A.S.O:

Controlador De Disco y Cinta SCSI

Autores:
César J. Caraballo Viña.
Alejandro Valido Hernández.

Indice.

1. Introducción.

La interfaz SCSI (Small Computer System Interface) ha sido tradicionalmente relegada a tareas y entornos de ámbito profesional, en los que prima más el rendimiento, la flexibilidad y la fiabilidad. Para empezar, SCSI es una estructura de bus separada del bus del sistema. De esta forma, evita las limitaciones propias del bus del PC.

Además, en su versión más sencilla esta norma permite conectar hasta 7 dispositivos SCSI (serían 8 pero uno de ellos ha de ser la propia controladora) en el equipo; y las ventajas no se reducen al número de periféricos, sino también a su tipo: se puede conectar prácticamente cualquier dispositivo (escáner, impresoras, CD-ROM, unidades removibles, etc.) siempre que cumplan con esta norma.

Otra enorme ventaja de SCSI es su portabilidad; esto quiere decir que podemos conectar nuestro disco duro o CD-ROM (o lo que sea) a ordenadores Macintosh, Amiga, etc., que empleen también la norma SCSI. Un detalle a resaltar que todos los periféricos SCSI son inteligentes; es decir, cada uno posee su propia ROM donde almacena sus parámetros de funcionamiento. En especial, es la controladora el dispositivo más importante de la cadena SCSI, que al poseer su propia BIOS puede sobrepasar limitaciones de la ROM BIOS del sistema.

Entre los distintos entornos de trabajo en los que se usan los sistemas SCSI destacan los servidores de Web (por el gran volumen de información que tienen que manejar para los usuarios), los sistemas de grabación de CDs (por su fiabilidad de mantener una tasa de transferencia constante), los servidores de red (por la rapidez de acceso y tasa de transferencia de la información contenida en ellos), y para la edición de vídeo (tanto para la digitalización como para su procesado).

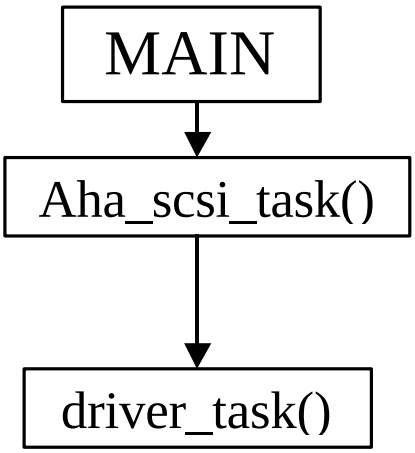
Posiblemente lo que hace destacar a SCSI en su rendimiento, bastante superior a IDE al no depender del bus del sistema; no obstante, no todo iban a ser ventajas: SCSI es más caro que IDE, y en la mayoría de las ocasiones, más complejo de configurar, aunque esto último es cada vez menos problemáticos, ya que es preciso resaltar que la norma SCSI también ha evolucionado y mejorado; citaremos a continuación sus diferentes modalidades.

Norma Scsi	**Ancho Bus**	**Megas/segundo**
Scsi-1	8 bits	3 M/s
Scsi-2	8 bits	5 M/s
Fast Scsi-2	8 bits	10 M/s
Fast/wide Scsi-2	16 bits	20 M/s
Ultra Scsi	8/16 bits	20/40 M/s
Ultra 2 Scsi lvd	8/16 bits	40/80 M/s

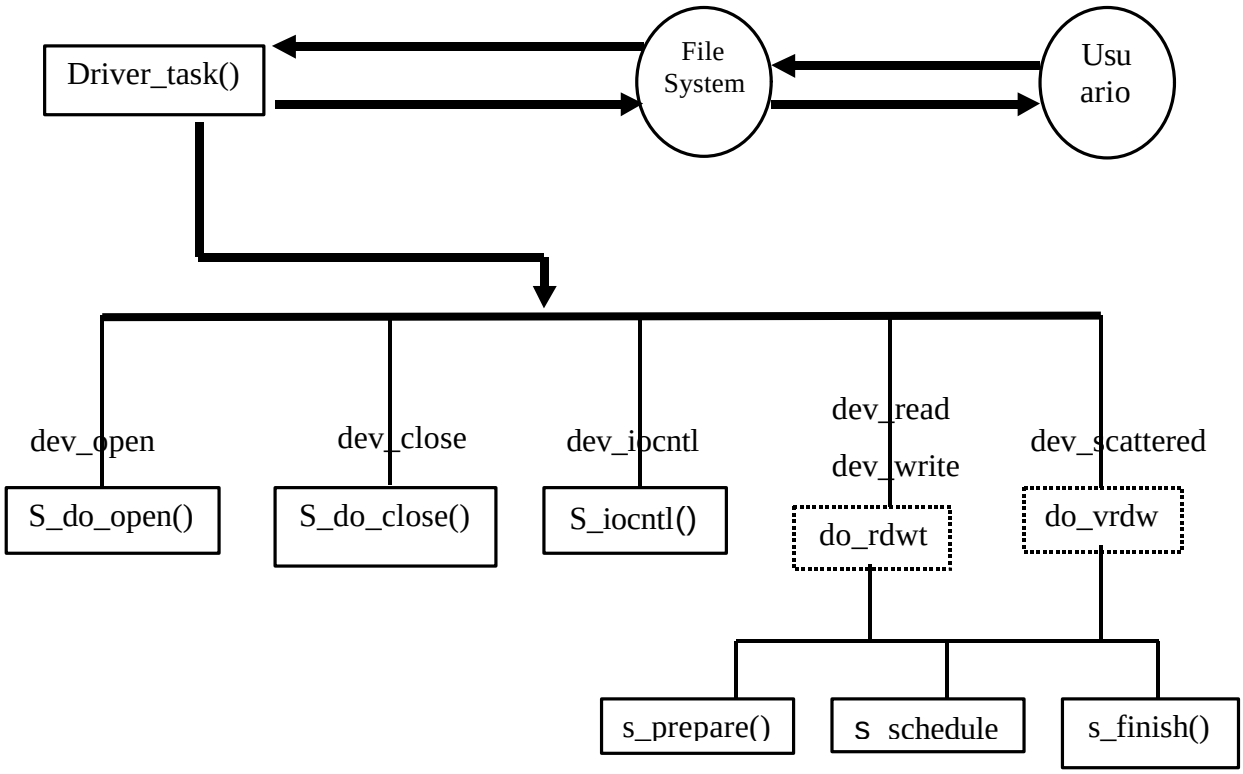
Para que un sistema sea capaz de controlar todos los dispositivos SCSI que no vienen incorporados en la BIOS del mismo es necesario añadir una nueva capa de servicios denominada ASPI (Advanced SCSI Programming Interface) que se encarga de traducir las habituales instrucciones del sistema al formato propio de los dispositivos SCSI

Con la capa ASPI los programas se pueden dividir en dos bloques, el gestor de bajo nivel, que es totalmente dependiente del sistema en el que se ejecute y el propio modulo ASPI, que contiene la información de los distintos dispositivos y el modo de acceso a cada uno de ellos.

En el proceso de arranque de minix la función main llama la función Aha_scsi_task, que inicializa los dispositvos existentes, seguidamente llama a función driver_task (driver.c) pasando una tabla con las funciones de manejo de los dispositivos Scsi. Acontinución se muestra un esquema.



A continuación mostramos un gráfico donde se representan las asociaciones de funciones de driver con las Scsi.



2. Estructuras.

2.1. Cdb0_t, Cdb1_t.

Estructuras en la que se recogen las condiciones del **bloque descriptor de comandos (command descriptor block**) de un dispositivo. Donde se indica la operación que se lleva a cabo, sobre que unidad y sobre que número de bloques. Sus principales campos para dispositivos de disco son los siguientes:

Cdb0_t	**Cdb1_t**	
byte d_scsi_op	byte d_scsi_op	Codigo de operación
big24 d_lba	byte d_lunra, big32 d_lba	guarda el LUN (logic unit number) y la dirección lógica de los bloques
Byte d_nblocks	big16 d_nblocks	el número de bloques a transferir
Byte d_control	byte d_control	byte de control

Para el caso de que el dispositivo sea una cinta la estructura tiene los siguientes campos:

cdb0_t	
byte t_scsi_op	es el código de operación
byte t_fixed	indica si el dispositivo tiene longitud fija
big24 t_trlength	la cantidad de información a transferir
byte t_control	byte de control

```
/* SCSI Grupo 0 , estructura Bloque descriptor de comando */
typedef union {
   struct {        /* comandos e/s para disco */
         byte d_scsi_op;                  /* SCSI código de operación */
#            define SCSI_UNITRDY   0x00      /* Test dispositivo funciona */
#            define SCSI_REWIND    0x01      /* Rebobinar */
#            define SCSI_REQSENSE 0x03      /* Petición de búsqueda dispositivo */
#            define SCSI_RDLIMITS  0x05      /* Leer bloque limite */
#            define SCSI_READ        0x08        /* Grupo 0 código de lectura */
#            define SCSI_WRITE       0x0A        /* Grupo 0 código escritura */
#            define SCSI_WREOF      0x10      /* Escribir fin de fichero */
#            define SCSI_SPACE    0x11        /* Espacio sobre marcas fichero/bloques */
#            define SCSI_INQUIRY  0x12       /* Grupo 0 requerimiento características */
#            define SCSI_MDSELECT 0x15     /* Grupo 0 modo selección */
#            define SCSI_ERASE    0x19                  /* borrar cinta */
#            define SCSI_MDSENSE  0x1A     /* Grupo 0 modo búsqueda */
#            define SCSI_STRTSTP  0x1B     /* Empezar/parar */
#            define SCSI_LOADUNLD 0x1B     /* Cargar */
      big24 d_lba;                              /* LUN y dirección lógica de bloque */
         byte d_nblocks;                        /* Tamaño de la transferencia en bloques
*/
         byte d_control;                        /* Bits de control */
   } d;

struct { /* comandos e/s para cinta */
         byte t_scsi_op;                  /* SCSI código de operación */
         byte t_fixed;                    /* Longitud fija? */
         big24 t_trlength;                  /* longitud transferencia */
         byte t_control;          /* Bits de control */
```

```
   } t;
} cdb0_t;

typedef union {
   struct {       /* Comandos de entrada salida de disco */
         byte d_scsi_op;                    /* Código de operación SCSI */
#           define SCSI_CAPACITY 0x25      /* Capacidad de lectura */
#           define SCSI_READ1    0x28             /* Grupo 1 lectura */
#           define SCSI_WRITE1   0x2A      /* Grupo 1 escritura */
         byte d_lunra;                 /* LUN etc. */
       big32 d_lba;                        /* Dirección lógica de bloque */
         byte reserved;
         big16 d_nblocks;                  /* tamaño de la transferencia en bloques */
         byte d_control;                   /* Bits de control */
   } d;
} cdb1_t;
```

2.2. Ccb_t.

Es la estructura para el bloque de control de comandos.Una estructura de este tipo estará incluida en cada petición (request_r) y se utilizará para controlar la ejecución de cada comando:

Byte opcode	Tipo de operación
big24 datalen	Longitud de dato
big24 dataptr	Puntero de dato
Byte hastat	Estado de la controladora
Byte tarstat	Estado del dispositivo
Byte cmd[sizeof(cbd1_t)]	Bloque descriptor de comandos SCSI
Byte sense[sizeof(sense_t))	Información de sobre el estado de las peticiones

```
/* AHA estructura Bloque de control de comandos */
typedef struct {
   byte opcode;              /* Código de operación */
#        define CCB_INIT            0x00        /* Comando de iniciación */
#        define CCB_TARGET          0x01        /* Comando modo objetivo */
#        define CCB_SCATTER         0x02        /* iniciar con dispersión/reunión */
   byte addrcntl;            /* Dirección y control de dirección */
#        define ccb_scid(id)        (((id)<<5)&0xE0)      /* campo SCSI ID */
#        define CCB_OUTCHECK        0x10         /* comprobar longitud */
#        define CCB_INCHECK         0x08         /* comprobar longitud (obl.)*/
#        define CCB_NOCHECK         0x00         /* No comprobar longitud */
#        define ccb_lun(lun)        ((lun)&0x07)    /* campo SCSI LUN  */
   byte cmdlen;              /* Longitud comandos SCSI (6 para Grupo 0) */
   byte senselen;
#        define CCB_SENSEREQ        0x0E        /* requerimiento búsqueda */
#        define CCB_SENSEOFF        0x01        /* deshabilitar búsqueda */
   big24 datalen;            /* Longitud dato:  3 bytes, big endian */
   big24 dataptr;            /* Puntero dato: 3 bytes, big endian */
   big24 linkptr;            /* Puntero enlace: 3 bytes, big endian */
   byte linkid;          /* indentificador enlace de comandos */
```

```
   byte hastat;          /* Estado Host Adapter */
#        define HST_TIMEOUT        0x11          /* SCSI selección timeout */
   byte tarstat;         /* Estado del dispositivo actual */
#        define TST_CHECK          0x02          /* comprobado estado */
#        define TST_LUNBUSY        0x08          /* la unidad esta ocupada */
   byte reserved[2];           /* reservado */
   byte cmd[sizeof(cdb1_t)];   /* Bloque descriptor de comandos SCSI */
   byte sense[sizeof(sense_t)];/* Información de dispositivo SCSI */
} ccb_t;
```

2.3. Inquiry_t.

En esta estructura se almacenan los detalles físicos de un dispositivo (vendedor, tipo de unidad, número de versión, etc.):

byte devtype	tipo de dispositivo
byte devqual	Cualificación del tipo de dispositivo
byte stdver	Versión del estándar aprobado
byte format	Formato de los datos de respuesta
byte len	Longitud de la información sobrante
char vendor	Marca
char product	Producto
char revision	Versión
char extra	Especificaciones del vendedor

```
/* SCSI información */
typedef struct {
   byte devtype;              /* Tipo de dispositivo */
#        define SCSI_DEVDISK        0       /* acceso directo */
#        define SCSI_DEVTAPE        1       /* acceso secuencial */
#        define SCSI_DEVPRN         2       /* impresora */
#        define SCSI_DEVCPU         3       /* procesador */
#        define SCSI_DEVWORM        4       /* una escritura/múltiple lectura */
#        define SCSI_DEVCDROM       5       /* solo lectura acceso directo */
#        define SCSI_DEVSCANNER     6       /* Scanner */
#        define SCSI_DEVOPTICAL     7       /* memoria óptica */
#        define SCSI_DEVJUKEBOX     8       /* medio de carga */
#        define SCSI_DEVCOMM        9       /* dispositivo de comunicación */
#        define SCSI_DEVMAX         9       /* último dispositivo conocido */
#        define SCSI_DEVUNKNOWN     10      /* no conocido */
   byte devqual;              /* Cualificador */
#        define scsi_rmb(d)     (((d) & 0x80) != 0)     /* Removible? */
   byte stdver;          /* Versión */
#        define scsi_isover(v)   (((v) & 0xC0) >> 6)    /* ISO versión */
#        define scsi_ecmaver(v)  (((v) & 0x38) >> 3)  /* ECMA versión */
#        define scsi_ansiver(v)  ((v) & 0x07)          /* ANSI versión */
   byte format;          /* formato datos */
   byte len;             /* longitud de información */
   byte reserved[2];
   byte flags;
#        define scsi_sync(f)     (((f) & 0x10) != 0)          /* síncrono? */
```

```
    char vendor[8];             /* nombre vendedor */
    char product[16];   /* nombre producto */
    char revision[4];   /* nivel de revisión */
    char extra[20];             /* especificaciones del vendedor */
} inquiry_t;
```

2.4. Scsi.

Estructura donde se recogen los datos lógicos de un dispositivo:(nombre lógico de la unidad, estado actual, número de procesos que lo están usando, etc.).Se crea un vector con tantas estructuras como dispositivos SCSI permitidos.

También en esta estructura se diferencia ente los dispositivos de disco y los de cinta.

char targ	identificador del dispositivo	
char lun	número de unidad lógica	
char state	puede tomar los valores S_PRESENT, S_READY O S_RDONLY	
Unsigned block_size	tamaño de bloque del dispositivo	
Unsigned open_ct	número de procesos utilizando el dispositivo	
struct tape	char open_mode	abierto en modo lectura o escritura
	char at_eof	está en la marca de final de fichero
struct disk	struct device part[DEV_PER_DRIVE]	particiones primarias sd[0-4]
	Struct device subpart[SUB_PER_DRIVE]	subparticiones sd[1-4][a-d]

```
PRIVATE struct scsi {           /* para la tabla de dispositivos */
    char targ;                  /* SCSI ID actual */
    char lun;                   /* SCSI número unidad lógica (LUN) */
    char state;                 /* en línea? */
#       define S_PRESENT   0x01    /* existe el dispositivo */
#       define S_READY     0x02    /* el dispositivo esta listo */
#       define S_RDONLY    0x04    /* el dispositivo es de solo lectura */
    char devtype;                   /* SCSI_DEVDISK, SCSI_DEVTAPE, ... */
    unsigned block_size;            /* tamaño de bloque */
    unsigned count_max;             /* máximo en una escritura o lectura */
    unsigned open_ct;           /* numero de procesos utilizando dispositivo */
    union {
        struct {            /* datos para cinta */
            char open_mode;   /* abierto para lectura o escritura? */
            char at_eof;          /* marca fin fichero */
            char need_eof;              /* se necesita escribir fin de fichero */
            char tfixed;          /* cinta en modo fijo */
            struct mtget tstat;   /* información de estado */
            struct device dummypart;  /* algo para que s_prepare lo retorne */
        } tape;
        struct {            /* datos de disco */
            struct device part[DEV_PER_DRIVE];      /* primarias: sd[0-4] */
```

```
            struct device subpart[SUB_PER_DRIVE];           /* subparticiones: sd[1-4][a-
d] */
        } disk;
    } u;
} scsi[MAX_DEVICES];
```

2.5. Request.

En esta estructura se guardan los datos de cada petición a los dispositivos (número de bytes a transferir,número de reintentos permitidos, primer byte a transferir, etc.).Se hace uso de la estructura del bloque de control de comandos descrita anteriormente.

unsigned count	número de bytes a transferir
unsigned retry	número de intentos permitidos
unsigned long pos	primer byte en el dispositivo para transferir
ccb_t ccb	contiene un bloque de control de comando en el que se especifica la orden a ejecutar
dma_t dmalist[NR_IOREQS]	peticiones de acceso por medio de la DMA (como máximo NR_IOREQS=MIN(NR_BUF,64))

```
/* Administration for one SCSI request. */
typedef struct request {
  unsigned count;                   /* número de bytes a transferir */
  unsigned retry;                   /* número de intentos permitidos */
  unsigned long pos;            /* primer byte en dispositivo para transferir */
  ccb_t ccb;                    /* Bloque de control de comandos */
  dma_t dmalist[NR_IOREQS];                 /* dispersa/reunida dma  */
  dma_t *dmaptr;                    /* para añadir a las entradas dispersas/reunidas */
  dma_t *dmalimit;                  /* lista de limitaciones dma */
  struct iorequest_s *iov[NR_IOREQS];       /* requerimientos de e/s afectadas */
} request_t;
```

2.6. S_dtab.

Es la estructura que se le pasa a driver_task donde se indican las funciones que se utilizan para el acceso a la controladora.

s_name	nombre del dispositivo
s_do_open	Petición de abrir o montar,inicializa el dispositivo
s_do_close	libera el dispositivo
s_do_ioctl	obtener /fijar la geometría de una partición
s_prepare	preparar para entrada salida un dispositivo menor
s_schedule	planifica transferencias de E/S
s_finish	realiza la E/S
nop_cleanup	no se necesita operación de cleanunp
s_geometry	informar sobre la geometría de un disco

```
PRIVATE struct driver s_dtab = {
```

```
  s_name,               /* nombre dispositivo actual */
  s_do_open,   /* iniciar dispositivo */
  s_do_close,  /* cerrar dispositivo */
  s_do_ioctl,    /* control de cinta y particiones */
  s_prepare,    /* preparación de e/s */
  s_schedule,  /* organización de e/s: obtener parámetros, etc. */
  s_finish,             /* realizar la operación de e/s*/
  nop_cleanup,          /* no necesita formato */
  s_geometry   /* proporciona los datos geométricos del dispositivo */
};
```

2.7. Variables globales importantes.

Entre las variables globales problamente las que se citan aquí son las más importantes, por su utilidad para las funciones.

request_t request	petición actual
unsigned long s_nextpos	próximo byte a transferir
unsigned long s_buf_blk	bloque de disco actualmente en el buffer
int s_opcode	operación de lectura o escritura
int aha_irq	canal de interrupción configurado
int s_must	¿la petición actual debe ser finalizada? Variable usada por s_finish
inquiry_t inqdata	resultado de un comando “Inquiry”

3. Funciones.

3.1. Aha_scsi_task.

Se ejecuta en el momento del encendido del sistema.
Prepara todos los dispositivos tipo Scsi existentes (discos, cintas, cd-rom, etc.), para ello va asignando a cada unidad un nombre lógico con s_name, y va inicializando cada dispositivo con s_prepare.
Esta función es llamada por main(), y pasa la estructura s_dtab (donde se indican las funciones de acceso a los dispositivos scsi) al módulo driver.c.

```
PUBLIC void aha_scsi_task()
{/* Establecer funciones y números undidades lógicas */
int i;
struct scsi *sp;
long v;
char *name;
static char fmt[] = "d,d";
for (i = 0; i < MAX_DEVICES; i++)
  {
        (void) s_prepare(i * DEV_PER_DRIVE);
        sp = s_sp;

        /* Buscar en las variables de entorno información */
        name = s_name();
```

```
        v = i;
        (void) env_parse(name, fmt, 0, &v, 0L, 7L);
        sp->targ = v;

        v = 0;
        (void) env_parse(name, fmt, 1, &v, 0L, 7L);
        sp->lun = v;
  }

  driver_task(&s_dtab);
}
```

3.2. S_prepare.

Prepara un dispositivo scsi para realizar operaciones de entrada/salida.
En función del valor de device, vamos mirando el tipo de dispositivo sobre el que se va a trabajar.
Inicializamos las variables 's_sp' (puntero al dispositivo) 's_type' (tipo de dispositivo) y 's_dv' (partición activa)
Todas estas operaciones permitirán realizar operaciones de entrada/salida sobre el dispositivo SP del sistema (dispositivo activo en este momento).

```
PRIVATE struct device *s_prepare(device)
  int device;{    /* preparación de e/s */
  rq->count = 0;          /* no hay requerimientos */
  s_must = TRUE;      /* se tiene que realizar la primera tranferencia */
  s_buf_blk = -1;         /* invalidar buffer */

  if (device < NR_DISKDEVS) {                           /* sd0, sd1,... */
        s_type = TYPE_SD;
        s_sp = &scsi[device / DEV_PER_DRIVE];
        s_dv = &s_sp->part[device % DEV_PER_DRIVE];
  } else
  if ((unsigned) (device - MINOR_hd1a) < NR_SUBDEVS){    /* sd1a, sd1b, ... */
        device -= MINOR_hd1a;
        s_type = TYPE_SD;
        s_sp = &scsi[device / SUB_PER_DRIVE];
        s_dv = &s_sp->subpart[device % SUB_PER_DRIVE];
  } else
  if ((unsigned) (device - MINOR_st0) < NR_TAPEDEVS){    /* nrst0, rst0,... */
        device -= MINOR_st0;
        s_type = device & 1 ? TYPE_RST : TYPE_NRST;
        s_sp = &scsi[device >> 1];
        s_dv = &s_sp->dummypart;
  } else {
        return(NIL_DEV);
  }
  return(s_dv);}
```

3.3. S_name.

Devuelve una cadena con el nombre lógico del dispositivo actual.

```
PRIVATE char *s_name(){     /* Devuleve el nombre del dispositivo actual */
```

```
static char name[] = "sd35";
  int n = (s_sp - scsi);
  switch (s_type) {
  case TYPE_SD:                         /* disco: sd* */
          name[1] = 'd';
          n *= DEV_PER_DRIVE;
          break;
  case TYPE_RST:              /* dispositivo de cinta: st* */
  case TYPE_NRST:
          name[1] = 't';
          break;
  }
  if (n < 10) {
          name[2] = '0' + n;                    /* número disco */
          name[3] = 0;
  } else {
          name[2] = '0' + n / 10;           /* número disco */
          name[3] = '0' + n % 10;                   /* número partición */
  }
  return name;}
```

3.4. S_do_open.

Realiza una apertura o una instalación de un dispositivo, inicializándolo**.** Hacemos una llamada a AHA_RESET con lo que conseguimos obtener variables de ambiente de la controladora.
A continuación procedemos a verificar si el dispositivo es accesible.
En el caso de realizar una apertura en modo de escritura de un dispositivo de solo lectura generaremos un error.
En el caso de realizar esta operación sobre un dispositivo de tipo disco, comprueba que no sea la primera vez que se abra, en tal caso montará el dispositivo con la función PARTITION (DRVLIB.C) que inicializa las tablas de particiones de un dispositivo.
Para el caso de una cinta, entonces comprobará que está no esté ocupada con otro proceso (Solo un proceso puede acceder a la cinta).
Incrementará en una variable asociada al dispositivo que indica el número de procesos activos que le están haciendo peticiones.

```
PRIVATE int s_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
   struct scsi *sp;
   int r;
   if (aha_irq == 0 && !aha_reset()) return(EIO);
/* no hay controladora */

  if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);         /* no existe
dispositivo */
  sp = s_sp;

  if ((r = scsi_probe()) != OK) return(r);

  if (sp->state & S_RDONLY && m_ptr->COUNT & W_BIT) return(EACCES);
switch (sp->devtype) {
```

```
  case SCSI_DEVDISK:
  case SCSI_DEVWORM:
  case SCSI_DEVCDROM:
  case SCSI_DEVOPTICAL:
        /* leer tabla de particiones */
        if (sp->open_ct == 0) {          /* si no hay procesos utilizando */
                partition(&s_dtab, (int) (sp-scsi) * DEV_PER_DRIVE, P_PRIMARY);
        }
        break;

case SCSI_DEVTAPE:
        /* dispositivo de cinta */
        if (sp->open_ct > 0) return(EBUSY);  /* esta en uso */

        sp->open_mode = m_ptr->COUNT;

        /* si open(..., O_WRONLY) después escribir una marca fin fichero */

        sp->need_eof = ((sp->open_mode & (R_BIT|W_BIT)) == W_BIT);
        break;
  }
  sp->open_ct++;       /* otro proceso más */
  return(OK);
}
```

3.5. Scsi_probe.

Comprueba si existe un dispositivo y está preparado, obteniendo los datos del mismo.
Verifica que el dispositivo está presente, usando la función scsi_sense.
Si es la primera vez que accedemos a él, entonces con la función scsi_inquiry obtenemos todos los datos del dispositivo.
Verifica que el dispositivo está listo, y si no lo está inicia un comando para que este pase al estado de listo.
Enviamos de nuevo un comando de petición de respuesta de unidad lista.
En caso de que no esté lista verificamos la razón.
Una vez solucionado el problema, verificamos nuevamente si el dispositivo está preparado, obteniendo sus características.
Si no está listo el dispositivo, se obtienen sus datos geométricos (tamaños de bloque, capacidades, etc.)

```
PRIVATE int scsi_probe(){/* mirar si el dispositivo existe y está listo */
  struct scsi *sp = s_sp;
  sense_t *sense;
  int r, key;

  /* Hay algo? */
  if ((r = scsi_sense()) != OK) {
        if (sp->state & S_PRESENT) {
                printf("%s: offline\n", s_name());
                sp->state = 0;
        }
        return(r);
```

```
  }

  if (!(sp->state & S_PRESENT)) {
        /* primer contacto con un dispositivo, que tipo dispositivo es? */

        if ((r = scsi_inquiry()) != OK) return(r);

        sp->devtype = inqdata.devtype;
  }

  if (!(sp->state & S_READY)) {

        /* si es un disco arrancar, si es una cinta cargar */
        (void) scsi_simple(SCSI_STRTSTP, 1);
  }
/* Mirar si la unidad esta lista.  Un disco tiene que tener una velocidad de giro,
un
    floppy o una cinta tiene que estar libre. */

  while ((key = scsi_simple(SCSI_UNITRDY, 0)) != SENSE_NO_SENSE) {
        /* Porque no esta listo? */

        sp->state &= ~S_READY;

        switch (key) {
        case SENSE_UNIT_ATT:
                /* El medio sufrio un cambio, volver a intentar */
                break;
        case SENSE_NOT_READY:
                /* Obtener más datos para saber porque no esta lista */
                sense = &ccb_sense(rq);
                switch ((sense->add_code << 8) | sense->add_qual) {
                case 0x0401:
                        /* si se esta poniendose a listo, esperar */
                        milli_delay(1000);
                        break;
                case 0x0402:
                        /* Se requiere un comando de inicialización */
                        if (scsi_simple(SCSI_STRTSTP, 1) !=
SENSE_NO_SENSE)
                                return(EIO);
                        break;
                case 0x0403:
                        /* se requiere interveción manual */
                case 0x3A00:
                        /* el dispositivo no aparece */
                        printf("%s: no media loaded\n", s_name());
                        return(EIO);
                default:
                        /* por alguna razon no esta dizponible */
```

```
                        printf("%s: not ready\n", s_name());
                        return(EIO);
                }
                break;
        default:
                /* el dispositivo esta en un estado desconocido  */
                if (key != SENSE_NOT_READY) {
                        printf("%s: hardware error\n", s_name());
                        return(EIO);
                }
        }
  }

if (!(sp->state & S_PRESENT)) {
        /* realizar otra petición de información */
        if (scsi_inquiry() != OK) return(EIO);
        /* Decir que clase de dispositvo se encontro. */
        printf("%s: %-7s %.48s\n",
                s_name(),
                inqdata.devtype > SCSI_DEVMAX ? "UNKNOWN"
                                : scsi_devstr[inqdata.devtype],
                inqdata.vendor        /* + producto + revisión + extra */);
  }
if (!(sp->state & S_READY)) {
        /* Obtener geometria, limites, etc. */
        switch (sp->devtype) {
        case SCSI_DEVDISK:
        case SCSI_DEVWORM:
        case SCSI_DEVCDROM:
        case SCSI_DEVOPTICAL:
                if (scsi_ndisk() != OK) return(EIO);
                break;
        case SCSI_DEVTAPE:
                if (scsi_ntape() != OK) return(EIO);
                break;
        default:
                printf("%s: unsupported\n", s_name());
                return(EIO);
        }
  }
  return(OK);
}
```

3.6. Scsi_sense.

Realiza una búsqueda de un determinado dispositivo (vemos si dicho dispositivo existe y su estado actual).
Inicialmente, se realiza una petición de búsqueda de dispositivo a la función 'scsi-simple'. Esta configurará los valores del struct SP.

Según la respuesta, emitiremos un error (si el dispositivo no existe o si hay algún problema con el host adapter) o veremos la disponibilidad del dispositivo (en caso de una respuesta favorable).

```
PRIVATE int scsi_sense(){
int key;
sense_t *sense = (sense_t *) tmp_buf;  /* Do a request sense to find out if a target
exists or to check out  a unit attention condition. */

  key = scsi_simple(SCSI_REQSENSE, sizeof(sense_t));
  if (rq->ccb.hastat == HST_TIMEOUT) return(ENXIO);       /* nothing there */
  if (rq->ccb.hastat != 0) return(EIO);             /* muy malo */

  /* There is something out there for sure. */
  if (key == SENSE_UNIT_ATT || sense_key(sense->key) == SENSE_UNIT_ATT) {

          /* dispositivo en estado ”mirame" , probablemente se cambio el medio */
          s_sp->state &= ~S_READY;
  }
  return(OK);
}
```

3.7. Scsi_inquiry.

Pregunta por las características 'físicas' de un determinado dispositivo.
Hace una llamada a 'scsi-simple' a fin de que se inicialice el struct inqdata del tipo inquiry_t, donde quedan almacenados los valores específicos del dispositivo en cuestión.
En caso que el dispositivo a analizar no sea reconocido por el sistema, se devolverá 'unknown'.

```
PRIVATE int scsi_inquiry()
{
  /* vaciar la estructura (vaciar el buffer). */
  memset(tmp_buf, '\0', sizeof(inquiry_t));

  /* llamar a scsi_inqury para obtener información físicas del dispositivo */
  if (scsi_simple(SCSI_INQUIRY, sizeof(inquiry_t)) != SENSE_NO_SENSE)
          return(EIO);
  inqdata = * (inquiry_t *) tmp_buf;

  if (inqdata.len == 0) {
          /* el dispositivo no retornó texto significativo. */
          strcpy(inqdata.vendor, "(unknown)");
  }

  /* Los tres bits superiores del dispositivo tienen que ser cero para que su LUN exista
*/
  if ((inqdata.devtype & 0xE0) != 0) return(ENXIO);

  return(OK);
}
```

3.8. Scsi_ndisk.

Obtiene el tamaño de bloque y la capacidad de un dispositivo tipo disco.
Actualiza los campos de la estructura scsi donde se hace referencia al estado del dispositivo, el tamaño de los bloques, y la capacidad del dispositivo.
El dispositivo debe ser de tipo disco sino da un error.
Si el dispositivo es un CDROM pone su estado a sólo lectura.
Llama a Scsi_simple para comprobar si el disco está protegido contra escritura en cuyo caso da un error.

```
PRIVATE int scsi_ndisk(){      /* Gather disk data, capacidad y tamaño bloque */
struct scsi *sp = s_sp;
unsigned long capacity = -1, block_size = SECTOR_SIZE;
byte *buf = tmp_buf;  /* el dispositivo tiene que ser un disco */
  if (s_type != TYPE_SD) return(EIO);

  if (sp->devtype == SCSI_DEVCDROM) {

          /* Solo lectura por definición */
          sp->state |= S_RDONLY;
  } else {

          /* SCSI modesense para buscar si el dispositivo esta protegido contra escritura
*/
          if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE) return(EIO);

          /* Protegido contra escritura? */
          sp->state &= ~S_RDONLY;
          if (buf[2] & 0x80) sp->state |= S_RDONLY;

          /* no se puede escribir en un disco worm, no es prudenteen este momento */
          if (sp->devtype == SCSI_DEVWORM) sp->state |= S_RDONLY;
  }
/* Obtener la capacidad y el tamaño de bloque */
  group1();
  rq->ccb.opcode = CCB_INIT;
  ccb_cmd1(rq).scsi_op = SCSI_CAPACITY;

if (scsi_command(tmp_phys, 8) == SENSE_NO_SENSE) {
          capacity = b2h32(buf + 0) + 1;
          block_size = b2h32(buf + 4);
          printf("%s: capacity %lu x %lu bytes\n",s_name(), capacity, block_size);
  } else {
          printf("%s: unknown capacity\n", s_name());
  }

/* si se devuelve un tamaño de bloque superior a 4Kb se supone que hubo un error */
  if (block_size > 4096) {
          printf("%s: can't handle %lu byte blocks\n", s_name(), block_size);
          return(EIO);
  }

  sp->block_size = block_size;
#if _WORD_SIZE > 2
  /* Lo mantenemos sin llegar grupo 0 de comandos */
```

```
  sp->count_max = 0x100 * block_size;
#else
  sp->count_max = block_size > UINT_MAX/0x100 ? UINT_MAX : 0x100 * block_size;
#endif


/* No soporta disco mayores de 4Gb */

  if (capacity > ((unsigned long) -1) / block_size)
        sp->part[0].dv_size = -1;
  else
        sp->part[0].dv_size = capacity * block_size;

  /* Para finalizar se indica que el dispositivo existe */
  sp->state |= S_PRESENT|S_READY;

  return(OK);
}
```

3.9. Scsi_ntape.

Comprueba que el dispositivo sea de cinta y si no da un error.
Obtiene los límites de bloques y si los tamaños de los bloques son uniformes o no.
Actualiza los campos de la estructura para los dispositivos tipo cinta de los scsi.

```
PRIVATE int scsi_ntape(){/*datos cinta, limtes de bloque, tamaño de bloque fijo o no */
  struct scsi *sp = s_sp;
  unsigned minblk;
  unsigned long maxblk;
  byte *buf = tmp_buf;  /* es dispositivo tiene que ser una cinta */
  if (s_type != TYPE_RST && s_type != TYPE_NRST) return(EIO);

  /* limites de lectura */
  if (scsi_simple(SCSI_RDLIMITS, 6) != SENSE_NO_SENSE) return(EIO);
  minblk = b2h16(buf + 4);
  maxblk = b2h24(buf + 1);

  printf("%s: limits: min block len %u, max block len %lu\n",
        s_name(), minblk, maxblk);

  if (sp->state & S_PRESENT) {
        /* Obtener el tamaño de bloque actual */
        if (sp->tfixed) minblk= maxblk= sp->block_size;
  }

  sp->tstat.mt_dsreg = DS_OK;
  sp->tstat.mt_erreg = 0;
  sp->tstat.mt_fileno = 0;
  sp->tstat.mt_blkno = 0;
  sp->tstat.mt_resid = 0;
if (minblk == maxblk) {
        /*longitud de bloque fija */
        sp->tfixed = TRUE;
        sp->block_size = minblk;
```

```
            sp->tstat.mt_blksize = minblk;
            sp->count_max = UINT_MAX;

} else {
            /* longitud de bloque variable */
            sp->tfixed = FALSE;
            sp->block_size = 1;
            sp->tstat.mt_blksize = 0;
            sp->count_max = maxblk == 0 ? UINT_MAX : maxblk;
  }
/* SCSI modesense. */
  if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE) return(EIO);

  /* ¿protegido contra escritura? */
  sp->state &= ~S_RDONLY;
  if (buf[2] & 0x80) sp->state |= S_RDONLY;
/*densidad y tamaño de bloque */
  if (buf[3] >= 8) {
            printf("%s: density 0x%02x, nblocks %lu, block len ",
                        s_name(),
                        buf[4],
                        b2h24(buf + 4 + 1));
            printf(sp->tfixed ? "%lu\n" : "variable\n", b2h24(buf + 4 + 5));
  }

  sp->state |= S_PRESENT|S_READY;
  return(OK);
}
```

3.10. S_schedule.

Recogemos los datos necesarios para realizar una petición de entrada/salida.
En principio, recogemos unos datos generales, como son el número de bytes a leer/escribir, posición sobre la que trabajar, tipo de operación, etc.
Diferenciaremos dos casos: dispositivos tipo disco y tipo cinta.
En los primeros, nos interesa el bloque sobre el que escribir/leer (las operaciones sobre CD-ROM se realizan aparte) y su posición en el dispositivo.
Si estamos usando una unidad de cinta, nos interesa contrastar que el número de bytes a leer/escribir sea múltiplo del tamaño de bloque de la unidad, así como del estado actual del dispositivo (por si está en una condición errónea, en cuyo caso habría que resetear el dispositivo).
Utilizamos numap una función definida en system.c y que calcula la dirección de memoria física dada una dirección virtual .
Por último, añade la nueva petición al contador de peticiones y prepara el struct rq con los datos necesarios para realizarla.

```
PRIVATE int s_schedule(proc_nr, iop)
int proc_nr;                        /* proceso que solicita */
struct iorequest_s *iop;                        /* puntero a posicion lectura o escritura */
{
/* Gather las operaciones e/s en bloques consecutivos deben ser leidas/escritas
 * en un solo comando Scsi usando scatter/gather DMA.
```

```
 */
  struct scsi *sp = s_sp;
  int r, opcode, spanning;
  unsigned nbytes, count;
  unsigned long pos;
  phys_bytes user_phys, dma_phys;
  static unsigned dma_count;
  static struct iorequest_s **iopp;         /* para añadir a las peticiones de e/s */
  static phys_bytes dma_last;               /* direccion de la última entrada */

  /* número de bytes a leer/escribir */
  nbytes = iop->io_nbytes;

  /* posicion en el dispositivo */
  pos = iop->io_position;

  /* dirección fisica de usuario de donde leer o escribir*/
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);
/* lectura o escritura? */
  opcode = iop->io_request & ~OPTIONAL_IO;

  switch (sp->devtype) {
  case SCSI_DEVCDROM:
  case SCSI_DEVWORM:
  case SCSI_DEVDISK:
  case SCSI_DEVOPTICAL:
        /* que bloque en disco y como cerrar? */

if (pos >= s_dv->dv_size) return(OK);                   /* At EOF */

        if (pos + nbytes > s_dv->dv_size) nbytes = s_dv->dv_size - pos;
        pos += s_dv->dv_base;

        if ((nbytes % sp->block_size) != 0 || (pos % sp->block_size) != 0) {
                /* No en dispositivo de bloque boundary.  CD-ROM? */
                return(s_rdcdrom(proc_nr, iop, pos, nbytes, user_phys));
        }
        break;
case SCSI_DEVTAPE:
        if ((nbytes % sp->block_size) != 0)
                return(iop->io_nbytes = EINVAL);

        /* Vieja condicion de error? */
        if (sp->tstat.mt_dsreg == DS_ERR) return(iop->io_nbytes = EIO);

        if (opcode == DEV_READ && sp->at_eof) return(OK);
        s_nextpos = pos = 0;  /* pos ignorado */
        break;

  default:
        return(iop->io_nbytes = EIO);
  }
/* Comprueba si un dispositivo no esta preparado */
  if (!(sp->state & S_READY) && scsi_probe() != OK) return(EIO);
```

```
  if (rq->count > 0 && pos != s_nextpos) {
        /* Esta nueva petición no puede ser encadenada para la tarea actual */
        if ((r = s_finish()) != OK) return(r);
  }
/* Cominezo del siguiente bloque a encadenar */
  s_nextpos = pos + nbytes;
  spanning = FALSE;  /* establece si la petición spans varios vectores DMA */

  /* Mientras hallan bytes sin organizar en la petición */
  do {
        dma_phys = user_phys;
        if (rq->count > 0 && (
                rq->count == sp->count_max
                || rq->dmaptr == rq->dmalimit
                || !DMA_CHECK(dma_last, dma_phys)
        )) {
                /* Esta petición no se puede añadir a  scatter/gather list. */
                if ((r = s_finish()) != OK) return(r);
                s_must = spanning;

                continue;       /* intentar otra vez */
        }
if (rq->count == 0) {
                /* La primera petición en la cola, iniciar. */
                rq->pos = pos;
                s_opcode = opcode;
                iopp = rq->iov;
                rq->dmaptr = rq->dmalist;
                rq->retry = 2;
        }
        count = nbytes;
/* No sobrepasar el contador máximo de transferencia. */
        if (rq->count + count > sp->count_max)
                count = sp->count_max - rq->count;

        /* Nueva scatter/gather entrada. */
        h2b24(rq->dmaptr->dataptr, dma_phys);
        h2b24(rq->dmaptr->datalen, (u32_t) (dma_count = count));
        rq->dmaptr++;
        dma_last = dma_phys + count;

        /* Que peticion e/s? */
        *iopp++ = iop;

        /* Actualizar contadores. */
        rq->count += count;
        pos += count;
        user_phys += count;
        nbytes -= count;
        if (!(iop->io_request & OPTIONAL_IO)) s_must = TRUE;

        spanning = TRUE;     /* el resto de la petición debe ser realizada */
  } while (nbytes > 0);
```

```
  return(OK);
}
```

3.11. S_finish.

Envía una petición ya almacenada en el struct apuntado por rq a la controladora.
Al comenzar, verificamos la existencia de alguna petición.
Si el dispositivo es un disco, estaremos tratando con un dispositivo de acceso aleatorio, por lo que debemos permitir operaciones de este tipo.
Otro parámetro a considerar trabajando con discos es su tamaño, pues definirá el tipo de instrucciones que debemos usar (discos grandes requerirán operaciones del grupo uno, discos pequeños, del grupo 0).
Los estados que pueden darse al trabajar con estos dispositivos cuando termina la ejecución del comando son: un error (en caso de que no se pueda efectuar la instrucción) o reintentar la petición (en caso de encontrar la controladora ocupada).
Si el dispositivo de trabajo fuese uno tipo cinta, el tratamiento se diferenciará en la gran cantidad de problemas que pueden surgir y que hay que controlar (problemas con las marcas de fin de bloque, al rebobinar la cinta, al situarla en la posición correcta, etc.).

```
PRIVATE int s_finish()
{
/* Enviar la peticion de e/s gathered en *rq para el  host adapter. */

  struct scsi *sp = s_sp;
  unsigned long block;
  struct iorequest_s **iopp, *iop;
  int key;

  if (rq->count == 0) return(OK);         /* spurious finish */

  show_req();

  /* Si todas las peticiones son opcionales entonces no hagas este pizco. */
  if (!s_must && rq->count < 0x2000) {
          rq->count = 0;
          return(OK);
  }

iopp = rq->iov;
  iop = *iopp++;
retry:
  switch (sp->devtype) {
  case SCSI_DEVCDROM:
  case SCSI_DEVWORM:
  case SCSI_DEVDISK:
  case SCSI_DEVOPTICAL:
          /* Un comando SCSI de lectura/escritura para un dispositivo de acceso
aleatorio */
  block = rq->pos / sp->block_size;

          if (block < (1L << 21)) {
                  /* Usamos un comandop del grupo 0 para los disco pequeños. */
                  group0();
```

```
                rq->ccb.opcode = CCB_SCATTER;
                ccb_cmd0(rq).scsi_op =
                        s_opcode == DEV_WRITE ? SCSI_WRITE : SCSI_READ;
                h2b24(ccb_cmd0(rq).lba, block);
                ccb_cmd0(rq).nblocks = rq->count / sp->block_size;
        } else {
                /* Los discos grandes requieren un comando del grupo 1 */
                group1();
                rq->ccb.opcode = CCB_SCATTER;
                ccb_cmd1(rq).scsi_op =
                        s_opcode == DEV_WRITE ? SCSI_WRITE1 : SCSI_READ1;
                h2b32(ccb_cmd1(rq).lba, block);
                h2b16(ccb_cmd1(rq).nblocks, rq->count / sp->block_size);
        }
key = scsi_command(0L, 0);

        if (key == SENSE_NO_SENSE) {
                /* bien */;
        } else
if (key == SENSE_UNIT_ATT || key == SENSE_ABORTED_CMD) {
                /* Comprobar condición?   Bus reset most likely. */
                /* Abortar comando?  Maybe inentarlo con ayuda. */
                if (--rq->retry > 0) goto retry;
                return(iop->io_nbytes = EIO);
        } else
        if (key == SENSE_RECOVERED) {
                /* manejador de disco recuperandose de un error. */
                printf("%s: soft read error at block %lu (recovered)\n",
                        s_name(), b2h32(ccb_sense(rq).info));
                key = SENSE_NO_SENSE;
                break;
        } else {
                /* Ocurrió un error fatal. */
                return(iop->io_nbytes = EIO);
        }
        break;
case SCSI_DEVTAPE:
        /* Un comnado Scsi de lectura/escritura para dispositivos de acceso secuencial
*/
        group0();
        rq->ccb.opcode = CCB_SCATTER;
        ccb_cmd0(rq).scsi_op = s_opcode == DEV_WRITE ? SCSI_WRITE :
SCSI_READ;
        ccb_cmd0(rq).fixed = sp->tfixed;
        h2b24(ccb_cmd0(rq).trlength, rq->count / sp->block_size);

        key = scsi_command(0L, 0);

        if (key != SENSE_NO_SENSE) {
                /* Either at EOF o EOM, o un error de e/s. */
                if (sense_eof(key) || sense_eom(key)) {
                        /* no es un error, pero EOF or EOM. */
                        sp->at_eof = TRUE;
                        sp->tstat.mt_dsreg = DS_EOF;
/*  residual dice cuanto no fue leido. */
```

```
                        rq->count -= sp->tstat.mt_resid * sp->block_size;
                        if (sense_eof(key)) {
                                /* Fue sobre una marca de archivo. */
                                sp->tstat.mt_blkno = !sp->tfixed ? -1 :
                                        - (int) (rq->count / sp->block_size);
                                sp->tstat.mt_fileno++;
                        }
                }

                if (sense_ili(key)) {
                        /* longitud erronea en tamaño de bloque de la cinta. */

                        if (sp->tstat.mt_resid <= 0) {
                                /* un gran bloque no se puede leer. */
                                return(iop->io_nbytes = EIO);
                        }
                        /* Se leyó un bloque pequeño, esto es valido. */
                        rq->count -= sp->tstat.mt_resid;
                        sp->tstat.mt_dsreg = DS_OK;
                }
                if (key == SENSE_RECOVERED) {
                        /* El manejador de la cinta se esta recuperando de un error. */
                        printf("%s: soft %s error (recovered)\n",       s_name(),
                                s_opcode == DEV_READ ? "read" : "write");
                        key = SENSE_NO_SENSE;
                        sp->tstat.mt_dsreg = DS_OK;
                }
                if (sp->tstat.mt_dsreg == DS_ERR) {
                        /* erro fatal. */
                        return(iop->io_nbytes = EIO);
                }
} else {
                sp->tstat.mt_dsreg = DS_OK;
        }
        if (!sp->tfixed) {
                /* Las cintas de bloque variable leen registro por registro. */
                sp->tstat.mt_blkno++;
        } else
{
                /* Cintas de longitud fija, se transfieren multiples bloques. */
                sp->tstat.mt_blkno += rq->count / sp->block_size;
        }
        sp->need_eof = (s_opcode == DEV_WRITE);
        break;
  default:
        assert(0);
  }
  /* Eliminar los bytes transfridos de las peticiones de e/s. */
  for (;;) {
        if (rq->count > iop->io_nbytes) {
                rq->count -= iop->io_nbytes;
                iop->io_nbytes = 0;
        } else {
                iop->io_nbytes -= rq->count;
                rq->count = 0;
```

```
                break;
        }
        iop = *iopp++;
 }
 return(key == SENSE_NO_SENSE ? OK : EIO);}  /* retornar EIO para EOF */
```

3.12. S-rdcdrom.

Realiza lecturas del cd-rom.
Verifica que la petición sea de lectura. Ejecuta cualquier acceso al dispositivo (s_finish) y comienza a realizar la lectura de la información requerida.
En primer lugar verifica que la información pedida no se encuentre en el bufer del cd-rom. Si lo está, recoge la información desde esa fuente, en caso contrario, mediante scsi_command realiza la petición de la información al dispositivo hasta que toda la información sea recogida.

```
PRIVATE int s_rdcdrom(proc_nr, iop, pos, nbytes, user_phys)
int proc_nr;                    /* proceso de la petición */
struct iorequest_s *iop;                /* puntero a la petición de lectura o escritura */
unsigned long pos;                      /* posición de byte */
unsigned nbytes;                        /* número de bytes */
phys_bytes user_phys;                   /* dirección de usuario */
{
/* los CD-ROM tienen un bloque basico de tamaño 2k. Podriamos intentar establecer
un tamaño de bloque virtual más pequeño, pero muchos no lo soportan. Por lo tanto
utilizamos esta función.
 */
  struct scsi *sp = s_sp;
  int r, key;
  unsigned offset, count;
  unsigned long block;

  /* Solo lectura. */
  if ((iop->io_request & ~OPTIONAL_IO) != DEV_READ)
        return(iop->io_nbytes = EINVAL);

  /* terminar cualquier operación en espera. */
  if ((r = s_finish()) != OK) return(r);
do {
        /* Comprobar que el dispositivo este preparado. */
        if (!(sp->state & S_READY) && scsi_probe() != OK) return(EIO);

        block = pos / sp->block_size;
        if (block == s_buf_blk) {
                /* Alguna de las peticiones estan en el buffer. */
                offset = pos % sp->block_size;
                count = sp->block_size - offset;
                if (count > nbytes) count = nbytes;
                phys_copy(tmp_phys + offset, user_phys, (phys_bytes) count);
                pos += count;
                user_phys += count;
                nbytes -= count;
                iop->io_nbytes -= count;
```

```
        } else {
                /* Leer el bloque que contiene los bloques buscados */
                rq->retry = 2;
                do {
                        group1();
                        rq->ccb.opcode = CCB_INIT;
                        ccb_cmd1(rq).scsi_op = SCSI_READ1;
                        h2b32(ccb_cmd1(rq).lba, block);
                        h2b16(ccb_cmd1(rq).nblocks, 1);
                        key = scsi_command(tmp_phys, sp->block_size);
                } while (key == SENSE_UNIT_ATT && --rq->retry > 0);

                if (key != SENSE_NO_SENSE) return(iop->io_nbytes = EIO);

                s_buf_blk = block;      /* guardar el bloque en el buffer */
        }
  } while (nbytes > 0);
  return(OK);
}
```

3.13. S_do_close.

Elimina un proceso de los que están usando un determinado dispositivo.
Inicialmente, vemos el estado del dispositivo sobre el que se está trabajando. Si el dispositivo está preparado, tomamos su dirección en el struct SP.
Restamos uno del campo de SP que almacena el número de procesos activos (sp->open_ct).
Si el dispositivo es tipo disco, o una partición, habremos acabado el proceso. Sin embargo, si estuviésemos trabajando con un dispositivo tipo cinta, puede ocurrir que tengamos que resetearlo (rebobinarlo) y/o ponerle una marca de fin de fichero.

```
PRIVATE int s_do_close(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
  struct scsi *sp;

  if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
  sp = s_sp;

  sp->open_ct--;

  /* Disco y similares no causan problemas. */
  if (sp->devtype != SCSI_DEVTAPE) return(OK);

  sp->at_eof = FALSE;

  /* Escribir marca fichero si tuvo lugar una escritura. */
  if (sp->need_eof && sp->tstat.mt_dsreg != DS_ERR) {
        if (scsi_simple(SCSI_WREOF, 1) != SENSE_NO_SENSE) {
                printf("%s: failed to add filemark\n", s_name());
        } else {
                sp->tstat.mt_dsreg = DS_OK;
```

```
                sp->tstat.mt_blkno = 0;
                sp->tstat.mt_fileno++;
        }
  }
/*Rebobinar if es rebobinable. */
  if (s_type == TYPE_RST) {
        if (scsi_simple(SCSI_REWIND, 1) != SENSE_NO_SENSE) {
                printf("%s: failed to rewind\n", s_name());
        } else {
                sp->tstat.mt_dsreg = DS_OK;
                sp->tstat.mt_blkno = 0;
                sp->tstat.mt_fileno = 0;
        }
  }
  return(OK);
}
```

3.14. S_do_ioctl.

Realiza operaciones de control a los dispositivos.
Verifica que el dispositivo esté listo para operaciones de entrada/salida.
Si el dispositivo es de tipo disco, comprueba si la petición a realizar es de tipo extraer. Si lo es, emite el comando apropiado usando scsi_simple. En caso de que no lo sea, hace una llamada a do_diocntl, con la operación a realizar.
Si el dispositivo es de tipo cinta, entonces usando la función scsi_simple, emite los comandos para cada una de las operaciones,tales como rebobinado, tensado, etc..., generando mensajes de error si se produce algún tipo de problema.

```
PRIVATE int s_do_ioctl(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{  struct scsi *sp;
if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
 sp = s_sp;  /* control sobre el dispositivo especificado */
  switch (sp->devtype) {
  case SCSI_DEVDISK:
  case SCSI_DEVWORM:
  case SCSI_DEVCDROM:
  case SCSI_DEVOPTICAL:

        if (m_ptr->REQUEST == DIOCEJECT) {
                /* Expulsar disco */
                if (sp->open_ct > 1) return(EBUSY);

                /* Enviar un comando empezar/parar con código 2: parar y expulsar. */
                if (scsi_simple(SCSI_STRTSTP, 2) != SENSE_NO_SENSE)
                        return(EIO);
                return(OK);
        }
        /* Llamar al código común para discos y similares. */
        return(do_diocntl(dp, m_ptr));
```

```
  default:
        return(ENOTTY);
case SCSI_DEVTAPE:
        break;
  }
  /* A continuación control sobre cintas */

  if (m_ptr->REQUEST == MTIOCTOP) {
        struct mtop op;
        phys_bytes op_phys;
        long delta;
        int key;
        byte *buf = tmp_buf;

        /* Comandos básicos sobre cintas: rebobinar, espacio, escribir marca EOF ... */
        op_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
sizeof(op));
        if (op_phys == 0) return(EINVAL);
        phys_copy(op_phys, vir2phys(&op), (phys_bytes) sizeof(op));

        switch(op.mt_op) {
        case MTREW:
        case MTOFFL:
        case MTRETEN:
        case MTFSF:
        case MTFSR:
        case MTBSF:
        case MTBSR:
        case MTEOM:
/* Escriibir una marca EOF antes de spacing. */
                if (sp->need_eof && sp->tstat.mt_dsreg != DS_ERR) {
                        if (scsi_simple(SCSI_WREOF, 1) != SENSE_NO_SENSE)
                                return(EIO);
                        sp->tstat.mt_blkno = 0;
                        sp->tstat.mt_fileno++;
                        sp->need_eof = FALSE;
                        }
                sp->at_eof = FALSE;
        }
        switch(op.mt_op) {
        case MTREW:
        case MTOFFL:
        case MTRETEN:
        case MTERASE:
                /* Rebobinar, Fuera línea, tensionar, borrar. */
                switch(op.mt_op) {
                case MTOFFL:
                        if (scsi_simple(SCSI_LOADUNLD, 0) != SENSE_NO_SENSE)
                                return(EIO);
                        sp->state &= ~S_READY;
                        break;
                case MTRETEN:
                        if (scsi_simple(SCSI_LOADUNLD, 3) != SENSE_NO_SENSE)
                                return(EIO);
```

```
                break;
        case MTERASE:
                if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
                        return(EIO);
                if (scsi_simple(SCSI_ERASE, 1) != SENSE_NO_SENSE)
                        return(EIO);
                /* rebobinar una vez. */
                /*FALL THROUGH*/
        case MTREW:
                if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
                        return(EIO);
        }
        sp->tstat.mt_dsreg = DS_OK;
        sp->tstat.mt_blkno = 0;
        sp->tstat.mt_fileno = 0;
        break;
case MTFSF:
case MTFSR:
case MTBSF:
case MTBSR:
        if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);
        group0();
        rq->ccb.opcode = CCB_INIT;
        ccb_cmd0(rq).scsi_op = SCSI_SPACE;
        delta = op.mt_count;

        if (op.mt_op == MTBSR) delta = -delta;
        if (op.mt_op == MTBSF) delta = -delta - 1;
        h2b24(ccb_cmd0(rq).trlength, delta);
        ccb_cmd0(rq).fixed =
                        op.mt_op == MTFSR || op.mt_op == MTBSR ? 0 : 1;

if ((key = scsi_command(0L, 0)) != SENSE_NO_SENSE) {
        if (sense_key(key) != SENSE_NO_SENSE) return(EIO);
        if (sense_eom(key)) {
                /* Estamos en final de cinta. */
                if (op.mt_op == MTBSF || op.mt_op == MTBSR) {
                        /* Retroceder al principio de la cinta. */
                        sp->tstat.mt_dsreg = DS_EOF;
                        sp->tstat.mt_blkno = 0;
                        sp->tstat.mt_fileno = 0;
                } else {
                        /* No podemos avanzar */
                        return(EIO);
                }
        }
        if (sense_eof(key)) {
                /* Llegamos a una marca de fichero. */
                sp->tstat.mt_dsreg = DS_EOF;
                sp->at_eof = TRUE;
                if (op.mt_op == MTFSR) {
                        /* Avanzar */
                        sp->tstat.mt_blkno = 0;
                        sp->tstat.mt_fileno++;
                } else {
```

```
                        /* Retroceder (mala idea!) */
                        sp->tstat.mt_blkno = -1;
                        sp->tstat.mt_fileno--;
                }
        }

} else {
                if (op.mt_op == MTFSR || op.mt_op == MTBSR) {
                        sp->tstat.mt_blkno += delta;
                } else {
                        sp->tstat.mt_blkno = 0;
                        sp->tstat.mt_fileno += delta;}
                if (op.mt_op == MTBSF) {
                        /* n+1 retrocesos, y 1 avance. */
                        group0();
                        rq->ccb.opcode = CCB_INIT;
                        ccb_cmd0(rq).scsi_op = SCSI_SPACE;
                        h2b24(ccb_cmd0(rq).trlength, 1L);
                        ccb_cmd0(rq).fixed = 1;


                        if (scsi_command(0L, 0) != SENSE_NO_SENSE)
                                return(EIO);
                        sp->tstat.mt_fileno++;
                }
                sp->tstat.mt_dsreg = DS_OK;
        }
        break;
case MTWEOF:
        /* escribir EOF. */
        if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);
                if (op.mt_count < 0) return(EIO);
        if (op.mt_count == 0) return(OK);
        group0();
        rq->ccb.opcode = CCB_INIT;
        ccb_cmd0(rq).scsi_op = SCSI_WREOF;
        h2b24(ccb_cmd0(rq).trlength, op.mt_count);
        if (scsi_command(0L, 0) != SENSE_NO_SENSE) return(EIO);
        sp->tstat.mt_dsreg = DS_OK;
        sp->tstat.mt_blkno = 0;
        sp->tstat.mt_fileno += op.mt_count;
        sp->need_eof = FALSE;
        break;

case MTEOM:
        /* Avanzar espacio para termino de medio. */
        if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);

        do {
                group0();
                rq->ccb.opcode = CCB_INIT;
                ccb_cmd0(rq).scsi_op = SCSI_SPACE;
                h2b24(ccb_cmd0(rq).trlength, 0x7FFFFF);
                ccb_cmd0(rq).fixed = 1;
                key = scsi_command(0L, 0);
```

```
                        sp->tstat.mt_blkno = 0;
                        sp->tstat.mt_fileno += 0x7FFFFF;
                        if (key != SENSE_NO_SENSE) {
                                if (key != SENSE_BLANK_CHECK) return(EIO);
                                sp->tstat.mt_fileno -= sp->tstat.mt_resid;
                        }
                } while (key == SENSE_NO_SENSE);
                sp->tstat.mt_dsreg = DS_OK;
                break;
        case MTBLKZ:
        case MTMODE:
                /* Seleccionar tamaño bloque cinta o densidad cinta. */

                /* Rebobinar. */
                if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
                        return(EIO);
                sp->tstat.mt_dsreg = DS_OK;
                sp->tstat.mt_blkno = 0;
                sp->tstat.mt_fileno = 0;
                if (op.mt_op == MTBLKZ && op.mt_count == 0) {
                        /* Petición para modo tamaño de bloque variable. */
                        sp->tfixed = FALSE;
                        sp->block_size = 1;
                } else {
                /* Primero coger los valores actuales. */
                        if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE)
                                return(EIO);

                        /* Debe tener al menos un bloque descriptor. */
                        if (buf[3] < 8) return(EIO);
                        buf[0] = 0;
                        buf[1] = 0;
                        /* buf[2]: almacena mode y velocidad */
                        buf[3] = 8;
                        if (op.mt_op == MTMODE)     /* Nueva densidad */
                                buf[4 + 0] = op.mt_count;
                        /* buf[4 + 1]: número de bloques */
                        buf[4 + 4] = 0;
                        if (op.mt_op == MTBLKZ)      /* Nuevo tamaño de bloque */
                                h2b24(buf + 4 + 5, (long) op.mt_count);

                        /* Establecer nueva densidad/tamaño bloque. */
                        if (scsi_simple(SCSI_MDSELECT, 4+8) != SENSE_NO_SENSE)
                                return(EIO);

                        if (op.mt_op == MTBLKZ) {
                                sp->tfixed = TRUE;
                                sp->block_size= op.mt_count;
                        }
                }
if (m_ptr->REQUEST == MTIOCGET) {
        /* Peticion estado cinta. */
        phys_bytes get_phys;

        get_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
```

```
                                                    sizeof(sp->tstat));
        if (get_phys == 0) return(EINVAL);

        if (sp->tstat.mt_dsreg == DS_OK) {
                /* Un error de datos viejo nunca es limpiado (por ahora). */
                sp->tstat.mt_erreg = 0;
                sp->tstat.mt_resid = 0;
        }
        phys_copy(vir2phys(&sp->tstat), get_phys,(phys_bytes) sizeof(sp->tstat));
  } else {
        /* No implementado. */
        return(ENOTTY);
  }
  return(OK);
}
```

3.15. Scsi_simple

Prepara un comando para ser interpretado por el procedimiento 'scsi-command'
En el campo 'opcode' del struct que define la petición ('rq'), ponemos el comando CCB_INIT, que es el comando inicializador.
A partir de entonces, y en función de los parámetros opcode y count, vamos configurando el bloque descriptor de comandos.
Al final, se llama al procedimiento 'scsi-command', que se encarga de llevar a cabo la petición.

```
PRIVATE int scsi_simple(opcode, count)
int opcode;                             /* SCSI código operación */
int count;                                      /* contador o  flag */
{
/* The average comando grupo 0 SCSI  con una simple flag o contador. */

  vir_bytes len = 0;                            /* A veces se usa un buffer. */

  group0();
  rq->ccb.opcode = CCB_INIT;
  ccb_cmd0(rq).scsi_op = opcode;

  /* Rellena el argumento count en le lugar adecuado. */
  switch (opcode) {
  case SCSI_REQSENSE:
  case SCSI_INQUIRY:
  case SCSI_MDSENSE:
  case SCSI_MDSELECT:
        ccb_cmd0(rq).nblocks = count;
        len = count;
        break;
case SCSI_STRTSTP:
   /* SCSI_LOADUNLD: (synonym) */
        ccb_cmd0(rq).nblocks = count;
        break;
```

```
  case SCSI_RDLIMITS:
        len = count;
        break;

  case SCSI_WREOF:
        h2b24(ccb_cmd0(rq).trlength, (long) count);
        break;

  case SCSI_REWIND:
  case SCSI_ERASE:
        ccb_cmd0(rq).fixed = count;
        break;
  }
  return(scsi_command(tmp_phys, len));
}
```

3.16. Scsi_command.

Ejecuta comandos SCSI y retorna el resultado de dicha ejecución.
Copia el comando a ejecutar a una estructura de request, y usando la función out_byte inicia la ejecución del comando.
A traves de la estructura mailbox (se reliza intercambio de mensajes), se realiza la comunicación, y cuando el comando se ha ejecutado, se realiza la verificación del resultado de la misma, devolviendo el resultado para que el procedimiento que realizó la llamada interprete que sucedió (el valor SENSE_NO_SENSE indica que la operación se realizó con éxito).

```
PRIVATE int scsi_command(data, len)
phys_bytes data;
vir_bytes len;
{
/* Ejecuta un comando SCSI y retorna el resultado. Esta rutina retorna sense key de
un comando SCSI como OK or EIO.
 */
  struct scsi *sp = s_sp;
  int key;
  message intr_mess;

  rq->ccb.addrcntl = ccb_scid(s_sp->targ) | ccb_lun(s_sp->lun);

  if (rq->ccb.opcode == CCB_SCATTER) {
        /* dispositivo lee/escribe; añade comprobación y usa scatter/gather vector. */
        rq->ccb.addrcntl |= s_opcode == DEV_READ ? CCB_INCHECK :
CCB_OUTCHECK;
        data = vir2phys(rq->dmalist);
        len = (byte *) rq->dmaptr - (byte *) rq->dmalist;
        if (aha_model == AHA1540) {
              /* A plain 1540 can't do s/g. */
              rq->ccb.opcode = CCB_INIT;
              data = b2h24(rq->dmalist[0].dataptr);
              len = b2h24(rq->dmalist[0].datalen);
        }
```

```
 }
h2b24(rq->ccb.datalen, (u32_t) len);
 h2b24(rq->ccb.dataptr, data);
 dump_scsi_cmd();

  mailbox[0].status = AHA_MBOXSTART;

  out_byte(AHA_DATAREG, AHACOM_STARTSCSI);  /* hey, tienes un mensajel! */

  /* Espera hasta que se complete el comando. */
  while (mailbox[1].status == AHA_MBOXFREE) {
          /* no hay mensaje, esperar por una interrupción. */
          receive(HARDWARE, &intr_mess);
  }
  mailbox[1].status = AHA_MBOXFREE;        /* liberar recepción de mensaje */

  /* Comprobar los resultados de la operación. */
  if (rq->ccb.hastat != 0) {
          /* error estado host adapter  */
          printf("%s: host adapter error 0x%02x%s\n", s_name(), rq->ccb.hastat,
                  rq->ccb.hastat == HST_TIMEOUT ? " (Selection timeout)" : "");
                  errordump();
          if (sp->devtype == SCSI_DEVTAPE) sp->tstat.mt_dsreg = DS_ERR;
          memset((void *) &ccb_sense(rq), 0, sizeof(sense_t));
          return(SENSE_HARDWARE);
  }

  if (rq->ccb.tarstat != 0) {
          /*  ocurrio un error SCSI . */
          sense_t *sense = &ccb_sense(rq);
          if (sense->len < 2) {
                  /* No código y calificador añadido, entonces 0. */
                  sense->add_code = sense->add_qual = 0;
          }
          /* Comprobar datos estado, reporta error si es interesante. */
          if (rq->ccb.tarstat == TST_CHECK) {
                  if ((sense->errc & 0x7E) == 0x70) {
                          /* error SCSI estandar. */
                          key = sense->key;
                  } else {
                          /* requerir al vendedor en otro caso. */
                          key = SENSE_VENDOR;
                  }
          } else {
                  if (rq->ccb.tarstat == TST_LUNBUSY) {
                          /* La unidad esta demasiado ocupada para responder... */
                          key = SENSE_NOT_READY;
                  } else {
                          /* El controlador no puede hacer... */
                          key = SENSE_HARDWARE;
                  }
                  memset((void *) sense, 0, sizeof(sense_t));
          }
          if (sense_serious(sense_key(key))) {
                  /* Algo malo ocurrio. */
```

```
                printf("%s: error on command 0x%02x,", s_name(),rq->ccb.cmd[0]);
if (rq->ccb.tarstat != TST_CHECK) {
                        printf("target status 0x%02x\n", rq->ccb.tarstat);
                } else {
                     printf("sense key 0x%02x (%s), additional 0x%02x%02x\n",
                                sense->key,
                                str_scsi_sense[sense_key(key)],
                                sense->add_code, sense->add_qual);
                }
                errordump();
        }

if (sp->devtype == SCSI_DEVTAPE) {
                /* Almacena detalles del error de cinta. */
                sp->tstat.mt_dsreg = DS_ERR;
                sp->tstat.mt_erreg = key;
                sp->tstat.mt_resid = b2h32(sense->info);
        }

        /* Coger solo ILI, EOM y EOF bits de key 0. */
        if (sense_key(key) != SENSE_NO_SENSE) key = sense_key(key);

        return(key);
  }
  return(SENSE_NO_SENSE);
}
```

3.17. Aha_command.

Envía comandos de bajo nivel a la controladora.
Envía un comando a través de un puerto mediante la función out_byte y recibe el resultado de ese comando, el cual devuelve en una variable de salida.

```
PRIVATE void aha_command(outlen, outptr, inlen, inptr)
int outlen, inlen;
byte *outptr, *inptr;
{
  /* Enviar comando de bajo nivel a la controladora. */
  int i;

  /* Enviar bytes del comando. */
  for (i = 0; i < outlen; i++) {
        while (in_byte(AHA_STATREG) & AHA_CDF) {}       /* !! timeout */
        out_byte(AHA_DATAREG, *outptr++);
  }
  /* Recibir datos. */
  for (i = 0; i < inlen; i++) {
        while (!(in_byte(AHA_STATREG) & AHA_DF)
                && !(in_byte(AHA_INTRREG) & AHA_HACC)) {}  /* !! timeout */
        *inptr++ = in_byte(AHA_DATAREG);
  }

  /* Esperar a que se complete el comando. */
```

```
  while (!(in_byte(AHA_INTRREG) & AHA_HACC)) {}          /* !! timeout */
  out_byte(AHA_CNTLREG, AHA_IRST);      /* limpiar interrupción */
  if (aha_irq != 0) enable_irq(aha_irq);

  /* !! Deberia comprobar el registro de estado para un comando invalido*/
}
```

3.18. Aha_reset.

Inicializa los parámetros de comunicación con el dispositivo, así como la BIOS y la declaración de los TIMEOUTS.
Obtiene información de las variables de ambiente (registro base del dispositivo, bus, velocidad de transmisión).
Realiza un test, para comprobar errores en el dispositivo.
Obtiene información del tipo de controlador y de su configuración.
Inicializa las interfaces de comunicación, así como las interrupciones y prepara y actualiza los timeouts del dispositivo.

```
PRIVATE int aha_reset()
{
  int stat;
  int irq, bus_on, bus_off, tr_speed;
  unsigned sg_max;
  long v;
  static char aha0_env[] = "AHA0", aha_fmt[] = "x:d:d:x";
  byte cmd[5], haidata[4], getcdata[3], extbios[2];
  struct milli_state ms;

  /* Obtener información de configuración del entorno. */
  v = AHA_BASEREG;
  if (env_parse(aha0_env, aha_fmt, 0, &v, 0x000L, 0x3FFL) == EP_OFF) return 0;
  aha_basereg = v;

  v = 15;
  (void) env_parse(aha0_env, aha_fmt, 1, &v, 2L, 15L);
  bus_on = v;
  v = 1;
  (void) env_parse(aha0_env, aha_fmt, 2, &v, 1L, 64L);
  bus_off = v;
  v = 0x00;
  (void) env_parse(aha0_env, aha_fmt, 3, &v, 0x00L, 0xFFL);
  tr_speed = v;
/* Resetear controladorar, esperar a que se autocompruebe. */
  out_byte(AHA_CNTLREG, AHA_HRST);
  milli_start(&ms);
  while ((stat = in_byte(AHA_STATREG)) & AHA_STST) {
          if (milli_elapsed(&ms) >= AHA_TIMEOUT) {
                  printf("aha0: AHA154x controller not responding\n");
                  return(0);
          }
  }
  /* Comprobar fallos en autotest. */
  if ((stat & (AHA_DIAGF | AHA_INIT | AHA_IDLE | AHA_CDF | AHA_DF))
```

```
                                                != (AHA_INIT | AHA_IDLE)) {
        printf("aha0: AHA154x controller failed self-test\n");
        return(0);
  }

/* Obtener información sobre el tipo controladora y conhfiguración. */
  cmd[0] = AHACOM_HAINQUIRY;
  aha_command(1, cmd, 4, haidata);

  cmd[0] = AHACOM_GETCONFIG;
  aha_command(1, cmd, 3, getcdata);

  /* Los primeros bytes indican el tipo de placa. */
  aha_model = haidata[0];
/* Desbloquea la interface de mensajes1540C or 1540CF.  (Esto es para proteger
viejos drivers del adaptador si las caracteristicas extendidas están habilitadas)   */
  if (aha_model >= AHA1540C) {
        cmd[0] = AHACOM_EXTBIOS;        /* Obtiene información extendida de la
BIOS */
        aha_command(1, cmd, 2, extbios);
        if (extbios[1] != 0) {
                /* la interface de mensaje esta bloqueada, desbloquear. */
                cmd[0] = AHACOM_MBOX_ENABLE;
                cmd[1] = 0;             /* bit 0 = 0 (permitir mensaje) */
                cmd[2] = extbios[1];    /* código para desbloquear mensaje */
                aha_command(3, cmd, 0, 0);
        }
  }

/* La longitud máxima de la lista scatter/gather DMA depende del modelo de placa. */
  sg_max = 16;
  if (aha_model == AHA1540) sg_max = 1;             /* 1540 has no s/g */
  if (aha_model >= AHA1540C) sg_max = 255;          /* 1540C has plenty */
 /* Levantar el canal DMA. */
  switch (getcdata[0]) {
  case 0x80:              /* canal 7 */
        out_byte(0xD6, 0xC3);
        out_byte(0xD4, 0x03);
        break;

case 0x40:              /* canal 6 */
        out_byte(0xD6, 0xC2);
        out_byte(0xD4, 0x02);
        break;
  case 0x20:            /* canal 5 */
        out_byte(0xD6, 0xC1);
        out_byte(0xD4, 0x01);
        break;
case 0x01:              /* canal 0 */
        out_byte(0x0B, 0x0C);
        out_byte(0x0A, 0x00);
        break;
  default:
        printf("aha0: AHA154x: strange DMA channel\n");
        return(0);
```

```
  }

  /* Conseguir información de IRQ. */
  switch (getcdata[1]) {
  case 0x40:    irq = 15;         break;
  case 0x20:    irq = 14;         break;
  case 0x08:    irq = 12;         break;
  case 0x04:    irq = 11;         break;
  case 0x02:    irq = 10;         break;
  case 0x01:    irq =  9;         break;
default:
        printf("aha0: strange IRQ setting\n");
        return(0);
  }

  /* Permitir interrupciones en irq obtenida. */
  put_irq_handler(irq, s_handler);
  aha_irq = irq;
  enable_irq(irq);

  /* Inicializar los datos de peticiones: Bloque Control Comandos, buzones.
   * (Queremos tener los buzones inicializados, porque la 1540C quiere conocerlo.)*/
/* Iniciar ccb. */
  rq->ccb.senselen = CCB_SENSEREQ;      /* siempre quiere información sense */
  h2b24(rq->ccb.linkptr, 0L);           /* nunca enlazar comandos */
  rq->ccb.linkid = 0;
  rq->ccb.reserved[0] = 0;
  rq->ccb.reserved[1] = 0;

  /* Scatter/gather máximo. */
  rq->dmalimit = rq->dmalist + (sg_max < NR_IOREQS ? sg_max : NR_IOREQS);

  /* Liberar buzon. */
  mailbox[0].status = AHA_MBOXFREE;
  h2b24(mailbox[0].ccbptr, vir2phys(&rq->ccb));
/* buzon de llegada. */
  mailbox[1].status = AHA_MBOXFREE;
  /* mailbox[1].ccbptr rellenado por la controladora después de un comando. */

  /* Decir a la controladora donde estan los buzones y cuantos. */
  cmd[0] = AHACOM_INITBOX;
  cmd[1] = 1;
  h2b24(cmd + 2, vir2phys(mailbox));
  aha_command(5, cmd, 0, 0);


  /* Establecer bus on, bus off y velocidad de transferencia. */
  cmd[0] = AHACOM_BUSON;
  cmd[1] = bus_on;
  aha_command(2, cmd, 0, 0);

cmd[0] = AHACOM_BUSOFF;
  cmd[1] = bus_off;
  aha_command(2, cmd, 0, 0);
  cmd[0] = AHACOM_SPEED;
```

```
  cmd[1] = tr_speed;
  aha_command(2, cmd, 0, 0);


/* Establecer SCSI timeout. */
  cmd[0] = AHACOM_SETIMEOUT;
  cmd[1] = SCSI_TIMEOUT != 0;               /* timeouts si/no */
  cmd[2] = 0;                           /* reservado */
  cmd[3] = SCSI_TIMEOUT / 256;              /* MSB */
  cmd[4] = SCSI_TIMEOUT % 256;              /* LSB */
  aha_command(5, cmd, 0, 0);
  return(1);
}
```

3.19. Show_req.

Muestra las peticiones pendientes de resolver por pantalla. En modo depuración.

```
#if AHA_DEBUG & 4
PRIVATE void show_req()
{
  struct iorequest_s **iopp;
  dma_t *dmap;
  unsigned count, nbytes, len;

  iopp = rq->iov;
  dmap = rq->dmalist;
  count = rq->count;
  nbytes = 0;
  printf("%lu:%u", rq->pos, count);
  while (count > 0) {
        if (iopp == rq->iov || *iopp != iopp[-1])
              nbytes = (*iopp)->io_nbytes;
        printf(" (%u,%lx,%u)", nbytes, b2h24(dmap->dataptr),
                                          len = b2h24(dmap->datalen));
        dmap++;
        iopp++;
        count -= len;
        nbytes -= len;
  }
  if (nbytes > 0) printf(" ...(%u)", nbytes);
  printf("\n");
}
#endif /* AHA_DEBUG & 4 */
```

3.20. Dump_scsi_cmd.

Imprime el bloque descriptor de comando. En modo depuración.

```
#if AHA_DEBUG & 8

PRIVATE void dump_scsi_cmd()
{
```

```
  int i;

  printf("scsi cmd:");
  for (i = 0; i < rq->ccb.cmdlen; i++) printf(" %02x", rq->ccb.cmd[i]);
  printf("\n");
}
#endif /* AHA_DEBUG & 8 */
```

3.21. S_geometry.

Suministra información necesaria sobre el driver a Fdisk.
Es necesario tener el número de cilindros, cabezas y sectores de cada dispositivo como entradas en la tabla de particiones. Esta función se encarga de ello.

```
PRIVATE void s_geometry(entry)
struct partition *entry;
{
/* La geometría de un dispositivo SCSI es inutil, el controlador BIOS en la placa
 * makes the drive look like a regular drive on the outside.  Un programa DOS
 * coge la dirección lógica de un bloque,calcula cilindro, cabeza y sector cómo la
 * interrupción 0x13 de la BIOS espera que la llamen, y el Controlador la vuelve a
 *  convertir en una dirección de bloque otra vez.
 * La única razon que tenemos para hacer esto, es que muchos idiotas ponen
 * cilindros, cabezas y número de sectores en la Tabla de Particiones, entonces fdisk
 * necesita conocer esta geometría.*/
  unsigned long size = s_sp->part[0].dv_size;
  unsigned heads, sectors;

  if (size < 1024L * 64 * 32 * 512) {
          /* dispositivo pequeño. */
          heads = 64;
          sectors = 32;
  } else {
          /* Asume que la BIOS está configurada para dispositivos grandes. */
          heads = 255;
          sectors = 63;
  }
  entry->cylinders = (size >> SECTOR_SHIFT) / (heads * sectors);
  entry->heads = heads;
  entry->sectors = sectors;
}
```

3.22. Errordump.

Imprime el bloque de control de comando. En modo depuración.

```
#if AHA_DEBUG & 2
PRIVATE void errordump()
{
  int i;
  printf("aha ccb dump:");
  for (i = 0; i < sizeof(rq->ccb); i++) {
```

```
            if (i % 26 == 0) printf("\n");
            printf(" %02x", ((byte *) &rq->ccb)[i]);
  }
  printf("\n");
}
#endif /* AHA_DEBUG & 2 */
```

3.23. Otras Funciones.

- Funciones que preparan la estructura ccb según el tipo de dispositivo:

```
PRIVATE void group0()
{
  /* Prepara la estructura ccb para un grupo 0 de comando SCSI. */

  rq->ccb.cmdlen = sizeof(cdb0_t);

  /* Clear cdb to zeros the ugly way. */
  * (u32_t *) (rq->ccb.cmd + 0) = 0;
  * (u16_t *) (rq->ccb.cmd + 4) = 0;
}


PRIVATE void group1()
{
  rq->ccb.cmdlen = sizeof(cdb1_t);
  * (u32_t *) (rq->ccb.cmd + 0) = 0;
  * (u32_t *) (rq->ccb.cmd + 4) = 0;
  * (u16_t *) (rq->ccb.cmd + 8) = 0;

}
```

- Funciones de conversión:

```
PRIVATE void h2b16(b, h)
big16 b;
U16_t h;
{
/* Host byte order to Big Endian conversion. */
  b[0] = h >> 8;
  b[1] = h >> 0;
}


PRIVATE void h2b24(b, h)
big24 b;
u32_t h;
{
  b[0] = h >> 16;
  b[1] = h >>  8;
  b[2] = h >>  0;
}

PRIVATE void h2b32(b, h)
big32 b;
```

```
u32_t h;
{
  b[0] = h >> 24;
  b[1] = h >> 16;
  b[2] = h >>  8;
  b[3] = h >>  0;
}

PRIVATE u16_t b2h16(b)
big16 b;
{
  return  ((u16_t) b[0] << 8)
          | ((u16_t) b[1] << 0);
}

PRIVATE u32_t b2h24(b)
big24 b;
{
  return  ((u32_t) b[0] << 16)
          | ((u32_t) b[1] <<  8)
          | ((u32_t) b[2] <<  0);
}

PRIVATE u32_t b2h32(b)
big32 b;
{
  return  ((u32_t) b[0] << 24)
          | ((u32_t) b[1] << 16)
          | ((u32_t) b[2] <<  8)
          | ((u32_t) b[3] <<  0);
}
```