FORMATOS CD-ROM

1. Estructura física

Su diámetro es de 12 cm, su espesor, de 1,2 mm y el agujero que posee en medio tiene un diámetro de 15 mm. La información, almacenada en una espiral de diminutos hoyos sobre una capa metálica brillante, que es protegida a su vez por una laca transparente. Las informaciones a almacenar se impresionan sobre la capa metálica en forma de los llamados pits y lands, que son pequeñas protuberancias y cavidades que representan los diferentes bit. Los pits y los lands se alinean a lo largo de una única espiral que va desde dentro hacia fuera y cubre todo el CD. La densidad de un CD alcanza casi las 16.000 pistas por pulgadas (tracks per inch, TPI), lo cual resulta difícilmente comparable con las 135 TPI que presenta los disquetes de alta densidad de 3.5" y los varios cientos de los discos duros. La longitud de esta espiral es aproximadamente de 6 kilómetros en los que se albergan no menos de dos billones de pits. Por esta razón, el rayo de luz láser también debe ser reducido para detectar las secuencias de pits y lands; esto se consigue mediante el empleo de lentes, que concentran todos los rayos en un solo punto. La razón de usar tecnología de luz láser es que en este tipo de luz los rayos viajan en paralelo con gran precisión, sin dispersarse.

Encabezamiento del CD

La superficie grabable de un CD se divide en tres partes: el lead in, la zona de datos y el lead out. El lead in (el 'encabezamiento') ocupa los cuatro primeros milímetros del CD en el margen interior y contiene una especie de índice. A continuación sigue la zona de datos que como mucho ocupa 33 mm, dependiendo del nivel de ocupación del CD. Por último, la parte final la constituye la zona de lead out, que es una especie de marca final. Se encuentra inmediatamente detrás del final de la zona de datos ocupada y tiene una anchura de 1 mm.

Diferencias entre CDs y discos duros

En un disco magnético existen muchas pistas, cada una con el mismo número de sectores; estas pistas son mayores cuanto más exteriores son, de lo que se deduce que los sectores son también mayores cuanto más exterior es la pista que los contiene. Esto quiere decir que en un disco magnético la densidad de grabación es menor cuanto más nos aproximamos al exterior del disco. La explicación de esto se encuentra en que estos discos se mueven a velocidad angular constante (CAV), es decir, barren un sector a la

misma velocidad de giro, independiente de si la cabeza se encuentra en un sector interno o en uno externo

Por el contrario, en un CD existe una única pista. Los sectores ocupan todos el mismo espacio (son de igual tamaño) y tienen la misma densidad de grabación, de forma que para poder leerlos, la cabeza debe pasar sobre ellos a la misma velocidad lineal. En un disco magnético, que se mueve con velocidad angular (ω) constante, al irnos acercando al exterior del disco, el radio R va aumentando, lo cual quiere decir que la velocidad lineal (v) a la que pasa la cabeza por encima del disco es cada vez mayor, y por esto los sectores son cada vez mayores en longitud y la densidad de grabación dentro de ellos es menor. Como los sectores en un CD son de igual longitud, si debe pasar a la misma velocidad por todos ellos, está claro que hay que modificar la velocidad de rotación del disco. Por esto se dice que los CDs se mueven con velocidad lineal constante (CLV). Es necesario, por tanto, que el CD ajuste su velocidad de rotación (≡ velocidad angular) a la posición actual del cabezal: Esta es una de las razones por la que una unidad CD-ROM presenta velocidades de acceso mucho menores que los discos magnéticos. Otra razón es que es más complicado encontrar un sector en una espiral de 6 Km de longitud que en un medio elegantemente dividido en pistas y sectores.

Almacenamiento de bit y Byte

El valor 1 de un bit se contempla siempre como el paso de un pit a un land o al revés. La longitud del pit o del land representa el número de bits con valor cero que siguen al bit con valor 1.

Un "Channel 1" es un bit con valor 1 y un "Channel 0" es un bit con valor 0.

No se pueden situar dos channel 1 seguidos. Pero entonces, ¿cómo puede haber dos cambios, de pit a land y luego de land a pit, sin que haya al menos un pequeño land en medio que los separe? A un channel 1 necesariamente le debe seguir como mínimo dos channel 0. Sólo entonces la distancia hasta el próximo channel 1 es suficientemente grande como para que no pase desapercibido a la óptica de la lectura. Por otro lado, los pits y los lands no deben ser demasiados largos, como máximo 11 bit.

Todas estas condiciones desembocan en el procedimiento EFM, "eight to fourteen modulation", en el que un Byte a almacenar se traduce junto con sus ocho bit en 14 channel bit. Una ristra de 14 bits proporciona 16384 combinaciones, de las cuales más de 256 cumplen los requerimientos de codificación que perseguimos. Se hace entonces una correlación entre las 256 combinaciones de una ristra de 8 bits y estas 256 combinaciones que cumplen el formato. La tabla de conversión es parte integrante de la electrónica de control de cada unidad CD-ROM.

De todas maneras, la conversión vía tabla EFM no contempla un problema: la separación de bit unitarios. Por este motivo, a cada Byte con sus 14 channel bit se le añade tres channel bit más que se denominan

'merging bit'. Estos separan los Bytes uno de otro y con ello elevan el número de channel bit a 17 por Byte.

Frame

Un frame es el bloque de información coherente más pequeño de un CD. Un frame contiene 24 Bytes (cada uno con 17 channel bit) que, junto con alguna otra información, constituyen el frame como bloque de datos. El inicio está formado por lo que se denomina Sync-Pattern, un diseño concreto de, en total, 27 channel bits, que indica a la unidad el comienzo de un nuevo frame. A continuación se encuentra un byte de control y le siguen los 24 Bytes de datos del frame.

Un frame acaba con 8 Bytes de corrección de errores que a su vez están también constituidos por 17 channel bit. Así, sumando, se llega a un total de 588 channel bit por frame.

Sectores

En el siguiente nivel se engloban 98 frames para constituir un sector, donde, por un lado, se juntan los distintos Bytes de datos de los frames y, por otro, los Bytes de control y los Bytes para la corrección de errores.

Los sectores se reproducen en un espacio concreto de tiempo, el direccionamiento se lleva a cabo por unidad de tiempo y en realidad, en el formato minutos/segundos/sectores.

Sub channel

Los 98 Bytes de control de los 98 frame de un sector contienen información muy importante para el direccionamiento. Los Bytes se desdoblan en lo que se denomina sub channel.

Todos los bits 0 de los 98 Bytes forman los Sub channel P, el cual se utiliza para indicar el inicio de una grabación de audio, es decir, datos digitales.

Los segundos bits de los 98 Bytes de control forman los Sub channel Q, que contiene las informaciones cruciales que indican la posición de un sector en el CD. El canal Q tiene una tarea especial en la zona de lead in de un CD, pues aquí se encuentra lo que se conoce como Table Of Contents (TOC), de donde se obtiene, por ejemplo, el número de pistas de audio de un CD.

Los restantes 6 bits de los 98 Bytes de control se engloban en el Sub channel R/W, que se utilizan para tareas de sincronización.

Capacidad de almacenamiento

Del número de sectores se obtiene la capacidad total de un CD-ROM que depende de si se utiliza o no toda la superficie impresionable del CD-ROM. Varían de 500 a 700 MB.

Corrección de errores

Los Bytes que se guardan en cada sector para corrección de errores están basados en el algoritmo "Cross Interleave Reed Solomon Code" (CIRC), y que en todos los reproductores de CD de audio y unidades CD-ROM tiene como misión preocuparse a nivel hardware de una transferencia de sectores libres de errores.

Mediante este procedimiento se obtiene un ratio de errores de 10^{-8}, es decir, cada cien millones de bit, aparece uno que por error no se reconoce y por tanto no se corrige. Mientras que para escuchar un CD apenas se percibe, este ratio es todavía demasiado grande para el ámbito de los ordenadores. Por este motivo, los CD-ROM trabajan con un formato de sector ligeramente modificado.

Formato de CD-ROM (*Yellow Book*)

En el ámbito de sectores sólo se diferencia del formato de CD de audio (Red Book) en la zona de datos, para disminuir el número de errores. La zona de datos se reduce de 2352 a 2048 Bytes (2KB).

Existen dos formatos para los sectores del CD-ROM, los denominados modo 1 y modo2. En el modo 2 se ahorran de nuevo las informaciones para la corrección de errores (la corrección de errores básica de la unidad de CD-ROM se conserva) puesto que estos sectores están previstos para el almacenamiento de todo tipo de información para lo que un error de lectura no conlleva graves influencias sobre un programa.

La velocidad de transferencia de datos de una unidad aumenta con sectores modo 2 (75 sectores/segundo * 2336 KB/sector = 171 KB por segundo), aunque prácticamente todos los CD-ROM contienen únicamente sectores en modo 1.

Ambos formatos tienen en común 12 Bytes de sincronización al principio del sector, así como una cabecera de 4 Bytes.

El formato XA

Las modernas unidades de CD-ROM y controladores son capaces de gestionar también un formato que fue desarrollado por voluntad de sus autores, Sony y Phillips en colaboración con Microsoft en 1989, y ha llegado a convertirse en formato estándar para multimedia sobre CD-ROM.

La principal característica del estándar XA "eXtended Architecture" es su capacidad de interdireccionar archivos por el procedimiento conocido como Interleave. Por ejemplo, típico problema de las aplicaciones multimedia donde se tiene que mostrar un texto en la pantalla mientras se reproduce un vídeo al tiempo que, como fondo, suena una música de acompañamiento. La aplicación tiene que trabajar con tres fuentes de datos de tres archivos diferentes y además en tiempo real. La especificación XA puede proporcionar ayuda a través del interleaving, pues éste se ocupa de que la CPU pueda cargar parte de texto, vídeo y audio que precisa sin necesidad de desplazar la cabeza lectora, puesto que los archivos sencillamente están anidados. Primero hay tres sectores con texto, después cuatro con vídeo y tres con audio y se repite de nuevo la secuencia hasta la finalización de las tres fuentes de datos.

En la información contenida en el directorio a nivel lógico, se determina exactamente cómo están intrincados los archivos, si primero hay dos o tres sectores con texto a los que siguen cinco o quince sectores con datos de vídeo, etc. A nivel físico, sólo se consigue, con un nuevo formato de sector, la posibilidad de asignar individualmente los sectores a determinados archivos.

El formato CD-ROM XA tiene dos formatos de sector diferentes: forma 1 y forma 2. Ambos se parecen al modo 1 del formato de sector de CD-ROM pero presentan pequeñas diferencias en la parte inicial del formato del sector, donde se almacenan los datos.

Tanto la forma 1 como la forma 2 usan los ocho Bytes que el modo 1 de CD-ROM deja sin utilizar entre los datos para la detección de errores (EDC) y para la corrección de errores (ECC). En el formato XA estos Bytes se trasladan al principio del sector donde constituyen un *subheader* inmediatamente a continuación del header en el cual se almacenan las informaciones adicionales, por ejemplo para el interleaving, que diferencian el formato XA del formato normal CD-ROM.

2. El formato lógico- High Sierra

Si se quiere acceder a los datos almacenados no en forma de sectores sino como archivos y directorios, se precisa un formato lógico. Naturalmente cada fabricante puede asignar libremente el formato lógico que desee a sus CD-ROM, pero entonces se precisará del controlador apropiado bajo un sistema operativo. Esto es razón suficiente como para desarrollar una especificación que estandarice y regule la distribución de un CD-ROM en

archivos y directorios. En 1985, diferentes distribuidores de software y fabricantes de hardware trabajaron conjuntamente obteniendo el formato HSG, "High Sierra Group". Un año después, las autoridades de normalización americanas ISO estandarizaron la propuesta, que se presentó bajo el título "Volume and File Structure of Compact Read Only Optical Disk for Information Interchange". Desde entonces, se habla de la norma "ISO 9660".

Existen pequeñas diferencias entre ISO y HSG que se hacen patentes sobre todo en la estructura de las entradas de los directorios.

Sectores lógicos

El formato HSG define el sector lógico en cuanto a su tamaño orientado a los sectores físicos, 2048 Bytes. Cada sector posee un número inequívoco, "Logical Sector Number". El primer LSN direccionable lleva el número 0 y se corresponde con el sector físico cuya dirección es 00:02:00. Es decir, los primeros 150 sectores físicos que constituyen los dos primeros segundos de un CD no pueden direccionarse desde el nivel de formato lógico. De esto se deduce la fórmula de conversión entre las direcciones (mm:ss:ff) y LSN.

LSN (mm:ss:ff) = (mm*60 + ss) * 75 -150

Bloques lógicos

HSG divide el sector lógico en varios bloques lógicos. Cada bloque lógico (LBN) puede tener un tamaño de 512 Bytes, 1024 Bytes o 2048 Bytes. En el último caso coincide con el tamaño del sector lógico.

Por ejemplo: Para un tamaño de 512KB para el bloque lógico

Sector lógico 0				Sector lógico 1			
0	1	2	3	4	5	6	7

Archivos y nombres de archivos

Los archivos en HSG se almacenan como una secuencia continua de bloques lógicos, los que denomina Extent. Por este motivo no existe una File Allocation Table (FAT). Si se conoce la posición del inicio de un archivo y su longitud, se conocen todos los LBN en los que está guardado el archivo. Esto resulta así de sencillo debido a que los archivos no se pueden borrar de un CD_ROM (desaparece la necesidad de aprovechar los espacios vacíos que

quedan al eliminar archivos para almacenar fragmentos de otros archivos nuevos).

ISO y HSG se diferencian en los nombres de archivos. Las reglas de HSG dejan entrever a Microsoft, puesto que los nombres de archivo deben seguir el arquetipo de DOS, es decir, un máximo de ocho caracteres para el nombre a continuación un punto y por último un máximo de tres caracteres para la extensión. Permite utilizar las cifras del 0 al 9, las letras mayúsculas de la A a la Z y el carácter _ o subguión.

ISO coincide con HSG en cuanto a los caracteres permitidos, pero por otra parte se inclina por la longitud de nombre de archivo de UNIX, esto es, un máximo de 31 caracteres con o sin punto de separación, aunque uno sólo, en cualquier lugar. El nombre debe concluir con un punto y coma, que separa la entrada opcional del número de la versión del nombre del archivo.

Directorios y subdirectorios

Un CD ISO contiene un directorio principal a partir del cual se pueden declarar subdirectorios, limitando el número máximo de niveles a ocho. El directorio principal, así como los subdirectorios que parten de él se almacenan como archivos.

Existe un problema de intercambio de datos entre dos sistemas. En todos los números para cuya representación se precisan más de 8 bit se plantea la cuestión de en qué secuencia deben disponerse los diferentes Bytes que lo componen (*endianess*). La respuesta de Intel: primero el menos significativo y después el más significativo. Sin embargo, muchos procesadores trabajan exactamente al revés, como por ejemplo los de Motorola. Por este motivo, toda información de 16 o 32 bit se guarda por duplicado, una vez en formato Intel (sufijo I) y otra en formato Motorola (sufijo M). Con ello el sistema operativo tiene la posibilidad de escoger el campo en el formato con el que trabaja el procesador sobre el que corre el propio sistema operativo.

Path Table

Es una especie de abreviación de los subdirectorios para ayudar en la búsqueda de archivos. En la Path Table se enumeran los nombres de todos los directorios y subdirectorios de un CD con el número del sector lógico en que comienza cada uno de ellos. Si se tiene esta tabla en la memoria, basta la lectura de un sector para averiguar la dirección de un archivo, siempre que la entrada de directorio del archivo se encuentre en el primer sector de los datos de directorio, de lo contrario, se han de ir cargando los diferentes sectores de datos de directorio hasta encontrarlo.

Dado que el Path Table contiene como números de sector números enteros de 32 bit, en el CD-ROM hay dos copias del Path Table, una con el formato Intel y otra en formato Motorola.

Extended Attribut Records (XAR)

Estos registros ofrecen la posibilidad de, al crear un archivo, almacenar cualquier información además de unos atributos predefinidos, como por ejemplo una identificación del usuario, derechos de acceso, informaciones sobre la estructura del bloque de datos que está guardado en el archivo y otros más.

Para que los archivos de directorios no aumenten innecesariamente con estas informaciones, éstas no se guardan en la entrada de directorio de un archivo, sino que constituye el primer sector lógico del mismo.

Bajo DOS no tienen sentido, por que este sistema operativo ignora todas estas entradas.

Volúmenes

El conjunto de archivos y directorios que están almacenados en un CD constituye un volumen.

HSG describe un formato de volumen basado en dos componentes: una zona de sistema y una zona de datos. La zona de sistema contiene los 16 primeros sectores lógicos de un CD (LSN 0 hasta LSN 15). Su utilización está reservada para los correspondientes sistemas operativos.

La zona de datos está encabezada por "Volume Descriptors" (VD), de los hay hasta 5 diferentes definidos por HSG. Cada uno describe un aspecto concreto del medio y ocupa un sector lógico completo. Sólo es imprescindible el "Standard Volume Descriptor", los restantes 4 son opcionales. Éste contiene información como la dirección del archivo de directorios con el directorio principal y la dirección de la Path Table además indican los nombres de los archivos 'Copyright' y 'Abstract file' que son archivos incluidos en el directorio principal.

DRIVER DE CD-ROM

Introducción

El fichero que vamos a describir (mcd.c), contiene el manejador de dispositivo del controlador de CD-ROM Mitsumi, perteneciente al MINIX 2.0.

El driver proporciona un conjunto de funciones para dar soporte a las principales tareas que realizamos con un CD-ROM. Siguiendo la filosofía con respecto a los drivers de Minix, existe una parte que no depende de los dispositivos y otra que sí. Nosotros nos vamos a ocupar claramente de ésta última.

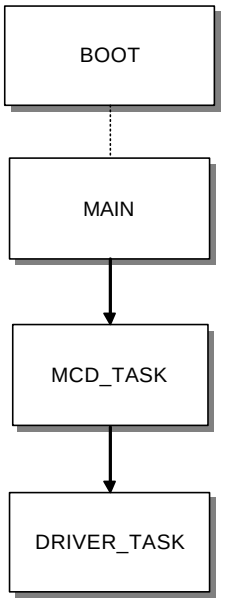
La programación del controlador se realiza a través de tres registros de E/S, a los que mandamos los comandos y desde los que leemos los datos de salida. Los tres registros se denominan: ***puerto de datos, puerto de flag y puerto de control***. A partir de sus nombres podemos hacernos una idea de cuál es su misión. Del puerto de datos recibimos y enviamos todos los datos. El puerto de flag nos indica el estado de la unidad tras realizar alguna operación. Al puerto de control enviamos los comandos que controlan el funcionamiento de la unidad.

El modo de operación básicamente consiste en enviar al puerto de control un comando. Seguidamente al puerto de datos enviamos o recibimos (según el comando) la información que fuera necesaria, y por último podemos inspeccionar el puerto de flag para comprobar que todo ha ido bien.

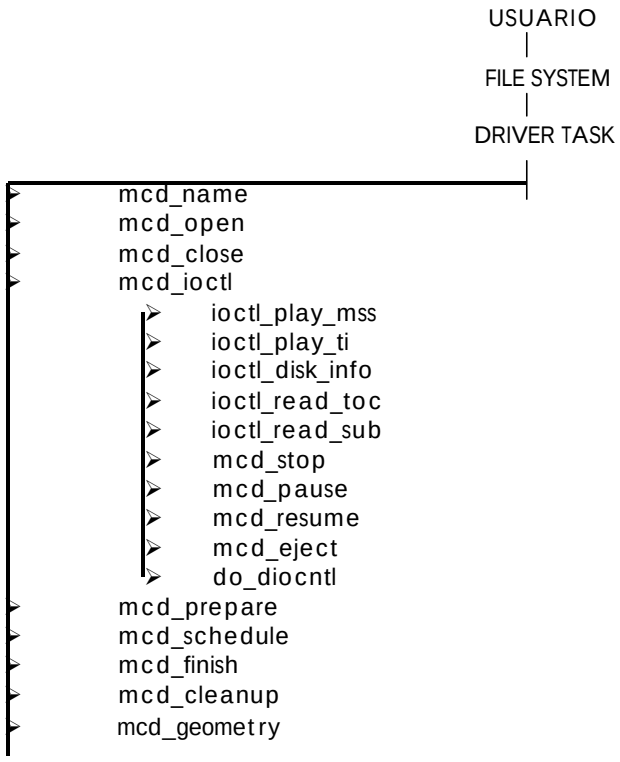
El driver del CD-ROM se basa en múltiples funciones sencillas que se encargan de realizar las tareas más comunes que se efectúan sobre un CD: play, stop, close, etc. El único punto de entrada a este controlador será la función **mcd_task**, que se encargará de llamar a la función **driver_task**, al igual que hacen los demás controladores de dispositivos de bloques. A esta función se le pasan punteros a las distintas funciones que componen este controlador. De esta manera, driver_task tratará a todos los dispositivos de bloques (Disco duro, floppy, CD-ROM) por igual, llamando a funciones genéricas, que funcionarán según las particularidades de cada dispositivo.

El elemento clave en el manejo del CD-ROM es el estado del mismo; en el controlador se usará la variable **McdStatus**, que nos va a servir para saber qué está haciendo el CD en cada momento: si está reproduciendo, si está parado, etc. Para ello usará multitud de máscaras para comparar con el valor de McdStatus.

Orden de ejecución



Funciones



Listado

```
/* This file contains the driver for a Mitsumi cdrom controller.
 * The file contains one entry point:
 *
 *
 *   mcd_task:            main entry when system is brought up
 *
 *  Mar 30 1995                 Author: Michel R. Prevenier
 */


#include "kernel.h"
#include "driver.h"
#include "drvlib.h"
#include <minix/cdrom.h>
#include <sys/ioctl.h>

/* en 'include/minix/config.h' se establece si se incluye o no el driver */
#if ENABLE_MITSUMI_CDROM

#define MCD_DEBUG         0      /* debug level */

/* Default IRQ. */
#define MCD_IRQ                 10

/* Default I/O ports (offset from base address */
#define MCD_IO_BASE_ADDRESS      0x300
#define MCD_DATA_PORT            (mcd_io_base+0)
#define MCD_FLAG_PORT            (mcd_io_base+1)
#define MCD_CONTROL_PORT  (mcd_io_base+2)

/* Miscellaneous constants. */
#define MCD_SKIP          150   /* Skip first 150 blocks on cdrom */
#define MCD_BLOCK_SIZE    2048  /* Block size in cooked mode */
#define MCD_BLOCK_SHIFT   11     /* for division */
#define MCD_BLOCK_MASK    2047   /* and remainder */
#define BYTES_PER_SECTOR  2048   /* Nr. of bytes in a sector */
#define SECTORS_PER_SECOND 75    /* Nr. of sectors in a second */
#define SECONDS_PER_MINUTE 60   /* You never know, things change :-) */
#define MCD_RETRIES       2      /* Number of retries for a command */
#define REPLY_DELAY       5000   /* Count to wait for a reply */
#define MAX_TRACKS        104    /* Maximum nr. of tracks */
#define LEAD_OUT                0xAA   /* Lead out track is always 0xAA */

/* NR_PARTITIONS : Número de entradas en la tabla de particiones ® 4 */
#define SUB_PER_DRIVE           (NR_PARTITIONS * NR_PARTITIONS)

/* Comandos que se le pueden enviar a la unidad */
/* Drive commands */
#define MCD_GET_VOL_INFO  0x10   /* Read volume information */
#define MCD_GET_Q_CHANNEL 0x20   /* Read q-channel information */
#define MCD_GET_STATUS    0x40   /* Read status of drive   */
#define MCD_SET_MODE      0x50   /* Set transmission mode */
#define MCD_RESET         0x60   /* Reset controller */
#define MCD_STOP_AUDIO    0x70   /* Stop audio playing */
#define MCD_SET_DRIVE_MODE 0xA0  /* Set drive mode */
#define MCD_READ_FROM_TO   0xC0  /* Read from .. to .. */
#define MCD_GET_VERSION   0xDC   /* Get version number */
#define MCD_STOP          0xF0   /* Stop everything */
#define MCD_EJECT         0xF6   /* Eject cd */
#define MCD_PICKLE        0x04   /* Needed for newer drive models */
/* Command bits for MCD_SET_MODE command */
#define MCD_MUTE_DATA     0x01   /* 1 = Don't play back data as audio */
#define MCD_GET_TOC       0x04   /* 0 = Get toc on next GET_Q_CHANNEL */
#define MCD_ECC_MODE      0x20   /* 0 = Use secondary ecc */
#define MCD_DATALENGTH    0x40   /* 0 = Read user data only */
```

```
#define MCD_COOKED        (MCD_MUTE_DATA)
#define MCD_TOC           (MCD_MUTE_DATA | MCD_GET_TOC)


/* Status bits */
#define MCD_CMD_ERROR     0x01   /* Command error */
#define MCD_AUDIO_BUSY    0x02   /* Audio disk is playing */
#define MCD_READ_ERROR    0x04   /* Read error */
#define MCD_AUDIO_DISK    0x08   /* Audio disk is in */
#define MCD_SPINNING      0x10   /* Motor is spinning */
#define MCD_DISK_CHANGED  0x20   /* Disk has been removed or changed */
#define MCD_DISK_IN       0x40   /* Disk is in */
#define MCD_DOOR_OPEN     0x80   /* Door is open */

/* Flag bits */
#define MCD_DATA_AVAILABLE 0x02  /* Data available */
#define MCD_BUSY           0x04  /* Drive is busy */

/* Function prototypes */
FORWARD _PROTOTYPE ( int mcd_init, (void));
FORWARD _PROTOTYPE ( int c_handler, (int irq));
FORWARD _PROTOTYPE ( int mcd_play_mss, (struct cd_play_mss));
FORWARD _PROTOTYPE ( int mcd_play_tracks, (struct cd_play_track tracks));
FORWARD _PROTOTYPE ( int mcd_stop, (void));
FORWARD _PROTOTYPE ( int mcd_eject, (void));
FORWARD _PROTOTYPE ( int mcd_pause, (void));
FORWARD _PROTOTYPE ( int mcd_resume, (void));
FORWARD _PROTOTYPE ( u8_t bin2bcd, (u8_t b));
FORWARD _PROTOTYPE ( void bcd2bin, (u8_t *bcd));
FORWARD _PROTOTYPE ( long mss2block, (u8_t *mss));
FORWARD _PROTOTYPE ( void block2mss, (long block, u8_t *mss));
FORWARD _PROTOTYPE ( int mcd_get_reply, (u8_t *reply, int delay));
FORWARD _PROTOTYPE ( int mcd_get_status, (int f));
FORWARD _PROTOTYPE ( int mcd_ready, (int delay));
FORWARD _PROTOTYPE ( int mcd_data_ready, (int delay));
FORWARD _PROTOTYPE ( int mcd_set_mode, (int mode));
FORWARD _PROTOTYPE ( int mcd_send_command, (int command));
FORWARD _PROTOTYPE ( int mcd_get_disk_info, (void));
FORWARD _PROTOTYPE ( int mcd_read_q_channel, (struct cd_toc_entry *qc));
FORWARD _PROTOTYPE ( int mcd_read_toc, (void));
FORWARD _PROTOTYPE ( int ioctl_read_toc, (message *m_ptr));
FORWARD _PROTOTYPE ( int ioctl_disk_info, (message *m_ptr));
FORWARD _PROTOTYPE ( int ioctl_read_sub, (message *m_ptr));
FORWARD _PROTOTYPE ( int ioctl_disk_info, (message *m_ptr));
FORWARD _PROTOTYPE ( int ioctl_play_mss, (message *m_ptr));
FORWARD _PROTOTYPE ( int ioctl_play_ti, (message *m_ptr));
FORWARD _PROTOTYPE ( int mcd_open, (struct driver *dp, message *m_ptr));
FORWARD _PROTOTYPE ( int mcd_close, (struct driver *dp, message *m_ptr));
FORWARD _PROTOTYPE ( int mcd_ioctl, (struct driver *dp, message *m_ptr));
FORWARD _PROTOTYPE ( char *mcd_name, (void));
FORWARD _PROTOTYPE ( struct device *mcd_prepare, (int dev));
FORWARD _PROTOTYPE ( int mcd_schedule, (int proc_nr, struct iorequest_s *iop));
FORWARD _PROTOTYPE ( int mcd_finish, (void));
FORWARD _PROTOTYPE ( void mcd_geometry, (struct partition *entry));

/* Flags displaying current status of cdrom, used with the McdStatus variable */
#define TOC_UPTODATE   0x001     /* Table of contents is up to date */
#define INFO_UPTODATE  0x002     /* Disk info is up to date */
#define DISK_CHANGED   0x004     /* Disk has changed */
#define AUDIO_PLAYING  0x008     /* Cdrom is playing audio */
#define AUDIO_PAUSED   0x010     /* Cdrom is paused (only audio) */
#define AUDIO_DISK     0x020     /* Disk contains audio */
#define DISK_ERROR     0x040     /* An error occured */
#define NO_DISK        0x080     /* No disk in device */
```

```
/* Entry points to this driver. */
PRIVATE struct driver mcd_dtab =
{
#if __minix_vmd
  NULL,             /* No private request buffer */
#endif
  mcd_name,         /* Current device's name */
  mcd_open,         /* Open request read table of contents */
  mcd_close,        /* Release device */
  mcd_ioctl,        /* Do cdrom ioctls */
  mcd_prepare,      /* Prepare for I/O */
  mcd_schedule,     /* Precompute blocks */
  mcd_finish,       /* Do the I/O */
  nop_cleanup,      /* No cleanup to do */
  mcd_geometry      /* Tell geometry */
};


PRIVATE struct trans
{
  struct iorequest_s *tr_iop;    /* Belongs to this I/O request */
  unsigned long tr_pos;          /* Byte position to transfer from */
  int tr_count;                  /* Byte count */
  phys_bytes tr_phys;            /* User physical address */
} mcd_trans[NR_IOREQS];


/* Globals */
#if __minix_vmd
PRIVATE int mcd_tasknr = ANY;
#endif

PRIVATE int mcd_avail;                /* Set if Mitsumi device exists */
PRIVATE int mcd_irq;                  /* Interrupt request line */
PRIVATE int mcd_io_base;         /* I/O base register */

/* Partición Activa
   Estructura que se utiliza para guardar la dirección base de un dispositivo
  o partición y el tamaño de éste en bytes.  */
PRIVATE struct device *mcd_dv;   /* Active partition */

PRIVATE struct trans *mcd_tp;          /* Pointer to add transfer requests */

/* Número de bytes a transferir * /
PRIVATE unsigned mcd_count;            /* Number of bytes to transfer */

PRIVATE unsigned long mcd_nextpos;     /* Next consecutive position on disk */

/* Seleccionar partición
  DEV_PER_DRIVE : Número de particiones + 1 = 4 + 1 = 5
  SUB_PER_DRIVE : Número de particiones * Número de particiones = 4 * 4    */

/* Partición Primaria */
PRIVATE struct device mcd_part[DEV_PER_DRIVE];
                                /* Primary partitions: cd[0-4] */
/* Subpartición */
PRIVATE struct device mcd_subpart[SUB_PER_DRIVE];
                                /* Subpartitions: cd[1-4][a-d] */

/* Número de referencias a la unidad, número de open's realizados */
PRIVATE int mcd_open_ct;            /* in-use count */

PRIVATE int McdStatus = NO_DISK;        /* A new (or no) disk is inserted */
PRIVATE struct cd_play_mss PlayMss;     /* Keep track of where we are if we
                                           pause, used by resume */
/* Almacena la cabecera TOC */
PRIVATE struct cd_disk_info DiskInfo;   /* Contains toc header */
PRIVATE struct cd_toc_entry Toc[MAX_TRACKS];  /* Buffer for toc */
```

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

Es el punto de entrada al driver, y la única función accesible desde el exterior. Realiza una serie de configuraciones previas (configura la dirección base de los puertos de entrada salida, la interrupción asociada al CD-ROM y pasa una serie de variables al entorno) antes de llamar a la función *driver_task*. Esta función es invocada por todos los drivers de dispositivos de bloques y se le pasa una estructura tipo driver conteniendo punteros a las funciones propias de cada driver.

```
PUBLIC void mcd_task()
{
  long v;
  static char var[] = "MCD";
  static char fmt[] = "x:d";

#if __minix_vmd
  mcd_tasknr = proc_number(proc_ptr);
#endif

  /* Configure I/O base and IRQ. */
  v = MCD_IO_BASE_ADDRESS;
  (void) env_parse(var, fmt, 0, &v, 0x000L, 0x3FFL);
  mcd_io_base = v;

  v = MCD_IRQ;
  (void) env_parse(var, fmt, 0, &v, 0L, (long) NR_IRQ_VECTORS - 1);
  mcd_irq = v;

  driver_task(&mcd_dtab);       /* Start driver task for cdrom */
}
```

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

Esta función verifica que el CDROM es accesible y devuelve un mensaje de error en caso de que no lo sea. Espera a que se introduzca un CD e intenta leer el TOC del mismo y rellenar la estructura device. Además, calcula el número total de bytes en el CD. Incrementa la variable global *mcd_open_ct*, que lleva la cuenta del número de open's. Al final del open, se llama a la función *partition*, que inicializa la tabla de particiones del dispositivo.

```
PRIVATE int mcd_open(dp, m_ptr)
struct driver *dp;  /* pointer to this drive */
message *m_ptr;           /* OPEN */
{
  int i, status;
  /*
     La variable mcd_avail la inicializa la función mcd_init al
    inicilizar la unidad. La pone a 1.
     EIO : error de entrada salida
  */
  if (!mcd_avail && mcd_init() != OK) return EIO;

  /* Selecciona la partición del dispositivo */
  /* ENXIO : no reconoce dispositivo o dirección */
  if (mcd_prepare(m_ptr->DEVICE) == NIL_DEV) return ENXIO;
```

```
  /* W_BIT bit de protección de escritura */
  /* EACCES : permiso denegado */
  /* A CD-ROM is read-only by definition. */
  if (m_ptr->COUNT & W_BIT) return EACCES;

  /* Primer open */
  if (mcd_open_ct == 0)
  {
    i = 20;
    for (;;) {
      /* Si no puede obtener el estado, devolver EIO */
      if (mcd_get_status(1) == -1) return EIO;   /* set McdStatus flags */
      /* Hay disco ? */
      if (!(McdStatus & NO_DISK)) break;
      /*
       if (--i == 0) return EIO;
      milli_delay(100);
    }
    /* Try to read the table of contents of the CD currently inserted */
    if ((status = mcd_read_toc()) != OK)
      return status;
    /* Incrementa el numero de accesos */
    mcd_open_ct++;

    /* Rellena la estructura device */
    /* fill in size of device (= nr. of bytes on the disk) */
    mcd_part[0].dv_base = 0;
    mcd_part[0].dv_size =
     ((((unsigned long)DiskInfo.disk_length_mss[MINUTES] * SECONDS_PER_MINUTE
      + (unsigned long)DiskInfo.disk_length_mss[SECONDS]) * SECTORS_PER_SECOND)
      + (unsigned long)DiskInfo.disk_length_mss[SECTOR]) * BYTES_PER_SECTOR;

#if MCD_DEBUG >= 1
    printf("cd size: %lu\n", mcd_part[0].dv_size);
#endif
    /* Se llama a la función 'partition' para que se inicialice la tabla de
      particiones del dispositivo */
    /* Partition the disk. */
    partition(&mcd_dtab, 0, P_PRIMARY);
  }
  return OK;
}


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

Decrementa la variable *mcd_open_ct*, que lleva la cuenta de las referencias (open's) que se han hecho a la unidad.

```
PRIVATE int mcd_close(dp, m_ptr)
struct driver *dp;  /* pointer to this drive */
message *m_ptr;           /* CLOSE */
{
  /* El dispositivo tiene una referencia menos */
  mcd_open_ct--;
  return OK;
}



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

Devuelve un nombre para el dispositivo. En este caso, siempre es '*cd0*'.

```
PRIVATE char *mcd_name()
```

```
{
  /* Return a name for the device */
  return "cd0";
}


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

Realiza operaciones genéricas para cambiar el estado del dispositivo. Por ejemplo, iniciar la reproducción de una canción, parar la reproducción, hacer pausa de reproducción, expulsar la bandeja, leer TOC, leer la cabecera del TOC. En función de la operación genérica solicitada, se llama a la función adecuada.

```
/* Operaciones genéricas para cambiar el estado del dispositivo */
PRIVATE int mcd_ioctl(dp, m_ptr)
struct driver *dp;  /* pointer to the drive */
message *m_ptr;           /* contains ioctl command */
{
  /* Perform the ioctl request */

  int status;

  if (mcd_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  mcd_get_status(1);      /* Update the status flags */
  /* Si no hay disco y la petición es distinta de sacar la bandeja, error */
  if ((McdStatus & NO_DISK) && m_ptr->REQUEST != CDIOEJECT)
    return EIO;

  switch(m_ptr->REQUEST)
  {
    case CDIOPLAYMSS:     status = ioctl_play_mss(m_ptr);break;
    case CDIOPLAYTI:      status = ioctl_play_ti(m_ptr);break;
    case CDIOREADTOCHDR:  status = ioctl_disk_info(m_ptr);break;
    case CDIOREADTOC:     status = ioctl_read_toc(m_ptr);break;
    case CDIOREADSUBCH:   status = ioctl_read_sub(m_ptr);break;
    case CDIOSTOP:        status = mcd_stop();break;
    case CDIOPAUSE:       status = mcd_pause();break;
    case CDIORESUME:      status = mcd_resume();break;
    case CDIOEJECT:       status = mcd_eject();break;
    default:              status = do_diocntl(dp, m_ptr);
  }
  return status;
}
```

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

Espera para obtener una respuesta de la unidad tras el envío de un comando. Para ello comprueba que la unidad esté lista (*mcd_ready*), y en caso afirmativo obtiene la respuesta. Si la unidad no estaba preparada (no responde) devuelve un Error de Entrada/Salida (EIO).

```
PRIVATE int mcd_get_reply(reply, delay)
u8_t *reply;        /* variable to put reply in */
int delay;          /* count to wait for the reply */
{
  /* Get a reply from the drive */

  if (mcd_ready(delay) != OK) return EIO;           /* wait for drive to
                                                       become available */
  *reply = in_byte(MCD_DATA_PORT);     /* get the reply */
  return OK;
}
```

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

Esta función espera por la unidad hasta que esté de nuevo disponible. Para ello entra en un bucle preguntando continuamente si el valor del puerto de flag indica que la unidad está ocupada. En cuanto no lo esté, devuelve OK. Si transcurre un cierto intervalo de tiempo y la unidad no responde, se devuelve un Error de Entrada/Salida (EIO).

```
PRIVATE int mcd_ready(delay)
int delay;   /* count to wait for drive to become available again */
{
  /* Wait for drive to become available */

  struct milli_state ms;

  milli_start(&ms);
  do
 {
    /* Si la unidad está ocupada, esperamos, en caso contrario devolvemos OK. No
esperamos eternamente. Si se cumple un tiempo de guarda retornamos un Error de
Entrada/Salida (EIO) */
    if (!(in_byte(MCD_FLAG_PORT) & MCD_BUSY)) return OK; /* OK, drive ready */
 } while(milli_elapsed(&ms) < delay);

  return EIO; /* Timeout */
}
```

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

Espera por la unidad hasta que ésta le indique que los datos están disponibles. Para ello pregunta continuamente a la unidad si los datos están preparados. Desde que lo estén, devuelve un OK. Si transcurrido un tiempo de guarda la unidad no responde, se retorna un EIO.

```
PRIVATE int mcd_data_ready(delay)
int delay;          /* count to wait for the data */
{
  /* Wait for the drive to get the data */
  struct milli_state ms;

  milli_start(&ms);

  do
  {
    if (!(in_byte(MCD_FLAG_PORT) & 2)) return OK; /* OK, data is there */
  } while(milli_elapsed(&ms) < delay);

  return EIO;  /* Timeout */
}
```

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

Devuelve información de estado de la unidad y actualiza la variable global *McdStatus*. Se le pasa un flag a esta función cuyo significado es el siguiente: si es 1, se le envía un comando a la unidad para obtener el estado; si no, se obtiene el estado directamente de la unidad, se lee del puerto de flag (no se envía un comando explícito para obtener el estado). En caso de no poder obtener el estado, devuelve un -1.

```
PRIVATE int mcd_get_status(f)
int f;              /* flag */
{
  /* Return status info from the drive and update the global McdStatus */

  u8_t status;

  /* If f = 1, we first send a get_status command, otherwise we just get
     the status info from the drive */

  if (f) out_byte(MCD_DATA_PORT, MCD_GET_STATUS);        /* Try to get status */
  if (mcd_get_reply(&status,REPLY_DELAY) != OK) return -1;

  McdStatus &= ~(NO_DISK | DISK_CHANGED | DISK_ERROR);

  /* Fill in the McdStatus variable */
  if (status & MCD_DOOR_OPEN ||
     !(status & MCD_DISK_IN))         McdStatus = NO_DISK;
  else if (status & MCD_DISK_CHANGED) McdStatus = DISK_CHANGED;
  else if (status & MCD_READ_ERROR ||
           status & MCD_CMD_ERROR)    McdStatus = DISK_ERROR;
  else
  {
    if (status & MCD_AUDIO_DISK)
    {
      McdStatus |= AUDIO_DISK;
```

```
      if (!(status & MCD_AUDIO_BUSY)) McdStatus &= ~(AUDIO_PLAYING);
      else McdStatus |= AUDIO_PLAYING;
    }
  }
#if MCD_DEBUG >= 3
  printf("mcd_get_status(%d) = %02x, McdStatus = %02x\n",
       f, status, McdStatus);
#endif
  return status;    /* Return status */
}


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

Establece un modo para la unidad. Envía el comando *establecer modo* a la unidad y a continuación le envía el modo concreto que se quiere establecer. Lo intenta un número de veces *MCD_RETRIES*, tras lo cual retorna un EIO.

```
PRIVATE int mcd_set_mode(mode)
int mode; /* new drive mode */
{
  /* Set drive mode */
  int i;

  for (i = 0; i < MCD_RETRIES; i++)
  {
    out_byte(MCD_DATA_PORT, MCD_SET_MODE); /* Send set mode command */
    out_byte(MCD_DATA_PORT, mode);         /* Send which mode */
    /* Si el estado después del comando es != -1 (no error) y no se ha
      producido un Error de Disco, retornamos OK */
    if (mcd_get_status(0) != -1 &&
        !(McdStatus & DISK_ERROR)) break;
  }
  if (i == MCD_RETRIES) return EIO;        /* Operation failed */

  return OK; /* Operation succeeded */
}
/*==========================================================================*
 *                          mcd_send_command                                *
 *==========================================================================*/
```

Envía un comando a la unidad. Lo intenta un número de veces *MCD_RETRIES*, tras lo cual retorna un EOI. Si todo va bien, devuelve OK.

```
PRIVATE int mcd_send_command(command)
int command;        /* command to send */
{
  int i;

  for (i = 0; i < MCD_RETRIES; i++)
  {
    out_byte(MCD_DATA_PORT, command);      /* send command */
    if (mcd_get_status(0) != -1 &&
        !(McdStatus & DISK_ERROR)) break;
  }
  if (i == MCD_RETRIES) return EIO;        /* operation failed */

  return OK;
}
```

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

Inicializa la unidad y obtiene los bytes de versión de la misma (modelo en concreto). Para inicializar la unidad resetea el puerto de flag y elimina los datos pendientes. Tras esto, comprueba el estado y si es el correcto obtiene la versión del CD y la muestra por pantalla. Por último registra la interrupción del CD-ROM y establece su manejador ante una interrupción. Habilita la interrupción y pone a 1 la variable global que indica que la unidad está disponible (*mcd_avail*). En caso de error devuelve un -1, y si todo va bien devuelve OK.

```
PRIVATE int mcd_init()
{
  /* Initialize the drive and get the version bytes, this is done only
     once when the system gets up. We can't use mcd_ready because
     the clock task is not available yet.
   */

  u8_t version[3];
  int i;
  u32_t n;
  struct milli_state ms;

  /* Reset the flag port and remove all pending data, if we do
   * not do this properly the drive won't cooperate.
   */
  out_byte(MCD_FLAG_PORT, 0x00);
  for (n = 0; n < 1000000; n++)
    (void) in_byte(MCD_FLAG_PORT);

  /* Now see if the drive will report its status */
  if (mcd_get_status(1) == -1)
  {
    /* Too bad, drive will not listen */
    printf("%s: init failed, no Mitsumi cdrom present\n", mcd_name());
    return -1;
  }

  /* Find out drive version */
  out_byte(MCD_DATA_PORT, MCD_GET_VERSION);
  milli_start(&ms);
  for (i = 0; i < 3; i++)
  {
    while (in_byte(MCD_FLAG_PORT) & MCD_BUSY)
    {
      if (milli_elapsed(&ms) >= 1000)
      {
       printf("%s: can't get version of Mitsumi cdrom\n", mcd_name());
       return -1;
      }
    }
    version[i] = in_byte(MCD_DATA_PORT);
  }

  if (version[1] == 'D')
    printf("%s: Mitsumi FX001D CD-ROM\n", mcd_name());
  else
    printf("%s: Mitsumi CD-ROM version %02x%02x%02x\n", mcd_name(),
            version[0], version[1], version[2]);

  /* Newer drive models need this */
  if (version[1] >= 4) out_byte(MCD_CONTROL_PORT, MCD_PICKLE);
```

```
  /* Register interrupt vector and enable interrupt
   * currently the interrupt is not used because
   * the controller isn't set up to do dma.  XXX
   */
  put_irq_handler(mcd_irq, c_handler);
  enable_irq(mcd_irq);

  mcd_avail = 1;
  return OK;
}


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

Esta función es llamada cuando se produce la interrupción asociada al CD-ROM. En principio nunca se producen, pero este código está por si acaso. Lo que hace es enviar un mensaje de interrupción a la tarea asociada al CD-ROM.

```
PRIVATE int c_handler(irq)
int irq;     /* irq number */
{
  /* The interrupt handler, I never got an interrupt but its here just
   * in case...
   */
  /* Enviamos un mensaje de interrupción al manejador */
#if XXX
#if __minix_vmd
  interrupt(mcd_tasknr);
#else
  interrupt(CDROM);
#endif
#endif

  return 1;
}


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

Indica a la unidad que comience la reproducción de un CD de audio en una posición determinada, que viene dada en el formato minuto:segundo:sector. Si todo va bien, comienza la reproducción y se actualiza la variable que guarda el estado de la unidad. Ha de indicarse la posición de comienzo y final de la reproducción, que se envía a la unidad codificado en BCD. La posición final de la reproducción es guardada, por si se desea continuar con la misma.

```
PRIVATE int mcd_play_mss(mss)
struct cd_play_mss mss;  /* from where to play minute:second.sector */
{
  /* Command the drive to start playing at min:sec.sector */

  int i;

#if MCD_DEBUG >= 1
  printf("Play_mss: begin: %02d:%02d.%02d  end: %02d:%02d.%02d\n",
          mss.begin_mss[MINUTES], mss.begin_mss[SECONDS],
```

```
          mss.begin_mss[SECTOR], mss.end_mss[MINUTES],
          mss.end_mss[SECONDS], mss.end_mss[SECTOR]);
#endif

  for (i=0; i < MCD_RETRIES; i++)     /* Try it more than once */
  {
    /* Mientras se esté enviando este comando a la unidad no se permmiten
      interrupciones */
    lock();        /* No interrupts when we issue this command */

    /* Le enviamos el comando a la unidad para leer desde una posición
      a otra determinada */
    out_byte(MCD_DATA_PORT, MCD_READ_FROM_TO);
    out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[MINUTES]));
    out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[SECONDS]));
    out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[SECTOR]));
    out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[MINUTES]));
    out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[SECONDS]));
    out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[SECTOR]));

   /* Permitimos las interrupciones de nuevo */
    unlock();
   /* ¿Ha ido todo bien ? */
    mcd_get_status(0);                  /* See if all went well */
    if (McdStatus & AUDIO_PLAYING) break; /* Ok, we're playing */
  }

  if (i == MCD_RETRIES) return EIO;       /* Command failed */

  /* keep in mind where we going in case of a future resume */
  PlayMss.end_mss[MINUTES] = mss.end_mss[MINUTES];
  PlayMss.end_mss[SECONDS] = mss.end_mss[SECONDS];
  PlayMss.end_mss[SECTOR] = mss.end_mss[SECTOR];

  /* Actualizamos el estado de la unidad, borramos el bit de PAUSED */
  McdStatus &= ~(AUDIO_PAUSED);

  return(OK);
}


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

Esta función se utiliza para reproducir un conjunto de pistas consecutivas, indicándose a la unidad las pistas inicial y final. Se comprueba que éstas son correctas leyendo la tabla de contenidos, luego se pasan ambas pistas a formato minuto:segundo:sector y se llama finalmente a la función *mcd_play_mss* con esta información para que inicie la reproducción.

```
PRIVATE int mcd_play_tracks(tracks)
struct cd_play_track tracks;     /* which tracks to play */
{
  /* Command the drive to play tracks */

  int i, err;
  struct cd_play_mss mss;

#if MCD_DEBUG >= 1
  printf("Play tracks: begin: %02d end: %02d\n",
           tracks.begin_track, tracks.end_track);
#endif

  /* First read the table of contents */
  if ((err = mcd_read_toc()) != OK) return err;
```

```
  /* Si los parámetros no son correctos, se retorna un EINVAL : argumento
    no es correcto */

  /* Check if parameters are valid */
  if (tracks.begin_track < DiskInfo.first_track ||
      tracks.end_track > DiskInfo.last_track ||
      tracks.begin_track > tracks.end_track)
    return EINVAL;


  /* Convert the track numbers to min:sec.sector */
  for (i=0; i<3; i++)
  {
    mss.begin_mss[i] = Toc[tracks.begin_track].position_mss[i];
    mss.end_mss[i] = Toc[tracks.end_track+1].position_mss[i];
  }

  return(mcd_play_mss(mss));     /* Start playing */
}



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

Obtiene información del disco insertado en la unidad, para lo que le envía el comando *MCD_GET_VOL_INFO*. Con la información que obtiene rellena la variable *DiskInfo*, del tipo *cd_disk_info*. La información devuelta por la unidad está en formato BCD, por lo que la rutina se encarga de pasarla a binario. Al final, actualiza el estado de la variable global *McdStatus* y retorna. Entre las informaciones que proporciona podemos destacar el comienzo de la primera pista y el tamaño total del disco, ambos en formato minuto:segundo:sector.

```
PRIVATE int mcd_get_disk_info()
{
  /* Get disk info */

  int i, err;

  if (McdStatus & INFO_UPTODATE) return OK; /* No need to read info again */

  /* Issue the get volume info command */
  if ((err = mcd_send_command(MCD_GET_VOL_INFO)) != OK) return err;

  /* Fill global DiskInfo */
  for (i=0; i < sizeof(DiskInfo); i++)
  {
    if ((err = mcd_get_reply((u8_t *)(&DiskInfo) + i, REPLY_DELAY)) !=OK)
      return err;
    bcd2bin((u8_t *)(&DiskInfo) + i);
  }

#if MCD_DEBUG >= 1
  printf("Mitsumi disk info: first: %d last: %d first %02d:%02d.%02d length:
%02d:%02d.%02d\n",
      DiskInfo.first_track,
      DiskInfo.last_track,
      DiskInfo.first_track_mss[MINUTES],
      DiskInfo.first_track_mss[SECONDS],
      DiskInfo.first_track_mss[SECTOR],
      DiskInfo.disk_length_mss[MINUTES],
```

```
      DiskInfo.disk_length_mss[SECONDS],
      DiskInfo.disk_length_mss[SECTOR]);
#endif
  /* Update global status info */
  McdStatus |= INFO_UPTODATE; /* toc header has been read */
  McdStatus &= ~TOC_UPTODATE; /* toc has not been read yet */

  return OK;
}


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

Lee información del canal Q. Si actualmente estamos reproduciendo, devuelve las posiciones relativa y absoluta de dónde estamos, en el ya conocido formato min:seg:sec. Si no estamos reproduciendo, devuelve una entrada de la tabla de contenidos. Con la información que obtiene rellena una variable del tipo *cd_toc_entry*.

```
PRIVATE int mcd_read_q_channel(qc)
struct cd_toc_entry *qc;  /* struct to return q-channel info in */
{
  /* Read the qchannel info, if we we're already playing this returns
   * the relative position and the absolute position of where we are
   * in min:sec.sector. If we're not playing, this returns an entry
   * from the table of contents
   */

  int i, err;

  /* Issue the command */
  if ((err = mcd_send_command(MCD_GET_Q_CHANNEL)) != OK) return err;

  /* Read the info */
  for (i=0; i < sizeof(struct cd_toc_entry); i++)
  {
    /* Return error on timeout */
    if ((err = mcd_get_reply((u8_t *)qc + i, REPLY_DELAY)) != OK)
      return err;

    bcd2bin((u8_t *)qc + i);  /* Convert value to binary */
  }
#if MCD_DEBUG >= 2
  printf("qchannel info: ctl_addr: %d track: %d index: %d length %02d:%02d.%02d
pos: %02d:%02d.%02d\n",
      qc->control_address,
      qc->track_nr,
      qc->index_nr,
      qc->track_time_mss[MINUTES],
      qc->track_time_mss[SECONDS],
      qc->track_time_mss[SECTOR],
      qc->position_mss[MINUTES],
      qc->position_mss[SECONDS],
      qc->position_mss[SECTOR]);
#endif

  return OK;  /* All done */
}
```

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

Lee la tabla de contenidos del CD y la pasa al buffer de la misma en memoria. Para ello, efectúa hasta 600 lecturas arbitrarias del canal Q hasta tener toda la tabla.

```
PRIVATE int mcd_read_toc()
{
  /* Read the table of contents (TOC) */

  struct cd_toc_entry q_info;
  int current_track, current_index;
  int err,i;

  if (McdStatus & TOC_UPTODATE) return OK; /* No need to read toc again */

  /* Clear toc table */
  for (i = 0; i < MAX_TRACKS; i++) Toc[i].index_nr = 0;

  /* Read disk info */
  if ((err = mcd_get_disk_info()) != OK) return err;

  /* Calculate track to start with */
  current_track = DiskInfo.last_track - DiskInfo.first_track + 1;

  /* Set read toc mode */
  if ((err = mcd_set_mode(MCD_TOC)) != OK) return err;

  /* Read the complete TOC, on every read-q-channel command we get a random
   * TOC entry depending on how far we are in the q-channel, collect entries
   * as long as we don't have the complete TOC. There's a limit of 600 here,
   * if we don't have the complete TOC after 600 reads we quit with an error
   */
  for (i = 0; (i < 600 && current_track > 0); i++)
  {
    /* Try to read a TOC entry */
    if ((err = mcd_read_q_channel(&q_info)) != OK) break;

    /* Is this a valid track number and didn't we have it yet ? */
    current_index = q_info.index_nr;
    if (current_index >= DiskInfo.first_track &&
        current_index <= DiskInfo.last_track &&
        q_info.track_nr == 0)
    {
      /* Copy entry into toc table */
      if (Toc[current_index].index_nr == 0)
     {
      Toc[current_index].control_address = q_info.control_address;
      Toc[current_index].track_nr = current_index;
      Toc[current_index].index_nr = 1;
      Toc[current_index].track_time_mss[MINUTES]=q_info.track_time_mss[MINUTES];
      Toc[current_index].track_time_mss[SECONDS]= q_info.track_time_mss[SECONDS];
      Toc[current_index].track_time_mss[SECTOR] = q_info.track_time_mss[SECTOR];
      Toc[current_index].position_mss[MINUTES] = q_info.position_mss[MINUTES];
      Toc[current_index].position_mss[SECONDS] = q_info.position_mss[SECONDS];
      Toc[current_index].position_mss[SECTOR] = q_info.position_mss[SECTOR];
      current_track--;
     }
    }
  }
  if (err) return err;     /* Do we have all toc entries? */

  /* Fill in lead out */
  current_index = DiskInfo.last_track + 1;
```

```
  Toc[current_index].control_address = Toc[current_index-1].control_address;
  Toc[current_index].track_nr = 0;
  Toc[current_index].index_nr = LEAD_OUT;
  Toc[current_index].position_mss[MINUTES] = DiskInfo.disk_length_mss[MINUTES];
  Toc[current_index].position_mss[SECONDS] = DiskInfo.disk_length_mss[SECONDS];
  Toc[current_index].position_mss[SECTOR] = DiskInfo.disk_length_mss[SECTOR];

  /* Return to cooked mode */
  if ((err = mcd_set_mode(MCD_COOKED)) != OK) return err;

  /* Update global status */
  McdStatus |= TOC_UPTODATE;

#if MCD_DEBUG >= 1
  for (i = DiskInfo.first_track; i <= current_index; i++)
  {
    printf("Mitsumi toc %d: trk %d  index %d  time %02d:%02d.%02d  pos:
%02d:%02d.%02d\n",
        i,
        Toc[i].track_nr,
        Toc[i].index_nr,
        Toc[i].track_time_mss[MINUTES],
        Toc[i].track_time_mss[SECONDS],
        Toc[i].track_time_mss[SECTOR],
        Toc[i].position_mss[MINUTES],
        Toc[i].position_mss[SECONDS],
        Toc[i].position_mss[SECTOR]);
  }
#endif

  return OK;
}


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

Envía el comando *MCD_STOP* a la unidad y actualiza el estado de la variable global *McdStatus*.

```
PRIVATE int mcd_stop()
{
  int err;

  if ((err = mcd_send_command(MCD_STOP)) != OK ) return err;
  McdStatus &= ~(AUDIO_PAUSED);
  return OK;
}


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

Envía un comando a la unidad para que expulse el CD de la unidad.

```
PRIVATE int mcd_eject()
{
  int err;

  if ((err = mcd_send_command(MCD_EJECT)) != OK) return err;
  return OK;
}
```

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

Envía el comando *MCD_STOP_AUDIO* a la unidad para detener por un tiempo la reproducción. Se ha de estar en modo reproducción para poder enviar este comando. Almacena la posición actual de reproducción (para continuar posteriormente donde estábamos) y se actualiza la variable global de estado. Para averiguar la posición actual, que se guarda en la variable *PlayMss*, se hace una llamada a la función *mcd_read_q_channel*.

```
PRIVATE int mcd_pause()
{
  int err;
  struct cd_toc_entry qc;

  /* We can only pause when we are playing audio */
  if (!(McdStatus & AUDIO_PLAYING)) return EINVAL;

  /* Look where we are */
  if ((err = mcd_read_q_channel(&qc)) != OK) return err;

  /* Stop playing */
  if ((err = mcd_send_command(MCD_STOP_AUDIO)) != OK) return err;

  /* Keep in mind were we have to start again */
  PlayMss.begin_mss[MINUTES] = qc.position_mss[MINUTES];
  PlayMss.begin_mss[SECONDS] = qc.position_mss[SECONDS];
  PlayMss.begin_mss[SECTOR] = qc.position_mss[SECTOR];

  /* Update McdStatus */
  McdStatus |= AUDIO_PAUSED;

#if MCD_DEBUG >= 1
  printf("Mcd_paused at: %02d:%02d.%02d\n",
      PlayMss.begin_mss[MINUTES],
      PlayMss.begin_mss[SECONDS],
      PlayMss.begin_mss[SECTOR]);
#endif

  return OK;
}
```

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

Reanuda la reproducción de un CD que había sido parada (estará en estado *AUDIO_PAUSED*). Recuperamos la información de dónde dejamos la reproducción de la variable *PlayMss*; llamamos a la función *mcd_play_mss* con esta información y por último actualizamos la variable global de estado.

```
PRIVATE int mcd_resume()
{
  int err;
  /* Solamente podemos reanudar si estamos en el estado de AUDIO_PAUSED */
  /* EINVAL : Argumento no válido */
  if (!(McdStatus & AUDIO_PAUSED)) return EINVAL;

  /* start playing where we left off */
  if ((err = mcd_play_mss(PlayMss)) != OK) return err;
```

```
  McdStatus &= ~(AUDIO_PAUSED);

#if MCD_DEBUG >= 1
  printf("Mcd resumed at: %02d:%02d.%02d\n",
      PlayMss.begin_mss[MINUTES],
      PlayMss.begin_mss[SECONDS],
      PlayMss.begin_mss[SECTOR]);
#endif

  return OK;
}


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

Lee un subcanal, mediante la llamada a *mcd_read_q_channel*. Esta función solamente tiene sentido si estamos en modo reproducción de audio. Después de leer la información la copia en la zona del usuario.

```
PRIVATE int ioctl_read_sub(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  struct cd_toc_entry sub;
  int err;

  /* We can only read a sub channel when we are playing audio */
  if (!(McdStatus & AUDIO_PLAYING)) return EINVAL;

  /* Read the sub channel */
  if ((err = mcd_read_q_channel(&sub)) != OK) return err;

  /* Copy info to user */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(sub));
  if (user_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&sub), user_phys, (phys_bytes) sizeof(sub));

  return OK;
}


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

Intenta leer la tabla de contenidos (TOC) -para lo que emplea la función *mcd_read_toc*- calcula su tamaño y lo copia a la zona de usuario.

```
PRIVATE int ioctl_read_toc(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  int err, toc_size;

  /* Try to read the table of contents */
  if ((err = mcd_read_toc()) != OK) return err;

  /* Get size of toc */
  toc_size = (DiskInfo.last_track + 1) * sizeof(struct cd_toc_entry);

  /* Copy to user */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, toc_size);
  if (user_phys == 0) return(EFAULT);
```

```
  phys_copy(vir2phys(&Toc), user_phys, (phys_bytes) toc_size);

  return OK;
}


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

Intenta leer la cabecera del TOC, que contiene información del disco, y la copia en la zona de usuario.

```
PRIVATE int ioctl_disk_info(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  int err;

  /* Try to read the toc header */
  if ((err = mcd_get_disk_info()) != OK) return err;


  /* Copy info to user */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(DiskInfo));
  if (user_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&DiskInfo), user_phys, (phys_bytes) sizeof(DiskInfo));

  return OK;
}



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

Inicia la reproducción del CD en el lugar indicado en la zona de usuario, mediante el formato min:seg:sec. Para comenzar la reproducción, llama a la función *mcd_play_mss*.

```
PRIVATE int ioctl_play_mss(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  struct cd_play_mss mss;

  /* Get user data */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(mss));
  if (user_phys == 0) return(EFAULT);
  phys_copy(user_phys, vir2phys(&mss), (phys_bytes) sizeof(mss));

  /* Try to play */
  return mcd_play_mss(mss);
}
```

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

Obtiene información del usuario para comenzar la reproducción de pistas, llamando a su vez a la función *mcd_play_tracks*.

```
PRIVATE int ioctl_play_ti(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  struct cd_play_track tracks;

  /* Get user data */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(tracks));
  if (user_phys == 0) return(EFAULT);
  phys_copy(user_phys, vir2phys(&tracks), (phys_bytes) sizeof(tracks));

  /* Try to play */
  return mcd_play_tracks(tracks);
}
```

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

Selecciona una partición para el dispositivo. Lo intenta primero con una partición primaria y luego con una secundaria; si no es posible, devuelve error.

```
PRIVATE struct device *mcd_prepare(device)
int device;
{
  /* Nothing to transfer as yet. */
  mcd_count = 0;

  /* Selecciona partición para el dispositivo.
    mcd_dv es un puntero a un aestructura de tipo partición
    que apunta a la partición activa. */

  if (device < DEV_PER_DRIVE) {               /* cd0, cd1, ... */
    mcd_dv = &mcd_part[device];
  } else
  /* MINOR_hd1a = 128 */
  if ((unsigned) (device -= MINOR_hd1a) < SUB_PER_DRIVE) { /* cd1a, cd1b, ... */
    mcd_dv = &mcd_subpart[device];
  }
else {
    return NIL_DEV;
  }
  return mcd_dv;
}
```

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

Reúne las peticiones sobre bloques consecutivos de modo que puedan ser leídos/escritos en un solo comando del controlador. Esto implica la suposición de que existe suficiente tiempo para obtener la siguiente petición mientras un bloque no deseado pasa por el láser durante la lectura/escritura.

```
PRIVATE int mcd_schedule(proc_nr, iop)
int proc_nr;              /* process doing the request */
struct iorequest_s *iop;  /* pointer to read or write request */
{
/* Gather I/O requests on consecutive blocks so they may be read/written
 * in one controller command.  (There is enough time to compute the next
 * consecutive request while an unwanted block passes by.)
 */
  int r, opcode;
  unsigned long pos;
  unsigned nbytes;
  phys_bytes user_phys;

  /* This many bytes to read */
  nbytes = iop->io_nbytes;

  /* From/to this position on the device */
  pos = iop->io_position;

  /* 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 or write? */
  opcode = iop->io_request & ~OPTIONAL_IO;

  /* Only read permitted on cdrom */
  if (opcode != DEV_READ) return EIO;

  /* What position on disk and how close to EOF? */
  if (pos >= mcd_dv->dv_size) return(OK);     /* At EOF */
  if (pos + nbytes > mcd_dv->dv_size) nbytes = mcd_dv->dv_size - pos;
  pos += mcd_dv->dv_base;

  if (mcd_count > 0 && pos != mcd_nextpos) {
       /* This new request can't be chained to the job being built */
       if ((r = mcd_finish()) != OK) return(r);
  }
  /* Next consecutive position. */
  mcd_nextpos = pos + nbytes;

  if (mcd_count == 0)
  {
    /* The first request in a row, initialize. */
    mcd_tp = mcd_trans;
  }

  /* Store I/O parameters */
  mcd_tp->tr_iop = iop;
  mcd_tp->tr_pos = pos;
  mcd_tp->tr_count = nbytes;
  mcd_tp->tr_phys = user_phys;

  /* Update counters */
  mcd_tp++;
  mcd_count += nbytes;
  return(OK);
}
```

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

Lleva a cabo las peticiones de entrada/salida reunidas por la función *mcd_schedule* en la estructura *mcd_trans*.

```
PRIVATE int mcd_finish()
{
/* Carry out the I/O requests gathered in mcd_trans[]. */

  struct trans *tp = mcd_trans;
  int err, errors;
  u8_t mss[3];
  unsigned long pos;
  unsigned count, n;

  if (mcd_count == 0) return(OK);      /* we're already done */

  /* Update status */
  mcd_get_status(1);
  if (McdStatus & (AUDIO_DISK | NO_DISK))
    return(tp->tr_iop->io_nbytes = EIO);

  /* Set cooked mode */
  if ((err = mcd_set_mode(MCD_COOKED)) != OK)
    return(tp->tr_iop->io_nbytes = err);
  while (mcd_count > 0)
  {
    /* Position on the CD rounded down to the CD block size */
    pos = tp->tr_pos & ~MCD_BLOCK_MASK;

    /* Byte count rounded up. */
    count = (pos - tp->tr_pos) + mcd_count;
    count = (count + MCD_BLOCK_SIZE - 1) & ~MCD_BLOCK_MASK;

    /* XXX transfer size limits? */
    if (count > MCD_BLOCK_SIZE) count = MCD_BLOCK_SIZE;

    /* Compute disk position in min:sec:sector */
    block2mss(pos >> MCD_BLOCK_SHIFT, mss);

    /* Now try to read a block */
    errors = 0;
    while (errors < MCD_RETRIES)
    {
      lock();
      out_byte(MCD_DATA_PORT, MCD_READ_FROM_TO);
      out_byte(MCD_DATA_PORT, bin2bcd(mss[MINUTES]));
      out_byte(MCD_DATA_PORT, bin2bcd(mss[SECONDS]));
      out_byte(MCD_DATA_PORT, bin2bcd(mss[SECTOR]));
      out_byte(MCD_DATA_PORT, 0);
      out_byte(MCD_DATA_PORT, 0);
      out_byte(MCD_DATA_PORT, 1);      /* XXX count in mss form? */
      unlock();

      /* Wait for data */
      if (mcd_data_ready(REPLY_DELAY) == OK) break;
      printf("Mcd: data time out\n");
      errors++;
    }
    if (errors == MCD_RETRIES) return(tp->tr_iop->io_nbytes = EIO);

    /* Prepare reading data. */
    out_byte(MCD_CONTROL_PORT, 0x04);

    while (pos < tp->tr_pos)
```

```
    {
      /* Discard bytes before the position we are really interested in. */
      n = tp->tr_pos - pos;
      if (n > DMA_BUF_SIZE) n = DMA_BUF_SIZE;
      port_read_byte(MCD_DATA_PORT, tmp_phys, n);
#if XXX
printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count
= %u\n",
count, n, 0, 0, 0, mcd_count);
#endif
      pos += n;
      count -= n;
    }

    while (mcd_count > 0 && count > 0)
    {
      /* Transfer bytes into the user buffers. */
      n = tp->tr_count;
      if (n > count) n = count;
      port_read_byte(MCD_DATA_PORT, tp->tr_phys, n);
#if XXX
printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count
= %u\n",
count, n, tp->tr_pos, tp->tr_iop->io_nbytes, tp->tr_count, mcd_count);
#endif
      tp->tr_phys += n;
      tp->tr_pos += n;
      tp->tr_iop->io_nbytes -= n;
      if ((tp->tr_count -= n) == 0) tp++;
      count -= n;
      mcd_count -= n;
    }

    while (count > 0)
    {
      /* Discard excess bytes. */
      n = count;
      if (n > DMA_BUF_SIZE) n = DMA_BUF_SIZE;
      port_read_byte(MCD_DATA_PORT, tmp_phys, n);
#if XXX
printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count
= %u\n",
count, n, 0, 0, 0, mcd_count);
#endif
      count -= n;
    }

    /* Finish reading data. */
    out_byte(MCD_CONTROL_PORT, 0x0c);
#if 0 /*XXX*/
    mcd_get_status(1);
    if (!(McdStatus & DISK_ERROR)) done = 1; /* OK, no errors */
#endif
  }

  return OK;
}
```

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

Esta función devuelve la geometría de la unidad. Aunque ésta no tiene nada que ver con la geometría de un disco normal (disco duro o disquete), se inventa para mantener la homogeneidad de los drivers.

```
PRIVATE void mcd_geometry(entry)
struct partition *entry;
{
/* The geometry of a cdrom doesn't look like the geometry of a regular disk,
 * so we invent a geometry to keep external programs happy.
 */
  entry->cylinders = (mcd_part[0].dv_size >> SECTOR_SHIFT) / (64 * 32);
  entry->heads = 64;
  entry->sectors = 32;
}

/*=============================================================================*
 *                           misc functions                                    *
 *=============================================================================*/
PRIVATE u8_t bin2bcd(u8_t b)
{
  /* Convert a number to binary-coded-decimal */
  int u,t;

  u = b%10;
  t = b/10;
  return (u8_t)(u | (t << 4));
}


PRIVATE void bcd2bin(u8_t *bcd)
{
  /* Convert binary-coded-decimal to binary :-) */

  *bcd = (*bcd >> 4) * 10 + (*bcd & 0xf);
}


PRIVATE void block2mss(block, mss)
long block;
u8_t *mss;
{
  /* Compute disk position of a block in min:sec:sector */

  block += MCD_SKIP;
  mss[MINUTES] = block/(SECONDS_PER_MINUTE * SECTORS_PER_SECOND);
  block %= (SECONDS_PER_MINUTE * SECTORS_PER_SECOND);
  mss[SECONDS] = block/(SECTORS_PER_SECOND);
  mss[SECTOR] = block%(SECTORS_PER_SECOND);
}


PRIVATE long mss2block(u8_t *mss)
{
  /* Compute block number belonging to
   * disk position min:sec:sector
   */

  return ((((unsigned long) mss[MINUTES] * SECONDS_PER_MINUTE
       + (unsigned long) mss[SECONDS]) * SECTORS_PER_SECOND)
       + (unsigned long) mss[SECTOR]) - MCD_SKIP;
}
#endif /* ENABLE_MITSUMI_CDROM */
```

INDICE

Ampliación de sistemas operativos

Driver cd-rom mitsumi

mcd.c

José Adal Vega Pérez
Mario Rodríguez Hernández