El disco duro en Minix

Antonio Ossian Buitrago Ekvall
David Brito Melado

Tabla de Contenidos

El disco duro en Minix 2.0

El manejador del disco duro es la parte del Minix que tiene que ver con un amplio rango de diferentes tipos de hardware. Antes de discutir detalles sobre el manejador, consideremos brevemente algunos de los problemas que las diferencias de hardware pueden causar. El IBM-PC es una familia de diferentes computadores; no sólo se trata de diferentes procesadores sino que hay algunas diferencias en el hardware básico. Los primeros miembros de la familia usaban un bus de 8 bits apropiado para el interfaz externo del procesador 8088. La siguiente generación, PC AT, uso un bus de 16 bits diseñado de forma que los antiguos periféricos de 8 bits pudieran ser usados. Los periféricos de 16 bits generalmente no pueden ser usados en sistemas PC XT. El bus del AT fue diseñado para sistemas con el procesador 80286 y es usado por sistemas basados en el 80386, 80486 y Pentium. Sin embargo, dado que estos procesadores tienen un interfaz de 32 bits, ahora hay varios buses de 32 bits en el mercado, tales como el bus PCI de Intel.

Para cada bus hay una familia diferente de adaptadores de E/S, los cuales se conectan a la placa base del sistema. Todos los periféricos pertenecientes a un diseño particular de bus deben ser compatibles con el estándar de ese bus. En la familia IBM PC, cada bus viene también con firmware en su BIOS que es diseñado para salvar la diferencia entre el S.O. y las peculiaridades del hardware. Algunos periféricos pueden incluso traer extensiones de la BIOS en chips ROM. La dificultad a la que hace frente un diseñador de sistemas operativos es que el BIOS de los PC fue diseñado para un S.O., MS-DOS, que no soporta multiprogramación y corre en modo real de 16 bits.

El diseñador de un nuevo SO para el PC se encuentra ante varias opciones. Una es si usar el manejador que se encuentra en el BIOS, o escribir uno nuevo. Esto no era una elección difícil en el diseño original de Minix, ya que el BIOS no se adecuaba en muchos aspectos a las necesidades del Minix. Por supuesto, a la hora de empezar el programa de arranque del Minix usa el BIOS para hacer la carga inicial del sistema – ya sea del disco duro o del floppy – pues no hay otra alternativa.

La segunda opción nos hace considerar que hay al menos cuatro tipos diferentes de controladoras de disco duro: el original de los XT, el de los AT y dos diferentes de los PS/2. Hay varias maneras de tratar este problema:

1. Compilar una versión única del sistema para cada tipo de controladora.
2. Proporcionar varios manejadores en el Kernel y hacer que, en el arranque, éste determine cual usar.
3. Proporcionar varios manejadores en el Kernel y hacer que, de alguna manera, el usuario determine cual usar.

En el primer caso, haría falta distribuir cuatro discos diferentes de arranque (uno por controladora). Esto resultaría para un proveedor muy caro, por lo que la primera opción no es muy recomendable. El segundo caso haría necesario que el sistema verificara los dispositivos de E/S conectados, complicando el código de arranque y añadiendo el problema de que puedan existir periféricos no estándar. Así que la segunda opción es también algo arriesgada.

El tercer método, que es el que usa Minix, es permitir la compilación de varios manejadores, habiendo uno por defecto. El programa de arranque permite introducir varios parámetros en la inicialización. Hay otras dos estrategias que Minix aplica para minimizar los problemas con los múltiples manejadores existentes. Una es que, después de todo, existe un manejador que hace de interfaz entre el Minix y el BIOS.

Este manejador funciona con garantías en cualquier sistema y puede ser elegido como parámetro de inicialización (hd=bios). Generalmente este debería ser el último recurso, ya que el Minix corre en modo protegido pero el código del BIOS en modo real, y esta conmutación de modos es muy lenta.

La segunda estrategia es que el Minix pospone la inicialización de los manejadores hasta el último momento. Así, si ninguno de los manejadores del disco duro funciona podemos seguir usando el floppy y no habrá problemas hasta que no se intente usar el disco duro. Las primeras versiones de Minix intentaban inicializar el disco duro nada más arrancar el sistema y si no había disco duro el sistema se colgaba.

En este tema, tomaremos como modelo el manejador de disco AT, que es el manejador por defecto de la distribución estándar de Minix. Este es un manejador versátil que trata con controladoras que van desde las primeras de los 80286, hasta las modernas EIDE que manejan 1 Gbyte de capacidad.

El bucle principal de la tarea de disco duro usa el mismo código que ya hemos visto en temas anteriores y se pueden hacer los seis tipos de peticiones estándar. Una petición DEV_OPEN, por ejemplo, puede implicar gran cantidad de trabajo, pues puede haber particiones o subparticiones en el disco duro. Algunas controladoras de disco duro también soportan CD-ROM, cuya presencia no obligatoriamente implica la existencia de un disco en su interior. Así que el DEV_OPEN debe verificar esta circunstancia. Por otro lado, en un CD-ROM la petición DEV_CLOSE debería verificar que la puerta no esté atascada, etc. Esta y otras complicaciones de los medios extraibles no se verán en este tema.

wini.c

El fichero *wini.c* tiene la tarea del ocultar el manejador de disco duro presente al resto del kernel. Esto nos permite seguir el tercer método comentado en la introducción, reuniendo varios manejadores de disco duro en una única imagen del kernel, y seleccionando el que queremos usar durante el arranque del sistema. Posteriormente, la instalación ya personalizada, será compilada con sólo el manejador realmente necesitado.

Para implementar todo esto, *wini.c* contiene una definición de estructura de datos, *hdmap*, que es un vector que asocia un nombre con la dirección de una función. Este vector es inicializado por el compilador con tantos elementos como se necesite para el número de manejadores de disco duro habilitados en *include/minix/config.h* por el usuario. El vector es usado por la función *winchester_task*, que es una de las entradas de la tabla de tareas *task_tab* utilizada en el arranque. Cuando se llama a *winchester_task* intenta encontrar una variable del entorno denominada *hd*, mediante una función del kernel que se ocupa de leer el entorno creado por el monitor de arranque del MINIX. Si hay un valor definido para *hd* se busca en el vector *hdmap;* de lo contrario se usa la primera entrada del mismo, llamando a la función correspondiente que en la distribución estándar de MINIX es *at_winchester_task*. Esta función es el punto de entrada al manejador de dico duro AT que se encuentra en el fichero *at_wini.c* que tratamos en el siguiente apartado.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                  src/kernel/wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* De los diferentes manejadores que pueden ser compilados, sólo uno puede
   correr. Este es elegido usando la variable de entorno ‘hd’ */

#include "kernel.h"
#include "driver.h"
#if ENABLE_WINI

/* Mapear nombre de manejador con tarea */
struct hdmap {
char *name;
task_t *task;
} hdmap[] = {

#if ENABLE_AT_WINI
      { "at", at_winchester_task },
#endif
#if ENABLE_BIOS_WINI
      { "bios", bios_winchester_task },
#endif
#if ENABLE_ESDI_WINI
      { "esdi", esdi_winchester_task },
#endif
#if ENABLE_XT_WINI
      { "xt", xt_winchester_task },
#endif
};

/* WINCHESTER_TASK ********************************************/
/* Llamar a la tarea winchester por defecto o la seleccionada */
/**************************************************************/
PUBLIC void winchester_task()
{
      char *hd;
      struct hdmap *map;
      hd = k_getenv("hd");
      for (map = hdmap; map < hdmap + sizeof(hdmap)/sizeof(hdmap[0]); map++) {
            if (hd == NULL || strcmp(hd, map->name) == 0) {
                  /*Ejecutar la tarea winchester seleccionada */
                  (*map->task)();}
            }
            panic("no hd driver", NO_NUM);
      }
#endif /* ENABLE_WINI */
```

at_wini.c

En este fichero se encuentran definidas las estructuras de datos, macros que especifican registros de la controladora, bits de estado y comandos, y prototipos, además de las funciones que componen el manejador.

Cabe destacar la estructura **WINI** que contiene diversos parámetros relacionados con el disco duro, habiendo una entrada por cada dispositivo:

State	Estado del dispositivo
Base	Registro base del fichero de registros
Irq	Línea de petición de interrupción
Lcylinders	Número lógico de cilindros (BIOS)
Lheads	Número lógico de cabezas
Lsectors	Número lógico de sectores por pista
Pcylinders	Número físico de cilindros
Pheads	Número físico de cabezas
Psectors	Número físico de sectores por pista
Ldhpref	Cuatro bytes superiores del registro LDH
Precomp	Cilindro de precompensación de escritura / 4
Max_count	Máximo número de bytes a transferir
Open_ct	Contador de uso
Part[DEV_PER_DRIVE]	Particiones primarias: hd[0-4]
Subpart[SUB_PER_DRIVE]	Subparticiones: hd[1-4][a-d]

Otra estructura de interés es la **COMMAND** que se utiliza para mandar comandos a la controladora . Seguidamente se muestran sus campos:

Precomp	Comienzo de precompensación de escritura.
Count	Sectores a transferir.
Sector	Número de sector de comienzo
Cyl_lo	Byte inferior del número de cilindro.
Cyl_hi	Byte superior del número de cilindro.
Ldh	LBA, unidad y cabeza.
Command	Tipo de comando para la controladora.

También es de interés la estructura **WTRANS** que almacena las peticiones de transferencia que van a ser traducidas a comandos:

Iop	Puntero a peticiones de E/S
Block	Primer sector a transferir.
Count	Contador de bytes.
Phys	Dirección física de usuario

Así mismo destacaremos la estructura **W_DTAB** que contiene las direcciones de las funciones que realizan las diversas tareas. Un puntero a esta, es enviado a Driver_task para que llame estas a la hora de atender las peticiones.

W_name	Nombre de la unidad actual.
W_do_open	Petición de abrir o montar,inicializa device.
W_do_close	Liberar dispositivo.
Do_diocntl	Obtener /fijar la geometría de una partición.
W_prepare	Preparar para E/S un dispositivo menor .
W_schedule	Planifica transferencias de entrada/salida.
W_finish	Realizar la entrada/salida.
Nop_cleanup	No se necesita operación de cleanup.
W_geometry	Informar sobre la geometría del disco.

Finalmente enumeraremos las variables globales más importantes que manejaremos a lo largo del listado:

***w_tp :** puntero al vector wtrans de peticiones de transferencia.
W_count: número de bytes a transferir.
W_nextblock: próximo bloque en disco a transferir.

W_opcode:	operación DEV_READ o DEV_WRITE.
W_command:	comando actual en ejecución.
W_status:	estado después de una interrupción.
W_drive:	unidad seleccionada.
***w_dv:**	puntero a estructura device(base y tamaño del dispositivo).

A continuación se muestra el listado de la parte de definición de variables y estructuras:

```
/* Este fichero contiene la parte dependiente del dispositivo de un manejador
   para la controladora winchester para IBM-AT. El fichero tiene un punto de
   entrada at_winchester_task: entrada principal cuando el sistema arranca */

#include "kernel.h"
#include "driver.h"
#include "drvlib.h"

#if ENABLE_AT_WINI
/* Puertos de E/S usados por controladoras winchester */

/* Registros de L/E */
#define REG_BASE0       0x1F0  /* registro base de la controladora 0 */
#define REG_BASE1       0x170  /* registro base de la controladora 1 */
#define REG_DATA        0      /* registro de datos (offset dede el reg. base */
#define REG_PRECOMP     1      /* comienzo de la precompensación de escritura */
#define REG_COUNT       2      /* sectores a transferir */
#define REG_SECTOR      3      /* numero de sector */
#define REG_CYL_LOW     4      /* byte inferior del numero de cilindro */
#define REG_CYL_HI      5      /* byte superior del numero de cilindro */
#define REG_LDH         6      /* lba, dispositivo y cabeza */
#define LDH_DEFAULT     0xA0   /* ECC habilitado, 512 bytes por sector */
#define LDH_LBA         0x40   /* Usar direccionamiento LBA */
#define ldh_init(drive) (LDH_DEFAULT | ((drive) << 4))

/* Registros de solo lectura */
#define REG_STATUS      7     /* registro de estado */
#define STATUS_BSY      0x80  /* controladora ocupada */
#define STATUS_RDY      0x40  /* dispositivo listo */
#define STATUS_WF       0x20  /* fallo de escritura */
#define STATUS_SC       0x10  /* búsqueda completada (obsoleto) */
#define STATUS_DRQ      0x08  /* datos corregidos */
#define STATUS_IDX      0x02  /* pulso guía */
#define STATUS_ERR      0x01  /* error */
#define REG_ERROR       1     /* código de error */
#define ERROR_BB        0x80  /* bloque defectuoso */
#define ERROR_ECC       0x40  /* bytes del ECC incorrectos */
#define ERROR_ID        0x10  /* id no encontrado */
#define ERROR_AC        0x04  /* comando abortado */
#define ERROR_TK        0x02  /* error en pista cero */
#define ERROR_DM        0x01  /* no hay marca de dir. de datos */

/* Registros de solo escritura */
#define REG_COMMAND           7     /* comando */
#define CMD_IDLE              0x00  /* para w_command: dispositivo ocioso */
#define CMD_RECALIBRATE       0x10  /* recalibrar dispositivo */
#define CMD_READ              0x20  /* lectura de datos */
#define CMD_WRITE             0x30  /* escritura de datos */
#define CMD_READVERIFY        0x40  /* verificación de lectura */
#define CMD_FORMAT            0x50  /* formatear pista */
#define CMD_SEEK              0x70  /* buscar cilindro */
#define CMD_DIAG              0x90  /* ejecutar diagnostico de dispositivo*/
#define CMD_SPECIFY           0x91  /* especificar parámetros */
#define ATA_IDENTIFY          0xEC  /* identificar dispositivo */
#define REG_CTL               0x206 /* registro de control */
#define CTL_NORETRY           0x80  /* deshabilitar reintento de acceso */
```

```
#define CTL_NOECC       0x40  /* deshabilitar reintento de ECC */
#define CTL_EIGHTHEADS  0x08  /* más de ocho cabezas */
#define CTL_RESET       0x04  /* resetear controladora */
#define CTL_INTDISABLE  0x02  /* deshabilitar interrupciones */

/* Líneas de petición de interrupción */
#define AT_IRQ0         14    /* nº interrupción para controladora 0 */
#define AT_IRQ1         15    /* nº interrupción para controladora 1 */

/* Bloque común de orden */
struct command {
u8_t   precomp;   /* REG_PRECOMP, etc. */
u8_t   count;
u8_t   sector;
u8_t   cyl_lo;
u8_t   cyl_hi;
u8_t   ldh;
u8_t   command;
};

/* Códigos de error */
#define ERR             (-1)  /* error general */
#define ERR_BAD_SECTOR  (-2)  /* detectado bloque marcado como erróneo */

/* Algunas controladoras no interrumpen, nos avisa el reloj */
#define WAKEUP            (32*HZ)  /* el dispositivo puede estar 31 segs */

/* Varios */
#define MAX_DRIVES      4     /* este manejador soporta 4 dispositivos */

#if _WORD_SIZE > 2
      #define MAX_SECS 256    /* el controlador puede transferir estos */
#else
      #define MAX_SECS 127    /* sectores pero no a un proceso de 16 bits */
#endif

#define MAX_ERRORS      4     /* veces para intentar L/E antes de abandonar */
#define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
#define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
#define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
#define TIMEOUT         32000       /* timeout de controladora en ms */
#define RECOVERYTIME    500         /* tiempo de recuperación de controladora */
#define INITIALIZED     0x01        /* el dispositivo está inicializado */
#define DEAF            0x02        /* la controladora debe ser reseteada */
#define SMART           0x04        /* el manejador soporta comandos ATA */

/* Variables */
PRIVATE struct wini {   /* estructura ppal del manejador,1 entrada por device)*/
 unsigned state;        /* estado dispositivo: sordo,inicializado,muerto */
 unsigned base;         /* registro base del banco de registros */
 unsigned irq;          /* línea de petición de interrupción */
 unsigned lcylinders;   /* número lógico de cilindros (BIOS) */
 unsigned lheads;       /* número lógico de cabezas */
 unsigned lsectors;     /* número lógico de sectores por pista */
 unsigned pcylinders;   /* número físico de sectores (traducido) */
 unsigned pheads;       /* número físico de cabezas */
 unsigned psectors;     /* número físico de sectores por pista */
 unsigned ldhpref;      /* cuatro bytes superiores del registro LDH */
 unsigned precomp;      /* cilindro de precompensación de escritura / 4 */
 unsigned max_count;    /* máximo nº de peticiones para este dispositivo */
 unsigned open_ct;      /* contador en uso */
 struct device part[DEV_PER_DRIVE];/* particiones primarias: hd[0-4] */
 struct device subpart[SUB_PER_DRIVE]; /* subparticiones: hd[1-4][a-d] */
} wini[MAX_DRIVES], *w_wn;
```

```
PRIVATE struct trans {
 struct iorequest_s *iop;          /* pertenece a esta petición de E/S */
 unsigned long block;              /* primer sector a transferir */
 unsigned count;                   /* byte contador */
 phys_bytes phys;                  /* dirección física del usuario */
} wtrans[NR_IOREQS];
PRIVATE struct trans *w_tp;        /* pa'añadir peticiones de transferencia */
PRIVATE unsigned w_count;          /* número de bytes a transferir */
PRIVATE unsigned long w_nextblock; /* sig.bloque de disco a transferir */
PRIVATE int w_opcode;              /* DEV_READ o DEV_WRITE */
PRIVATE int w_command;             /* orden en ejecución */
PRIVATE int w_status;              /* estado después de la interrupción */
PRIVATE int w_drive;               /* dispositivo seleccionado */
PRIVATE struct device *w_dv;       /* base y tamaño del dispositivo */

FORWARD _PROTOTYPE( void init_params, (void) );
FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
FORWARD _PROTOTYPE( int w_identify, (void) );
FORWARD _PROTOTYPE( char *w_name, (void) );
FORWARD _PROTOTYPE( int w_specify, (void) );
FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
FORWARD _PROTOTYPE( int w_finish, (void) );
FORWARD _PROTOTYPE( int com_out, (struct command *cmd) );
FORWARD _PROTOTYPE( void w_need_reset, (void) );
FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( int com_simple, (struct command *cmd) );
FORWARD _PROTOTYPE( void w_timeout, (void) );
FORWARD _PROTOTYPE( int w_reset, (void) );
FORWARD _PROTOTYPE( int w_intr_wait, (void) );
FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value) );
FORWARD _PROTOTYPE( int w_handler, (int irq) );
FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry) );

#define waitfor(mask, value) \
      ((in_byte(w_wn->base+REG_STATUS)&mask) == value || w_waitfor(mask, value))

/* Puntos de entrada para este manejador */
PRIVATE struct driver w_dtab = {
w_name,           /* nombre del dispositivo actual */
w_do_open,        /* petición apertura/montaje, inicializar dispositivo */
w_do_close,       /* liberar dispositivo */
do_diocntl,       /* averiguar o fijar la geometría de una partición */
w_prepare,        /* preparar para E/S en un dispositivo menor dado */
w_schedule,       /* precalcular cilindro, cabeza, sector, etc. */
w_finish,         /* hacer E/S */
nop_cleanup,      /* nada que limpiar */
w_geometry,       /* decir la geometría del disco */
};

#if ENABLE_ATAPI
      #include "atapi.c"          /* código extra para ATAPI CD-ROM */
#endif
```

at_winchester_task

No acepta ni devuelve nada

Descripción: Es el punto de entrada del manejador de disco duro

- Lo primero que realiza *at_winchester_task* es llamar a *init_params* para inicializar la estructura **WINI** con la nformación a cerca de la configuración lógica del disco duro

- Seguidamente llama a la función *driver_task* del fichero *driver.c* que entrará en un bucle infinito hasta que se produzca un intento de acceso al disco duro

```
/*===========================================================*
 * AT_WINCHESTER_TASK                                        *
 * Coloca los parámetros especiales del disco y llama al     *
 * bucle genérico especial                                   *
 *===========================================================*/

PUBLIC void at_winchester_task()
{
      /* inicializa la estructura WINI */
      init_params();

      /* le pasa un puntero a la tabla de tareas del manejador */
      driver_task(&w_dtab);
}
```

init_params

No acepta ni devuelve nada

Descripción: Su tarea principal es inicializar la estructura **WINI** con información acerca de la configuracíón lógica del disco duro

- Obtiene el número de unidades de disco del área de datos del BIOS.
- Para cada unidad lee sus parámetros del BIOS y los copia en su entrada correspondiente de la estructura WINI.

```
/*============================================================*
 * INIT_PARAMS                                                *
 * Esta rutina es llamada en el arranque para inicializar los *
 * parametros de las unidades                                 *
 *============================================================*/

PRIVATE void init_params()
{
 u16_t parv[2];
 unsigned int vector;
 int drive, nr_drives, i;
 struct wini *wn;
 u8_t params[16];
 phys_bytes param_phys = vir2phys(params);

 /* Obtiene el número de unidades del area de datos del BIOS */
 phys_copy(0x475L, param_phys, 1L);

 if ((nr_drives = params[0]) > 2) nr_drives = 2;

 for (drive = 0, wn = wini; drive < MAX_DRIVES; drive++, wn++) {
      if (drive < nr_drives) {
            /* Copiar el vector de parámetros del BIOS */
            vector = drive == 0 ? WINI_0_PARM_VEC : WINI_1_PARM_VEC;
            phys_copy(vector * 4L, vir2phys(parv), 4L);

            /* Calcular la dirección de los parámetros y copiarlos */
            phys_copy(hclick_to_physb(parv[1]) + parv[0], param_phys, 16L);

            /*Copia los parámetros en la estructura de la unidad */
            wn->lcylinders = bp_cylinders(params);
            wn->lheads = bp_heads(params);
            wn->lsectors = bp_sectors(params);
            wn->precomp = bp_precomp(params) >> 2;
      }
```

```
      wn->ldhpref = ldh_init(drive);
      wn->max_count = MAX_SECS << SECTOR_SHIFT;
      if (drive < 2) {
            /* Controladora 0. */
            wn->base = REG_BASE0;
            wn->irq = AT_IRQ0;
      } else {
            /* Controladora 1. */
            wn->base = REG_BASE1;
            wn->irq = AT_IRQ1;
      }
 }
}
```

w_do_open

Acepta *message y *driver y devuelve:OK o error

Descripción: Se llama cuando un mensaje requiere una operación DEV_OPEN

- Llama a *w_prepare* para determinar si el dispositivo requerido es válido
- Posteriormente llama a *w_identify* que identifica el tipo de dispositivo e inicializa algunos parámetros adicionales en la estructura **WINI**
- Seguidamente, se usa un contador en la estructura **WINI** para comprobar si es la primera vez que el dispositivo es abierto desde que MINIX fue inicializado (*open_ct*)
- Incrementa el contador y si es la primera operación con DEV_OPEN se llama a la función *partition* del fichero *drvlib.c*

```
/*==========================================*
 * W_DO_OPEN                                *
 * Abrir dispositivo: Inicializar el controlador y *
 * leer la tabla de particiones             *
 *==========================================*/

PRIVATE int w_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
 int r;
 struct wini *wn;
 struct command cmd;
 if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
 wn = w_wn;

 /* comprobamos si está inicializado */
 if (wn->state == 0) {
      /* Intenta identificar el dispositivo */
      if (w_identify() != OK) {
            printf("%s: probe failed\n", w_name());
            if (wn->state & DEAF) w_reset();
            wn->state = 0;
            return(ENXIO);
      }
 }

 /* Lee particiones del disco si es la 1ra vez que se abre el dispositivo */
 if (wn->open_ct++ == 0) {
      partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
 }
 return(OK);
}
```

w_prepare

Acepta el dispositivo menor de la unidad a utilizar y devuelve *device o NILDEV

Descripción:

- Acepta como parámetro de entrada un entero que es el número del dispositivo menor dentro de la unidad o de partición a utilizar y retorna un puntero a la entrada del dispositivo en la estructura WINI que indica la dirección base y el tamaño del mismo entre otras cosas.
- Se puede determinar si un dispositivo es una unidad, una partición o una subpartición por el número de su dispositivo menor.

```
/*==============================*
 * W_PREPARE                    *
 * Preparar dispositivo para E/S *
 *==============================*/

PRIVATE struct device *w_prepare(device)
int device;
{
  w_count = 0; /* No se ha realizado ninguna transferencia hasta ahora */

      /* para las particiones */
 if (device < NR_DEVICES) {              /* hd0, hd1, ... */
      w_drive = device / DEV_PER_DRIVE;  /* obtiene el número de la unidad */
      w_wn = &wini[w_drive];             /* apunta a su entrada en WINI */
      w_dv = &w_wn->part[device % DEV_PER_DRIVE]; /* apunta a la partición */
 } else
 /*para las subparticiones*/
 if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) {     /* hd1a, hd1b, ... */
   w_drive = device / SUB_PER_DRIVE;        /* obtiene el número de la unidad */
   w_wn = &wini[w_drive];                   /* apunta a su entrada en WINI    */
   w_dv = &w_wn->subpart[device % SUB_PER_DRIVE]; /* apunta a la subpartición */
 } else {
   return(NIL_DEV);     /*dispositivo no válido*/
 }
 return(w_dv); /*retorna puntero a la entrada en WINI del dispositivo elegido */
}
```

w_identify

No acepta nada y devuelve OK o ERR

Descripción: Distingue entre los diversos tipos de disco existentes

- El primer paso es comprobar que existe un puerto de E/S que se pueda leer y escribir donde debiera haberlo para la familia AT de controladoras
- Si es así se pone la dirección del manejador de interrupciones de disco duro en la tabla de descriptores de interrupción
- Se habilita dicho manejador para que responda a esa interrupción
- Se manda a la controladora el comando ATA_IDENTIFY para obtener de ella datos como modelo del disco, parámetros de los sectores, cilindros y cabezas (no obligatoriamente coincidentes con su configuración en la BIOS)
- También se averiguará si el disco es compatible con el direccionamiento lineal de bloques (LBA).Pues si es así se pueden ignorar los parámetros de sector, cabeza y cilindro y direccionar los sectores mediante números absolutos, lo que es mucho más simple
- Si *init_params* no consiguió información del BIOS sobre la configuración lógica del disco duro, se suplen estos parámetros con la información obtenida de la controladora
- Si el comando ATA_IDENTIFY falla significa que el disco es un modelo antiguo que no soporta el comando. Nos tendremos que conformar con los valores proporcionados por *init_params*, que de ser válidos se copiarían en los campos correspondientes a los parámetros fisicos del disco en WINI, si no se devolverá error y el disco no será utilizable

- El tamaño que el manejador puede gestionar debe ser limitado si el producto cilindros x cabezas x sectores es demasiado grande (> 4Gbytes). La dirección base y el tamaño se introducen en WINI
- Finalmente se llama a *w_specify* para pasar a la controladora los parámetros que han de ser usados y se saca por pantalla el nombre del dispositivo determinado por *w_name* y la ristra de identificación o –en caso de ser un disco antiguo- los parámetros de la cabeza, sector y cilindro obtenidos de la BIOS

```
/*============================================================*
 * W_IDENTIFY                                                 *
 * Averigua si un dispositivo existe, si es un disco antiguo  *
 * AT o un moderno ATA o si es un disco extraible, etc.       *
 *============================================================*/

PRIVATE int w_identify()
{
struct wini *wn = w_wn;
 struct command cmd;
 char id_string[40];
 int i, r;
 unsigned long size;
#define id_byte(n)   (&tmp_buf[2 * (n)])
#define id_word(n)   (((u16_t) id_byte(n)[0] << 0)|((u16_t) id_byte(n)[1] << 8))
#define id_longword(n)  (((u32_t) id_byte(n)[0] << 0) \
                  |((u32_t) id_byte(n)[1] << 8) \
                  |((u32_t) id_byte(n)[2] << 16) \
                  |((u32_t) id_byte(n)[3] << 24))
 /* Comprueba si un registro determinado existe */
 r = in_byte(wn->base + REG_CYL_LO);
 out_byte(wn->base + REG_CYL_LO, ~r);
 if (in_byte(wn->base + REG_CYL_LO) == r) return(ERR);
 /* Ok: dir. del manejador a tabla de descriptores de interrupción */
 put_irq_handler(wn->irq, w_handler);
 enable_irq(wn->irq);                    /* habilita la linea de interrupción */
 cmd.ldh = wn->ldhpref;
 cmd.command = ATA_IDENTIFY;             /* se envia orden ATA */
 if (com_simple(&cmd) == OK) {
      /* Es un dispositivo ATA. */
      wn->state |= SMART;
      /* Obtener información de la controladora */
      port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);
      for (i = 0; i < 40; i++) id_string[i] = id_byte(27)[i^1];

/* se guardan los parametros fisicos */
      wn->pcylinders = id_word(1);
      wn->pheads = id_word(3);
      wn->psectors = id_word(6);
      size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;

      if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
  /* El dispositivo puede realizar LBA y tiene capacidad suficiente para ello */
            wn->ldhpref |= LDH_LBA;
            size = id_longword(60);
      }

      if (wn->lcylinders == 0) {
            /* ¿No hay parámetros BIOS? Ponemos los de la controladora */
            wn->lcylinders = wn->pcylinders;
            wn->lheads = wn->pheads;
            wn->lsectors = wn->psectors;
            while (wn->lcylinders > 1024) {
                  wn->lheads *= 2;
                  wn->lcylinders /= 2;
            }
      }
 } else {
```

```
      /* No es un dispositivo ATA; no hay ni características especiales
      ¡ No tocar si la BIOS no la identifica ! */
      if (wn->lcylinders == 0) return(ERR);    /* no hay parámetros BIOS */
      wn->pcylinders = wn->lcylinders;
      wn->pheads = wn->lheads;
      wn->psectors = wn->lsectors;
      size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
 }
      /* El tope está en 4 GB */
      if (size > ((u32_t) -1) / SECTOR_SIZE)
            size = ((u32_t) -1) / SECTOR_SIZE;
      /* Base y tamaño de la unidad completa */
      wn->part[0].dv_base = 0;
      wn->part[0].dv_size = size << SECTOR_SHIFT;

      if (w_specify() != OK && w_specify() != OK) return(ERR);

      /* sacar por pantalla identificación o parámetros */
      printf("%s: ", w_name());

      if (wn->state & SMART) {
            printf("%.40s\n", id_string);
      } else {
            printf("%ux%ux%u\n", wn->pcylinders, wn->pheads, wn->psectors);
      }
      return(OK);
}
```

w_name

No acepta nada y devuelve un puntero a ristra con el nombre de la partición

Descripción: Devuelve un puntero a una ristraque contiene el nombre de la partición

```
/*============================================*
 * W_NAME                                     *
 * Devuelve un nombre para el dispositivo actual *
 *============================================*/

PRIVATE char *w_name()
{
static char name[] = "at-hd15";
unsigned device = w_drive * DEV_PER_DRIVE;

if (device < 10) {
      name[5] = '0' + device;
      name[6] = 0;
} else {
      name[5] = '0' + device / 10;
      name[6] = '0' + device % 10;
}
return name;
}
```

w_specify

No acepta nada y devuelve OK

Descripción: Pasa los parámetros a la controladora y si se trata de un disco antiguo, hace un recalibrado del dispositivo haciendo una búsqueda del cilindro cero

```
/*============================================================*
 * W_SPECIFY                                                  *
 * Rutina para inicializar el dispositivo después del arranque *
```

```
 * o cuando se necesita de una reinicialización                *
 *============================================================*/
PRIVATE int w_specify()
{
struct wini *wn = w_wn;
 struct command cmd;

 if ((wn->state & DEAF) && w_reset() != OK) return(ERR);
 /* Especificar parámetros: precompensación, número de cabezas y sectores */

 cmd.precomp = wn->precomp;
 cmd.count = wn->psectors;
 cmd.ldh = w_wn->ldhpref | (wn->pheads - 1);
 cmd.command = CMD_SPECIFY;        /* especificar algunos parametros */

 if (com_simple(&cmd) != OK) return(ERR);

 /* Calibrar un disco antiguo */
 if (!(wn->state & SMART)) {
      cmd.sector = 0;
      cmd.cyl_lo = 0;
      cmd.cyl_hi = 0;
      cmd.ldh = w_wn->ldhpref;
      cmd.command = CMD_RECALIBRATE;

      if (com_simple(&cmd) != OK) return(ERR);
 }
 wn->state |= INITIALIZED;
 return(OK);
}
```

w_schedule

Acepta un número de proceso, un puntero a la petición de L/E y devuelve OK o EINVAL

Descripción: Agrupa peticiones de E/S de bloques consecutivos de forma que puedan ser leidos/escritos con un comando de controladora (hay tiempo suficiente para computar la siguiente petición consecutiva mientras pasa de largo un sector no deseado).

- Configura los parámetros básicos como de donde vienen los datos, adónde van, número de bytes a transferir y tipo de operación ,comprobando que son correctos
- Comprueba si la petición se excede del último byte del dispositivo y si es así la reduce y se calcula el siguiente bloque a leer
- Si hay peticiones pendientes (*w_count* >0) y el siguiente bloque a leer no es contiguo al anterior se llama a *w_finish* para que atienda las peticiónes contiguas
- Si el bloque es contiguo incrementa *w_nextblock* y se entra en un bucle que lo que hace es añadir nuevas peticiones de sector al array de peticiones. Así hasta que se llegue al número máximo permitido (*max_count*)

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

PRIVATE int w_schedule(proc_nr, iop)
int proc_nr;                 /* proceso que realiza la petición */
struct iorequest_s *iop;     /* puntero a petición de lectura o escritura */
{
 struct wini *wn = w_wn;
 int r, opcode;
 unsigned long pos;
 unsigned nbytes, count;
 unsigned long block;
 phys_bytes user_phys;
 /* Hay que leer/escribir todos estos bytes */
```

```
 nbytes = iop->io_nbytes;
 if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
 /* desde/hasta esta posición en el dispositivo */
 pos = iop->io_position;
 if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
 /* desde/hasta esta dirección de usuario */
 user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
 if (user_phys == 0) return(iop->io_nbytes = EINVAL);

 /* ¿Escribir o leer ? */
 opcode = iop->io_request & ~OPTIONAL_IO;
 /* ¿Qué bloque del disco y cuán cerca del final del fichero? */
 if (pos >= w_dv->dv_size) return(OK); /* si final del fichero no se trata */
 if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
 block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;

 /* Esta nueva petición no puede ser encadenada a la tarea que construimos */
 if (w_count > 0 && block != w_nextblock) {
      if ((r = w_finish()) != OK) return(r);
 }
 /* Siguiente bloque consecutivo */
 w_nextblock = block + (nbytes >> SECTOR_SHIFT);

 /* Mientras haya bytes sin planificar en la petición */
 do {
      count = nbytes;

      if (w_count == wn->max_count) {
            /* El dispositivo no puede abarcar más de max_count de una vez */
            if ((r = w_finish()) != OK) return(r);
      }
      if (w_count + count > wn->max_count)
            count = wn->max_count - w_count;
      if (w_count == 0) {
                        /* Inicializar la primera petición de la serie*/
            w_opcode = opcode;
            w_tp = wtrans;
      }
      /* Almacena los parametros de E/S */
      w_tp->iop = iop;
      w_tp->block = block;
      w_tp->count = count;
      w_tp->phys = user_phys;
      /* Actualiza contadores */
      w_tp++;
      w_count += count;
      block += count >> SECTOR_SHIFT;
      user_phys += count;
      nbytes -= count;
 } while (nbytes > 0);
 return(OK);
}
```

w_finish

No acepta nada y devuelve OK o EIO (error E/S)

Descripción: Su función principal es crear los comandos que van a enviarse a la controladora a partir de las peticiones de lectura/escritura

- Si hay peticiones pendientes, se fuerza la reinicialización de la controladora y se entra en un bucle de creación de comandos
- Se llama a *w_specify* para comprobar si la reinicialización ha tenido éxito en cuyo caso se prepara el comando que se va a enviar a la controladora

- Según el tipo de direccionamiento de la controladora se creará el comando bien como un bloque de 28 bits (caso del direccionamiento lineal por bloques LBA) o bien especificando cilindro, cabeza y sector
- Se almacena el tipo de operación en el comando y se llama a *com_out* para iniciar la transferencia
- Si la controladora no está preparada se vuelve a intentar hasta un número máximo de intentos. Si, por el contrario, la controladora acepta el comando, la tarea de disco se bloquea para dar tiempo a que se lean o escriban los datos llamando a *w_intr_wait*
- En caso de lectura: si *w_intr_wait* no devuelve un error, se transfieren del puerto de la controladora al bloque de caché del sistema de ficheros, tantos bytes como indique SECTOR_SIZE. Si devuelve error pueden haberse leido datos defectuosos que, de haberse corregido, se leerán igualmente
- Si el contador de bytes de la petición actual llega a cero, se avanza el puntero del vector de peticiones.
- En el caso de escritura: se espera a que la controladora esté lista para recibir datos mediante una llamada *waitfor*. Entonces, se transfieren los datos desde la memoria al puerto de datos de la controladora y se llama a *w_intr_wait* para bloquear la tarea de disco. Cuando la interrupción llega y se despierta la tarea de disco, se actualizan los contadores de bytes
- Finalmente, si han ocurrido errores en la lectura/escritura, se tratarán según su tipo. Si la controladora informa que el error se debe a un sector defectuoso, no lo intenta nuevamente, pero ante otro tipo de error, se intenta un número de veces igual a MAX_ERRORS/2, y si persiste el error se llama a *w_need_reset* para forzar la reinicialización de la controladora. En cualquier caso, la variable *command* se pone a CMD_IDLE para distinguir si el error que se ha producido es de tipo hardware o no

```
/*===========================================================*
 * W_FINISH                                                  *
 * Lleva a cabo las peticiones de E/S acumuladas en wtrans[] *
 *===========================================================*/
PRIVATE int w_finish()
{
 struct trans *tp = wtrans;
 struct wini *wn = w_wn;
 int r, errors;
 struct command cmd;
 unsigned cylinder, head, sector, secspcyl;

 if (w_count == 0) return(OK);           /* Termina espúreamente. */

 r = ERR;                                /* Disparar el primer com_out */
 errors = 0;
 do {
      if (r != OK) { /* La controladora debe ser reprogramada */
            /*Primero verificar si se necesita reinicializar.*/
            if (!(wn->state & INITIALIZED) && w_specify() != OK)
                  return(tp->iop->io_nbytes = EIO);
            /* Decir a la controladora que transfiera w_count bytes */
            cmd.precomp = wn->precomp;
            cmd.count = (w_count >> SECTOR_SHIFT) & BYTE;
            if (wn->ldhpref & LDH_LBA) {            /* comando LBA */
                  cmd.sector = (tp->block >> 0) & 0xFF;
                  cmd.cyl_lo = (tp->block >> 8) & 0xFF;
                  cmd.cyl_hi = (tp->block >> 16) & 0xFF;
                  cmd.ldh = wn->ldhpref | ((tp->block >> 24) & 0xF);
            } else {                                /* comando antiguo */
                  secspcyl = wn->pheads * wn->psectors;
                  cylinder = tp->block / secspcyl;
                  head = (tp->block % secspcyl) / wn->psectors;
                  sector = tp->block % wn->psectors;
                  cmd.sector = sector + 1;
                  cmd.cyl_lo = cylinder & BYTE;
                  cmd.cyl_hi = (cylinder >> 8) & BYTE;
                  cmd.ldh = wn->ldhpref | head;
            }

      cmd.command = w_opcode == DEV_WRITE ? CMD_WRITE : CMD_READ;

      if ((r = com_out(&cmd)) != OK) {
```

```
                if (++errors == MAX_ERRORS) {
                        w_command = CMD_IDLE;
                        return(tp->iop->io_nbytes = EIO);
                }
                continue;   /* Reintentar */
        }
    }
/* Para cada sector ,esperar por interrupción y traer datos (lectura)
 o suministrar datos a la controladora y esperar por interrupcion (escritura)*/
    if (w_opcode == DEV_READ) {
        if ((r = w_intr_wait()) == OK) {
        /*Copiar datos del buffer del dispositivo al espacio de usr */
            port_read(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
            tp->phys += SECTOR_SIZE;
            tp->iop->io_nbytes -= SECTOR_SIZE;
            w_count -= SECTOR_SIZE;
            if ((tp->count -= SECTOR_SIZE) == 0) tp++;
        } else {                /* ¿Hay datos defectuosos? */
            if (w_status & STATUS_DRQ)
                port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);
        }
    } else {
        /* Espera por los datos requeridos */
        if (!waitfor(STATUS_DRQ, STATUS_DRQ)) {
            r = ERR;
        } else {
            /* Llena el búfer del dispositivo */
            port_write(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
            r = w_intr_wait();
        }
        if (r == OK) {
        /* Contabiliza los bytes que han sido escritos con éxito */
            tp->phys += SECTOR_SIZE;
            tp->iop->io_nbytes -= SECTOR_SIZE;
            w_count -= SECTOR_SIZE;
            if ((tp->count -= SECTOR_SIZE) == 0) tp++;
        }
    }

    if (r != OK) {

   /* No reintentar si el sector es defectuoso o hay demasiados errores */

        if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS) {
            w_command = CMD_IDLE;
            return(tp->iop->io_nbytes = EIO);
        }

   /* Reinicializar a mitad del max de errores, salir si es E/S opcional */
        if (errors == MAX_ERRORS / 2) {
            w_need_reset();
            if (tp->iop->io_request & OPTIONAL_IO) {
                w_command = CMD_IDLE;
                return(tp->iop->io_nbytes = EIO);
            }
        }
        continue;   /* Reintentar */
    }
    errors = 0;
 } while (w_count > 0);

 w_command = CMD_IDLE;
 return(OK);
}
```

com_out

Acepta un puntero a la estructura comando *cmd* y devuelve OK o ERR

Descripción: Envía un comando a la controladora winchester y retorna el estado

- Se lee el registro de estado para determinar que la controladora no esté ocupada, comprobando el bit STATUS_BSY mediante una llamada a *waitfor*
- Si la controladora está lista se escribe un byte para seleccionar el drive, la cabeza y modo de operación, y se vuelve a llamar a *wait_for*
- Para asegurar que un fallo del dispositivo no deje colgada la tarea de disco, se envía un mensaje a la tarea del reloj para que ordene la reactivación de aquella tras un tiempo establecido
- Antes de que esto ocurra, se escriben los distintos parámetros en los registros y la orden de lectura/escritura. Esto último, así como, la modificación de *w_command* y *w_status*, se realiza deshabilitando las interrupciones por tratarse de una sección crítica

.

```
/*==========================================================================*
 * COM_OUT                                                                  *
 * Sacar bloque de orden a la controladora winchester y retornar estado *
 *==========================================================================*/

PRIVATE int com_out(cmd)
struct command *cmd;          /* Bloque de orden */
{
 struct wini *wn = w_wn;
 unsigned base = wn->base;
 if (!waitfor(STATUS_BSY, 0)) {
      printf("%s: controller not ready\n", w_name());
      return(ERR);
 }                                        /* Seleccionar unidad. */
 out_byte(base + REG_LDH, cmd->ldh);
 if (!waitfor(STATUS_BSY, 0)) {
      printf("%s: drive not ready\n", w_name());
      return(ERR);
 }

 /* Planificar una llamada de reactivación */
 /* pues algunas controladoras se cuelgan  */
 clock_mess(WAKEUP, w_timeout);
 out_byte(base + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
 out_byte(base + REG_PRECOMP, cmd->precomp);
 out_byte(base + REG_COUNT, cmd->count);
 out_byte(base + REG_SECTOR, cmd->sector);
 out_byte(base + REG_CYL_LO, cmd->cyl_lo);
 out_byte(base + REG_CYL_HI, cmd->cyl_hi);
 lock();
 out_byte(base + REG_COMMAND, cmd->command);
 w_command = cmd->command;
 w_status = STATUS_BSY;
 unlock();
 return(OK);
}
```

w_need_reset

No acepta ni devuelve nada

Descripción: Esta función es llamada por *w_finish* cuando se han producido MAX_ERRORS/2 intentos de acceso, o bien, cuando el disco no está preparado. Su función es simplemente marcar la variable *state* de cada una de las entradas de la estructura wini, es decir, de cada dispositivo, como no inicializada para forzar la inicialización en el siguiente acceso.

```
/*=========================================*
 * W_NEED_RESET                            *
 * La controladora necesita se reinicializada *
 *=========================================*/
PRIVATE void w_need_reset()
{
      struct wini *wn;

      for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
            wn->state |= DEAF;
            wn->state &= ~INITIALIZED;
      }
}
```

w_do close

Acepta un puntero a dispositivo *dp* y un puntero a mensaje *m_ptr*. Devuelve OK

Descripción: Esta función tiene poco que hacer en un disco duro convencional. Se encarga de decrementar el campo *open_ct* de la estructura wini del dispositivo que queremos liberar. Aunque habría que añadirle más código si quisieramos manejar CD-ROM´s para controlar la bandeja de entrada, entre otras cosas ...

```
/*======================*
 * W_DO_CLOSE           *
 * Libera un dispositivo *
 *======================*/

PRIVATE int w_do_close(dp, m_ptr)
  struct driver *dp;
  message *m_ptr;
{
    if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
    w_wn->open_ct--;
    return(OK);
}
```

com_simple

Acepta un puntero a una ordne *cmd* y devuelve OK o error

Descripción: Es llamada para enviar comandos a la controladora que terminan rápidamente sin realizar transferencia de datos. Estos son aquellos que identifican el disco, configuran algunos parámetros y realizan la recalibración

```
/*=========================================================*
 * COM_SIMPLE                                              *
 * Una orden simple de la controladora: sólo una interrupcion *
 * sin salida de datos                                     *
 *=========================================================*/

PRIVATE int com_simple(cmd)
  struct command *cmd;       /* Bloque de orden */
{
     int r;

     if ((r = com_out(cmd)) == OK) r = w_intr_wait();
     w_command = CMD_IDLE;
     return(r);
}
```

w_timeout

No acepta ni devuelve nada

Descripción: Es la función que *com_out* manda a la tarea de reloj para ser despertada cuando ocurre un timeout después de un fallo de la controladora de disco

- Si se llama a esta función y *w_command* es igual a CMD_IDLE, simplemente retornará
- Si *w_command* vale CMD_WRITE o CMD_READ, significará que no se ha completado una operación de lectura o escritura. Esto se intentará resolver reduciendo el tamaño de las peticiones de E/S; ya sea a 8 sectores o a uno
- Para cada timeout se escribirá un mensaje por pantalla y se llamará a *w_need_reset* para forzar la reinicialización de todos los dispositivos en el siguiente intento de acceso
- Se llama a *interrupt* para enviar un mensaje a la tarea de disco y simular la interrupción hardware que debía haber ocurrido al final de la operación de disco

```
PRIVATE void w_timeout()
{
 struct wini *wn = w_wn;

 switch (w_command) {
 case CMD_IDLE:
      break;            /* No hay nada que hacer */
 case CMD_READ:
 case CMD_WRITE:
      /* La controladora no responde */
      /* Se puede resolver limitando la E/S multisector */
      if (wn->max_count > 8 * SECTOR_SIZE) {
            wn->max_count = 8 * SECTOR_SIZE;
      } else {
            wn->max_count = SECTOR_SIZE;
      }
 default:
      printf("%s: timeout on command %02x\n", w_name(), w_command);
      w_need_reset();
      w_status = 0;
      interrupt(WINCHESTER);
 }
```

w_reset

No acepta nada y devuelve OK o ERR.

Descripción: Ordena una reinicialización a la controladora después de cualquier catástrofe, como puede ser falta de respuesta por parte de la controladora

- Después de un retardo inicial para dar tiempo a que el dispositivo se recupere de la operación previa, se pone un bit del registro de control de la controladora a uno y de nuevo a cero
- Se llama a *waitfor* para dar tiempo al dispositivo a que pase a preparado. Si esto no ocurre se mostrará un mensaje notificándolo y se retornará error
- Se marcan las entradas de WINI, en su campo *estate*, como reinicializado

```
/*============================================================================*
 * W_RESET                                                                    *
 * Mandar un reset a la controladora. Esto se hace tras cualquier catastrofe, *
 * como la falta de respuesta por por parte de la controladora                *
 *============================================================================*/

PRIVATE int w_reset()
{
 struct wini *wn;
 int err;

 /* Espera por una recuperación interna de la unidad */
 milli_delay(RECOVERYTIME);
 out_byte(w_wn->base + REG_CTL, CTL_RESET);    /* Activa el bit de reset */
 milli_delay(1);
 out_byte(w_wn->base + REG_CTL, 0);
```

```
 milli_delay(1);
 /* Espera a que la controladora esté lista */
 if (!w_waitfor(STATUS_BSY | STATUS_RDY, STATUS_RDY)) {
      printf("%s: reset failed, drive busy\n", w_name());
      return(ERR);
 }
 /* Se marca cada entrada de WINI en state como inicializado */
 for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
      if (wn->base == w_wn->base) wn->state &= ~DEAF;
 }
 return(OK);
}
```

w_intr_wait

No acepta nada y devuelve status

Descripción: Es llamada después de escribir o leer un sector. Esta se queda esperando hasta que le llegue una interrupción que ponga *w_status* a “no ocupado”.Entonces se comprueba el estado de la petición. Esto se realiza desactivando las interrupciones para garantizar que no se cambie *w_status*

```
/*===========================================================================*
 * W_INTR_WAIT                                                               *
 * Espera por la interrupción de fin de una tarea y retornar resultados *
 *===========================================================================*/

PRIVATE int w_intr_wait()
{
 message mess;
 int r;
 /* Espera por una interrupción que ponga w_status a no ocupado */
 while (w_status & STATUS_BSY) receive(HARDWARE, &mess);

 /* Comprueba el estado */
 lock();
 if ((w_status & (STATUS_BSY|STATUS_RDY|STATUS_WF|STATUS_ERR)) == STATUS_RDY) {
      r = OK;
      w_status |= STATUS_BSY; /* asumimos que esta ocupada con la E/S */
 } else
 if ((w_status & STATUS_ERR) && (in_byte(w_wn->base + REG_ERROR) & ERROR_BB)) {
      r = ERR_BAD_SECTOR;     /* sector defectuoso, no servirá reintentar */
 } else {
      r = ERR;                /* cualquier otro error.*/
 }
 unlock();
 return(r);
}
```

w_waitfor

Acepta mascara, valor y devuelve 0 ó1

Descripción: Arranca un timer mediante *milli_start* y luego entra en un bucle que comprueba alternativamente el registro de estado y el propio timer. Si ocurre un timeout, se llama *w_need_reset* para preparar las cosas para que hay una reinicialización de la controladora la próxima vez que se soliciten sus servicios.

```
/*===========================================================*
 * W_WAITFOR                                                 *
 * Espera hasta que la controladora pase al estado deseado. *
 * Retorna cero si hay timeout                               *
 *===========================================================*/

PRIVATE int w_waitfor(mask, value)
int mask;               /* mascara de estado */
```

```
int value;              /* status requerido */
{
 struct milli_state ms;

 milli_start(&ms);
 do {
     if ((in_byte(w_wn->base + REG_STATUS) & mask) == value) return 1;
 } while (milli_elapsed(&ms) < TIMEOUT);

 w_need_reset();             /* Hay que reinicializarla */
 return(0);
}
```

w_handler

Acepta un entero irq (solicitud de interrupción) y devuelve un 1

Descripción: Es el manejador de interrupciones. Su dirección es puesta en la tabla de descriptores de interrupción por *w_identify* cuando la tarea de disco duro es activada por primera vez. Cuando se produce una interrupción de disco, el registro de estado de la controladora es copiado a *w_status* y entonces la función de interrupción del kernel es llamada para replanificar la tarea de disco duro

```
/*============================================================*
 * W_HANDLER                                                  *
 * Interrupción de disco, envia mensaje a la tarea winchester y *
 * rehabilita las interrupciones                              *
 *============================================================*/

PRIVATE int w_handler(irq)
int irq;
{
 w_status = in_byte(w_wn->base + REG_STATUS);  /* reconocer interrupción */
 interrupt(WINCHESTER);
 return 1;
}
```

w_geometry

Entrada/salida**:** Puntero a la estructura partición.

Descripción: Como parámetro de salida devolverá los valores lógicos máximos del cilindro,la cabeza y el sector del dispositivo de disco duro seleccionado. En este caso los números son auténticos no inventados como el caso del disco RAM.

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

PRIVATE void w_geometry(entry)
struct partition *entry;
{
 entry->cylinders = w_wn->lcylinders;
 entry->heads = w_wn->lheads;
 entry->sectors = w_wn->lsectors;
}
```