Driver para el chip DSP de la tarjeta SB16

Paqui Martín

José Juan Mendoza Rodríguez

Contenido **Página**

1. El chip DSP

1.1. El muestreo (Sampling)

Se llama muestreo o sampling a la técnica mediante la cual se graban señales analógicas desde un CD, un equipo estéreo o a través de un micrófono, con la mayor calidad posible, se guardan en un soporte (disco duro) y se reproducen posteriormente sin pérdidas de calidad.

Básicamente, en el muestreo sólo se realizan conversiones analógico/digitales o digital/analógicas. Al grabar, la tarjeta de sonido toma la señal de sonido analógica y la transforma en 'muestras' digitales. En la reproducción, se toman las señales digitales de las muestras almacenadas y se convierten en una señal analógica, para ser escuchadas.

La coincidencia de la señal de sonido analógica/digital con la señal analógica original depende de la llamada frecuencia de muestreo y de su resolución. No es posible tomar infinitas muestras en el tiempo de la señal original, porque el almacenamiento también sería infinito; por lo tanto, sólo se tomas un número determinado de muestras por segundo. Cuanto mayor es el número de muestras tomadas en este intervalo de tiempo, mayor será la calidad de la muestra. La calidad depende también la resolución de las muestras, que representa el número de sonidos distintos representable.; será mejor cuanto mayor sea el número de bits en la resolución.

La grabación y reproducción es posible debido a los procesadores digitales de sonido, que se explican a continuación.

1.2. DSP (Digital Sound Processor)

El proceso de muestreo, como se acaba de indicar, es posible gracias a la programación del Procesador de Sonido Digital (DSP), que se encuentra en la tarjeta SoundBlaster.

Existen diferentes versiones del DSP, que indican la versión de la tarjeta SoundBlaster instalada. Los principales criterios que identifican las distintas versiones de la DSP son dos:

1) La velocidad, o número de muestras que se pueden grabar o reproducir por segundo.

2) La profundidad de los bits, que definen la resolución de la imagen de la señal analógica original y su representación digital. Cuanto más alta es la resolución, la grabación será más precisa y la calidad de sonido será mayor.

La medida de estos parámetros define la calidad CD.

1.3. Modo de Transferencia DSP

Existen dos modos básicos de muestreo. Uno de ellos utiliza el controlador DMA para llevar a cabo la transferencia de datos entre la memoria principal y el DSP, y el otro utiliza el procedimiento Polling. En este caso, se explicará únicamente el modo que utiliza la versión de Minix que es objeto de estudio.

La versión Minix 2.0.0 utiliza el controlador DMA para la transferencia de información entre memoria y DSP.

Para cada bloque a transferir tienen que programarse separadamente el controlador DMA y el DSP. Debido a las limitaciones del controlador DMA, un bloque puede abarcar un máximo de 64KB (así el controlador DMA entiende que la memoria está estructurada en particiones de 64KB). Además, todo bloque tiene que poder ser ubicado dentro de una página de memoria de 64KB, por lo que no puede estar en una zona de memoria próxima al final de una partición de forma que su tamaño sea tal que ocupe el principio de otra partición.

La información a transferir está fragmentada en bloques de igual tamaño.

El final de una transmisión lo indica el DSP mediante la ejecución de una interrupción.

El DSP puede procesar muestras de 8 ó 16 bits.

1.4. Interacción entre el controlador DMA y el DSP

Antes de explicar estas interacciones, se indicarán los parámetros de los que depende la secuencia exacta que realiza cada muestra. Así, se señaliza que las muestras:

a) Pueden estar grabadas en mono o en estéreo.

b) Pueden estar compuestas de 8 ó 16 bits.

c) Pueden ser con signo o sin signo. Las de 8 bits siempre son sin signo. Sin signo quiere decir que las muestras se grabarán siempre en forma de números positivos (en caso contrario, tendrá números positivos y negativos). Esto influye únicamente en el rango de valores que toma la estructura.

En la reproducción de muestras, el controlador DMA transfiere los datos de muestras a la CPU, pasando por un buffer de la memoria principal al DSP. El controlador DSP la convierte en señales analógicas y las traslada al amplificador de la tarjeta de la SoundBlaster. Entonces, el usuario ya puede recibirla (a través de auriculares, amplificadores...).

En el caso de la grabación, el DSP toma las señales analógicas de un fuente de sonido (un micrófono, CD...), lo convierte en una muestra digital y lo transfiere a un buffer en la memoria principal, por medio del controlador DMA.

Esta interacción se regula a través del software. Por tanto, lo primero que hay que hacer es configurar los drivers. Primero se configura el DMA, a través del canal que éste tiene conectado a la tarjeta SoundBlaster. Para ello se configura la dirección inicial del buffer de transferencia y la longitud de transferencia.

Posteriormente, se configura el DSP. Los parámetros a configurar en este procesador son la frecuencia de muestreo y el número de muestras que deben ser leídas y reproducidas desde la memoria o desde una fuente de sonido analógica. Una vez realizadas las configuraciones, el DSP recibe un comando que le envía el software de muestreo, indicándole que ya puede comenzar con la grabación o la reproducción de las muestras.

Antes de escribir o leer en cualquier puerto, hay que asegurarse de que éste está vacío, porque si no, significa que el DSP no ha procesando completamente el último comando recibido. Los registros de estado (Write Buffer Status y Read Buffer Status) son los que indican si están vacíos los registros de escritura y lectura. La forma de averiguarlo es leyendo el bit 7 de estos registros.

La escritura en el DSP se realiza a través del registro Write Data, y la lectura a través del registro Read Data.

La programación del DSP debe estar siempre precedida por un reset o reinicialización del chip. Se realiza a través del puerto indicado en la tabla.

Los puertos correspondientes a los registros anteriormente citados aparecen en la tabla siguiente:

Puerto	Nombre	Leer	Escribir	Tarea
+06h	Reset		*	Reinicialización del DSP
+0Ah	Read Data	*		Lectura de datos del DSP
+0Ch	Write Command/Data		*	Salida de datos y comandos
+0Ch	Write Buffer Status	*		Indica si el DSP está listo
+0Eh	Read Buffer Status	*		Indica si hay datos para leer

Tabla 1. Distintos puertos DSP.

Otros parámetros a tener en cuenta son:

a) La longitud de transferencia. Para que el DSP pueda saber cuántas muestras leer o reproducir antes de que termine su trabajo y ejecute la interrupción.

b) Activar y desactivar el altavoz. Si se quieren reproducir muestras, se activa (comando D3h). Si se quieren grabar o leer datos de muestras, se desactivan (comando D1h).

c) Frecuencia de muestreo deseada (comandos 41h y 42h), que incide en la calidad del sonido.

2. La E/S en Minix 2

El sistema Minix 2.0.0 está estructurado en cuatro capas o niveles de protección de memoria como se muestra en la figura 1.

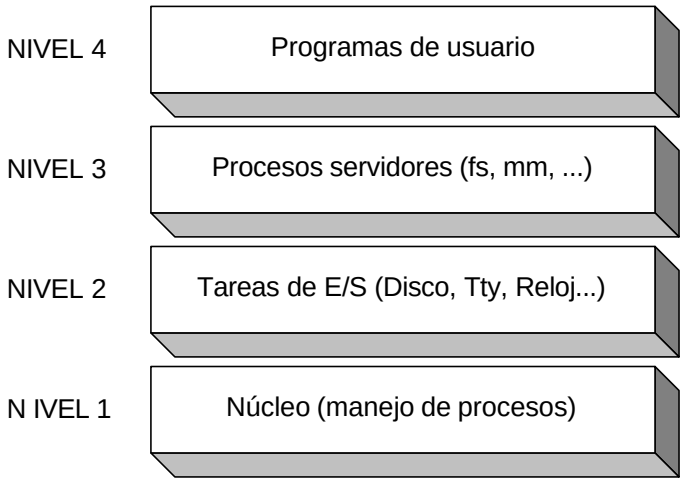

figura 1

El nivel 4 corresponde a los procesos que pone en marcha el usuario, y tiene el nivel más bajo de prioridad en el planificador de la CPU (figura 2), el cual mantiene colas de procesos separadas para cada nivel de prioridad. Los procesos del nivel 4 se planifican mediante un esquema *round robin*.

En el nivel 3 están los procesos servidores de memoria (MM) y del sistema de ficheros (FS), entre otros. Es éste último el que nos interesa ahora, ya que toda la entrada/salida se canaliza a través de peticiones al sistema de ficheros. El nivel 2 corresponde a los procesos que se encargan directamente de la entrada/salida (los *drivers*), llamados normalmente *tareas* de E/S. Entre ellos está la tarea DSP (ó `dsp_task`, usando el identificador que aparece en el fichero fuente `sb16_dsp.c`). A los procesos del nivel 3 y del nivel 2, que tienen más prioridad que el nivel 4, no se les «roba el ciclo»: se ejecutan hasta que terminan lo que tienen que hacer y se bloquean (esperando por un mensaje del nivel superior). Primero, el planificador ejecuta cada tarea en el orden en que aparecen en la cola de tareas hasta que se bloquean. Luego trata de ejecutar MM o FS si alguno de ellos está listo. Finalmente, si todos los procesos de los niveles 3 y 4 están bloqueados, entonces le pasa el control al siguiente proceso de usuario que esté listo en la cola del *round robin*.

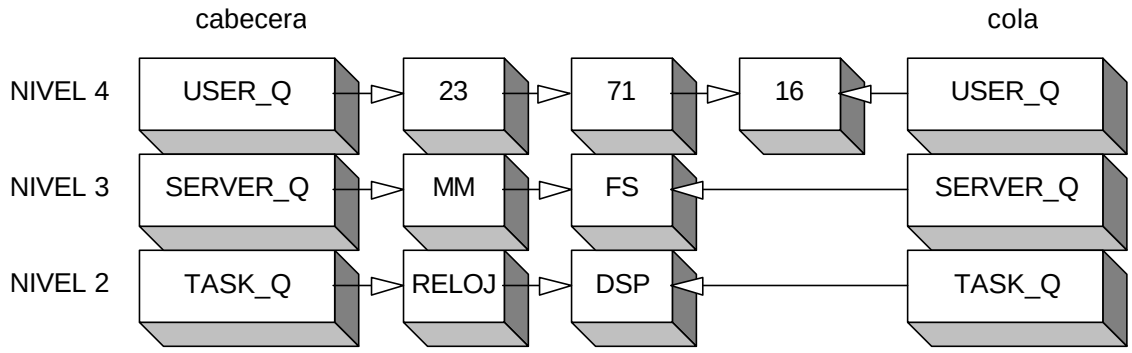

figura 2

El nivel 1 no tiene nada parecido a una cola de procesos. En el nivel 1 se encuentra el propio planificador de la CPU, el cual está «disparado» por las interrupciones del reloj, y el resto del *software* del núcleo, como los manejadores de interrupciones (*interrupt handlers*); entre ellos, el que nos ocupa es `dsp_handler()`, que procesa las interrupciones provenientes del chip DSP.

El proceso de una llamada de E/S en Minix 2.0.0 se resume en la figura 3, aunque los detalles pueden variar según el dispositivo. En general, el programa de usuario invoca al sistema de ficheros (proceso FS) mediante una llamada al sistema del tipo `write`, `read`, `open`, `close` o `ioctl`. El servidor sistema de ficheros se encarga de proporcionar al nivel superior un interfaz homogéneo de manera que el programador no tenga que preocuparse por los detalles concretos del dispositivo que está utilizando. En concreto, el servidor del sistema de ficheros determinará el tipo y el número identificador del dispositivo invocado y fragmentará las operaciones de lectura/escritura en caso de que sea necesario. Los datos concretos de la operación que se desea realizar se introducen en la estructura `message` y se envía un mensaje a la tarea de E/S correspondiente. Luego, el servidor FS se bloquea esperando por el mensaje de respuesta.

Una tarea de entrada salida típica tiene este esquema:

```
message mess;
void io_task() {
      int caller, err;
      initialize();     /* esto se hace sólo una vez, durante la ... */
```

```
        while (1) {       /* inicialización del sistema */
              receive(ANY, &mess);  /* esperamos la llamada del FS */
              caller = mess.m_source;
              switch (mess.type) {
                    case READ: { err = /* leer */ } break;
                    case WRITE: { err = /* escribir */ } break;
                    case OPEN: { err = /* abrir dispositivo */ } break;
                    case CLOSE: { err = /* cerrar dispositivo */ } break;
                    case IOCTL: { err = /* control del disp. */ } break;
                    default: err = ERROR;
              }
              mess.type = TASK_REPLY;
              mess.status = err;
              send(caller, &mess);
        }
  }
```

Básicamente, la tarea de E/S está bloqueada esperando por un mensaje del servidor FS. Cuando el mensaje llega, invoca la operación expresada en la estructura `message` y luego envía al servidor FS un mensaje de respuesta con el status de la operación que se ha efectuado. Es posible que algún controlador de interrupciones intervenga en la operación efectuada (como en el caso del chip DSP), pero esto dependerá del tipo de dispositivo.

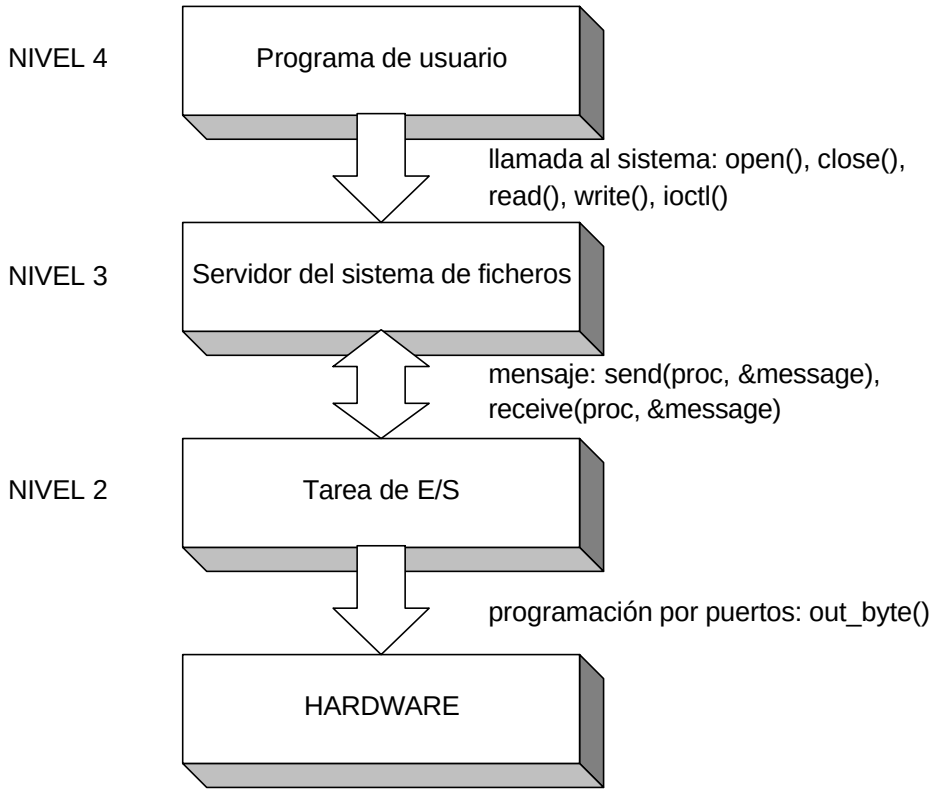

figura 3

En la figura 4 se indica el proceso de una llamada de E/S para el caso del controlador DSP. La tarea `dsp_task` atiende exactamente las cuatro peticiones típicas que se muestran en la figura: `write`, `read`, `open`, `close` o `ioctl`, aunque durante el proceso de algunas de ellas (luego veremos) no toca el dispositivo DSP para nada, tan sólo actualiza sus propias variables. La tarea no sólo programa el chip DSP, sino que también utiliza el dispositivo DMA para las transferencias entre la memoria y el DSP. Al igual que ocurre con otros dispositivos de lectura/escritura que usan el DMA, se utiliza alguna línea de interrupción del procesador iAPx86 (en este caso la IRQ5) para señalizar el final de la transferencia. La subrutina `dsp_handler`, contenida también en el módulo `sb16_dsp.c` implementa el manejador de esta interrupción. Luego veremos estas cuestiones en detalle.

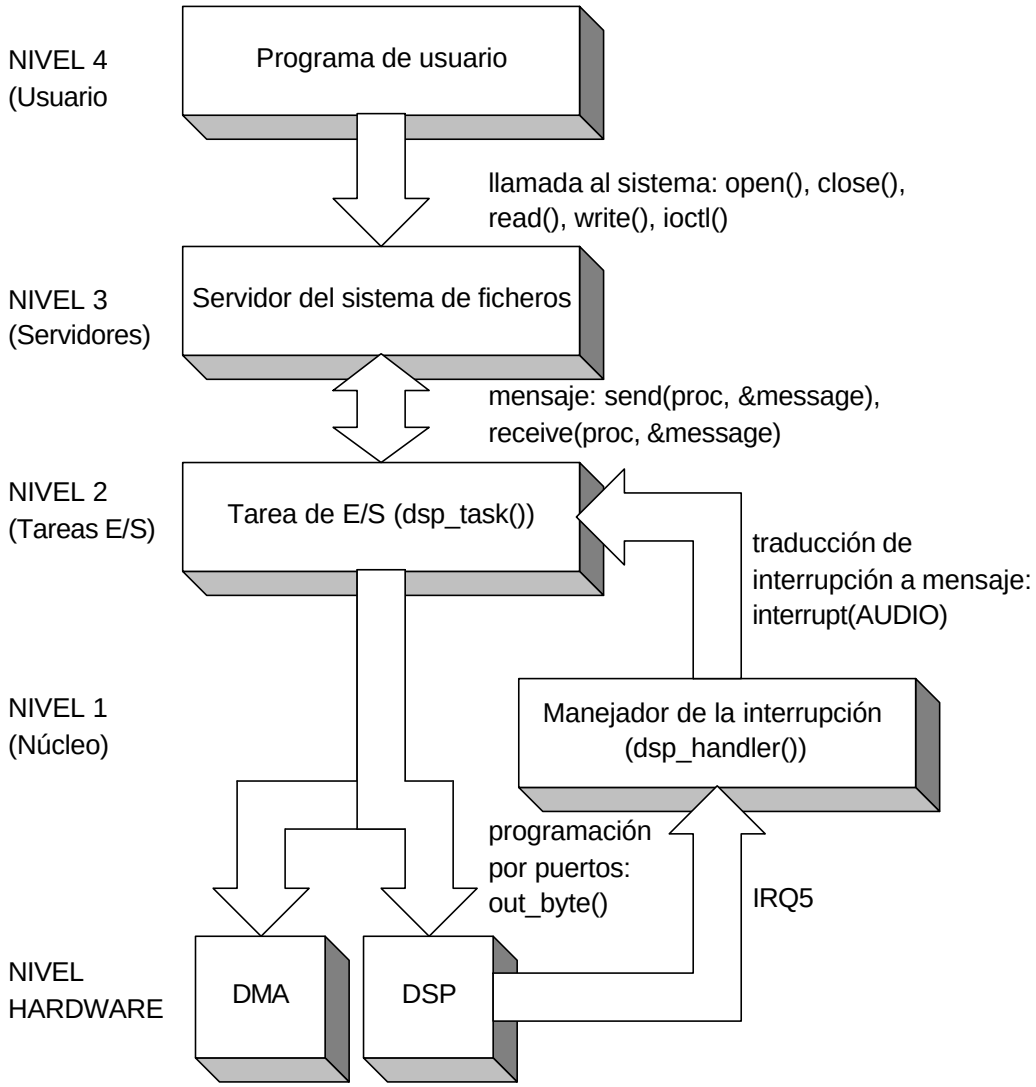

figura 4

3. El driver DSP

A continuación describimos en detalle las funciones que componen el driver DSP del Minix 2.0.0.

3.1. dsp_task()

La función dsp_task implementa el driver del chip DSP de la tarjeta de sonido Sound Blaster 16 (figura 5). Este driver es un proceso de E/S del nivel 1 de Minix. La función comienza llamando a init_buffer() para inicializar la posición del buffer de DMA. Esto se ejecuta una sola vez al iniciarse el sistema. Luego ejecuta un bucle infinito, al comienzo del cual se bloquea esperando por la llegada de un mensaje. Los mensajes pueden provenir del servidor del sistema de ficheros (una tarea del nivel superior) o del manejador de interrupciones (implementado por la función dsp_handler); este último tipo de mensaje simplemente se acepta y se ignora, ya que se trata de una interrupción residual correspondiente a la finalización de transferencia del último bloque de una serie de bloques de lectura/escritura (ver dsp_read(), dsp_write()). Los mensajes del sistema de ficheros provienen de las llamadas al sistema que invocan los programas de usuario, y que el servidor del sistema de ficheros se encarga de transmitir a la tarea dsp_task mediante el paso de mensajes. Estos mensajes son:

DEV_OPEN correspondiente a la llamada open()
DEV_CLOSE " " " " close()
DEV_IOCTL " " " " ioctl()
DEV_READ " " " " read()
DEV_WRITE " " " " write()

La función dsp_task() detecta el tipo de mensaje mediante una sentencia *switch* e invoca a la función correspondiente que implementa la operación de E/S. Estas funciones son: dps_open(), dsp_close(), dsp_ioctl(), dsp_read() y dsp_write().

Finalmente, la función prepara y envía un mensaje de respuesta al servidor del sistema de ficheros, en el cual se especifica el número de bytes transferidos (leídos o escritos en el dispositivo DSP) o un código de error en caso de que la operación se haya efectuado de manera errónea.

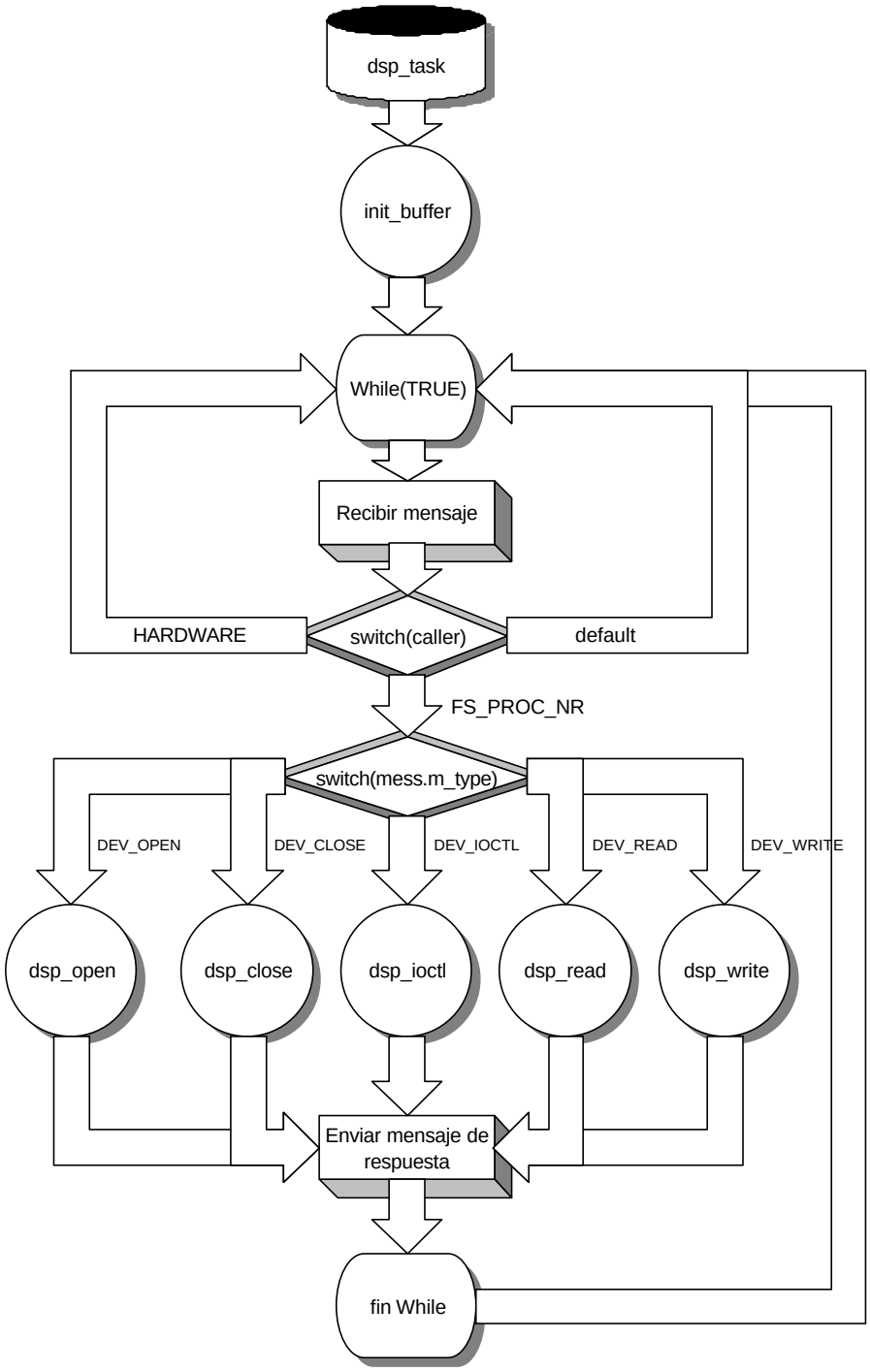

figura 5

3.2. dsp_open()

La función dsp_open() se invoca en respuesta a un mensaje DEV_OPEN enviado por el servidor del sistema de ficheros el cual, a su vez, ha sido originado por una llamada open(). La función mira

primero si efectivamente existe tarjeta en la placa. Para ello lee la variable DspAvail y, si esta variable está a 0, entonces intenta inicializar la tarjeta llamando a la función dsp_init(). La función dsp_init coloca DspAvail a 1 si la inicialización tuvo éxito (esto garantiza que la tarjeta sólo se inicialice la primera vez que se invoca dsp_open). Si la inicialización no tiene éxito, entonces regresa inmediatamente devolviendo un código de error. Si la inicialización funciona, entonces mira si hay en curso una operación de E/S sobre el chip DSP (observando el flag DspBusy). Para ello inspecciona el contenido de la variable DspBusy. Si es así retorna inmediatamente, porque el driver DSP no admite operaciones concurrentes. Luego, reinicializa el chip DSP llamando a la función dsp_reset(). Finalmente, asigna las variables DspStereo, DspSpeed, DspBits, DspSign y DspFragmentSize a sus valores por defecto, colocando luego el flag DspBusy a 1 y el flag DmaBusy a 0.

3.3. dsp_close()

La función dsp_close() responde a un mensaje DEV_CLOSE, originado por una llamada close(). En realidad, esta función no interacciona con el chip DSP, sino que simplemente asigna 0 a los flags DspBusy y DmaBusy, indicando que tanto el dispositivo DMA como el chip DSP están libres. No hace nada más.

3.4. dsp_ioctl()

La función dsp_ioctl() responde a un mensaje DEV_IOCTL, originado por una llamada ioctl() (figura 6). Esta función se encarga de modificar los parámetros o reinicializar el chip DSP. En primer lugar, comprueba que el DMA no esté ocupado en una transferencia. Luego obtiene, de la estructura *message mess* el tipo y el valor del parámetro que vamos a asignar. Con estos datos, entra en una sentencia *switch* donde, según el tipo del parámetro, elige la función correspondiente que permite asignar el valor al parámetro. Estas opciones son:

DSPIORATE	dsp_set_speed()	selecciona velocidad de muestreo
DSPIOSTEREO	dsp_set_stereo()	habilita el modo estéreo
DSPIOBITS	dsp_set_bits()	selecciona el número de bits (la resolución)

DSPIOSIZE	dsp_set_size()	asigna el tamaño de la transmisión
DSPIOSIGN	dsp_set_sign()	indica si las muestras tienen o no signo
DSPIOMAX	-	devuelve el tamaño maximo del buffer DMA,
DSPIORESET	dsp_reset()	reinicializa el chip DSP

Al final devuelve el status que retorna la función invocada.

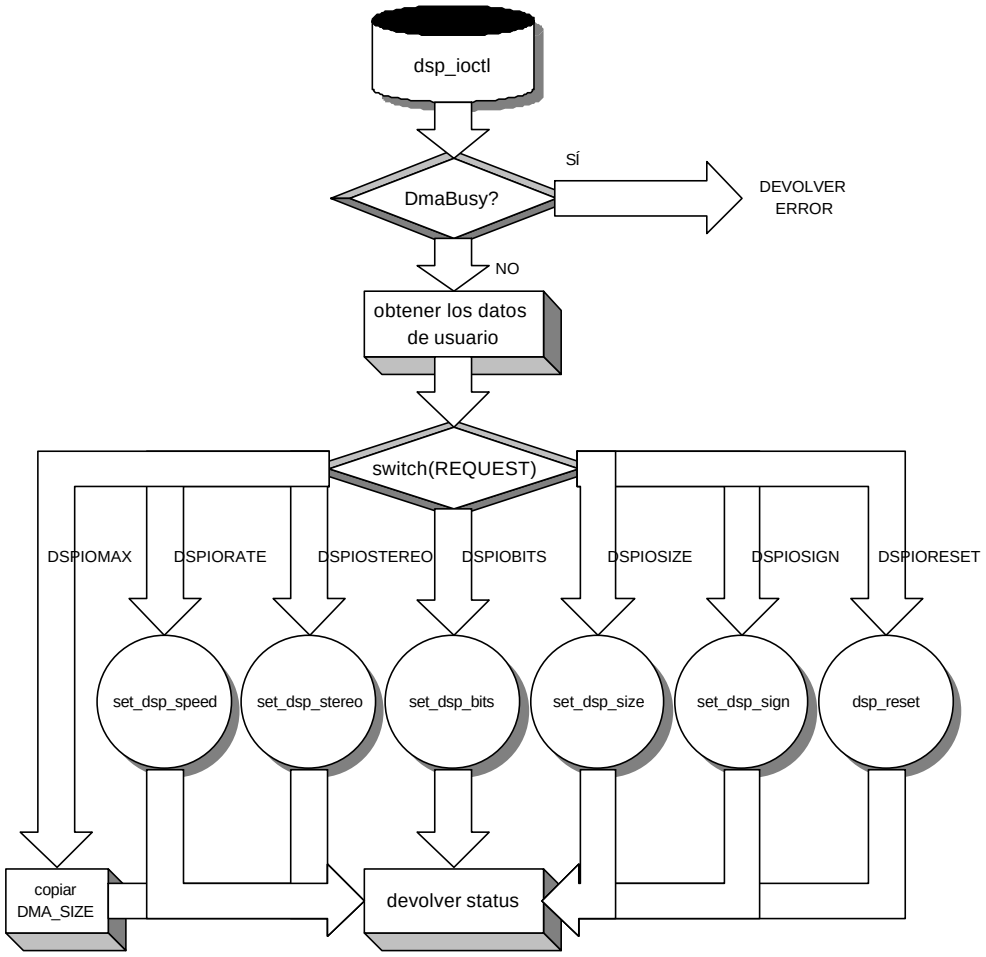

figura 6

3.5. dsp_write()

La función dsp_write() responde a un mensaje DEV_WRITE, originado por una llamada write() (figura 7). Esta función se encarga de transferir al chip DSP un sólo fragmento de datos de longitud constante. Se supone que el servidor del sistema de ficheros se encarga de fragmentar la información y,

si es necesario, rellenar el último fragmento, y de enviar varios mensajes al proceso dsp_task() para escribir cada fragmento.

En primer lugar, comprueba que el número de bytes contenidos en el mensaje sea igual al del tamaño del fragmento (DspFragmentSize). Luego obtiene la dirección física de los datos a transferir en el espacio de direcciones del programa de usuario. A continuación, el comportamiento de la función depende de si el bloque de datos a transferir es el primero de la serie o no. Esto se comprueba leyendo el valor del flag DmaBusy. Si este flag está activo, y el tipo de mensaje es DEV_WRITE, supone que no se trata del primer bloque de la serie. En caso contrario se trata del primero.

Si se trata del primer bloque de la serie, asigna el modo del DMA colocando DmaMode a DEV_WRITE, copia el bloque desde la memoria física del espacio de usuario al buffer de DMA, señalado por DmaPhys (se trata de la dirección física de DmaBuffer), inicializa el DMA llamando a dsp_dma_setup() y el chip DSP llamando a dsp_setup() y, finalmente, activa el flag DmaBusy, indicando que el DMA va a estar ocupado.

Si se trata del segundo o sucesivos bloques de la serie, espera por la interrupción que genera el DSP señalizando el final de la transferencia anterior y, cuando ésta llega copia en el buffer del DMA el siguiente fragmento de la serie.

En ambos casos, al final se devuelve el número de bytes transferidos.

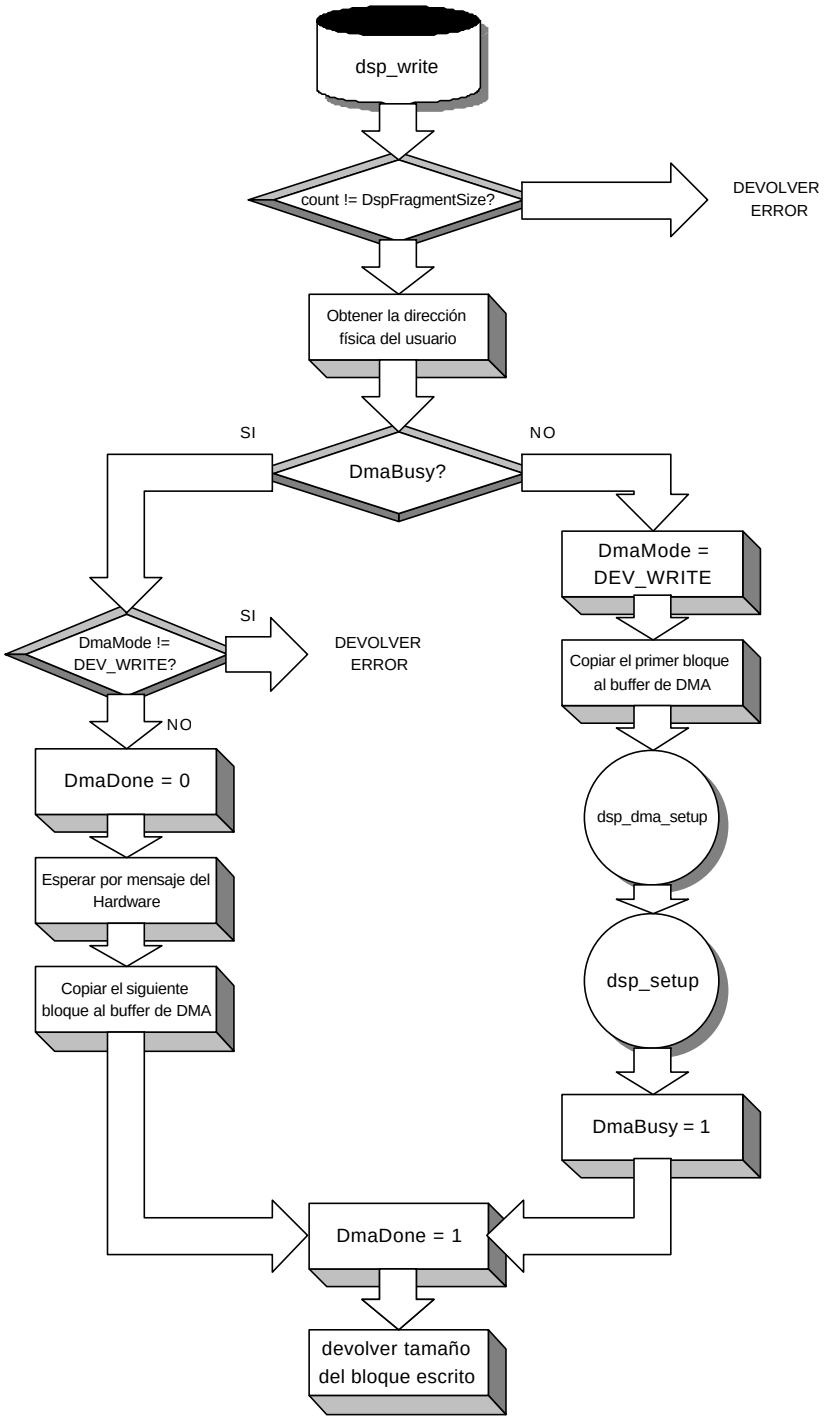

figura 7

3.6. dsp_read()

La función dsp_read() responde a un mensaje DEV_READ, originado por una llamada read() (figura 8). Esta función se encarga de leer del chip DSP un sólo fragmento de datos de longitud

constante. Se supone que el servidor del sistema de ficheros se encarga de fragmentar las peticiones de lectura, enviando varios mensajes al proceso dsp_task() para leer cada fragmento, y concatenar los fragmentos leídos.

En primer lugar, comprueba que el número de bytes contenidos en el mensaje sea igual al del tamaño del fragmento (DspFragmentSize). Luego obtiene la dirección física de los datos a transferir en el espacio de direcciones del programa de usuario. A continuación, el comportamiento de la función depende de si el bloque de datos a transferir es el primero de la serie o no. Esto se comprueba leyendo el valor del flag DmaBusy. Si este flag está activo, y el tipo de mensaje es DEV_READ, supone que no se trata del primer bloque de la serie. En caso contrario se trata del primero.

Si se trata del primer bloque de la serie, asigna el modo del DMA colocando DmaMode a DEV_READ, inicializa el DMA llamando a dsp_dma_setup() y el chip DSP llamando a dsp_setup(), activa el flag DmaBusy, indicando que el DMA va a estar ocupado, espera por el mensaje del manejador dsp_handler indicando la finalización de la transferencia de la lectura y, finalmente,copia el bloque desde el buffer de DMA, señalado por DmaPhys (se trata de la dirección física de DmaBuffer), a la memoria física del espacio de usuario.

Si se trata del segundo o sucesivos bloques de la serie, espera por la interrupción que genera el DSP señalizando el final de la transferencia anterior y, cuando ésta llega lee del buffer del DMA el siguiente fragmento de la serie.

En ambos casos, al final se devuelve el número de bytes transferidos.

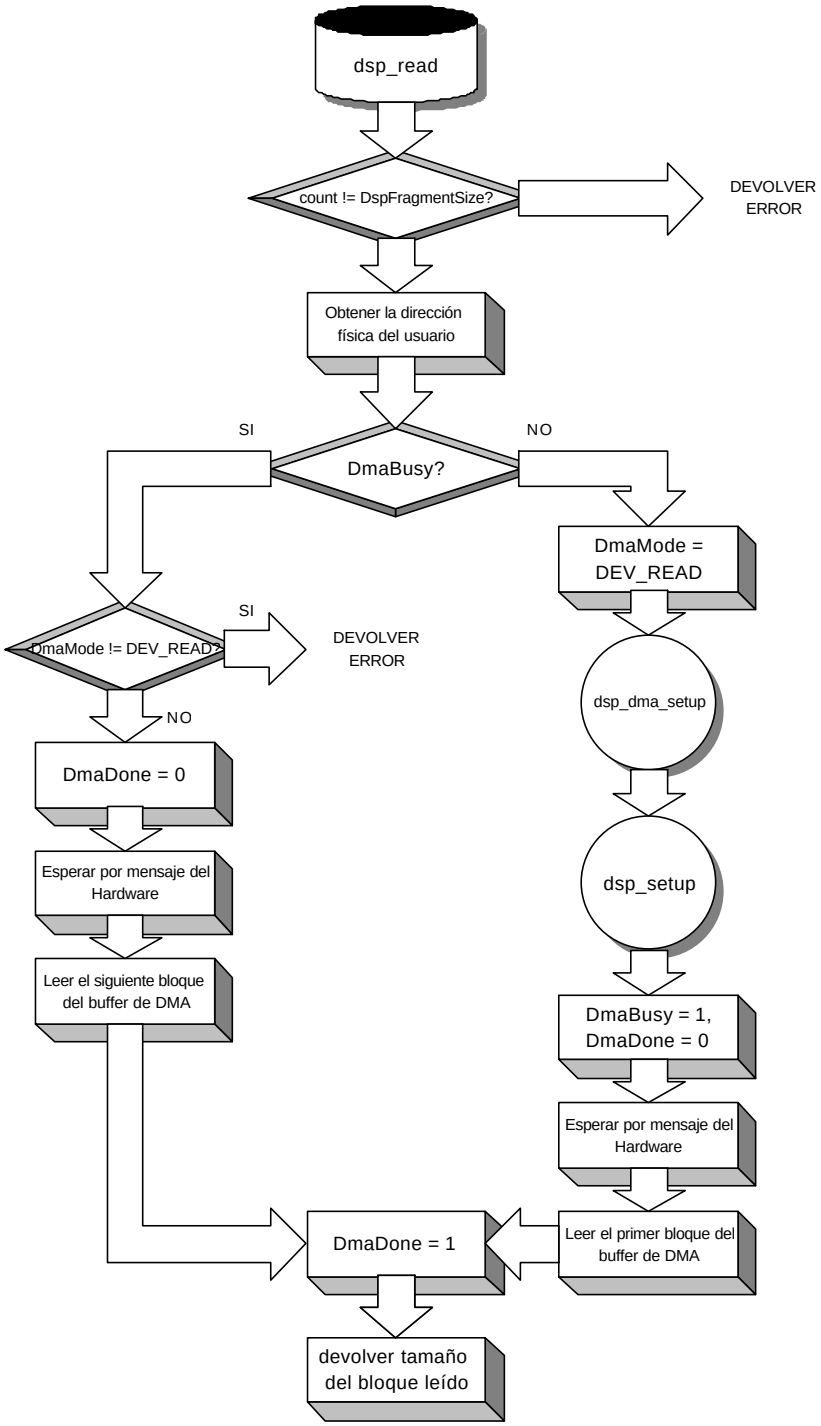

figura 8

3.7. init_buffer()

La función init_buffer() se invoca una sola vez, durante la inicialización del sistema. Se encarga de asegurar que el buffer del DMA no cruce la frontera del segmento de 64K. El buffer de DMA tiene un

tamaño de 32K, si la CPU es 386 o superior, o 8K, en caso contrario, y está definido en la macro DMA_SIZE. La función calcula la cantidad 64K - DMA_SIZE, que es lo que falta hasta el siguiente cruce de segmento; si este valor es menor que DMA_SIZE, entonces significa que el buffer cruza la frontera del segmento. En ese caso, se le suma DMA_SIZE para desplazarlo hacia arriba hasta el siguiente segmento.

3.8. dsp_init()

La función dsp_init se utiliza para detectar la existencia de la tarjeta de audio, identificarla y para inicializar algunos parámetros de hardware. Para detectar la tarjeta invoca la función dsp_reset. Para leer la versión de la tarjeta, utiliza un bucle que espera a que haya datos disponibles (testeando el bit 7 del registro Read Buffer Status Register). Cuando hay datos, los lee del registro Read Data Register. Luego mira si la versión de la tarjeta es correcta (el *driver* sólo puede manejar tarjetas de la versión 4.xx o posteriores).

Lo último que hace esta función es asignar los parámetros del mezclador, instalar el manejador de interrupciones dsp_handler, habilitar la interrupción IRQ5 y colocar el flag DspAvail, para que la inicialización de la tarjeta de audio no se repita la próxima vez que se invoque dsp_open.

3.9. dsp_handler

Esta función es el manejador de la interrupción IRQ5, la cual es utilizada por el chip DSP para señalizar el final de una transferencia de lectura/escritura. La función testea el valor del flag DmaDone para ver si la transferencia se ha completado. En ese caso, envía un comando para detener el DMA. Luego traduce la interrupción hardware en un mensaje para la tarea dsp_task. Este mensaje se recoge en las subrutinas dsp_write o dsp_read (si el último bloque transferido no fuera el último) o en la rutina dsp_task (si el bloque transferido es el último). Finalmente, lee un vector de interrupción *dummy* del registro Read Buffer Status Register para reconocer la interrupción.

3.10. dsp_command

La función dsp_command se utiliza para enviar comandos al chip DSP, escribiendo en su registro Write Command. Antes de escribir hay que esperar a que el DSP esté listo para recibir comandos. Esto se hace en un bucle que lee el Write Buffer Status y espera hasta que el bit 7 de este registro esté inactivo. Si se produce un *timeout* la función devuelve un código de error.

3.11. dsp_reset

Esta función se utiliza para resetear el chip DSP. El procedimiento es el siguiente. Primero se envía un 1 hacia el registro Reset del chip DSP. Luego hay que esperar al menos 3 segundos. Esto se hace mediante un simple bucle for. Luego se envía un 0 hacia el mismo registro. Si el reset se ha efectuado correctamente, el chip DSP colocará el valor hexadecimal AA en el Read Data Register (para poder leer ese registro, como de costumbre, hay que esperar a que el bit 7 del Read Buffer Status Register se active). Finalmente, desactiva el flag DmaBusy, indicando que el DMA está libre, y activa el flag DmaDone, indicando que no hay operación de DMA en curso.

3.12. dsp_dma_setup, dsp_setup

En realidad, a pesar de sus nombres, estas funciones no inicializan nada, si no que programan el comienzo de una transferencia de lectura/escritura.

La función dsp_dma_setup asigna en el DMA los parámetros de la transferencia. En primer lugar deshabilita las interrupciones y el propio canal DMA. Luego toma la dirección del buffer del DMA, y obtiene de ella la página de 64K donde se halla el buffer y su desplazamiento y los envía al DMA. Envía también al DMA el número de bytes a leer o escribir. Finalmente rehabilita el canal DMA y las interrupciones.

La función dsp_setup asigna los parámetros del muestreo/reproducción del chip DMA. En primer lugar, asigna el modo de operación (lectura/escritura), enviando el comando mediante función dsp_command, y enciende o apaga la salida de altavoz de la tarjeta según estemos en modo escritura o

lectura, respectivamente. Luego invoca dsp_command para colocar el modo estéreo/mono y asignar el número de bytes de la transferencia.

3.13. dsp_set_stereo, dsp_set_speed, dsp_set_bits, dsp_set_size, dsp_set_sign

Estas funciones son invocadas por dsp_ioctl para cambiar los parámetros del driver DSP. Algunas de estas funciones no utilizan los puertos en absoluto, si no que actualizan los valores de las variables del driver del DSP.

dsp_set_speed()	selecciona velocidad de muestreo, inhabilitando las interrupciones e invocando a la función dsp_command;
dsp_set_stereo()	habilita/inhabilita el modo estéreo, asignando el valor de la variable DspStereo;
dsp_set_bits()	selecciona el número de bits (la resolución del muestreo) asignando la variable DspBits;
dsp_set_size()	asigna el tamaño de la transmisión asignando el valor de la variable DspFragmentSize;
dsp_set_sign()	indica el signo de las muestras, asignando el valor de la variable DspSign.

3.14. Variables del driver DSP.

DspTasknr	número de la tarea dsp_task, si estamos en modo protegido (asignada por la función dsp_task);
DspVersion	versión de la tarjeta de audio (asignada por la función dsp_init);
DspStereo	modo estéreo/mono (asignada por dsp_set_stereo);
DspSpeed	velocidad de muestreo (asignada por dsp_set_speed);
DspBits	número de bits de las muestras (asignada por dsp_set_bits);
DspSign	flag que indica si las muestras tienen signo (asignada por

	dsp_set_sign);
DspFragmentSize	tamaño de la transmisión (asignada por dsp_open y dsp_set_size)
DspAvail	1 si existe tarjeta de audio en el PC (asignada por dsp_init)
DspBusy	1 si el chip DSP está siendo utilizado (asignada por dsp_open, dsp_close);
DmaBusy	1 si el chip DMA está siendo utilizado (asignada por dsp_open, dsp_close, dsp_handler, dsp_reset, dsp_read, dsp_write);
DmaDone	indica si el DMA ha terminado una transferencia (asignada por dsp_reset, dsp_write, dsp_read);
DmaMode	indica el modo DMA, DEV_WRITE/DEV_READ (asignada por dsp_write, dsp_read);
DmaBuffer	el buffer del DMA;
DmaPtr	dirección virtual del buffer de DMA, que puede ser distinta de DmaBuffer si el buffer reservado cruza la frontera del segmento (asignada por dsp_init);
DmaPhys	dirección física del buffer de DMA (asignada por dsp_init).

4. Apéndice: listados

4.1. sb16.h

```
#ifndef SB16_H
#define SB16_H

#define SB_DEBUG        0     /* 1 = print debug info */
#define SB_DEBUG_2            0     /* 1 = print more debug info */

#define SB_TIMEOUT            32000   /* timeout count */

/* IRQ, base address and DMA channels */
#define SB_IRQ          5
#define SB_BASE_ADDR    0x220       /* 0x210, 0x220, 0x230, 0x240,
                                     * 0x250, 0x260, 0x280
                                     */
#define SB_DMA_8  1           /* 0, 1, 3 */
```

```
#define SB_DMA_16 5           /* 5, 6, 7 */
#if _WORD_SIZE == 2
#define DMA_SIZE  8192        /* Dma buffer MUST BE MULTIPLE OF 2 */
#else
#define DMA_SIZE  32768       /* Dma buffer MUST BE MULTIPLE OF 2 */
#endif

/* Some defaults for the DSP */
#define DEFAULT_SPEED         22050      /* Sample rate */
#define DEFAULT_BITS          8        /* Nr. of bits */
#define DEFAULT_SIGN          0        /* 0 = unsigned, 1 = signed */
#define DEFAULT_STEREO        0        /* 0 = mono, 1 = stereo */

/* DMA port addresses */
#define DMA8_ADDR ((SB_DMA_8 & 3) << 1) + 0x00
#define DMA8_COUNT      ((SB_DMA_8 & 3) << 1) + 0x01
#define DMA8_MASK 0x0A
#define DMA8_MODE 0x0B
#define DMA8_CLEAR      0x0C


/* If after this preprocessing stuff DMA8_PAGE is not defined
 * the 8-bit DMA channel specified is not valid
 */
#if SB_DMA_8 == 0
#  define DMA8_PAGE     0x87
#else
#  if SB_DMA_8 == 1
#    define DMA8_PAGE   0x83
#  else
#    if SB_DMA_8 == 3
#      define DMA8_PAGE 0x82
#    endif
#  endif
#endif


#define DMA16_ADDR      ((SB_DMA_16 & 3) << 2) + 0xC0
#define DMA16_COUNT     ((SB_DMA_16 & 3) << 2) + 0xC2
#define DMA16_MASK      0xD4
#define DMA16_MODE      0xD6
#define DMA16_CLEAR     0xD8


/* If after this preprocessing stuff DMA16_PAGE is not defined
 * the 16-bit DMA channel specified is not valid
 */
#if SB_DMA_16 == 5
#  define DMA16_PAGE      0x8B
#else
#  if SB_DMA_16 == 6
#    define DMA16_PAGE    0x89
#  else
#    if SB_DMA_16 == 7
#      define DMA16_PAGE  0x8A
#    endif
#  endif
#endif
```

```
/* DMA modes */
#define DMA16_AUTO_PLAY       0x58 + (SB_DMA_16 & 3)
#define DMA16_AUTO_REC        0x54 + (SB_DMA_16 & 3)
#define DMA8_AUTO_PLAY        0x58 + SB_DMA_8
#define DMA8_AUTO_REC         0x54 + SB_DMA_8


/* IO ports for soundblaster */
#define DSP_RESET 0x6 + SB_BASE_ADDR
#define DSP_READ  0xA + SB_BASE_ADDR
#define DSP_WRITE 0xC + SB_BASE_ADDR
#define DSP_COMMAND     0xC + SB_BASE_ADDR
#define DSP_STATUS      0xC + SB_BASE_ADDR
#define DSP_DATA_AVL    0xE + SB_BASE_ADDR
#define DSP_DATA16_AVL  0xF + SB_BASE_ADDR
#define MIXER_REG 0x4 + SB_BASE_ADDR
#define MIXER_DATA      0x5 + SB_BASE_ADDR
#define OPL3_LEFT 0x0 + SB_BASE_ADDR
#define OPL3_RIGHT      0x2 + SB_BASE_ADDR
#define OPL3_BOTH 0x8 + SB_BASE_ADDR


/* DSP Commands */
#define DSP_INPUT_RATE        0x42  /* set input sample rate */
#define DSP_OUTPUT_RATE       0x41  /* set output sample rate */
#define DSP_CMD_SPKON         0xD1  /* set speaker on */
#define DSP_CMD_SPKOFF        0xD3  /* set speaker off */
#define DSP_CMD_DMA8HALT      0xD0  /* halt DMA 8-bit operation */
#define DSP_CMD_DMA8CONT      0xD4  /* continue DMA 8-bit operation */
#define DSP_CMD_DMA16HALT     0xD5  /* halt DMA 16-bit operation */
#define DSP_CMD_DMA16CONT     0xD6  /* continue DMA 16-bit operation */
#define DSP_GET_VERSION       0xE1  /* get version number of DSP */
#define DSP_CMD_8BITAUTO_IN   0xCE  /* 8 bit auto-initialized input */
#define DSP_CMD_8BITAUTO_OUT  0xC6  /* 8 bit auto-initialized output */
#define DSP_CMD_16BITAUTO_IN  0xBE  /* 16 bit auto-initialized input */
#define DSP_CMD_16BITAUTO_OUT 0xB6  /* 16 bit auto-initialized output */
#define DSP_CMD_IRQREQ8       0xF2  /* Interrupt request 8 bit        */
#define DSP_CMD_IRQREQ16      0xF3  /* Interrupt request 16 bit        */


/* DSP Modes */
#define DSP_MODE_MONO_US      0x00  /* Mono unsigned */
#define DSP_MODE_MONO_S       0x10  /* Mono signed */
#define DSP_MODE_STEREO_US    0x20  /* Stereo unsigned */
#define DSP_MODE_STEREO_S     0x30  /* Stereo signed */


/* MIXER commands */
#define MIXER_RESET           0x00  /* Reset */
#define MIXER_DAC_LEVEL       0x04  /* Used for detection only */
#define MIXER_MASTER_LEFT     0x30  /* Master volume left */
#define MIXER_MASTER_RIGHT    0x31  /* Master volume right */
#define MIXER_DAC_LEFT        0x32  /* Dac level left */
#define MIXER_DAC_RIGHT       0x33  /* Dac level right */
#define MIXER_FM_LEFT         0x34  /* Fm level left */
#define MIXER_FM_RIGHT        0x35  /* Fm level right */
```

```
#define MIXER_CD_LEFT         0x36  /* Cd audio level left */
#define MIXER_CD_RIGHT        0x37  /* Cd audio level right */
#define MIXER_LINE_LEFT       0x38  /* Line in level left */
#define MIXER_LINE_RIGHT      0x39  /* Line in level right */
#define MIXER_MIC_LEVEL       0x3A  /* Microphone level */
#define MIXER_PC_LEVEL        0x3B  /* Pc speaker level */
#define MIXER_OUTPUT_CTRL     0x3C  /* Output control */
#define MIXER_IN_LEFT         0x3D  /* Input control left */
#define MIXER_IN_RIGHT        0x3E  /* Input control right */
#define MIXER_GAIN_IN_LEFT    0x3F  /* Input gain control left */
#define MIXER_GAIN_IN_RIGHT   0x40  /* Input gain control right */
#define MIXER_GAIN_OUT_LEFT   0x41  /* Output gain control left */
#define MIXER_GAIN_OUT_RIGHT  0x42  /* Output gain control rigth */
#define MIXER_AGC       0x43  /* Automatic gain control */
#define MIXER_TREBLE_LEFT     0x44  /* Treble left */
#define MIXER_TREBLE_RIGHT    0x45  /* Treble right */
#define MIXER_BASS_LEFT       0x46  /* Bass left */
#define MIXER_BASS_RIGHT      0x47  /* Bass right */
#define MIXER_SET_IRQ         0x80  /* Set irq number */
#define MIXER_SET_DMA         0x81  /* Set DMA channels */
#define MIXER_IRQ_STATUS      0x82  /* Irq status */

/* Mixer constants */
#define MIC             0x01  /* Microphone */
#define CD_RIGHT        0x02
#define CD_LEFT               0x04
#define LINE_RIGHT            0x08
#define LINE_LEFT       0x10
#define FM_RIGHT        0x20
#define FM_LEFT               0x40

/* DSP constants */
#define DSP_MAX_SPEED         44100      /* Max sample speed in KHz */
#define DSP_MIN_SPEED         4000       /* Min sample speed in KHz */
#define DSP_MAX_FRAGMENT_SIZE DMA_SIZE   /* Maximum fragment size */
#define DSP_MIN_FRAGMENT_SIZE 1024     /* Minimum fragment size */


/* Number of bytes you can DMA before hitting a 64K boundary: */
#define dma_bytes_left(phys)    \
   ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))


/* Function prototypes used by mixer and dsp */
_PROTOTYPE(int mixer_set, (int reg, int data));

#endif /* SB16_H */
```

4.2. sb16_dsp.c

```
/* This file contains the driver for a DSP (Digital Sound Processor) on
 * a SoundBlaster 16 (ASP) soundcard.
 *
 * The driver supports the following operations (using message format m2):
 *
 *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
```

```
 * ------------------------------------------------------------------
 * |  DEV_OPEN  | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_CLOSE | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_READ  | device  | proc nr |  bytes  |         | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_WRITE | device  | proc nr |  bytes  |         | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
 * ------------------------------------------------------------------
 *
 * The file contains one entry point:
 *
 *   dsp_task:    main entry when system is brought up
 *
 *  May 20 1995               Author: Michel R. Prevenier
 */


#include "kernel.h"
#include <minix/com.h>
#include <minix/callnr.h>
#include <sys/ioctl.h>
#if __minix_vmd
#include "proc.h"
#include "config.h"
#endif
#include "sb16.h"

#if ENABLE_SB_AUDIO

/* prototypes */
FORWARD _PROTOTYPE( void init_buffer, (void));
FORWARD _PROTOTYPE( int dsp_init, (void));
FORWARD _PROTOTYPE( int dsp_handler, (int irq));
FORWARD _PROTOTYPE( int dsp_open, (message *m_ptr));
FORWARD _PROTOTYPE( int dsp_close, (message *m_ptr));
FORWARD _PROTOTYPE( int dsp_ioctl, (message *m_ptr));
FORWARD _PROTOTYPE( int dsp_write, (message *m_ptr));
FORWARD _PROTOTYPE( int dsp_read, (message *m_ptr));
FORWARD _PROTOTYPE( int dsp_reset, (void));
FORWARD _PROTOTYPE( int dsp_command, (int value));
FORWARD _PROTOTYPE( int dsp_set_speed, (unsigned int speed));
FORWARD _PROTOTYPE( int dsp_set_size, (unsigned int size));
FORWARD _PROTOTYPE( int dsp_set_stereo, (unsigned int stereo));
FORWARD _PROTOTYPE( int dsp_set_bits, (unsigned int bits));
FORWARD _PROTOTYPE( int dsp_set_sign, (unsigned int sign));
FORWARD _PROTOTYPE( void dsp_dma_setup, (phys_bytes address, int count));
FORWARD _PROTOTYPE( void dsp_setup, (void));

/* globals */
#if __minix_vmd
PRIVATE int DspTasknr = ANY;
#endif
PRIVATE int DspVersion[2];
PRIVATE unsigned int DspStereo = DEFAULT_STEREO;
PRIVATE unsigned int DspSpeed = DEFAULT_SPEED;
```

```
PRIVATE unsigned int DspBits = DEFAULT_BITS;
PRIVATE unsigned int DspSign = DEFAULT_SIGN;
PRIVATE unsigned int DspFragmentSize = DSP_MAX_FRAGMENT_SIZE;
PRIVATE int DspAvail = 0;
PRIVATE int DspBusy = 0;
PRIVATE int DmaBusy = 0;
PRIVATE int DmaDone = 1;
PRIVATE int DmaMode = 0;

PRIVATE char DmaBuffer[(long)2 * DMA_SIZE];
PRIVATE char *DmaPtr;
PRIVATE phys_bytes DmaPhys;


/*===========================================================================*
 *                      dsp_task                                 *
 *===========================================================================*/
PUBLIC void dsp_task()
{
  message mess;
  int err, caller, proc_nr;

#if __minix_vmd
  DspTasknr = proc_number(proc_ptr);
#endif

  /* initialize the DMA buffer */
  init_buffer();

  /* Here is the main loop of the sound task. It waits for a message, carries
   * it out, and sends a reply.
   */
  while (TRUE)
  {
      receive(ANY, &mess);

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

      switch (caller)
        {
        case HARDWARE:
            /* Leftover interrupt. */
            continue;
        case FS_PROC_NR:
            /* The only legitimate caller. */
            break;
        default:
            printf("sb16: got message from %d\n", caller);
            continue;
      }

      /* Now carry out the work. */
      switch(mess.m_type)
        {
          case DEV_OPEN:      err = dsp_open(&mess);break;
          case DEV_CLOSE:     err = dsp_close(&mess);break;
          case DEV_IOCTL:     err = dsp_ioctl(&mess);break;
```

```
          case DEV_READ:      err = dsp_read(&mess);break;
          case DEV_WRITE:     err = dsp_write(&mess);break;
          default:            err = EINVAL;break;
      }

      /* Finally, prepare and send the reply message. */
      mess.m_type = TASK_REPLY;
      mess.REP_PROC_NR = proc_nr;

      mess.REP_STATUS = err;  /* #bytes transfered or error code */
      send(caller, &mess);    /* send reply to caller */
  }
}


/*============================================================================*
 *                      init_buffer                                           *
 *============================================================================*/
PRIVATE void init_buffer()
{
/* Select a buffer that can safely be used for dma transfers.
 * Its absolute address is 'DmaPhys', the normal address is 'DmaPtr'.
 */

  DmaPtr = DmaBuffer;
  DmaPhys = vir2phys(DmaBuffer);

  if (dma_bytes_left(DmaPhys) < DMA_SIZE) {
      /* First half of buffer crosses a 64K boundary, can't DMA into that */
      DmaPtr += DMA_SIZE;
      DmaPhys += DMA_SIZE;
  }
}


/*============================================================================*
 *                      dsp_open                                              *
 *============================================================================*/
PRIVATE int dsp_open(m_ptr)
message *m_ptr;
{

#if SB_DEBUG
  printf("sb16_open\n");
#endif

  /* try to detect SoundBlaster card */
  if (!DspAvail && dsp_init() != OK) return EIO;

  /* Only one open at a time with soundcards */
  if (DspBusy) return EBUSY;

  /* Start with a clean DSP */
  if (dsp_reset() != OK) return EIO;

  /* Setup default values */
  DspStereo = DEFAULT_STEREO;
  DspSpeed = DEFAULT_SPEED;
```

```
  DspBits = DEFAULT_BITS;
  DspSign = DEFAULT_SIGN;
  DspFragmentSize = DMA_SIZE;

  DspBusy = 1;
  DmaBusy = 0;

  return OK;
}


/*============================================================================*
 *                        dsp_close                                           *
 *============================================================================*/
PRIVATE int dsp_close(m_ptr)
message *m_ptr;
{

#if SB_DEBUG
  printf("dsp_close\n");
#endif

  DspBusy = 0;                  /* soundcard available again */
  DmaBusy = 0;

  return OK;
}


/*============================================================================*
 *                        dsp_ioctl                                           *
 *============================================================================*/
PRIVATE int dsp_ioctl(m_ptr)
message *m_ptr;
{
  int status;
  phys_bytes user_phys;
  unsigned int val;

  /* Cannot change parameters during play or recording */
  if (DmaBusy) return EBUSY;

  /* Get user data */
  if (m_ptr->REQUEST != DSPIORESET)
  {
    user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
                                                    sizeof(unsigned int));
    if (user_phys == 0) return(EFAULT);
    phys_copy(user_phys, vir2phys(&val), (phys_bytes) sizeof(val));
  }

#if SB_DEBUG
  printf("dsp_ioctl: got ioctl %d, argument: %d\n", m_ptr->REQUEST, val);
#endif

  switch(m_ptr->REQUEST)
  {
    case DSPIORATE:     status = dsp_set_speed(val);break;
```

```
    case DSPIOSTEREO:   status = dsp_set_stereo(val);break;
    case DSPIOBITS:     status = dsp_set_bits(val);break;
    case DSPIOSIZE:     status = dsp_set_size(val);break;
    case DSPIOSIGN:     status = dsp_set_sign(val);break;
    case DSPIOMAX:      {
                           val = DMA_SIZE;
                           phys_copy(vir2phys(&val), user_phys,
                                          (phys_bytes) sizeof(val));
                           status = OK;
                        };break;
    case DSPIORESET:    status = dsp_reset();break;
    default:            status = ENOTTY;break;
  }

  return status;
}


/*==========================================================================*
 *                      dsp_init                                *
 *==========================================================================*/
PRIVATE int dsp_init()
{
  int i;

  if (dsp_reset () != OK)
  {
    printf("sb16: No SoundBlaster card detected\n");
    return -1;
  }

  DspVersion[0] = DspVersion[1] = 0;
  dsp_command(DSP_GET_VERSION);     /* Get DSP version bytes */

  for (i = 1000; i; i--)
  {
    if (in_byte (DSP_DATA_AVL) & 0x80)
    {
      if (DspVersion[0] == 0)
        DspVersion[0] = in_byte (DSP_READ);
      else
      {
        DspVersion[1] = in_byte (DSP_READ);
        break;
      }
    }
  }

  if (DspVersion[0] < 4)
  {
    printf("sb16: No SoundBlaster 16 compatible card detected\n");
    return -1;
  }
  else
    printf ("sb16: SoundBlaster DSP version %d.%d detected\n",
          DspVersion[0], DspVersion[1]);

  /* set IRQ and DMA channels */
```

```
  mixer_set(MIXER_SET_IRQ, (1 << (SB_IRQ / 2 - 1)));
  mixer_set(MIXER_SET_DMA, (1 << SB_DMA_8 | 1 << SB_DMA_16));

  /* register interrupt vector and enable irq */
  put_irq_handler(SB_IRQ, dsp_handler);
  enable_irq(SB_IRQ);

  DspAvail = 1;
  return OK;
}


/*===========================================================================*
 *                      dsp_handler                          *
 *===========================================================================*/
PRIVATE int dsp_handler(irq)
int irq;
{

#if SB_DEBUG2
  printf("SoundBlaster interrupt %d\n", irq);
#endif

  if (DmaDone)     /* Dma transfer is done */
  {
    /* Send DSP command to stop dma */
    dsp_command((DspBits == 8 ? DSP_CMD_DMA8HALT : DSP_CMD_DMA16HALT));

    DmaBusy = 0;  /* Dma available again */
  }

  /* Send interrupt to audio task and enable again */
#if __minix_vmd
  interrupt(DspTasknr);
#else
  interrupt(AUDIO);
#endif

  /* Acknowledge the interrupt on the DSP */

  (void) in_byte((DspBits == 8 ? DSP_DATA_AVL : DSP_DATA16_AVL));

  return 1;
}


/*===========================================================================*
 *                      dsp_command                          *
 *===========================================================================*/
PRIVATE int dsp_command(value)
int value;
{
  int             i;

  for (i = 0; i < SB_TIMEOUT; i++)
  {
    if ((in_byte (DSP_STATUS) & 0x80) == 0)
    {
```

```
      out_byte (DSP_COMMAND, value);
      return OK;
    }
  }

  printf ("sb16: SoundBlaster: DSP Command(%x) timeout\n", value);
  return -1;
}


/*=========================================================================*
 *                       dsp_reset                         *
 *=========================================================================*/
PRIVATE int dsp_reset(void)
{
  int i;

  out_byte (DSP_RESET, 1);
  for(i =0; i<1000; i++); /* wait a while */
  out_byte (DSP_RESET, 0);

  for (i = 0; i < 1000 && !(in_byte (DSP_DATA_AVL) & 0x80); i++);

  if (in_byte (DSP_READ) != 0xAA) return EIO; /* No SoundBlaster */


  DmaBusy = 0;
  DmaDone = 1;

  return OK;
}


/*=========================================================================*
 *                       dsp_set_speed                             *
 *=========================================================================*/
static int dsp_set_speed(speed)
unsigned int speed;
{
#if SB_DEBUG
  printf("sb16: setting speed to %u, stereo = %d\n", speed, DspStereo);
#endif

  if (speed < DSP_MIN_SPEED || speed > DSP_MAX_SPEED)
    return EPERM;

  /* Soundblaster 16 can be programmed with real sample rates
   * instead of time constants
   *
   * Since you cannot sample and play at the same time
   * we set in- and output rate to the same value
   */

   lock();                              /* disable interrupts */
   dsp_command(DSP_INPUT_RATE);               /* set input rate */
   dsp_command(speed >> 8);             /* high byte of speed */
   dsp_command(speed);                  /* low byte of speed */
   dsp_command(DSP_OUTPUT_RATE);        /* same for output rate */
```

```
   dsp_command(speed >> 8);
   dsp_command(speed);
   unlock();                              /* enable interrupts */

   DspSpeed = speed;

   return OK;
}


/*=========================================================================*
 *                      dsp_set_stereo                                     *
 *=========================================================================*/
static int dsp_set_stereo(stereo)
unsigned int stereo;
{
  if (stereo)
    DspStereo = 1;
  else
    DspStereo = 0;

  return OK;
}


/*=========================================================================*
 *                      dsp_set_bits                                       *
 *=========================================================================*/
static int dsp_set_bits(bits)
unsigned int bits;
{
  /* Sanity checks */
  if (bits != 8 && bits != 16) return EINVAL;

  DspBits = bits;

  return OK;
}


/*=========================================================================*
 *                      dsp_set_size                                       *
 *=========================================================================*/
static int dsp_set_size(size)
unsigned int size;
{

#if SB_DEBUG
  printf("sb16: set fragment size to %u\n", size);
#endif

  /* Sanity checks */
  if (size < DSP_MIN_FRAGMENT_SIZE ||
      size > DSP_MAX_FRAGMENT_SIZE ||
      size % 2 != 0)
    return EINVAL;

  DspFragmentSize = size;
```

```
  return OK;
}


/*==========================================================================*
 *                       dsp_set_sign                                       *
 *==========================================================================*/
static int dsp_set_sign(sign)
unsigned int sign;
{

#if SB_DEBUG
  printf("sb16: set sign to %u\n", sign);
#endif

  DspSign = (sign > 0 ? 1 : 0);

  return OK;
}


/*==========================================================================*
 *                       dsp_dma_setup                                      *
 *==========================================================================*/
PRIVATE void dsp_dma_setup(address, count)
phys_bytes address;
int count;

{
#if SB_DEBUG
  printf("Setting up %d bit DMA\n", DspBits);
#endif

  if (DspBits == 8)    /* 8 bit sound */
  {
    count--;

    lock();
    out_byte(DMA8_MASK, SB_DMA_8 | 0x04);      /* Disable DMA channel */
    out_byte(DMA8_CLEAR, 0x00);                  /* Clear flip flop */

    /* set DMA mode */
    out_byte(DMA8_MODE,
               (DmaMode == DEV_WRITE ? DMA8_AUTO_PLAY : DMA8_AUTO_REC));

    out_byte(DMA8_ADDR, address >>  0);        /* Low_byte of address */
    out_byte(DMA8_ADDR, address >>  8);        /* High byte of address */
    out_byte(DMA8_PAGE, address >> 16);        /* 64K page number */
    out_byte(DMA8_COUNT, count >> 0);          /* Low byte of count */
    out_byte(DMA8_COUNT, count >> 8);          /* High byte of count */
    out_byte(DMA8_MASK, SB_DMA_8);         /* Enable DMA channel */
    unlock();
  }
  else  /* 16 bit sound */
  {
    count-= 2;
```

```
    lock();
    out_byte(DMA16_MASK, (SB_DMA_16 & 3) | 0x04); /* Disable DMA channel */
    out_byte(DMA16_CLEAR, 0x00);                  /* Clear flip flop */

    /* Set dma mode */
    out_byte(DMA16_MODE,
               (DmaMode == DEV_WRITE ? DMA16_AUTO_PLAY : DMA16_AUTO_REC));

    out_byte(DMA16_ADDR, (address >> 1) & 0xFF);  /* Low_byte of address */
    out_byte(DMA16_ADDR, (address >> 9) & 0xFF);  /* High byte of address */
    out_byte(DMA16_PAGE, (address >> 16) & 0xFE); /* 128K page number */
    out_byte(DMA16_COUNT, count >> 1);            /* Low byte of count */
    out_byte(DMA16_COUNT, count >> 9);            /* High byte of count */
    out_byte(DMA16_MASK, SB_DMA_16 & 3);          /* Enable DMA channel */
    unlock();
  }
}


/*=========================================================================*
 *                       dsp_setup                                         *
 *=========================================================================*/
PRIVATE void dsp_setup()
{
  /* Set current sample speed */
  dsp_set_speed(DspSpeed);

  /* Put the speaker on */
   if (DmaMode == DEV_WRITE)
   {
     dsp_command (DSP_CMD_SPKON); /* put speaker on */

      /* Program DSP with dma mode */
      dsp_command((DspBits == 8 ? DSP_CMD_8BITAUTO_OUT : DSP_CMD_16BITAUTO_OUT));
   }
   else
   {
     dsp_command (DSP_CMD_SPKOFF); /* put speaker off */

     /* Program DSP with dma mode */
     dsp_command((DspBits == 8 ? DSP_CMD_8BITAUTO_IN : DSP_CMD_16BITAUTO_IN));
   }

  /* Program DSP with transfer mode */
  if (!DspSign)
    dsp_command((DspStereo == 1 ? DSP_MODE_STEREO_US : DSP_MODE_MONO_US));
  else
    dsp_command((DspStereo == 1 ? DSP_MODE_STEREO_S : DSP_MODE_MONO_S));

  /* Give length of fragment to DSP */
  if (DspBits == 8) /* 8 bit transfer */
  {
    /* #bytes - 1 */
    dsp_command((DspFragmentSize - 1) >> 0);
    dsp_command((DspFragmentSize - 1) >> 8);
  }
  else              /* 16 bit transfer */
  {
```

```
    /* #words - 1 */
    dsp_command((DspFragmentSize - 1) >> 1);
    dsp_command((DspFragmentSize - 1) >> 9);
  }
}


/*==========================================================================*
 *                       dsp_write                                          *
 *==========================================================================*/
PRIVATE int dsp_write(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  message mess;

  if (m_ptr->COUNT != DspFragmentSize) return EINVAL;

  /* From this user address */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes)m_ptr->ADDRESS, DspFragmentSize);
  if (user_phys == 0) return EINVAL;

  if (DmaBusy)    /* Dma already started */
  {
    if (DmaMode != m_ptr->m_type) return EBUSY;

    DmaDone = 0;  /* No, we're not done yet */

    /* Wait for next block to become free */
    receive(HARDWARE, &mess);

    /* Copy first block to dma buffer */
    phys_copy(user_phys, DmaPhys, (phys_bytes) DspFragmentSize);

  }
  else /* A new dma transfer has started */
  {
    DmaMode = DEV_WRITE;           /* Dma mode is writing */

    /* Copy fragment to dma buffer */
    phys_copy(user_phys, DmaPhys, (phys_bytes) DspFragmentSize);

    /* Set up the dma chip */
    dsp_dma_setup(DmaPhys, DspFragmentSize);

    /* Set up the DSP */
    dsp_setup();

    DmaBusy = 1;         /* Dma is busy */
  }

  DmaDone = 1;            /* dma done for now */

  return(DspFragmentSize);
}


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

```
 *                      dsp_read                                 *
 *===============================================================================*/
PRIVATE int dsp_read(m_ptr)
message *m_ptr;
{
  phys_bytes user_phys;
  message mess;

  if (m_ptr->COUNT != DspFragmentSize) return EINVAL;

  /* To this user address */
  user_phys = numap(m_ptr->PROC_NR, (vir_bytes)m_ptr->ADDRESS, DspFragmentSize);
  if (user_phys == 0) return EINVAL;

  if (DmaBusy)     /* Dma already started */
  {
    if (DmaMode != m_ptr->m_type) return EBUSY;

    DmaDone = 0;   /* No, we're not done yet */

    /* Wait for a full dma buffer */
    receive(HARDWARE, &mess);

    /* Copy the buffer */
    phys_copy(DmaPhys, user_phys, (phys_bytes) DspFragmentSize);
  }
  else  /* A new dma transfer has started */
  {
    DmaMode = DEV_READ;                /* Dma mode is reading */

    /* Set up the dma chip */
    dsp_dma_setup(DmaPhys, DspFragmentSize);

    /* Set up the DSP */
    dsp_setup();

    DmaBusy = 1;       /* Dma has started */
    DmaDone = 0;       /* Dma not done */

    /* Wait for dma to finish with first block */
    receive(HARDWARE, &mess);

    /* Copy dma buffer to user */
    phys_copy(DmaPhys, user_phys, (phys_bytes) DspFragmentSize);
  }

  DmaDone = 1;   /* dma done for now */

  return(DspFragmentSize);
}
#endif /* ENABLE_AUDIO */
```

5. Cuestiones

5.1. ¿Cuál es el tratamiento que recibe una llamada al sistema en Minix 2.0?

5.2. ¿Cuál es la rutina básica que sigue una tarea de E/S en Minix 2.0?

5.3. ¿Para qué se utiliza el DMA en la grabación/reproducción de sonido?

5.4. ¿Por qué es necesario pasarle a la función dsp_dma_setup la dirección física del buffer de DMA y no simplemente su dirección virtual?

5.5. ¿Cómo sabe la tarea dsp_task que ha finalizado una transferencia de datos?