) {
91         printk("Disk error while looking for ramdisk!\n");
92         return;
93     }
94     // 将 s 指向缓冲区中的磁盘超级块。(d_super_block 磁盘中超级块结构)。
95     *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
96     brelse(bh);
97     if (s.s_magic != SUPER_MAGIC)          // 如果超级块中魔数不对, 则说明不是 minix 文件系统。
98         /* No ram disk image present, assume normal floppy boot */
99         /* 磁盘中没有 ramdisk 映像文件, 退出去执行通常的软盘引导 */
100         return;
101     // 块数 = 逻辑块数(区段数) * 2^(每区段块数的次方)。
102     // 如果数据块数大于内存中虚拟盘所能容纳的块数, 则也不能加载, 显示出错信息并返回。否则显示
103     // 加载数据块信息。
104     nblocks = s.s_nzones << s.s_log_zone_size;
105     if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
106         printk("Ram disk image too big! (%d blocks, %d avail)\n",
107            nblocks, rd_length >> BLOCK_SIZE_BITS);
108         return;
109     }
110     printk("Loading %d bytes into ram disk... 0000k",
111        nblocks << BLOCK_SIZE_BITS);
112     // cp 指向虚拟盘起始处, 然后将磁盘上的根文件系统映像文件复制到虚拟盘上。
113     cp = rd_start;
114     while (nblocks) {
115         if (nblocks > 2) // 如果需读取的块数多于 3 块则采用超前预读方式读数据块。
116             bh = breada(ROOT_DEV, block, block+1, block+2, -1);
117         else // 否则就单块读取。
118             bh = bread(ROOT_DEV, block);
119         if (!bh) {
120             printk("I/O error on block %d, aborting load\n",
121                block);
122             return;
123         }
124         (void) memcpy(cp, bh->b_data, BLOCK_SIZE); // 将缓冲区中的数据复制到 cp 处。
125         brelse(bh); // 释放缓冲区。
126         printk("\010\010\010\010\010%4dk", i); // 打印加载块计数值。
127         cp += BLOCK_SIZE; // 虚拟盘指针前移。
128         block++;
129         nblocks--;
130         i++;

```

```

122     }
123     printk("\010\010\010\010\010done \n");
124     ROOT_DEV=0x0101;           // 修改 ROOT_DEV 使其指向虚拟盘 ramdisk。
125 }
126

```

6.8 floppy.c 程序

6.8.1 功能描述

本程序是软盘控制器驱动程序。与其它块设备驱动程序一样，该程序也以请求项操作函数 `do_fd_request()` 为主，执行对软盘上数据的读写操作。

考虑到软盘驱动器在不工作时马达通常不转，所以在实际能对驱动器中的软盘进行读写操作之前，我们需要等待马达启动并达到正常的运行速度。与计算机的运行速度相比，这段时间较长，通常需要 0.5 秒左右的时间。

另外，当对一个磁盘的读写操作完毕，我们也需要让驱动器停止转动，以减少对磁盘表面的摩擦。但我们也不能在对磁盘操作完后就立刻让它停止转动。因为，可能马上又需要对其进行读写操作。因此，在一个驱动器没有操作后还是需要让驱动器空转一段时间，以等待可能到来的读写操作，若驱动器在一个较长时间内都没有操作，则程序让它停止转动。这段维持旋转的时间可设定在大约 3 秒左右。

当一个磁盘的读写操作发生错误，或某些其它情况导致一个驱动器的马达没有被关闭。此时我们也需要让系统在一定时间之后自动将其关闭。Linux 在程序中把这个延时值设定在 100 秒。

由此可见，在对软盘驱动器进行操作时会用到很多延时（定时）操作。因此在该驱动程序中涉及较多的定时处理函数。还有几个与定时处理关系比较密切的函数被放在了 `kernel/sched.c` 中（行 201-262）。这是软盘驱动程序与硬盘驱动程序之间的最大区别，也是软盘驱动程序比硬盘驱动程序复杂的原因。

虽然本程序比较复杂，但对软盘读写操作的工作原理却与其它块设备是一样的。本程序也是使用请求项和请求项链表结构来处理所有对软盘的读写操作。因此请求项操作函数 `do_fd_request()` 仍然是本程序中的重要函数之一。在阅读时应该以该函数为主线展开。另外，软盘控制器的使用比较复杂，其中涉及到很多控制器的执行状态和标志。因此在阅读时，还需要频繁地参考程序后的有关说明以及本程序的头文件 `include/linux/fdreg.h`。该文件定义了所有软盘控制器参数常量，并说明了这些常量的含义。

6.8.2 代码注释

程序 6-6 linux/kernel/blk_drv/floppy.c

```

1  /*
2   *  linux/kernel/floppy.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  02.12.91 - Changed to static variables to indicate need for reset
9   *  and recalibrate. This makes some things easier (output_byte reset
10  *  checking etc), and means less interrupt jumping in case of errors,
11  *  so the code is hopefully easier to understand.
12  */
13  /*

```

```

* 02.12.91 - 修改成静态变量，以适应复位和重新校正操作。这使得某些事情
* 做起来较为方便（output_byte 复位检查等），并且意味着在出错时中断跳转
* 要少一些，所以也希望代码能更容易被理解。
*/
13
14 /*
15  * This file is certainly a mess. I've tried my best to get it working,
16  * but I don't like programming floppies, and I have only one anyway.
17  * Urgel. I should check for more errors, and do more graceful error
18  * recovery. Seems there are problems with several drives. I've tried to
19  * correct them. No promises.
20  */
/*
* 这个文件当然比较混乱。我已经尽我所能使其能够工作，但我不喜欢软驱编程，
* 而且我也只有一个软驱。另外，我应该做更多的查错工作，以及改正更多的错误。
* 对于某些软盘驱动器，本程序好象还存在一些问题。我已经尝试着进行纠正了，
* 但不能保证问题已消失。
*/
21
22 /*
23  * As with hd.c, all routines within this file can (and will) be called
24  * by interrupts, so extreme caution is needed. A hardware interrupt
25  * handler may not sleep, or a kernel panic will happen. Thus I cannot
26  * call "floppy-on" directly, but have to set a special timer interrupt
27  * etc.
28  *
29  * Also, I'm not certain this works on more than 1 floppy. Bugs may
30  * abund.
31  */
/*
* 如同 hd.c 文件一样，该文件中的所有子程序都能够被中断调用，所以需要特别
* 地小心。硬件中断处理程序是不能睡眠的，否则内核就会傻掉（死机）☹。因此不能
* 直接调用“floppy-on”，而只能设置一个特殊的定时中断等。
*
* 另外，我不能保证该程序能在多于 1 个软驱的系统上工作，有可能存在错误。
*/
32
33 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
// 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
34 #include <linux/fs.h> // 文件系统头文件。定义文件表结构（file, buffer_head, m_inode 等）。
35 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
36 #include <linux/fdreg.h> // 软驱头文件。含有软盘控制器参数的一些定义。
37 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
38 #include <asm/io.h> // io 头文件。定义硬件端口输入/输出宏汇编语句。
39 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
40
41 #define MAJOR_NR 2 // 软驱的主设备号是 2。
42 #include "blk.h" // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。
43
44 static int recalibrate = 0; // 标志：需要重新校正。
45 static int reset = 0; // 标志：需要进行复位操作。
46 static int seek = 0; // 标志：需要执行寻道。
47

```

```

// 当前数字输出寄存器(Digital Output Register), 定义在 sched.c, 204 行。
// 该变量是软驱操作中的重要标志变量, 请参见程序后对 DOR 寄存器的说明。
48 extern unsigned char current_DOR;
49
50 #define immoutb_p(val,port) \      // 字节直接输出(嵌入汇编语言宏)。
51 __asm__( "outb %0,%1\n\tjmp lf\n1:\tjmp lf\n1::\"a\" ((char) (val)), \"i\" (port))
52
// 这两个定义用于计算软驱的设备号。次设备号 = TYPE*4 + DRIVE。计算方法参见列表后。
53 #define TYPE(x) ((x)>>2)          // 软驱类型(2--1.2Mb, 7--1.44Mb)。
54 #define DRIVE(x) ((x)&0x03)        // 软驱序号(0--3 对应 A--D)。
55 /*
56  * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
57  * max 8 times - some types of errors increase the errorcount by 2,
58  * so we might actually retry only 5-6 times before giving up.
59  */
/*
* 注意, 下面定义 MAX_ERRORS=8 并不表示对每次读错误尝试最多 8 次 - 有些类型
* 的错误将把出错计数值乘 2, 所以我们实际上在放弃操作之前只需尝试 5-6 遍即可。
*/
60 #define MAX_ERRORS 8
61
62 /*
63  * globals used by 'result()'
64  */
/* 下面是函数 'result()' 使用的全局变量 */
// 这些状态字节中各比特位的含义请参见 include/linux/fdreg.h 头文件。
65 #define MAX_REPLIES 7              // FDC 最多返回 7 字节的结果信息。
66 static unsigned char reply_buffer[MAX_REPLIES]; // 存放 FDC 返回的应答结果信息。
67 #define ST0 (reply_buffer[0])      // 返回结果状态字节 0。
68 #define ST1 (reply_buffer[1])      // 返回结果状态字节 1。
69 #define ST2 (reply_buffer[2])      // 返回结果状态字节 2。
70 #define ST3 (reply_buffer[3])      // 返回结果状态字节 3。
71
72 /*
73  * This struct defines the different floppy types. Unlike minix
74  * linux doesn't have a "search for right type"-type, as the code
75  * for that is convoluted and weird. I've got enough problems with
76  * this driver as it is.
77  *
78  * The 'stretch' tells if the tracks need to be boubled for some
79  * types (ie 360kB diskette in 1.2MB drive etc). Others should
80  * be self-explanatory.
81  */
/*
* 下面的软盘结构定义了不同的软盘类型。与 minix 不同的是, linux 没有
* "搜索正确的类型"-类型, 因为对其处理的代码令人费解且怪怪的。本程序
* 已经让我遇到了许多的问题了。
*
* 对某些类型的软盘(例如在 1.2MB 驱动器中的 360kB 软盘等), 'stretch' 用于
* 检测磁道是否需要特殊处理。其它参数应该是自明的。
*/
// 软盘参数有:
// size      大小(扇区数);

```



```

// sect      每磁道扇区数;
// head      磁头数;
// track     磁道数;
// stretch  对磁道是否要特殊处理 (标志);
// gap       扇区间隙长度(字节数);
// rate      数据传输速率;
// spec1     参数 (高 4 位步进速率, 低四位磁头卸载时间)。
82 static struct floppy_struct {
83     unsigned int size, sect, head, track, stretch;
84     unsigned char gap, rate, spec1;
85 } floppy_type[] = {
86     { 0, 0, 0, 0, 0, 0x00, 0x00, 0x00 },      /* no testing */
87     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF },   /* 360kB PC diskettes */
88     { 2400, 15, 2, 80, 0, 0x1B, 0x00, 0xDF },  /* 1.2 MB AT-diskettes */
89     { 720, 9, 2, 40, 1, 0x2A, 0x02, 0xDF },   /* 360kB in 720kB drive */
90     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF },  /* 3.5" 720kB diskette */
91     { 720, 9, 2, 40, 1, 0x23, 0x01, 0xDF },   /* 360kB in 1.2MB drive */
92     { 1440, 9, 2, 80, 0, 0x23, 0x01, 0xDF },  /* 720kB in 1.2MB drive */
93     { 2880, 18, 2, 80, 0, 0x1B, 0x00, 0xCF }, /* 1.44MB diskette */
94 };
95 /*
96  * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
97  * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
98  * H is head unload time (1=16ms, 2=32ms, etc)
99  *
100 * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
101 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
102 */
/*
 * 上面速率 rate: 0 表示 500kb/s, 1 表示 300kbps, 2 表示 250kbps。
 * 参数 spec1 是 0xSH, 其中 S 是步进速率 (F=1 毫秒, E=2ms, D=3ms 等),
 * H 是磁头卸载时间 (1=16ms, 2=32ms 等)
 *
 * spec2 是 (HLD<<1 | ND), 其中 HLD 是磁头加载时间 (1=2ms, 2=4ms 等)
 * ND 置位表示不使用 DMA (No DMA), 在程序中硬编码成 6 (HLD=6ms, 使用 DMA)。
 */
103
104 extern void floppy_interrupt(void);      // system_call.s 中软驱中断过程标号。
105 extern char tmp_floppy_area[1024];      // boot/head.s 132 行处定义的软盘缓冲区。
106
107 /*
108  * These are global variables, as that's the easiest way to give
109  * information to interrupts. They are the data used for the current
110  * request.
111  */
/*
 * 下面是一些全局变量, 因为这是将信息传给中断程序最简单的方式。它们是
 * 用于当前请求项的数据。
 */
112 static int cur_spec1 = -1;
113 static int cur_rate = -1;
114 static struct floppy_struct * floppy = floppy_type;
115 static unsigned char current_drive = 0;

```



```

116 static unsigned char sector = 0;
117 static unsigned char head = 0;
118 static unsigned char track = 0;
119 static unsigned char seek_track = 0;
120 static unsigned char current_track = 255;          // 当前磁头所在磁道号。
121 static unsigned char command = 0;
    // 软驱已选定标志。在处理一个软驱的请求项之前需要首先选定一个软驱。
122 unsigned char selected = 0;
123 struct task_struct * wait on floppy select = NULL;
124
    //// 取消选定软驱。复位软驱已选定标志 selected。
    // 数字输出寄存器(DOR)的低2位用于指定选择的软驱(0-3对应A-D)。
125 void floppy_deselect(unsigned int nr)
126 {
127     if (nr != (current_DOR & 3))
128         printk("floppy_deselect: drive not selected\n|r");
129     selected = 0;
130     wake up(&wait on floppy select);
131 }
132
133 /*
134  * floppy-change is never called from an interrupt, so we can relax a bit
135  * here, sleep etc. Note that floppy-on tries to set current_DOR to point
136  * to the desired drive, but it will probably not survive the sleep if
137  * several floppies are used at the same time: thus the loop.
138  */
    /*
    * floppy-change()不是从中断程序中调用的,所以这里我们可以轻松一下,睡眠等。
    * 注意 floppy-on()会尝试设置 current_DOR 指向所需的驱动器,但当同时使用几个
    * 软盘时不能睡眠:因此此时只能使用循环方式。
    */
    //// 检测指定软驱中软盘更换情况。如果软盘更换了则返回1,否则返回0。
139 int floppy_change(unsigned int nr)
140 {
141     repeat:
142         floppy on(nr);          // 启动指定软驱 nr (kernel/sched.c, 251)。
    // 如果当前选择的软驱不是指定的软驱 nr, 并且已经选定了其它软驱, 则让当前任务进入可中断
    // 等待状态。
143     while ((current_DOR & 3) != nr && selected)
144         interruptible sleep on(&wait on floppy select);
    // 如果当前没有选择其它软驱或者当前任务被唤醒时, 当前软驱仍然不是指定的软驱 nr, 则循环等待。
145     if ((current_DOR & 3) != nr)
146         goto repeat;
    // 取数字输入寄存器值, 如果最高位(位7)置位, 则表示软盘已更换, 此时关闭马达并退出返回1。
    // 否则关闭马达退出返回0。
147     if (inb(FD DIR) & 0x80) {
148         floppy off(nr);
149         return 1;
150     }
151     floppy off(nr);
152     return 0;
153 }
154

```

```

155 复制内存缓冲块，共 1024 字节。
156 #define copy_buffer(from,to) \
157 __asm__ ("cld ; rep ; movsl" \
158         : "c" (BLOCK_SIZE/4), "S" ((long)(from)), "D" ((long)(to)) \
159         : "cx", "di", "si")
160
161 设置（初始化）软盘 DMA 通道。
162 static void setup_DMA(void)
163 {
164     long addr = (long) CURRENT->buffer; // 当前请求项缓冲区所处内存中位置（地址）。
165
166     cli();
167     // 如果缓冲区处于内存 1M 以上的地方，则将 DMA 缓冲区设在临时缓冲区域(tmp_floppy_area)处。
168     // 因为 8237A 芯片只能在 1M 地址范围内寻址。如果是写盘命令，则需要把数据复制到该临时区域。
169     if (addr >= 0x100000) {
170         addr = (long) tmp_floppy_area;
171         if (command == FD_WRITE)
172             copy_buffer(CURRENT->buffer, tmp_floppy_area);
173     }
174
175     /* mask DMA 2 */ /* 屏蔽 DMA 通道 2 */
176     // 单通道屏蔽寄存器端口为 0x10。位 0-1 指定 DMA 通道(0-3)，位 2: 1 表示屏蔽，0 表示允许请求。
177     immoutb_p(4|2, 10);
178
179     /* output command byte. I don't know why, but everyone (minix, */
180     /* sanches & canton) output this twice, first to 12 then to 11 */
181     /* 输出命令字节。我是不知道为什么，但是每个人(minix, */
182     /* sanches 和 canton) 都输出两次，首先是 12 口，然后是 11 口 */
183     // 下面嵌入汇编代码向 DMA 控制器端口 12 和 11 写方式字（读盘 0x46，写盘 0x4A）。
184     __asm__ ("outb %%a1, $12\n\tjmp 1f\n1:\tjmp 1f\n1:\t\t"
185             "outb %%a1, $11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
186             "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
187
188     /* 8 low bits of addr */ /* 地址低 0-7 位 */
189     // 向 DMA 通道 2 写入基/当前地址寄存器（端口 4）。
190     immoutb_p(addr, 4);
191     addr >>= 8;
192
193     /* bits 8-15 of addr */ /* 地址高 8-15 位 */
194     immoutb_p(addr, 4);
195     addr >>= 8;
196
197     /* bits 16-19 of addr */ /* 地址 16-19 位 */
198     // DMA 只可以在 1M 内存空间内寻址，其高 16-19 位地址需放入页面寄存器(端口 0x81)。
199     immoutb_p(addr, 0x81);
200
201     /* low 8 bits of count-1 (1024-1=0x3ff) */ /* 计数器低 8 位(1024-1 = 0x3ff) */
202     // 向 DMA 通道 2 写入基/当前字节计数器值（端口 5）。
203     immoutb_p(0xff, 5);
204
205     /* high 8 bits of count-1 */ /* 计数器高 8 位 */
206     // 一次共传输 1024 字节（两个扇区）。
207     immoutb_p(3, 5);
208
209     /* activate DMA 2 */ /* 开启 DMA 通道 2 的请求 */
210     // 复位对 DMA 通道 2 的屏蔽，开放 DMA2 请求 DREQ 信号。
211     immoutb_p(0|2, 10);
212     sti();
213 }
214
215 向软驱控制器输出一个字节（命令或参数）。

```

```

// 在向控制器发送一个字节之前，控制器需要处于准备好状态，并且数据传输方向是 CPU→FDC，
// 因此需要首先读取控制器状态信息。这里使用了循环查询方式，以作适当延时。
194 static void output_byte(char byte)
195 {
196     int counter;
197     unsigned char status;
198
199     if (reset)
200         return;
// 循环读取主状态控制器 FD_STATUS(0x3f4) 的状态。如果状态是 STATUS_READY 并且 STATUS_DIR=0
// (CPU→FDC)，则向数据端口输出指定字节。
201     for(counter = 0 ; counter < 10000 ; counter++) {
202         status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
203         if (status == STATUS_READY) {
204             outb(byte, FD_DATA);
205             return;
206         }
207     }
// 如果到循环 1 万次结束还不能发送，则置复位标志，并打印出错信息。
208     reset = 1;
209     printk("Unable to send byte to FDC\n\r");
210 }
211
///// 读取 FDC 执行的结果信息。
// 结果信息最多 7 个字节，存放在 reply_buffer[] 中。返回读入的结果字节数，若返回值=-1
// 表示出错。程序处理方式与上面函数类似。
212 static int result(void)
213 {
214     int i = 0, counter, status;
215
// 若复位标志已置位，则立刻退出。
216     if (reset)
217         return -1;
218     for (counter = 0 ; counter < 10000 ; counter++) {
219         status = inb_p(FD_STATUS) & (STATUS_DIR | STATUS_READY | STATUS_BUSY);
// 如果控制器状态是 READY，表示已经没有数据可取，返回已读取的字节数。
220         if (status == STATUS_READY)
221             return i;
// 如果控制器状态是方向标志置位 (CPU←FDC)、已准备好、忙，表示有数据可读取。于是把控制器
// 中的结果数据读入到应答结果数组中。最多读取 MAX_REPLIES (7) 个字节。
222         if (status == (STATUS_DIR | STATUS_READY | STATUS_BUSY)) {
223             if (i >= MAX_REPLIES)
224                 break;
225             reply_buffer[i++] = inb_p(FD_DATA);
226         }
227     }
228     reset = 1;
229     printk("Getstatus times out\n\r");
230     return -1;
231 }
232
///// 软盘操作出错中断调用函数。由软驱中断处理程序调用。
233 static void bad_flp_intr(void)

```

```

234 {
235     CURRENT->errors++;          // 当前请求项出错次数增 1。
// 如果当前请求项出错次数大于最大允许出错次数，则取消选定当前软驱，并结束该请求项（缓冲
// 区内容没有被更新）。
236     if (CURRENT->errors > MAX_ERRORS) {
237         floppy_deselect(current_drive);
238         end_request(0);
239     }
// 如果当前请求项出错次数大于最大允许出错次数的一半，则置复位标志，需对软驱进行复位操作，
// 然后再试。否则软驱需重新校正一下，再试。
240     if (CURRENT->errors > MAX_ERRORS/2)
241         reset = 1;
242     else
243         recalibrate = 1;
244 }
245
246 /*
247  * Ok, this interrupt is called after a DMA read/write has succeeded,
248  * so we check the results, and copy any buffers.
249  */
/*
* OK，下面的中断处理函数是在 DMA 读/写成功后调用的，这样我们就可以检查执行结果，
* 并复制缓冲区中的数据。
*/
///// 软盘读写操作中中断调用函数。
// 在软驱控制器操作结束后引发的中断处理过程中被调用（Bottom half）。
250 static void rw_interrupt(void)
251 {
// 读取 FDC 执行的结果信息。如果返回结果字节数不等于 7，或者状态字节 0、1 或 2 中存在出错
// 标志，那么，若是写保护就显示出错信息，释放当前驱动器，并结束当前请求项。否则就执行出错
// 计数处理。然后继续执行软盘请求项操作。以下状态的含义参见 fdreg.h 文件。
// ( 0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR )
// ( 0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM, 应该是 0xb7)
// ( 0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM )
252     if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
253         if (ST1 & 0x02) {          // 0x02 = ST1_WP - Write Protected.
254             printk("Drive %d is write protected\n\r", current_drive);
255             floppy_deselect(current_drive);
256             end_request(0);
257         } else
258             bad_flp_intr();
259         do_fd_request();
260         return;
261     }
// 如果当前请求项的缓冲区位于 1M 地址以上，则说明此次软盘读操作的内容还放在临时缓冲区内，
// 需要复制到当前请求项的缓冲区中（因为 DMA 只能在 1M 地址范围寻址）。
262     if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
263         copy_buffer(tmp_floppy_area, CURRENT->buffer);
// 释放当前软驱（放弃不选定），执行当前请求项结束处理：唤醒等待该请求项的进行，唤醒等待空
// 闲请求项的进程（若有的话），从软驱设备请求项链表中删除本请求项。再继续执行其它软盘请求
// 项操作。
264     floppy_deselect(current_drive);
265     end_request(1);

```

```

266     do_fd_request();
267 }
268
269 // 设置 DMA 并输出软盘操作命令和参数 (输出 1 字节命令+ 0~7 字节参数)。
270 inline void setup_rw_floppy(void)
271 {
272     setup_DMA(); // 初始化软盘 DMA 通道。
273     do_floppy = rw_interrupt; // 置软盘中断调用函数指针。
274     output_byte(command); // 发送命令字节。
275     output_byte(head<<2 | current_drive); // 参数 (磁头号+驱动器号)。
276     output_byte(track); // 参数 (磁道号)。
277     output_byte(head); // 参数 (磁头号)。
278     output_byte(sector); // 参数 (起始扇区号)。
279     output_byte(2); // /* sector size = 512 */ // 参数 (字节数 (N=2) 512 字节)。
280     output_byte(floppy->sect); // 参数 (每磁道扇区数)。
281     output_byte(floppy->gap); // 参数 (扇区间隔长度)。
282     output_byte(0xFF); // /* sector size (0xff when n!=0 ?) */
283     // 参数 (当 N=0 时, 扇区定义的字节长度), 这里无用。
284
285 // 若复位标志已置位, 则继续执行下一软盘操作请求。
286 if (reset)
287     do_fd_request();
288 }
289
290 /*
291  * This is the routine called after every seek (or recalibrate) interrupt
292  * from the floppy controller. Note that the "unexpected interrupt" routine
293  * also does a recalibrate, but doesn't come here.
294  */
295
296 // 该子程序是在每次软盘控制器寻道 (或重新校正) 中断后被调用的。注意
297 // "unexpected interrupt" (意外中断) 子程序也会执行重新校正操作, 但不在此地。
298
299 // 寻道处理结束后中断过程中调用的函数。
300 // 首先发送检测中断状态命令, 获得状态信息 ST0 和磁头所在磁道信息。若出错则执行错误计数
301 // 检测处理或取消本次软盘操作请求项。否则根据状态信息设置当前磁道变量, 然后调用函数
302 // setup_rw_floppy() 设置 DMA 并输出软盘读写命令和参数。
303 static void seek_interrupt(void)
304 {
305     /* sense drive status */ // 检测驱动器状态 */
306     // 发送检测中断状态命令, 该命令不带参数。返回结果信息是两个字节: ST0 和磁头当前磁道号。
307     output_byte(FD_SENSEI);
308     // 读取 FDC 执行的结果信息。如果返回结果字节数不等于 2, 或者 ST0 不为寻道结束, 或者磁头所在
309     // 磁道 (ST1) 不等于设定磁道, 则说明发生了错误, 于是执行检测错误计数处理, 然后继续执行软盘
310     // 请求项, 并退出。
311     if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
312         bad_flp_intr();
313         do_fd_request();
314         return;
315     }
316     current_track = ST1; // 设置当前磁道。
317     setup_rw_floppy(); // 设置 DMA 并输出软盘操作命令和参数。
318 }
319
320
321

```

```

304 /*
305  * This routine is called when everything should be correctly set up
306  * for the transfer (ie floppy motor is on and the correct floppy is
307  * selected).
308  */
309 /*
310  * 该函数是在传输操作的所有信息都正确设置好后被调用的（也即软驱马达已开启
311  * 并且已选择了正确的软盘（软驱））。
312  */
313 // 读写数据传输函数。
314
315 static void transfer(void)
316 {
317     // 首先看当前驱动器参数是否就是指定驱动器的参数，若不是就发送设置驱动器参数命令及相应
318     // 参数（参数 1：高 4 位步进速率，低四位磁头卸载时间；参数 2：磁头加载时间）。
319     if (cur_spec1 != floppy->spec1) {
320         cur_spec1 = floppy->spec1;
321         output_byte(FD_SPECIFY); // 发送设置磁盘参数命令。
322         output_byte(cur_spec1); // 发送参数。
323         output_byte(6); // Head load time =6ms, DMA
324     }
325     // 判断当前数据传输速率是否与指定驱动器的一致，若不是就发送指定软驱的速率值到数据传输
326     // 速率控制寄存器(FD_DCR)。
327     if (cur_rate != floppy->rate)
328         outb_p(cur_rate = floppy->rate, FD_DCR);
329     // 若返回结果信息表明出错，则再调用软盘请求函数，并返回。
330     if (reset) {
331         do_fd_request();
332         return;
333     }
334     // 若寻道标志为零（不需要寻道），则设置 DMA 并发送相应读写操作命令和参数，然后返回。
335     if (!seek) {
336         setup_rw_floppy();
337         return;
338     }
339     // 否则执行寻道处理。置软盘中断处理调用函数为寻道中断函数。
340     do_floppy = seek_interrupt;
341     // 如果起始磁道号不等于零则发送磁头寻道命令和参数
342     if (seek_track) {
343         output_byte(FD_SEEK); // 发送磁头寻道命令。
344         output_byte(head<<2 | current_drive); //发送参数：磁头号+当前软驱号。
345         output_byte(seek_track); // 发送参数：磁道号。
346     } else {
347         output_byte(FD_RECALIBRATE); // 发送重新校正命令（磁头归零）。
348         output_byte(head<<2 | current_drive); //发送参数：磁头号+当前软驱号。
349     }
350     // 如果复位标志已置位，则继续执行软盘请求项。
351     if (reset)
352         do_fd_request();
353 }
354
355 /*
356  * Special case - used after a unexpected interrupt (or reset)

```

```

342 */
343 /*
344 * 特殊情况 - 用于意外中断（或复位）处理后。
345 */
346 // 软驱重新校正中断调用函数。
347 // 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志，否则复位重新
348 // 校正标志。然后再次执行软盘请求。
349 static void recal_interrupt(void)
350 {
351     output_byte(FD_SENSEI); // 发送检测中断状态命令。
352     if (result() != 2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
353         reset = 1; // 异常结束，则置复位标志。
354     else // 否则复位重新校正标志。
355         recalibrate = 0;
356     do_fd_request(); // 执行软盘请求项。
357 }
358
359 // 意外软盘中断请求中断调用函数。
360 // 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志，否则置重新
361 // 校正标志。
362 void unexpected_floppy_interrupt(void)
363 {
364     output_byte(FD_SENSEI); // 发送检测中断状态命令。
365     if (result() != 2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
366         reset = 1; // 异常结束，则置复位标志。
367     else // 否则置重新校正标志。
368         recalibrate = 1;
369 }
370
371 // 软盘重新校正处理函数。
372 // 向软盘控制器 FDC 发送重新校正命令和参数，并复位重新校正标志。
373 static void recalibrate_floppy(void)
374 {
375     recalibrate = 0; // 复位重新校正标志。
376     current_track = 0; // 当前磁道号归零。
377     do_floppy = recal_interrupt; // 置软盘中断调用函数指针指向重新校正调用函数。
378     output_byte(FD_RECALIBRATE); // 发送命令：重新校正。
379     output_byte(head << 2 | current_drive); // 发送参数：（磁头号加）当前驱动器号。
380     if (reset) // 如果出错（复位标志被置位）则继续执行软盘请求。
381         do_fd_request();
382 }
383
384 // 软盘控制器 FDC 复位中断调用函数。在软盘中断处理程序中调用。
385 // 首先发送检测中断状态命令（无参数），然后读出返回的结果字节。接着发送设定软驱参数命令
386 // 和相关参数，最后再次调用执行软盘请求。
387 static void reset_interrupt(void)
388 {
389     output_byte(FD_SENSEI); // 发送检测中断状态命令。
390     (void) result(); // 读取命令执行结果字节。
391     output_byte(FD_SPECIFY); // 发送设定软驱参数命令。
392     output_byte(cur_spec1); // 发送参数。
393     output_byte(6); // Head load time = 6ms, DMA
394     do_fd_request(); // 调用执行软盘请求。

```



```

381 }
382
383 /*
384  * reset is done by pulling bit 2 of DOR low for a while.
385  */
386 /* FDC 复位是通过将数字输出寄存器(DOR)位 2 置 0 一会儿实现的 */
387 /* 复位软盘控制器。
388 static void reset_floppy(void)
389 {
390     int i;
391
392     reset = 0; // 复位标志置 0。
393     cur_spec1 = -1;
394     cur_rate = -1;
395     recalibrate = 1; // 重新校正标志置位。
396     printk("Reset-floppy called\n\r"); // 显示执行软盘复位操作信息。
397     cli(); // 关中断。
398     do_floppy = reset_interrupt; // 设置在软盘中断处理程序中调用的函数。
399     outb_p(current_DOR & ~0x04, FD_DOR); // 对软盘控制器 FDC 执行复位操作。
400     for (i=0 ; i<100 ; i++) // 空操作, 延迟。
401         __asm__("nop");
402     outb(current_DOR, FD_DOR); // 再启动软盘控制器。
403     sti(); // 开中断。
404 }
405
406 /* 软驱启动定时中断调用函数。
407 // 首先检查数字输出寄存器(DOR), 使其选择当前指定的驱动器。然后调用执行软盘读写传输
408 // 函数 transfer()。
409 static void floppy_on_interrupt(void)
410 {
411     /* We cannot do a floppy-select, as that might sleep. We just force it */
412     /* 我们不能任意设置选择的软驱, 因为这样做可能会引起进程睡眠。我们只是迫使它自己选择 */
413     selected = 1; // 置已选定当前驱动器标志。
414     // 如果当前驱动器号与数字输出寄存器 DOR 中的不同, 则需要重新设置 DOR 为当前驱动器
415     current_drive。
416     // 定时延迟 2 个滴答时间, 然后调用软盘读写传输函数 transfer()。否则直接调用软盘读写传输函数。
417     if (current_drive != (current_DOR & 3)) {
418         current_DOR &= 0xFC;
419         current_DOR |= current_drive;
420         outb(current_DOR, FD_DOR); // 向数字输出寄存器输出当前 DOR。
421         add_timer(2, &transfer); // 添加定时器并执行传输函数。
422     } else
423         transfer(); // 执行软盘读写传输函数。
424 }
425
426 /* 软盘读写请求项处理函数。
427 //
428 void do_fd_request(void)
429 {
430     unsigned int block;
431
432     seek = 0;
433     // 如果复位标志已置位, 则执行软盘复位操作, 并返回。

```

```

422     if (reset) {
423         reset_floppy();
424         return;
425     }
426     // 如果重新校正标志已置位, 则执行软盘重新校正操作, 并返回。
427     if (recalibrate) {
428         recalibrate_floppy();
429         return;
430     }
431     // 检测请求项的合法性, 若已没有请求项则退出(参见 blk.h, 127)。
432     INIT_REQUEST;
433     // 将请求项结构中软盘设备号中的软盘类型(MINOR(CURRENT->dev)>>2)作为索引取得软盘参数块。
434     floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;
435     // 如果当前驱动器不是请求项中指定的驱动器, 则置标志 seek, 表示需要进行寻道操作。
436     // 然后置请求项设备为当前驱动器。
437     if (current_drive != CURRENT_DEV)
438         seek = 1;
439     current_drive = CURRENT_DEV;
440     // 设置读写起始扇区。因为每次读写是以块为单位(1块为2个扇区), 所以起始扇区需要起码比
441     // 磁盘总扇区数小2个扇区。否则结束该次软盘请求项, 执行下一个请求项。
442     block = CURRENT->sector; // 取当前软盘请求项中起始扇区号→block。
443     if (block+2 > floppy->size) { // 如果 block+2 大于磁盘扇区总数, 则
444         end_request(0); // 结束本次软盘请求项。
445         goto repeat;
446     }
447     // 求对在磁道上的扇区号, 磁头号, 磁道号, 搜寻磁道号(对于软驱读不同格式的盘)。
448     sector = block % floppy->sect; // 起始扇区对每磁道扇区数取模, 得磁道上扇区号。
449     block /= floppy->sect; // 起始扇区对每磁道扇区数取整, 得起始磁道数。
450     head = block % floppy->head; // 起始磁道数对磁头数取模, 得操作的磁头号。
451     track = block / floppy->head; // 起始磁道数对磁头数取整, 得操作的磁道号。
452     seek_track = track << floppy->stretch; // 相应于驱动器中盘类型进行调整, 得寻道号。
453     // 如果寻道号与当前磁头所在磁道不同, 则置需要寻道标志 seek。
454     if (seek_track != current_track)
455         seek = 1;
456     sector++; // 磁盘上实际扇区计数是从1算起。
457     if (CURRENT->cmd == READ) // 如果请求项中是读操作, 则置软盘读命令码。
458         command = FD_READ;
459     else if (CURRENT->cmd == WRITE) // 如果请求项中是写操作, 则置软盘写命令码。
460         command = FD_WRITE;
461     else
462         panic("do_fd_request: unknown command");
463     // 向系统添加定时器。为了能对软驱进行读写操作, 需要首先启动驱动器马达并达到正常运转速度。
464     // 这需要一定的时间。因此这里利用 ticks_to_floppy_on() 计算启动延时时间, 设定一个定时器。
465     // 当定时时间到时, 就调用函数 floppy_on_interrupt()。
466     add_timer(ticks_to_floppy_on(current_drive), &floppy_on_interrupt);
467 }
468
469 // 软盘系统初始化。
470 // 设置软盘块设备的请求处理函数(do_fd_request()), 并设置软盘中断门(int 0x26, 对应硬件
471 // 中断请求信号 IRQ6), 然后取消对该中断信号的屏蔽, 允许软盘控制器 FDC 发送中断请求信号。
472 void floppy_init(void)
473 {
474     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // = do_fd_request()。

```

```

460     set_trap_gate(0x26, &floppy_interrupt); //设置软盘中断门 int 0x26(38)。
461     outb(inb_p(0x21)&~0x40, 0x21);          // 复位软盘的中断请求屏蔽位，允许
                                              // 软盘控制器发送中断请求信号。
462 }
463

```

6.8.3 其它信息

6.8.3.1 软盘驱动器的设备号

在 Linux 中，软驱的主设备号是 2，次设备号 = TYPE*4 + DRIVE，其中 DRIVE 为 0-3，分别对应软驱 A、B、C 或 D；TYPE 是软驱的类型，2 表示 1.2M 软驱，7 表示 1.44M 软驱，也即 floppy.c 中 85 行定义的软盘类型（floppy_type[]）数组的索引值，见表 6-11 所示。

表 6-11 软盘驱动器类型

类型	说明
0	不用。
1	360KB PC 软驱。
2	1.2MB AT 软驱。
3	360kB 在 720kB 驱动器中使用。
4	3.5" 720kB 软盘。
5	360kB 在 1.2MB 驱动器中使用。
6	720kB 在 1.2MB 驱动器中使用。
7	1.44MB 软驱。

例如，因为 $7*4 + 0 = 28$ ，所以 /dev/PS0 (2,28)指的是 1.44M A 驱动器,其设备号是 0x021c。

同理 /dev/at0 (2,8)指的是 1.2M A 驱动器，其设备号是 0x0208。

6.8.3.2 软盘控制器

对软盘控制器的编程比较烦琐。在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有表 6-12 中的一些端口。

表 6-12 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器（DOR）(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器(STATUS)
0x3f5	读/写	FDC 数据寄存器(DATA)
0x3f7	只读	数字输入寄存器（DIR）
	只写	磁盘控制寄存器(DCR)(传输率控制)

数字输出端口 DOR（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。该寄存器各比特位的含义见表 6-13 所示。

表 6-13 数字输出寄存器定义

位	名称	说明
---	----	----

7	MOT_EN3	启动软驱 D 马达：1-启动；0-关闭。
6	MOT_EN2	启动软驱 C 马达：1-启动；0-关闭。
5	MOT_EN1	启动软驱 B 马达：1-启动；0-关闭。
4	MOT_EN0	启动软驱 A 马达：1-启动；0-关闭。
3	DMA_INT	允许 DMA 和中断请求；0-禁止 DMA 和中断请求。
2	RESET	允许软盘控制器 FDC 工作。0-复位 FDC。
1	DRV_SEL1	00-11 用于选择软盘驱动器 A-D。
0	DRV_SEL0	

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。见表 6-14 所示。

表 6-14 FDC 主状态控制器 MSR 定义

位	名称	说明
7	RQM	数据口就绪：控制器 FDC 数据寄存器已准备就绪。
6	DIO	传输方向：1- FDC→CPU；0- CPU→FDC
5	NDM	非 DMA 方式：1- 非 DMA 方式；0- DMA 方式
4	CB	控制器忙：FDC 正处于命令执行忙碌状态
3	DDB	软驱 D 忙
2	DCB	软驱 C 忙
1	DBB	软驱 B 忙
0	DAB	软驱 A 忙

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

数据输入寄存器（DIR）只有位 7（D7）对软盘有效，用来表示盘片更换状态。其余七位用于硬盘控制器接口。

磁盘控制寄存器(DCR)用于选择盘片在不同类型驱动器上使用的数据传输率。仅使用低 2 位(D1D0)，00 - 500kbps，01 - 300kbps，10 - 250kbps。

Linux 0.11 内核中，驱动程序与软驱中磁盘之间的数据传输是通过 DMA 控制器实现的。在进行读写操作之前，需要首先初始化 DMA 控制器，并对软驱控制器进行编程。对于 386 兼容 PC，软驱控制器使用硬件中断 IR6（对应中断描述符 0x26），并采用 DMA 控制器的通道 2。有关 DMA 控制处理的内容见后面小节。

6.8.3.3 软盘控制器命令

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0-8 字节的参数。

执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。

结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0~7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

由于 Linux 0.11 的软盘驱动程序中只使用其中 6 条命令，因此这里仅对这些用到的命令进行描述。

1. 重新校正命令 (FD_RECALIBRATE)

该命令用来让磁头退回到 0 磁道。通常用于在软盘操作出错时对磁头重新校正定位。其命令码是 0x07，参数是指定的驱动器号 (0—3)。

该命令无结果阶段，程序需要通过执行“检测中断状态”来获取该命令的执行结果。见表 6-15 所示。

表 6-15 重新校正命令 (FD_RECALIBRATE)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	0	1	1	1	重新校正命令码: 0x07
	1	0	0	0	0	0	0	US1	US2	驱动器号
执行										磁头移动到 0 磁道
结果		无。								需使用命令获取执行结果。

2. 磁头寻道命令 (FD_SEEK)

该命令让选中驱动器的磁头移动到指定磁道上。第 1 个参数指定驱动器号和磁头号，位 0-1 是驱动器号，位 2 是磁头号，其它比特位无用。第 2 个参数指定磁道号。

该命令也无结果阶段，程序需要通过执行“检测中断状态”来获取该命令的执行结果。见表 6-16 所示。

表 6-16 磁头寻道命令 (FD_SEEK)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	1	1	1	1	磁头寻道命令码: 0x0F
	1	0	0	0	0	0	HD	US1	US2	磁头号、驱动器号。
	2	C								磁道号。
执行										磁头移动到指定磁道上。
结果		无。								需使用命令获取执行结果。

3. 读扇区数据命令 (FD_READ)

该命令用于从磁盘上读取指定位置开始的扇区，经 DMA 控制传输到系统内存中。每当一个扇区读完，参数 4 (R) 就自动加 1，以继续读取下一个扇区，直到 DMA 控制器把传输计数终止信号发送给软盘控制器。该命令通常是在磁头寻道命令执行后磁头已经位于指定磁道后开始。见表 6-17 所示。

返回结果中，磁道号 C 和扇区号 R 是当前磁头所处位置。因为在读完一个扇区后起始扇区号 R 自动增 1，因此结果中的 R 值是下一个未读扇区号。若正好读完一个磁道上最后一个扇区 (即 EOT)，则磁道号也会增 1，并且 R 值复位成 1。

表 6-17 读扇区数据命令 (FD_READ)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	MT	MF	SK	0	0	1	1	0	读命令码: 0xE6 (MT=MF=SK=1)
	1	0	0	0	0	0	0	US1	US2	驱动器号。
	2	C								磁道号
	3	H								磁头号
	4	R								起始扇区号
	5	N								扇区字节数
	6	EOT								磁道上最大扇区号
	7	GPL								扇区之间间隔长度 (3)
	8	DTL								N=0 时, 指定扇区字节数
执行										数据从磁盘传送到系统
结果	1	ST0								状态字节 0
	2	ST1								状态字节 1
	3	ST2								状态字节 2
	4	C								磁道号
	5	H								磁头号
	6	R								扇区号
	7	N								扇区字节数

其中 MT、MF 和 SK 的含义分别为:

MT 表示多磁道操作。MT=1 表示允许在同一磁道上两个磁头连续操作。

MF 表示记录方式。MF=1 表示选用 MFM 记录方式, 否则是 FM 记录方式。

SK 表示是否跳过有删除标志的扇区。SK=1 表示跳过。

返回的个状态字节 ST0、ST1 和 ST2 的含义分别见表 6-18、表 6-19 和表 6-20 所示。

表 6-18 状态字节 0 (ST0)

位	名称	说明
7	ST0_INTR	中断原因。00 – 命令正常结束; 01 – 命令异常结束; 10 – 命令无效; 11 – 软盘驱动器状态改变。
6		
5	ST0_SE	寻道操作或重新校正操作结束。(Seek End)
4	ST0_ECE	设备检查出错 (零磁道校正出错)。(Equip. Check Error)
3	ST0_NR	软驱未就绪。(Not Ready)
2	ST0_HA	磁头地址。中断时磁头号。(Head Address)
1	ST0_DS	驱动器选择号 (发生中断时驱动器号)。(Drive Select) 00 – 11 分别对应驱动器 0—3。
0		

表 6-19 状态字节 1 (ST1)

位	名称	说明
7	ST1_EOC	访问超过磁道上最大扇区号 EOT。(End of Cylinder)
6		未使用 (0)。
5	ST1_CRC	CRC 校验出错。

4	ST1_OR	数据传输超时，DMA 控制器故障。(Over Run)
3		未使用 (0)。
2	ST1_ND	未找到指定的扇区。(No Data - unreadable)
1	ST1_WP	写保护。(Write Protect)
0	ST1_MAM	未找到扇区地址标志 ID AM。(Missing Address Mask)

表 6-20 状态字节 2 (ST2)

位	名称	说明
7		未使用 (0)。
6	ST2_CM	SK=0 时，读数据遇到删除标志。(Control Mark = deleted)
5	ST2_CRC	扇区数据场 CRC 校验出错。
4	ST2_WC	扇区 ID 信息的磁道号 C 不符。(Wrong Cylinder)
3	ST2_SEH	检索(扫描)条件满足要求。(Scan Equal Hit)
2	ST2_SNS	检索条件不满足要求。(Scan Not Satisfied)
1	ST2_BC	扇区 ID 信息的磁道号 C=0xFF，磁道坏。(Bad Cylinder)
0	ST2_MAM	未找到扇区数据标志 DATA AM。(Missing Address Mask)

4. 写扇区数据命令 (FD_WRITE)

该命令用于将内存中的数据写到磁盘上。在 DMA 传输方式下，软驱控制器把内存中的数据串行地写到磁盘指定扇区中。每写完一个扇区，起始扇区号自动增 1，并继续写下一个扇区，直到软驱控制器收到 DMA 控制器的计数终止信号。见表 6-21 所示，其中缩写名称的含义与读命令中的相同。

表 6-21 写扇区数据命令 (FD_WRITE)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	MT	MF	0	0	0	1	0	1	写数据命令码：0xC5 (MT=MF=1)
	1	0	0	0	0	0	0	US1	US2	驱动器号。
	2	C								磁道号
	3	H								磁头号
	4	R								起始扇区号
	5	N								扇区字节数
	6	EOT								磁道上最大扇区号
	7	GPL								扇区之间间隔长度 (3)
	8	DTL								N=0 时，指定扇区字节数
执行										数据从系统传送到磁盘
结果	1	ST0								状态字节 0
	2	ST1								状态字节 1
	3	ST2								状态字节 2
	4	C								磁道号
	5	H								磁头号
	6	R								扇区号
	7	N								扇区字节数

5. 检测中断状态命令 (FD_SENSEI)

发送该命令后软驱控制器会立刻返回常规结果 1 和 2（即状态 ST0 和磁头所处磁道号 PCN）。它们是控制器执行上一条命令后的状态。通常在一个命令执行结束后会向 CPU 发出中断信号。对于读写扇区、读写磁道、读写删除标志、读标识场、格式化和扫描等命令以及非 DMA 传输方式下的命令引起的中断，可以直接根据主状态寄存器的标志知道中断原因。而对于驱动器就绪信号发生变化、寻道和重新校正（磁头回零道）而引起的中断，由于没有返回结果，就需要利用本命令来读取控制器执行命令后的状态信息。见表 6-22 所示。

表 6-22 检测中断状态命令 (FD_SENSEI)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	1	0	0	0	检测中断状态命令码: 0x08
执行										
结果	1	ST0								状态字节 0
	2	C								磁头所在磁道号

6. 设定驱动器参数命令 (FD_SPECIFY)

该命令用于设定软盘控制器内部的三个定时器初始值和选择传输方式，即把驱动器马达步进速率（STR）、磁头加载/卸载（HLT/HUT）时间和是否采用 DMA 方式来传输数据的信息送入软驱控制器。见表 6-23 所示。

表 6-23 设定驱动器参数命令 (FD_SPECIFY)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	0	0	1	1	设定参数命令码: 0x03
	1	SRT (单位 2ms)				HUT (单位 32ms)				马达步进速率、磁头卸载时间
	2	HLT (单位 4ms)							ND	磁头加载时间、非 DMA 方式
执行										设置控制器，不发生中断
结果		无								无

6.8.3.4 软盘控制器编程方法

在 PC 机中，软盘控制器一般采用与 NEC PD765 或 Intel 8287A 兼容的芯片，例如 Intel 的 82078。由于软盘的驱动程序比较复杂，因此下面对这类芯片构成的软盘控制器的编程方法进行较为详细的介绍。

典型的磁盘操作不仅仅包括发送命令和等待控制器返回结果，的软盘驱动器的控制是一种低级操作，它需要程序在不同阶段对其执行状况进行干涉。

◆命令与结果阶段的交互

在上述磁盘操作命令或参数发送到软盘控制器之前，必须首先查询控制器的主状态寄存器（MSR），以获知驱动器的就绪状态和数据传输方向。软盘驱动程序中使用了一个 output_byte(byte)函数来专门实现该操作。该函数的等效框图见图 6-6 所示。

该函数一直循环到主状态寄存器的数据口就绪标志 RQM 为 1，并且方向标志 DIO 是 0(CPU→FDC)，此时控制器就已准备好接受命令和参数字节。循环语句起超时计数功能，以应付控制器没有响应的情况。本驱动程序中把循环次数设置成了 10000 次。对这个循环次数的选择需要仔细，以避免程序作出不正确的超时判断。在 Linux 内核版本 0.1x 至 0.9x 中就经常会碰到需要调整这个循环次数的问题，因为当时人们所使用的 PC 机运行速度差别较大（16MHz -- 40MHz），因此循环所产生的实际延时也有很大的区别。这可以参见早期 Linux 的邮件列表中的许多文章。为了彻底解决这个问题，最好能使用系统硬件时钟来

产生固定频率的延时值。

对于读取控制器的结果字节串的结果阶段，也需要采取与发送命令相同的操作方法，只是此时数据传输方向标志要求是置位状态（FDC→CPU）。本程序中对应的函数是 `result()`。该函数把读取的结果状态字节存放到了 `reply_buffer[]` 字节数组中。

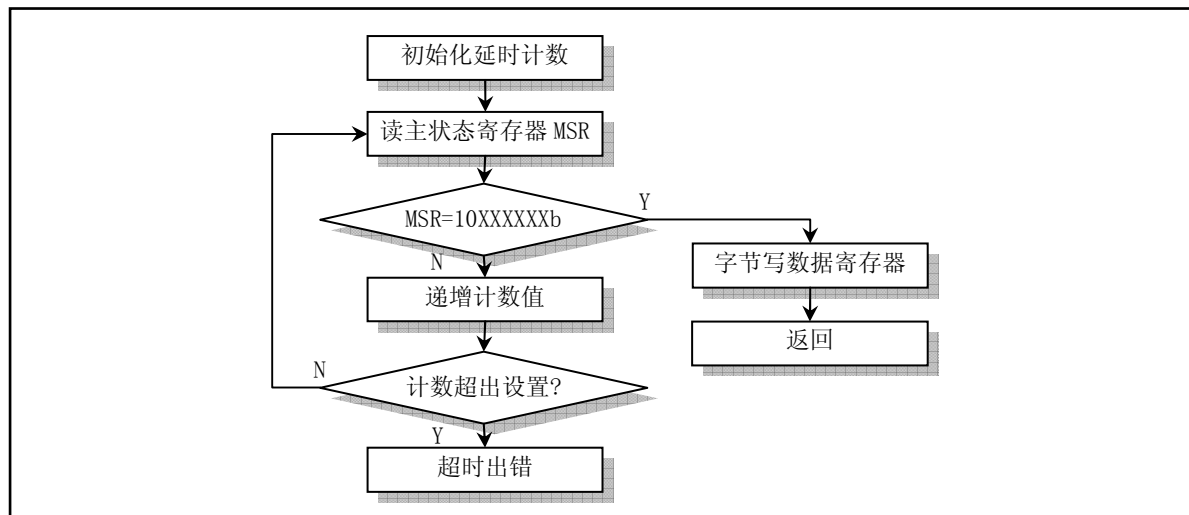


图 6-6 向软盘控制器发送命令或参数字节

◆软盘控制器初始化

对软盘控制器的初始化操作包括在控制器复位后对驱动器进行适当的参数配置。控制器复位操作是指对数字输出寄存器 DOR 的位 2（启动 FDC 标志）置 0 然后再置 1。在机器复位之后，“指定驱动器参数”命令 SPECIFY 所设置的值就不再有效，需要重新建立。在 `floppy.c` 程序中，复位操作在函数 `reset_floppy()` 和中断处理 C 函数 `reset_interrupt()` 中。前一个函数用于修改 DOR 寄存器的位 2，让控制器复位，后一个函数用于在控制器复位后使用 SPECIFY 命令重新建立控制器中的驱动器参数。在数据传输准备阶段，若判断出与实际的磁盘规格不同，还在传输函数 `transfer()` 开始处对其另行进行重新设置。

在控制器复位后，还应该向数字控制寄存器 DCR 发送指定的传输速率值，以重新初始化数据传输速率。如果机器执行了复位操作（例如热启动），则数据传输速率会变成默认值 250Kpbs。但通过数字输出寄存器 DOR 向控制器发出的复位操作并不会影响设置的数据传输速率。

◆驱动器重新校正和磁头寻道

驱动器重新校正（FD_RECALIBRATE）和磁头寻道（FD_SEEK）是两个磁头定位命令。重新校正命令让磁头移动到零磁道，而磁头寻道命令则让磁头移动到指定的磁道上。这两个磁头定位命令与典型的读/写命令不同，因为它们没有结果阶段。一旦发出这两个命令之一，控制器将立刻会在主状态寄存器（MSR）返回就绪状态，并以后台形式执行磁头定位操作。当定位操作完成后，控制器就会产生中断以请求服务。此时就应该发送一个“检测中断状态”命令，以结束中断和读取定位操作后的状态。由于驱动器和马达启动信号是直接由数字输出寄存器（DOR）控制的，因此，如果驱动器或马达还没有启动，那么写 DOR 的操作必须在发出定位命令之前进行。流程图见图 6-7 所示。

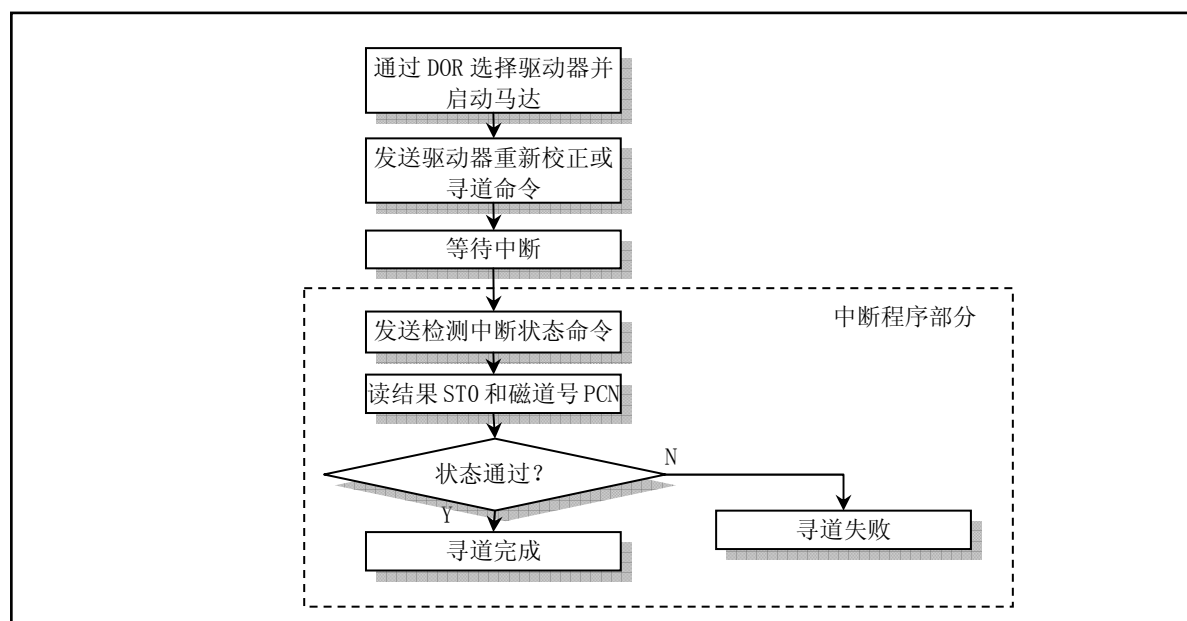


图 6-7 重新校正和寻道操作

◆数据读/写操作

数据读或写操作需要分几步来完成。首先驱动器马达需要开启，并把磁头定位到正确的磁道上，然后初始化 DMA 控制器，最后发送数据读或写命令。另外，还需要定出发生错误时的处理方案。典型的操作流程见图 6-8 所示。

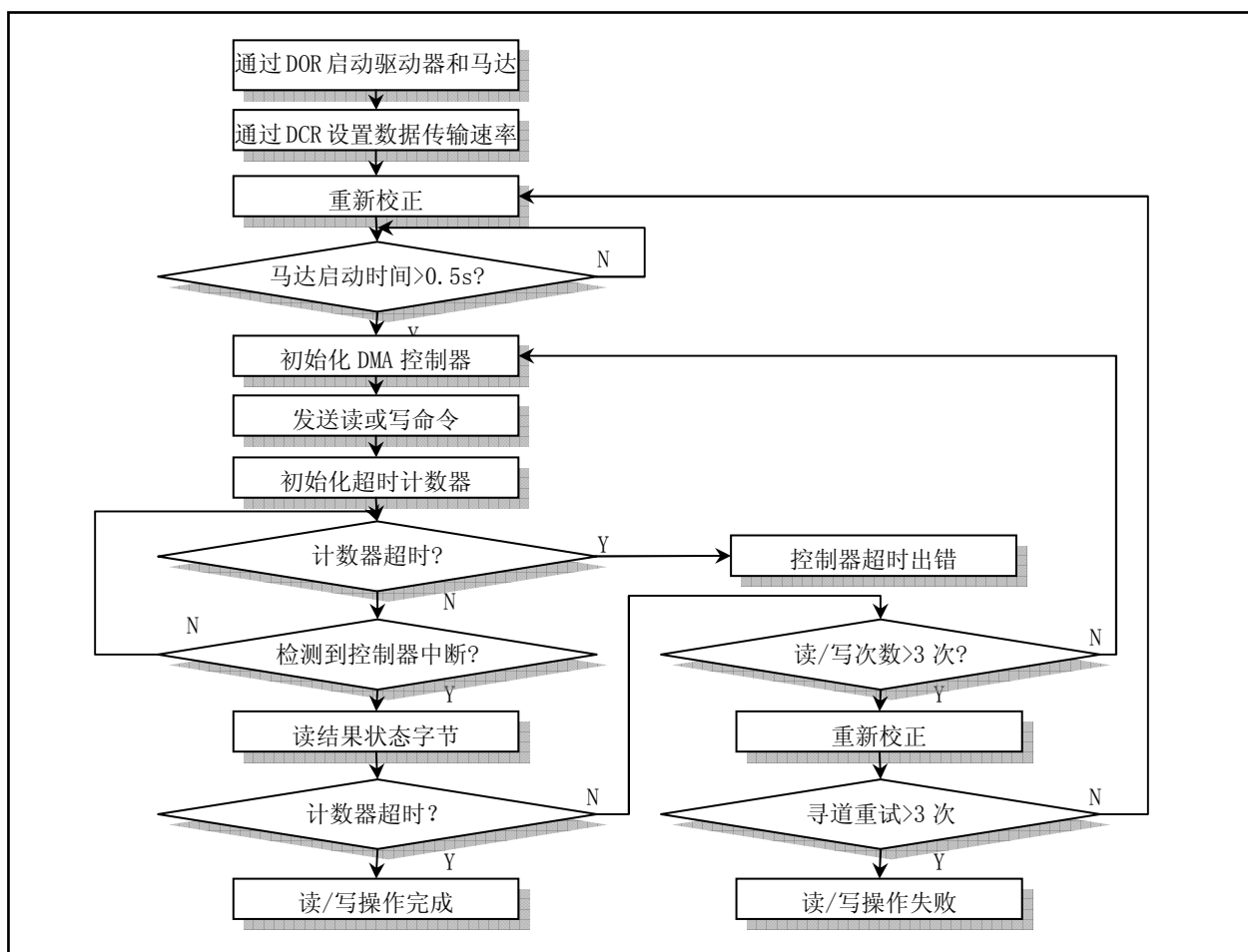


图 6-8 数据读/写操作流程图

在对磁盘进行数据传输之前，磁盘驱动器的马达必须首先达到正常的运转速度。对于大多数 3 $\frac{1}{2}$ 英寸软驱来讲，这段启动时间大约需要 300ms，而 5 $\frac{1}{4}$ 英寸的软驱则需要大约 500ms。在 floppy.c 程序中将这个启动延迟时间设置成了 500ms。

在马达启动后，就需要使用数字控制寄存器 DCR 设置与当前磁盘介质匹配的数据传输率。

如果隐式寻道方式没有开启，接下来就需要发送寻道命令 `FD_SEEK`，把磁头定位到正确的磁道上。在寻道操作结束后，磁头还需要花费一段到位（加载）时间。对于大多数驱动器，这段延迟时间起码需要 15ms。当使用了隐式寻道方式，那么就可以使用“指定驱动器参数”命令指定的磁头加载时间（HLT）来确定最小磁头到位时间。例如在数据传输速率为 500Kbps 的情况下，若 `HLT=8`，则有效磁头到位时间是 16ms。当然，如果磁头已经在正确的磁道上到位了，也就无须确保这个到位时间了。

然后对 DMA 控制器进行初始化操作，读写命令也随即执行。通常，在数据传输完成后，DMA 控制器会发出终止计数（TC）信号，此时软盘控制器就会完成当前数据传输并发出中断请求信号，表明操作已到达结果阶段。如果在操作过程中出现错误或者最后一个扇区号等于磁道最后一个扇区（EOT），那么软盘控制器也会马上进入结果阶段。

根据上面流程图，如果在读取结果状态字节后发现错误，则会通过重新初始化 DMA 控制器，再尝试重新开始执行数据读或写操作命令。持续的错误通常表明寻道操作并没有让磁头到达指定的磁道，此时应该多次重复对磁头执行重新校准，并再次执行寻道操作。若此后还是出错，则最终控制器就会向驱动程序报告读写操作失败。

◆ 磁盘格式化操作

Linux 0.11 内核中虽然没有实现对软盘的格式化操作，但作为参考，这里还是对磁盘格式化操作进行简单说明。磁盘格式化操作过程包括把磁头定位到每个磁道上，并创建一个用于组成数据字段（场⁴）的固定格式字段。

在马达已启动并且设置了正确的数据传输率之后，磁头会返回零磁道。此时磁盘需要在 500ms 延迟时间内到达正常和稳定的运转速度。

在格式化操作期间磁盘上建立的标识字段（ID 字段）是在执行阶段由 DMA 控制器提供。DMA 控制器被初始化成为每个扇区标识场提供磁道（C）、磁头（H）、扇区号（R）和扇区字节数的值。例如，对于每个磁道具有 9 个扇区的磁盘，每个扇区大小是 2（512 字节），若是用磁头 1 格式化磁道 7，那么 DMA 控制器应该被编程为传输 36 个字节的数据（9 扇区 x 每扇区 4 个字节），数据字段应该是：7,1,1,2, 7,1,2,2, 7,1,3,2, ..., 7,1,9,2。因为在格式化命令执行期间，软盘控制器提供的数据会被直接作为标识字段记录在磁盘上，数据的内容可以是任意的。因此有些人就利用这个功能来防止保护磁盘复制。

在一个磁道上的每个磁头都已经执行了格式化操作以后，就需要执行寻道操作让磁头前移到下一磁道上，并重复执行格式化操作。因为“格式化磁道”命令不含有隐式的寻道操作，所以必须使用寻道命令 SEEK。同样，前面所讨论的磁头到位时间也需要在每次寻道后设置。

6.8.3.5 DMA 控制器编程

DMA（Direct Memory Access）是“直接存储器访问”的缩写。DMA 控制器的主要功能是通过让外部设备直接与内存传输数据来增强系统的性能。通常它由机器上的 Intel 8237 芯片或其兼容芯片实现。通过对 DMA 控制器进行编程，外设与内存之间的数据传输能在不受 CPU 控制的条件下进行。因此在数据传输期间，CPU 可以做其它事。DMA 控制器传输数据的工作过程如下：

1. 初始化 DMA 控制器。

程序通过 DMA 控制器端口对其进行初始化操作。该操作包括：① 向 DMA 控制器发送控制命令；② 传输的内存起始地址；③ 数据长度。发送的命令指明传输使用的 DMA 通道、是内存传输到外设（写）还是外设数据传输到内存、是单字节传输还是批量（块）传输。对于 PC 机，软盘控制器被指定使用 DMA 通道 2。在 Linux 0.11 内核中，软盘驱动程序采用的是单字节传输模式。由于 Intel 8237 芯片只有 16 根地址引脚（其中 8 根与数据线合用），因此只能寻址 64KB 的内存空间。为了能让它访问 1MB 的地址空间，DMA 控制器采用了一个页面寄存器把 1MB 内存分成了 16 个页面来操作，见表 6-24 所示。因此传输的内存起始地址需要转换成所处的 DMA 页面值和页面中的偏移地址。每次传输的数据长度也不能超过 64KB。

表 6-24 DMA 页面对应的内存地址范围

DMA 页面	地址范围（64KB）
0x0	0x00000 - 0x0FFFF
0x1	0x10000 - 0x1FFFF
0x2	0x20000 - 0x2FFFF
0x3	0x30000 - 0x3FFFF
0x4	0x40000 - 0x4FFFF
0x5	0x50000 - 0x5FFFF
0x6	0x60000 - 0x6FFFF
0x7	0x70000 - 0x7FFFF
0x8	0x80000 - 0x8FFFF
0x9	0x90000 - 0x9FFFF

⁴ 关于磁盘格式的说明资料，以前均把 filed 翻译成场。其实对于程序员来讲，翻译成字段或域或许更顺耳一些。☺

0xA	0x20000 - 0xAFFFF
0xB	0x20000 - 0xBFFFF
0xC	0x20000 - 0xCFFFF
0xD	0x00000 - 0xDFFFF
0xE	0x00000 - 0xEFFFF
0xF	0x10000 - 0xFFFFF

2. 数据传输

在初始化完成之后，对 DMA 控制器的屏蔽寄存器进行设置，开启 DMA 通道 2，从而 DMA 控制器开始进行数据的传输。

3. 传输结束

当所需传输的数据全部传输完成，DMA 控制器就会产生“操作完成”（EOP）信号发送到软盘控制器。此时软盘控制器即可执行结束操作：关闭驱动器马达并向 CPU 发送中断请求信号。

在 PC/AT 机中，DMA 控制器有 8 个独立的通道可使用，其中后 4 个通道是 16 位的。软盘控制器被指定使用 DMA 通道 2。在使用一个通道之前必须首先对其设置。这牵涉到对三个端口的操作，分别是：页面寄存器端口、（偏移）地址寄存器端口和数据计数寄存器端口。由于 DMA 寄存器是 8 位的，而地址和计数值是 16 位的，因此各自需要发送两次。首先发送低字节，然后发送高字节。每个通道对应的端口地址见表 6-25 所示。

表 6-25 DMA 各通道使用的页面、地址和计数寄存器端口

DMA 通道	页面寄存器	地址寄存器	计数寄存器
0	0x87	0x00	0x01
1	0x83	0x02	0x03
2	0x81	0x04	0x05
3	0x82	0x06	0x07
4	0x8F	0xC0	0xC2
5	0x8B	0xC4	0xC6
6	0x89	0xC8	0xCA
7	0x8A	0xCC	0xCE

对于通常的 DMA 应用，有 4 个常用寄存器用于控制 DMA 控制器的状态。它们是命令寄存器、请求寄存器、单屏蔽寄存器和清除字节指针触发器。见表 6-26 所示。Linux 0.11 内核使用了表中带阴影的 3 个寄存器端口（0x0A, 0x0B, 0x0C）。

表 6-26 DMA 编程常用的 DMA 寄存器

名称	端口地址	
	（通道 0-3）	（通道 4-7）
命令寄存器	0x08	0xD0
请求寄存器	0x09	0xD2
单屏蔽寄存器	0x0A	0xD4
方式寄存器	0x0B	0xD6
清除先后触发器	0x0C	0xD8

命令寄存器用于规定 DMA 控制器芯片的操作要求，设定 DMA 控制器的总体状态。通常它在开机初始化之后就无须变动。在 Linux 0.11 内核中，软盘驱动程序就直接使用了开机后 ROM BIOS 的设置值。作为参考，这里列出命令寄存器各比特位的含义，见表 6-27 所示。（在读该端口时，所得内容是 DMA 控制器状态寄存器的信息）

表 6-27 DMA 命令寄存器格式

位	说明
7	DMA 响应外设信号 DACK：0-DACK 低电平有效；1-DACK 高电平有效。
6	外设请求 DMA 信号 DREQ：0-DREQ 低电平有效；1-DREQ 高电平有效。
5	写方式选择：0-选择迟后写；1-选择扩展写；X-若位 3=1。
4	DMA 通道优先方式：0-固定优先；1-轮转优先。
3	DMA 周期选择：0-普通定时周期（5）；1-压缩定时周期（3）；X-若位 0=1。
2	开启 DMA 控制器：0-允许控制器工作；1-禁止控制器工作。
1	通道 0 地址保持：0-禁止通道 0 地址保持；1-允许通道 0 地址保持；X-若位 0=0。
0	内存传输方式：0-禁止内存至内存传输方式；1-允许内存至内存传输方式。

请求寄存器用于记录外设对通道的请求服务信号 DREQ。每个通道对应一位。当 DREQ 有效时对应位置 1，当 DMA 控制器对其作出响应时会对该位置 0。如果不使用 DMA 的请求信号 DREQ 引脚，那么也可以通过编程直接设置相应通道的请求位来请求 DMA 控制器的服务。在 PC 机中，软盘控制器与 DMA 控制器的通道 2 有直接的请求信号 DREQ 连接，因此 Linux 内核中也无须对该寄存器进行操作。作为参考，这里还是列出请求通道服务的字节格式，见表 6-28 所示。

表 6-28 DMA 请求寄存器各比特位的含义

位	说明
7-3	不用。
2	屏蔽标志。0-请求位置位；1-请求位复位（置 0）。
1	通道选择。00-11 分别选择通道 0-3。
0	

单屏蔽寄存器的端口是 0x0A（对于 16 位通道则是 0xD4）。一个通道被屏蔽，是指使用该通道的外设发出的 DMA 请求信号 DREQ 得不到 DMA 控制器的响应，因此也就无法让 DMA 控制器操作该通道。该寄存器各比特位的含义见表 6-29 所示。

表 6-29 DMA 单屏蔽寄存器各比特位的含义

位	说明
7-3	不用。
2	屏蔽标志。1-屏蔽选择的通道；0-开启选择的通道。
1	通道选择。00-11 分别选择通道 0-3。
0	

方式寄存器用于指定某个 DMA 通道的操作方式。在 Linux 0.11 内核中，使用了其中读（0x46）和写（0x4A）两种方式。该寄存器各位的含义见表 6-30 所示。

表 6 - 30 DMA 方式寄存器各比特位的含义

位	说明
7	选择传输方式。
6	00-请求模式；01-单字节模式；10-块字节模式；11-接连模式。
5	地址增减方式。0-地址递减；1-地址递增。
4	自动预置（初始化）。0-自动预置；1-非自动预置。
3	传输类型。
2	00-DMA 校验；01-DMA 写传输；10-DMA 读传输。11-无效。
1	通道选择。00-11 分别选择通道 0-3。
0	

通道的地址和计数寄存器可以读写 16 位的数据。清除先后触发器端口 0x0C 就是用于在读/写 DMA 控制器中地址或计数信息之前把字节先后触发器初始化为默认状态。当字节触发器为 0 时，则访问低字节；当字节触发器为 1 时，则访问高字节。每访问一次，该触发器就变化一次。写 0x0C 端口就可以将触发器置成 0 状态。

在使用 DMA 控制器时，通常需要按照一定的步骤来进行，下面以软盘驱动程序使用 DMA 控制器的方式来加以说明：

1. 关中断，以排除任何干扰；
2. 修改屏蔽寄存器（端口 0x0A），以屏蔽需要使用的 DMA 通道。对于软盘驱动程序来说就是通道 2；
3. 向 0x0C 端口写操作，置字节先后触发器为默认状态；
4. 写方式寄存器（端口 0x0B），以设置指定通道的操作方式字。对于；
5. 写地址寄存器（端口 0x04），设置 DMA 使用的内存页面中的偏移地址。先写低字节，后写高字节；
6. 写页面寄存器（端口 0x81），设置 DMA 使用的内存页面；
7. 写计数寄存器（端口 0x05），设置 DMA 传输的字节数。应该是传输长度-1。同样需要针对高低字节分别写一次。本书中软盘驱动程序每次要求 DMA 控制器传输的长度是 1024 字节，因此写 DMA 控制器的长度值应该是 1023（即 0x3FF）；
8. 再次修改屏蔽寄存器（端口 0x0A），以开启 DMA 通道；
9. 最后，开启中断，以允许软盘控制器在传输结束后向系统发出中断请求。

