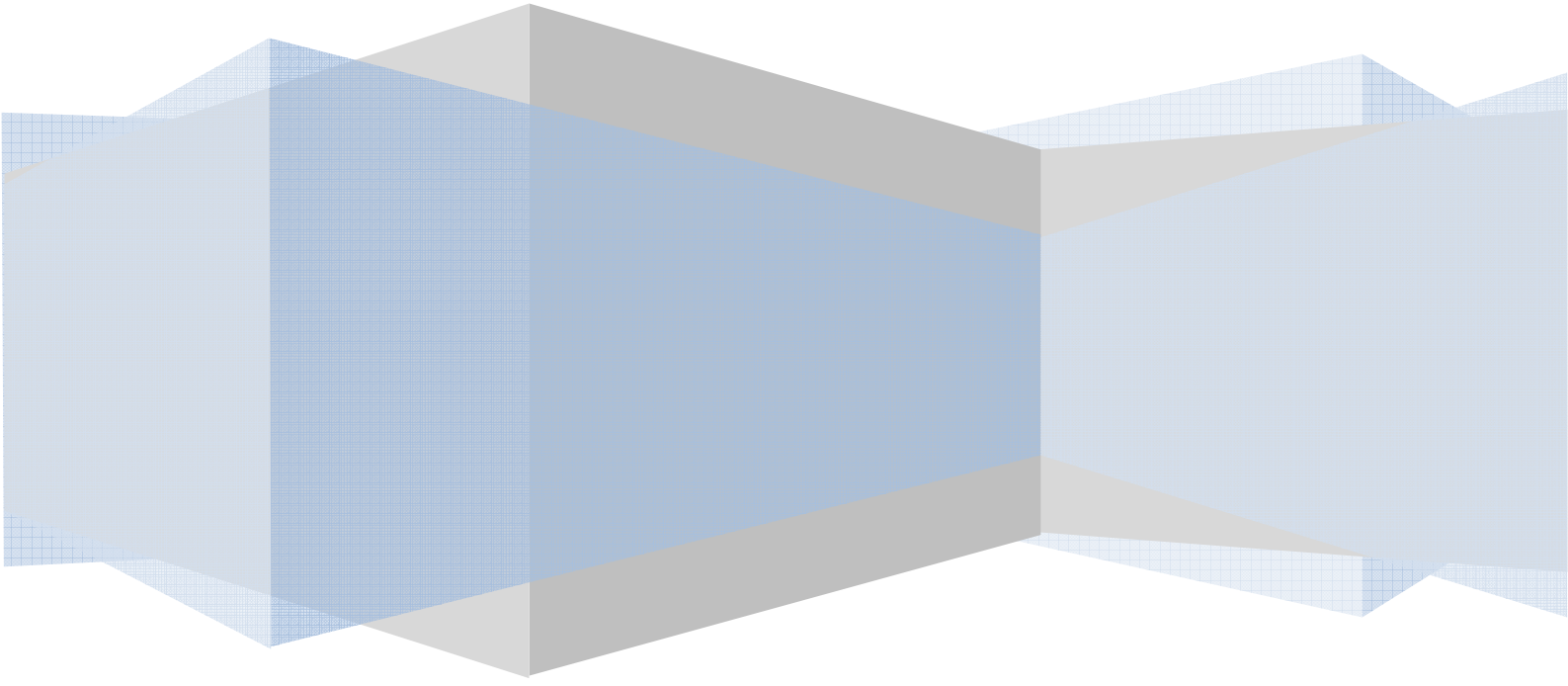
Diseño de Sistemas Operativos

CD-ROM

IDE_DRIVER



ÍNDICE

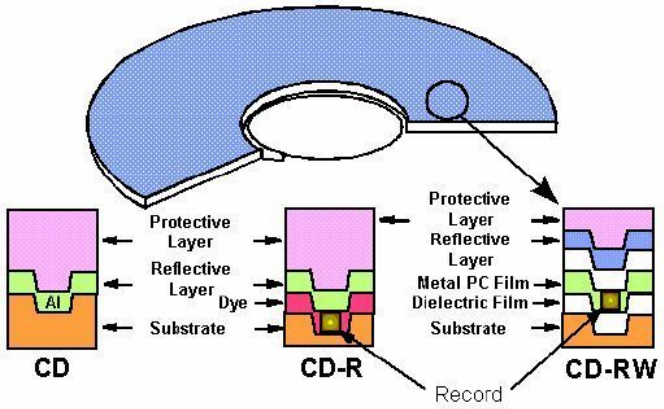
INTRODUCCIÓN

El **disco compacto** (conocido popularmente como **CD**) es un soporte digital óptico utilizado para almacenar cualquier tipo de información. Fue desarrollado conjuntamente en 1980 por las empresas Sony y Philips, y comenzó a comercializarse en 1982. Hoy en día tecnologías como el DVD han hecho que su uso sea cada vez menor.

Cada fabricante emplea diversos materiales en la fabricación de los discos. Sin embargo, todos tienen un patrón común: la información se almacena en un sustrato de policarbonato plástico. A éste se le añade una capa refractante de aluminio que reflejará la luz del láser, una capa protectora que lo cubre y de forma opcional una etiqueta en la parte superior.

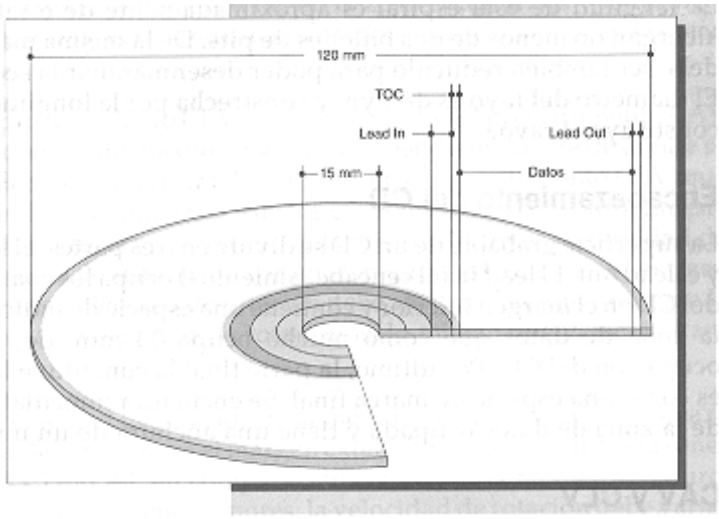
Características

- **Información almacenada**: Audio, video, imágenes, texto, datos, etc.
- **Capacidad**: Lo común es 650 ó 750 MB, aunque los hay de diferentes capacidades.
- **Forma**: Circular, con un orificio al centro.
- **Diámetro**: Originalmente 120 mm. Hay versiones reducidas de 80 mm.
- **Grosor**: 1,2 mm.
- **Material**: Policarbonato plástico con una capa reflectante de aluminio.
- **Velocidad Angular. RPM**: No es constante.
- **Vida útil**: Alrededor de 10 años (aunque en condiciones especiales de humedad y temperatura se calcula que pueden durar unos 217 años).
- **Tipos**:
  - De sólo lectura del inglés, CD-ROM (Compact Disc - Read Only Memory).
  - Grabable: del inglés, CD-R (Compact Disc - Recordable).
  - Regrabable: del inglés CD-RW (Compact Disc - ReWritable).

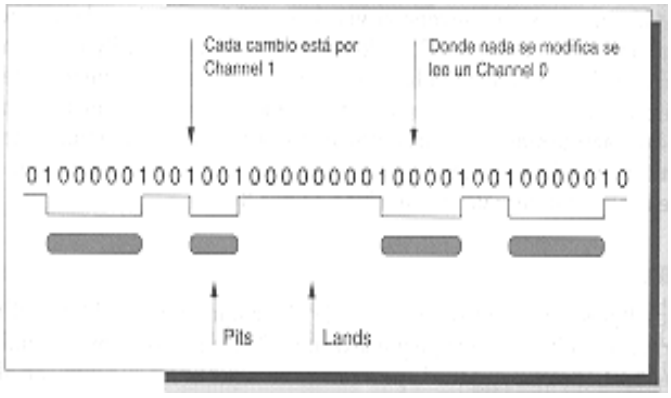


Almacenamiento de la información

En un CD la información se almacena en formato digital, es decir, utiliza un sistema binario para guardar los datos. Estos datos se graban en una única espiral que comienza desde el interior del disco (próximo al centro), y finaliza en la parte externa. Los datos binarios se almacenan en forma de hoyos y valles, de tal forma que al incidir el haz de luz del láser, el ángulo de reflexión es distinto en función de si se trata de un hoyo o un valle.

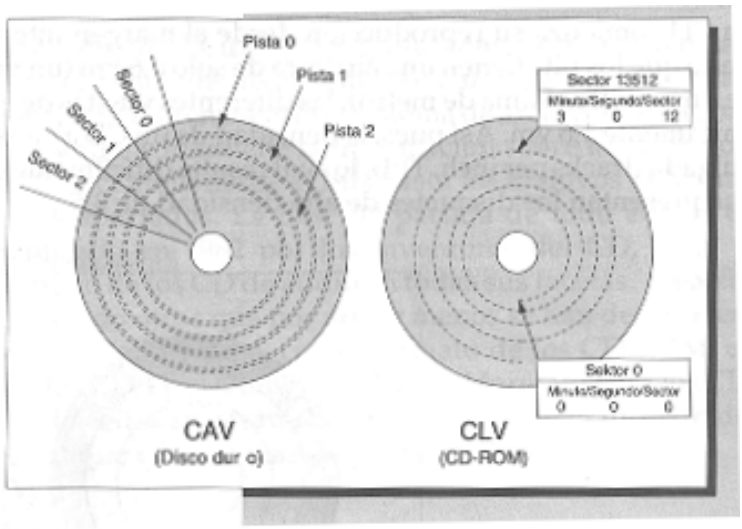


Es creencia muy común el pensar que un pozo corresponde a un valor binario y un llano al otro valor. Sin embargo, esto no es así, sino que los valores binarios son detectados por las *transiciones* de pozo a llano, y viceversa: una transición determina un 1 binario, mientras que la longitud de un pozo o un llano indica el número consecutivo de 0 binarios.



Además, los bits de información no son insertados "tal cual" en la pista del disco. En primer lugar, se utiliza una codificación conocida como modulación EFM (*Eigth to Fourteen Modulation*, o 'modulación ocho a catorce') cuya técnica consiste en igualar un bloque de ocho bits a uno de catorce, donde cada 1 binario debe estar separado (al menos) por dos 0 binarios.

El almacenamiento de la información se realiza mediante *tramas*. Cada trama supone un total de 588 bits, de los cuales 24 bits son de sincronización, 14 bits son de control, 536 bits son de datos y los últimos 14 bits son de corrección de errores. De los 536 bits de datos, hay que tener en cuenta que están codificados por modulación EFM, y que cada bloque de 14 bits está separado del siguiente por tres bits.



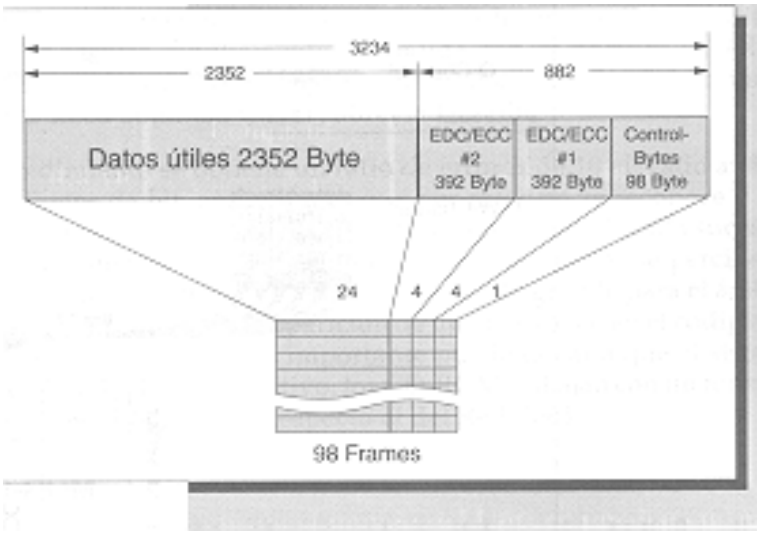
En un disco magnético existen muchas pistas, cada una con el mismo número de sectores; estas pistas son mayores cuanto más exteriores son, de lo que se deduce que los sectores son también mayores cuanto más exterior es la pista que los contiene. Esto quiere decir que en un disco magnético la densidad 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 KB de longitud que en un medio elegantemente dividido en pistas y sectores.

Trama/frame: Bloque de datos más pequeño de un cd. Cada una tiene un total de 588 bits, de los que 24 son de sincronización, 14 de control, 536 de datos y los últimos 14 bits son de corrección de errores. 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.

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.

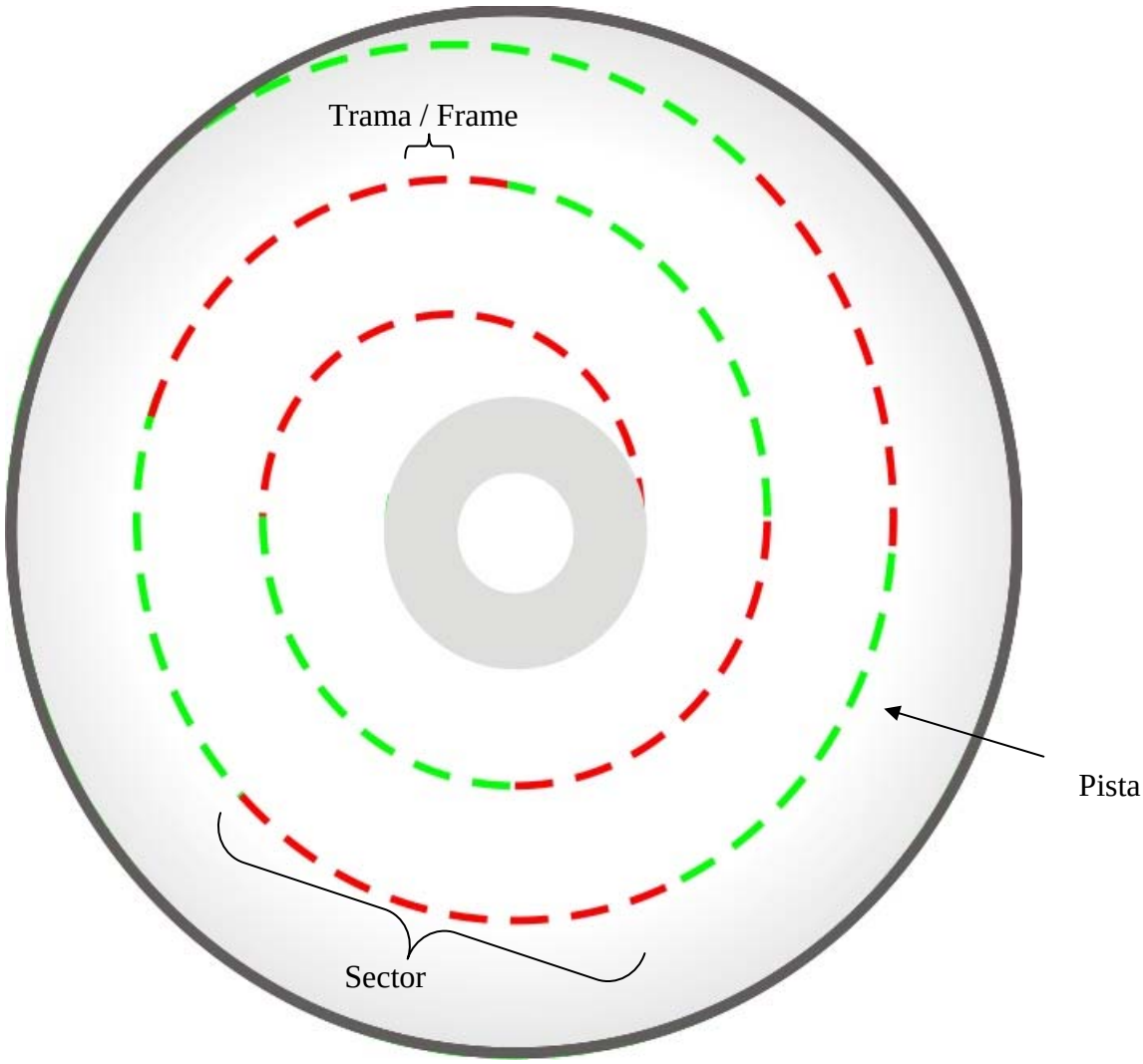


Pista: Existe sólo una. Es una espiral que va del centro del disco al exterior. Al no tener varias, como los discos magnéticos, hace que el tiempo de acceso a los datos sea menor en un CD.

Como todos los sectores tienen la misma longitud y tienen la misma densidad de grabación, para que la lectura se pueda hacer de forma correcta, el

disco debe girar más lento cuanto más al centro del CD se encuentre la cabeza lectora.

Esto se debe a que el disco se mueve con una velocidad angular (ω), cuando la cabeza lectora se encuentra cerca del borde del disco, dispone de más tiempo para leer un sector que cuando se encuentra más cerca del centro.



TOC

La tabla de contenido (TOC) es el índice central de cada cd. Contiene las posiciones iniciales de cada pista del cd. Por ejemplo:

Pista 1: Posición 02:03:30

Pista 2: Posición 07:34:34

Pista 3: Posición 11:23:46

.....

Fin del CD: Posición 14:45:24

Un reproductor de audio lee la TOC cuando se inserta el cd y así obtiene donde comienza cada una de las pistas y cual su tamaño. En los cds de datos el TOC contiene una referencia a las pistas de datos.

¿Por qué redondo? ¿Por qué un haz de luz?

La configuración en forma de disco le da a este soporte de datos versatilidad a la hora de acceder a cualquier parte de su superficie sin apenas movimientos del cabezal de lectura, usando solamente dos partes móviles, el cabezal que se mueve del centro al exterior del disco en línea recta y el eje de rotación que gira sobre sí mismo para trabajar conjuntamente con el cabezal y así obtener cualquier posición de la superficie con datos.

Este sistema de acceso a la información es superior a sistemas de cinta pues tiene menos calentamiento del soporte a altas velocidades, y el haz de luz no supone un problema de rozamiento (no toca el disco, sólo refleja luz).

El formato CD ha sido superado por algunos formatos posteriores que permiten mayor calidad como pueden ser:

- DVD:

El DVD (más conocido como "Digital Versatile Disc", aunque también se le puede denominar como "Digital Video Disc") es un formato multimedia de almacenamiento óptico que puede ser usado para guardar datos, incluyendo películas con alta calidad de vídeo y audio. Se asemeja a los discos compactos en cuanto a sus dimensiones físicas (diámetro de 12 u 8 cm), pero están codificados en un formato distinto y a una densidad mucho mayor. A diferencia de los CDs, todos los DVDs deben guardar los datos utilizando un sistema de archivos denominado UDF, el cual es una extensión del estándar ISO 9660, usado para CDs de datos.

MONTAJE DE UN CD-ROM

Durante el arranque de Linux se inicializa y arranca el driver del CD-ROM que tengamos, una vez realizado estas operaciones y siempre que se haya realizado de manera correcta, el gestor de auto montaje del SO (supermount antiguamente, en las últimas versiones se ha sustituido por udev debido a la inestabilidad generada por el antiguo superdemonio) podrá montar la unidad de cd-rom.

Posteriormente al montaje del cd-rom este se abre mediante las diferentes operaciones opens (realizandose comprobaciones sobre la unidad, se intenta comprobar la existencia de un disco en la unidad y en caso de que haya un CD o DVD en la unidad trata de averiguar su tipo (Dato, Música, SuperVideo,…)).

Tras realizar estas operaciones de comprobación, el driver trata de actualizar la tabla TOC, obtener la configuración de la unidad (Velocidad, estado del dispositivo, etc).

Por último antes y después de crear una nueva tabla TOC se obtienen y comprueban los distintos canales.

Una vez finalizados estos pasos el cd-rom esta listo para hacer un request y empezar a transferir y/o hacer consultas para la escritura.

NOTA: Hay que tener en cuenta que este modo de funcionamiento es genérico y los distintos driver añaden algunas peculiaridades para obtener el máximo rendimiento de sus dispositivos y también que la ejecución de los pasos comentados arriba depende del estado de los flags por lo que el orden de estos puede verse alterado.

IDE CDROM. Estructuras

Primeramente, mostramos las estructuras de datos relacionadas con el proceso de búsqueda dentro de un cd-rom.

Struct atapi_msf

Las unidades de CD-ROM especifican el tiempo bien en formato MSF (mins/segs/frames), o directamente en frames. Un frame es una unidad estándar de tiempo en el CD, que se corresponde con 1/75 segundos. Esta estructura determina las direcciones MSF en el cd-rom:

```
/* Structure of a MSF cdrom address. */
```

```
114struct atapi_msf {
115        byte reserved;
116        byte minute;
117        byte second;
118        byte frame;
119};
```

Struct atapi_toc_header

Esta estructura compone la información cabecera de la tabla de contenidos:

```
struct atapi_toc_header {
124        unsigned short toc_length; /* Tamaño de la TOC*/
125        byte first_track; /* Primera pista */
126        byte last_track;  /* Última pista */
127};
```

Struct atapi_toc_entry

Esta estructura compone la información cabecera de la tabla de contenidos:

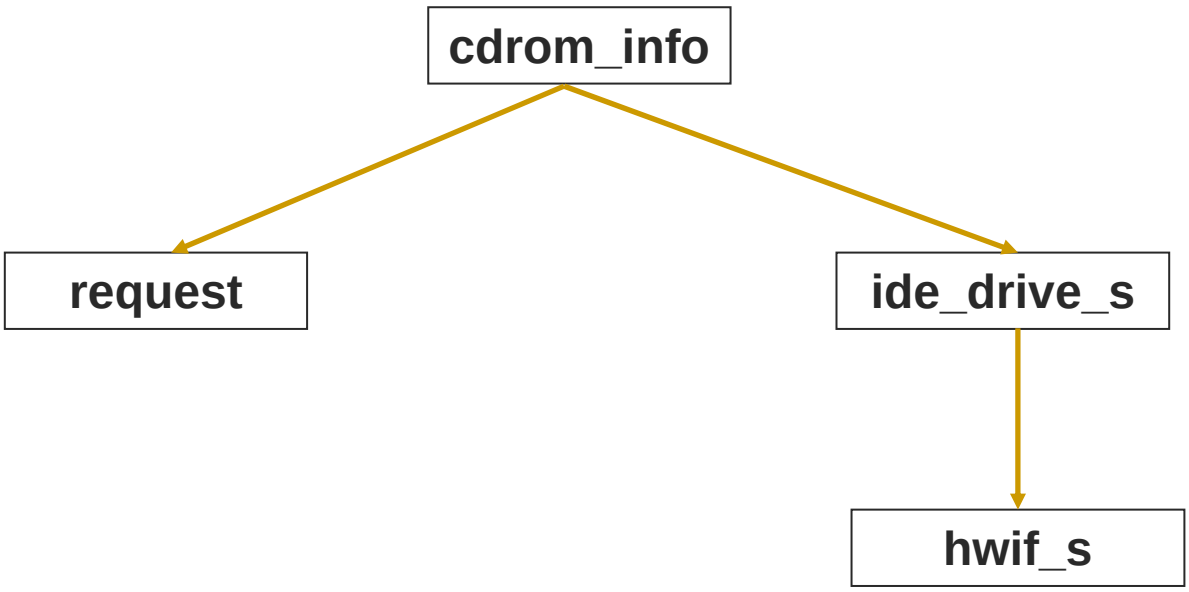
```
129struct atapi_toc_entry {
130        byte reserved1;
131#if defined(__BIG_ENDIAN_BITFIELD)
132        __u8 adr     : 4;
133        __u8 control : 4;
134#elif defined(__LITTLE_ENDIAN_BITFIELD)
135        __u8 control : 4;
136        __u8 adr     : 4;
137#else
138#error "Please fix <asm/byteorder.h>"
139#endif
140        byte track; /* Número de pista */
141        byte reserved2;
142        union {
143                unsigned lba;
144                struct atapi_msf msf;
145        } addr; /* Dirección de la pista en lba y en MSF*/
146};
```

Struct atapi_TOC

Esta estructura compone la información cabecera de la tabla de contenidos:

```
148 struct atapi_toc {
149        int    last_session_lba;
150        int    xa_flag;
151        unsigned long capacity;
152        struct atapi_toc_header hdr;
153        struct atapi_toc_entry  ent[MAX_TRACKS+1];
154          /* One extra for the leadout. */
155 };
```

A continuación, se muestran las estructuras **más destacadas** del driver IDE que se usan para la configuración y las operaciones de lectura y escritura en el cd-rom.



Struct cdrom_info

Esta estructura proporciona información sobre el dispositivo de cd-rom. Se encuentra definida en el fichero "/drivers/ide-cd.h".

```
110 /* Extra per-device info for cdrom drives. */
111 struct cdrom_info {
112         ide_drive_t     *drive; → Puntero a la estructura ide_drive_t
113         ide_driver_t    *driver;
114         struct gendisk  *disk;
115         struct kref     kref;
116
117         → Búfer para la tabla de contenidos (TOC).  NULL si no se ha
118            reservado memoria en la TOC para ese dispositivo.
119
120         struct atapi_toc *toc;
121
122         unsigned long   sector_buffered; → Primer sector en el búfer
123         unsigned long   nsectors_buffered; → Número de sectores
124         unsigned char   *buffer;
125
126         /* The result of the last successful request sense command
127            on this device. */
128         struct request_sense sense_data;
129
130         struct request request_sense_request; → Puntero a la
                                                    estructura request
131         int dma;
132         unsigned long last_block;
133         unsigned long start_seek;
```

```
134
135         unsigned int cd_flags;
136
137         u8 max_speed;           /* Max speed of the drive. */
138         u8 current_speed;       /* Current speed of the drive. */
139
140         /* Per-device info needed by cdrom.c generic driver. */
141         struct cdrom_device_info devinfo;
142
143     unsigned long write_timeout;
144 };
```

Struct ide_drive_s

Estructura que guarda los atributos principales de un dispositivo IDE. Se renombra como **ide_drive_t** y aparece con este nombre en otras partes del código. Con estructuras de este tipo, se forma una lista circular, en la que hay un nodo para cada uno de los dispositivos IDE.

Está definida en "/include/linux/ide.h"

```
544 typedef struct ide_drive_s {
545         char name[4];     → nombre del dispositivo
546         char driver_req[10]; /* requests specific driver */
547
548         request_queue_t *queue; → cola de peticiones
549
550         struct request  *rq; → petición actual
551         struct ide_drive_s *next; → lista circular
552         void *driver_data;   /* extra driver data */
553         struct hd_driveid *id;→ información del modelo del
dispositivo

554         struct proc_dir_entry *proc;    /* /proc/ide/ directory entry
*/
555         struct ide_settings_s *settings;/* /proc/ide/ drive settings */
556         char            devfs_name[64]; /* devfs crap */
557
558         struct hwif_s *hwif;  → puntero a estructura hwif_s
559
560         unsigned long sleep; → valor que indica el tiempo que tiene
que estar el dispositivo inactivo
561         unsigned long service_start; → momento en el que se comenzó
la última petición
562         unsigned long service_time;     /* service time of last request
*/
563         unsigned long timeout; → tiempo máximo que se espera por una
interrupción del dispositivo
564
565         special_t special;        /* special action flags */
566         select_t select;         /* basic drive/head select reg value
*/
567
568         u8      keep_settings; → reestablece la configuración tras un
reinicio
569         u8      autodma; → el dispositivo puede usar dma

570         u8      using_dma; → el disco usa dma para lectura/escritura
```

```
571         u8      retry_pio;              /* retrying dma capable host in
pio */
572         u8      state;                  /* retry state */
573         u8      waiting_for_dma; → dma en proceso
574         u8      unmask;                 /* okay to unmask other irqs */
575         u8      bswap;                  /* byte swap data */
576         u8      dsc_overlap;            /* DSC overlap */
577         u8      nice1;                  /* give potential excess
bandwidth */
578
579         unsigned present        : 1;    /* drive is physically present
*/
580         unsigned dead           : 1;    /* device ejected hint */
581         unsigned id_read        : 1;    /* 1=id read from disk 0 =
synthetic */
582         unsigned noprobe        : 1;    /* from:  hdx=noprobe */
583         unsigned removable      : 1;    /* 1 if need to do
check_media_change */
584         unsigned attach         : 1;    /* needed for removable devices
*/
585         unsigned forced_geom    : 1;    /* 1 if hdx=c,h,s was given at
boot */
586         unsigned no_unmask      : 1;    /* disallow setting unmask bit
*/
587         unsigned no_io_32bit    : 1;    /* disallow enabling 32bit I/O
*/
588         unsigned atapi_overlap  : 1;    /* ATAPI overlap (not
supported) */
589         unsigned nice0          : 1;    /* give obvious excess
bandwidth */
590         unsigned nice2          : 1;    /* give a share in our own
bandwidth */
591         unsigned doorlocking    : 1;    /* for removable only: door
lock/unlock works */
592         unsigned autotune       : 2;    /* 0=default, 1=autotune,
2=noautotune */
593         unsigned remap_0_to_1   : 1;    /* 0=noremap, 1=remap 0->1 (for
EZDrive) */
594         unsigned blocked        : 1;    /* 1=powermanagment told us not
to do anything, so sleep nicely */
595         unsigned vdma           : 1;    /* 1=doing PIO over DMA 0=doing
normal DMA */
596         unsigned addressing;            /*      : 3;
597                                          *  0=28-bit
598                                          *  1=48-bit
599                                          *  2=48-bit doing 28-bit
600                                          *  3=64-bit
601                                          */
602         unsigned scsi           : 1;    /* 0=default, 1=ide-scsi
emulation */
603         unsigned sleeping       : 1;    /* 1=sleeping & sleep field
valid */
604         unsigned post_reset     : 1;
605
606         u8      quirk_list;     /* considered quirky, set for a
specific host */
607         u8      init_speed; → velocidad de transferencia inicial
608         u8      pio_speed;      /* unused by core, used by some drivers
for fallback from DMA */
609         u8      current_speed;  /* current transfer rate set */
610         u8      dn;             /* now wide spread use */
```

```
611         u8      wcache;         /* status of write cache */
612         u8      acoustic;       /* acoustic management */
613         u8      media;          /* disk, cdrom, tape, floppy, ... */
614         u8      ctl;            /* "normal" value for IDE_CONTROL_REG
*/
615         u8      ready_stat;     /* min status value for drive ready */
616         u8      mult_count;     /* current multiple sector setting */
617         u8      mult_req;       /* requested multiple sector setting */
618         u8      tune_req;       /* requested drive tuning setting */
619         u8      io_32bit;       /* 0=16-bit, 1=32-bit, 2/3=32bit+sync
*/
620         u8      bad_wstat;      /* used for ignoring WRERR_STAT */
621         u8      nowerr;         /* used for ignoring WRERR_STAT */
622         u8      sect0;          /* offset of first sector for DM6:DDO
*/
623         u8      head;   → número de cabezas
624         u8      sect;   → número de sectores por pista
625         u8      bios_head;      /* BIOS/fdisk/LILO number of heads */
626         u8      bios_sect;      /* BIOS/fdisk/LILO sectors per track */
627
628         unsigned int    bios_cyl;       /* BIOS/fdisk/LILO number of
cyls */
629         unsigned int    cyl;            /* "real" number of cyls */
630         unsigned int    drive_data;     /* use by tuneproc/selectproc
*/
631         unsigned int    usage;          /* current "open()" count for
drive */
632         unsigned int    failures;       /* current failure count */
633         unsigned int    max_failures;   /* maximum allowed failure
count */
634
635         u64             capacity64;   → número de sectores total
636
637         int             lun;            /* logical unit */
638         int             crc_count;      /* crc counter to reduce drive
speed */
639         struct list_head list;
640         struct device   gendev;
641     struct completion gendev_rel_comp;     /* to deal with device release() */
642 } ide_drive_t;
```

Struct hwif_s

Estructura de la que dispone todo dispositivo IDE. En relación con el cd-rom, es importante porque tiene funciones abstractas como las de transferencia de datos entre otras.

Se renombra como "ide_hwif_t", y en algunas partes del código aparece con ese nombre.

Al igual que con ide_drive_s, se forma una lista circular con estructuras de este tipo, en la que cada nodo hace referencia a un dispositivo IDE determinado.

Se encuentra definida en "/include/linux/ide.h".

```
652 typedef struct hwif_s {
653         struct hwif_s *next; → puntero al siguiente de la lista
circular
```

```
654         struct hwif_s *mate;            /* other hwif from same PCI
chip */
655         struct hwgroup_s *hwgroup;      /* actually (ide_hwgroup_t *)
*/
656         struct proc_dir_entry *proc;    /* /proc/ide/ directory entry
*/
657
658         char name[6];  → nombre de la interfaz
659
660                 /* task file registers for pata and sata */
661         unsigned long   io_ports[IDE_NR_PORTS];
662         unsigned long   sata_scr[SATA_NR_PORTS];
663         unsigned long   sata_misc[SATA_NR_PORTS];
664
665         hw_regs_t       hw;             /* Hardware info */
666         ide_drive_t     drives[MAX_DRIVES];     /* drive info */
667
668         u8 major;       /* our major number */
669         u8 index; → índice que indica el dispositivo
670         u8 channel;     /* for dual-port chips: 0=primary, 1=secondary
*/
671         u8 straight8;   /* Alan's straight 8 check */
672         u8 bus_state;   /* power state of the IDE bus */
673
674         u8 atapi_dma;   /* host supports atapi_dma */
675         u8 ultra_mask;
676         u8 mwdma_mask;
677         u8 swdma_mask;
678
679         hwif_chipset_t chipset; /* sub-module for tuning.. */
680
681         struct pci_dev  *pci_dev;       /* for pci chipsets */
682         struct ide_pci_device_s *cds;   /* chipset device struct */
683
684         void (*rw_disk)(ide_drive_t *, struct request *);
685
686 #if 0
687         ide_hwif_ops_t  *hwifops;
688 #else
689         /* routine to tune PIO mode for drives */
690         void    (*tuneproc)(ide_drive_t *, u8);
691         /* routine to retune DMA modes for drives */
692         int     (*speedproc)(ide_drive_t *, u8);
693         /* tweaks hardware to select drive */
694         void    (*selectproc)(ide_drive_t *);
695         /* chipset polling based on hba specifics */
696         int     (*reset_poll)(ide_drive_t *);
697         /* chipset specific changes to default for device-hba resets */
698         void    (*pre_reset)(ide_drive_t *);
699         /* routine to reset controller after a disk reset */
700         void    (*resetproc)(ide_drive_t *);
701         /* special interrupt handling for shared pci interrupts */
702         void    (*intrproc)(ide_drive_t *);
703         /* special host masking for drive selection */
704         void    (*maskproc)(ide_drive_t *, int);
705         /* check host's drive quirk list */
706         int     (*quirkproc)(ide_drive_t *);
707         /* driver soft-power interface */
708         int     (*busproc)(ide_drive_t *, int);
709 //      /* host rate limiter */
710 //      u8      (*ratemask)(ide_drive_t *);
```

```
711 //      /* device rate limiter */
712 //      u8      (*ratefilter)(ide_drive_t *, u8);
713 #endif
714
715         void (*ata_input_data)(ide_drive_t *, void *, u32);
716         void (*ata_output_data)(ide_drive_t *, void *, u32);
717
718         void (*atapi_input_bytes)(ide_drive_t *, void *, u32); → función para leer  datos. La implementará cada dispositivo
719         void (*atapi_output_bytes)(ide_drive_t *, void *, u32); → función para escribir  datos. La implementará cada dispositivo
720
721         int (*dma_setup)(ide_drive_t *); → configuración del modo dma
722         void (*dma_exec_cmd)(ide_drive_t *, u8);
723         void (*dma_start)(ide_drive_t *); → empieza el modo dma
724         int (*ide_dma_end)(ide_drive_t *drive); ); →  termina el modo dma

725         int (*ide_dma_check)(ide_drive_t *drive);
726         int (*ide_dma_on)(ide_drive_t *drive);
727         int (*ide_dma_off_quietly)(ide_drive_t *drive);
728         int (*ide_dma_test_irq)(ide_drive_t *drive);
729         int (*ide_dma_host_on)(ide_drive_t *drive);
730         int (*ide_dma_host_off)(ide_drive_t *drive);
731         int (*ide_dma_lostirq)(ide_drive_t *drive);
732         int (*ide_dma_timeout)(ide_drive_t *drive);
733
734         void (*OUTB)(u8 addr, unsigned long port);
735         void (*OUTBSYNC)(ide_drive_t *drive, u8 addr, unsigned long
port);
736         void (*OUTW)(u16 addr, unsigned long port);
737         void (*OUTL)(u32 addr, unsigned long port);
738         void (*OUTSW)(unsigned long port, void *addr, u32 count);
739         void (*OUTSL)(unsigned long port, void *addr, u32 count);
740
741         u8  (*INB)(unsigned long port);
742         u16 (*INW)(unsigned long port);
743         u32 (*INL)(unsigned long port);
744         void (*INSW)(unsigned long port, void *addr, u32 count);
745         void (*INSL)(unsigned long port, void *addr, u32 count);
746
747         /* dma physical region descriptor table (cpu view) */
748         unsigned int    *dmatable_cpu;
749         /* dma physical region descriptor table (dma view) */
750         dma_addr_t      dmatable_dma;
751         /* Scatter-gather list used to build the above */
752         struct scatterlist *sg_table;
753         int sg_max_nents;               /* Maximum number of entries in
it */
754         int sg_nents;                   /* Current number of entries in
it */
755         int sg_dma_direction;           /* dma transfer direction */
756
757         /* data phase of the active command (currently only valid for
PIO/DMA) */
758         int             data_phase;
759
760         unsigned int nsect;
761         unsigned int nleft;
762         unsigned int cursg;
763         unsigned int cursg_ofs;
```

```
764
765         int             mmio;           /* hosts iomio (0) or custom
(2) select */
766         int             rqsize;         /* max sectors per request */
767         int             irq;            /* our irq number */
768
769         unsigned long   dma_master;     /* reference base addr dmabase
*/
770         unsigned long   dma_base;       /* base addr for dma ports */
771         unsigned long   dma_command;    /* dma command register */
772         unsigned long   dma_vendor1;    /* dma vendor 1 register */
773         unsigned long   dma_status;     /* dma status register */
774         unsigned long   dma_vendor3;    /* dma vendor 3 register */
775         unsigned long   dma_prdtable;   /* actual prd table address */
776         unsigned long   dma_base2;      /* extended base addr for dma
ports */
777
778         unsigned        dma_extra;      /* extra addr for dma ports */
779         unsigned long   config_data;    /* for use by chipset-specific
code */
780         unsigned long   select_data;    /* for use by chipset-specific
code */
781
782         unsigned        noprobe    : 1; /* don't probe for this
interface */
783         unsigned        present    : 1; /* this interface exists */
784         unsigned        hold       : 1; /* this interface is always
present */
785         unsigned        serialized : 1; /* serialized all channel
operation */
786         unsigned        sharing_irq: 1; /* 1 = sharing irq with another
hwif */
787         unsigned        reset      : 1; /* reset after probe */
788         unsigned        autodma    : 1; /* auto-attempt using DMA at
boot */
789         unsigned        udma_four  : 1; /* 1=ATA-66 capable, 0=default
*/
790         unsigned        no_lba48   : 1; /* 1 = cannot do LBA48 */
791         unsigned        no_lba48_dma : 1; /* 1 = cannot do LBA48 DMA */
792         unsigned        no_dsc     : 1; /* 0 default, 1 dsc_overlap
disabled */
793         unsigned        auto_poll  : 1; /* supports nop auto-poll */
794         unsigned        sg_mapped  : 1; /* sg_table and sg_nents are
ready */
795         unsigned        no_io_32bit : 1; /* 1 = can not do 32-bit IO
ops */
796
797         struct device   gendev;
798         struct completion gendev_rel_comp; /* To deal with device
release() */
799
800         void            *hwif_data;     /* extra hwif data */
801
802         unsigned dma;
803
804     void (*led_act)(void *data, int rw);
805 } ____cacheline_internodealigned_in_smp ide_hwif_t;
```

Struct request

Estructura que almacena información sobre la petición que se ha ha hecho. Se encuentra definida en "/include/linux/blkdev.h".

```
144 struct request {
145         struct list_head queuelist;
146         struct list_head donelist;
147
148         struct request_queue *q; → cola de peticiones

149
150         unsigned int cmd_flags;
151         enum rq_cmd_type_bits cmd_type; → indica para qué tipo de
dispositivo se hace la  peticición (ejemplo: para IDE ATA/ATAPI)
152
153         /* Maintain bio traversal state for part by part I/O
submission.
154          * hard_* are block layer internals, no driver should touch
them!
155          */
156
157         sector_t sector; → siguiente sector a leer/escribir
158         sector_t hard_sector;           /* next sector to complete */
159         unsigned long nr_sectors → número de sectores restantes
160         unsigned long hard_nr_sectors;  /* no. of sectors left to
complete */
161         /* no. of sectors left to submit in the current segment */
162         unsigned int current_nr_sectors; → número de sectores que
quedan por completar en el segmento actual
163
164         /* no. of sectors left to complete in the current segment */
165         unsigned int hard_cur_sectors;
166
167         struct bio *bio;
168         struct bio *biotail;
169
170         struct hlist_node hash; /* merge hash */
171         /*
172          * The rb_node is only used inside the io scheduler, requests
173          * are pruned when moved to the dispatch queue. So let the
174          * completion_data share space with the rb_node.
175          */
176         union {
177                 struct rb_node rb_node; /* sort/lookup */
178                 void *completion_data;
179         };
180
181         /*
182          * two pointers are available for the IO schedulers, if they
need
183          * more they have to dynamically allocate it.
184          */
185         void *elevator_private;
186         void *elevator_private2;
187
188         struct gendisk *rq_disk; → información específica del disco

189         unsigned long start_time;
190
191         /* Number of scatter-gather DMA addr+len pairs after
192          * physical address coalescing is performed.
```

```
193          */
194         unsigned short nr_phys_segments;
195
196         /* Number of scatter-gather addr+len pairs after
197          * physical and DMA remapping hardware coalescing is performed.
198          * This is the number of scatter-gather entries the driver
199          * will actually have to deal with after DMA mapping is done.
200          */
201         unsigned short nr_hw_segments;
202
203         unsigned short ioprio;
204
205         void *special;
206         char *buffer; → búfer en el que se guardará la petición
207
208         int tag;
209         int errors; →  errores

210
211         int ref_count; →  indica las veces que ha sido referenciada
212
213         /*
214          * when request is used as a packet command carrier
215          */
216         unsigned int cmd_len;
217         unsigned char cmd[BLK_MAX_CDB];
218
219         unsigned int data_len;
220         unsigned int extra_len; /* length of alignment and padding */
221         unsigned int sense_len;
222         void *data;
223         void *sense;
224
225         unsigned int timeout; →  tiempo máximo de espera por una
petición
226         int retries;
227
228         /*
229          * completion callback.
230          */
231         rq_end_io_fn *end_io;
232         void *end_io_data;
233
234         /* for bidi */
235           struct request *next_rq;
236 };
```

IDE_CDROM. Funciones

En este apartado comentaremos las funciones más importantes de los drivers IDE_CDROM genéricos. Se encuentran en **/drivers/ide/id-cd.c**. Hay que comentar que existen otros drivers más específicos en la carpeta **/driver/cdrom**.

Todas las funciones son llamadas desde el archivo **ide-cd_ioctl.c**, en el que se encuentran las funciones asociadas a los comandos de la unidad de cd, como expulsar la bandeja del cd-rom, bloquearla para que no se pueda abrir, para ver el estado, resetearlo, etc.

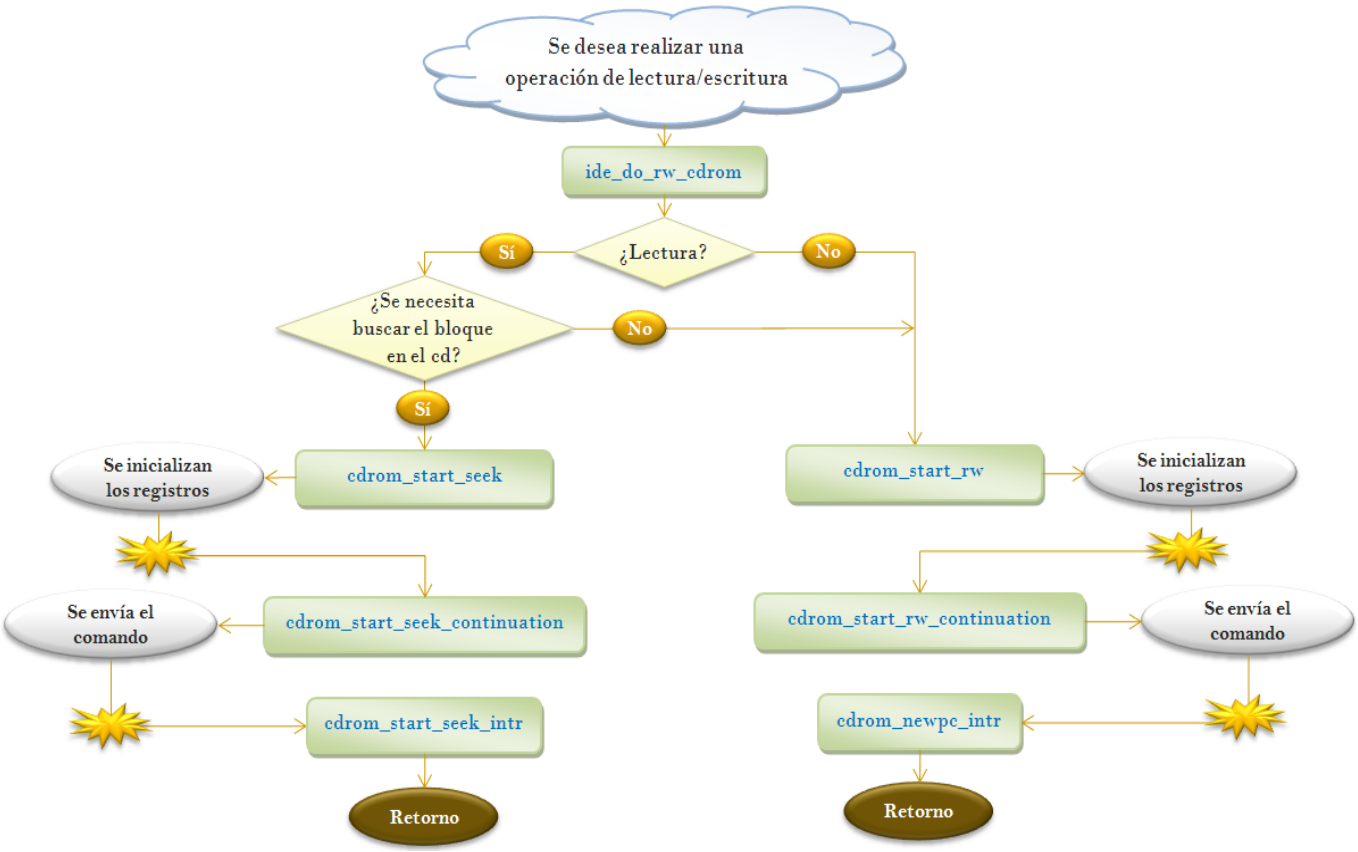
No obstante nos centraremos en las funciones del archivo id-cd.c.

Tener en cuenta que a todas se les pasa (desde las funciones de **ide-cd_ioctl.c**) la estructura genérica **`ide_drive_t *drive`** con la información del dispositivo.

Por último, aclarar que algunas funciones devuelven un dato del tipo **ide_startstop_t**, que simplemente es un "enum" (definido en **include/linux/ide.h**) con dos valores:

```
332typedef enum {
333         ide_stopped,     → Indica que no ha comenzado ninguna operación
334         ide_started,     → Indica que ya se ha comenzado una operación
335} ide_startstop_t;
```

Dibujo aclaratorio



Inicializaciones

- **Inicialización de una petición**

→ Inicializa la petición de un comando. Por ejemplo, cuando se pulsa expulsar disco en la unidad de CD-ROM se llama a una función de "ide-cd_ioctl.c" (cdrom_eject) que llamará a esta función con el tipo de petición requerida en el parámetro "rq".

```
200void ide_cd_init_rq(ide_drive_t *drive, struct request *rq)
201{
            → Obtenemos información extra del dispositivo (recordar que era un
            puntero tipo void)
202         struct cdrom_info *cd = drive->driver_data;

            → Inicializa la estructura rq (la coloca toda a cero)
204         ide_init_drive_cmd(rq);

            → Establecemos el tipo de petición (en este caso se indica que es
            para un dispositivo IDE ATA/ATAPI)
205         rq->cmd_type = REQ_TYPE_ATA_PC;

            → Guardamos en la petición información específica del CD-ROM (rq_disk
            también se usa para discos duros u otros dispositivos)
206         rq->rq_disk = cd->disk;
207}
```

→ Llamada por la función anterior. Establece a cero la memoria correspondiente a la estructura de la petición e inicializa las veces que ha sido referenciada.

```
1594void ide_init_drive_cmd (struct request *rq)
1595{
1596         memset(rq, 0, sizeof(*rq));
1597         rq->ref_count = 1;
1598}
```

- **Preparación del dispositivo para recibir una orden**

→ Inicializa los registros del dispositivo para poder transferirle el comando y comienza dicha transferencia a través de la rutina "handler". Si el dispositivo admite interrupciones, se llamará a "handler" cuando ocurra la interrupción correspondiente (pulsar "expulsar", termina una lectura, etc.), si no, se le llama en el acto.

```
512static ide_startstop_t cdrom_start_packet_command(ide_drive_t *drive,
513                                                   int xferlen,
514                                                   ide_handler_t *handler)
515{
516         ide_startstop_t startstop;
            → Se obtiene información específica del dispositivo
517         struct cdrom_info *info = drive->driver_data;
            → "hwif" es una estructura que dispone todo dispositivo IDE
            (abstracta, con las funciones de transferencia de datos entre otras
            cosas)
518         ide_hwif_t *hwif = drive->hwif;
519
```

```
520         → Espera a que el dispositivo esté disponible (preguntando
            continuamente). Si la función ide_wait_stat devuelve error (por
            ejemplo, se pasó el timeout), pues se retorna
521         if (ide_wait_stat(&startstop, drive, 0, BUSY_STAT, WAIT_READY))
522                 return startstop;
523
524         → Se averigua si el dispositivo admite DMA.
525         if (info->dma)
526                 info->dma = !hwif->dma_setup(drive);
527
528         → Aquí es donde se inicializan los registros.

529         ide_pktcmd_tf_load(drive, IDE_TFLAG_OUT_NSECT | IDE_TFLAG_OUT_LBAL |
530                            IDE_TFLAG_NO_SELECT_MASK, xferlen, info->dma);
531
            → Si admitía interrupciones, se espera a que se produzca.
532         if (info->cd_flags & IDE_CD_FLAG_DRQ_INTERRUPT) {
533
534                 if (info->dma)
535                         drive->waiting_for_dma = 0;
536
537                 → Se envía el comando llamando a "handler" a través de esta
                  función (que espera por la interrupción)
538                 ide_execute_command(drive, WIN_PACKETCMD, handler,
                                    ATAPI_WAIT_PC, cdrom_timer_expiry);
539                 return ide_started;
540         } else {
541                 unsigned long flags;
542
543                 → Comienza el envío sin esperar por la interrupción (de
                  forma atómica)
544                 spin_lock_irqsave(&ide_lock, flags);
545                 hwif->OUTBSYNC(drive, WIN_PACKETCMD, IDE_COMMAND_REG);

                    → El dispositivo tardará 400 ns en responder
546                 ndelay(400);
547                 spin_unlock_irqrestore(&ide_lock, flags);
549                 return (*handler) (drive);
550         }
551}
```

→ Envía el comando al dispositivo. Antes se debe haber llamado a la función anterior para que se inicialicen los registros. Ahora "handler" será la función que se llamará cuando el comando se termine de enviar.

```
559static ide_startstop_t cdrom_transfer_packet_command (ide_drive_t *drive,
560                                                    struct request *rq,
561                                                    ide_handler_t *handler)
562{
563         ide_hwif_t *hwif = drive->hwif;
564         int cmd_len;
565         struct cdrom_info *info = drive->driver_data;
566         ide_startstop_t startstop;
567         → Comprueba si se admiten interruciones
568         if (info->cd_flags & IDE_CD_FLAG_DRQ_INTERRUPT) {
                  → En caso afirmativo, hemos llegado aquí porque se ha
                  producido una, por lo que simplemente se verifica si hubo
                  errores y continuamos
573                 if (cdrom_decode_status(drive, DRQ_STAT, NULL))
574                         return ide_stopped;
```

```
575
577                 if (info->dma)
578                         drive->waiting_for_dma = 1;
579         } else {
580                 → No se admitían interrupciones, se espera a que el
                    dispositivo esté preparado preguntando continuamente
                    (pooling)
581                 if (ide_wait_stat(&startstop, drive, DRQ_STAT,
582                                   BUSY_STAT, WAIT_READY))
583                         return startstop;
584         }
585
586         → Carga la función (handler) que se llamará cuando el comando se
          haya enviado y el dispositivo esté preparado
587         ide_set_handler(drive, handler, rq->timeout, cdrom_timer_expiry);
588
590         cmd_len = COMMAND_SIZE(rq->cmd[0]);
591         if (cmd_len < ATAPI_MIN_CDB_BYTES)
592                 cmd_len = ATAPI_MIN_CDB_BYTES;
593
594         → Envía el comando al dispositivo
595         HWIF(drive)->atapi_output_bytes(drive, rq->cmd, cmd_len);
596
586         → Si se admite DMA, se activa
598         if (info->dma)
599                 hwif->dma_start(drive);
600
601         return ide_started;
602}
```

Lectura/escritura

- **Funciones auxiliares**

→ Guarda en un buffer (local, no uno del dispositivo) los sectores a transmitir desde el dispositivo. Una vez guardado el primer sector, se asume que el resto de sectores son continuos.

```
634static void cdrom_buffer_sectors (ide_drive_t *drive, unsigned long sector,
635                                   int sectors_to_transfer)
636{
          → Se obtinene información del driver (estructura que lo representa)
637         struct cdrom_info *info = drive->driver_data;
638
639         → Se calcula el número de sectores a transferir como el mínimo
          entre el nºpasado por parámetro y el tamaño del buffer menos los
          sectores que aún se encuentran en el buffer (por ejemplo de
          anteriores transferencias)
640         int sectors_to_buffer = min_t(int, sectors_to_transfer,
641                                       (SECTOR_BUFFER_SIZE >> SECTOR_BITS) -
642                                         info->nsectors_buffered);
643
644         char *dest;
645
646         → Se comprueba que realmente haya un buffer. Si no, no se
          transfiere nada
647         if (info->buffer == NULL)
648                 sectors_to_buffer = 0;
649
```

```
650         → Si es el primer sector del buffer, se almacena su número (el
            resto son consecutivos)
651         if (info->nsectors_buffered == 0)
652                 info->sector_buffered = sector;
653
654         → Se calcula el destino dentro del buffer y se transfieren los
            datos
655         dest = info->buffer + info->nsectors_buffered * SECTOR_SIZE;
656         while (sectors_to_buffer > 0) {
657                 HWIF(drive)->atapi_input_bytes(drive, dest, SECTOR_SIZE);
658                 --sectors_to_buffer;
659                 --sectors_to_transfer;
660                 ++info->nsectors_buffered;
661                 dest += SECTOR_SIZE;
662         }
663
664         → Deschamos los datos que no quepan en el buffer
665         ide_cd_drain_data(drive, sectors_to_transfer);
666}
```

→ Ante una petición de lectura, intenta leer los datos desde el buffer anterior. Devuelve 0 si no consigue completar toda la petición, distinto de 0 en el caso contrario

```
740static int cdrom_read_from_buffer (ide_drive_t *drive)
741{
742         struct cdrom_info *info = drive->driver_data;
743         struct request *rq = HWGROUP(drive)->rq;
744         unsigned short sectors_per_frame;
745
746         sectors_per_frame = queue_hardsect_size(drive->queue) >> SECTOR_BITS;
747
748         → Si no hay buffer no se puede leer
749         if (info->buffer == NULL) return 0;
750
751         → Leemos datos (los copiamos desde el buffer local del dispositivo
            al buffer de la petición) hasta completar la petición y mientras el
            siguiente sector se encuentre en el buffer, es decir, el nº del
            primer sector de la  petición (rq->sector) está entre el primero
            almacenado en el buffer (info->sector_buffered) y el último
            (info->sector_buffered + info->nsectors_buffered)

753         while (rq->nr_sectors > 0 &&
754                rq->sector >= info->sector_buffered &&
755                rq->sector < info->sector_buffered + info->nsectors_buffered)
{
756                 if (rq->current_nr_sectors == 0)
757                         cdrom_end_request(drive, 1);
758
                    → Realizamos el volcado de memoria
759                 memcpy (rq->buffer,
760                         info->buffer +
761                         (rq->sector - info->sector_buffered) * SECTOR_SIZE,
762                         SECTOR_SIZE);

                    → Actualizamos el puntero del buffer de la petición y los
                    contadores correspondientes
763                 rq->buffer += SECTOR_SIZE;
764                 --rq->current_nr_sectors;
765                 --rq->nr_sectors;
```

```
766                 ++rq->sector;
767         }
769         → Si todos los datos se pudieron leer del buffer, se retorna
            satisfactoriamente
771         if (rq->nr_sectors == 0) {
772                 cdrom_end_request(drive, 1);
773                 return -1;
774         }
776
777         if (rq->current_nr_sectors == 0)
778                 cdrom_end_request(drive, 1);
779
784         if (rq->current_nr_sectors < bio_cur_sectors(rq->bio) &&
785             (rq->sector & (sectors_per_frame - 1))) {
786                 printk(KERN_ERR
                               "%s: cdrom_read_from_buffer: buffer botch (%ld)\n",
787                                drive->name, (long)rq->sector);
788                 cdrom_end_request(drive, 0);
789                 return -1;
790         }
            → No se pudieron leer todos los datos del buffer
792         return 0;
793}
```

- **Funciones secundarias**

Dentro del grupo de funciones de búsqueda y del de control de lectura/escritura, las rutinas se llaman entre sí indirectamente, por medio de las funciones **cdrom_start_packet_command** y **cdrom_transfer_packet_command** (vistas anteriormente). Recordar que éstas llamaban a la función que se les pasaba por parámetro cuando captaban una interrupción del dispositivo.

Pues cada rutina de las que veremos a continuación colocará en dicho parámetro la función que desea que se ejecute cuando se reciba la siguiente interrupción.

- Funciones de búsqueda

```
→ Comienza el envío del comando de petición de búsqueda de un bloque al
dispositvo.
```

```
890static ide_startstop_t cdrom_start_seek (ide_drive_t *drive, unsigned int block)
891{
892         struct cdrom_info *info = drive->driver_data;
893
894         info->dma = 0;
895         info->start_seek = jiffies;

            → Comienza el envío del comando de búsqueda. Se inicializan los
            registros por medio de la siguiente función, que esperará a que el
            dispositivo esté preparado. Luego llamará a la rutina que se le pasa
            como tercer parámetro
896         return cdrom_start_packet_command(drive, 0,
                                              cdrom_start_seek_continuation);
897}
```

→ Continúa el envío del comando de búsqueda. Se llama indirectamente desde la función anterior

```
875static ide_startstop_t cdrom_start_seek_continuation (ide_drive_t *drive)
876{
            → Se obtiene la estructura request por defecto
877         struct request *rq = HWGROUP(drive)->rq;
878         sector_t frame = rq->sector;
879
880         sector_div(frame, queue_hardsect_size(drive->queue) >> SECTOR_BITS);
881
882         memset(rq->cmd, 0, sizeof(rq->cmd));

            → Se especifica que el comando será de búsqueda
883         rq->cmd[0] = GPCMD_SEEK;
884         put_unaligned(cpu_to_be32(frame), (unsigned int *) &rq->cmd[2]);885
886         rq->timeout = ATAPI_WAIT_PC;

            → Se envía por fin la petición con el comando al dispositivo.
            A la función del tercer parámetro se llamará cuando el comando se
            termine, en este caso cuando ya se ha realizado la búsqueda.
887         return cdrom_transfer_packet_command(drive, rq, &cdrom_seek_intr);
888}
```

→ Función que se llama (indirectamente desde la anterior) cuando ya se ha realizado la búsqueda

```
851static ide_startstop_t cdrom_seek_intr (ide_drive_t *drive)
852{
853         struct cdrom_info *info = drive->driver_data;
854         int stat;
855         static int retry = 10;

            → Se  verifica se hubo errores.
857         if (cdrom_decode_status(drive, 0, &stat))
858                 return ide_stopped;
859
            → Se indica que se ha buscado el bloque.
860         info->cd_flags |= IDE_CD_FLAG_SEEKING;
861
862         if (retry && time_after(jiffies,
                                    info->start_seek + IDECD_SEEK_TIMER)) {
863                 if (--retry == 0) {
869                         drive->dsc_overlap = 0;
870                 }
871         }

            → Ahora el dispositivo se encuentra parado.
872         return ide_stopped;
873}
```

- Funciones de control de lectura/escritura

→ Función que comienza el envío del comando de lectura/escritura al dispositivo (realmente sólo inicializa los registros). Si se trata de una lectura, se intenta satisfacer desde el buffer local.

```
1223 static ide_startstop_t cdrom_start_rw(ide_drive_t *drive,
                                          struct request *rq)
1224 {
1225         struct cdrom_info *cd = drive->driver_data;

             → Se averigua se el comando es de escritura
1226         int write = rq_data_dir(rq) == WRITE;

1227         unsigned short sectors_per_frame =
1228                 queue_hardsect_size(drive->queue) >> SECTOR_BITS;
1229
             → Si es escritura y el disco está protegido contra ella, se termina la
             petición y se retorna "dispositivo parado"
1230         if (write) {

1234                 if (cd->disk->policy) {
                             → Función que finaliza la petición en curso
1235                         cdrom_end_request(drive, 0);
1236                         return ide_stopped;
1237                 }
1238         } else {
                         → Se corrige cualquier posible rareza que haya en el paquete
                         de petición
1243                     restore_request(rq);

1245                     → Se intenta leer del buffer. En caso de satisfacer la
                         petición completa se retorna.
1246                     if (cdrom_read_from_buffer(drive))
1247                             return ide_stopped;
1248         }

             → Si llegamos aquí, se trataba de una escritura y el disco no estaba
             protegido contra ella, o era una lectura que no se satisfizo con el
             buffer

             → Se intenta activar el DMA. Para ello, los frames deben  estar
             alineados (similar a las palabras en la memoria de un PC).
             Para la escritura dicho requisito es indispensable, si no, se aborta
             el proceso (en la lectura simplemente no se activa el DMA)
1253         if ((rq->nr_sectors & (sectors_per_frame - 1)) ||
1254             (rq->sector & (sectors_per_frame - 1))) {
1255                 if (write) {
1256                         cdrom_end_request(drive, 0);
1257                         return ide_stopped;
1258                 }
1259                 cd->dma = 0;
1260         } else
1261                 cd->dma = drive->using_dma;
1262

             → Limpiamos el buffer
1264         cd->nsectors_buffered = 0;
1265
1266         if (write)
1267                 cd->devinfo.media_written = 1;
```

```
1268
            → Comenzamos a enviar el commando, indicando la función que se
            llamará cuando el dispositivo esté preparado (tercer parámetro)
1270        return cdrom_start_packet_command(drive, 32768,
                                              cdrom_start_rw_cont);
1271 }
```

→ Función que indirectamente desde la función anterior (realmente desde la que se encuentra en el return) cuando el dispositivo interrumpe (avisando de que los registros ya se encuentran inicializados).
   Envía el comando de lectura/escritura, dejando preparada la función manejadora de la interrupción que tratará la petición.

```
803 static ide_startstop_t cdrom_start_rw_cont(ide_drive_t *drive)
804 {
            → Obtenemos la petición
805         struct request *rq = HWGROUP(drive)->rq;
806
            → rq_data_dir(rq) es una macro que extrae de rq el campo
            correspondiente que indica el tipo de petición
807         if (rq_data_dir(rq) == READ) {
808                 unsigned short sectors_per_frame =
809                         queue_hardsect_size(drive->queue) >> SECTOR_BITS;

810                 int nskip = rq->sector & (sectors_per_frame - 1);

                    → Si el comando es de lectura y el sector pedido no está
                    alineado(es decir, no comienza al principio de un bloque de
                    datos) se ajusta para que lo esté. Es decir resta a
                    rq->current_nr_sectors lo necesario para que esté a
                    comienzo de un bloque de datos
822                 if (nskip > 0) {
823
824                         if (rq->current_nr_sectors !=
825                             bio_cur_sectors(rq->bio)) {
826                                 printk(KERN_ERR
                                           "%s: %s: buffer botch (%u)\n",
827                                        drive->name, __FUNCTION__,
828                                        rq->current_nr_sectors);
829                                 cdrom_end_request(drive, 0);
830                                 return ide_stopped;
831                         }
                            → Realmente es una resta, ya que nskip debe ser < 0
832                         rq->current_nr_sectors += nskip;
833                 }
834         }
835 #if 0
836         else
837
838                 rq->cmd[1] = 1 << 3;
839 #endif
840         → Establecemos el timeout de respuesta del dispositivo
841         rq->timeout = ATAPI_WAIT_PC;
842
843         → Enviamos el commando, indicando la función manejadora de la
            interrupción que se producirá cuando el dispositivo haya recibido
            el comando y esté disponible para enviar o recibir datos
844         return cdrom_transfer_packet_command(drive, rq, cdrom_newpc_intr);
845 }
```

- Función realiza la lectura/escritura

→ Ésta es la función que lee o escribe del/en el dispositivo. Será llamada cuando el dispositivo haya recibido la petición correspondiente y esté disponible para recibir/enviar datos (que lo comunica con una interrupción).
Si se usó DMA, se llamará a esta función cuando la transferencia haya terminado

```
993static ide_startstop_t cdrom_newpc_intr(ide_drive_t *drive)
994{
995         struct cdrom_info *info = drive->driver_data;
996         struct request *rq = HWGROUP(drive)->rq;
997         xfer_func_t *xferfunc;
998         ide_expiry_t *expiry = NULL;
999         int dma_error = 0, dma, stat, ireason, len, thislen, uptodate = 0;
1000        int write = (rq_data_dir(rq) == WRITE) ? 1 : 0;

1001        unsigned int timeout;
1002        u8 lowcyl, highcyl;
1003
1004        → Descubrimos si hubo errores de DMA
1005        dma = info->dma;
1006        if (dma) {
1007                info->dma = 0;
1008                dma_error = HWIF(drive)->ide_dma_end(drive);
1009                if (dma_error) {
1010                        printk(KERN_ERR "%s: DMA %s error\n", drive->name,
1011                                        write ? "write" : "read");
1012                        ide_dma_off(drive);
1013                }
1014        }

1015        → Comprobamos errores reportados por el dispositivo
1016        if (cdrom_decode_status(drive, 0, &stat))
1017                return ide_stopped;
1018
1019        → Si se usó DMA, la transferencia ya se ha completado
1022        if (dma) {
                → Si hubo error retornamos
1023                if (dma_error)
1024                        return ide_error(drive, "dma error", stat);
1025                if (blk_fs_request(rq)) {
1026                        ide_end_request(drive, 1, rq->nr_sectors);
1027                        return ide_stopped;
1028                }

                → Si no, saltamos al final para finalizar la petición
1029                goto end_request;
1030        }
1031
1032        → No se usó DMA, se emplea el modo PIO (E/S programada, es decir,
        que el procesador interviene en la transferencia)

        → Se obtiene el motivo de la interrupción (se usa más abajo) y el
        tamaño de los datos a transferir.
1035        ireason = HWIF(drive)->INB(IDE_IREASON_REG) & 0x3;
1036        lowcyl  = HWIF(drive)->INB(IDE_BCOUNTL_REG);
1037        highcyl = HWIF(drive)->INB(IDE_BCOUNTH_REG);
1038
1039        len = lowcyl + (256 * highcyl);
```

```
1040
1041        thislen = blk_fs_request(rq) ? len : rq->data_len;
1042        if (thislen > len)
1043                thislen = len;
1044
1045        → Se comprueba si realmente hay algo que transmitir y se tratan
            ciertos errores
1048        if ((stat & DRQ_STAT) == 0) {
1049                if (blk_fs_request(rq)) {
1054                        uptodate = 1;
1055                        if (rq->current_nr_sectors > 0) {
1056                                printk(KERN_ERR "%s: %s: data underrun "
1057                                                "(%d blocks)\n",
1058                                                drive->name, __FUNCTION__,
1059                                                rq->current_nr_sectors);
1060                                if (!write)
1061                                        rq->cmd_flags |= REQ_FAILED;
1062                                uptodate = 0;
1063                        }
1064                        cdrom_end_request(drive, uptodate);
1065                        return ide_stopped;
1066                } else if (!blk_pc_request(rq)) {
1067                        ide_cd_request_sense_fixup(rq);
1069                        uptodate = rq->data_len ? 0 : 1;
1070                }
1071                goto end_request;
1072        }
1073        → Se verifica el contenido de la petición (rq)
1077        if (ide_cd_check_ireason(drive, rq, len, ireason, write))
1078                return ide_stopped;
1079

            → Se realizan varias comprobaciones sin importancia didáctica
1080        if (blk_fs_request(rq)) {
1081                if (write == 0) {
1082                        int nskip;
1083
1084                        if (ide_cd_check_transfer_size(drive, len)) {
1085                                cdrom_end_request(drive, 0);
1086                                return ide_stopped;
1087                        }
1088
1093                        nskip = min_t(int, rq->current_nr_sectors
1094                                        - bio_cur_sectors(rq->bio),
1095                                        thislen >> 9);
1096                        if (nskip > 0) {
1097                                ide_cd_drain_data(drive, nskip);
1098                                rq->current_nr_sectors -= nskip;
1099                                thislen -= (nskip << 9);
1100                        }
1101                }
1102        }
1103

            → Se asigna a xferfunc la función adecuada dependiendo de si se
            trata de una lectura (ireason = 0) o una escritura (ireason != 0)
1104        if (ireason == 0) {
1105                write = 1;
                        → atapi_output_bytes: envía datos al dispositivo
1106                xferfunc = HWIF(drive)->atapi_output_bytes;
1107        } else {
1108                write = 0;
                        → atapi_input_bytes: recibe datos del dispositivo
```

```
1109                    xferfunc = HWIF(drive)->atapi_input_bytes;
1110            }
1111
                → Por fin se transfieren los datos
1115            while (thislen > 0) {
1116                    u8 *ptr = blk_fs_request(rq) ? NULL : rq->data;
1117                    int blen = rq->data_len;
1118
1119                    → Se realizan varias comprobaciones
1122                    if (rq->bio) {
1123                            if (blk_fs_request(rq)) {
1124                                    ptr = rq->buffer;
1125                                    blen = rq->current_nr_sectors << 9;
1126                            } else {
1127                                    ptr = bio_data(rq->bio);
1128                                    blen = bio_iovec(rq->bio)->bv_len;
1129                            }
1130                    }
1131
1132                    if (!ptr) {
1133                            if (blk_fs_request(rq) && !write)
1138                                    cdrom_buffer_sectors(drive, rq->sector,
1139                                                         thislen >> 9);
1140                            else {
1141                                    printk(KERN_ERR
                                               "%s: confused, missing data\n",
1142                                                       drive->name);
1143                                    blk_dump_rq_flags(rq, rq_data_dir(rq)
1144                                                      ? "cdrom_newpc_intr, write"
1145                                                      : "cdrom_newpc_intr, read");
1146                            }
1147                            break;
1148                    }
1149
1150                    if (blen > thislen)
1151                            blen = thislen;
1152
                    → Se llama a la función para leer o escribir los datos
                    (bloque a bloque)
1153                    xferfunc(drive, ptr, blen);
1154
                    → Se disminuye el número de datos a transmitir en un tamaño
                    igual al tamaño de un bloque
1155                    thislen -= blen;
1156                    len -= blen;
1157
                    → Se actualizan los buffers y sus contadores
1158                    if (blk_fs_request(rq)) {
1159                            rq->buffer += blen;
1160                            rq->nr_sectors -= (blen >> 9);
1161                            rq->current_nr_sectors -= (blen >> 9);
1162                            rq->sector += (blen >> 9);
1163
1164                            if (rq->current_nr_sectors == 0 && rq->nr_sectors)
1165                                    cdrom_end_request(drive, 1);
1166                    } else {
1167                            rq->data_len -= blen;

1175                            if (rq->bio)
1176                                    blk_end_request_callback(rq, 0, blen,
1177                                                             cdrom_newpc_intr_dummy_cb);
```

```
1178                        else
1179                                rq->data += blen;
1180                }
1181                if (!write && blk_sense_request(rq))
1182                        rq->sense_len += blen;
1183        }
1184
                → Comprobaciones varias y reinicialización de los timeouts
1188        if (!blk_fs_request(rq) && len > 0)
1189                ide_cd_pad_transfer(drive, xferfunc, len);
1190
1191        if (blk_pc_request(rq)) {
1192                timeout = rq->timeout;
1193        } else {
1194                timeout = ATAPI_WAIT_PC;
1195                if (!blk_fs_request(rq))
1196                        expiry = cdrom_timer_expiry;
1197        }
1198
1199        ide_set_handler(drive, cdrom_newpc_intr, timeout, expiry);
1200        return ide_started;
1201
                → Aquí se llega encaso de error desde otras partes de la función
1202end_request:
1203        if (blk_pc_request(rq)) {
1204                unsigned long flags;
1205                unsigned int dlen = rq->data_len;
1206
1207                if (dma)
1208                        rq->data_len = 0;
1209
1210                spin_lock_irqsave(&ide_lock, flags);
1211                if (__blk_end_request(rq, 0, dlen))
1212                        BUG();
1213                HWGROUP(drive)->rq = NULL;
1214                spin_unlock_irqrestore(&ide_lock, flags);
1215        } else {
1216                if (!uptodate)
1217                        rq->cmd_flags |= REQ_FAILED;
1218                cdrom_end_request(drive, uptodate);
1219        }
1220        return ide_stopped;
1221}
```

- **Función principal**

→ Función de la que parte todo. Es la que se debe llamar cuando se quiera hacer una lectura o escritura.

```
1320 static ide_startstop_t
1321 ide_do_rw_cdrom (ide_drive_t *drive, struct request *rq,
                     sector_t block)
1322 {
1323         ide_startstop_t action;
1324         struct cdrom_info *info = drive->driver_data;
1325
1326         if (blk_fs_request(rq)) {

                 → Comprobación de errores (...)

                 → Si se trata de una lectura y el bloque pedido no se ha
                 buscado dentro del disco, se busca.
1340             if ((rq_data_dir(rq) == READ) &&
                     IDE_LARGE_SEEK(info->last_block, block,
                                    IDECD_SEEK_THRESHOLD) &&
                     drive->dsc_overlap) {

1341                     action = cdrom_start_seek(drive, block);
1342             } else
                 → Si era de escritura o de lectura y ya se había buscado el
                 bloque, se escriben o leen los datos
1343                     action = cdrom_start_rw(drive, rq);
1344             info->last_block = block;

                 → Se retorna el estado del dispositivo (parado o activo)
1345             return action;

         → Resto de acciones menos importantes (como reseto, etc)
         (...)
1346         } else if (blk_sense_request(rq) || blk_pc_request(rq) ||
1347                    rq->cmd_type == REQ_TYPE_ATA_PC) {
1348             return cdrom_do_block_pc(drive, rq);
1349         } else if (blk_special_request(rq)) {
1350
1353             cdrom_end_request(drive, 1);
1354             return ide_stopped;
1355         }
1356
         → Hubo error
1357         blk_dump_rq_flags(rq, "ide-cd bad flags");
1358         cdrom_end_request(drive, 0);
1359         return ide_stopped;
1360 }
```

BIBLIOGRAFÍA

- http://lxr.free-electrons.com/source/drivers/
- Linux cross reference (lxr.linux.no)
- www.wiquipedia.org