DISPOSITIVOS POR BLOQUES

DRIVER.C Y DRVLIB.C

María Dácil González Ortega
Israel Hernández González

INDICE

1.- INTRODUCCIÓN.

Cada clase de dispositivo de entrada/salida tiene asociado un controlador de dispositivo individual, el cual es un proceso con sus propios estados registros y pila. Los controladores de dispositivos se comunican con el sistema de archivo usando el mecanismo de transferencia de mensajes. Los controladores de dispositivo sencillos se escriben como archivos fuentes únicos, mientras que otros controladores (RAM, Disco Duro, Disco Flexible) tiene, además de un archivo fuente para manejar cada tipo de dispositivo, un conjunto de rutinas comunes que apoyan todos los tipos de hardware distinto. La separación de las partes del software dependientes del hardware e independientes del hardware facilita la adaptación a una diversidad de configuraciones de hardware diferentes. Aunque se use algo de código fuente común, el controlador de cada tipo de disco se ejecuta como proceso aparte, con el objeto de realizar transferencias de datos rápidas.

En el caso de grupo de dispositivos como los discos, para los cuales hay varios archivos fuentes, también hay archivos cabecera. Driver.h apoya todos los dispositivos por bloques.

La diferencia principal entre los controladores de dispositivos y otros procesos es que los primeros se enlazan juntos en el kernel, y, por tanto, comparten el mismo espacio de dirección.

Cuando un proceso de usuario solicita una operación de E/S, envía un mensaje al proceso del sistema de ficheros .Este a su vez , envía un mensaje al manejador del dispositivo pidiéndole que realice la operación .Una vez que la operación se haya llevado a cabo el manejador de dispositivo enviará un mensaje de respuesta al proceso que la solicitó. Esto se muestra en la figura 1.

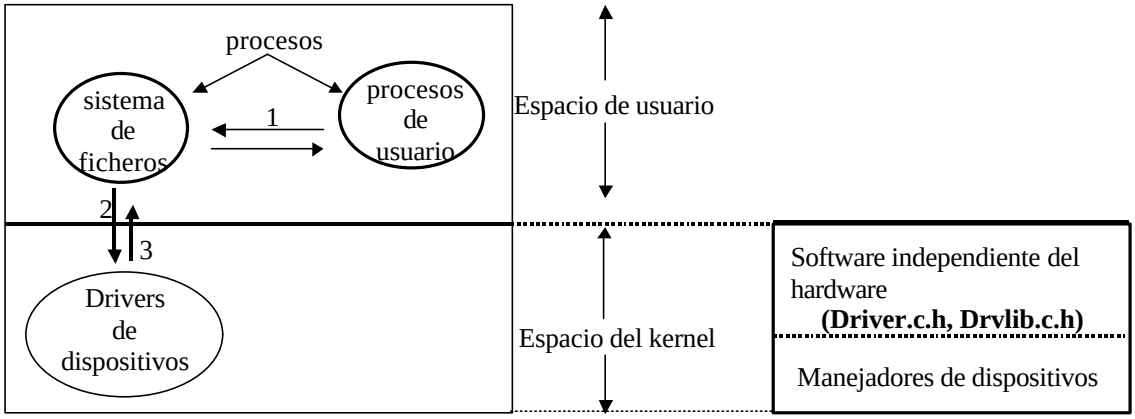

Figura 1.- Sistema estructurado en procesos.

Dentro del Kernel se encuentran los drivers de dispositivos de bloques y dentro de este conjunto de código podemos diferenciar dos niveles, un nivel donde encontramos un software independiente del hardware (donde se encuentra Driver.c.h y Drvlib.c.h) y un segundo nivel donde se encuentran los drivers propios de cada dispositivo, de tal forma de que si se cambia un dispositivo solo habrá que cambiar el driver de este dispositivo.

Para los dispositivos de bloques los mensajes de solicitud y respuesta son los que se muestran en la figura 2. El mensaje de petición incluye la dirección de un área de almacenamiento intermedio que contiene los datos que se van a trasmitir o en la que se esperan los datos recibidos. La respuesta incluye información de estado que permite al proceso solicitante verificar que su petición se llevó a cabo correctamente.

CAMPO	**TIPO**	**SIGNIFICADO**
m.m_type	int	Operación solicitada
m.DEVICE	int	Dispositivo secundario a utilizar
m.POSITION	long	Posición en el dispositivo
m.PROC_NR	int	Proceso que solicita la E/S
m.ADDRESS	char*	Dirección dentro de PROC_NR
m.COUNT	int	Bytes por transferir

RESPUESTA		
CAMPO	**TIPO**	**SIGNIFICADO**
m.m_type	int	Operación solicitada : siempre TASK_REPLY
m.REP_PROC_NR	int	Proceso de usuario que solicita la E/S m.PROC_NR
m.REP_STATUS	int	Bytes transferidos o código de error

Figura 2.- Campos de mensajes.

2.- GENERALIDADES DE LOS CONTROLADORES DE DISPOSITIVOS POR BLOQUES EN MINIX.

Los procedimientos principales de todas las tareas de E/S tienen una estructura similar. Minix tiene al menos tres tareas de dispositivo por bloque (RAM, Floppy y varios controladores de disco duro) compilados en el sistema. Aunque el controlador de cada uno de ellos se ejecuta como proceso independiente, el hecho de que todos se compilen como parte del código ejecutable del kernel permite compartir código.

Todos los controladores de dispositivo por bloque se invocan individualmente para la inicialización específica para el hardware, y luego, cada controlador invoca a la función que contiene el ciclo principal común. Este ciclo se ejecuta indefinidamente, no hay retorno al invocador. Dentro del mismo se recibe un mensaje, se invoca a la función que realice la operación requerida por el mensaje, y se construye el mensaje respuesta.

Existe una sola copia de código ejecutable del ciclo principal en memoria ,la cual es ejecutada por los distintos drivers de dispositivos de bloques de manera independiente, después de realizar una serie de operaciones inicialización.

Esto se consigue haciendo que cada driver individual pase al ciclo principal un puntero a una tabla que contiene las direcciones de las funciones que el driver utilizará para cada operación. Por lo tanto estas funciones son llamadas indirectamente.

Esta única copia de código ejecutable se ejecuta como ciclo principal de tres o más procesos distintos. Cada uno de ellos estará en un punto distinto del código en un instante dado, y opera con su conjunto de datos y su pila.

En el código 1 se muestra un bosquejo del ciclo principal.

```
message mess;

void share_io_task (struct driver_table *entry_points)  {

  while (TRUE) {
        receive(ANY,&mess);                /* se espera por un mensaje */
        caller=mess.source;                /* caller = proceso de donde llega el mensaje */
         switch(mess.type)
              {
                 case READ:       rcode=(*entry_ points->dev_read)(&mess); break;
                case WRITE:     rcode=(*entry_points->dev_write)(&mess);break;
                 /* existen otros tipos de operaciones que también se incluyen aquí  */
                 /* (OPEN,CLOSE,IOCTL )                                              */
                 default:             rcode= ERROR;
               }
          mess.type=  TASK_REPLY;
          mess.status = rcode;              /* código resultado */
          send(caller,&mess);               /* mensaje de respuesta */
}}
```

Código 1.- Ciclo principal.

Existen seis operaciones que pueden ser solicitadas a cualquier controlador de dispositivo, que se corresponden con los distintos valores que pueden ser encontrados en el campo m.m_type del mensaje:

1. OPEN y CLOSE: tienen un significado similar a las llamadas al sistema open y close para manejo de ficheros.

   OPEN: verifica que el dispositivo es accesible y devuelve un mensaje de error en caso de que no lo sea.

CLOSE: garantiza que todos los datos en buffer que han sido escritos por el invocador sean completamente transferidos al dispositivo.

2. READ y WRITE: por cada una de estas operaciones un bloque de memoria es transferido al dispositivo o viceversa.

3. IOCTL : Se utiliza para cambiar los parámetros propios de cada dispositivo. Como pueden ser los parámetros de una partición.

4. SCATTERED : esta operación permite al File System hacer una solicitud para leer o escribir múltiples bloques.

En un READ no todos los bloques tienen que haber sido solicitados por el proceso que realiza la llamada, el sistema operativo puede intentar anticipar futuras solicitudes. De esta forma no todas las transferencias de bloques solicitadas tienen que ser realizadas. La solicitud de cada bloque debe de ser notificada junto con un flag que indica al driver que la solicitud es opcional. De esta forma el driver podrá llevar a cabo la política que considere mejor a la hora de responder a estas peticiones. En un WRITE no es opcional si se escribe o no un determinado bloque. Sin embargo, el sistema operativo puede utilizar bufferes para conseguir que la escritura en el dispositivo se haga de forma más eficiente.

En una petición de este tipo la lista de bloques solicitados está ordenada, lo que hace la operación más eficiente. Además, sólo se hace una llamada al controlador, lo que reduce el número de mensajes enviados dentro del Minix.

3.- SOFTWARE CONTROLADOR DE DISPOSITIVOS DE BLOQUES COMÚN.

3.1.- DRIVER.H

Están localizadas todas las definiciones, estructuras y variables que utilizan todos los drivers de dispositivos por bloques, además de los prototipos de las funciones que se encuentran en driver.c.

Lo más importante de este archivo es la estructura **driver**, es utilizada por todos los controladores para pasar una lista con las direcciones de las funciones específicas de cada controlador. Esta estructura se muestra en el código 2.

```
struct driver {
        _PROTOTYPE ( char *(*dr_name) , (void) );
        _PROTOTYPE ( int (*dr_open), (struct driver *dp, message *m_ptr));
        _PROTOTYPE ( int (*dr_close), (struct driver *dp ,message *m_ptr));
        _PROTOTYPE ( int (*dr_ioctl), (struct driver *dp, message *m_ptr));
        _PROTOTYPE ( struct device *(*dr_prepare), (int device));
        _PROTOTYPE ( int (*dr_schedule), (int proc_nr, struct iorequest_s *request));
        _PROTOTYPE (int (*dr_finish), (void));
        _PROTOTYPE ( void (*dr_cleanup), (void));
        _PROTOTYPE ( void (*dr_geometry), (struct partition *entry));
};
```

Código 2.- Estructura driver.

La descripción de la estructura es la siguiente:

- dr_name: devuelve el nombre del dispositivo.
- dr_open: comprueba que el dispositivo es accesible.
- dr_close: comprueba que se han realizado todas las trasferencias desde el buffer. Como en el caso de la RAM esta operación no tiene sentido, este campo tiene un puntero a la función do_nop.
- dr_ioctl: operación para cambiar parámetros del dispositivo.
- dr_prepare: comprueba que el identificador del dispositivo es valido y devuelve una estructura device con la base y tamaño del dispositivo en bytes.
- dr_shedule: realiza la operación de lectura o escritura de un solo bloque. En el caso del Disco Duro no se realizan todas las operaciones en el momento de la invocación.
- dr_finish: realiza las operaciones de E/S pendientes (solo para el Disco Duro).
- dr_cleanup: función que sólo tiene sentido en el caso del floppy, insertando un tiempo de retardo. El driver de la RAM y el disco duro tienen en este campo un puntero op_cleanup.
- dr_geometry: devuelve la geometría de un dispositivo sectores, head, cilindros, ...

Lo siguiente que aparece en DRIVER.H es una función que, dada una dirección de memoria phys devuelve el numero de bytes desde esta dirección a la frontera de 64K. Es utilizada para comprobar que no se sobrepasa frontera de 64K en init_buffer. Ésta se muestra en el código 3.

```
#define dma_bytes_left
(phys((unsigned) (sizeof(int) == 2 ? 0 : 0x100000) - (unsigned) ((unsigned) ((phys)&0xFFFF))
```

Código 3.- Función DMA_BYTES_LEFT

También se define aquí la estructura device, que se utiliza para guardar la dirección virtual base de un dispositivo o partición y el tamaño en bytes, tal y como aparece en el código 4.

```
struct device {
        unsigned long dv_base;
        unsigned long dv_size;
}
```

Código 4.- Estructura device.

3.2.- DRIVER.C

En DRIVER.C se encuentra el ciclo principal y las funciones compartidas de todas las tareas de controlador por bloque.

Después de efectuar toda la inicialización específica del hardware que pudiera ser necesaria, cada controlador invoca a **driver_task**, pasando una estructura **driver** como argumento de la llamada.

En el código 5 se muestra la función **driver_task**.

```
PUBLIC void driver_task (dp)
struct driver *dp;
{
  int r,caller,proc_nr;
  message mess;

  init_buffer();                          /* Obtener un buffer de DMA*/

  while (TRUE) {                          /* Bucle infinito */
    receive(ANY,&mess);                   /* Espera por una solicitud */

    caller=mess.m_source;
    proc_nr=mess.PROC_NR;

    switch (caller) {                     /* Comprueba que es un mensaje de FS */
      case HARDWARE: continue;            /* Interrupción permanente */
      case FS_PROC_NR: break;             /* El único invocador legítimo */
      default : printf("%s:got message from %d", (*dp->dr_name)(), caller);
                continue;
    }

    switch(mess.m_type) {                 /* Realizar la operación solicitada */
      case DEV_OPEN:                  r=(*dp->dr_open) (dp,&mess); break;
      case DEV_CLOSE:                 r=(*dp->dr_close) (dp,&mess); break;
      case DEV_IOCTL:                 r=(*dp->dr_ioctl) (dp,&mess); break;
      case DEV_READ:
      case DEV_WRITE:                 r=do_rdwt(dp,&mess); break;
      case DEV_SCATTERED_IO:          r=do_vrdwt(dp,&mess),break;
      default :                       r=EINVAL;
    }

    (*dp->dr_cleanup)();                  /* Aseo, función propia de cada dispositivo */

    mess.m_type=TASK_REPLY;               /* Prepara mensaje */
    mess.REP_PROC_NR=proc_nr;

    mess.REP_STATUS=r;                    /* # bytes transferidos o código de error */
    send(caller,&mess);                   /* Enviar respuesta al invocador */
  }
}
```

Código 5.- Función driver_task.

Como se puede observar, primero se obtiene la dirección del buffer que se usará para operaciones de DMA, y luego se entra en el ciclo principal, que se ejecuta indefinidamente.

El sistema de archivos es el único proceso que se supone enviará mensajes a una tarea de controlador. Así, en el primer switch se ignora una interrupción hardware, y cualquier otro mensaje mal dirigido sólo producirá una advertencia. En el siguiente switch, los primeros tres tipos de mensajes dan lugar a llamadas indirectas empleando las direcciones que se pasaron en la estructura **driver**. El resto, producen llamadas directas a **do_rdwt** o **do_vrdwt**.

Después de realizar las operaciones solicitadas quizás sea necesario realizar operaciones de aseo. También se usa una llamada indirecta para esto. Después del aseo se construye un mensaje de respuesta y se envía al invocador.

Lo primero que hace cada tarea después de entrar en **driver_task** es invocar **init_buffer**, función que se utiliza para obtener la dirección de un buffer para uso de operaciones con DMA.

Esta función existe debido al problema de que el buffer de DMA no puede traspasar frontera de 64K. Este problema se daba en los equipos antiguos de IBM, debido a que utilizaban el chip de DMA 8732A, el cual contiene los 16 bits de orden bajo de la dirección de DMA. Pero como el DMA utiliza direcciones absolutas (de 20 bits), los 4 bits de orden superior se cargan en un latch de 4 bits. Cuando el 8732A pasa de 0xFFFF a 0x0000, no genera un bit de acarreo que se sume al latch, de modo que la dirección de DMA salta repentinamente 64K hacia abajo en la memoria.

Un programa en C portable no puede especificar una posición de memoria absoluta para una estructura de datos, por lo que no se puede evitar que el compilador coloque el buffer en un lugar inadecuado. La solución consiste en asignar memoria a una arreglo de bytes dos veces más grande que lo necesario en **buffer**, y reservar un apuntador **tmp_buf** que se usará para acceder realmente a este arreglo. Esto lo realiza la función **init_buffer**, como se muestra en el código 6.

```
Private u8_t buffer[(unsigned )2*DMA_BUF_SIZE + BUF_EXTRA];
u8_t *tmp_buf;
phys_bytes tmp_phys;

PRIVATE void init_buffer() {
    tmp_buf=buffer;
    tmp_phys=vir2phys(buffer);

    if (tmp_phys==0) panic("no DMA buffer", NO_NUM);

   if (dma_bytes_left(tmp_phys)<DMA_BUF_SIZE)  {
      /* Primera mitad de buffer cruza un límite de 64K, no acceso por DMA ahí */
      tmp_buf+= DMA_BUF_SIZE;
      tmp_phys+=DMA_BUF_SIZE;
  }
}
```

Código 6.- Función init_buffer.

Init_buffer realiza un ajuste provisional de **tmp_buf**, apuntando al principio de **buffer**, y luego prueba para determinar si deja suficiente espacio antes de llegar a la frontera de 64K. Si el ajuste provisional no provee suficiente espacio, se incrementa **tmp_buf** en el número de bytes que realmente se

requieren. Así, siempre se desperdicia algo de espacio en alguno de los extremos del asignado en **buffer**, pero nunca hay fallo debido a que el buffer quede en una frontera de 64K. Todo esto se ve en la siguiente figura.

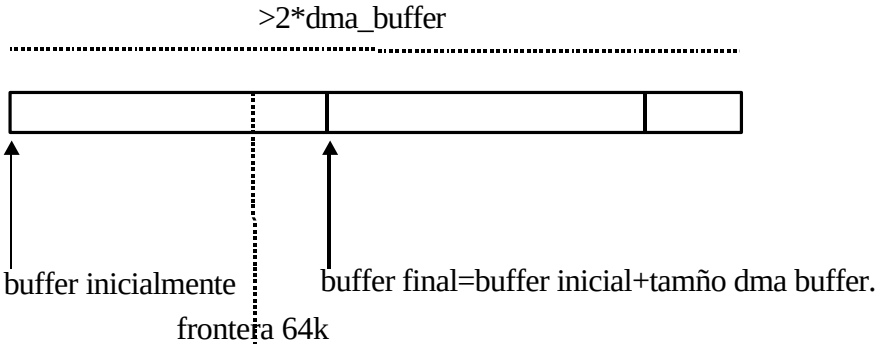

Figura 3.- Gestión del buffer de DMA.

La siguiente función de DRIVER.C es **do_rdwt**, que es invocada cuando se quieren hacer lecturas o escrituras de un solo bloque.

```
PUBLIC int do_rdwt(dp ,m_ptr)
struct driver dp;                    /* Puntos de entrada dependientes del dispositivo */
message m_ptr;                       /* Apuntador para leer o escribir mensaje */
{
 /* Ejecutar una sola solicitud de leer o escribir */
 struct iorequest_s ioreq;
 int r;

 if (m_ptr→ COUNT <=0) return(EINVAL);  /* ¿ # bytes a transferir > 0 ? */
 /* Comprueba se el identificador de dispisitivo es correcto */
 if ((*dp→ dr_prepare)(m_ptr→ DEVICE)==NIL_DEV) return(ENXIO);

 ioreq.io_request=m_ptr→ m_type;
 ioreq.io_buf=m_ptr→ ADDRESS;
 ioreq.io_position=m_ptr→ POSITION;
 ioreq.io_nbytes=m_ptr→ COUNT;

 r=(*dp→ dr_shedule)(m_ptr→ PROC_NR,&ioreq);

 if  (r==OK) (void)(*dp→ dr_finish)();
 /*Se devuelve la diferencia entre los bytes solicitados y los transferidos, o código de
error*/
 r=ioreq.io_nbytes;
 return( r<0 ? r : m_ptr→ COUNT-r);}
```

Código 7.- Función do_rdwt.

Después de verificar que el número de bytes en la petición es positiva, se invoca a ***dr_prepare**. Ésta debe tener éxito, ya que sólo puede fallar si se especifica un dispositivo no válido en una operación OPEN. A continuación se llena una estructura **iorequest_s** estándar, como la mostrada en el código 8. Luego viene otra llamada indirecta, esta vez a ***dr_schedule**, que permite a un dispositivo manejar las peticiones en la forma que le resulte óptima. En el caso de un disco real, **dr_finish** apunta a una rutina que lleva a cabo todas las transferencias de datos pendientes solicitadas en todas las llamadas anteriores a ***dr_schedule** después de la última llamada a ***dr_finish**.

```
include/minix/type.h

struct iorquest_s {
        long io_position;           /* posición en el dispositivo */
        char *io_buf ;              /* buffer en procedimiento de usuario */
        int io_nbytes;              /* número de bytes a transferir */
        unsigned short io_request;  /* tipo de operación (lectura o escritura) */
};
```

Código 8.- Estructura iorequest_s.

El contador **io_nbytes** de la estructura **iorequest_s** devuelve un número negativo si hubo un error, o uno positivo para indicar la diferencia entre el número de bytes especificados en la petición y los bytes que se transfirieron con éxito. Al regresar al ciclo principal, el código de error negativo se devuelve en el campo REP_STATUS del mensaje de respuesta si hubo un error. Si no, los bytes que faltan por transferirse se restan de la petición original en el campo COUNT del mensaje, y el resultado se devuelve en REP_STATUS en el mensaje de respuesta de **driver_task**.

La siguiente función, **do_vrdwt**, maneja todas las peticiones de E/S dispersas. Para ello se utiliza el campo ADDRESS del mensaje para apuntar a un vector de estructuras **iorequest_s**, cada una de las cuales especifica la información que se necesita para una petición (la dirección del buffer, el desplazamiento dentro del dispositivo, el número de bytes y si se va a leer o escribir). Las operaciones deben ser todas de lectura o todas de escritura, y estar ordenadas en orden de bloques dentro del dispositivo.

El vector de peticiones debe copiarse en el espacio del kernel, después de esto se realizan las tres llamadas indirectas a las rutinas dependientes del dispositivo (***dr_prepare**, ***dr_schedule**, ***dr_finish**). La diferencia consiste en que **dr_schedule** se ejecuta una vez para cada petición, o hasta que ocurra un error. Por último se vuelve a copiar el vector de peticiones en el lugar de donde se obtuvo. El campo **io_nbytes** de cada elemento del vector se habrá modificado de forma que refleje el número de bytes transferidos.

El código de esta función es el siguiente:

```
PUBLIC int do_vrdwt(dp, m_ptr)
struct driver *dp;       /* Estrucura device con base y tamaño en bytes del dispositivo*/
message *m_ptr;          /* Puntero a mensaje de I/O */

/*En este caso el campo ADDRESS y COUNT dentro del mensaje tendrán un puntero a un vector
de  estructuras iorquest_s (un elemento por bloque a transferir) y el numero de bloques a transferir
respectivamente. */
{
  struct iorequest_s *iop;
  static struct iorequest_s iovec[NR_IOREQS];
  phys_bytes iovec_phys;
  unsigned nr_requests;
  int request;
  int r;
  phys_bytes user_iovec_phys;

  nr_requests = m_ptr→COUNT;

  /* Se comprueba si el número de peticiones de I/O (m_ptr→ COUNT) se corresponde con el
número de elementos del vector con iorequest_s ( estructuras).*/

  if (nr_requests > sizeof iovec / sizeof iovec[0])
        panic("FS passed too big an I/O vector", nr_requests);

  /*devuelve la direccion fisica de iovec (estructura dentro del kernel)*/
  iovec_phys = vir2phys(iovec);
 /* devuelve la direccion fisica de m_ptr->ADDRESS (vector iorequest_s dentro  del  espcio de
usuario) */
  user_iovec_phys = numap(m_ptr→ PROC_NR, (vir_bytes) m_ptr→ ADDRESS, (vir_bytes)
(nr_requests * sizeof iovec[0]));

 if (user_iovec_phys == 0)
        panic("FS passed a bad I/O vector", (int) m_ptr→ ADDRESS);

  /* Se copia el vector dentro del espacio del Kernel. */
  phys_copy(user_iovec_phys, iovec_phys,(phys_bytes) nr_requests * sizeof iovec[0]);

if ((*dp→ dr_prepare)(m_ptr→ DEVICE) == NIL_DEV) return(ENXIO);
```

```
/* Una llamada a schedule es hecha por cada elemento del vector de solicitudes. En el campo
io_nbytes de cada vector se devuelve el número de bytes transferidos en cada operación. */
 for (request = 0, iop = iovec; request < nr_requests; request++, iop++) {
        if ((r = (*dp→ dr_schedule)(m_ptr→ PROC_NR, iop)) != OK) break;
}

  /*Se llevaran a cabo las transferencias pendientes desde la última llamada a  esta función */
  if (r == OK) (void) (*dp→ dr_finish)();

  /* Se copia el vector dentro del espacio de usuario. Con el campo io_nbytes modificado */
  phys_copy(iovec_phys, user_iovec_phy(phys_bytes) nr_requests * sizeof iovec[0]);

  return(OK);
}
```

Código 9.- Función do_vrdwt.

Hemos visto que existen casos en que hay drivers en los que ciertas funciones no tienen sentido, como por ejemplo las operaciones de close, finish, cleanup para la RAM. Las siguientes rutinas de DRIVER.C proporcionan apoyo general a las operaciones anteriores, y que pueden ser compartidas por distintos driver en estos casos en donde estas operaciones no tengan sentido.

Se usa ***dr_name** para obtener el nombre de un dispositivo. Si el dispositivo no tiene un nombre específico, la función **no_name** obtiene el nombre de dicho dispositivo de la tabla de tarea.

```
PUBLIC char *no_name()
{
  /* Si no hay nombre específico para el dispositivo */
  return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);
}
```

Código 10.- Función no_name.

La función **do_nop** atiende la petición devolviendo diversos códigos en función del tipo de petición.

```
PUBLIC int do_nop(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Nada ahí, nada que hacer */

  switch (m_ptr->m_type) {
  case DEV_OPEN:    return(ENODEV);
  case DEV_CLOSE:   return(OK);
  case DEV_IOCTL:   return(ENOTTY);
  default:          return(EIO);
  }
}
```

Código 11.- Función do_nop.

Tanto **nop_finish**, como **nop_cleanup**, son rutinas ficticias similares para dispositivos que no requieren los servicios ***dr_finish** ni de ***dr_cleanup**.

```
PUBLIC int nop_finish()
{
/* Nada que terminar, dp→dr_schedule hizo todo el trabajo */
  return(OK);
}


PUBLIC void nop_cleanup()
{
/* Nada que asear */
}
```

Código 12.- Funciones nop_finish y nop_cleanup.

Algunas funciones de dispositivos de disco requieren retaso; por ejemplo, para esperar que el motor de una unidad de disquete alcance la velocidad de operación. Así, **clock_mess**, envía mensajes a la tarea de reloj invocándose con el número de tics de reloj que hay que esperar y la dirección de la función que debe invocarse cuando haya transcurrido el plazo.

```
PUBLIC void clock_mess(ticks, func)
int ticks;                          /* numero de ticks en operación con reloj */
watchdog_t func;                    /* función que se llama después del time out */
{
/* envía a la tarea de reloj un mensaje */

  message mess;

  mess.m_type = SET_ALARM;
  mess.CLOCK_PROC_NR = proc_number(proc_ptr);
  mess.DELTA_TICKS = (long) ticks;
  mess.FUNC_TO_CALL = (sighandler_t) func;
  sendrec(CLOCK, &mess);
}
```

Código 13.- Función clock_mess.

La última función de DRIVER.C es **do_diocntl**, que lleva a cabo peticiones DEV_IOCTL para un dispositivo por bloque. Las operaciones que se pueden solicitar son DIOGETP (leer) y DIOSETP (escribir) información de particiones. La función invoca a su vez a ***dr_prepare** para verificar que el dispositivo es válido y obtener un apuntador a la estructura **device** que describe la base y el tamaño de las particiones en bytes. En una lectura, se invoca además a ***dr_geometry** para obtener la información de cilindro, cabeza y sector de la partición. El código de esta función es el siguiente:

```
do_diocntl(dp,m_ptr)
struct driver *dp;              /* puntero a tabla de funciones */
message *m_ptr;                 /* puntero a mensaje de lectura o escritura, el campo REQUEST nos
                                   indica si se va a realizar una lectura o escritura (DIOCGETP o
                                   DIOCSETP), el campo ADDRESS es un puntero a una estructura
                                   partition */
{
  struct device *dv;
  phys_bytes user_phys, entry_phys;
  struct partition entry;

  /* Comprueba que el campo REQUEST tiene una operación permitida */
  if (m_ptr→REQUEST != DIOCSETP && m_ptr→REQUEST != DIOCGETP)
        return(ENOTTY);

  /* dr_prepare devuelve la estructura device asociada al dispositivo o partición */
  if ((dv = (*dp→dr_prepare)(m_ptr→DEVICE)) == NIL_DEV) return(ENXIO);

/* Obtiene la dirección física del buffer del procedimiento de usuario donde se va a realizar la
operación de lectura o escritura*/
user_phys = numap(m_ptr→PROC_NR, (vir_bytes) m_ptr→ ADDRESS, sizeof(entry));
```

```
/* Obtiene la dirección física en kernel de entry (estructura partition) */
if (user_phys == 0) return(EFAULT);
  entry_phys = vir2phys(&entry);

 if (m_ptr→ REQUEST == DIOCSETP) {
   /* En caso de escritura  se copia la estructura partition del buffer (espacio de usuario) en  el kernel
y se actualiza la estructura device.*/

   phys_copy(user_phys, entry_phys, (phys_bytes) sizeof(entry));
   dv→ dv_base = entry.base;
   dv→ dv_size = entry.size;
  }
  else {
   /* En caso de lectura  se lee la estructura device del dispositivo y se llama a la función geometry
para obtener la geometría del dispositivo, la cual será guardada en la dirección del buffer dentro del
espacio del procedimiento de usuario */

   entry.base = dv→ dv_base;
   entry.size = dv→ dv_size;
   (*dp→ dr_geometry)(&entry);
   phys_copy(entry_phys, user_phys, (phys_bytes) sizeof(entry));
 }
  return(OK);
}
```

Código 14.- Función do_diocntl.

La estructura **partition** utilizada en el código anterior se encuentra en **minix/partition.h**, y es la siguiente:

```
struct partition {
      u32_t base;             /* desplazamiento en bytes de la base de la partición */
      u32_t size;             /* numero de bytes de la partición */
      unsigned cylinders;     /* geometría de la partición */
      unsigned heads ;
      unsigned sectors;
};
```

Código 15.- Estructura partition.

4.- BIBLIOTECA DE CONTROLADORES.

Los archivos DRVLIB.H Y DRVLIB.C contienen código dependiente del sistema que maneja las particiones de disco en computadoras compatibles con IBM PC.

Minix soporta particiones en disco duro y en floppy, en un disco duro incluso una partición puede tener subparticiones.

Debido a que un dispositivo puede estar ocupado por varios S.O. diferentes es necesario que todos estos compartan un mismo formato de la tabla de particiones. Minix, Linux, OS/2 usan formatos que pueden coexistir con MS-DOS. El hecho de que el código fuente de Minix que apoya a las particiones se coloca en DRVLIB.C facilita el traslado de Minix a hardware distinto.

4.1.- DRVLIB.H

La estructura de datos básica se define en **include/ibm/partition.h**, y contiene información sobre la geometría cilindro-cabeza-sector de cada partición, así como códigos que identifican el tipo de sistema de archivos de la partición y una bandera activa si es arrancable.

```
struct part_entry {
        unsigned char   booting ;       /* boot indicador */
        unsigned char   start_head ;    /* cabeza del primer sector*/
        unsigned char start_sec ;       /* primer sector */
        unsigned char start_cyl;        /* cilindro del primer sector */
        unsigned char  sysind;          /* indicador de sistema */
        unsigned char last_head ;       /* ultima cabeza*/
        unsigned char last_sec;         /* ultimo sector */
        unsigned char last_cyl;         /* ultimo cilindro */
        unsigned char low_sec;          /* primer sector logico */
        unsigned char size;             /* tamaño de particion en sectores*/
};
```

Código 16.- Estructura part_entry.

4.2.- DRVLIB.C

La función **partition** se invoca cuando se abre por primera vez (OPEN) un dispositivo por bloque para inicializar su estructura **device** y sus particiones. Además comprueba que la tabla de particiones es coherente (el tamaño de sus particiones es mayor que 0 y no sobre pasa los límites del dispositivo).

Sus argumentos incluyen una estructura **driver**, para que pueda invocar funciones especificas del dispositivo, un número de dispositivo secundario inicial y un parámetro que indica si el estilo de partición es disco flexible, partición primario o subpartición. En caso de que el tipo de partición sea primaria se ordena la tabla de particiones, ya que puede ser compartida por otros sistemas operativos en los que es necesario.

```
PUBLIC void partition(dp, device, style)
struct driver *dp;              /* puntero a tabla de funciones */
int device;                     /* device de particion (En Minix cada dispositivo y cada partición
                                tienen un identificador */
int style;                      /* tipo partición: floppy, primary, sub. (En Minix cada partición
                                primaria del disco duro puede ser particionada en subparticiones */
{
 struct part_entry table[NR_PARTITIONS], *pe;   /* en table se va a leer la tabla de particiones */
 int disk, par;
 struct device *dv;
 unsigned long base, limit, part_limit;

 /* Prepare devuelve un puntero a la base y el tamaño de la partición */
 if ((dv = (*dp→ dr_prepare)(device)) == NIL_DEV || dv→ dv_size == 0) return;

/* Se haya la base y el límite en sectores del dispositivo */
 base = dv→ dv_base >> SECTOR_SHIFT;
 limit = base + (dv→ dv_size >> SECTOR_SHIFT);

 /* Lee la tabla de particiones y la guarda en table */
 if (!get_part_table(dp, device, 0L, table)) return;

 /* Obtiene el identificador de la primera partición del dispositivo */
 switch (style) {
  case P_FLOPPY:
        device += MINOR_fd0a;
        break;
  case P_PRIMARY:
        sort(table);     /*En caso de que se trate de una partición primaria se ordena la tabla */
        device += 1;
        break;
```

```
 case P_SUB:
        disk = device / DEV_PER_DRIVE;
        par = device % DEV_PER_DRIVE - 1;
        device = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
 }

  /* Se llama a la función dr_prepare con el identificador calculado, devolviendo un puntero a un
vector de estructuras device con una entrada por cada partición del dispositivo */

  if ((dv = (*dp→ dr_prepare)(device)) == NIL_DEV) return;

  /* Establecer geometría de particiones con base en la tabla */

  for (par = 0; par < NR_PARTITIONS; par++, dv++) {
    /* Encoger para que quepa en dispositivo */
    pe = &table[par];
    part_limit = pe→ lowsec + pe→ size;       /*se comprueba que el tamaño de la partición>0 */
    if (part_limit < pe→ lowsec) part_limit = limit;   /*y no sobrepasa los limites del dispositivo */
    if (part_limit > limit) part_limit = limit;
    if (pe->lowsec < base) pe→ lowsec = base;
    if (part_limit < pe→ lowsec) part_limit = pe→ lowsec;

    /* Transforma sectores en bytes e inicializa la estructura device asociada a la partición */
    dv→ dv_base = pe→ lowsec << SECTOR_SHIFT;
    dv→ dv_size = (part_limit – pe→ lowsec) << SECTOR_SHIFT;

    if (style == P_PRIMARY){
      if (pe→ sysind == MINIX_PART)          /*en caso de particion minix (puede subdividirse ) */
          partition(dp, device + par, P_SUB);  /*llamada recursiva para subparticiones */
      if (pe→ sysind == EXT_PART)            /*en caso de particion no minix ( una partición */
          extpartition(dp, device + par, pe→ lowsec); /* extendida tiene particiones lógicas ) */
    }
  }
}
```

Código 17.- Función partition.

Como se puede ver en el código anterior se invoca a la función ***dr_prepare** específica para el dispositivo a fin de verificar que es válido, y colocar la dirección base y el tamaño en una estructura **device**. Luego se calcula la base y el tamaño del dispositivo en sectores a partir de la información anterior. Posteriormente se invoca a **get_part_table** para determinar si está presente una tabla de particiones y leerla. Si existe tabla de particiones se calcula el identificador de la primera partición usando las reglas de numeración de dispositivo secundario que apliquen al estilo de partición especificada en la llamada original. En el caso de particiones primarias la tabla está ordenada de modo que sea congruente con el que usan otros sistemas operativos.

Se invoca otra vez ***dr_prepare** usando el número de dispositivo calculado para la primera partición. La función devolverá un vector de estructuras **device**, con un elemento por cada partición. Si el subdispositivo es válido, se procesarán cíclicamente todas las entradas de la tabla, comprobando que los valores leídos de la tabla en el dispositivo no estén fuera del intervalo calculado anteriormente para la base y el tamaño de todo el dispositivo; en caso contrario la tabla en memoria se ajusta de modo que sea congruente.

Si la partición se identifica como Minix, se invoca **partition** recursivamente para realizar la misma operación con las subparticiones. Si es una partición extendida se invoca a **extpartition**.

La función extpartition no la vamos a ver muy a fondo porque no tiene mucho que ver con el Minix, más bien tiene que ver con la forma en que MS-DOS organiza y maneja sus subparticiones o particiones extendidas.

De cualquier forma el código de la función es el siguiente:

```
PRIVATE void extpartition(dp, extdev, extbase)
struct driver *dp;              /* Puntos de entrada dependientes del dispositivo */
int extdev;                     /* Partición extendida que explorar */
unsigned long extbase;          /* Distancia sector de la base de la partición extendida */
{
 /* Las particiones extendiadas no pueden ignorarse porque la gente gusta de mover archivos entre
particiones de DOS */
  struct part_entry table[NR_PARTITIONS], *pe;
  int subdev, disk, par;
  struct device *dv;
  unsigned long offset, nextoffset;

  disk = extdev / DEV_PER_DRIVE;
  par = extdev % DEV_PER_DRIVE - 1;

  /*obtiene identificador  de subparticion*/
  subdev = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
  offset = 0;

  do {
        if (!get_part_table(dp, extdev, offset, table)) return;
        sort(table);

        /* La tabla debe contener una partición lógica y opcionalmente otra extendida ( es una lista
        enlazada ) */
        nextoffset = 0;
```

```
        for (par = 0; par < NR_PARTITIONS; par++) {
                pe = &table[par];
                if (pe→sysind == EXT_PART) {
                        nextoffset = pe->lowsec;
                } else
                if (pe→sysind != NO_PART) {
                        if ((dv = (*dp→dr_prepare)(subdev)) == NIL_DEV) return;

                        dv→dv_base = (extbase + offset
                                        + pe→owsec) << SECTOR_SHIFT;
                        dv→dv_size = pe→size << SECTOR_SHIFT;

                        /* ¿ No más dispositivos ? */
                        if (++subdev % NR_PARTITIONS == 0) return;
                }
        }
  } while ((offset = nextoffset) != 0);
}
```

Código 18.- Función extpartition.

La función **get_part_table** lee la tabla de particiones de un dispositivo y devuelve TRUE si no hubo errores. Primero se construye un mensaje de solicitud de lectura del sector donde se encuentra la tabla. Luego se invoca a la función **do_rdwt** para obtener el sector del dispositivo o subdispositivo donde se encuentra la tabla. El desplazamiento que se incluye como argumento es 0 si la invocación es para obtener una partición primaria, y distinto de 0 en el caso de una subpartición. La función busca el número mágico (0xAA55) y devuelve un estado de verdadero o falso para indicar si se encontró o no una tabla de particiones válida. Si se encuentra la tabla, se copia en la dirección de la que se pasó como argumento.

```
PRIVATE int get_part_table(dp, device, offset, table)
struct driver *dp;
int device;
unsigned long offset;           /* sector en donde esta la tabla */
struct part_entry *table;       /* variable de salida que contendrá la tabla */
{
  message mess;         /* Construye mensaje de solicitud para leer el sector donde está la tabla */
  mess.DEVICE = device;
  mess.POSITION = offset << SECTOR_SHIFT;      /*transforma sectores en bytes */
  mess.COUNT = SECTOR_SIZE;
  mess.ADDRESS = (char *) tmp_buf;
```

```
  mess.PROC_NR = proc_number(proc_ptr);
  mess.m_type = DEV_READ;

  if (do_rdwt(dp, &mess) != SECTOR_SIZE) {  /* do_rdwt realiza la operación de lectura */
        printf("%s: can't read partition table\n", (*dp→ dr_name)());
        return 0;
  }

  /* Chequea que en determinada posición se encuentre un cierto código */
  if (tmp_buf[510] != 0x55 || tmp_buf[511] != 0xAA) {
    /* Tabla de particiones no válida */
        return 0;
  }

  /* Se copia la tabla en la variable de salida table */
  memcpy(table, (tmp_buf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
  return 1;
}
```

Código 19.- Función get_part_table.

Por último, la función **sort** ordena las entradas de una tabla de particiones en orden ascendente por sector. Las entradas marcadas como carentes de particiones se excluyen del ordenamiento, que es de tipo burbuja.

```
PRIVATE void sort(table)
struct part_entry *table;
{
/* Ordenar una tabla de particiones */
  struct part_entry *pe, tmp;
  int n = NR_PARTITIONS;

  do {
        for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
              if (pe[0].sysind == NO_PART
                    || (pe[0].lowsec > pe[1].lowsec
                                && pe[1].sysind != NO_PART)) {
                    tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
              }
        }
  } while (--n > 0);
}
```

Código 20.- Función sort.