CD-ROM

Pablo Amoedo Paz

Álvaro López Espinosa

Introducción

- El **disco compacto (CD)** es un soporte digital óptico utilizado para almacenar cualquier tipo de información.
- Fue desarrollado en 1980 por Sony y Philips.
- Hoy en día tecnologías como el DVD han hecho que su uso sea cada vez menor.

Introducción

Características:

- Información almacenada: Audio, vídeo, imágenes, texto…
- Capacidad: 650 ó 700 MB
- Material: Policarbonato plástico con una capa reflectante de aluminio
- Vida útil: Alrededor de 10 años
- Tipos:
  - CD-ROM: Sólo lectura
  - CD-R: Grabable
  - CD-RW: Regrabable

Introducción:

Almacenamiento de la Información

- Información almacenada en formato digital
- La superficie del disco no es uniforme, está formada por hoyos y valles
- El ángulo de reflexión de la luz láser depende de si incide en un hoyo o un valle, lo que se aprovecha para almacenar información

Almacenamiento de la Información

- **Trama**: Bloque de datos más pequeño. Tiene 588 bits:
  - 24 de sincronización
  - 14 de control
  - 536 de datos
  - 14 de control
- **Sector**: Conjunto de 98 tramas

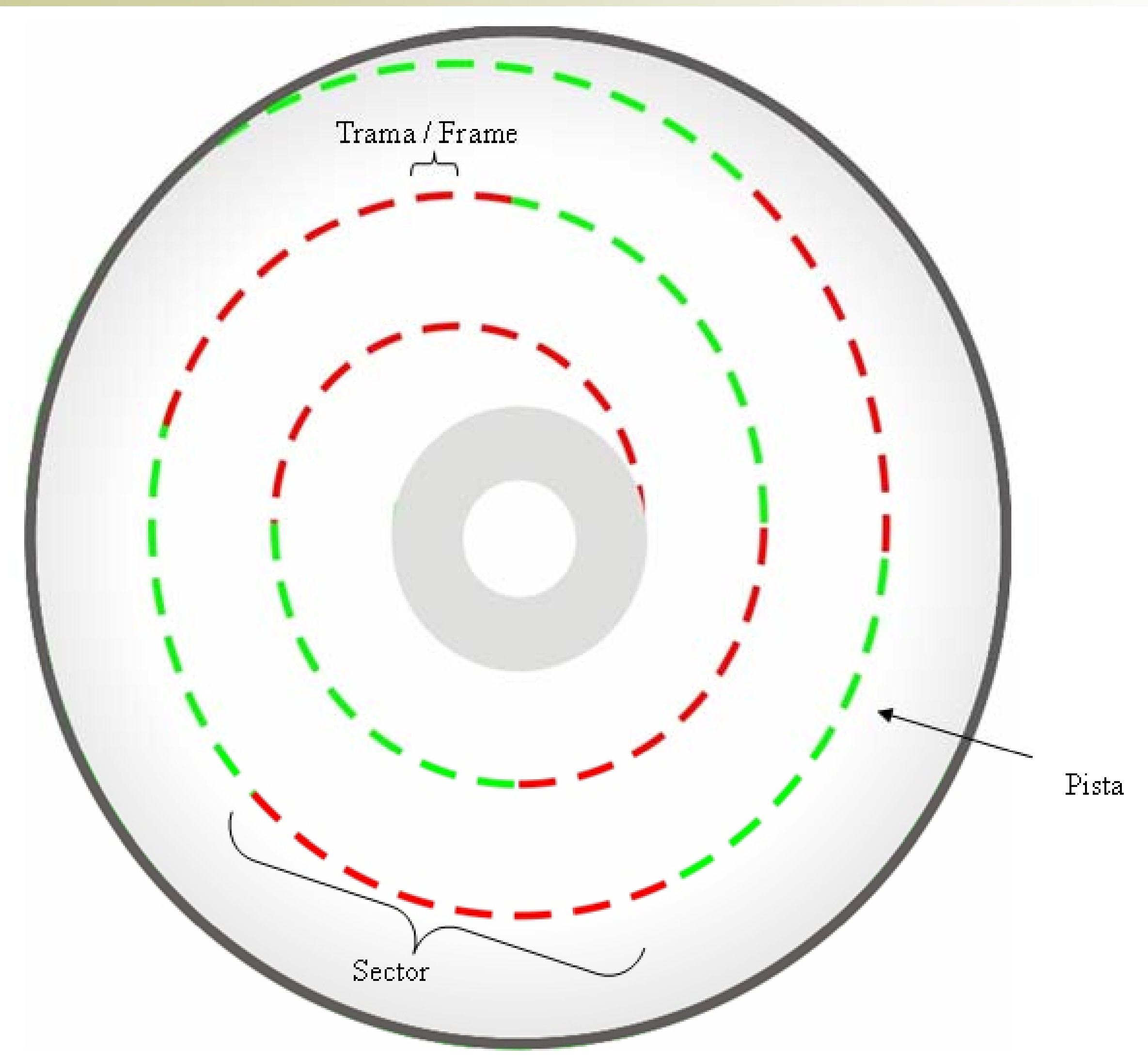
Introducción:

Almacenamiento de la Información

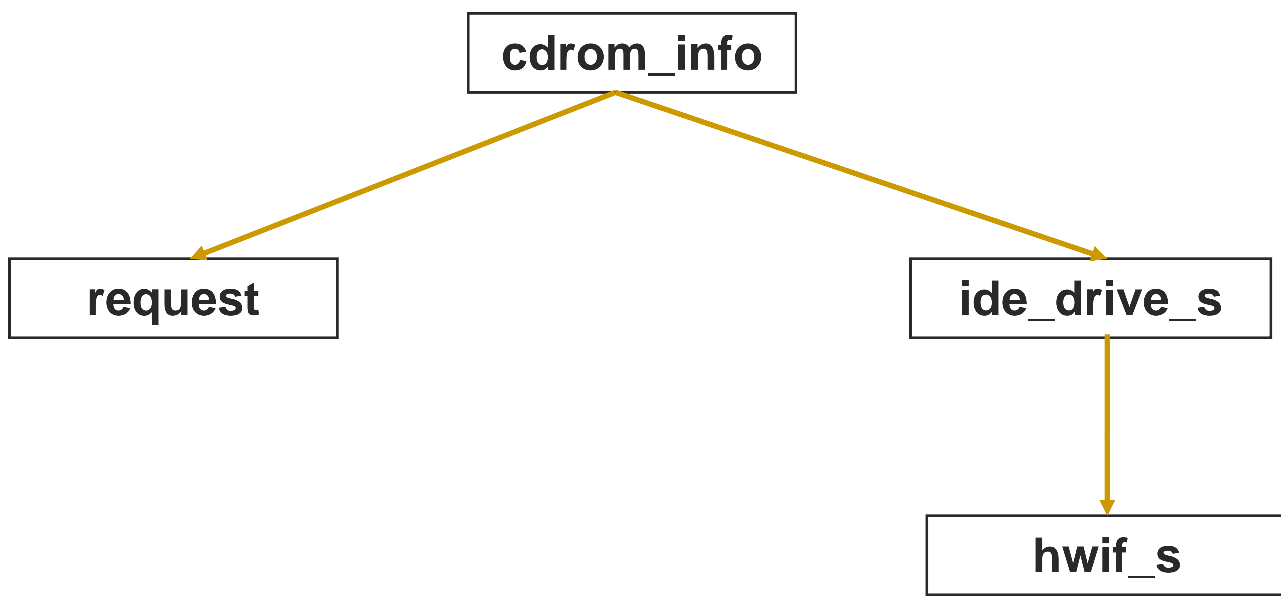
- **Pista:** Única, va del centro al exterior del disco
- El disco tiene que girar más lento cuando la cabeza lectora está cerca del centro
- **TOC**: Tabla de contenidos, es el índice de cada CD, guarda la posición inicial de cada secuencia contigua de datos

Introducción:

Almacenamiento de la Información



Estructuras



Estructuras: cdrom_info

- Proporciona información sobre el dispositivo de cd-rom.
- Definida en el fichero “/drivers/ide-cd.h”.

```
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;
```

Estructuras: cdrom_info

```
122         unsigned long   sector_buffered; → Primer sector en el búfer
123         unsigned long   nsectors_buffered; → Número de sectores
124         unsigned char   *buffer; → Búfer
        …
130         struct request request_sense_request; → Puntero a la
estructura request
131         int dma; → Indica si el dma está activo
132         unsigned long last_block; → Último bloque
133         unsigned long start_seek; → Posición de comienzo de búsqueda
        …
137         u8 max_speed; → Velocidad máxima del dispositivo
138         u8 current_speed; → Velocidad actual del dispositivo
141         struct cdrom_device_info devinfo; → Información genérica
143         unsigned long write_timeout;
144 };
```

Estructuras: ide_drive_s

- 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”

Estructuras: ide_drive_s

```
544 typedef struct ide_drive_s {
545         char name[4]; → nombre del dispositivo
548         request_queue_t *queue; → cola de peticiones
550         struct request  *rq; → petición actual
551         struct ide_drive_s *next; → lista circular
552         void *driver_data; → información extra del driver
553         struct hd_driveid *id; → información del modelo del dispositivo
554         struct proc_dir_entry *proc; → carpeta del dispositivo
555         struct ide_settings_s *settings; → carpeta en la que está la
configuracion del dispositivo
556         char            devfs_name[64]; /* devfs crap */
558         struct hwif_s *hwif; → puntero a estructura hwif_s
560         unsigned long sleep; → valor que indica el tiempo que tiene que estar
el dispositivo inactivo
```

Estructuras: ide_drive_s

```
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
    …
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
    …
573         u8      waiting_for_dma; → dma en proceso
    …
607         u8      init_speed; → velocidad de transferencia inicial
    …
624         u8      sect; → número de sectores por pista
    …
635         u64     capacity64; → número de sectores total
    …
} ide_drive_t;
```

Estructuras: hwif_s

- Estructura de la que dispone todo dispositivo IDE
- Se renombra como **ide_hwif_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"

Estructuras: hwif_s

```
652 typedef struct hwif_s {
653         struct hwif_s *next; → puntero al siguiente de la lista circular
     …
658         char name[6];  → nombre de la interfaz
     …º
669         u8 index; → índice que indica el dispositivo
     …
718         void (*atapi_input_bytes)(ide_drive_t *, void *, u32); → lectura
719         void (*atapi_output_bytes)(ide_drive_t *, void *, u32); → escritura
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 *); → comienza 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);
     ...
}____cacheline_internodealigned_in_smp ide_hwif_t;
```

Estructuras: request

- Estructura que almacena información sobre la petición hecha
- Definida en el fichero "/include/linux/blkdev.h"

```
144 struct request {
        …
148         struct request_queue *q; →  cola de peticiones
150         unsigned int cmd_flags;
151         enum rq_cmd_type_bits cmd_type; →  indica para qué tipo de dispositivo
se hace la  petición (ejemplo: para IDE ATA/ATAPI)
157         sector_t sector; →  siguiente sector a leer/escribir
159         unsigned long nr_sectors →  número de sectores restantes
```

Estructuras: request

```
162         unsigned int current_nr_sectors; → número de sectores que quedan por completar en el segmento actual
…
188         struct gendisk *rq_disk; → información específica del disco
189         unsigned long start_time; → fecha de la petición
...
206         char *buffer; → búfer en el que se guardará la petición
...
225         unsigned int timeout; → tiempo máximo de espera
...
236 };
```

Funciones

- Hay que tener presente que se trata de un dispositivo lento, por lo que se debe trabajar con interrupciones.
- A todas las funciones se les pasa la estructura genérica *ide_drive_t* con la información del dispositivo.
- Nos centraremos en las funciones de lectura/escritura, pues son las más importantes.
- Hay muchas más como expulsar la bandeja, reinicializar la unidad, ect.

Funciones

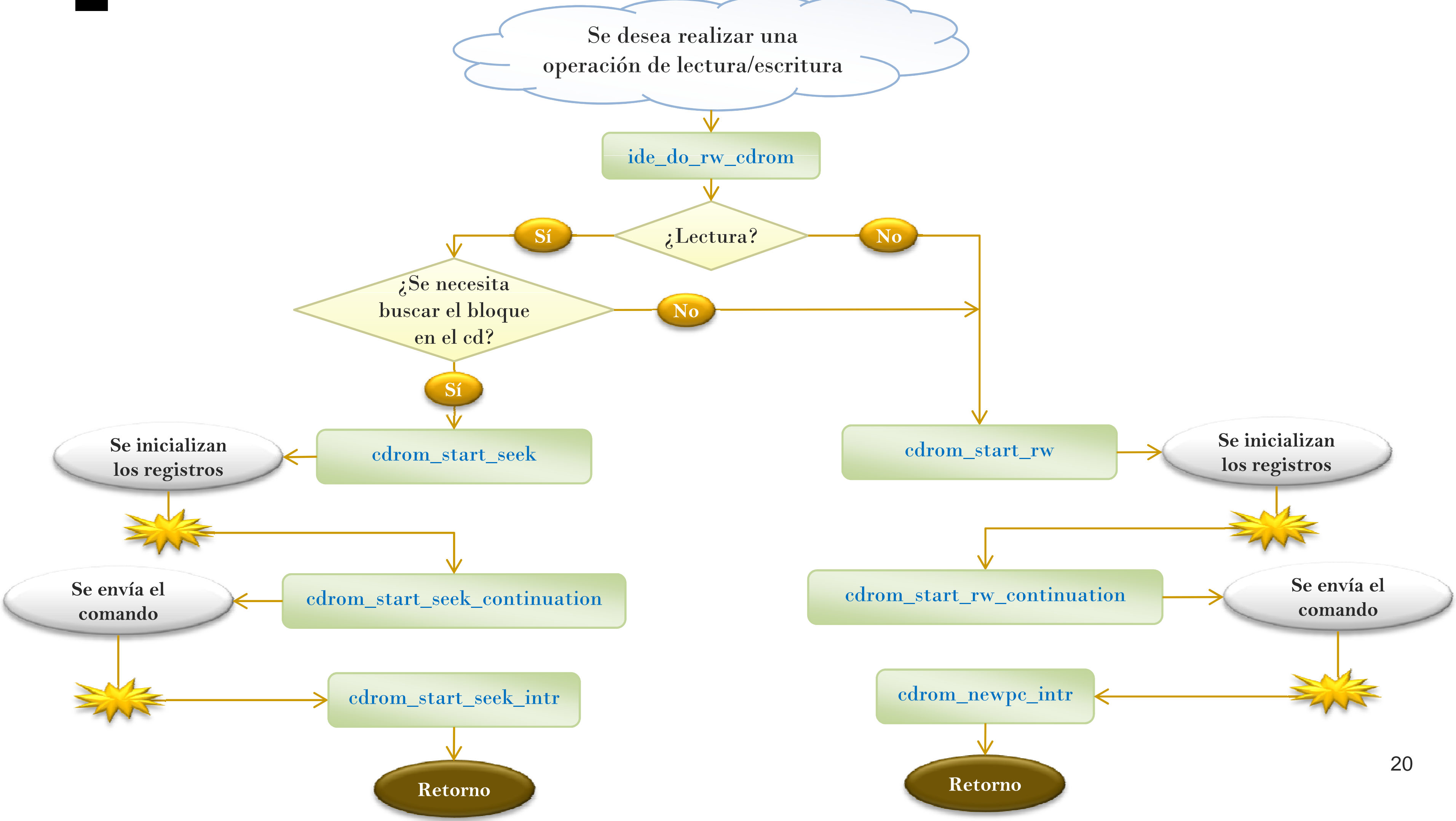
- Muchas funciones devuelven un valor del siguiente "enum":

```
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;
```

- Lectura y escritura comparten funciones

Funciones.

Esquema general



Funciones.

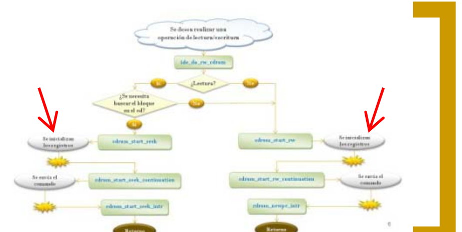
Inicialización de una petición

→ 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}
```

Funciones.

Inicialización de registros



→ 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
```

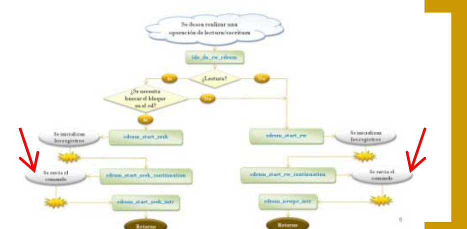
Funciones.

Inicialización de registros

```
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;
```

Funciones.

Envío de un comando

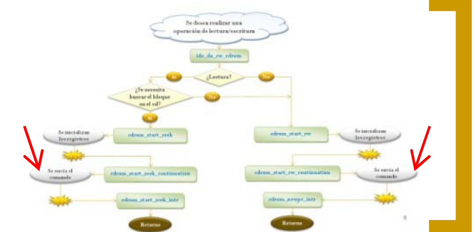


→ Envía el comando al dispositivo. Antes se debe haber llamado a la función anterior para que se inicialicen los registros. Ahora "handler" será la función que se llamará cuando el comando se termine de enviar.

```
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;
```

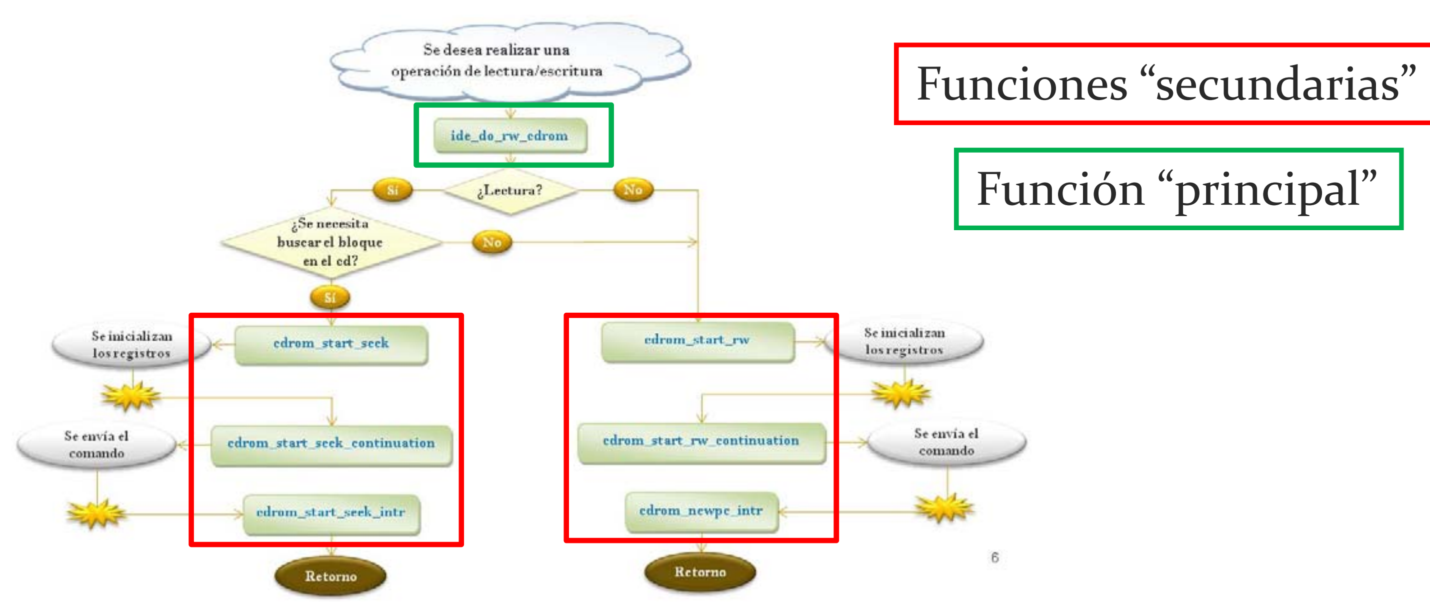
Funciones.

Envío de un comando



```
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}
```

Funciones. Lectura/Escritura



- Usan las anteriores como "nexo" para llamarse unas a otras.

- Se valen de funciones "auxiliares" para tratar con un buffer interno (de Linux, no del dispositivo)

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
```

Funciones auxiliares

```
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}
```

Funciones. Lectura/Escritura

Funciones auxiliares

→ 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

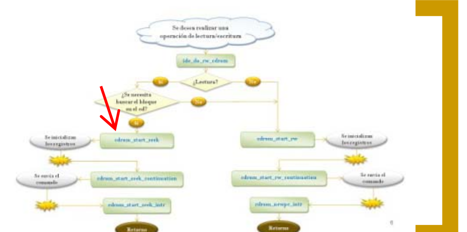
```
740static int cdrom_read_from_buffer (ide_drive_t *drive)
741{
742         struct cdrom_info *info = drive->driver_data;
743         struct request *rq = HWGROUP(drive)->rq;
744         unsigned short sectors_per_frame;
745
746         sectors_per_frame = queue_hardsect_size(drive->queue) >> SECTOR_BITS;
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)
```

Funciones auxiliares

```
                    → 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         }
768
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         }

777         if (rq->current_nr_sectors == 0)
778                 cdrom_end_request(drive, 1);
779
            (...)
791
            → No se pudieron leer todos los datos del buffer
792         return 0;
```

Funciones. Lectura/Escritura

Funciones Secundarias



→ 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}
```

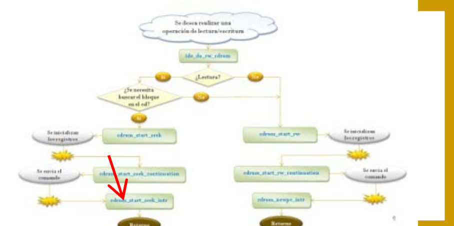
Funciones Secundarias

→ 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}
```

Funciones. Lectura/Escritura

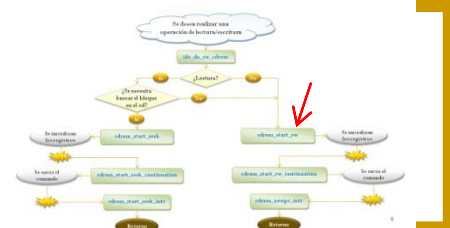
Funciones Secundarias



→ 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}
```

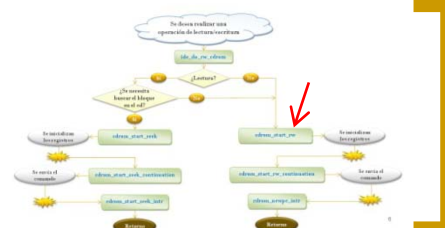
Funciones Secundarias



→ 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         }
```

Funciones Secundarias



→ 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 }
```

Funciones Secundarias

→ 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;

            (...) Alineaciones de datos y demás comprobaciones (nada importante)

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 }
```

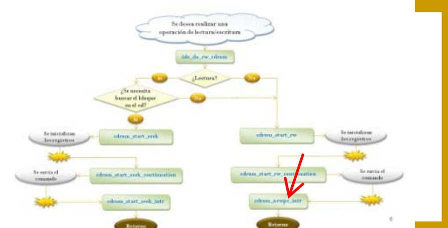
Funciones. Lectura/Escritura

Funciones Secundarias

→ 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{
1003
1004        (...)
            Declaraiones de variables y comprobación de errores
            (...)
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        }
```

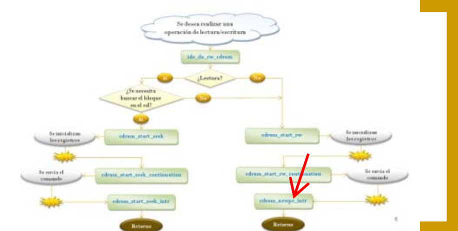
Funciones Secundarias



```
                → 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
                (...)
```

Funciones. Lectura/Escritura

Funciones Secundarias



```
            → 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        }
```

Funciones Secundarias

```
                → 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
                        → 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                    }
                        (...) Más comprobaciones (...)
1183            }
```

Funciones. Lectura/Escritura

Funciones Secundarias

→ 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        }

            (...)

1203
1220        return ide_stopped;
1221}
```

Funciones. Lectura/Escritura

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)
         (...)
```