```
/*
 *  linux/kernel/floppy.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 02.12.91 - Changed to static variables to indicate need for reset
 * and recalibrate. This makes some things easier (output_byte reset
 * checking etc), and means less interrupt jumping in case of errors,
 * so the code is hopefully easier to understand.
 */
/*
 * 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。
 */

/*
 * This file is certainly a mess. I've tried my best to get it working,
 * but I don't like programming floppies, and I have only one anyway.
 * Urgel. I should check for more errors, and do more graceful error
 * recovery. Seems there are problems with several drives. I've tried to
 * correct them. No promises.
 */
/*
 * 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。
 */

/*
 * As with hd.c, all routines within this file can (and will) be called
 * by interrupts, so extreme caution is needed. A hardware interrupt
 * handler may not sleep, or a kernel panic will happen. Thus I cannot
 * call "floppy-on" directly, but have to set a special timer interrupt
 * etc.
 *
 * Also, I'm not certain this works on more than 1 floppy. Bugs may
 * abund.
 */
/*
 * 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。
 */

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
#include <linux/fs.h>     // 文件系统头文件。含文件表结构（file、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
   // 定义软驱主设备号符号常数。在驱动程序中，主设备号必须在包含 blk.h 文件之前被定义。
   // 因为 blk.h 文件中要用到这个符号常数值来确定一些列其他相关符号常数和宏。
41 #define MAJOR_NR 2        // 软驱的主设备号是 2。
42 #include "blk.h"          // 块设备头文件。定义请求结构、块设备结构和宏函数等信息。
43
44 static int recalibrate = 0;       // 标志：1 表示需要重新校正磁头位置（磁头归零道）。
45 static int reset = 0;             // 标志：1 表示需要进行复位操作。
46 static int seek = 0;              // 标志：1 表示需要执行寻道操作。
47
   // 当前数字输出寄存器 DOR（Digital Output Register），定义在 kernel/sched.c，223 行。
   // 该变量含有软驱操作中的重要标志，包括选择软驱、控制电机启动、启动复位软盘控制器以
   // 及允许/禁止 DMA 和中断请求。请参见程序列表后对 DOR 寄存器的说明。
48 extern unsigned char current_DOR;
49
   // 字节直接数输出（嵌入汇编宏）。把值 val 输出到 port 端口。
50 #define immoutb_p(val,port) \
51 __asm__("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"::"a" ((char) (val)),"i" (port))
52
   // 这两个宏定义用于计算软驱的设备号。
   // 参数 x 是次设备号。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
    * 的错误会把出错计数值乘 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。另参见列表后说明。
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])                   // 结果状态字节 0。
68 #define ST1 (reply_buffer[1])                   // 结果状态字节 1。
69 #define ST2 (reply_buffer[2])                   // 结果状态字节 2。
70 #define ST3 (reply_buffer[3])                   // 结果状态字节 3。
71
72 /*
73  * This struct defines the different floppy types. Unlike minix
```

```
 * linux doesn't have a "search for right type"-type, as the code
 * for that is convoluted and weird. I've got enough problems with
 * this driver as it is.
 *
 * The 'stretch' tells if the tracks need to be boubled for some
 * types (ie 360kB diskette in 1.2MB drive etc). Others should
 * be self-explanatory.
 */
/*
 * 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
 * 已经让我遇到太多的问题了。
 *
 * 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。其他参数应该是自明的。
 */
// 定义软盘结构。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）。
static struct floppy_struct {
        unsigned int size, sect, head, track, stretch;
        unsigned char gap,rate,spec1;
} floppy_type[] = {
        {    0,  0,0,  0,0,0x00,0x00,0x00 },        /* no testing */
        {  720,  9,2,40,0,0x2A,0x02,0xDF },        /* 360kB PC diskettes */
        { 2400,15,2,80,0,0x1B,0x00,0xDF },        /* 1.2 MB AT-diskettes */
        {  720,  9,2,40,1,0x2A,0x02,0xDF },        /* 360kB in 720kB drive */
        { 1440,  9,2,80,0,0x2A,0x02,0xDF },        /* 3.5" 720kB diskette */
        {  720,  9,2,40,1,0x23,0x01,0xDF },        /* 360kB in 1.2MB drive */
        { 1440,  9,2,80,0,0x23,0x01,0xDF },        /* 720kB in 1.2MB drive */
        { 2880,18,2,80,0,0x1B,0x00,0xCF },        /* 1.44MB diskette */
};

/*
 * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
 * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
 * H is head unload time (1=16ms, 2=32ms, etc)
 *
 * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
 */
/*
 * 上面速率 rate：0 表示 500kbps，1 表示 300kbps，2 表示 250kbps。
 * 参数 spec1 是 0xSH，其中 S 是步进速率（F-1ms，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）。
     */
    // 注意，上述磁头加载时间的缩写 HLD 最好写成标准的 HLT（Head Load Time）。
104
    // floppy_interrupt()是 sys_call.s 程序中软驱中断处理过程标号。这里将在软盘初始化
    // 函数 floppy_init()（第 469 行）使用它初始化中断陷阱门描述符。
105 extern void floppy_interrupt(void);
    // 这是 boot/head.s 第 132 行处定义的临时软盘缓冲区。如果请求项的缓冲区处于内存 1MB
    // 以上某个地方，则需要将 DMA 缓冲区设在临时缓冲区域处。因为 8237A 芯片只能在 1MB 地
    // 址范围内寻址。
106 extern char tmp_floppy_area[1024];
107
108 /*
109  * These are global variables, as that's the easiest way to give
110  * information to interrupts. They are the data used for the current
111  * request.
112  */
    /*
     * 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
     * 用于当前请求项的数据。
     */
    // 这些所谓的“全局变量”是指在软盘中断处理程序中调用的 C 函数使用的变量。当然这些
    // C 函数都在本程序内。
113 static int cur_spec1 = -1;                         // 当前软盘参数 spec1。
114 static int cur_rate = -1;                          // 当前软盘转速 rate。
115 static struct floppy_struct * floppy = floppy_type; // 软盘类型结构数组指针。
116 static unsigned char current_drive = 0;             // 当前驱动器号。
117 static unsigned char sector = 0;                    // 当前扇区号。
118 static unsigned char head = 0;                      // 当前磁头号。
119 static unsigned char track = 0;                     // 当前磁道号。
120 static unsigned char seek_track = 0;                // 寻道磁道号。
121 static unsigned char current_track = 255;           // 当前磁头所在磁道号。
122 static unsigned char command = 0;                   // 读/写命令。
123 unsigned char selected = 0;    // 软驱已选定标志。在处理请求项之前要首先选定软驱。
124 struct task_struct * wait_on_floppy_select = NULL;  // 等待选定软驱的任务队列。
125
    //// 取消选定软驱。
    // 如果函数参数指定的软驱 nr 当前并没有被选定，则显示警告信息。然后复位软驱已选定标志
    // selected，并唤醒等待选择该软驱的任务。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）。
126 void floppy_deselect(unsigned int nr)
127 {
128         if (nr != (current_DOR & 3))
129                 printk("floppy_deselect: drive not selected\n\r");
130         selected = 0;                               // 复位软驱已选定标志。
131         wake_up(&wait_on_floppy_select);            // 唤醒等待的任务。
132 }
133
134 /*
135  * floppy-change is never called from an interrupt, so we can relax a bit
136  * here, sleep etc. Note that floppy-on tries to set current_DOR to point
137  * to the desired drive, but it will probably not survive the sleep if
138  * several floppies are used at the same time: thus the loop.
```

```
139  */
     /*
      * 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。
     // 参数 nr 是软驱号。如果软盘更换了则返回 1，否则返回 0。
     // 该函数首先选定参数指定的软驱 nr，然后测试软盘控制器的数字输入寄存器 DIR 的值，以判
     // 断驱动器中的软盘是否被更换过。该函数由程序 fs/buffer.c 中的 check_disk_change()函
     // 数调用（第 119 行）。
140  int floppy_change(unsigned int nr)
141  {
     // 首先要让软驱中软盘旋转起来并达到正常工作转速。这需要花费一定时间。采用的方法是利
     // 用 kernel/sched.c 中软盘定时函数 do_floppy_timer()进行一定的延时处理。floppy_on()
     // 函数则用于判断延时是否到（mon_timer[nr]==0?），若没有到则让当前进程继续睡眠等待。
     // 若延时到则 do_floppy_timer()会唤醒当前进程。
142  repeat:
143          floppy_on(nr);          // 启动并等待指定软驱 nr（kernel/sched.c，第 251 行）。
     // 在软盘启动（旋转）之后，我们来查看一下当前选择的软驱是不是函数参数指定的软驱 nr。
     // 如果当前选择的软驱不是指定的软驱 nr，并且已经选定了其他软驱，则让当前任务进入可
     // 中断等待状态，以等待其他软驱被取消选定。参见上面 floppy_deselect()。 如果当前没
     // 有选择其他软驱或者其他软驱被取消选定而使当前任务被唤醒时， 当前软驱仍然不是指定
     // 的软驱 nr，则跳转到函数开始处重新循环等待。
144          while ((current_DOR & 3) != nr && selected)
145                  sleep_on(&wait_on_floppy_select);
146          if ((current_DOR & 3) != nr)
147                  goto repeat;
     // 现在软盘控制器已选定我们指定的软驱 nr。于是取数字输入寄存器 DIR 的值，如果其最高
     // 位（位 7）置位，则表示软盘已更换，此时即可关闭马达并返回 1 退出。 否则关闭马达返
     // 回 0 退出。表示磁盘没有被更换。
148          if (inb(FD_DIR) & 0x80) {
149                  floppy_off(nr);
150                  return 1;
151          }
152          floppy_off(nr);
153          return 0;
154  }
155
     //// 复制内存缓冲块，共 1024 字节。
     // 从内存地址 from 处复制 1024 字节数据到地址 to 处。
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
     //// 设置（初始化）软盘 DMA 通道。
     // 软盘中数据读写操作是使用 DMA 进行的。因此在每次进行数据传输之前需要设置 DMA 芯片
     // 上专门用于软驱的通道 2。有关 DMA 编程方法请参见程序列表后的信息。
161  static void setup_DMA(void)
162  {
163          long addr = (long) CURRENT->buffer;       // 当前请求项缓冲区所处内存地址。
164
```

```
    // 首先检测请求项的缓冲区所在位置。如果缓冲区处于内存 1MB 以上的某个地方，则需要将
    // DMA 缓冲区设在临时缓冲区域（tmp_floppy_area）处。因为 8237A 芯片只能在 1MB 地址范
    // 围内寻址。如果是写盘命令，则还需要把数据从请求项缓冲区复制到该临时区域。
165         cli();
166         if (addr >= 0x100000) {
167                 addr = (long) tmp_floppy_area;
168                 if (command == FD_WRITE)
169                         copy_buffer(CURRENT->buffer,tmp_floppy_area);
170         }
    // 接下来我们开始设置 DMA 通道 2。在开始设置之前需要先屏蔽该通道。单通道屏蔽寄存器
    // 端口为 0x0A。位 0-1 指定 DMA 通道(0--3)，位 2：1 表示屏蔽，0 表示允许请求。 然后向
    // DMA 控制器端口 12 和 11 写入方式字（读盘是 0x46，写盘则是 0x4A）。 再写入传输使用
    // 缓冲区地址 addr 和需要传输的字节数 0x3ff（0--1023）。最后复位对 DMA 通道 2 的屏蔽，
    // 开放 DMA2 请求 DREQ 信号。
171 /* mask DMA 2 */    /* 屏蔽 DMA 通道 2 */
172         immoutb_p(4|2,10);
173 /* output command byte. I don't know why, but everyone (minix, */
174 /* sanches & canton) output this twice, first to 12 then to 11 */
    /* 输出命令字节。我是不知道为什么，但是每个人（minix，*/
    /* sanches 和 canton）都输出两次，首先是 12 口，然后是 11 口 */
    // 下面嵌入汇编代码向 DMA 控制器的“清除先后触发器”端口 12 和方式寄存器端口 11 写入
    // 方式字（读盘时是 0x46，写盘是 0x4A）。
    // 由于各通道的地址和计数寄存器都是 16 位的，因此在设置他们时都需要分 2 次进行操作。
    // 一次访问低字节，另一次访问高字节。而实际在写哪个字节则由先后触发器的状态决定。
    // 当触发器为 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，
    // 该触发器的状态就变化一次。而写端口 12 就可以将触发器置成 0 状态，从而对 16 位寄存
    // 器的设置从低字节开始。
175         __asm__("outb %%al,$12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
176         "outb %%al,$11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
177         "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
178 /* 8 low bits of addr */     /* 地址低 0-7 位 */
    // 向 DMA 通道 2 写入基/当前地址寄存器（端口 4）。
179         immoutb_p(addr,4);
180         addr >>= 8;
181 /* bits 8-15 of addr */    /* 地址高 8-15 位 */
182         immoutb_p(addr,4);
183         addr >>= 8;
184 /* bits 16-19 of addr */    /* 地址 16-19 位 */
    // DMA 只可以在 1MB 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)。
185         immoutb_p(addr,0x81);
186 /* low 8 bits of count-1 (1024-1=0x3ff) */  /* 计数器低 8 位(1024-1 = 0x3ff) */
    // 向 DMA 通道 2 写入基/当前字节计数器值（端口 5）。
187         immoutb_p(0xff,5);
188 /* high 8 bits of count-1 */    /* 计数器高 8 位 */
    // 一次共传输 1024 字节（两个扇区）。
189         immoutb_p(3,5);
190 /* activate DMA 2 */    /* 开启 DMA 通道 2 的请求 */
191         immoutb_p(0|2,10);
192         sti();
193 }
194
    //// 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，并且数据传输方向必须设置
```

```
    // 成从 CPU 到 FDC，因此函数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。若出错，则会设置复位标志 reset。
195 static void output_byte(char byte)
196 {
197         int counter;
198         unsigned char status;
199
    // 循环读取主状态控制器 FD_STATUS（0x3f4）的状态。如果所读状态是 STATUS_READY 并且
    // 方向位 STATUS_DIR = 0（CPU→FDC），则向数据端口输出指定字节。
200         if (reset)
201                 return;
202         for(counter = 0 ; counter < 10000 ; counter++) {
203                 status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
204                 if (status == STATUS_READY) {
205                         outb(byte,FD_DATA);
206                         return;
207                 }
208         }
    // 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。
209         reset = 1;
210         printk("Unable to send byte to FDC\n\r");
211 }
212
    //// 读取 FDC 执行的结果信息。
    // 结果信息最多 7 个字节，存放在数组 reply_buffer[]中。返回读入的结果字节数，若返回
    // 值 = -1，则表示出错。程序处理方式与上面函数类似。
213 static int result(void)
214 {
215         int i = 0, counter, status;
216
    // 若复位标志已置位，则立刻退出。去执行后续程序中的复位操作。否则循环读取主状态控
    // 制器 FD_STATUS（0x3f4）的状态。如果读取的控制器状态是 READY，表示已经没有数据可
    // 取，则返回已读取的字节数 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。
217         if (reset)
218                 return -1;
219         for (counter = 0 ; counter < 10000 ; counter++) {
220                 status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
221                 if (status == STATUS_READY)
222                         return i;
223                 if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
224                         if (i >= MAX_REPLIES)
225                                 break;
226                         reply_buffer[i++] = inb_p(FD_DATA);
227                 }
228         }
    // 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。
229         reset = 1;
230         printk("Getstatus times out\n\r");
231         return -1;
232 }
233
```

```
    //// 软盘读写出错处理函数。
    // 该函数根据软盘读写出错次数来确定需要采取的进一步行动。如果当前处理的请求项出错
    // 次数大于规定的最大出错次数 MAX_ERRORS（8 次），则不再对当前请求项作进一步的操作
    // 尝试。如果读/写出错次数已经超过 MAX_ERRORS/2，则需要对软驱作复位处理，于是设置
    // 复位标志 reset。否则若出错次数还不到最大值的一半，则只需重新校正一下磁头位置，
    // 于是设置重新校正标志 recalibrate。真正的复位和重新校正处理会在后续的程序中进行。
234 static void bad_flp_intr(void)
235 {
    // 首先把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         CURRENT->errors++;
237         if (CURRENT->errors > MAX_ERRORS) {
238                 floppy_deselect(current_drive);
239                 end_request(0);
240         }
    // 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。
241         if (CURRENT->errors > MAX_ERRORS/2)
242                 reset = 1;
243         else
244                 recalibrate = 1;
245 }
246
247 /*
248  * Ok, this interrupt is called after a DMA read/write has succeeded,
249  * so we check the results, and copy any buffers.
250  */
    /*
     * 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。
     */
    //// 软盘读写操作中断调用函数。
    // 该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。函数首先读取操作结果状
    // 态信息，据此判断操作是否出现问题并作相应处理。 如果读/写操作成功，那么若请求项
    // 是读操作并且其缓冲区在内存 1MB 以上位置，则需要把数据从软盘临时缓冲区复制到请求
    // 项的缓冲区。
251 static void rw_interrupt(void)
252 {
    // 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 )
253         if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
254                 if (ST1 & 0x02) {         // 0x02 = ST1_WP - Write Protected。
255                         printk("Drive %d is write protected\n\r",current_drive);
256                         floppy_deselect(current_drive);
257                         end_request(0);
258                 } else
259                         bad_flp_intr();
260                 do_fd_request();
261                 return;
```

```
262         }
    // 如果当前请求项的缓冲区位于 1MB 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 1MB 地址范围寻址）。最后
    // 释放当前软驱（取消选定），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
    // 行其他软盘请求项操作。
263         if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
264                 copy_buffer(tmp_floppy_area,CURRENT->buffer);
265         floppy_deselect(current_drive);
266         end_request(1);
267         do_fd_request();
268 }
269
    //// 设置 DMA 通道 2 并向软盘控制器输出命令和参数（输出 1 字节命令 + 0~7 字节参数）。
    // 若 reset 标志没有置位，那么在该函数退出并且软盘控制器执行完相应读/写操作后就会
    // 产生一个软盘中断请求，并开始执行软盘中断处理程序。
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：(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 ?) */
                                    // 参数：当 N=0 时，扇区定义的字节长度，这里无用。
    // 若上述任何一个 output_byte()操作出错，则会设置复位标志 reset。此时即会立刻去执行
    // do_fd_request()中的复位处理代码。
283         if (reset)
284                 do_fd_request();
285 }
286
287 /*
288  * This is the routine called after every seek (or recalibrate) interrupt
289  * from the floppy controller. Note that the "unexpected interrupt" routine
290  * also does a recalibrate, but doesn't come here.
291  */
    /*
     * 该子程序是在每次软盘控制器寻道（或重新校正）中断中被调用的。注意
     * "unexpected interrupt"(意外中断)子程序也会执行重新校正操作，但不在此地。
     */
    //// 寻道处理结束后中断过程中调用的 C 函数。
    // 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 setup_rw_floppy()设置 DMA 并输出软盘读写命令和参数。
292 static void seek_interrupt(void)
293 {
    // 首先发送检测中断状态命令，以获取寻道操作执行的结果。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。然后读取 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，然后继续执行软盘请求项或执行复位处理。
294 /* sense drive status */    /* 检测驱动器状态 */
295         output_byte(FD_SENSEI);
296         if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
297                 bad_flp_intr();
298                 do_fd_request();
299                 return;
300         }
    // 若寻道操作成功，则继续执行当前请求项的软盘操作，即向软盘控制器发送命令和参数。
301         current_track = ST1;        // 设置当前磁道。
302         setup_rw_floppy();          // 设置 DMA 并输出软盘操作命令和参数。
303 }
304
305 /*
306  * This routine is called when everything should be correctly set up
307  * for the transfer (ie floppy motor is on and the correct floppy is
308  * selected).
309  */
    /*
     * 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（即软驱马达已开启
     * 并且已选择了正确的软盘（软驱）。
     */
    //// 读写数据传输函数。
310 static void transfer(void)
311 {
    // 首先检查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）。
    // 然后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
    // 数据传输速率控制寄存器(FD_DCR)。
312         if (cur_spec1 != floppy->spec1) {           // 检测当前参数。
313                 cur_spec1 = floppy->spec1;
314                 output_byte(FD_SPECIFY);            // 发送设置磁盘参数命令。
315                 output_byte(cur_spec1);             /* hut etc */  // 发送参数。
316                 output_byte(6);                     /* Head load time =6ms, DMA */
317         }
318         if (cur_rate != floppy->rate)               // 检测当前速率。
319                 outb_p(cur_rate = floppy->rate,FD_DCR);
    // 若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。因此这里
    // 我们需要检测一下 reset 标志。若 reset 真的被置位了，就立刻去执行 do_fd_request()
    // 中的复位处理代码。
320         if (reset) {
321                 do_fd_request();
322                 return;
323         }
    // 如果此时寻道标志为零（即不需要寻道），则设置 DMA 并向软盘控制器发送相应操作命令
    // 和参数后返回。否则就执行寻道处理，于是首先置软盘中断处理调用函数为寻道中断函数。
    // 如果起始磁道号不等于零则发送磁头寻道命令和参数。所使用的参数即是第 112--121 行
    // 上设置的全局变量值。如果起始磁道号 seek_track 为 0，则执行重新校正命令让磁头归零
    // 位。
324         if (!seek) {
325                 setup_rw_floppy();                  // 发送命令参数块。
326                 return;
```

```
327         }
328         do_floppy = seek_interrupt;                     // 寻道中断调用的 C 函数。
329         if (seek_track) {                               // 起始磁道号。
330                 output_byte(FD_SEEK);                   // 发送磁头寻道命令。
331                 output_byte(head<<2 | current_drive);// 发送参数：磁头号+当前软驱号。
332                 output_byte(seek_track);                // 发送参数：磁道号。
333         } else {
334                 output_byte(FD_RECALIBRATE);            // 发送重新校正命令（磁头归零）。
335                 output_byte(head<<2 | current_drive);// 发送参数：磁头号+当前软驱号。
336         }
    // 同样地，若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。
    // 若 reset 真的被置位了，就立刻去执行 do_fd_request()中的复位处理代码。
337         if (reset)
338                 do_fd_request();
339 }
340
341 /*
342  * Special case - used after a unexpected interrupt (or reset)
343  */
    /*
     * 特殊情况 - 用于意外中断（或复位）处理后。
     */
    //// 软驱重新校正中断调用函数。
    // 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。否则重新
    // 校正标志清零。然后再次执行软盘请求项处理函数作相应操作。
344 static void recal_interrupt(void)
345 {
346         output_byte(FD_SENSEI);                         // 发送检测中断状态命令。
347         if (result()!=2 || (ST0 & 0xE0) == 0x60)        // 如果返回结果字节数不等于 2 或命令
348                 reset = 1;                              // 异常结束，则置复位标志。
349         else
350                 recalibrate = 0;                        // 否则复位重新校正标志。
351         do_fd_request();                                // 作相应处理。
352 }
353
    //// 意外软盘中断请求引发的软盘中断处理程序中调用的函数。
    // 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
    // 校正标志。
354 void unexpected_floppy_interrupt(void)
355 {
356         output_byte(FD_SENSEI);                         // 发送检测中断状态命令。
357         if (result()!=2 || (ST0 & 0xE0) == 0x60)        // 如果返回结果字节数不等于 2 或命令
358                 reset = 1;                              // 异常结束，则置复位标志。
359         else
360                 recalibrate = 1;                        // 否则置重新校正标志。
361 }
362
    //// 软盘重新校正处理函数。
    // 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。当软盘控制器执行完
    // 重新校正命令就会再其引发的软盘中断中调用 recal_interrupt()函数。
363 static void recalibrate_floppy(void)
364 {
365         recalibrate = 0;                                // 复位重新校正标志。
```

```
366         current_track = 0;                          // 当前磁道号归零。
367         do_floppy = recal_interrupt;                // 指向重新校正中断调用的 C 函数。
368         output_byte(FD_RECALIBRATE);                // 命令：重新校正。
369         output_byte(head<<2 | current_drive);       // 参数：磁头号 + 当前驱动器号。
    // 若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。因此这里
    // 我们需要检测一下 reset 标志。若 reset 真的被置位了，就立刻去执行 do_fd_request()
    // 中的复位处理代码。
370         if (reset)
371                 do_fd_request();
372 }
373
    //// 软盘控制器 FDC 复位中断调用函数。
    // 该函数会在向控制器发送了复位操作命令后引发的软盘中断处理程序中被调用。
    // 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
    // 参数命令和相关参数，最后再次调用请求项处理函数 do_fd_request() 去执行重新校正
    // 操作。但由于执行 output_byte() 函数出错时复位标志又会被置位，因此也可能再次去
    // 执行复位处理。
374 static void reset_interrupt(void)
375 {
376         output_byte(FD_SENSEI);                     // 发送检测中断状态命令。
377         (void) result();                            // 读取命令执行结果字节。
378         output_byte(FD_SPECIFY);                    // 发送设定软驱参数命令。
379         output_byte(cur_spec1);                     /* hut etc */  // 发送参数。
380         output_byte(6);                             /* Head load time =6ms, DMA */
381         do_fd_request();                            // 调用执行软盘请求。
382 }
383
384 /*
385  * reset is done by pulling bit 2 of DOR low for a while.
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 */
    //// 复位软盘控制器。
    // 该函数首先设置参数和标志，把复位标志清 0，然后把软驱变量 cur_spec1 和 cur_rate
    // 置为无效。因为复位操作后，这两个参数就需要重新设置。接着设置需要重新校正标志，
    // 并设置 FDC 执行复位操作后引发的软盘中断中调用的 C 函数 reset_interrupt()。最后
    // 把 DOR 寄存器位 2 置 0 一会儿以对软驱执行复位操作。当前数字输出寄存器 DOR 的位 2
    // 是启动/复位软驱位。
387 static void reset_floppy(void)
388 {
389         int i;
390
391         reset = 0;                                  // 复位标志置 0。
392         cur_spec1 = -1;                             // 使无效。
393         cur_rate = -1;
394         recalibrate = 1;                            // 重新校正标志置位。
395         printk("Reset-floppy called\n\r");          // 显示执行软盘复位操作信息。
396         cli();                                      // 关中断。
397         do_floppy = reset_interrupt;                // 设置在中断处理程序中调用的函数。
398         outb_p(current_DOR & ~0x04,FD_DOR);         // 对软盘控制器 FDC 执行复位操作。
399         for (i=0 ; i<100 ; i++)                     // 空操作，延迟。
400                 __asm__("nop");
401         outb(current_DOR,FD_DOR);                   // 再启动软盘控制器。
402         sti();                                      // 开中断。
```

```
403 }
404
    //// 软驱启动定时中断调用函数。
    // 在执行一个请求项要求的操作之前，为了等待指定软驱马达旋转起来到达正常的工作转速，
    // do_fd_request()函数为准备好的当前请求项添加了一个延时定时器。本函数即是该定时器
    // 到期时调用的函数。它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函数 transfer()。
405 static void floppy_on_interrupt(void)          // floppy_on() interrupt。
406 {
407 /* We cannot do a floppy-select, as that might sleep. We just force it */
    /* 我们不能任意设置选择的软驱，因为这可能会引起进程睡眠。我们只是迫使它自己选择 */
    // 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。
    // 在向数字输出寄存器输出当前 DOR 以后，使用定时器延迟 2 个滴答时间，以让命令得到执
    // 行。然后调用软盘读写传输函数 transfer()。若当前驱动器与 DOR 中的相符，那么就可以
    // 直接调用软盘读写传输函数。
408         selected = 1;                           // 置已选定当前驱动器标志。
409         if (current_drive != (current_DOR & 3)) {
410                 current_DOR &= 0xFC;
411                 current_DOR |= current_drive;
412                 outb(current_DOR,FD_DOR);       // 向数字输出寄存器输出当前 DOR。
413                 add_timer(2,&transfer);         // 添加定时器并执行传输函数。
414         } else
415                 transfer();                     // 执行软盘读写传输函数。
416 }
417
    //// 软盘读写请求项处理函数。
    // 该函数是软盘驱动程序中最主要的函数。主要作用是：①处理有复位标志或重新校正标志置
    // 位情况；②利用请求项中的设备号计算取得请求项指定软驱的参数块；③利用内河定时器启
    // 动软盘读/写操作。
418 void do_fd_request(void)
419 {
420         unsigned int block;
421
    // 首先检查是否有复位标志或重新校正标志置位，若有则本函数仅执行相关标志的处理功能
    // 后就返回。如果复位标志已置位，则执行软盘复位操作并返回。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。
422         seek = 0;                               // 清寻道标志。
423         if (reset) {                            // 复位标志已置位。
424                 reset_floppy();
425                 return;
426         }
427         if (recalibrate) {                      // 重新校正标志已置位。
428                 recalibrate_floppy();
429                 return;
430         }
    // 本函数的真正功能从这里开始。首先利用 blk.h 文件中的 INIT_REQUEST 宏来检测请求项的
    // 合法性，如果已没有请求项则退出（参见 blk.h,127）。然后利用请求项中的设备号取得请
    // 求项指定软驱的参数块。这个参数块将在下面用于设置软盘操作使用的全局变量参数块（参
    // 见 112 - 122 行）。请求项设备号中的软盘类型 (MINOR(CURRENT->dev)>>2) 被用作磁盘类
    // 型数组 floppy_type[]的索引值来取得指定软驱的参数块。
431         INIT_REQUEST;
432         floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;
```

```
    // 下面开始设置 112--122 行上的全局变量值。如果当前驱动器号 current_drive 不是请求项
    // 中指定的驱动器号，则置标志 seek，表示在执行读/写操作之前需要先让驱动器执行寻道处
    // 理。然后把当前驱动器号设置为请求项中指定的驱动器号。
433         if (current_drive != CURRENT_DEV)    // CURRENT_DEV 是请求项中指定的软驱号。
434                 seek = 1;
435         current_drive = CURRENT_DEV;

    // 设置读写起始扇区 block。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
    // 需要起码比磁盘总扇区数小 2 个扇区。否则说明这个请求项参数无效，结束该次软盘请求项
    // 去执行下一个请求项。
436         block = CURRENT->sector;             // 取当前软盘请求项中起始扇区号。
437         if (block+2 > floppy->size) {        // 如果 block + 2 大于磁盘扇区总数，
438                 end_request(0);              // 则结束本次软盘请求项。
439                 goto repeat;
440         }
    // 再求对应在磁道上的扇区号、磁头号、磁道号、搜寻磁道号（对于软驱读不同格式的盘）。
441         sector = block % floppy->sect;   // 起始扇区对每磁道扇区数取模，得磁道上扇区号。
442         block /= floppy->sect;           // 起始扇区对每磁道扇区数取整，得起始磁道数。
443         head = block % floppy->head;     // 起始磁道数对磁头数取模，得操作的磁头号。
444         track = block / floppy->head;    // 起始磁道数对磁头数取整，得操作的磁道号。
445         seek_track = track << floppy->stretch; // 相应于软驱中盘类型进行调整，得寻道号。

    // 再看看是否还需要首先执行寻道操作。如果寻道号与当前磁头所在磁道号不同，则需要进行
    // 寻道操作，于是置需要寻道标志 seek。最后我们设置执行的软盘命令 command。
446         if (seek_track != current_track)
447                 seek = 1;
448         sector++;                            // 磁盘上实际扇区计数是从 1 算起。
449         if (CURRENT->cmd == READ)            // 如果请求项是读操作，则置读命令码。
450                 command = FD_READ;
451         else if (CURRENT->cmd == WRITE)  // 如果请求项是写操作，则置写命令码。
452                 command = FD_WRITE;
453         else
454                 panic("do_fd_request: unknown command");
    // 在上面设置好 112--122 行上所有全局变量值之后，我们可以开始执行请求项操作了。该操
    // 作利用定时器来启动。因为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。而这需要一定的时间。因此这里利用 ticks_to_floppy_on() 来计算启动延时
    // 时间，然后使用该延时设定一个定时器。当时间到时就调用函数 floppy_on_interrupt()。
455         add_timer(ticks_to_floppy_on(current_drive),&floppy_on_interrupt);
456 }
457
    // 各种类型软驱磁盘含有的数据块总数。
458 static int floppy_sizes[] ={
459            0,    0,    0,    0,
460          360,  360 ,360,  360,
461         1200,1200,1200,1200,
462          360,  360,  360,  360,
463          720,  720,  720,  720,
464          360,  360,  360,  360,
465          720,  720,  720,  720,
466         1440,1440,1440,1440
467 };
468
    //// 软盘系统初始化。
```

```
    // 设置软盘块设备请求项的处理函数 do_fd_request()，并设置软盘中断门（int 0x26，对应
    // 硬件中断请求信号 IRQ6）。 然后取消对该中断信号的屏蔽，以允许软盘控制器 FDC 发送中
    // 断请求信号。中断描述符表 IDT 中陷阱门描述符设置宏 set_trap_gate()定义在头文件
    // include/asm/system.h 中。
469 void floppy_init(void)
470 {
    // 设置软盘中断门描述符。floppy_interrupt（kernel/sys_call.s，267 行）是其中断处理
    // 过程。中断号为 int 0x26（38），对应 8259A 芯片中断请求信号 IRQ6。
471         blk_size[MAJOR_NR] = floppy_sizes;
472         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // = do_fd_request()。
473         set_trap_gate(0x26,&floppy_interrupt);          // 设置陷阱门描述符。
474         outb(inb_p(0x21)&~0x40,0x21);                   // 复位软盘中断请求屏蔽位。
475 }
476
```