Ampliación de Sistemas Operativos

Controlador de Disco Flexible

(FDC y Floppy.c)

Enrique Pérez Díaz
Miguel Ángel Umpiérrez Artiles
© Universidad de Las Palmas de Gran Canaria

• Introducción.

El controlador de disco flexible es el código encargado de controlar el dispositivo Floppy.

Su trabajo consiste en aceptar solicitudes (en alto nivel) del software independientes del dispositivo y observar que se cumplan dichas solicitudes.

Un ejemplo es la lectura de un byte en este dispositivo que es de bloque.

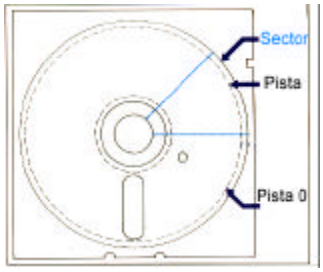
Se divide en dos módulos:

Módulo independiente del dispositivo: driver.c
Módulo dependiente del dispositivo: floppy.c

El manejador de dispositivos FDC, se encuentra en la CAPA 2 de la estructura interna de MINIX (ver apuntes capítulo 4.1.). No comparte el espacio de direcciones con el kernel (UNIX), sino que son procesos independientes (MODULAR).

• Introducción a los dispositivos flexibles.

- **Pistas** (tracks): 40 ó 80 círculos concéntricos; la pista 0 la más externa.
- **Sectores** (sectors): Cada pista se puede dividir en sectores circulares (1 ...).
- **Cabeza** (Head): Indica la cara del disco. 2 cabezas.
- **Cilindros**: Nº Pistas por Cara (Pistas/Cabezas).



En la parte mecánica encontramos dos motores:

- Un motor que hace girar el disco cuando se va acceder a él.
- Y otro motor que avanza o retrocede un poco con cada pulso de corriente que se le aplica. Este último se usa para que las cabezas de lectura/escritura (brazo) se muevan por las pistas.

• Controlador Físico de Floppy (Intel 8272A).

Vista Previa del Controlador.

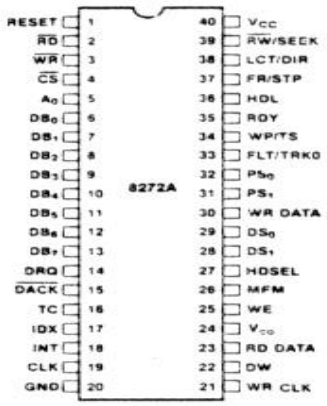
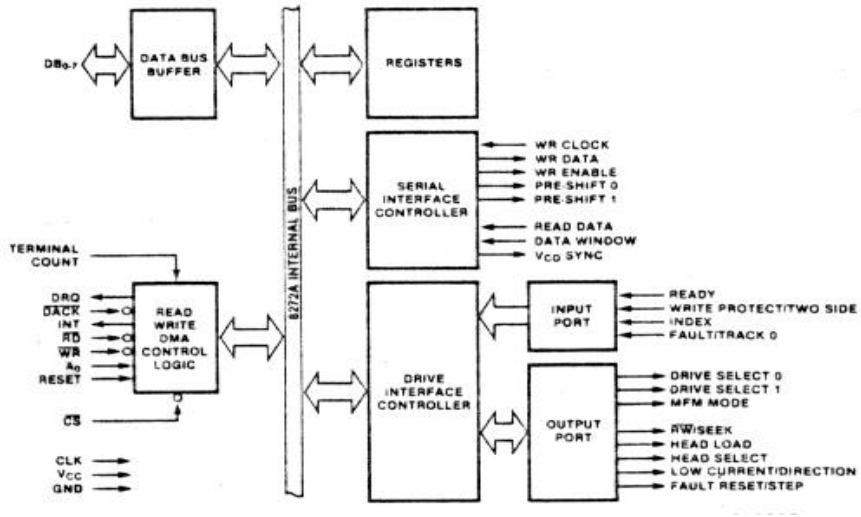


Diagrama de Bloques.



Características principales:

- Permite dos formatos de grabación: simple (FM) y doble densidad (MFM) (compatible con IBM).
- Longitud de registro de datos programables: 128, 256, 512 ó 1024 bytes/sector.
- Transferencia Multi-Sector y Multi-Pista.
- Puede manejar 4 dispositivos de Floppy.
- Permite discos de 8", 5 1/4" y 3 1/2".
- Transferencia DMA y Non-DMA.
- Operación de Búsqueda o posicionamiento en paralelo en las 4 unidades.

Modos de Funcionamiento

En modo Non-DMA, el FDC genera interrupciones al procesador para cada transferencia de bytes de datos entre la CPU y el FDC (INT 1). En modo DMA, el procesador necesita sólo cargar un comando en el FDC y toda la transferencia de datos ocurrirá bajo el control de el 8271A y el controlador de DMA.

Comunicación con el Procesador

- El 8272A contiene dos registros que pueden ser accedidos por el procesador:
- Registro de Estado Principal (Main Status Register) 0x3F4h:

  Indica el estado del controlador de floppy.
  Puede ser accedido en cualquier momento.

- Registros de Datos (Data Register) 0x3F5h:

  Indica el estado en la ejecución después de la ejecución de un comando (4 registros).

  Sólo puede ser accedido en la fase de resultados de la ejecución de un comando. Hasta que no se lean los 4 registros no se puede lanzar otro comando.

Main Status Register

0: El dispositivo floppy 0 ocupado (Seek mode).
1: El dispositivo floppy 1 ocupado (Seek mode).
2: El dispositivo floppy 2 ocupado (Seek mode).
3: El dispositivo floppy 3 ocupado (Seek mode).
4: Comando de lectura o escritura en proceso.
5: Modo non-DMA (sin acceso Directo Memoria).
6: Sentido:
0 = CPU ? FDC ;
1 = FDC ? CPU;
7: Indica que el Registro de Datos está listo para recibir o enviar datos del o para el procesador.

Data Register

Se divide en 4 registros, que se leen por el mismo puerto (3F5h) de forma secuencial:

Command Status Register 0 (ST0)

¦7¦6¦5¦4¦3¦2¦1¦0¦ Command Status Register 0 at port 3F5h

```
¦ ¦ ¦ ¦ ¦ ¦ +-+---- unit selected at interrupt (0=A, 1=B, 2=...)
¦ ¦ ¦ ¦ ¦ +------- head number at interrupt (head 0 or 1)
¦ ¦ ¦ ¦ +-------- not ready on read/write or SS access to head 1
¦ ¦ ¦ +--------- equipment check (see note)
¦ ¦ +---------- set to 1 when FDD completes a seek command
+-+----------- last command status (see below)
```

Bits

76 Last Command Status

00 command terminated successfully

01 command execution started but terminated abnormally

10 invalid command issued

11 command terminated abnormally due to a change in state of the Ready Signal from the FDC (reserved on PS/2)

Command Status Register 1 (ST1)

```
¦7¦6¦5¦4¦3¦2¦1¦0¦  Command Status Register 1 at port 3F5h
¦ ¦ ¦ ¦ ¦ ¦ ¦ +---- FDC cannot find ID address mark (see reg 2)
¦ ¦ ¦ ¦ ¦ ¦ +----- write protect detected during write
¦ ¦ ¦ ¦ ¦ +------ FDC cannot find sector ID
¦ ¦ ¦ ¦ +------- unused (always zero)
¦ ¦ ¦ +-------- over-run;  FDC not serviced in reasonable time
¦ ¦ +--------- data error (CRC) in ID field or data field
¦ +---------- unused (always zero)
+----------- end of cylinder; sector# greater than sectors/track
```

Bit 0 of Status Register 1 and bit 4 of Status Register 2 are related and mimic each other.

Command Status Register 2 (ST2)

```
¦7¦6¦5¦4¦3¦2¦1¦0¦  Command Status Register 2 at port 3F5h
¦ ¦ ¦ ¦ ¦ ¦ ¦ +---- missing address mark in data field
¦ ¦ ¦ ¦ ¦ ¦ +----- bad cylinder, ID not found and Cyl Id=FFh
```

¦ ¦ ¦ ¦ ¦ +------ scan command failed,sector not found in cyl.
¦ ¦ ¦ ¦ +------- scan command equal condition satisfied
¦ ¦ ¦ +-------- wrong cylinder detected
¦ ¦ +--------- CRC error detected in sector data
¦ +---------- sector with deleted data address mark detected
+----------- unused (always zero)

Command Status Register 3 (ST3)

¦7¦6¦5¦4¦3¦2¦1¦0¦ Command Status Register 3 at port 3F5h
¦ ¦ ¦ ¦ ¦ ¦ +-+--- FDD unit selected status (0=A, 1=B, 2=...)
¦ ¦ ¦ ¦ ¦ +------ FDD side head select status (0=head 0, 1=head 1)
¦ ¦ ¦ ¦ +------- FDD two sided status signal
¦ ¦ ¦ +-------- FDD track zero status signal
¦ ¦ +--------- FDD ready status signal
¦ +---------- FDD write protect status signal
+----------- FDD fault status signal

Otros Registros

- Digital Output Register (DOR): Se utiliza para controlar los motores, selección de unidad, ...

  Bit 0 y 1: Indica el floppy seccionado (0=A, 1=B, ...)
  Bit 2:
  0 = Estado hold (reseteo)
  1 = FDC habilitado.
  Bit 3: la interface DMA & I/O está habilitada.
  Bit 4,5,6,7: Pone en marcha el motor del disco A,B, ... respectivamente.
  Durante el reseteo del controlador todos los bits se ponen a cero.

- Tranfer Rate Register (FDC_RATE): Los valores que puede tomar son: 0=500kbps, 1=300 kbps, 2=250kbps.

Comandos

Hay 15 comandos que el 8272A ejecuta. Cada uno de estos comandos requiere multiples bytes para especificar completamente la operación que el procesador desea que realice el FDC.

Relación de comandos empleados en el driver:

Read Data	Write Data
Read ID	Format a Track
Seek	Recalibrate
Sense Interrupt Status	Specify

Fases de Ejecución

Cada comando es inicializado por múltiples bytes transmitidos por el procesador (CPU ? FDC).
El resultado después de la ejecución de un comando puede ser también una transferencia de múltiples bytes (FDC? CPU).
Debido a esto es conveniente dividir los comandos en tres fases.

- **Fase de Comando:**

  - Se envían los comandos desde la CPU hasta el FDC.
  - Los bits 6 y 7 del Main Status Register deben estar a 0 y a 1 respectivamente.
  - El Main Status Register debe ser leído por el procesador antes de que cada byte de información sea escrito en el Data Register.

- **Fase de Ejecución:**

  - El FDC ejecuta el comando y cuando termina:
  - Genera una INT 1 (en modo non-DMA) por cada byte de datos.
    En modo DMA no se generan interrupciones sino que se generan señales (patas procesador) entre el controlador Floppy y el controlador de DMA, para sincronizar transferencia .

- **Fase de Resultados:**

- La información del estado y los resultados están disponibles a la CPU. Al igual que en la etapa de comandos, el Main Status Register debe ser leído por el procesador antes de que cada byte de información sea leído del Data Register.

- Sólo durante esta fase se puede acceder a los 4 registros de datos mencionados anteriormente, dependiendo del comando se leerá uno u otros registros.

• Comunicación entre driver.c, floppy.c y otros módulos.

El controlador de disco flexible es inicializado por el main.c, en la fase de inicialización de tareas llamando a floppy_task.

Luego éste llama a driver_task (driver.c) pasándole una estructura con todas las funciones que necesita para las solicitudes del sistema de fichero (f_dtab).

El driver.c (en driver_task) entra ahora en un bucle en espera de solicitudes del sistema de ficheros.

¡Ojo! los datos se leen o escriben directamente a la zona de memoria del proceso que hace la llamada.

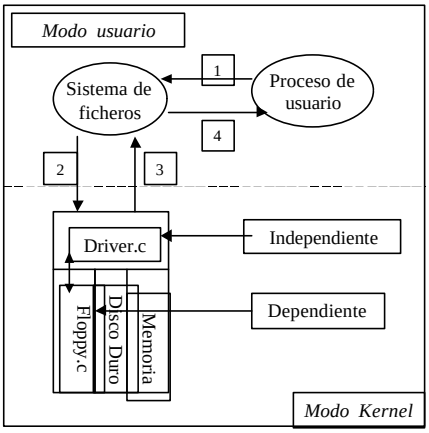
Driver.c

```
PUBLIC void driver_task(dp)
struct driver *dp;   /* Device dependent entry points. */
{ ...
    while (TRUE) {
    receive(ANY, &mess); ...
    switch(mess.m_type) {
       case DEV_OPEN:     r = (*dp->dr_open)(dp, &mess);        break;
       case DEV_CLOSE:    r = (*dp->dr_close)(dp, &mess);       break;
       case DEV_READ:
       case DEV_WRITE:    r = do_rdwt(dp, &mess);               break;
       default:           r = EINVAL;                  break;
    }
    (*dp->dr_cleanup)();
   ....
    send(caller, &mess);    /* send reply to caller */
 }}
```

- Las operaciones de lectura y escritura se separan en 3 funciones.

- Supuestamente porque el objetivo es crear funciones modulares, que puedan ser reutilizadas.
- En do_rdwt es donde se hacen las 3 llamadas, entre otras cosas, de las operaciones de lectura y escritura.

Otros Módulos.



- **Floppy.c.**

Declaraciones.

```
#include "kernel.h"
#include "driver.h"
#include "drvlib.h"
#include <ibm/diskparm.h>

/* I/O Ports used by floppy disk task. */
#define DOR           0x3F2           /* motor drive control bits */
#define FDC_STATUS    0x3F4   /* floppy disk controller status register */
#define FDC_DATA      0x3F5   /* floppy disk controller data register */
#define FDC_RATE      0x3F7   /* transfer rate register */
#define DMA_ADDR      0x004   /* port for low 16 bits of DMA address */
#define DMA_TOP       0x081   /* port for top 4 bits of 20-bit DMA addr */
#define DMA_COUNT     0x005  /* port for DMA count (count =  bytes - 1) */
#define DMA_FLIPFLOP  0x00C/* DMA byte pointer flip-flop */
#define DMA_MODE      0x00B  /* DMA mode port */
#define DMA_INIT      0x00A   /* DMA init port */
#define DMA_RESET_VAL  0x06

/* Status registers returned as result of operation. */
#define ST0           0x00  /* status register 0 */
#define ST1           0x01  /* status register 1 */
#define ST2           0x02  /* status register 2 */
#define ST3           0x00  /* status register 3 (return by DRIVE_SENSE) */
#define ST_CYL        0x03          /* slot where controller reports cylinder */
#define ST_HEAD       0x04          /* slot where controller reports head */
#define ST_SEC        0x05          /* slot where controller reports sector */
#define ST_PCN        0x01          /* slot where controller reports present cyl */
```

```
/* Fields within the I/O ports. */
/* Main status register. */
#define CTL_BUSY        0x10    /* bit is set when read or write in progress */
#define DIRECTION       0x40    /* bit is set when reading data reg is valid */
#define MASTER          0x80    /* bit is set when data reg can be accessed */

/* Digital output port (DOR). */
#define MOTOR_SHIFT     4       /* high 4 bits control the motors in DOR */
#define ENABLE_INT      0x0C    /* used for setting DOR port */

/* ST0. */
#define ST0_BITS        0xF8    /* check top 5 bits of seek status */
#define TRANS_ST0       0x00    /* top 5 bits of ST0 for READ/WRITE */
#define SEEK_ST0        0x20    /* top 5 bits of ST0 for SEEK */

/* ST1. */
#define BAD_SECTOR      0x05    /* if these bits are set in ST1, recalibrate */
#define WRITE_PROTECT   0x02    /* bit is set if diskette is write protected */

/* ST2. */
#define BAD_CYL         0x1F    /* if any of these bits are set, recalibrate */

/* ST3 (not used). */
#define ST3_FAULT       0x80    /* if this bit is set, drive is sick */
#define ST3_WR_PROTECT  0x40    /* set when diskette is write protected */
#define ST3_READY       0x20    /* set when drive is ready */

/* Floppy disk controller command bytes. */
#define FDC_SEEK        0x0F    /* command the drive to seek */
#define FDC_READ        0xE6    /* command the drive to read */
#define FDC_WRITE       0xC5    /* command the drive to write */
#define FDC_SENSE       0x08    /* command the controller to tell its status */
#define FDC_RECALIBRATE 0x07    /* command the drive to go to cyl 0 */
#define FDC_SPECIFY     0x03    /* command the drive to accept params */
#define FDC_READ_ID     0x4A    /* command the drive to read sector identity */
#define FDC_FORMAT      0x4D    /* command the drive to format a track */

/* DMA channel commands. */
#define DMA_READ        0x46    /* DMA read opcode */
#define DMA_WRITE       0x4A    /* DMA write opcode */

/* Parameters for the disk drive. */
#define HC_SIZE         2880    /* # sectors on largest legal disk (1.44MB) */
#define NR_HEADS        0x02    /* two heads (i.e., two tracks/cylinder) */
#define MAX_SECTORS     18      /* largest # sectors per track */
#define DTL             0xFF    /* determines data length (sector size) */
#define SPEC2           0x02    /* second parameter to SPECIFY */
#define MOTOR_OFF       3*HZ    /* how long to wait before stopping motor */
#define WAKEUP                  2*HZ    /* timeout on I/O, FDC won't quit. */

/* Error codes */
#define ERR_SEEK        (-1)    /* bad seek */
#define ERR_TRANSFER    (-2)    /* bad transfer */
#define ERR_STATUS      (-3)    /* something wrong when getting status */
#define ERR_READ_ID     (-4)    /* bad read id */
```

```
#define ERR_RECALIBRATE  (-5)          /* recalibrate didn't work properly */
#define ERR_DRIVE        (-6)        /* something wrong with a drive */
#define ERR_WR_PROTECT   (-7)          /* diskette is write protected */
#define ERR_TIMEOUT      (-8)    /* interrupt timeout */

/* No retries on some errors. */
#define err_no_retry(err)((err) <= ERR_WR_PROTECT)

/* Encoding of drive type in minor device number. */
#define DEV_TYPE_BITS   0x7C /* drive type + 1, if nonzero */
#define DEV_TYPE_SHIFT     2    /* right shift to normalize type bits */
#define FORMAT_DEV_BIT  0x80           /* bit in minor to turn write into format */

/* Miscellaneous. */
#define MAX_ERRORS        6     /* how often to try rd/wt before quitting */
#define MAX_RESULTS       7     /* max number of bytes controller returns */
#define NR_DRIVES         2         /* maximum number of drives */
#define DIVISOR         128         /* used for sector size encoding */
#define SECTOR_SIZE_CODE   2           /* code to say "512" to the controller */
#define TIMEOUT                    500      /* milliseconds waiting for FDC */
#define NT              7     /* number of diskette/drive combinations */
#define UNCALIBRATED      0    /* drive needs to be calibrated at next use */
#define CALIBRATED        1       /* no calibration needed */
#define BASE_SECTOR       1       /* sectors are numbered starting at 1 */
#define NO_SECTOR         0       /* current sector unknown */
#define NO_CYL                     (-1)      /* current cylinder unknown, must seek */
#define NO_DENS                    100       /* current media unknown */
#define BSY_IDLE          0        /* busy doing nothing */
#define BSY_IO            1        /* doing I/O */
#define BSY_WAKEN         2        /* got a wakeup call */
```

Estructuras de Datos.

- La estructura de datos principal que se utiliza en el manejador de disco flexible es **floppy.**
- Dicha estructura es un array de estructuras, una por cada unidad.
- Cada estructura contiene información acerca del estado de su unidad, es decir posición cabeza (cilindro), sector, cabeza direccionada, estado de la calibración,…

```
PRIVATE struct floppy
{     /* estructura de la unidad principal, una entrada por unidad */
  int fl_curcyl;          /* posición lógica del cilindro actual (deseada)*/
  int fl_hardcyl;         /* posición física (saltos de cabezas) del cilindro actual*/
  int fl_cylinder;        /* nº de cilindro direccionado */
  int fl_sector;          /* sector direccionado */
  int fl_head; /* número de cabeza direccionada */
  char fl_calibration;  /* CALIBRATED or UNCALIBRATED */
  char fl_density;        /* NO_DENS = ?, 0 = 360K; 1 = 360K/1.2M; etc.*/
  char fl_class;          /* mapa de bits de las posibles densidades */
```

```
  struct device fl_geom;        /* Geometría del dispositivo */
  struct device fl_part[NR_PARTITIONS];  /* partition's base & size */
} floppy[NR_DRIVES], *f_fp;
```

- Otra estructura utilizada es la estructura **trans**, que contiene los datos de transferencia de cada sector.

```
PRIVATE struct trans
{
  unsigned tr_count;            /* byte count */
  struct iorequest_s *tr_iop;   /* belongs to this I/O request */
  phys_bytes tr_phys;           /* user physical address */
  phys_bytes tr_dma;            /* DMA physical address */
} ftrans[MAX_SECTORS];
```

- Otra estructura, es la que se utiliza para el testeo de la unidad **test_order:**

```
#define b(d)      (1 << (d))        /* bit for density d. */
PRIVATE struct test_order {
                  char    t_density;        /* floppy/drive type */
                  char    t_class;          /* limit drive to this class of densities */
} test_order[NT-1] = {
{ 6,  b(3) | b(6) },                /* 1.44M  {720K, 1.44M} */
{ 1,  b(1) | b(4) | b(5) },         /* 1.2M   {1.2M, 360K, 720K} */
{ 3,  b(2) | b(3) | b(6) },         /* 720K   {360K, 720K, 1.44M} */
{ 4,  b(1) | b(4) | b(5) },         /* 360K   {1.2M, 360K, 720K} */
{ 5,  b(1) | b(4) | b(5) },         /* 720K   {1.2M, 360K, 720K} */
{ 2,  b(2) | b(3) },                /* 360K   {360K, 720K} */
};
```

- La última estructura definida es **f_dtab**, que es el único punto de entrada a este driver, que como ya comentamos, contiene todas las funciones que pueden ser llamadas desde Driver.c.

```
PRIVATE struct driver f_dtab = {
    f_name,         /* current device's name */
    f_do_open, /*open or mount request, sense type of diskette */
    do_nop,         /* nothing on a close */
    do_diocntl,     /* get or set a partitions geometry */
    f_prepare,      /* prepare for I/O on a given minor device */
    f_schedule,     /* precompute cylinder, head, sector, etc. */
    f_finish,       /* do the I/O */
    f_cleanup, /* cleanup before sending reply to user process */
    f_geometry      /* tell the geometry of the diskette */
};
```

Funciones.

- **Floppy_task.**

Es la única función que puede ser referida externamente, sin contar con la de parada, por lo que es la que pasa la estructura f_dtab a driver.c.

Esta función se divide en tres partes:

a) Inicializa el array floppy.

b) Pone en la tabla IRQ el handler asociado y habilita la entrada en dicha tabla. Ahora el sistema operativo, está preparado para recibir interrupciones del floppy.

c) Llama a driver_task, recordar que era la función principal de la parte independiente del dispositivo, pasándole f_dtab (puntos de entrada).

```
PUBLIC void floppy_task()
{   /* Inicializa el array floppy */
        struct floppy *fp;
        for (fp = &floppy[0]; fp < &floppy[NR_DRIVES]; fp++)
        {
                fp->fl_curcyl = NO_CYL;
                fp->fl_density = NO_DENS;
                fp->fl_class = ~0;
        }

/* Inserta handler en IRQ y habilita interrupción */
put_irq_handler(FLOPPY_IRQ, f_handler);
enable_irq(FLOPPY_IRQ);

/* Por último llama a driver_task (bucle) */
driver_task(&f_dtab);
```

- **f_handler.**

Función insertada en la tabla IRQ (Tarea asociada a este dispositivo). Es decir, cuando se produce una interrupción se llama a esta función, la

cual envía un mensaje al proceso que estaba esperando por una interrupción de floppy (interrupt), es decir a nosotros mismos.

```
PRIVATE int f_handler(irq)
int irq;
{
        interrupt(FLOPPY);
        return 0;
}
```

- **f_name.**

Devuelve el nombre del dispositivo que se utiliza actualmente:

```
PRIVATE char *f_name()
{
  static char name[] = "fd3";
  /* Añade fd+'0'+unidad; */
  name[2] = '0' + f_drive;
  return name;
}
```

- **f_cleanup.**

Para comprender el funcionamiento global del problema que existe con el encendido y el apagado de los motores, también deberá mirarse la función Start_Motor.

Esta función Pone en marcha un temporizador para parar todos los motores en unos pocos segundos. Nos encontramos entonces en una cuenta atrás peligrosa. Un temporizador inicializado anteriormente podría acabar antes y parar los motores prematuramente. El problema no se puede solucionar simplemente inicializando el contador después de enviar el mensaje, porque el nuevo temporizador podría acabar antes de que el temporizador sea inicializado. Entonces los motores deberían quedarse encendidos hasta después del siguiente acceso a disco. Esto podría ser solucionado pero requiere muchas tareas adicionales, por lo que suponemos que no es un error grave, sólo perdida de tiempo, y lo implementamos así.

En caso de que exista una solicitud de encendido durante este tiempo, no se parará los motores de las unidades de disco.

```
PRIVATE void f_cleanup()
{
  motor_goal = 0;                    //parar el motor
```

```
  clock_mess(MOTOR_OFF, stop_motor);
}
```

- **Stop_motor.**

Cuando se acaba el temporizador lanzado en f_cleanup => Si ninguna solicitud ha cambiado el estado de los motores (motor_goal), para todos los motores:

```
PRIVATE void stop_motor()
{
  if (motor_goal != motor_status) {
            out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT);
            motor_status = motor_goal;
  }
}
```

- **f_prepare.**

Prepara el dispositivo indicado, para permitir la entrada/salida.

Etapas:

- Eliminar trabajos pendientes después de un error de E/S.
- Inicializar y testear las variables globales.
- Comprobar que no referencia una partición, y en su caso configurar los parámetros.

```
 PRIVATE struct device *f_prepare(device)
int device;
{
  /* Elimina los trabajos pendientes después de un error E/S */
  if (f_count > 0) defuse();
 /* Inicializa las variables y testea valores correctos */
 f_device = device;
 f_drive = device & ~(DEV_TYPE_BITS | FORMAT_DEV_BIT);
 if (f_drive < 0 || f_drive >= NR_DRIVES) return(NIL_DEV);
  f_fp = &floppy[f_drive];
  f_dv = &f_fp->fl_geom;
  d = f_fp->fl_density;
  f_sectors = nr_sectors[d];
  f_must = TRUE;            /* the first transfers must be done */
/* Si referencia a un dispositivo menor */
  if ((device &= DEV_TYPE_BITS) >= MINOR_fd0a)
     f_dv=&f_fp->fl_part[(device-MINOR_fd0a)>>DEV_TYPE_SHIFT];
```

```
 /* Devuelve la base & tamaño */
  return f_dv;
}
```

- **Defuse.**

Invalida las solicitudes pendientes en el vector de transferencia ftrans (Este vector contiene las solicitudes de lectura / escritura de cada sector de la unidad seleccionada).

```
 PRIVATE void defuse()
{
/* Invalidate leftover requests in the transfer array.  */

  struct trans *tp;
  /* Asigna 0 al número de bytes que hay que transmitir por sector */
  for (tp = ftrans; tp < ftrans + MAX_SECTORS; tp++) tp->tr_count = 0;
  f_count = 0; /* Nº de bytes contiguos que hay que transmitir */
}
```

- **f_schedule.**

Planifica la E/S. Modificando el f_trans para la operación deseada y llama a f_finish en caso de que los datos no estén contiguos en el disco.

Pasos:

a) Cancela el temporizador de apagado de motor (existe trabajo que hacer).
b) Configura los parámetro básicos para la transferencia (nº bytes, dirección, ...).
c) Detecta si la operación es de lectura, escritura o formateo.
d) Prepara las posiciones de memoria que se vayan a escribir o leer por sector (No entraremos en mucho detalle).

```
PRIVATE int f_schedule(proc_nr, iop)
int proc_nr;                  /* identificador del proceso que hace la solicitud */
struct iorequest_s *iop;    /* puntero a la solicitud de Lectura o Escritura */
{
  int r, opcode, spanning;
  unsigned long pos;
  unsigned block; /* Seen any 32M floppies lately? */
  unsigned nbytes, count, dma_count;
  phys_bytes user_phys, dma_phys;
  struct trans *tp, *tp0;

  /* Ignora cualquier temporizador de apagado del motor. */
  motor_goal = motor_status;

/*Configura los parámetros básicos */
```

```
 /* Hay que leer o escribir todos estos bytes, en esta posición */
  nbytes = iop->io_nbytes;
  if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
  pos = iop->io_position;
  if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);

   /* Miramos la dirección de la transferencia */
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);

/* Vemos que operación hay que hacer: Read, write or format? */
  opcode = iop->io_request & ~OPTIONAL_IO;

/* Si es formateo comprobamos que sea escritura y algunas cosas más */
if (f_device & FORMAT_DEV_BIT)
{
  if (opcode != DEV_WRITE) return(iop->io_nbytes = EIO);
  if (nbytes != BLOCK_SIZE) return(iop->io_nbytes = EINVAL);
  phys_copy(user_phys + SECTOR_SIZE, ir2phys(&fmt_param),
                                              (phys_bytes) sizeof fmt_param);
  if (fmt_param.sectors_per_cylinder == 0)return (iop->io_nbytes = EIO);
  iop->io_nbytes = nbytes = SECTOR_SIZE;
}

/* A partir de ahora prepara la estructura ftrans para cada sector */

/* Que bloque del disquete y cuanto de cerca está del fin de fichero?  */
  if (pos >= f_dv->dv_size) return(OK);                 /* At EOF */
  if (pos + nbytes > f_dv->dv_size) nbytes = f_dv->dv_size - pos;
  block = (f_dv->dv_base + pos) >> SECTOR_SHIFT;

  spanning = FALSE;         /* por ahora no sabemos si el bloque
                sobrepasa una pista */
do {        /* Mientras queden bytes por planificar de la solicitud */
            count = nbytes;
            /* Envía los bytes preparados en caso de que todos no estén contiguos */
            if (f_count > 0 && block >= f_nexttrack)
            {
                        // The new job leaves the track, finish all gathered jobs
                        if ((r = f_finish()) != OK) return(r);
                        f_must = spanning;
            }
            if (f_count == 0)
            {
              /*La primera vez, calcula el cilindro y la cabeza */
              f_opcode = opcode; /* L/E */
              f_fp->fl_cylinder = block / (NR_HEADS * f_sectors);
              f_fp->fl_hardcyl = f_fp->fl_cylinder * steps_per_cyl[d];
              f_fp->fl_head= (block % (NR_HEADS * f_sectors))/f_sectors;
              f_nexttrack = (f_fp->fl_cylinder * NR_HEADS
                                                  + f_fp->fl_head + 1) * f_sectors;
            }
            /*Calcula el número de bytes que se transmiten contiguos. */
            if (block + (count >> SECTOR_SHIFT) > f_nexttrack)
                        count = (f_nexttrack - block) << SECTOR_SHIFT;
```

```
                // Si la transferencia es DMA calcula los parámetros necesarios y los buffer DMA.
                dma_phys = user_phys;
                dma_count = dma_bytes_left(dma_phys);
               #if _WORD_SIZE > 2 if (dma_phys >= 0x1000000) dma_count = 0; #endif

    if (dma_count < count)
    {
       if (dma_count >= SECTOR_SIZE) {count = dma_count    &~SECTOR_MASK;
    }
     else {
            count = SECTOR_SIZE; dma_phys = tmp_phys;
         }
}
  /* Se rellena la estructura f_trans para los bloques contiguos */
  tp = tp0 = &ftrans[block % f_sectors];
  block += count >> SECTOR_SHIFT;
  nbytes -= count;          f_count += count;
  if (!(iop->io_request & OPTIONAL_IO)) f_must = TRUE;
  do {
            tp->tr_count = count;
            tp->tr_iop = iop;
            tp->tr_phys = user_phys;   tp->tr_dma = dma_phys;
            tp++;
            user_phys += SECTOR_SIZE;
            dma_phys += SECTOR_SIZE;
             count -= SECTOR_SIZE;
     } while (count > 0);
     spanning = TRUE;       /* the rest of the block may span a track */
  } while (nbytes > 0);
  return(OK);}// Fin función Schedule
```

- **f_finish.**

Lleva a cabo la operación de lectura, escritura y formateo que fue preparada en la fase f_schedule;

Pasos:

a) Enciende el motor si es necesario.
b) Establece los parámetros de transferencia (kbps,...).
c) En 2 Bucles anidados; el 1º para repetir si ocurren errores y en el segundo:

   1) Coloca la cabeza(brazo), si no está ya colocada.
   2) Extrae el siguiente trabajo de f_trans.
   3) Si es una escritura, copiamos el buffer de la memoria del usuario a la memoria del DMA (phys_copy).
      Inicializamos el DMA (dma_setup) y Realizamos la transferencia (f_transfer).
   4) Si es una lectura copiamos el buffer de la memoria del DMA a la memoria del usuario (phys_copy).

5) Si se hace con éxito salimos y si no lo reintentamos.

```
PRIVATE int f_finish()
{
 /* Lleva a cabo la solicitud de L/E preparada en ftrans[].  */
 struct floppy *fp = f_fp;
 struct trans *tp; int r, errors;

 start_motor(); /*Mira si el motor está encendido y si no lo  esta lo enciende*/
/* Establece los parámetros de transferencia (kbps) */
 if (pc_at) out_byte(FDC_RATE, rate[d]);
 do {
      errors = 0;
      for (;;)
      {
            /*Posiciona el brazo, si es necesario */
             r = seek(fp);
            /*Extrae e l siguiente trabajo de ftrans hasta que se llegue a f_count*/
             if (fp->fl_sector != NO_SECTOR) {
             for (;;) {
                        if (fp->fl_sector >= BASE_SECTOR + f_sectors)
                   fp->fl_sector = BASE_SECTOR;
                        tp = &ftrans[fp->fl_sector - BASE_SECTOR];
                        if (tp->tr_count > 0) break;
                        fp->fl_sector++;
                        }// No se debe transmitir mas de f_count bytes.
            if (tp->tr_count > f_count) tp->tr_count = f_count;
      }
/* Si es una escritura, copiamos el buffer de la memoria del usuario a la   DMA. */
if (r == OK && tp->tr_dma == tmp_phys && f_opcode == DEV_WRITE)
{
    phys_copy(tp->tr_phys, tp->tr_dma,
                  (phys_bytes) tp-> tr_count);              }
/*Realiza la transferencia */
if (r == OK)
{
            dma_setup(tp);
            r = f_transfer(fp, tp);
 }
 /* Si es una lectura, copiamos el buffer de la memoria del DMA a la del usuario */
if (r == OK && tp->tr_dma == tmp_phys && f_opcode == DEV_READ)
{
  phys_copy(tp->tr_dma, tp->tr_phys,
                                        (phys_bytes) tp->t r _ c ount);
}
/* Si se hace con éxito salimos */
if (r == OK) break;
            /*No se reintenta si el disco está protegido */
if (err_no_retry(r) || errors++ == MAX_ERRORS)
{
    if (fp->fl_sector != 0) tp->tr_iop->io_nbytes = EIO;
    return(EIO);
}
```

```
            } // for de reintentos
    f_count -= tp->tr_count;
    tp->tr_iop->io_nbytes -= tp->tr_count;
  } while (f_count > 0);
  defuse(); /* Defuse the leftover partial jobs. */
  return(OK);
} //Final de la función f_finish
```

- **Seek.**

Esta función emite el comando que coloca el brazo (cilindro) en la unidad indicada a menos que el brazo ya esté colocado en la pista correcta.

Los pasos que da son:

a) Comprueba si ya está colocado en la pista correcta, recalibrando en el caso de que sea necesario.
b) Si no está posicionado, lanza el comando y espera por la interrupción.
c) Si tras un corto tiempo no se recibe señal => Error señal.
d) En caso contrario, verificamos estado unidad (ST0).
e) Esperamos un cierto tiempo en caso de que se necesite formateo.

```
PRIVATE int seek(fp)
struct floppy *fp;          /* pointer to the drive struct */
{
  int r;
  message mess;
  /* Comprueba si ya está posicionado en la pista correcta (el brazo). */
  if (fp->fl_calibration == UNCALIBRATED)
          if (recalibrate(fp) != OK) return(ERR_SEEK);
  if (fp->fl_curcyl == fp->fl_hardcyl) return(OK);
/* Lanza comando de posicionamiento. */
  fdc_out(FDC_SEEK);
  fdc_out((fp->fl_head << 2) | f_drive);
  fdc_out(fp->fl_hardcyl);
  if (need_reset) return(ERR_SEEK);

/* Espera a que responda el controlaor físico (interrupción) */
  if (f_intr_wait() != OK) return(ERR_TIMEOUT);
/* Si se recibió la interrupción.  Comprueba el estado de la unidad. */
   fdc_out(FDC_SENSE);  /* probe FDC to make it return status */
   r = fdc_results();                  /* get controller status bytes */
   if (r != OK || (f_results[ST0] & ST0_BITS) != SEEK_ST0
                                       || f_results[ST1] != fp->fl_hardcyl)
   {
          return(ERR_SEEK); /* Necesita recalibrado */
   }
/* Si se necesita formateo, le da un poco de tiempo para que se solucione */
if (f_device & FORMAT_DEV_BIT) {
```

```
            clock_mess(2, send_mess);
            receive(CLOCK, &mess);
    }
    fp->fl_curcyl = fp->fl_hardcyl;
    return(OK);}
```

- **Recalibrate.**

El FDC no tiene manera de determinar la posición absoluta de su brazo (pista). En lugar de ello, el avanza el brazo una pista cada vez y mantiene un registro de donde piensa él que está (en el software). Sin embargo, después de un SEEK, el hw lee información de el disquete diciéndole dónde el brazo está actualmente. Si el brazo está en un lugar erróneo se hace un recalibrado que fuerza el movimiento del brazo al cilindro 0. De esta manera el controlador puede sincronizar su posición con la real.

Los pasos dados son:

a) Lo primero que hay que hacer es arrancar el motor. No se puede realizar el recalibrado con el motor parado.
b) A continuación se envía el comando de recalibrado con los parámetros oportunos.
c) Vemos si hay errores. En caso de que el recalibrado haya fallado será necesario resetear el FDC, lo cual lo indicamos poniendo need_reset=TRUE y luego retornamos.
d) Si no se han producido errores marcamos la unidad como CALIBRATED y retornamos OK.

```
PRIVATE int recalibrate(fp)
struct floppy *fp;/* puntero a la estructura de la unidad */
{
  int r;

  start_motor();   /* arranca el motor de la unidad */
  fdc_out(FDC_RECALIBRATE); /*manda el comando al controlador */
  fdc_out(f_drive);
  if (need_reset) return(ERR_SEEK); /* comprueba los errores */
  if (f_intr_wait() != OK) return(ERR_TIMEOUT);
  fdc_out(FDC_SENSE);
  r = fdc_results();
  fp->fl_curcyl = NO_CYL;          /* force a SEEK next time */
  if (r != OK || (f_results[ST0] & ST0_BITS) != SEEK_ST0 || f_results[ST_PCN]   != 0)
      {/* controller would not respond */
        need_reset = TRUE;
        return(ERR_RECALIBRATE);
       }
else {
            /* unidad calibrada correctamente */
            fp->fl_calibration = CALIBRATED;
            return(OK);
```

```
  }
}
```

- **fdc_out.**

Esta función envía un byte al FDC, por el puerto correspondiente.

La función no es del todo trivial, ya que primeramente hay que comprobar si está listo el controlador para no perder el tiempo.

Para ello primero miramos que no se necesite resetear. Luego hay que comprobar que el bit Request for Master está a 1 y el bit I/O Direction a 0, como se mencionó en la fase de comando, que se corresponden con las máscaras MASTER y DIRECTION (preparado y dirección CPU -> Controlador).

Si después de un tiempo no hemos podido transmitir suponemos que el FDC está descontrolado y que es necesario hacer un reset del chip, need_reset=TRUE. En caso contrario enviamos al puerto FDC_DATA el byte.

```
PRIVATE void fdc_out(val)
int val;              /* escribe este byte en el controlador de disco floppy */
{  struct milli_state ms;

  if (need_reset) return;   /* si es necesario resetear, salimos (no hacemos nada) */
  milli_start(&ms); /* Testeamos los bits (varias oportunidades) */
  while ((in_byte(FDC_STATUS) & (MASTER | DIRECTION)) != (MASTER | 0))
  {
    if (milli_elapsed(&ms) >= TIMEOUT)
     {  /* No esperamos más, hay que resetear */
                    need_reset = TRUE;
                    return;
     }
  }
  out_byte(FDC_DATA, val); /* todo correcto, enviamos el byte */
}
```

- **f_intr_wait.**

Esta función espera por una interrupción, pero no eternamente.

```
PRIVATE int f_intr_wait()
{
  message mess;
   f_busy = BSY_IO;
   clock_mess(WAKEUP, f_timeout);
   receive(HARDWARE, &mess);
   if (f_busy == BSY_WAKEN) /* Lo que devuelve f_timeout = ERROR */
```

```
 {
        f_reset();
        return(ERR_TIMEOUT);
 }
 f_busy = BSY_IDLE;
 return(OK);
}
```

- **f_time_out.**

Cuando hemos esperado mucho tiempo por una respuesta del controlador (lo más probable que no exista un disco en la unidad), se llama a esta rutina.

La función activa el flag (para indicar este estado) y lanza la interrupción de FLOPPY, luego se tendrá que chequear el flag.

Es una de las mejoras de la nueva versión.

```
PRIVATE void f_timeout()
{
   if (f_busy == BSY_IO) /* No llegó respuesta, inicializada así antes
                            de llamar a clock_mess */
   {
        f_busy = BSY_WAKEN; /*despertado porque no responde */
        interrupt(FLOPPY);
   }
}
```

- **fdc_results.**

Esta función extrae el resultado del controlador después de una operación, luego habilita las interrupciones floppy.

Cuando se terminó la fase de ejecución el FDC activó la INT 1. Una vez que se ha leído el primer byte de datos durante la Fase de resultados, la interrupción es automáticamente reseteada INT=0.

La función extrae resultados mientras el flag CTL_BUSY (indica que hay una opearción de lectura / escritura en curso) esté activado. Si durante la extracción de los resultados finaliza el temporizador entendemos que es necesario resetear el FDC.

```
PRIVATE int fdc_results()
{
 int result_nr, status;  struct milli_state ms;
 result_nr = 0;  milli_start(&ms);
 do { /* Leemos los registros de datos durante la fase de resultados */
     status = in_byte(FDC_STATUS) & (MASTER | DIRECTION | CTL_BUSY);
   if (status == (MASTER | DIRECTION | CTL_BUSY))
```

```
    {
      if (result_nr >= MAX_RESULTS) break;     /* too many results  (7 max)*/
      f_results[result_nr++] = in_byte(FDC_DATA);
     continue;
    }
   if (status == MASTER) {          /* all read bit status != CTL_BUSY*/
                    enable_irq(FLOPPY_IRQ);
                    return(OK);         /* only good exit */
   }
  } while (milli_elapsed(&ms) < TIMEOUT);
  need_reset = TRUE;       /* se acabó el tiempo => hay que resetear el controlador*/
  enable_irq(FLOPPY_IRQ); /* habilita las interrupciones */
  return(ERR_STATUS);
}
```

- **dma_setup.**

En esta función se inicializan los registros del controlador DMA para que controle la transferencia de los datos.

Los pasos dados son:

a) Resetea el controlador DMA.
b) Establece el modo de funcionamiento (L/E).
c) Carga la dirección física.
d) Carga el contador de bytes a transferir.
e) Inicia el controlador.

```
PRIVATE void dma_setup(tp)
struct trans *tp;
{ /* Resetea el controlador de DMA*/
  out_byte(DMA_INIT, DMA_RESET_VAL);          /* reset the dma controller */
  out_byte(DMA_FLIPFLOP, 0);                  /* write anything to reset it */

   /* Establece el modo de funcionamiento */
    out_byte(DMA_MODE, f_opcode == DEV_WRITE ? DMA_WRITE : DMA_READ);
    out_byte(DMA_ADDR, (int) tp->tr_dma >>  0);
  /* Carga la dirección física */
   out_byte(DMA_ADDR, (int) tp->tr_dma >>  0);
   out_byte(DMA_ADDR, (int) tp->tr_dma >>  8);
   out_byte(DMA_TOP, (int) (tp->tr_dma >> 16));
 /* Carga el contador de bytes a transferir */
   out_byte(DMA_COUNT, (tp->tr_count - 1) >> 0);
   out_byte(DMA_COUNT, (tp->tr_count - 1) >> 8);
 /* Inicia el controlador DMA */
   out_byte(DMA_INIT, 2);          /* some sort of enable */
}
```

- **start_motor.**

Controlar los motores de los floppys es una gran molestia. Si un motor está apagado, hay que encenderlo, lo cual toma ½ segundo. No puedes dejarlo encendido todo el tiempo porque supondría un desgaste del disco. Sin embargo, si paras el motor después de cada operación el rendimiento del sistema sería muy bajo. El compromiso tomado aquí es dejarlo encendido durante unos pocos segundos después de cada operación. Si no es iniciada ninguna operación nueva, el temporizador se acaba y se para el motor. Esto se controla en el DOR (4 motores).

motor_bit : máscara de selección del motor de la unidad correspondiente
running : Distinto de cero si el motor ya está en marcha.
motor_goal : estado en que queremos que estén los motores.

Los pasos que da son:

a) Calcula el bit del DOR que pertenece a la unidad seleccionada actualmente y marca dicho bit a 1 para que se encienda motor de la disquetera.
b) Envía el DOR por el puerto y establece el nuevo estado del motor (encendido).
c) Si no estaba ya encendido, inicia un contador y espera a que finalice, en caso contrario no hace falta que espere.

```
PRIVATE void start_motor()
{
  int motor_bit, running;
  message mess;
  /* pone el bit a uno para que se encienda el motor de la disquetera */
  motor_bit = 1 << f_drive;                    /* bit mask for this drive */
  running = motor_status & motor_bit;          /* 1 if this motor is running */
  motor_goal = motor_status | motor_bit; /* want this drive running too */
  out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT | f_drive);
  motor_status = motor_goal;
  if (running) return;          /* el motor ya estaba encendido */
  clock_mess(mtr_setup[d], send_mess);  /* motor was not running */
  receive(CLOCK, &mess);                    /* wait for clock interrupt */
}
```

- **send_mess.**

Esta función es llamada cuando las tareas de reloj han terminado en la puesta en marcha de los motores.

```
PRIVATE void send_mess()
{
 message mess;
 send(FLOPPY, &mess);
}
```

- **floppy_stop.**

Junto con floppy_task es la única función que puede ser llamada desde otros módulos, simplemente hace una llamada para que se paren todos los motores, inicializando primero la variable de estado deseado de los motores (motor_goal).

```
PUBLIC void floppy_stop()
{
        motor_goal = 0;
        stop_motor();
}
```

- **f_transfer.**

Esta función se encarga de enviar al FDC los comandos necesarios para que se realice el formateo o la operación de lectura / escritura, de un bloque. ¡ojo primeramente debe estar posicionada !.

Para ello los pasos que da son:

a) Mira si el motor está apagado o la unidad no está calibrada. En tal caso sale.
b) A continuación envía el comando correspondiente (FDC_FORMAT, FDC_READ, FDC_WRITE) y los datos necesarios para completar el mismo (pista, sector, cabeza, ...).
c) Ahora espera a que el FDC produzca una interrupción para indicarle que ha terminado la ejecución. Si no se produce dará un timeout y retornará devolviendo un error.
d) Finalmente se obtiene el status del controlador y se chequean los posibles errores.

```
PRIVATE int f_transfer(fp, tp)
struct floppy *fp;      /* puntero a la estructura de la unidad */
struct trans *tp;       /* puntero a la estructura de transferencia */
{
 int r, s;
 /* Comprueba si la unidad está calibrada y el motor ON */
 if  (fp->fl_calibration == UNCALIBRATED) return(ERR_TRANSFER);
 if  ((motor_status & (1 << f_drive)) == 0) return(ERR_TRANSFER);
```

```
/* Enviamos el comando correspondiente:
 if (f_device & FORMAT_DEV_BIT)  { /* En caso de formateo */
      fdc_out(FDC_FORMAT);
      fdc_out((fp->fl_head << 2) | f_drive);
      fdc_out(fmt_param.sector_size_code);
      fdc_out(fmt_param.sectors_per_cylinder);
      fdc_out(fmt_param.gap_length_for_format);
      fdc_out(fmt_param.fill_byte_for_format);
}

else { / * Si no es una lectura o una escritura */
      fdc_out(f_opcode == DEV_WRITE ? FDC_WRITE : FDC_READ);
      fdc_out((fp->fl_head << 2) | f_drive);
      fdc_out(fp->fl_cylinder);
      fdc_out(fp->fl_head);
      fdc_out(fp->fl_sector);
      fdc_out(SECTOR_SIZE_CODE);
      fdc_out(f_sectors);
      fdc_out(gap[d]);  /* hueco sector */
      fdc_out(DTL);    /* longitud de los datos */
 }

/* Espera por la interrupción */
if (need_reset) return(ERR_TRANSFER);
if (f_intr_wait() != OK) return(ERR_TIMEOUT);

/* obtiene el estado del controlador y verifica los errores. */
 r = fdc_results();
 if (r != OK) return(r);
 if (f_results[ST1] & WRITE_PROTECT)
 {
     printf("%s: diskette is write protected.\n", f_name());
     return(ERR_WR_PROTECT);
 }
 if ((f_results[ST0] & ST0_BITS) != TRANS_ST0) return(ERR_TRANSFER);
 if (f_results[ST1] | f_results[ST2]) return(ERR_TRANSFER);
 if (f_device & FORMAT_DEV_BIT) return(OK);

 /* Compara el nº de sectores transmitidos con el esperado. */
 s = (f_results[ST_CYL] - fp->fl_cylinder) * NR_HEADS *  f_sectors;
 s += (f_results[ST_HEAD] - fp->fl_head) * f_sectors;
 s += (f_results[ST_SEC] - fp->fl_sector);
 if ((s << SECTOR_SHIFT) != tp->tr_count)
                          return(ERR_TRANSFER);
 /* El próximo sector es: */
 fp->fl_sector = f_results[ST_SEC];
 return(OK);
}
```

- **f_reset.**

Esta función se llama cuando el FDC ha perdido el control. Su función es "resetear" el controlador.

Los pasos dados son:

a) Marcar el flag need_reset con FALSE.
b) Luego, lo primero que hace es enviar un 0 al DOR con lo cual reseteamos el FDC.
c) A continuación habilitamos el chip escribiendo en el mismo registro ENABLE_INT y esperamos por la interrupción de RESET.
d) El FDC (físico) soporta 4 unidades y retorna un resultado por cada una de ellas. Entonces recogemos los resultados para las cuatros unidades.
e) Marcamos las unidades (2 para este driver) como UNCALIBRATED.

```
PRIVATE void f_reset()
{
 int i;
 message mess;
 /* Marcar el flag need_reset con FALSE. */
 need_reset = FALSE;
/* Reseteamos el controlador */
 lock();                          /* para que funcione */
 motor_status = 0;
 motor_goal = 0;
 out_byte(DOR, 0);
 /* Habilitamos el chip */
 out_byte(DOR, ENABLE_INT);
 unlock();
 receive(HARDWARE, &mess);

/* Recogemos los resultados para las cuatro unidades */
 for (i = 0; i < 4; i++) {
      fdc_out(FDC_SENSE);    /* probe FDC to make it return status */
      (void) fdc_results();        /* flush controller */
  }
/* Marcamos las unidades como UNCALIBRATED */
 for (i = 0; i < NR_DRIVES; i++)        /* clear each drive */
      floppy[i].fl_calibration = UNCALIBRATED;

 /* Los parámetros de tiempo deben ser especificados de nuevo. */
 current_spec1 = 0;
}
```

- **f_do_open.**

La función f_do_open se encarga de la petición de montar o abrir. En esta función se busca el tipo de unidad/disco presente, si todavía no se conoce.

Una unidad se puede determinar (en Minix 2.0) en el campo DEVICE de dos formas:

Indicar el tipo de unidad/disco directamente con uno de los 14 dispositivos diferentes que aparecen en /dev, p.ej. /dev/pc0 (disco de 360K 5.25" en la primera unidad) o /dev/PS1 (disco de 1.44MB 3.5" en la segunda unidad).

Indicar sólo la unidad, p.ej /dev/fd0 o /dev/fd1, y que el driver se encargue de detectar el tipo de diskette.

Entonces los pasos dados son:

a) Intenta preparar el dispositivo con los parámetros actuales.
b) Si no hay errores, entonces no hace falta encontrar el tipo de unidad.
c) Si no se especificó la unidad directamente hay que ir probando todos los tipos, hasta que una lectura con los parámetros de prueba sea correcta.

```
PRIVATE int f_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;                         /* pointer to open message */
{
  int dtype;
  struct test_order *top;
  /* intenta preparar el dispositivo con los parámetros actuales */
  if (f_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
  dtype = f_device & DEV_TYPE_BITS;      /* get density from minor dev */
  if (dtype >= MINOR_fd0a) dtype = 0;
  if (dtype != 0) {
        dtype = (dtype >> DEV_TYPE_SHIFT) - 1;
        if (dtype >= NT) return(ENXIO);
        f_fp->fl_density = dtype;
        f_fp->fl_geom.dv_size = (long) nr_blocks[dtype] << SECTOR_SHIFT;
        return(OK); }
  if (f_device & FORMAT_DEV_BIT) return(EIO);  /* Can't format /dev/fdx */
/* Comprobamos si no hace falta hacer el test con los distintos tipos */
  if (motor_status & (1 << f_drive)) return(OK);
  if (f_fp->fl_density != NO_DENS && test_read(f_fp->fl_density) == OK)
        return(OK);
/* Hay que hacer el test */
  for (top = &test_order[0]; top < &test_order[NT-1]; top++)
  {
        dtype = top->t_density;
        if (!(f_fp->fl_class & (1 << dtype))) continue;
        if (test_read(dtype) == OK)
        {
              f_fp->fl_class &= top->t_class;
              return(OK);
        }
```

```
            if (f_busy == BSY_WAKEN) break;
        }
        f_fp->fl_density = NO_DENS;
        return(EIO);
    }
```

- **test_read.**

Esta función lo único que hace es intentar leer el sector más alto en la pista segunda. Es empleada para detectar si la configuración del disco insertado en la disquetera corresponde con la que posee la controladora.

```
PRIVATE int test_read(density)
int density;
{
  message m;  int r, device;

  f_fp->fl_density = density;
  device = ((density + 1) << DEV_TYPE_SHIFT) + f_drive;
  f_fp->fl_geom.dv_size = (long) nr_blocks[density] << SECTOR_SHIFT;
  m.m_type = DEV_READ;  m.DEVICE = device;
  m.PROC_NR = FLOPPY;  m.COUNT = SECTOR_SIZE;
  m.POSITION = (long) test_sector[density] * SECTOR_SIZE;
  m.ADDRESS = (char *) tmp_buf;
  r = do_rdwt(&f_dtab, &m);  if (r != SECTOR_SIZE) return(EIO);
  partition(&f_dtab, f_drive, P_FLOPPY);  return(OK);
}
```

- **f_geometry.**

Última función de la estructura f_dtab. Nos devuelve la geometría de la unidad floppy (cilindros,cabezas,sectores).

```
PRIVATE void f_geometry(entry)
struct partition *entry;
{
  entry->cylinders = nr_blocks[d] / (NR_HEADS * f_sectors);
  entry->heads = NR_HEADS;
  entry->sectors = f_sectors;
}
```

- **do_nop y do_diocntl**

No están implementadas en este driver, lo mas seguro es que no las utiliza este driver.

- **Código Completo de Floppy.c**

```
/* This file contains the device dependent part of the driver for the
Floppy
 * Disk Controller (FDC) using the NEC PD765 chip.
 *
 * The file contains one entry point:
 *
 *   floppy_task: main entry when system is brought up
 *   floppy_stop: stop all activity
 *
 *  Changes:
 *    27 Oct. 1986 by Jakob Schripsema: fdc_results fixed for 8 MHz
 *    28 Nov. 1986 by Peter Kay: better resetting for 386
 *    06 Jan. 1988 by Al Crew: allow 1.44 MB diskettes
 *            1989 by Bruce Evans: I/O vector to keep up with 1-1
interleave
 *    13 May  1991 by Don Chapman: renovated the errors loop.
 *          1991 by Bruce Evans: len[] / motors / reset / step rate / ...
 *    14 Feb  1992 by Andy Tanenbaum: check drive density on opens only
 *    27 Mar  1992 by Kees J. Bot: last details on density checking
 *    04 Apr  1992 by Kees J. Bot: device dependent/independent split
 */

#include "kernel.h"
#include "driver.h"
#include "drvlib.h"
#include <ibm/diskparm.h>

/* I/O Ports used by floppy disk task. */
#define DOR             0x3F2  /* motor drive control bits */
#define FDC_STATUS      0x3F4  /* floppy disk controller status register */
#define FDC_DATA        0x3F5  /* floppy disk controller data register */
#define FDC_RATE        0x3F7  /* transfer rate register */
#define DMA_ADDR        0x004  /* port for low 16 bits of DMA address */
#define DMA_TOP         0x081  /* port for top 4 bits of 20-bit DMA addr */
#define DMA_COUNT       0x005  /* port for DMA count (count =  bytes - 1)
*/
#define DMA_FLIPFLOP    0x00C  /* DMA byte pointer flip-flop */
#define DMA_MODE        0x00B  /* DMA mode port */
#define DMA_INIT        0x00A  /* DMA init port */
#define DMA_RESET_VAL    0x06

/* Status registers returned as result of operation. */
#define ST0              0x00  /* status register 0 */
#define ST1              0x01  /* status register 1 */
#define ST2              0x02  /* status register 2 */
#define ST3              0x00  /* status register 3 (return by DRIVE_SENSE)
*/
#define ST_CYL           0x03  /* slot where controller reports cylinder */
#define ST_HEAD          0x04  /* slot where controller reports head */
#define ST_SEC           0x05  /* slot where controller reports sector */
#define ST_PCN           0x01  /* slot where controller reports present cyl
*/
```

```
/* Fields within the I/O ports. */
/* Main status register. */
#define CTL_BUSY        0x10  /* bit is set when read or write in progress
*/
#define DIRECTION       0x40  /* bit is set when reading data reg is valid
*/
#define MASTER          0x80  /* bit is set when data reg can be accessed
*/

/* Digital output port (DOR). */
#define MOTOR_SHIFT        4  /* high 4 bits control the motors in DOR */
#define ENABLE_INT      0x0C  /* used for setting DOR port */

/* ST0. */
#define ST0_BITS        0xF8  /* check top 5 bits of seek status */
#define TRANS_ST0       0x00  /* top 5 bits of ST0 for READ/WRITE */
#define SEEK_ST0        0x20  /* top 5 bits of ST0 for SEEK */

/* ST1. */
#define BAD_SECTOR      0x05  /* if these bits are set in ST1, recalibrate
*/
#define WRITE_PROTECT   0x02  /* bit is set if diskette is write protected
*/

/* ST2. */
#define BAD_CYL         0x1F  /* if any of these bits are set, recalibrate
*/

/* ST3 (not used). */
#define ST3_FAULT       0x80  /* if this bit is set, drive is sick */
#define ST3_WR_PROTECT  0x40  /* set when diskette is write protected */
#define ST3_READY       0x20  /* set when drive is ready */

/* Floppy disk controller command bytes. */
#define FDC_SEEK        0x0F  /* command the drive to seek */
#define FDC_READ        0xE6  /* command the drive to read */
#define FDC_WRITE       0xC5  /* command the drive to write */
#define FDC_SENSE       0x08  /* command the controller to tell its status
*/
#define FDC_RECALIBRATE 0x07  /* command the drive to go to cyl 0 */
#define FDC_SPECIFY     0x03  /* command the drive to accept params */
#define FDC_READ_ID     0x4A  /* command the drive to read sector identity
*/
#define FDC_FORMAT      0x4D  /* command the drive to format a track */

/* DMA channel commands. */
#define DMA_READ        0x46  /* DMA read opcode */
#define DMA_WRITE       0x4A  /* DMA write opcode */

/* Parameters for the disk drive. */
#define HC_SIZE         2880  /* # sectors on largest legal disk (1.44MB)
*/
#define NR_HEADS        0x02  /* two heads (i.e., two tracks/cylinder) */
#define MAX_SECTORS       18  /* largest # sectors per track */
#define DTL             0xFF  /* determines data length (sector size) */
#define SPEC2           0x02  /* second parameter to SPECIFY */
#define MOTOR_OFF       3*HZ  /* how long to wait before stopping motor */
```

```
#define WAKEUP          2*HZ  /* timeout on I/O, FDC won't quit. */

/* Error codes */
#define ERR_SEEK         (-1) /* bad seek */
#define ERR_TRANSFER     (-2) /* bad transfer */
#define ERR_STATUS       (-3) /* something wrong when getting status */
#define ERR_READ_ID      (-4) /* bad read id */
#define ERR_RECALIBRATE  (-5) /* recalibrate didn't work properly */
#define ERR_DRIVE        (-6) /* something wrong with a drive */
#define ERR_WR_PROTECT   (-7) /* diskette is write protected */
#define ERR_TIMEOUT      (-8) /* interrupt timeout */

/* No retries on some errors. */
#define err_no_retry(err)    ((err) <= ERR_WR_PROTECT)

/* Encoding of drive type in minor device number. */
#define DEV_TYPE_BITS   0x7C  /* drive type + 1, if nonzero */
#define DEV_TYPE_SHIFT     2  /* right shift to normalize type bits */
#define FORMAT_DEV_BIT  0x80  /* bit in minor to turn write into format */

/* Miscellaneous. */
#define MAX_ERRORS         6  /* how often to try rd/wt before quitting */
#define MAX_RESULTS        7  /* max number of bytes controller returns */
#define NR_DRIVES          2  /* maximum number of drives */
#define DIVISOR          128  /* used for sector size encoding */
#define SECTOR_SIZE_CODE   2  /* code to say "512" to the controller */
#define TIMEOUT          500  /* milliseconds waiting for FDC */
#define NT                 7  /* number of diskette/drive combinations */
#define UNCALIBRATED       0  /* drive needs to be calibrated at next use
*/
#define CALIBRATED         1  /* no calibration needed */
#define BASE_SECTOR        1  /* sectors are numbered starting at 1 */
#define NO_SECTOR          0  /* current sector unknown */
#define NO_CYL          (-1) /* current cylinder unknown, must seek */
#define NO_DENS          100  /* current media unknown */
#define BSY_IDLE     0  /* busy doing nothing */
#define BSY_IO             1  /* doing I/O */
#define BSY_WAKEN    2  /* got a wakeup call */

/* Variables. */
PRIVATE struct floppy {       /* main drive struct, one entry per drive */
  int fl_curcyl;        /* current cylinder */
  int fl_hardcyl;       /* hardware cylinder, as opposed to: */
  int fl_cylinder;            /* cylinder number addressed */
  int fl_sector;        /* sector addressed */
  int fl_head;                /* head number addressed */
  char fl_calibration;        /* CALIBRATED or UNCALIBRATED */
  char fl_density;            /* NO_DENS = ?, 0 = 360K; 1 = 360K/1.2M;
etc.*/
  char fl_class;        /* bitmap for possible densities */
  struct device fl_geom;      /* Geometry of the drive */
  struct device fl_part[NR_PARTITIONS];  /* partition's base & size */
} floppy[NR_DRIVES], *f_fp;

/* Gather transfer data for each sector. */
PRIVATE struct trans {        /* precomputed transfer params */
  unsigned tr_count;          /* byte count */
```

```
  struct iorequest_s *tr_iop; /* belongs to this I/O request */
  phys_bytes tr_phys;         /* user physical address */
  phys_bytes tr_dma;          /* DMA physical address */
} ftrans[MAX_SECTORS];

PRIVATE unsigned f_count;     /* this many bytes to transfer */
PRIVATE unsigned f_nexttrack; /* don't do blocks above this */
PRIVATE int motor_status;     /* bitmap of current motor status */
PRIVATE int motor_goal;       /* bitmap of desired motor status */
PRIVATE int need_reset;       /* set to 1 when controller must be reset */
PRIVATE int d;                /* diskette/drive combination */
PRIVATE int f_drive;          /* selected drive */
PRIVATE int f_device;         /* selected minor device */
PRIVATE int f_opcode;         /* DEV_READ or DEV_WRITE */
PRIVATE int f_sectors;        /* sectors per track of the floppy */
PRIVATE int f_must;           /* must do part of the next track? */
PRIVATE int f_busy;           /* BSY_IDLE, BSY_IO, BSY_WAKEN */
PRIVATE int current_spec1;    /* latest spec1 sent to the controller */
PRIVATE struct device *f_dv;  /* device's base and size */
PRIVATE struct disk_parameter_s fmt_param; /* parameters for format */
PRIVATE char f_results[MAX_RESULTS];/* the controller can give lots of
output */


/* Seven combinations of diskette/drive are supported.
 *
 * # Drive  diskette  Sectors  Tracks  Rotation Data-rate  Comment
 * 0  360K    360K      9       40     300 RPM  250 kbps   Standard PC
DSDD
 * 1  1.2M    1.2M     15       80     360 RPM  500 kbps   AT disk in AT
drive
 * 2  720K    360K      9       40     300 RPM  250 kbps   Quad density
PC
 * 3  720K    720K      9       80     300 RPM  250 kbps   Toshiba, et
al.
 * 4  1.2M    360K      9       40     360 RPM  300 kbps   PC disk in AT
drive
 * 5  1.2M    720K      9       80     360 RPM  300 kbps   Toshiba in AT
drive
 * 6  1.44M   1.44M    18     80     300 RPM  500 kbps   PS/2, et al.
 *
 * In addition, 720K diskettes can be read in 1.44MB drives, but that
does
 * not need a different set of parameters.  This combination uses
 *
 * X  1.44M   720K      9     80     300 RPM  250 kbps   PS/2, et al.
 */
PRIVATE char gap[NT] =
      {0x2A, 0x1B, 0x2A, 0x2A, 0x23, 0x23, 0x1B}; /* gap size */
PRIVATE char rate[NT] =
      {0x02, 0x00, 0x02, 0x02, 0x01, 0x01, 0x00}; /* 2=250,1=300,0=500
kbps*/
PRIVATE char nr_sectors[NT] =
      {9,    15,   9,    9,    9,    9,    18};   /* sectors/track */
PRIVATE int nr_blocks[NT] =
      {720,  2400, 720,  1440, 720,  1440, 2880}; /* sectors/diskette*/
PRIVATE char steps_per_cyl[NT] =
```

```
      {1,    1,    2,    1,    2,    1,     1};   /* 2 = dbl step */
PRIVATE char mtr_setup[NT] =
      {1*HZ/4,3*HZ/4,1*HZ/4,4*HZ/4,3*HZ/4,3*HZ/4,4*HZ/4}; /* in ticks */
PRIVATE char spec1[NT] =
      {0xDF, 0xDF, 0xDF, 0xDF, 0xDF, 0xDF, 0xDF}; /* step rate, etc. */
PRIVATE char test_sector[NT] =
      {4*9,  14,   2*9,  4*9,  2*9,  4*9,  17};   /* to recognize it */

#define b(d)      (1 << (d))  /* bit for density d. */

/* The following table is used with the test_sector array to recognize a
 * drive/floppy combination.  The sector to test has been determined by
 * looking at the differences in gap size, sectors/track, and double
stepping.
 * This means that types 0 and 3 can't be told apart, only the motor
start
 * time differs.  If a read test succeeds then the drive is limited to
the
 * set of densities it can support to avoid unnecessary tests in the
future.
 */

PRIVATE struct test_order {
      char  t_density;  /* floppy/drive type */
      char  t_class;    /* limit drive to this class of densities */
} test_order[NT-1] = {
      { 6,  b(3) | b(6) },          /* 1.44M  {720K, 1.44M} */
      { 1,  b(1) | b(4) | b(5) },   /* 1.2M   {1.2M, 360K, 720K} */
      { 3,  b(2) | b(3) | b(6) },   /* 720K   {360K, 720K, 1.44M} */
      { 4,  b(1) | b(4) | b(5) },   /* 360K   {1.2M, 360K, 720K} */
      { 5,  b(1) | b(4) | b(5) },   /* 720K   {1.2M, 360K, 720K} */
      { 2,  b(2) | b(3) },          /* 360K   {360K, 720K} */
      /* Note that type 0 is missing, type 3 can read/write it too
(alas). */
};

FORWARD _PROTOTYPE( struct device *f_prepare, (int device) );
FORWARD _PROTOTYPE( char *f_name, (void) );
FORWARD _PROTOTYPE( void f_cleanup, (void) );
FORWARD _PROTOTYPE( int f_schedule, (int proc_nr, struct iorequest_s
*iop) );
FORWARD _PROTOTYPE( int f_finish, (void) );
FORWARD _PROTOTYPE( void defuse, (void) );
FORWARD _PROTOTYPE( void dma_setup, (struct trans *tp) );
FORWARD _PROTOTYPE( void start_motor, (void) );
FORWARD _PROTOTYPE( void stop_motor, (void) );
FORWARD _PROTOTYPE( int seek, (struct floppy *fp) );
FORWARD _PROTOTYPE( int f_transfer, (struct floppy *fp, struct trans *tp)
);
FORWARD _PROTOTYPE( int fdc_results, (void) );
FORWARD _PROTOTYPE( int f_handler, (int irq) );
FORWARD _PROTOTYPE( void fdc_out, (int val) );
FORWARD _PROTOTYPE( int recalibrate, (struct floppy *fp) );
FORWARD _PROTOTYPE( void f_reset, (void) );
FORWARD _PROTOTYPE( void send_mess, (void) );
FORWARD _PROTOTYPE( int f_intr_wait, (void) );
FORWARD _PROTOTYPE( void f_timeout, (void) );
```

```
FORWARD _PROTOTYPE( int read_id, (struct floppy *fp) );
FORWARD _PROTOTYPE( int f_do_open, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( int test_read, (int density) );
FORWARD _PROTOTYPE( void f_geometry, (struct partition *entry));


/* Entry points to this driver. */
PRIVATE struct driver f_dtab = {
  f_name,   /* current device's name */
  f_do_open,      /* open or mount request, sense type of diskette */
  do_nop,   /* nothing on a close */
  do_diocntl,     /* get or set a partitions geometry */
  f_prepare,      /* prepare for I/O on a given minor device */
  f_schedule,     /* precompute cylinder, head, sector, etc. */
  f_finish, /* do the I/O */
  f_cleanup,      /* cleanup before sending reply to user process */
  f_geometry      /* tell the geometry of the diskette */
};


/*============================================================================*
 *                    floppy_task                                             *

*============================================================================*/
PUBLIC void floppy_task()
{
/* Initialize the floppy structure. */

  struct floppy *fp;

  for (fp = &floppy[0]; fp < &floppy[NR_DRIVES]; fp++) {
      fp->fl_curcyl = NO_CYL;
      fp->fl_density = NO_DENS;
      fp->fl_class = ~0;
  }

  put_irq_handler(FLOPPY_IRQ, f_handler);
  enable_irq(FLOPPY_IRQ);          /* ready for floppy interrupts */

  driver_task(&f_dtab);
}


/*============================================================================*
 *                    f_prepare                                               *

*============================================================================*/
PRIVATE struct device *f_prepare(device)
int device;
{
/* Prepare for I/O on a device. */

  /* Leftover jobs after an I/O error must be removed */
  if (f_count > 0) defuse();

  f_device = device;
  f_drive = device & ~(DEV_TYPE_BITS | FORMAT_DEV_BIT);
```

```
  if (f_drive < 0 || f_drive >= NR_DRIVES) return(NIL_DEV);

  f_fp = &floppy[f_drive];
  f_dv = &f_fp->fl_geom;
  d = f_fp->fl_density;
  f_sectors = nr_sectors[d];

  f_must = TRUE;  /* the first transfers must be done */

  /* A partition? */
  if ((device &= DEV_TYPE_BITS) >= MINOR_fd0a)
      f_dv = &f_fp->fl_part[(device - MINOR_fd0a) >> DEV_TYPE_SHIFT];

  return f_dv;
}


/*============================================================================*
 *                      f_name                                                *

*============================================================================*/
PRIVATE char *f_name()
{
/* Return a name for the current device. */
  static char name[] = "fd3";

  name[2] = '0' + f_drive;
  return name;
}


/*============================================================================*
 *                      f_cleanup                                             *

*============================================================================*/
PRIVATE void f_cleanup()
{
  /* Start watchdog timer to turn all motors off in a few seconds.
   * There is a race here.  An old watchdog might bite before the
   * new delay is installed, and turn of the motors prematurely.
   * This cannot be solved simply by resetting motor_goal after
   * sending the message, because the new watchdog might bite
   * before motor_goal is reset.  Then the motors would stay on
   * until after the next floppy access.  This could be fixed with
   * extra code (call the clock task twice in some cases).  Or
   * stop_motor() could be replaced by send_mess(), and send a
   * STOP_MOTOR message to be accepted by the clock task.  This
   * would be slower but have the advantage that this comment could
   * be deleted!
   *
   * Since it is not likely and not serious for an old watchdog to
   * bite, accept that possibility for now.  A full solution to the
   * motor madness requires a lots of extra work anyway, such as
   * a separate timer for each motor, and smaller delays for motors
   * that have just been turned off or start faster than the spec.
   * (is there a motor-ready bit?).
   */
```

```
  motor_goal = 0;
  clock_mess(MOTOR_OFF, stop_motor);
}


/*===========================================================================*
 *                      f_schedule                                           *

*===========================================================================*/
PRIVATE int f_schedule(proc_nr, iop)
int proc_nr;                    /* process doing the request */
struct iorequest_s *iop;        /* pointer to read or write request */
{
  int r, opcode, spanning;
  unsigned long pos;
  unsigned block; /* Seen any 32M floppies lately? */
  unsigned nbytes, count, dma_count;
  phys_bytes user_phys, dma_phys;
  struct trans *tp, *tp0;

  /* Ignore any alarm to turn motor off, now there is work to do. */
  motor_goal = motor_status;

  /* This many bytes to read/write */
  nbytes = iop->io_nbytes;
  if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);

  /* From/to this position on disk */
  pos = iop->io_position;
  if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);

  /* To/from this user address */
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);

  /* Read, write or format? */
  opcode = iop->io_request & ~OPTIONAL_IO;
  if (f_device & FORMAT_DEV_BIT) {
      if (opcode != DEV_WRITE) return(iop->io_nbytes = EIO);
      if (nbytes != BLOCK_SIZE) return(iop->io_nbytes = EINVAL);

      phys_copy(user_phys + SECTOR_SIZE, vir2phys(&fmt_param),
                                    (phys_bytes) sizeof fmt_param);

      /* Check that the number of sectors in the data is reasonable, to
       * avoid division by 0.  Leave checking of other data to the FDC.
       */
      if (fmt_param.sectors_per_cylinder == 0)
            return(iop->io_nbytes = EIO);

      /* Only the first sector of the parameters now needed. */
      iop->io_nbytes = nbytes = SECTOR_SIZE;
  }

  /* Which block on disk and how close to EOF? */
  if (pos >= f_dv->dv_size) return(OK);         /* At EOF */
  if (pos + nbytes > f_dv->dv_size) nbytes = f_dv->dv_size - pos;
```

```
  block = (f_dv->dv_base + pos) >> SECTOR_SHIFT;

  spanning = FALSE;     /* set if the block spans a track */

  /* While there are "unscheduled" bytes in the request: */
  do {
      count = nbytes;

      if (f_count > 0 && block >= f_nexttrack) {
            /* The new job leaves the track, finish all gathered jobs */
            if ((r = f_finish()) != OK) return(r);
            f_must = spanning;
      }

      if (f_count == 0) {
            /* This is the first job, compute cylinder and head */
            f_opcode = opcode;
            f_fp->fl_cylinder = block / (NR_HEADS * f_sectors);
            f_fp->fl_hardcyl = f_fp->fl_cylinder * steps_per_cyl[d];
            f_fp->fl_head = (block % (NR_HEADS * f_sectors)) / f_sectors;

            /* See where the next track starts, one is trouble enough */
            f_nexttrack = (f_fp->fl_cylinder * NR_HEADS
                              + f_fp->fl_head + 1) * f_sectors;
      }

      /* Don't do track spanning I/O. */
      if (block + (count >> SECTOR_SHIFT) > f_nexttrack)
            count = (f_nexttrack - block) << SECTOR_SHIFT;

      /* Memory chunk to DMA. */
      dma_phys = user_phys;
      dma_count = dma_bytes_left(dma_phys);

#if _WORD_SIZE > 2
      /* The DMA chip uses a 24 bit address, so don't DMA above 16MB. */
      if (dma_phys >= 0x1000000) dma_count = 0;
#endif
      if (dma_count < count) {
            /* Nearing a 64K boundary. */
            if (dma_count >= SECTOR_SIZE) {
                  /* Can read a few sectors before hitting the
                   * boundary.
                   */
                  count = dma_count & ~SECTOR_MASK;
            } else {
                  /* Must use the special buffer for this. */
                  count = SECTOR_SIZE;
                  dma_phys = tmp_phys;
            }
      }

      /* Store the I/O parameters in the ftrans slots for the sectors to
       * read.  The first slot specifies all sectors, the ones following
       * it each specify one sector less.  This allows I/O to be started
       * in the middle of a block.
       */
```

```
      tp = tp0 = &ftrans[block % f_sectors];

      block += count >> SECTOR_SHIFT;
      nbytes -= count;
      f_count += count;
      if (!(iop->io_request & OPTIONAL_IO)) f_must = TRUE;

      do {
            tp->tr_count = count;
            tp->tr_iop = iop;
            tp->tr_phys = user_phys;
            tp->tr_dma = dma_phys;
            tp++;

            user_phys += SECTOR_SIZE;
            dma_phys += SECTOR_SIZE;
            count -= SECTOR_SIZE;
      } while (count > 0);

      spanning = TRUE;  /* the rest of the block may span a track */
  } while (nbytes > 0);

  return(OK);
}


/*===========================================================================*
 *                      f_finish                                 *

*===========================================================================*/
PRIVATE int f_finish()
{
/* Carry out the I/O requests gathered in ftrans[].  */

  struct floppy *fp = f_fp;
  struct trans *tp;
  int r, errors;

  if (f_count == 0) return(OK);    /* Spurious finish. */

  /* If all the requests are optional then don't read from the next
track.
   * (There may be enough buffers to read the next track, but doing so is
   * unwise.  It's no good to be greedy on a slow device.)
   */
  if (!f_must) {
      defuse();
      return(EAGAIN);
  }

  /* See if motor is running; if not, turn it on and wait */
  start_motor();

  /* Let read_id find out the next sector to read/write if it pays to do
so.
   * Note that no read_id is done while formatting if there is one format
   * request per track as there should be.
```

```
 */
fp->fl_sector = f_count >= (6 * SECTOR_SIZE) ? 0 : BASE_SECTOR;

do {
    /* This loop allows a failed operation to be repeated. */
    errors = 0;
    for (;;) {
          /* First check to see if a reset is needed. */
          if (need_reset) f_reset();

          /* Set the stepping rate */
          if (current_spec1 != spec1[d]) {
              fdc_out(FDC_SPECIFY);
              current_spec1 = spec1[d];
              fdc_out(current_spec1);
              fdc_out(SPEC2);
          }

          /* Set the data rate */
          if (pc_at) out_byte(FDC_RATE, rate[d]);

          /* If we are going to a new cylinder, perform a seek. */
          r = seek(fp);

          if (fp->fl_sector == NO_SECTOR) {
              /* Don't retry read_id too often, we need tp soon */
              if (errors > 0) fp->fl_sector = BASE_SECTOR;

              /* Find out what the current sector is */
              if (r == OK) r = read_id(fp);
          }

          /* Look for the next job in ftrans[] */
          if (fp->fl_sector != NO_SECTOR) {
              for (;;) {
                    if (fp->fl_sector >= BASE_SECTOR + f_sectors)
                          fp->fl_sector = BASE_SECTOR;

                    tp = &ftrans[fp->fl_sector - BASE_SECTOR];
                    if (tp->tr_count > 0) break;
                    fp->fl_sector++;
              }
              /* Do not transfer more than f_count bytes. */
              if (tp->tr_count > f_count) tp->tr_count = f_count;
          }

          if (r == OK && tp->tr_dma == tmp_phys
                              && f_opcode == DEV_WRITE) {
              /* Copy the bad user buffer to the DMA buffer. */
              phys_copy(tp->tr_phys, tp->tr_dma,
                              (phys_bytes) tp->tr_count);
          }

          /* Set up the DMA chip and perform the transfer. */
          if (r == OK) {
              dma_setup(tp);
              r = f_transfer(fp, tp);
```

```
            }

            if (r == OK && tp->tr_dma == tmp_phys
                                    && f_opcode == DEV_READ) {
                  /* Copy the DMA buffer to the bad user buffer. */
                  phys_copy(tp->tr_dma, tp->tr_phys,
                                    (phys_bytes) tp->tr_count);
            }

            if (r == OK) break;     /* if successful, exit loop */

            /* Don't retry if write protected or too many errors. */
            if (err_no_retry(r) || ++errors == MAX_ERRORS) {
                  if (fp->fl_sector != 0) tp->tr_iop->io_nbytes = EIO;
                  return(EIO);
            }

            /* Recalibrate if halfway, but bail out if optional I/O. */
            if (errors == MAX_ERRORS / 2) {
                  fp->fl_calibration = UNCALIBRATED;
                  if (tp->tr_iop->io_request & OPTIONAL_IO)
                        return(tp->tr_iop->io_nbytes = EIO);
            }
      }
      f_count -= tp->tr_count;
      tp->tr_iop->io_nbytes -= tp->tr_count;
  } while (f_count > 0);

  /* Defuse the leftover partial jobs. */
  defuse();

  return(OK);
}


/*===========================================================================*
 *                      defuse                                               *

*===========================================================================*/
PRIVATE void defuse()
{
/* Invalidate leftover requests in the transfer array. */

  struct trans *tp;

  for (tp = ftrans; tp < ftrans + MAX_SECTORS; tp++) tp->tr_count = 0;
  f_count = 0;
}


/*===========================================================================*
 *                      dma_setup                                            *

*===========================================================================*/
PRIVATE void dma_setup(tp)
struct trans *tp;       /* pointer to the transfer struct */
{
```

```
/* The IBM PC can perform DMA operations by using the DMA chip.  To use
it,
 * the DMA (Direct Memory Access) chip is loaded with the 20-bit memory
address
 * to be read from or written to, the byte count minus 1, and a read or
write
 * opcode.  This routine sets up the DMA chip.  Note that the chip is not
 * capable of doing a DMA across a 64K boundary (e.g., you can't read a
 * 512-byte block starting at physical address 65520).
 */

  /* Set up the DMA registers.  (The comment on the reset is a bit
strong,
   * it probably only resets the floppy channel.)
   */
  out_byte(DMA_INIT, DMA_RESET_VAL);    /* reset the dma controller */
  out_byte(DMA_FLIPFLOP, 0);       /* write anything to reset it */
  out_byte(DMA_MODE, f_opcode == DEV_WRITE ? DMA_WRITE : DMA_READ);
  out_byte(DMA_ADDR, (int) tp->tr_dma >>  0);
  out_byte(DMA_ADDR, (int) tp->tr_dma >>  8);
  out_byte(DMA_TOP, (int) (tp->tr_dma >> 16));
  out_byte(DMA_COUNT, (tp->tr_count - 1) >> 0);
  out_byte(DMA_COUNT, (tp->tr_count - 1) >> 8);
  out_byte(DMA_INIT, 2);      /* some sort of enable */
}

/*===========================================================================*
 *                      start_motor                                  *

*===========================================================================*/
PRIVATE void start_motor()
{
/* Control of the floppy disk motors is a big pain.  If a motor is off,
you
 * have to turn it on first, which takes 1/2 second.  You can't leave it
on
 * all the time, since that would wear out the diskette.  However, if you
turn
 * the motor off after each operation, the system performance will be
awful.
 * The compromise used here is to leave it on for a few seconds after
each
 * operation.  If a new operation is started in that interval, it need
not be
 * turned on again.  If no new operation is started, a timer goes off and
the
 * motor is turned off.  I/O port DOR has bits to control each of 4
drives.
 * The timer cannot go off while we are changing with the bits, since the
 * clock task cannot run while another (this) task is active, so there is
no
 * need to lock().
 */

  int motor_bit, running;
  message mess;
```

```
  motor_bit = 1 << f_drive;        /* bit mask for this drive */
  running = motor_status & motor_bit;    /* nonzero if this motor is
running */
  motor_goal = motor_status | motor_bit;/* want this drive running too */

  out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT | f_drive);
  motor_status = motor_goal;

  /* If the motor was already running, we don't have to wait for it. */
  if (running) return;             /* motor was already running */
  clock_mess(mtr_setup[d], send_mess);   /* motor was not running */
  receive(CLOCK, &mess);           /* wait for clock interrupt */
}


/*===========================================================================*
 *                      stop_motor                                           *

*===========================================================================*/
PRIVATE void stop_motor()
{
/* This routine is called by the clock interrupt after several seconds
have
 * elapsed with no floppy disk activity.  It checks to see if any drives
are
 * supposed to be turned off, and if so, turns them off.
 */

  if (motor_goal != motor_status) {
      out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT);
      motor_status = motor_goal;
  }
}


/*===========================================================================*
 *                      floppy_stop                                          *

*===========================================================================*/
PUBLIC void floppy_stop()
{
/* Stop all activity. */

  motor_goal = 0;
  stop_motor();
}


/*===========================================================================*
 *                      seek                                                 *

*===========================================================================*/
PRIVATE int seek(fp)
struct floppy *fp;           /* pointer to the drive struct */
{
/* Issue a SEEK command on the indicated drive unless the arm is already
 * positioned on the correct cylinder.
```

```
 */

  int r;
  message mess;

  /* Are we already on the correct cylinder? */
  if (fp->fl_calibration == UNCALIBRATED)
      if (recalibrate(fp) != OK) return(ERR_SEEK);
  if (fp->fl_curcyl == fp->fl_hardcyl) return(OK);

  /* No.  Wrong cylinder.  Issue a SEEK and wait for interrupt. */
  fdc_out(FDC_SEEK);
  fdc_out((fp->fl_head << 2) | f_drive);
  fdc_out(fp->fl_hardcyl);
  if (need_reset) return(ERR_SEEK);/* if controller is sick, abort seek
*/
  if (f_intr_wait() != OK) return(ERR_TIMEOUT);

  /* Interrupt has been received.  Check drive status. */
  fdc_out(FDC_SENSE);         /* probe FDC to make it return status */
  r = fdc_results();          /* get controller status bytes */
  if (r != OK || (f_results[ST0] & ST0_BITS) != SEEK_ST0
                       || f_results[ST1] != fp->fl_hardcyl) {
      /* seek failed, may need a recalibrate */
      return(ERR_SEEK);
  }
  /* give head time to settle on a format, no retrying here! */
  if (f_device & FORMAT_DEV_BIT) {
      clock_mess(2, send_mess);
      receive(CLOCK, &mess);
  }
  fp->fl_curcyl = fp->fl_hardcyl;
  return(OK);
}


/*===========================================================================*
 *                      f_transfer                                           *

*===========================================================================*/
PRIVATE int f_transfer(fp, tp)
struct floppy *fp;            /* pointer to the drive struct */
struct trans *tp;       /* pointer to the transfer struct */
{
/* The drive is now on the proper cylinder.  Read, write or format 1
block. */

  int r, s;

  /* Never attempt a transfer if the drive is uncalibrated or motor is
off. */
  if (fp->fl_calibration == UNCALIBRATED) return(ERR_TRANSFER);
  if ((motor_status & (1 << f_drive)) == 0) return(ERR_TRANSFER);

  /* The command is issued by outputting several bytes to the controller
chip.
   */
```

```
  if (f_device & FORMAT_DEV_BIT) {
      fdc_out(FDC_FORMAT);
      fdc_out((fp->fl_head << 2) | f_drive);
      fdc_out(fmt_param.sector_size_code);
      fdc_out(fmt_param.sectors_per_cylinder);
      fdc_out(fmt_param.gap_length_for_format);
      fdc_out(fmt_param.fill_byte_for_format);
  } else {
      fdc_out(f_opcode == DEV_WRITE ? FDC_WRITE : FDC_READ);
      fdc_out((fp->fl_head << 2) | f_drive);
      fdc_out(fp->fl_cylinder);
      fdc_out(fp->fl_head);
      fdc_out(fp->fl_sector);
      fdc_out(SECTOR_SIZE_CODE);
      fdc_out(f_sectors);
      fdc_out(gap[d]);  /* sector gap */
      fdc_out(DTL);           /* data length */
  }

  /* Block, waiting for disk interrupt. */
  if (need_reset) return(ERR_TRANSFER);  /* if controller is sick, abort
op */

  if (f_intr_wait() != OK) return(ERR_TIMEOUT);

  /* Get controller status and check for errors. */
  r = fdc_results();
  if (r != OK) return(r);

  if (f_results[ST1] & WRITE_PROTECT) {
      printf("%s: diskette is write protected.\n", f_name());
      return(ERR_WR_PROTECT);
  }

  if ((f_results[ST0] & ST0_BITS) != TRANS_ST0) return(ERR_TRANSFER);
  if (f_results[ST1] | f_results[ST2]) return(ERR_TRANSFER);

  if (f_device & FORMAT_DEV_BIT) return(OK);

  /* Compare actual numbers of sectors transferred with expected number.
*/
  s =  (f_results[ST_CYL] - fp->fl_cylinder) * NR_HEADS * f_sectors;
  s += (f_results[ST_HEAD] - fp->fl_head) * f_sectors;
  s += (f_results[ST_SEC] - fp->fl_sector);
  if ((s << SECTOR_SHIFT) != tp->tr_count) return(ERR_TRANSFER);

  /* This sector is next for I/O: */
  fp->fl_sector = f_results[ST_SEC];
  return(OK);
}


/*===========================================================================*
 *                    fdc_results                                            *

*===========================================================================*/
PRIVATE int fdc_results()
```

```
{
/* Extract results from the controller after an operation, then allow
floppy
 * interrupts again.
 */

  int result_nr, status;
  struct milli_state ms;

  /* Extract bytes from FDC until it says it has no more.  The loop is
   * really an outer loop on result_nr and an inner loop on status.
   */
  result_nr = 0;
  milli_start(&ms);
  do {
      /* Reading one byte is almost a mirror of fdc_out() - the DIRECTION
       * bit must be set instead of clear, but the CTL_BUSY bit destroys
       * the perfection of the mirror.
       */
      status = in_byte(FDC_STATUS) & (MASTER | DIRECTION | CTL_BUSY);
      if (status == (MASTER | DIRECTION | CTL_BUSY)) {
            if (result_nr >= MAX_RESULTS) break;     /* too many results
*/
            f_results[result_nr++] = in_byte(FDC_DATA);
            continue;
      }
      if (status == MASTER) { /* all read */
            enable_irq(FLOPPY_IRQ);
            return(OK); /* only good exit */
      }
  } while (milli_elapsed(&ms) < TIMEOUT);
  need_reset = TRUE;          /* controller chip must be reset */
  enable_irq(FLOPPY_IRQ);
  return(ERR_STATUS);
}


/*===========================================================================*
 *                      f_handler                                            *

*===========================================================================*/
PRIVATE int f_handler(irq)
int irq;
{
/* FDC interrupt, send message to floppy task. */

  interrupt(FLOPPY);
  return 0;
}


/*===========================================================================*
 *                      fdc_out                                              *

*===========================================================================*/
PRIVATE void fdc_out(val)
int val;          /* write this byte to floppy disk controller */
```

```
{
/* Output a byte to the controller.  This is not entirely trivial, since
you
 * can only write to it when it is listening, and it decides when to
listen.
 * If the controller refuses to listen, the FDC chip is given a hard
reset.
 */

  struct milli_state ms;

  if (need_reset) return;     /* if controller is not listening, return */

  /* It may take several tries to get the FDC to accept a command. */
  milli_start(&ms);
  while ((in_byte(FDC_STATUS) & (MASTER | DIRECTION)) != (MASTER | 0)) {
      if (milli_elapsed(&ms) >= TIMEOUT) {
            /* Controller is not listening.  Hit it over the head. */
            need_reset = TRUE;
            return;
      }
  }
  out_byte(FDC_DATA, val);
}


/*===========================================================================*
 *                      recalibrate                                  *

*===========================================================================*/
PRIVATE int recalibrate(fp)
struct floppy *fp;      /* pointer tot he drive struct */
{
/* The floppy disk controller has no way of determining its absolute arm
 * position (cylinder).  Instead, it steps the arm a cylinder at a time
and
 * keeps track of where it thinks it is (in software).  However, after a
 * SEEK, the hardware reads information from the diskette telling where
the
 * arm actually is.  If the arm is in the wrong place, a recalibration is
done,
 * which forces the arm to cylinder 0.  This way the controller can get
back
 * into sync with reality.
 */

  int r;

  /* Issue the RECALIBRATE command and wait for the interrupt. */
  start_motor();        /* can't recalibrate with motor off */
  fdc_out(FDC_RECALIBRATE);   /* tell drive to recalibrate itself */
  fdc_out(f_drive);           /* specify drive */
  if (need_reset) return(ERR_SEEK);/* don't wait if controller is sick */
  if (f_intr_wait() != OK) return(ERR_TIMEOUT);

  /* Determine if the recalibration succeeded. */
  fdc_out(FDC_SENSE);         /* issue SENSE command to request results */
```

```
  r = fdc_results();          /* get results of the FDC_RECALIBRATE
command*/
  fp->fl_curcyl = NO_CYL;     /* force a SEEK next time */
  if (r != OK ||        /* controller would not respond */
     (f_results[ST0] & ST0_BITS) != SEEK_ST0 || f_results[ST_PCN] != 0) {
      /* Recalibration failed.  FDC must be reset. */
      need_reset = TRUE;
      return(ERR_RECALIBRATE);
  } else {
      /* Recalibration succeeded. */
      fp->fl_calibration = CALIBRATED;
      return(OK);
  }
}


/*===========================================================================*
 *                      f_reset                                              *

*===========================================================================*/
PRIVATE void f_reset()
{
/* Issue a reset to the controller.  This is done after any catastrophe,
 * like the controller refusing to respond.
 */

  int i;
  message mess;

  /* Disable interrupts and strobe reset bit low. */
  need_reset = FALSE;

  /* It is not clear why the next lock is needed.  Writing 0 to DOR
causes
   * interrupt, while the PC documentation says turning bit 8 off
disables
   * interrupts.  Without the lock:
   *   1) the interrupt handler sets the floppy mask bit in the 8259.
   *   2) writing ENABLE_INT to DOR causes the FDC to assert the
interrupt
   *      line again, but the mask stops the cpu being interrupted.
   *   3) the sense interrupt clears the interrupt (not clear which one).
   * and for some reason the reset does not work.
   */
  lock();
  motor_status = 0;
  motor_goal = 0;
  out_byte(DOR, 0);           /* strobe reset bit low */
  out_byte(DOR, ENABLE_INT);  /* strobe it high again */
  unlock();
  receive(HARDWARE, &mess);   /* collect the RESET interrupt */

  /* The controller supports 4 drives and returns a result for each of
them.
   * Collect all the results now.  The old version only collected the
first
```

```
   * result.  This happens to work for 2 drives, but it doesn't work for
3
   * or more drives, at least with only drives 0 and 2 actually connected
   * (the controller generates an extra interrupt for the middle drive
when
   * drive 2 is accessed and the driver panics).
   *
   * It would be better to keep collecting results until there are no
more.
   * For this, fdc_results needs to return the number of results (instead
   * of OK) when it succeeds.
   */
  for (i = 0; i < 4; i++) {
      fdc_out(FDC_SENSE);     /* probe FDC to make it return status */
      (void) fdc_results();   /* flush controller */
  }
  for (i = 0; i < NR_DRIVES; i++)  /* clear each drive */
      floppy[i].fl_calibration = UNCALIBRATED;

  /* The current timing parameters must be specified again. */
  current_spec1 = 0;
}


/*===========================================================================*
 *                      send_mess                                            *

*===========================================================================*/
PRIVATE void send_mess()
{
/* This routine is called when the clock task has timed out on motor
startup.*/

  message mess;

  send(FLOPPY, &mess);
}


/*===========================================================================*
 *                      f_intr_wait                                          *

*===========================================================================*/
PRIVATE int f_intr_wait()
{
/* Wait for an interrupt, but not forever.  The FDC may have all the time
of
 * the world, but we humans do not.
 */
  message mess;

  f_busy = BSY_IO;
  clock_mess(WAKEUP, f_timeout);
  receive(HARDWARE, &mess);

  if (f_busy == BSY_WAKEN) {
      /* No interrupt from the FDC, this means that there is probably no
```

```
       * floppy in the drive.  Get the FDC down to earth and return
error.
       */
      f_reset();
      return(ERR_TIMEOUT);
  }
  f_busy = BSY_IDLE;
  return(OK);
}


/*===========================================================================*
 *                      f_timeout                                *

*===========================================================================*/
PRIVATE void f_timeout()
{
/* When it takes too long for the FDC to get an interrupt (no floppy in
the
 * drive), this routine is called.  It sets a flag and fakes a hardware
 * interrupt.
 */
  if (f_busy == BSY_IO) {
      f_busy = BSY_WAKEN;
      interrupt(FLOPPY);
  }
}


/*===========================================================================*
 *                      read_id                                  *

*===========================================================================*/
PRIVATE int read_id(fp)
struct floppy *fp;      /* pointer to the drive struct */
{
/* Determine current cylinder and sector. */

  int result;

  /* Never attempt a read id if the drive is uncalibrated or motor is
off. */
  if (fp->fl_calibration == UNCALIBRATED) return(ERR_READ_ID);
  if ((motor_status & (1 << f_drive)) == 0) return(ERR_READ_ID);

  /* The command is issued by outputting 2 bytes to the controller chip.
*/
  fdc_out(FDC_READ_ID);       /* issue the read id command */
  fdc_out( (f_fp->fl_head << 2) | f_drive);

  /* Block, waiting for disk interrupt. */
  if (need_reset) return(ERR_READ_ID);    /* if controller is sick, abort
op */

  if (f_intr_wait() != OK) return(ERR_TIMEOUT);

  /* Get controller status and check for errors. */
```

```
  result = fdc_results();
  if (result != OK) return(result);

  if ((f_results[ST0] & ST0_BITS) != TRANS_ST0) return(ERR_READ_ID);
  if (f_results[ST1] | f_results[ST2]) return(ERR_READ_ID);

  /* The next sector is next for I/O: */
  f_fp->fl_sector = f_results[ST_SEC] + 1;
  return(OK);
}


/*===========================================================================*
 *                        f_do_open                                  *
 *===========================================================================*/
PRIVATE int f_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;              /* pointer to open message */
{
/* Handle an open on a floppy.  Determine diskette type if need be. */

  int dtype;
  struct test_order *top;

  /* Decode the message parameters. */
  if (f_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  dtype = f_device & DEV_TYPE_BITS;/* get density from minor dev */
  if (dtype >= MINOR_fd0a) dtype = 0;
  if (dtype != 0) {
      /* All types except 0 indicate a specific drive/medium
combination.*/
      dtype = (dtype >> DEV_TYPE_SHIFT) - 1;
      if (dtype >= NT) return(ENXIO);
      f_fp->fl_density = dtype;
      f_fp->fl_geom.dv_size = (long) nr_blocks[dtype] << SECTOR_SHIFT;
      return(OK);
  }
  if (f_device & FORMAT_DEV_BIT) return(EIO);  /* Can't format /dev/fdx
*/

  /* No need to test if the motor is still running. */
  if (motor_status & (1 << f_drive)) return(OK);

  /* The device opened is /dev/fdx.  Experimentally determine
drive/medium.
   * First check fl_density.  If it is not NO_DENS, the drive has been
used
   * before and the value of fl_density tells what was found last time.
Try
   * that first.
   */
  if (f_fp->fl_density != NO_DENS && test_read(f_fp->fl_density) == OK)
      return(OK);

  /* Either drive type is unknown or a different diskette is now present.
   * Use test_order to try them one by one.
```

```
   */
  for (top = &test_order[0]; top < &test_order[NT-1]; top++) {
      dtype = top->t_density;

      /* Skip densities that have been proven to be impossible */
      if (!(f_fp->fl_class & (1 << dtype))) continue;

      if (test_read(dtype) == OK) {
            /* The test succeeded, use this knowledge to limit the
             * drive class to match the density just read.
             */
            f_fp->fl_class &= top->t_class;
            return(OK);
      }
      /* Test failed, wrong density or did it time out? */
      if (f_busy == BSY_WAKEN) break;
  }
  f_fp->fl_density = NO_DENS;
  return(EIO);                /* nothing worked */
}


/*==========================================================================*
 *                      test_read                                           *
 *==========================================================================*/
PRIVATE int test_read(density)
int density;
{
/* Try to read the highest numbered sector on cylinder 2.  Not all floppy
 * types have as many sectors per track, and trying cylinder 2 finds the
 * ones that need double stepping.
 */

  message m;
  int r, device;

  f_fp->fl_density = density;
  device = ((density + 1) << DEV_TYPE_SHIFT) + f_drive;
  f_fp->fl_geom.dv_size = (long) nr_blocks[density] << SECTOR_SHIFT;
  m.m_type = DEV_READ;
  m.DEVICE = device;
  m.PROC_NR = FLOPPY;
  m.COUNT = SECTOR_SIZE;
  m.POSITION = (long) test_sector[density] * SECTOR_SIZE;
  m.ADDRESS = (char *) tmp_buf;
  r = do_rdwt(&f_dtab, &m);
  if (r != SECTOR_SIZE) return(EIO);

  partition(&f_dtab, f_drive, P_FLOPPY);
  return(OK);
}


/*==========================================================================*
 *                      f_geometry                                          *

*==========================================================================*/
```

```
PRIVATE void f_geometry(entry)
struct partition *entry;
{
  entry->cylinders = nr_blocks[d] / (NR_HEADS * f_sectors);
  entry->heads = NR_HEADS;
  entry->sectors = f_sectors;
}
```